POLITECNICO DI MILANO
SCUOLA INTERPOLITECNICA DI DOTTORATO
Research Doctorate Course in Information Engineering – XVIII Cycle

Final Dissertation

A SOURCE-LEVEL ESTIMATION AND OPTIMIZATION METHODOLOGY FOR EXECUTION TIME AND ENERGY CONSUMPTION OF EMBEDDED SOFTWARE

Daniele Paolo Scarpazza

Tutor	Coordinator of the Research Doctorate Course
Prof. William Fornaciari	Prof. Stefano Crespi Reghizzi

May 2006

This page intentionally left blank.

POLITECNICO DI MILANO
Dipartimento di Elettronica e Informazione
DOTTORATO DI RICERCA IN INGEGNERIA DELL'INFORMAZIONE

A SOURCE-LEVEL ESTIMATION AND OPTIMIZATION METHODOLOGY FOR EXECUTION TIME AND ENERGY CONSUMPTION OF EMBEDDED SOFTWARE

Doctoral Dissertation of:
Daniele Paolo Scarpazza

Advisor:
Prof. William Fornaciari

Tutor:
Prof. Fabrizio Ferrandi

Supervisor of the Doctoral Program:
Prof. Stefano Crespi Reghizzi

2006 - XVIII

This document was sent to the publisher on April 4th, 2006.

The author can be reached for comments, questions and suggestions at the following e-mail address: scarpaz@scarpaz.com.

Short Table of Contents

Detailed Table of Contents

List of Figures

List of Tables

Abstract

The contributions of this thesis are in the two main areas of the estimation and optimization of the software for embedded systems.

Estimation

Embedded system designers frequently face the problem of estimating how much energy and time are spent in significant clusters of operations in their software, such as loop bodies or their subportions. This is crucial to determine how fast their software runs on a platform, how much energy it consumes, where optimizations are needed, or what hardware it requires to ensure a given speed.

This problem is not effectively solved by current approaches: instruction-level simulation, static timing analysis and source-level instrumentation. Instruction level simulation provides too low-level information and is too slow; static timing analysis cannot deal with dynamism, while current source level instrumentation tools only provide global or function level estimates.

I propose a novel methodology which is able to provide time and energy estimates for any flexibly defined cluster of operations in the program. With it, designers do not need to define their interested clusters in advance, and not even to re-run the tool for different cluster choices. With this information, designers can better understand which portions of the source code cause the major consumptions, and apply optimizations there.

The tool which implements the methodology exhibits simulation times 10,000 shorter than a reference ISS, and a good estimation accuracy.

Optimization

Optimizing the software of embedded systems is of primary importance, and source-level transformations have been showed to offer the highest gains. However, exploring the source-level transformations is currently a slow task. In fact, the current approaches repeat many times an exploration loop which

involves slow steps such as instruction-level simulation. Additionally, the selection of the portions to transform is done manually, which is impractical for large projects.

Thanks to the availability of a source level estimation technique like the one presented in this thesis, it is now possible to develop techniques for the automatic selection and targeting of source-level transformations. These techniques allow a shorter and quicker exploration loop. I propose an example of such techniques, which is well suited for transformations which have a predominant local effect and show small mutual interactions. It employs a network of fuzzy rules for the steering task. It is designed to scale well on large projects, and it is modular, so new transformations can be added easily.

The application of this technique over a set of realistic benchmarks obtained significant energy and execution time savings (approx. 15%). The technique is fast enough to allow the optimization exploration of large-scale applications.

Chapter 1

Overview

THIS chapter provides an overview of this thesis, including its goal, its motivation, its fundamental approach and the benefits it offers. It first motivates why embedded system designers need a new generation of estimation and optimization techniques, on the basis of the current context. From this context I will derive the requirements that such techniques must meet. Finally, I will show how these requirements derive precise research choices, leading to a specific technique, which I will adopt.

1.1 Designers need a new generation of software estimation techniques

Modern embedded systems comprise larger and larger software components. When designing and engineering these products, designers need as soon as possible estimates of the energy and processing power requirements for the software they write. This information is crucial for many purposes: to determine whether the product is feasible, to compare different implementations, to optimize the code. One of the crucial consumption areas they are interested in is the processor core. This thesis deals with this very problem:

> Estimating the execution time and the energy consumption caused by the processor core when executing software on a given architecture, exposing the contribution of each individual element of the source code.

There are two reasons why I restrict my attention to the processor core rather than trying to achieve the same estimates on an entire system: the first reason is that the broader problem is so large that it cannot be effectively attacked in a single thesis; the second reason is that for some portions of the system (e.g. the data memory hierarchy) the state of the art already presents solutions

which satisfy the designers' needs effectively. I will motivate this claim in Chapter 2.

As far as the core is concerned, a new generation of estimation tools is needed. In fact, the characteristics of contemporary applications are making the current tool impractical to use. Applications are becoming larger, more complex and more dynamic, and the software estimation techniques are not keeping the pace. I will provide details for all of these claims in the next sections.

With respect to the problem of estimating the software energy consumption and execution time for the processor cores, designers need tools which satisfy the following requirements:

1. source level;
2. fine detail;
3. dynamic;
4. fast.

The following paragraphs clarify what I mean by each of the above requirements. Chapter 2 will show that none of the current approaches satisfy all the above requirements at the same time. Throughout all this thesis, the choices I will make while designing my methodology will be guided and constrained by the above requirements.

Source-level analysis: developers write code mostly in high level languages, rather than in assembly languages. Tools must be able to provide estimates at the same level of abstraction: the source code entities. Tools should be able to indicate how much energy consumption was caused by elements which appear in the source code. Techniques which cannot relate time and energy estimates to source-level entities are of little practical usability.

Fine detail: the "hot spots" of programs (i.e. the portions which cause the most energy and time consumption) are typically kernels of operations inside loop bodies. Tools must be able to provide estimates at a degree of detail which allow to resolve the single operation or cluster of operations. Rougher detail levels (i.e. the function) are insufficient. Techniques which cannot resolve this level of detail are insufficient for the contemporary and future needs.

Dynamic: modern applications are becoming more and more dynamic in nature. The behavior of multimedia encoders and decoders depend more and more on the contents of the streams they process, and applications from many other domains like wireless and gaming show the same trends. The variability in the behavior of algorithms is high and increasing, and it makes static estimation techniques unfit for the purpose. Estimation tools must be able to keep the actual input data into account.

Fast: the size and complexity of modern embedded applications is increasing significantly. Nowadays the simulation of an application of non-trivial complexity at the circuit or gate-level is unaffordable. Instruction-level simulation of contemporary video encoders may take days, even for

short sequences. In the future, simulation at the instruction level will become not affordable anymore[1]. Whichever estimation technique requires cycle accuracy or close-to-cycle accuracy is doomed to obsolescence quite soon. Estimation techniques with a higher performance are required, even at the expenses of inferior accuracy.

All of the above constraints must be met while keeping a reasonable estimation accuracy. Relative accuracy is important especially when designers need to compare alternative solutions. Absolute accuracy is important especially when designers want to evaluate the performance of the same algorithm on different architectures.

Unfortunately, as the sections dedicated to the related work will show, none of the approaches currently available tackle appropriately all the above requirements at the same time. For example, instruction-set simulation does not fulfill the speed and source-level requirements. Static timing analysis does not fulfill the dynamism requirement. Current source-level techniques and black-box techniques do not satisfy the detail requirement. An informal summary of the above considerations is given by the table below:

	source level	fine detail	dynamic	fast
static techniques			no	
instruction set simulation (ISS)	no			no
ISS with counter sampling		no		no
current source-level		no		
black-box techniques	no	no		

This thesis is dedicated to the research of an estimation technique which fulfills the above requirements.

1.2 Designing embedded software is getting more and more difficult

In the previous section, I derived the requirements for this research starting from designers' needs. This section is devoted to clarifying and motivating these needs.

In the last forty years we have assisted to a sustained growth in the ability of silicon manufacturers to fit more and more transistors in the same area, and to raise the clock frequency of their devices. The processing power made

[1] I have expressed the belief that there is a long-term drift toward higher levels in system simulation. I believe the reason is the following: the complexity of systems is growing faster than our capacity to simulate them, therefore a shift to-wards higher abstraction levels, where less details need to be simulated, is needed. I have been requested to comment on this. My interlocutors' belief is that the increasing complexity of simulating more complex systems could be counterbalanced by the increasing amount of computational power that these systems make available. I do not subscribe to this point of view. My forecasts will be certainly either proved or disproved in a twenty-year span.

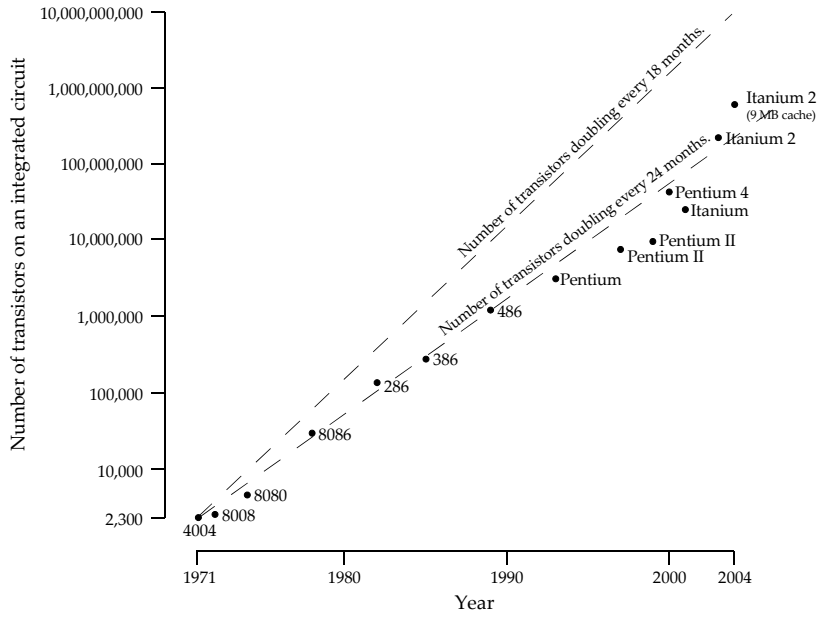


Figure 1.1: Growth of transistor counts for Intel processors (dots) and the Moore's Law (dashed line) with a 18-month and 24-month doubling period.

available by microprocessors and programmable devices grew accordingly. Gordon E. Moore was the first, in 1965 to recognize [1] that the transistor density was growing exponentially over the years, and to capture this observation in his famous "law", originally formulated as: «The complexity for minimum component costs has increased at a rate of roughly a factor of two per year». Moore's law and its less orthodox corollaries, which extend its predictions also on clock frequencies, process feature size, processing power (measured according to arbitrary benchmarks), and RAM capacities, still seem to hold on, even though undergoing a number of revisions through the years [2, 3].

There has been, and still there is, a number of potential threats which challenge the almost prophetic beneficial effects of this law in the coming future, mainly technological barriers such as power consumption, heat dissipation, and the approaching of atomic dimensions [5]. Nevertheless, the International Technology Road-map for Semiconductors [4], still estimates for the period 2003–2009 an exponential growth trend for the "functions per chip at introduction" (measured in millions of transistors), with an annual increase equal to 26%, which leads to doubling this quantity every 3 years.

One of the consequences of the availability of more and more computational power on a single chip in the embedded systems field was the dif-

fusion, on portable devices, of a number of complex and computationally heavy applications, which were previously possible on higher-performance systems. The availability of higher computing power and storage densities on portable terminals has enabled, in 1990s, the migration of a number of applications from analog to digital platforms, e.g. the switch from the first- to the second- generation cellular mobile telephony systems (such as ETACS and GSM respectively in Europe). The same effect is currently driving the migration to the third-generation mobile communications (e.g. the UMTS system [7]), which enable the delivery to user terminals of services such as full-motion video, Internet access, and videoconferencing; or the digital audio and video broadcasting (DAB and DVB-T). In the meanwhile, researchers are facing the challenges of fourth generation mobile platforms, which will have even more complex and dynamic algorithms.

At the same time, video players for DVDs and for high-compression digital formats (such as DivX), which were once available only as appliances, are now starting to appear in the form of portable devices. In the current and future years, developers will face the new challenges associated to the realization of portable devices able to deliver more and more complex contents, such as synthetic video flows, content-based manipulation, animated mixed media, for example as specified in the numerous parts of the MPEG-4 standard [8, 9]. The growth in the scale, complexity and flexibility of the applications that the designers want to implement on portable, battery-powered embedded systems is impressive.

Additionally, there is an increasing demand in the connectivity and interoperability of personal devices: more and more systems come now equipped with wireless devices on board, such as 802.11 wireless LAN [10] network adapters, or Bluetooth [11] personal area network adapters. Such a dynamic communication scenario introduces additional challenges, because modulation methods are becoming more and more complex and refined, to maximize signal-to-noise ration, bandwidth, range, spectrum usage and noise immunity, whereas communication protocols are increasing their complexity to serve the increased demands in flexibility, mobility, interoperability and security. In detail, modulation standards are evolving in order to better exploit the spectrum slices where less noise is present (e.g. with orthogonal frequency-division multiplexing, OFDM), and they are employing multiple transmitter and receiver antennas (multiple-input multiple-output, MIMO) to increase the data throughput (by exploiting spatial multiplexing) or to increase the range (by exploiting spatial diversity). Both OFDM and MIMO are part of a future extension to the 802.11 standard, on which the IEEE announced to start working in January 2004. On the other hand, new proposals are reaching maturity which try to adapt the current network protocols to new applications (such as sensor networks) or to extend the flexibility of the current ones, for example to allow the dynamic adaptive routing of packets over ad-hoc wireless networks, whose topology changes over time with the dynamic addition, leave and mobility of participants.

One of the main threat to the feasibility of products exploiting the new technologies and applications presented above is the increased importance of their computational requirements, which immediately impacts on their

energy consumption. Energy consumption has always been a parameter of extreme criticality in the design of all the embedded systems which depend on a limited source of energy. And in the above scenario, the energy-efficiency of the software components is dramatically more important than in the past. The degree of energy optimization of the software running on a battery-powered embedded device can determine its commercial success, and in cases indeed its feasibility. It is out of question that even a feature-rich portable product would be scarcely appealing to the customers, if its battery lifetime is short enough to makes it unusable.

The estimation of the energy demands of an innovative application is dramatically more difficult than it used to be years ago. In the past, when the first digital embedded system employed no or little software (written in assembly language), the estimation of their consumed power depending on the the different possible operating conditions was a moderately simple task, involving for the largest part considerations at the analog electronics level. Nowadays, embedded system comprise large software components, written in high level languages, which often include a complete operating system with a network stack and a middleware layer, and the load imposed on the system by modern applications such as the decoding of natural or synthetic video sequences, is extremely dependent on the data, and largely variable over different conditions.

Despite the criticality of this task, embedded software designers have limited means to evaluate the performance of their software and, consequently, to explore and compare optimization alternatives. Traditionally, they employ power-enabled architecture simulators or instruction-level simulators in order to obtain execution time and consumed energy estimates for the software they write (I report a complete review of these tools in Chapter 2). These tools provide accurate estimates, but they are very slow: their simulation times can be hundreds to tens of thousands time slower than their actual execution times. In such a scenario it is difficult for the designer to operate alternative choices among different libraries, algorithms, implementation alternatives, optimization techniques. Additionally, the slowness of instruction-level simulation techniques practically prevents the performance evaluation of very complex applications: simulating the decoding of a few video frames encoded according to the novel standards may require many hours of computation on the fastest workstations. Simulating the decoding of a complete typical movie could require months. Simulating the decoding of different movies would require years. This makes practically unfeasible the estimation of the system workload in the average- and worst-case scenarios, because this would require unacceptable simulation times.

Additionally, instruction-level tools provide estimates which are meaningful at the instruction level, whereas the complexity of contemporary application has forced developers to abandon almost completely the coding in assembly language. Embedded software developers now write code in high level languages. Due to the advent of complex architectures featuring deep pipelines, superscalarity and wide-issue or VLIW, the compilation steps from high-level languages to assembly has become less and less intuitive and transparent, and developers encounter increasing difficulties in

	Language	Number of lines of source code	Percentage
1	C	21461450	71.18%
2	C++	4575907	15.18%
3	Shell (Bourne-like)	793238	2.63%
4	Lisp	722430	2.40%
5	Assembly	565536	1.88%
6	Perl	562900	1.87%
7	Fortran	493297	1.64%
8	Python	285050	0.95%
9	Tcl	213014	0.71%
10	Java	147285	0.49%
11	yacc/bison	122325	0.41%
12	Expect	103701	0.34%
13	lex/flex	41967	0.14%
14	awk/gawk	17431	0.06%
15	Objective-C	14645	0.05%
16	Ada	13200	0.04%
17	C shell	10753	0.04%
18	Pascal	4045	0.01%
19	sed	3940	0.01%

Table 1.1: The various programming languages, sorted by the total number of source lines of code, in a representative Linux distribution [12].

interpreting instruction-level estimates obtained by the above tools.

The gap between the source and the assembly level increases, and the burden of bridging this gap in order to understand how to optimize the source code is entirely on the developer's shoulders. Additionally, source-level transformations have been shown to produce the highest energy gains. All these reasons motivate the crucial need for source-level, fine-detailed, fast, dynamic and accurate software estimation engines.

1.3 Why I choose the C language

A source-level estimation technique relates its estimates to entities in a source code, and a source code is a program, written in some programming language. Therefore, the choice of the programming language in which the source code is described is a crucial element for this technique.

The technique I propose is largely independent from the language to which it is applied. Although I show an instance of this technique which applies specifically to the C programming language, the technique does not rely on any specific feature which is provided by C only. Porting the same technique to another language, especially imperative languages is, in most cases, just a matter of syntactic flavors.

Nevertheless, the choice of the C language for the particular instance of the technique which I show in this thesis demands some justification. I choose C because it is currently the most widely used programming language in the embedded design community.

Table 1.1 lists the programming languages present in a popular Linux distribution, sorted by relevance (measured in physical lines of code), published in a 2001 study by Wheeler [12]. It may be argued that a Linux distribution is not a representative embedded application, and that even if it was, it is not representative of the market. Nevertheless, these are significant figures because they are measured in an objective, quantitative, non-debatable way, and because Linux (in its flavors) is one of the leading operating systems in the embedded design community, and more and more embedded systems feature some Linux distributions.

The study comments on the predominance of C, forecasts that this predominance is not to disappear soon, and motivates why:

> «[...] C is pre-eminent (with over 71% of the code), followed by C++, shell, LISP, assembly, Perl, Fortran, and Python. Some of the languages with smaller counts (such as objective-C and Ada) show up primarily as test cases or bindings to support users of those languages. [...]
>
> C++ has about 4.5 million lines of code, a very respectable showing, but is far less than C (over 21 million SLOC[2]). Still, there's increasing use of C++ code; in the last survey, C had 80.55% and C++ had 7.51%. There is slightly less C code in the total percentage of code, most of which is being taken by C++. One could ask why there's so much more C code, particularly against C++. One possible argument is that well-written C++ takes fewer lines of code than does C; while this is often true, that's unlikely to entirely explain this. Another important factor is that many of the larger programs were written before C++ became widely used, and no one wishes to rewrite their C programs into C++. Also, there are a significant number of software developers who prefer C over C++ (e.g., due to simplicity of understanding the entire language), which would certainly affect these numbers. There have been several efforts in the past to switch from C to C++ in the Linux kernel, and they have all failed (for a variety of reasons).»[12]

The above figures and comments suggest that the residual useful life of the C language is still long, and that a software estimation technique based on C (like the one proposed here) will be of great practical usefulness still for a long time to come. Were this not enough, the most important competitor of C is C++, and I will show in Section 5.3.1 (page 242) that the technique described in this thesis is extensible also to C++ with no significant conceptual difficulties, and with a reasonable effort.

[2] SLOC = physical Source Lines Of Code

To learn more in the exact version of the standard I choose to adopt for the implementation of the tools associated with this thesis, refer to Section 3.3.2.1 (page 81).

1.4 The fundamental approach of this thesis

In this section I provide a quick overview of my approach. I summarize its fundamental steps and motivate their choice in terms of the requirements expressed in Section 1.1 (page 21). The same reasoning is also summarized in Figure 1.2.

First, from now on I will generally use the term *cost* to denote execution time and energy consumption. Executing an instruction will cause some cost. Executing the assembly translation of a source code will have a cost. Executing an entire program with given data on a given architecture will have a cost. All this thesis is about cost, in this sense.

The 'source level' and 'fine detail' requirements constrain the estimation technique to provide results to the user at a precise abstraction level. The fundamental entities of this level must be the source-level objects in terms of which the designer thinks his program. This entities are identifiers, operators, expressions, statements, functions. The best representation for information at this abstraction level is the Abstract Syntax Tree (AST). A decorated AST is the perfect container for all the information that the methodology needs to collect, produce, store and report. For this reason, the entire technique will be AST-centric.

The AST level is also the perfect point where to make a "divide and conquer" operation. Although it is not generally true that the execution of the same portion of code leads to the same time and energy time consumption, this assumption leads to negligible estimation errors, which are statistically obliterated. This assumption is of great help to our approach. I express it more precisely in the following way:

$$C_i = n_i \cdot c_i$$

> the total cost C_i of executing a given AST node i shall be the product of n_i, the number of times it was executed (i.e. its execution count, or *profile*), and c_i, its *single-execution cost*.

This assumption allows me to split the problem of determining C_i into the two separate subproblems of determining c_i and n_i.

The determination of c_i needs not to be dynamic (thanks to the above assumption) and is, in the end, an arbitrarily complex static analysis of the AST. The technique I choose to adopt for this specific task is based on a multi-visit attribute grammar over the AST. It is probably the main single contribution of this thesis, and Chapter 4 is entirely dedicated to its description.

The task of determining n_i could be carried out exactly or, worst-case bounds could be found for it. Since modern applications are becoming more and more dynamic in nature, worst-case estimates are not representative of

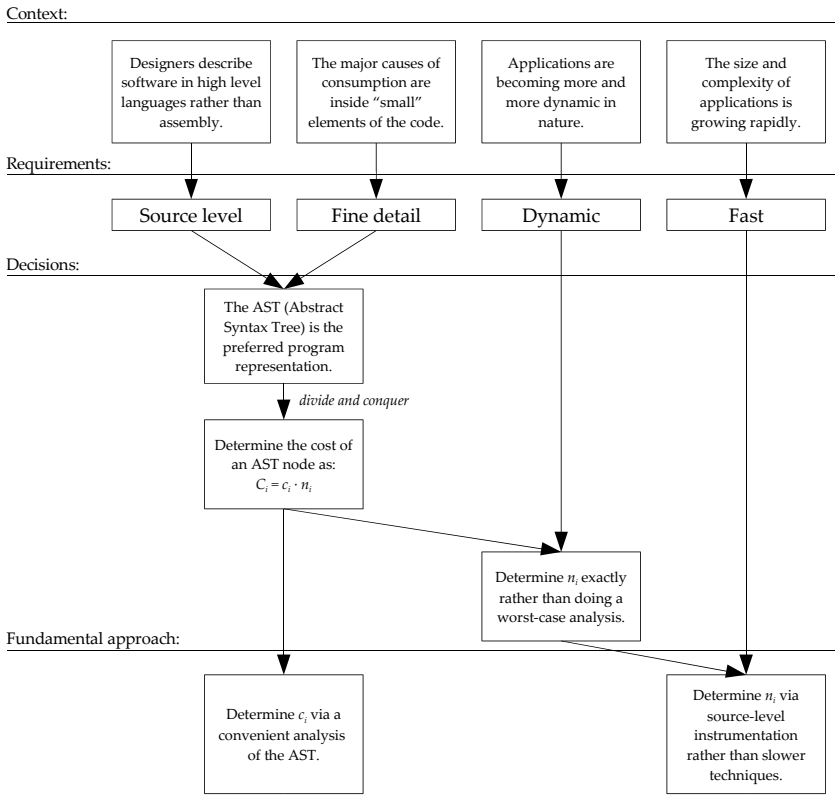


Figure 1.2: How the fundamental approach of this thesis derives from the designers' requirements which, in turn, derive from the current context in the embedded design scenario.

the actual system properties anymore. This is the spirit of the 'dynamic' requirements. Therefore, n_i should be determined exactly. This could be done at many layer of abstraction, but at the lower ones, this operation is computationally too demanding. As motivated before, for many modern applications instruction-level simulation is not a viable choice. For this reason, I choose to employ a source-level instrumentation technique which exhibits minimal overheads still allowing the exact determination of all the n_i. I propose an optimal source-level instrumentation technique which allows the determination of all the n_i, but which requires the insertion of one probe every generalized basic block (I will define this term later). I will show that simulation of source-level instrumented code is on the average 10,000 times faster than instruction-level simulation, and only 2 times slower than the execution of the same program at natural full speed, i.e. without any instrumentation.

As a consequence of this line of reasoning, the desired technique (which

is the objective of this thesis) should accept inputs, yield outputs, and be composed as I describe below.

The technique shall get as inputs:

- the source code which the developer wants to analyze;
- a corresponding set of input data which the source code accepts;
- a description of the compiler, at an appropriate level of abstraction;
- a description of the architecture and its energy/time consumption.

It shall yield as output an estimation of the energy and time consumed by each individual AST node in the source code.

It shall be composed of three fundamental phases:

1. cost analysis,
2. profiling,
3. collect.

More detailed explanations of the above steps follow.

In the 'cost analysis' phase, the source code is parsed and its AST determined. Then, a static analysis decorates each node with appropriated cost terms. These contributions may be expressed in a variety of forms and abstraction levels, on which I do not need to comment now. The choice of appropriate levels of abstraction for expressing costs is subject to some degree of freedom, and demands a thorough discussion, that I prefer to postpone. The definition of source-level cost terms and their attribution rules is, in fact, the most significant contribution of this thesis.

In the 'profiling' phase, the execution count of each AST node is determined by compiling and running an source-level instrumented version of the original program. This instrumented program is run with real input data.

In the 'collect' phase, C_i terms are determined as the product of c_i and n_i terms, determined above. Then, c_i costs are translated from the abstract units in which they are expressed, to physical units of time and energy (seconds and Joules). Also, statistical corrections may be applied to take into account second-order effects which are not accounted by the previous steps or are neglected by the above "divide and conquer" approach.

1.5 Many techniques are possible, just one is chosen

The above general approach allows many degrees of freedom. Many estimation techniques may be built, which all show the above general approach. I choose to develop and describe in detail just one of them, for reasons of time, simplicity and incrementality. In this section I detail and motivate this choice.

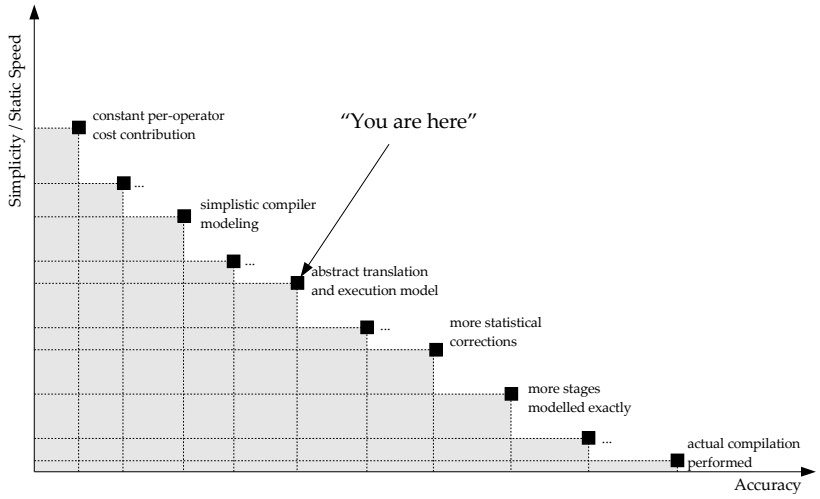


Figure 1.3: Even after discarding the non-Pareto-optimal ones, many techniques are possible within our fundamental approach. They exhibit different trade-offs between static speed and accuracy. In this thesis I choose one, which is based on a model of the architecture and of the compiler.

The 'cost analysis' step described above may be carried out in a very broad variety of ways. The very purpose of this step is to model how the source code contributes to spending time and energy when executed on the target processor. This is a complex process, which involves modeling how the compiler and the architecture work. Different approaches may be chosen in order to model the compiler and the architecture. For each of the stages which compose the compiler, and for each of the considered architectural components (provided that the technique remains 'fast'), one may choose to account exactly for the components' behavior, or to summarize its behavior in a small set of statistically-tuned parameters, or to model one of the many possible abstractions that lay between these two extremes.

More detailed models are more complex (therefore more difficult to build and to maintain, and slower to execute), but they may ensure better accuracy. All the possible modeling choices can be represented in a complexity/accuracy space. Among these models, the ones which are non-Pareto-optimal do not deserve any further attention, because they are surpassed by some other model in both the objectives. The remaining ones generate a discrete (very large) set of models which I intuitively represent in Figure 1.3.

The space and time allowed by a doctoral thesis permit the complete analysis of just one of these many points. I choose to focus my attention on single-issue architectures. The fundamental approach presented here can be extended also to more complex architectures. I have recently worked on an extended version of the estimation technique presented here, which is able to

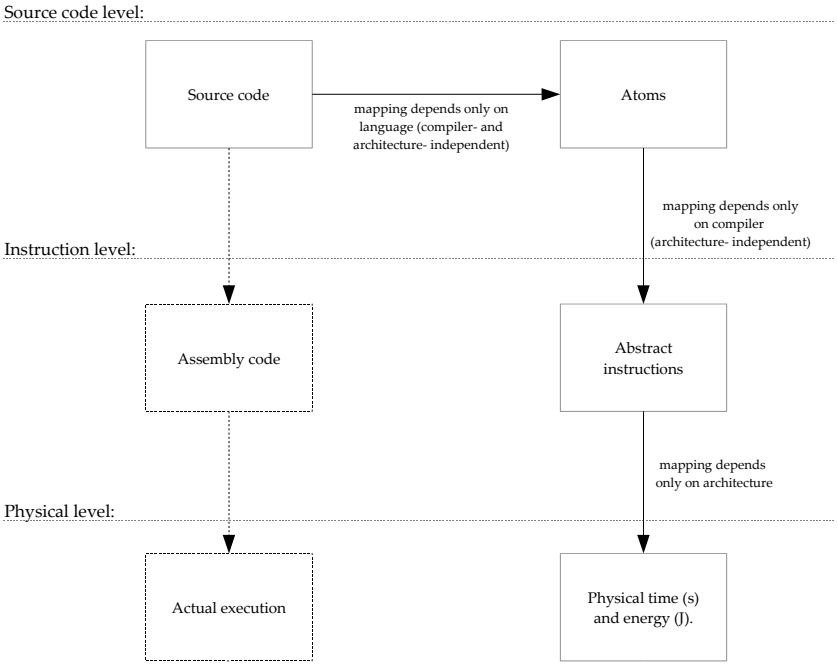


Figure 1.4: The physical occurrence of cost (time, energy) is a complex process, depending on language, compiler and architecture. In order to tackle one problem at a time, I introduce atoms and abstract instructions.

perform source-level estimation for VLIW architectures and data-level parallel architectures. I will not support these claim in this documents, as it would require too much time and space.

Single-issue architectures allow assumptions which facilitate modeling: more precisely, it is possible and convenient to model their instruction set in terms of smaller terms named *abstract instructions*, such that an abstract instruction represents at the same time a time and an energy contribution. For these architectures, it is true (with a negligible error) that abstract instructions are additive. Therefore, executing two sequential abstract instructions will cause a cumulative latency which is the sum of the latencies of the two sequential instructions, and an energy consumption which is the sum of the energies of the two abstract instructions.

For my convenience, I introduce an additional abstraction layer: the *atoms*. An atom is a source-level cost term, which represents the cost (of a portion of it) of executing a given source code element. Atoms are, in a way, the source-level unit of measurement of cost. The rules which determine what is the cost of an AST node expressed in atoms are compiler- and architecture-independent.

Thanks to abstract instructions and atoms, it is possible to account for the

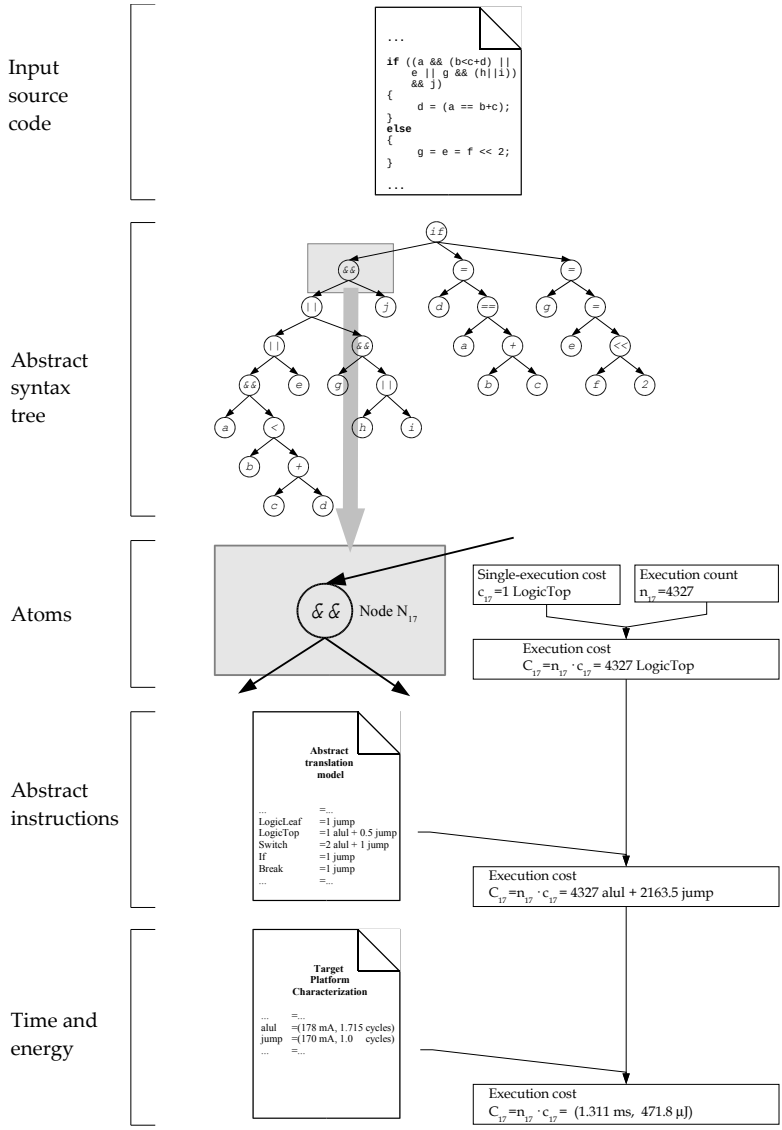


Figure 1.5: Abstraction levels involved in this methodology.

entire compilation and execution one element at a time, in a 'divide and conquer' style. Figure 1.4 depicts this concept. My technique associates atoms to AST nodes by just considering its language model, then translates atoms to abstract instructions by only considering its compiler model, finally it converts abstract instructions into time and energy values by only considering

its architecture model. An example of this process is given in Figure 1.5.

1.6 The final objective of this thesis

In the light of all the above considerations,

> this thesis researches a source-level, fine-detailed, dynamic and fast technique to estimate the execution time and the energy consumed by a given single-issue processor core when executing a program written in the C language, running on given input data.

The technique must comprise appropriate tools to support the estimation of the performance of realistic projects without need for modifications by the developer, and appropriate visualization tools which allow the inspection of the source code in search for the most resource-consuming portions. The methodology should be able to suggest which are the most beneficial source-level transformations to apply to a given project, and where to apply them.

1.7 Advantages of this approach

The approach proposed here offers a number of advantages with respect to its competitors:

- faster-than-real-time speed:
  the optimized source-level instrumentation technique I propose and employ for profiling impacts minimally on the speed of the instrumented program: profiling times are on the average 10 000 times shorter than ISS simulation times (when run on the same platform), and only 2.2 longer than the original, non-instrumented executables. Additionally, source-level profiling provides to users the possibility to profile their source code on a generic simulation platform which does not need to be the target platform. These two features, combined, allow to completely decouple the profiling speed on the simulation platform from the execution speed on the target platform. Since developers typically employ simulation workstations which are at least one order of magnitude faster than the embedded platforms for which their applications are targeted, this results in a simulation speed which can be actually faster than real-time;

- high level of abstraction:
  instruction-level estimation techniques can provide information at the level of the microprocessor's instruction, instead this technique provides estimates for each element of the original source code. There is a fundamental difference in the level of abstraction and the usability of the two pieces of information. The former is loosely coupled with the source code on which the programmer is working, and its usability for exploring the optimization space relies on the programmer's deep understanding of the target assembly language and insight on

the compiler's behavior. The latter, on the other side, is immediately usable. Souce-level estimation provides estimates for operators, at the same level of abstraction at which the user is thinking.

- silicon not required:
  users do not need the physical availability of their target platform. In fact, in order to be usable for estimation with respect to a given target platform, our methodology just needs an appropriate set of statistical parameters which describe that platform. Target specialists can determine these parameters and made them publicly available, enabling any developers to estimate and optimize their software for the target platform, even when the platform has not been released and without disclosing any intellectual property;

- build toolchain not required:
  source-level estimation is the only technique which yields accurate estimates for a specific scenario (with a given datapath width, given register width and length, ...) even when an entire toolchain (compiler, linker, simulator, instruction-level power model) is not available for the given scenario;

- purely symbolic evaluation possible:
  unlike any instruction-level simulation technique, this source-level estimation technique can provide architecture-independent estimates. These estimates allow the comparison of different implementations of the same algorithm, and allow the exploration of optimizations that improve the performance of the source code independently from the underlying architecture;

- integration with library and operating system call estimation flows:
  a source-level estimation technique needs to appropriately interface with other techniques designed to estimate portions of the software whose source code may not be available (libraries, operating system). The technique I present provides a seamless way to integrate the results of such estimation flows.

1.8 Frequently raised objections

When working in a research area which is populated and has been thoroughly investigated in the past, it is often difficult to convince one's own audience about the novelty of his approach and its soundness. Energy estimation for embedded system is for sure such a field. Additionally, in the specific topic covered by this thesis, research papers often "overclaim" their accomplishments in their titles, with respect to the actual contribution described in their texts. This practice pollutes the research concept space, inducing the idea that a problem was effectively solved when it was in fact just described, or tackled incompletely.

This section is designed to help reviewers not being mislead by these practices, and to anticipate criticism. It is therefore structured as a "Frequently Asked Questions" section of a manual. It is written in an informal, straight-to-the-point style. The reader will indulge me, as long as my claims are correct and motivated.

1.8.1 «Your novel contribution is not quite clear»

This work has novelty in its objective, in its method and in its results. The scientific merit is obviously in the method. This technique is the first work which attempts the definition of source-level cost terms, and which defines a set of rules, based on the language semantics, which determine this cost.

1.8.2 «Source-level estimation has already been done!»

Yes and no. The matter is ambiguous. It all depends on how you define "source-level estimation". Since this ambiguity is a potential danger to understanding the contribution of this thesis, then I prefer to disambiguate the term.

I motivated above why designers need a technique to characterize fine-grained elements of the source code, and such technique must rely on a source-level simulation (i.e. it must be fast). Because of this need, I define "source-level estimation" the ability to provide individual estimates for each source code entity, obtained while performing simulations at the source level.

Accordingly with this definition, this thesis presents **the first** source level estimation technique. No other techniques before have provided fine-grained estimates without resorting to lower level simulations (typically, the instruction level). On the other hand, all the techniques which employed source-level simulations, were unable to characterize fine elements (typically, the C function).

The terminology is debatable, you may apply broader definitions of the "source-level estimation" term. In this case, a number of works in literature may be considered to have already applied this technique. Nevertheless, the above considerations must be kept clearly in mind, otherwise the novel contribution of this work is neglected just for lack of precision in terminology.

1.8.3 «Your approach is too limited»

This thesis presents a broad methodology which may apply to a variety of architectures, but it details only one instance of the possible methods which may derive from it. This instance is especially suited for single-issue processors. I did that because I wanted to stick to a good principle: do one thing, do it well.

I claim that a solid but narrow theoretical foundation which may be extended with some effort is more desirable than than a set of broader but less solid foundations. It is clear that in the scope and duration of a single Ph.D. thesis it is impossible to achieve breadth and solidity at the same time. That

is obviously the ideal goal, but not practically achievable. To some extent, generality can be traded with accuracy, and I have chosen to privilege accuracy.

1.9 Estimation: A motivational example

This section illustrates how the technique applies on a practical example.

Consider the following fragment of code, taken from a real FFT implementation [150]:

```
74   for (i=rev=0; i < NumBits; i++)
75   {
76       rev = (rev << 1) | (index & 1);
77       index >>= 1;
78   }
```

With our technique, the designer obtains estimates for the loop, for each line of code (as shown in Figure 1.6), and he can also go further into the details of this view and inspect the consumptions of individual constructs and operators (such as '<<').

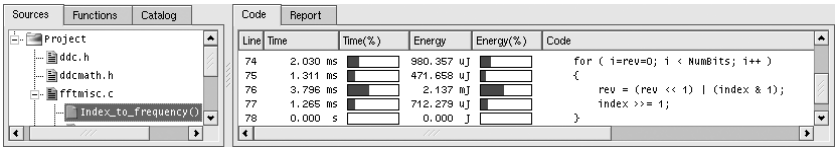


Figure 1.6: This technique allows to estimate the cost of individual loops, lines of code, and finer details, such as individual constructs and operators.

Currently, no other approach to the problem can provide source-level estimates at this fine granularity and with this speed. I motivate this claim in Chapter 2, by comparing this approach with the state of the art. Now I briefly explain how this result is obtained. First, the above code is parsed, thus yielding obtaining the AST in Figure 1.7. Then, for each of the above AST nodes i, the single-execution cost c_i is determined via a multi-visit attribute, described in Chapter 4). c_i is initially expressed as a sum of atoms. Atoms are language-level cost terms, independent from the compiler and the architecture, which represent the contribution to consumption of all the constructs and operators of the C language. Atoms are attributed to nodes depending on the context, analyzed in the attribute grammar.

Atoms have context-independent translation: given a compiler, a translation comprises always the same count and class of abstract instructions. The actual atoms added per each syntax node of the previous code fragment are listed below (nodes not listed have zero cost):

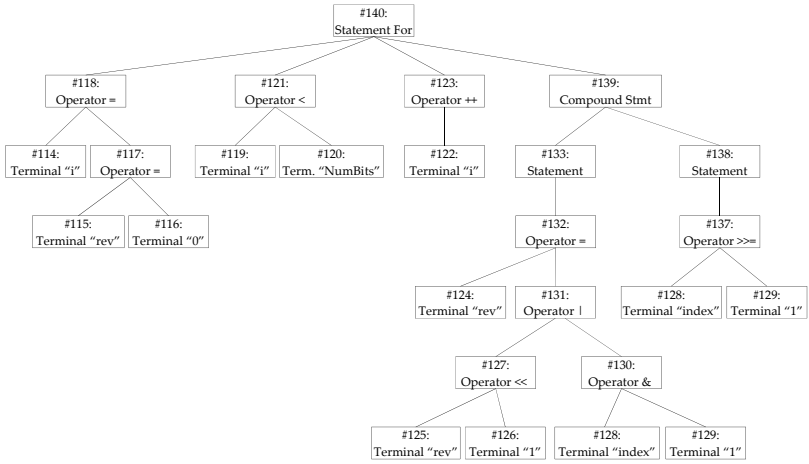


Figure 1.7: The abstract syntax tree corresponding to the sample fragment of code.

Node	Line	Atoms
#117	74	1 Assignment
#118	74	1 Assignment
#121	74	1 IntCompare
#123	74	1 IntAdd
#139	75	1 For
#127	76	1 BitwiseShift
#130	76	1 BitwiseOperation
#131	76	1 BitwiseOperation
#137	77	1 BitwiseShift

The attribute grammar evaluates the nodes' consumption exploiting all the contextual information such as type of variable and expressions, constancy, assignment direction, size of the transferred data and others. This is why two instances of the same operator (e.g. assignments in nodes #118 and #132 above) may have different costs, depending on their context.

Then, c_i terms are converted by mapping their atoms into abstract instructions. This mapping is general, but it can be arbitrarily specialized by amending it with architecture-dependent corrections, which account for instruction- and data-level parallelism. In the case of an ARM processor, it is as straightforward as follows:

Node	Line	Abstract instructions
#117	74	1 alul
#118	74	1 alul
#121	74	1 cmpl
#123	74	1 alul
#139	75	1 jump
#127	76	1 alul
#130	76	1 alul
#131	76	1 alul
#137	77	1 alul

Then, I instrument the code and I run it with real inputs to obtain the execution count n_i for each node. The usage of real input data is an advantage over static techniques, which either always assume the worst case, or interrogate the user. Finally, I determine each node's global execution cost $C_i = n_i \cdot c_i$.

From the cost of each node in terms of classes of abstract instructions, the 'collect' phase determines the consumed time, energy and memory footprint for each node and line of code, also accumulating estimates for the library and operating system calls provided by an external library estimation flow, thereby obtaining the view presented at the beginning of this section.

1.10 Source-level estimation can speed up optimization

In the previous sections I have introduced the novel estimation technique presented in this thesis. In this section, I discuss how this technique can enable a new generation of short-loop methodologies to explore the optimization space of source-level transformations. The proposed approach is especially suited for local transformations which exhibit negligible inter-effects.

Given the increasing importance of pervasive computing applications (e.g. ubiquitous computing, sensor networks, intelligent patches, etc.), and their stringent energy budgets, the efficiency of embedded software is a critical parameter. The degree of optimization of embedded software can determine the commercial success, and in cases indeed the feasibility of battery-operated products. As said before, the majority of embedded software developers does not write assembly code anymore. They employ high-level languages, mainly C, for reasons of portability, ease of debugging and maintainability. Unfortunately, the degree of optimization provided by conventional compilers for these high-level languages is much lower than writing efficient source code. Source-level transformations have been extensively studied as a means to reach this desired degree of optimization. They are not only highly portable, but have been proved to provide a much larger scope for performance improvements than any other low-level technique.

Nevertheless, selecting which transformations to apply –and where– is still an effort-intensive task, especially for modern, large-sized applications. From now on, I call *transformation steering* this task of selecting and targeting transformations.

Embedded software designers have limited means to explore the optimization space of source-level transformations. Optimization flows are often based on instruction set simulation (ISS), which exhibits unacceptable simulation times with modern applications. These flows have a long exploration loop: in order to use them, the developer must iterate many slow steps, mainly target compilation and instruction set simulation, as depicted in Figure 1.8(a). I will provide a complete discussion on these these approaches in Section 2.2 (page 58).

Thanks to the availability of source-level estimation techniques like the one presented in this thesis, I claim that it is possible to realize a short exploration loop, like the one in 1.8(b).

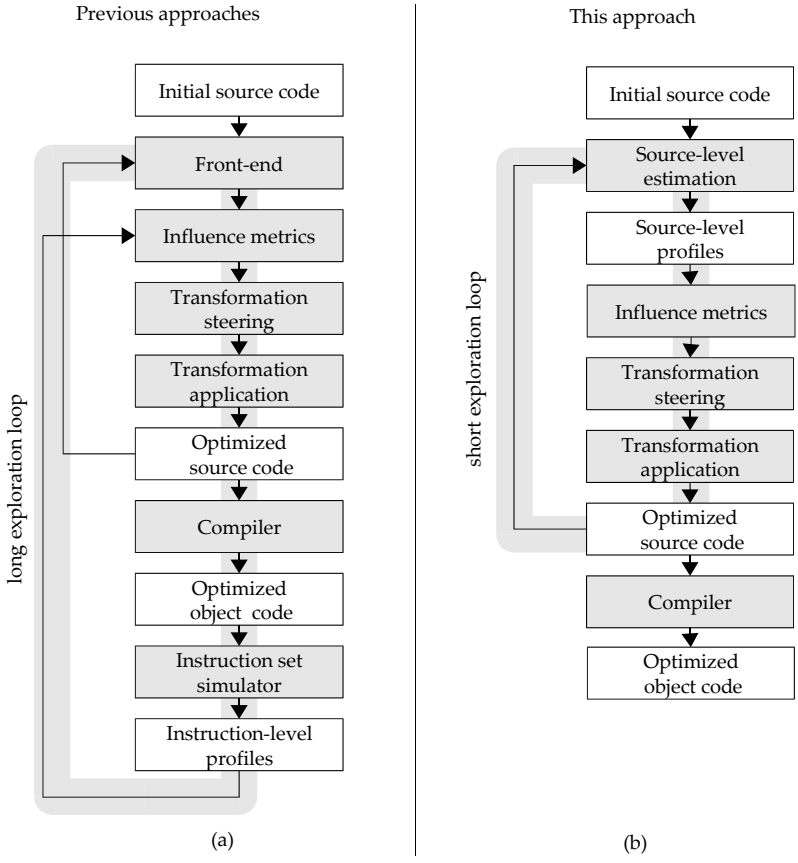


Figure 1.8: This approach allows a much shorter and quicker exploration loop, thanks to the use of source-level profiles in place of assembly-level profiles.

In the current state-of-the art, methodologies and tools for all the blocks in Figure 1.8(b) are present, except for the block "Transformation steering" block. Without this block, designers must either resort to ISS-based tools, which are too slow for modern applications, or perform transformation steering by hand, which is also impractical due to their large size. Here, I propose a methodology to fill this gap, automatically performing the transformation steering, on the basis of the source code of the program, and its source-level profiles. My approach generates a set of optimization guidelines specifying which transformations to apply and where to apply them.

1.11 Optimization: A motivational example

In order to provide a quick overview of the possibilities offered by this technique, I show how it is applied on a sample benchmark. As a benchmark, I employ a computer-vision application: a Hough transform used to detect vertical lines in panoramic images captured from conical mirrors. I import the benchmark project in the source-level estimation flow. After setting the architectural parameters and running the estimation, the time and energy estimates per line of code, as shown in the screenshot in Figure 1.9.

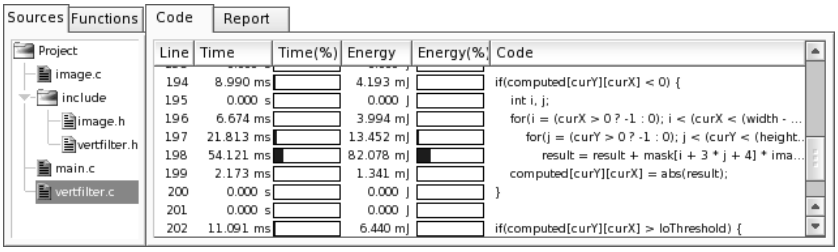


Figure 1.9: Time and energy estimates for a critical section of the example benchmark, as reported by the source-level estimation flow.

The source-level estimation of the entire project on a realistic input image takes less than a second on a modern workstation, and the following time and energy consumption figures are returned:

File	Time		Energy	
image.c	21.638	μs	16.561	μJ
main.c	28.962	μs	21.158	μJ
vertfilter.c	377.672	ms	421.048	mJ
(glibc)	305.800	μs	622.000	μJ
TOTAL	378.029	ms	421.708	mJ

I now employ the original transformation steering engine I propose. The engine takes as inputs the source-level analyses and profiles made available by the source-level estimation flow employed in the previous step, and it generates a list of optimization directives. Each directive consists of a title, such as "inline this function", a score, and the target point (file, function, line interval). The score is a number between 0 and 1 which indicates how much that transformation is beneficial for the program. It is calculated according to fuzzy-logic algorithm described in Section 3.4 (page 90), based on metrics to measure the influence of transformations already presented in literature. Figure 1.10 gives a visual representation of the set of optimization directives generated for the benchmark project, sorted by descending score.

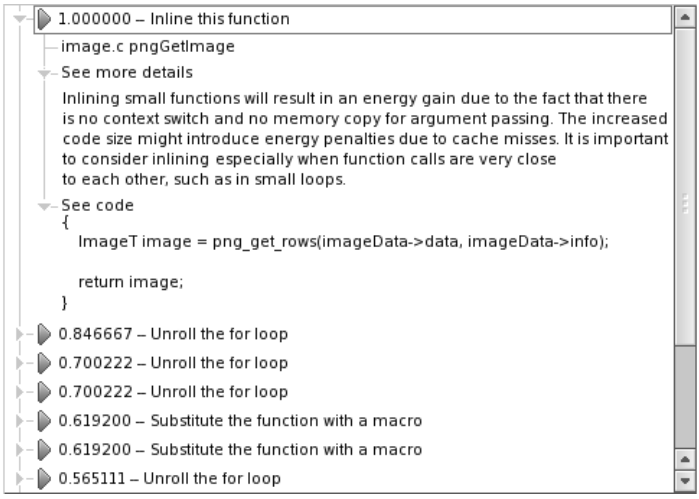


Figure 1.10: The list of optimization directives generated by the optimization flow when applied on the example benchmark.

It is beyond the scope of this research to also perform automatic application of source code transformations. Mature tools are available for the purpose and there is no conceptual difficulty in utilizing the optimization directives I generate to drive those tools. As an example, I apply by hand the first optimization directive in the above list.

Then, I run the source-level estimator over the transformed source code in order to verify the impact of the transformation, obtaining the figures below:

File	Time		Energy	
image.c	21.638	μs	16.561	μJ
main.c	28.962	μs	21.158	μJ
vertfilter.c	356.222	ms	396.261	mJ
(glibc)	305.800	μs	21.158	μJ
TOTAL	356.509	ms	396.921	mJ

Again, the estimation process takes less than a second. The figures show that the applied optimization caused an approximate 6% decrease in execution time decrease and energy consumption. As this motivational example shows, the proposed engine allows a fast, automated, profile-based selection and targeting of optimizations. Once connected with a tool able to apply them automatically, the optimization of even large scale programs can be performed in a small amount of time.

1.12 The organization of this thesis

This thesis is organized as follows.

In this chapter, I have proposed a fundamental approach to the estimation of the execution time and energy consumed by a program. The fundamental approach is broad and may lead to a large number of specialized instances, targeting narrower domains and internally relying on different sub-approaches.

In Chapter 2 I present, for the research areas related with the main objectives of this thesis, the most important works present in the literature. For the works which present competing approaches, I try to give a critical comparison, illustrating advantages and shortcomings.

In Chapter 3 I present the complete details of one of the possible instances of techniques inspired by my fundamental approach. The Chapter discusses the ideas which are at the basis of proposed technique, and all the steps which allow to realize it.

Chapter 4 illustrates the core of the methodology: a model for the determination of the execution cost of syntax elements in a given source code. I define this model in the form of a multi-visit attribute grammar. The semantic rules in this model account for all the semantic aspects of the C programming languages, including the type system, constant expressions, position with respect to assignment operators, et cetera.

Chapter 5 presents experimental results which prove the accuracy and utility of the technique presented, it draws final conclusions on the quality and breadth of the theory and results proposed here, and it sketches the current and future developments on the topic.

Chapter 2

Background

THIS chapter summarizes the state of the art for the research areas related with the main objectives of this thesis. It compares the most relevant works, also illustrating their advantages and shortcomings.

2.1 Performance estimation techniques

2.1.1 Static timing analysis

One of the techniques which has been traditionally employed to estimate the of software and, more generally, of systems including a software portion, is static timing analysis (STA).

The main objective of static timing analysis is to determine bounds on the execution time of a program on a given architecture, and the most useful measure in static timing analysis is the worst-case execution time (WCET). The WCET of a program is a critical factor in the design and verification of real-time systems. The determination of the WCET of a piece of software is, in general, a complex task which depends on the software and hardware components which compose the system under analysis: it must take into account software factors which include the operating system and its scheduling policies and the application (the execution flow, loop iterations, function calls) and hardware factors, such as interrupt handling mechanisms, caches and pipelines. A number of techniques in literature addressed the problem [19, 21, 22], typically by dividing the problem into the subproblems of program flow analysis and micro-architecture modeling.

Unfortunately, though being of major importance in the verification of real-time applications, static timing analysis becomes less and less useful in the general case. In fact, significant problems limit its usefulness when

performing system dimensioning, design space exploration, hardware/software partitioning and source-level optimization exploration:

- First of all, for the above purpose the designer may be much more interested in estimating the typical performance of the system in a set of realistic cases, representative of actual conditions in which the system will be employed, rather than in obtaining a theoretical proof of the performance in a worst-case, probably unrealistic, context. This is especially true for systems which feature user interaction or systems which are designed to deal with a multimedia or data stream. For example, the designer of a portable video player needs to know how much energy is required on the average to decode a frame, he needs to know what is the average power absorbed by the processor when decoding a real video, with a realistic distribution of I, P and B frames, and with real motion compensation data. Simply assuming the worst limits per each loop leads to estimation results which do not satisfy this need. In addition, the gap between typical and worst-case performance indicators is increasing in modern, highly dynamic and data-dependent applications like synthetic video decoding, interactive applications, dynamic topology networks, etc. In these applications the computational workload strongly depends on the input data, whereas static analysis considers only the algorithm. Studies [26, 27] report that worst-case analyses of the performance of video encoders lead to overestimates of one to two orders of magnitude with respect to the actual values.

- Furthermore, reasoning about the WCET is an undecidable problem, unless serious constraints are imposed on the program under analysis. These constraints limit very seriously the designer's freedom in terms of language constructs and programming style he can use. For example, the program must not contain dynamic data structures, unbounded loops, direct or indirect recursion or dynamic function references [19, 20]. Unfortunately, modern application are increasing their complexity indeed in those aspects where static techniques exhibit the most difficulties, especially unbounded loops and use of pointers. In practice, all these applications should be rewritten in order to be estimated with STA techniques, which is definitely impractical.

- WCET techniques often rely on information provided by the user to perform their analyses. This induces a strong limit on the size of the applications which can be practically treated; moreover, some authors [25] express concerns on the reliability of the obtained WCET, which may be error-prone because the information provided by the users is not completely reliable.

- Finally, the architectural modeling part of the timing analysis techniques become more and more complicated in the attempt to model the new complex features present in modern architecture, such as branch prediction, predication and instruction pre-fetching, cache

policies[23, 24]. It is not clear whether and how these techniques will evolve in order to account for the latest features like hardware multi-threading.

In the end, in the light of all the disadvantages just mentioned, static timing analysis does not provide an effective answer to the needs I have described in Section 1.1 (page 21).

2.1.2 Static Functional-level Power Analysis

Julien et al. [45, 46] proposed *SoftExplorer*, a fast technique to estimate the consumption of data-dominated loops from the C source code. It is based on a functional-level model of the architecture, and on statistical parameters extracted from the assembly code in which the original program is translated. The authors have shown the good accuracy of the technique by applying it to a set of multimedia and signal-processing benchmarks, obtaining estimation errors within 6% against measurements. The technique is very fast, virtually running in constant time with respect to the size of the input data of the program under analysis.

The technique has two major drawbacks, which make it inapplicable to satisfy the contemporary design needs:

- the technique is static, therefore it cannot determine the actual runtime execution flow of the program under analysis, i.e., the number of iteration of a loop. To overcome this limitation, the user must provide this data, which is impractical for programs of realistic size, and error-prone in all the circumstances;

- the technique is designed to work effectively on data-dominated software, without any dynamism. It is not clear whether and how it is possible to extend this technique in such a way that it handle source code with control-dominated portions and dynamic behavior. For example, it is impossible to analyze the motion compensation code in most implementations of video decoders (for example the H263 implementation by Telenor [151]) which are highly control-dominated and exhibit a strongly dynamic data-dependent behavior.

2.1.3 Instruction-set simulation

Tiwari at al. [28] were the first, in 1994, to recognize the increasing importance of estimating the software power consumption in the embedded systems. They also recognized the inadequacy of techniques such as circuit- or gate-level estimation for the purpose: in the best case, these techniques were too slow; otherwise they relied on low-level details of the modeled processors, which are not usually available. The authors developed an instruction-level power modeling technique, applicable on both processors and embedded cores. The resulting models are useful to evaluate software, for example to verify if it meets its power constraints, or to search the design space in software power optimization. In the experiments, authors employed a

digital ammeter and measured the current absorbed by a 486DX2 processor board when executing loops of instructions. The instruction loops were designed with the exact purpose of providing a stable power consumption read, therefore they were long enough to make the effects of the jump negligible, but short enough to avoid any cache misses. This way, a *base cost* for each instruction was determined, and discrepancies among different executions of the same instruction which employed different data turned out to be negligible. Variations due to circuit state, measured for each couple of instructions, is also negligible. Effects due to cache misses are simply accounted for on a global scale, by measuring the number of cache misses, and multiplying that by the average cache miss penalty. The same authors, on the basis of the above experimental results, also presented an overview [29] on the techniques to reduce energy consumption at the software level, including instruction reordering to minimize switching activity, code generation through pattern matching, reduction of memory operands.

Burger and Austin proposed SimpleScalar [31], an architectural simulation infrastructure including a compiler based on GCC, an assembler based on GAS, and a set of instruction set simulators. The simulators are designed to simulate a MIPS derivative, and they provide different levels of detail, ranging from fast functional simulation to a detailed simulation which fully accounts for superscalarity, out-of-order issue, speculative execution and non-blocking caches.

With the rising importance of power estimation, instruction set simulators were augmented in such a way that they could provide power figures.

Ye et al. proposed SimplePower [32], a cycle-accurate simulator based on SimpleScalar, providing RT-level energy estimates which account for the consumption of the datapath, memory and on-chip bus. Brooks et al. proposed Wattch [33], also an extension of SimpleScalar, including high-level power models for caches, register files, branch predictors, reorder buffers, TLBs, functional units, clock buffers and many other components.

Šimunić et al. [34] proposed an extension of the instruction-level simulator provided by ARM with energy models. The result is an cycle-accurate simulator capable of providing energy estimates for a system including a StrongARM processor, a memory hierarchy including two levels of caches, the memory, the interconnect and the DC-DC converter. The simulator shows a good accuracy: estimation errors are within a 5% tolerance with respect to real measurements.

Sinha and Chandrakasan proposed JouleTrack [42], a web-based software profiling tool that enables an application developer to estimate the energy consumption of software on the StrongARM processor. The user is supposed to upload his source code on the web service, selects the processor operating conditions, memory maps and compilation options. The service provides a cycle accurate report of the program execution, along with energy statistics. It supports a hierarchical profiling technique that trades off simulation accuracy for simulation time.

Suresh et al. proposed a Frequent Loop Analysis Tool set (FLAT) [37]. FLAT can provide profiling statistics at the granularity of loop and function, and it is mainly designed to help hardware/software partitioning. The tool

set adopts two techniques. The first technique employs a compiler which is a modified version of GCC, and which allows to insert basic-block based instrumentation, with a "compilation-based technique", discussed below. The second technique employs a conventional instruction set simulator to find the execution counts of the loops. The approach is valuable and effectively addresses the needs for which it was designed. Nevertheless, it is poorly applicable for the needs described in Section 1.1. In fact, the first technique is not at the source level and not fine-detailed; the second technique is not source-level, not fine-detailed, and not fast.

In general, instruction set simulation does not deal effectively with the needs discussed in Section 1.1. All the techniques proposed in this category are not at the source levels. Additionally, they exhibit different speed grades, depending on the level of abstraction of their power models, anyway all of them are too computationally demanding to satisfy the 'fast' requirement.

2.1.4 Binary instrumentation

This technique consists in rewriting the executable of a program with additional machine instructions, which allow to measure its desired properties. Binary instrumentation tools usually proceed by locating the executable code inside the text segment, disassembling the code, determining the basic blocks, inserting the desired measurement instructions at the beginning of each basic block, and repacking the final, instrumented executable file.

The technique presents a large number of technical difficulties, because no knowledge is available about the original structure of the program, and a large number of processor-specific, compiler-specific and operating-system-specific details must be taken into account. For example, the "code discovery" phase, which consists in finding executable instructions inside the text segment, may be especially cumbersome in those executable formats which interleave code with data and jump tables, in an order and in formats which depend from compiler to compiler. An incorrect or incomplete interpretation of these formats lead to broken instrumented executable. The disassembly phase could require significant effort in architectures with variable-length instructions (e.g. Intel x86), in order not to lose alignment. At run-time, a "module discovery" phase is often required, to identify and analyze the entire graph of dynamically loaded code, which may be data-dependent and may vary from execution to execution.

A number of tools have been proposed in literature which are based on binary instrumentation. The large variety of them is explained by the fact that these tools are architecture- and OS-dependent. Authors usually develop a solution for a given architecture and OS, and the effort involved in implementing a port of the same tool to a different architecture or OS is prohibitive.

Smith proposed Pixie [61], which allows to insert basic-block based instrumentation and obtain complete execution traces on MIPS architectures and Unix-like operating systems. Srivastava and Eustace proposed ATOM [62], a link-time program modification tool which allows to insert custom instrumentation and develop analysis tools for the DEC Alpha platform run-

ning the OSF/1 operating system. Larus and Schnarr proposed EEL [63], a C++ library designed for machine-independent executable editing on MIPS SPARC-based architectures and Unix-like operating systems. EEL internally constructs the control-flow graph of each routine, and it is able to add foreign code before or after almost every instruction. EEL is a generic building block to write applications like QPT [64], which perform binary instrumentation in a flexible way. Romer et al. proposed Etch [65], an instrumentation system based on binary rewriting for the program executable binaries in the Windows' PE format for Intel x86 architectures, inserting instrumentation code at each function and basic block.

Binary rewriting techniques exhibit a number of advantages:

- they do not require the source code, which is an advantage if the application under analysis has been provided by a third-party;
- they are independent from the programming language in which the code was written;
- they are fast; Uhlig and Mudge [41] in a comparative survey report a time slowdown in the range 10–60;
- since they operate directly on the final binary code, they account for any possible compiler optimization, therefore they may reach a high level of accuracy.

However, in the context of embedded software design, the first two of the above advantages are of little interest, because designers have full availability of the source code they are developing, and because quite it is infrequent to change the programming language once the development has started.

More important, these techniques show a critical drawback: they are not source-level techniques according to the definition in Section 1.1 (page 21). None of the tools allow to back-annotate the measured data on the original source code. Therefore, they do not address effectively the contemporary software design needs.

2.1.5 Compilation-based techniques

These techniques usually reach the same goals as binary instrumentation but, since they work in conjunction with a compiler and operate on some intermediate representation provided by it, they provide a higher degree of flexibility and architecture- and OS-independence.

For example, Larus and Ball propose Abstract Execution (AE)[66], a compiler which produces instrumented executable. AE is basically a modified version of GCC, and it relies on GCC's internal Register Transfer Language (RTL) representation. RTL is one of the intermediate representations used by GCC, and it consists of an assembly-like code, corresponding to a simple load/store architecture with infinite registers and an orthogonal instruction set. AE takes as an input the RTL representation of the program under analysis, it builds the control-flow graph (CFG) and analyzes it to determine the basic blocks and the points to instrument. Since AE is integrated in GCC, it

has complete visibility on the compilation output, and it can provide exact counts of the number of instructions per each type which have been executed on a given run.

Lajolo et al. [67] propose a similar compilation-based co-simulation technique. The authors have modified the GCC compiler providing a special back-end, which regenerates an assembly-level C code, which includes instrumentation required to perform a cycle-accurate simulation. By actually compiling the code, the technique accounts exactly for the effects of compilation optimizations, but it is specific to the GCC compiler and, again, a way to redistribute energy to source-level elements is not provided.

Thanks to the inclusion of GCC, a compiler which is available on many architectures and operating systems, these approaches do not suffer from the architecture, OS and executable format issues discussed in the previous section. Nevertheless, these approaches show the same critical drawback discussed in the previous section: they are not source-level techniques. The highest level of abstraction which they can provide is the RTL level, which is much closer to the assembly level than to the source level.

Additionally, Lajolo's tool does not perform basic block instrumentation, thus also suffering from the same problems of large overhead which I have discussed for instruction-set simulation tools. Lajolo's tool is valuable under many respects: for example, generating a co-simulator within his framework seems to involve significantly less effort than writing an architectural simulator. Nevertheless, this approach does not meet the 'fast' requirement presented in Section 1.1 (page 21).

2.1.6 gprof: Program counter sampling

GNU gprof is a popular profiling tool, to be used in conjunction with the GNU GCC compiler. GCC can be instructed to add extra code during compilation, which periodically samples the program counter in order to determine which is the currently active C function. Programs compiled within this framework write sampled execution times to a dump files. gprof is able to process these files, and determine the time spent in each function. This technique is fast, dynamic and even source-level. In fact, statistics are generated at the level of functions, and functions are entities at the source level. Unfortunately, the technique does not meet one of the requirements I set: the fine detail. It is not fine-detailed because it provides statistics on a per-function basis, and it does not allow to obtain profiles of code inside functions.

Simunic et al. [43] try to take advantage of the source level of abstraction provided by gprof by coupling it to an instruction set simulator. Obviously, this approach is able to determine the execution time of C functions, but not of finer-grained elements. This is good enough for comparing different algorithms, but not helpful for guiding precise optimization. As already motivated in the overview, in the vast majority of embedded applications, most of the time is spent inside loops entirely contained in one function, and the approach cannot resolve at that granularity.

Note that forcing the detail of the approach is not possible: moving loops or even individual statements to separate functions leads to meaningless re-

sults, because the overhead due to the added function calls would perturb significantly the estimates.

Additionally, note that gprof must be either executed on the target platform or in an emulated version of it, which includes a full-featured Unix-like operating system. Many embedded systems may not include an OS or may not have the necessary hardware requirements to support the memory footprint of an OS and the associated runtime framework needed to run gprof.

2.1.7 Source code instrumentation

Source code instrumentation analyze the input source code provided by the user, and they generate a modified version of the same source code, in which instrumentation code is added.

De Rose and Reed presented SvPablo [69], a multi-language and architecture-independent performance analysis system. SvPablo provides a framework for instrumenting application source code and browsing dynamic performance data. It supports interactive instrumentation of source codes for 4 different languages. It relies on hardware counters which are read at specific points, which may be inserted by the user at specific points, the instrumentable constructs. It has two major drawbacks:

- the instrumentable points are only function calls and outer loops, therefore it is not fine-grained;
- the instrumentation is interactive, therefore it requires multiple user's intervention. The user is supposed to mark the instrumentable points in which he is interested, and to run the tool. Then, on the basis of the knowledge acquired in this step, to refine his chosen points by repeat the process again and again.

Templer and Jeffery proposed a Configurable C Instrumentation tool (CCI) [68], which provides automatic instrumentation for ANSI C code. Unfortunately, the tool has no notion of basic block instrumentation, therefore it suffers from code explosion (and associated slow-down). This limits seriously its practical usability, unless for interactive use. Basically, the user must iteratively specify his investigation points, as in the previous case.

Bormans et al. proposed Atomium [136], an analysis tool which addresses the memory-related aspects of system design, by applying the Data Transfer and Storage Exploration (DTSE) methodology. This tool, now commercially available as PowerEscape [137], employs an analysis approach which is similar to ours, operating at the behavioral level of an application, expressed in C. Unfortunately, it is meant only for memory access analysis and does not account for operations and control, which are also our main focus. Therefore, it cannot be employed to perform any computational analysis. Additionally, data transfer and storage analysis performed is performed for structs and arrays only, therefore code containing dynamic structures should be rewritten.

Ravasi [47, 48] proposed Software Instrumentation Tool (SIT), a source-level technique designed for C code. SIT accepts C code as input, promotes

it to C++, and provides overloaded instrumented operators and data structures. Then, it compiles the instrumented code with the regular GCC compiler.

The approach is clever: it relies on GCC to insert instrumentation code transparently, thus avoiding the difficulties of parsing the C language. SIT is very valuable to compare different implementations of an algorithm not only from the point of view of the computation, but also for the memory-oriented aspects, because the overloaded data structures may simulate any form of memory hierarchy. From this point of view, SIT is probably the best tool in its category. Nevertheless, in our context, the tool suffers from two drawbacks:

- it provides estimates at the level of C function, therefore it is not fine-grained;
- it causes instrumentation code to be executed at every execution of an operation; this is significantly inefficient, and puts the tool at the same speed grade as instruction-set simulation.

The approach I propose in this thesis can be considered a different variant of the ones presented in this category. When compared with the techniques presented in this section, my technique shows the following advantages:

- it provides estimates at the level of any C syntax entity, therefore it has the finest possible grain;
- it inserts minimal instrumentation code (less than one probe per basic block), therefore it is efficient and fast.

2.1.8 Black-box techniques

The work group of Raghunathan and Jha proposed [50] a fast and high-level software energy estimation method, founded on characterization-based macro-modeling. This method associates to each function an energy macro-model which only requires black-box parameters to provide an energy estimate. During simulation, macro-models can be used instead of detailed models, resulting in orders of magnitude simulation speedup. The construction of these models is done either on the basis of a complexity analysis or profiling. The first technique is employed for those functions whose algorithmic complexity can be easily expressed in terms of some parameters (e.g. data-intensive functions). The second technique can be applied always, even to control-intensive functions with data-dependent execution flow; with this technique, internal profiling statistics are used as parameters for the energy macro-model. The same work group extended [51] the above technique in such a way that characterization-based high-level software macro-modeling is performed automatically. The extension automates the steps of parameter identification, data collection through detailed simulation, macro-model template selection, and fitting.

Brandolese et al. [59] have also presented an automated methodology to extract black-box execution time and energy consumption models. Their

methodology is also black-box during characterization, i.e. no internal detail about the characterized function is needed. The methodology is especially suited to provide high-level, statistically-accurate models of library functions, operating system calls and third-party modules, for which the source code could not be available. The methodology allows to reach a good accuracy thanks to its ability to model semantical properties of input data, such as their data structure and size.

Unfortunately, these techniques do not satisfy the requirements set in Section 1.1. They are fast, and they satisfy the 'dynamic' requirement to some extent, but they are not for sure source-level and fine-detailed.

2.1.9 Memory-oriented techniques

The energy consumption associated to accessing the memory hierarchy is becoming nowadays a significant portion of the overall energy consumed by the system. Additionally, since the speed gap between the processor and the memory is also increasing, memory plays a significant role also in determining the execution time of an application.

The approach presented in this thesis does not tackle the memory hierarchy, since it is quite difficult to tackle effectively both the problems of estimating the execution costs of the processor core and of the memory hierarchy at the same time. Additionally, good theory and tools are already available for the purpose.

The most common method for evaluating memory-system designs before implementing them, or for evaluating software implementations over a given platform, is trace-driven memory simulation. It consists in writing a simulation model which mimics the memory design, and then applying to it the sequence of memory accesses caused by a real algorithm which would run over the platform under study. The sequence of accessed memory locations is called an *address trace*, and the method is called *trace-driven memory simulation*. The task consists of three phases: trace collection, trace reduction and trace processing. All the phases involve significant difficulties because of the trace sizes (which may easily reach gigabytes) and of the complexity of the system, which may include multiple processes and processors, operating systems, and dynamically linked code. Researchers have been working extensively to tackle these difficulties at different levels, ranging from the hardware level up to the operating system, creating a range of more than 50 simulation tools. Uhlig and Mudge [41] offer a survey of these techniques, categorizing them on the bases of the approaches they employ, and comparing them on the basis of accuracy, speed, space, portability, cost and ease of use. The simulation models on which the above techniques rely simulate the policies which the different caches may implement, but they must also account for the time and energy required by each operation inside a given cache level. One of the models which is commonly used to address this need is CACTI, proposed by Wilton and Jouppi [40]. It is an analytical model which allows to derive the access and cycle times for on-chip caches.

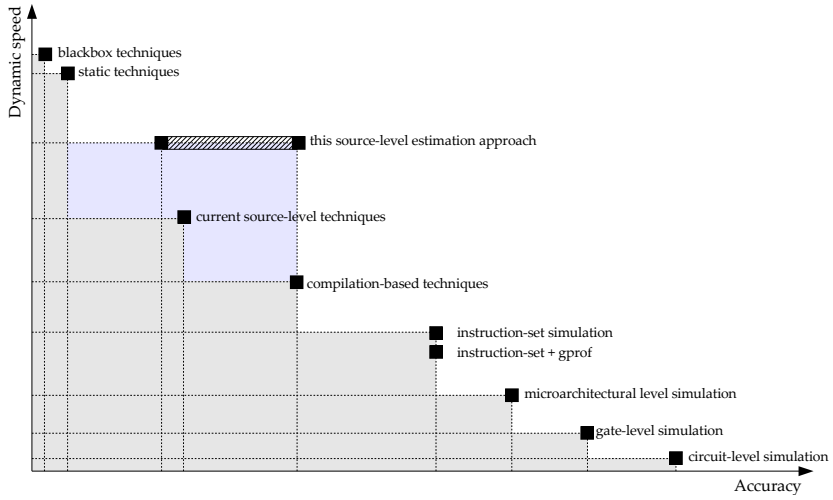


Figure 2.1: How the approach I propose compares with the currently available techniques in terms of speed and accuracy.

2.1.10 Conclusions

In the previous paragraphs I have presented an overview on the estimation methods for embedded software. In this section I give a comparison of these techniques against the technique advocated by this thesis.

All the above techniques can be compared with each other under many respects. Two of the most important ones are the accuracy and the dynamic speed. By accuracy I mean a small estimation error in determining the cost of execution caused by the processor core (this comparison does not consider the accuracy in estimating the cost due to the memory hierarchy, which some of these techniques tackle). By dynamic speed I mean a small simulation time. Here, I do not consider the static speed, which is the time required to analyze the application before simulating it.

The above techniques compare with each other, in terms of dynamic speed and accuracy, as qualitatively represented in Figure 2.1. Both axes are to be intended empirical (intuitively, it is helpful to consider the logarithmic). In this plot, the extremes are represented by circuit-level simulation (which is the most accurate technique, and the slowest one) and by static/black-box techniques, which may have less accuracy but have a virtually zero simulation time. Note that there is not clear superiority of one technique with respect to some other: all the techniques are Pareto-optimal, and each one of them could be the most appropriate, given a specific set of circumstances.

The many possible technique instances generated by the fundamental approach presented in this thesis are represented as a hatched bar, rather than a single point. In fact, depending on the level of detail in the modeling choices made, a specific instance may be accurate as much as compilation-

based techniques or less. All the instances show the same dynamic speed, since they all use generalized basic-block instrumentation. They may differ in terms of static speed, as already shown in Figure 1.3 (page 32).

2.2 Source-level optimization exploration techniques

The previous section was mainly concerned about estimation problems. This section, instead, identifies the fundamental problems which must be addressed to build a source-code transformation exploration flow, and surveying how the different approaches in literature have tackled these problems.

All the flows which allow to explore source transformations must deal with the following four problems (either manually or with some degree of automation):

(A) analyze the source code and expose optimization target points;

(B) given a target point, estimate the influence of transformations on it;

(C) select and target transformations;

(D) apply transformations on the code.

Additionally, if they want to realize a *short-loop transformation exploration* methodology, which I have advocated in the Introduction, they must solve the above problem in a more specific way, by providing respectively the following four components:

(A) a technique to analyze and profile programs at the source level, detailed enough to expose optimization target points;

(B) metrics to estimate the impact of transformation on a given target point;

(C) an automatic transformation steering engine, working on data provided by (A) and (B);

(D) a tool to automatically apply transformations selected by (C).

Briefly said, many approaches have been presented in literature to deal with problems (A), (B) and (D), while problem (C) has been so far neglected or treated marginally. I propose a technique to address problem (C), and a flow which consists of the following components:

- as component (A), the source-level estimation technique which is the main subject of this thesis;
- as component (B), metrics on the influence of source code transformations proposed by Brandolese [126];

- as component (C), a novel automatic transformation steering engine, working on data provided by (A) and (B), which is described extensively in Section 3.4 (page 90)

- finally, I have performed task (D) manually, but automatic tools for the purpose exist [127] and there are no conceptual difficulties in integrating them with my flow. The task only involves a major implementation effort.

To solve problem (A), many traditional approaches employed an instruction set simulators (ISS) [32, 33, 35, 42]. In these approaches, developers run the ISS to obtain estimates of the time and energy consumed by the program. Then, they iteratively search manually for the computationally-demanding portions of the program, and they manually optimize them. After each optimization, they verify its impact by running the ISS again. In these approaches, the ISS addresses problem (A), while (B),(C) and (D) are addressed manually, on the basis of the expertise and skill of the developer. However, these approaches are impractical for many reasons:

- ISSs only provide assembly-level estimates, which are difficult to relate to source-level entities; therefore, exposing the computational *kernels* is often far from trivial;

- ISSs are very slow; they usually run thousands of times slower than the target system, and this makes it impossible to apply them on many modern and future applications, such as video decoding softwares;

- the entire approach is manual: the developer selects manually the transformations and where to target them. There is no guarantee of optimality, and the approach is impractical for large-scale applications.

Simunic et al. have used their coupled ISS with gprof to guide the optimization of an MP3 player [43], by applying source-level optimizations at three different layers, as follows:

- at the algorithmic level, they have employed more efficient, published alternative algorithms;

- at the data level, they have reimplemented the computational kernels using fixed-point arithmetics in place of floating point;

- at the instruction-flow level, they have applied the usual loop transformation and they rewrite matrix multiplications by inlining assembly code which exploit a specific multiply-and-accumulate instruction (MLAL) available in the ARM core they have relied on.

Unfortunately, there is weak improvement for problem (A), but steps (B), (C) and (D) are clearly impossible to automate or to generalize to different applications. At the algorithmic level, the developer must drop the current implementation, search the literature for alternatives, choose one and implement it. It is clearly impossible to automate this step. At the data level, the

use of fixed-point arithmetics is also not generalizable, because it does not preserve the exact semantics of the original program, and it could endanger the signal-to-noise ratio. At the instruction-flow level, the use of inline assembly is clearly not a source-level transformation.

The static estimation approaches discussed in the previous section [22, 23, 45], may be a possible solution for problem (A). Although these techniques are faster than ISS and proved to be useful for verifying real-time constraints, they cannot assist optimization of modern applications. In fact, these approaches cannot deal with dynamic behaviors, while modern applications (e.g. object-based synthetic video decoders, wireless ad-hoc networks, distributed gaming, ...) exhibit more and more dynamism.

Also black-box techniques [51, 59] have been employed to characterize the performance of embedded software. These techniques are very fast, but due to their inability to consider the impact of any elements inside of the code, they cannot guide optimization.

The Software Instrumentation Tool proposed by Ravasi [48] is a valuable alternative for component (A), but it lacks fine detail, as motivated above.

I believe that problem (A) is currently fully solved only by a source-level approach I am proposing. This approach is able to provide time and energy estimates for any flexibly-defined cluster of operations in the program. With this tool, designers do not need to define their interested clusters in advance, and not even to re-run the tool for different cluster choices. With this information, designers understand precisely which portions of the source code cause the major consumptions, and apply optimizations there. The methodology exhibits simulation times 10,000 shorter than a reference ISS, and shows good estimation accuracy.

Henkel and Li [54, 55] propose a comprehensive environment for design-space exploration and optimization of low-power embedded systems. Their environment comprises both hardware- and software-oriented techniques. The software part is consistent with the scheme in Figure 1.8(a). It comprises an ISS to solve problem (A), a metric based on ratio between the estimated energy saving (EES) and the code size increase (CSI) to address problem (B), an iterative algorithm based on the procedure call graph to address (C), and SUIF [127] as component (D). Unfortunately, this estimation approach is trace-based, therefore it exhibits the same inacceptable simulation times for modern applications already discussed.

Agosta et al. [138] have recently proposed a technique able to co-explore source code transformations an architectural parameters in an automated way, also comprising a software flow. Unfortunately, it is again as in Figure 1.8(a). It employs an ISS in conjunction with the SUIF [127] front-end for (A), and the SUIF back-end for (D). Authors provide their own system-level metrics as component (B), and an original heuristic transformation space exploration module, based on Pareto Simulated Annealing for (C). The approach is sound and well automated, and the results are convincing. Unfortunately, also in this approach, module (B) relies on instruction-level profiles provided by module (A), which makes the method too slow.

Franke et al. [130] have proposed a probabilistic feed-back driven technique to select source-level transformations, with very good results. They

also employ a tool flow very similar to Figure 1.8(a), therefore suffering long simulation times.

Triantafyllis et al. [139] have proposed an interesting approach for compiler optimization-space exploration. The approach augments a regular compiler with an iterative compilation strategy. However, this approach is designed to optimally guide the choice of parameters for regular compiler optimizations, not to guide source-level transformations.

Zitzler et al. [140] proposed an optimization space exploration strategy for data-dominated code expressed in the form of a synchronous data-flow graph, based on genetic algorithms. However, this approach is also not oriented to source-level optimizations.

Chapter 3

An instance of the technique

In the overview I introduced a fundamental approach to construct source-level, fine-detailed, dynamic and fast techniques, to estimate the execution time and the energy consumed by embedded software when running on given input data. This fundamental approach may derive many techniques, depending on which modeling choices are taken. This chapter presents one instance of these possible techniques, which is especially suited for single-issue processors.

This chapter discusses the models on which this technique relies, the basic steps which compose it, the activities which it involves and the people which are supposed to carry them out.

3.1 Abstracting the reality, modeling the abstraction

The objective of this technique is to estimate the execution costs of source code. The real path which leads from a source code to its physical execution costs, in terms on energy and time, involves a large number of steps. Many of these steps show great complexity, as I detail below. For reasons of effort, performance and generality, it is not convenient to account completely and exactly for this complexity. Instead, a convenient trade-off between effort, performance, generality and accuracy should be chosen.

For example, a compiler step should be modeled exactly when the effort is acceptable, it leads to a model which is general enough to be easily tuned on other compilers, it has acceptable static time overhead and it leads to a significant increase in accuracy. Otherwise, it is more convenient to model its behavior in a statistically-consistent way. It may be a reasonable idea to even completely neglect the effect of a compiler step, if the incremental added accuracy is negligible.

In the light of the above considerations, this section discusses how to obtain a model of the real compilation and execution processes, which is presents an acceptable trade-off among the accuracy it can provide, how easily can be generalized, the design effort it involves and the static time overhead it causes.

I derive my technique by *abstracting* the real flow and obtaining an abstract flow. Then I *model* the abstract flow and obtain an model flow. The two terms "abstracting" and "modeling" are used in this context in the following sense:

- *abstracting* an object means replacing it with a simpler object, which is functionally equivalent and numerically consistent (either exactly or statistically) with the original one, but it exhibits a behavior which is easier to model. An abstraction is an operation which reduces complexity. For example, abstracting a real pipelined processor may mean replacing it with a simpler non-pipelined processor (where instruction latencies may be fractional numbers), such that the real and the abstract processors exhibit statistically-similar latencies measured in clock cycles and throughput. The original object and its abstraction are functionally equivalent: in the example, both execute instructions, both exhibit a deterministic behavior. Abstracting some stage of compiler, for example the target code generator, may mean replacing it with another stage where some optimization steps are not executed or where the same operations (e.g. register allocation) are performed in a trivial way, thanks to the simplification hypotheses introduced during the abstractions of the objects located in the previous stages of the flow. As just suggested, introducing an abstraction at a given stage induces a simplification not only on that stage, but also on its output information and, consequently, on the following stages in the flow. This means that not only behaviors but also information flows are subject to abstraction, and that abstraction of objects which belong to the same information-flow chain must be consistent with each other;

- *modeling* an object which takes some input and yields some output means replacing it with a function which estimates the cost (in a general sense) of the output from the cost of the input. Modeling is an operation which replaces a behavior with a function which accounts for the cost of that behavior. The model of an object is *not* functionally equivalent to the original object: a compiler (real or abstract) produces object code, whereas the model of a compiler yields an estimate of the single-execution cost of the instructions in an object code; a machine (real or abstract) executes object code, whereas the model of a machine yields an estimate of the cost of executing some code.

Here I examine a real compilation and execution flow, and I prepare an abstract version, providing abstract versions for most of its components. Finally, I prepare a model flow, which allows the determination of the cost of executing a given source code. Figure 3.1 (page 66) represents the details of

a real compilation and execution flow, of its abstraction and of its model I chose in this thesis.

I *do not* implement the abstract flow, since it would require the complete definition (down to the bit level) of the abstract machine, its assembly language, and the development of an associated compiler and simulator or (even worse) silicon device. Such an implementation is not useful for the purposes of this research. Instead, I *do* implement the model flow. The model flow *is* the fundamental goal of this thesis.

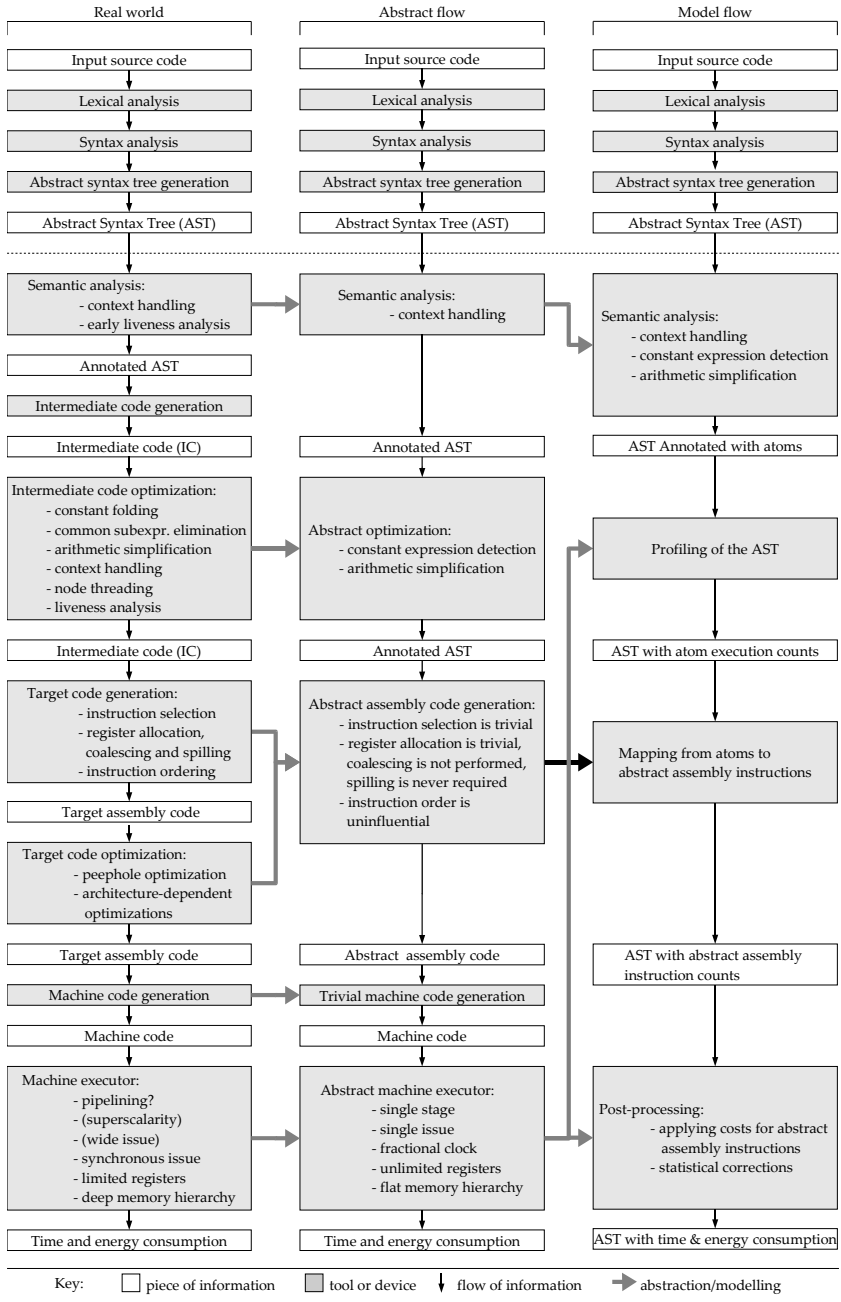


Figure 3.1: Real compilation and execution are so complex that it is not convenient to model all this complexity. Therefore I perform *abstraction* and *modeling* (see Section 3.1). This figure represents the original flow, the abstract flow and the model flow.

3.2 From reality to the abstract flow

3.2.1 Architecture abstraction

3.2.1.1 Instruction-set architecture

A modern architecture comprises a CPU including features like pipelining, dynamic scheduling, superscalarity, multiple issue and branch prediction, and a memory hierarchy which may span multiple on-chip and off-chip cache levels. Modeling the memory hierarchy is out of the goal of this work, and the instance of the technique presented in this chapter is especially suited for single-issue architectures.

I adopt an abstract architecture which is a single issue, single stage ideal executor. In this section I motivate how it is general enough to approximate the execution behavior of more complex, single-issue architectures.

A real machine, depending on its class (stack, accumulator, register-memory, load/store), has a corresponding instruction set composed of instructions with up to zero, one, two or three explicit operands, and instructions may feature many addressing modes, some of which involving remarkably complex address calculation steps. The registers available to the user may be a small or a larger number, and they may be primarily composed by general purpose or dedicated registers. Depending on the degree of orthogonality of the machine, a more or less complex set of constraints rule whether a given register may be used as an operand for a given instruction for a given data type in a given addressing mode.

On the other hand, my abstract architecture has an infinite number of user-available registers, which are all general-purpose. The instruction set is extremely simple, and instructions may have up to three explicit operands. It is a load/store architecture with three-operand instructions. ALU operations may operate on a set of three arbitrary registers with no limitations. The addressing modes are straightforward: being a pure load/store architecture, only two instructions ('mvld' and 'mvst' to load and store, respectively) can access data according to a register-indirect addressing mode. All the other instructions must use a register-direct addressing mode to specify the target, and either an immediate or register-direct addressing mode for the operands.

The latency of a given instruction in a real CPU may vary significantly depending on the context: branch mis-predictions, instruction cache misses and conflicts all cause stalls. Excluding the effects due to the memory hierarchy, which are not dealt with here, the same instruction, located at the same code segment address may exhibit significantly different latencies in two different executions, and this latency is influenced by the state of the system, in the most general sense. In my architecture, a given instruction always requires the same amount of time to execute.

In a real processor, the average amount of current absorbed by the portions of the processor involved in processing a given instruction over its latency time is affected by the actual instruction encoding and operand data, which cause a larger or lesser amount of switching activity on the busses. In my abstract architecture, the same instruction always absorbs the same amount of current during the time in which it is executed.

Modern computers are synchronous, since the processing which an instruction undergoes in a given functional unit always require an integer number of cycles; the results of a stage in the pipeline are not used by the subsequent stage even if they could become available before. On the other hand, the abstract CPU presented here has an exotic property, which is desirable for modeling reasons: it is asynchronous. It has a clock with the same period of the real CPU which it abstracts, but the latency of any instruction may be equal to a fractional number of clock cycles. It issues and executes the next instruction as soon as the current one is completed, without regard to the clock.

3.2.1.2 Memory

Modern architectures have registers, multiple level of caches and a RAM. The latency associated to the retrieval of an operand may be zero clock cycles (if the data is already in a register), one or a more clock cycles if the data is available at some level in the cache (depending on the design of the cache level and its operating frequency), or a large number of clock cycles if the requested data needs to be fetched from the RAM. The architecture has a flat memory hierarchy: all the addressable space is in the registers. Nevertheless, there is a fundamental difference on accessing operands depending whether their address is known by name or by address. Scalar variables and temporaries are allocated to fixed registers at compile time, and instructions can operate on them directly (register-direct addressing mode). On the other hand, for cells of an array or pointed by a pointer, their address must be determined at run-time before accessing them. Therefore, the compiler must emit code to load them into temporary register whose name is known (a 'mvld' instruction, which uses a register-indirect addressing mode), and to operate on that register by name.

As previously motivated, in this technique, I do not estimate the cost of the memory hierarchy. In the abstract architecture, the memory is byte-addressable. Registers are 4 byte wide and also memory words. Reading the ith register is the same operation as reading the memory at addresses $i \cdot 4, i \cdot 4 + 1, i \cdot 4 + 2, i \cdot 4 + 3$. I assume that misaligned memory operations are allowed, but they operate on bytes or double-bytes only, such that a single instruction cannot access two memory words.

3.2.2 Compiler abstraction

Although compilers can be significantly different one from another from a structural point of view, depending on the technique they employ to solve problems, they all have to tackle the same set of problems, such as instruction selection and register allocation. In Figure 3.1 (page 66), I tried to capture the general structure of a generic compiler which accepts a imperative language such as C and generates binary machine code. The compiler itself is, under the column named 'Real world', the collection of the gray boxes starting from 'Lexical analysis' to 'Machine code generation'.

3.2.2.1 Front-end

There is a portion of the structure of every compiler which has the responsibility of reading the input source code and determining its structure, according with the rules which define the grammar of the language. This part is usually called the *front-end* of the compiler. Despite the variety of compiler-construction tools and implementation techniques, the front-ends for the same language exhibit similar features. They employ a lexical analyzer, a syntax analyzer and most likely a set of semantic actions which generate an abstract syntax tree (AST).

A lexical analyzer (or *lexer*) reads the uniform flow of characters composing the input source code and it identifies tokens, such as operators, keywords, identifiers, string literals, and number literals. Lexical analyzers are either written by hand or generated automatically; in the second case, they are usually generated from a description of the lexical elements of the language. These descriptions are usually given in terms of regular expressions, therefore lexical analyzers are basically regular expression recognition tools.

A syntax analyzer (or *parser*) reads the flow of tokens identified by the lexer and determines which rules in the grammar allow to generate them. Parser are almost always generated automatically from the description of a grammar, given as a set of BNF (Backus-Naur form) rules, e.g. the ones in Section 4.5 (page 231). Informally, BNF rules tell how to rewrite a given symbol into a collection of new symbols, some of which are terminal symbols (i.e., they cannot be rewritten according to a rule), while the others, nonterminal symbols, can be further rewritten. When a symbol is rewritten into others, it is common to represent it in the form of a node connected with the children symbols in a tree diagram. If a program is correct, there must be a way to apply rewriting rules in the grammar, starting from the *start symbol* until the exact sequence of terminal symbols which constitute it is obtained. The tree which expresses how the start symbol is rewritten into the input source code is the parse tree, or concrete syntax tree (CST) associated to the program. Parsers do not usually output CSTs; instead they run designer-specified semantic actions, which specify what to do when a given construct is encountered. Simplistic compilers have semantic actions which directly emit assembly code or machine code. A common choice in real-world compilers is to design the parser's semantic actions in order to generate the abstract syntax tree (AST) of the input program. Then, the AST is used by further processed by the remaining part of the compiler; to learn more on concrete and abstract syntax trees, see Section 4.1.2 (page 102).

I have freedom to introduce as many simplifying abstractions on the architecture and on the back-end of the compiler as desired, but not in the front-end. Since the overall goal of this thesis is to determine the execution cost for syntax elements, I must comply to the standards as the front-end is concerned.

In fact, given a correct program and a grammar of its language, the CST of the program is automatically determined and, in principle, all the parsers in all the compilers for the C language should determine the same CST (apart from non-standard language extensions). And also the conversion from the

CST to the corresponding AST presents little degree of freedom. This is the reason why all the flows (real, abstract, model) represented in Figure 3.1 (page 66) are identical in their upper parts. They share the same, complete, real-world front-end.

3.2.2.2 Semantic analysis

3.2.2.2.1 Context handling

Generally speaking, in this phase, compilers perform two tasks: they check the context conditions required by the language specification, and they collect information for the semantic processing. Specifically speaking about the C language, in the first of the above tasks the compiler verifies the semantic conditions described in the standard [80]. The vast majority of these conditions regards the type system of the C language: e.g., assignments must occur only between expressions of compatible type, the types of the actual parameters of a function must be compatible with the types of the respective formal parameters, also the type of the operand in a '**return**' statement must be compatible with the declared return type of the enclosing function, arithmetic operators cannot operate on non scalar types (e.g., it is not possible to use the '+' operator between structs), the bitwise operators allow only integral operands, name of fields in structs and unions must be among the declared ones.

Other additional context checks, not immediately related to the type system, which are usually done in this phase include: the absence of nested functions, the presence of the target labels of '**goto**' statements in the scope where the statements appear, the absence of '**continue**' statements outside loops, the absence of '**break**' statements outside loops or '**switch**' statements, the compliance with the standard of the signature of the 'main()' function.

As justified before, the details of the abstract flow do not need to be specified, so I do not further examine the details of the context handling in the semantic phase of the abstract flow. On the other hand, I must detail the semantic analysis step of the model flow. I feel free to assume the input source code as correct. I assume that the designer submits his source code to my model flow only when it has already successfully passed the checks of a regular ANSI-compliant compiler. This assumption relieves me from implementing a large quantity of sanity checks. Still, my syntax analyzer needs to integrate a complete type system, since the cost of syntax elements depends, in general from the type of operands, as motivated in Section 4.2.2 (page 112).

3.2.2.2.2 Early liveness analysis

Informally said, liveness analysis determines, for each variable at a given point in a program, whether it holds a value which will be useful later. The results of this analysis are useful at two levels:

- they can detect unused variables and the use of uninitialized variables;
- liveness analysis guides register allocation;

in this paragraph I only consider the first one and I call it *early liveness analysis*.

The C language specifications do not forbid the declaration of unused variables or the use of uninitialized variables, nevertheless both cases are undesirable. The presence of unused variables unnecessarily increases the memory footprint of the final executable, whereas the use of uninitialized variables usually denotes some serious programmer's mistakes and may have catastrophic consequences at runtime.

To avoid this, modern compilers perform an early liveness analysis as a part of context handling, and generate warnings when one of the above cases is detected. In case of unused variables, the compiler may avoid to allocate those variables at all.

The abstract flow does not perform any semantic analysis on unused variables or uses of uninitialized variables, therefore its semantic analysis phase only comprises context handling.

3.2.2.3 Intermediate code generation

The concepts expressed by the nodes in the AST are at a high level of abstraction, and highly language-dependent. A compiler designer may try to naively translate an AST into the corresponding assembly code, by determining the assembly translation per each AST node. This is quite a simple tasks for arithmetic operators, but for some other types of nodes this is far from trivial. For example, consider a node which represents a dot operator in a C program ('.', the member-of- struct or -union resolution operator), and try to determine its assembly translation. See Section 4.4.2 (page 158) for a complete discussion on the topic.

To overcome the above difficulties, most compilers rewrite the AST into another tree, whose nodes express conceptually simpler operations, which are more easily representable at an assembly-language level but still architecture-independent, in order to preserve portability. This tree is called the *intermediate representation* (IR), and its linearized textual representation is often called the *intermediate code* (IC). Usually, IR nodes belong to a narrow set of categories: expressions with no more than two operands, function headings and jumps (including calls and returns). Due to the fact that high-level concepts may correspond to multiple lower-level nodes, during the translation from an AST to the corresponding IR, the number of nodes may grow significantly

I have selected simplification hypotheses on purpose, and I have no retargetability issues (I just need to compile for a single target architecture: the abstract machine). Thanks to this, I can provide an abstract translation model which describes how to generate abstract assembly code directly from the AST, without the need for an intermediate representation. This abstract translation process is described in Section 4.3 (page 117).

3.2.2.4 Intermediate code optimization

Most compilers apply a number of architecture-independent optimizations on the intermediate code.

3.2.2.4.1 Constant folding

Compilers recognize constant-value expressions and determine their value at static time, avoiding the generation of their corresponding assembly code. I abstract and model accurately this behavior by recognizing constant-value subexpressions during the semantic analysis phase, as I detail in Section 4.4.3 (page 169).

I do not support the detection of constant expression which involve function calls. Although sin(c), cos(c), pow(c), log(c), sqrt(c), ... are all constant expressions if 'c' is a constant expression, most compilers do not perform constant folding in these cases, sometimes in order to preserve the results produced by the hardware floating-point rounding. Therefore, I also do not support this cases, not even when integral values only are involved.

I also do not perform constant propagation [93] which is more complex and requires a static single assignment (SSA) representation or equivalent.

3.2.2.4.2 Arithmetic simplification

Arithmetic simplification is a task in which arithmetic operations are either removed or replaced with less expensive counterparts.

Arithmetic operations which can be removed completely are called *null sequences*. Null sequences are addition with zero, multiplication by one, bitwise shift with zero offset, logical 'and' with true, logical 'or' with false and similar. Although these expressions are not very likely to appear in human-written code, such expressions can quite easily appear from the expansion of macros. Modern compilers usually include the ability to detect and remove null sequences involving integral expressions and, sometimes, also with floating-point expressions.

In other cases, it is convenient to replace an arithmetic operation with another one which is guaranteed to be less expensive on every possible architecture for which the compiler could be targeted. This optimization is called *strength reduction* [96]. Integer multiplications and divisions by powers of two ('a * 2', 'a * 4', 'a * 8', ... and 'a / 2', 'a / 4', 'a / 8', ...) are replaced with left and right shifts respectively ('a << 1', 'a << 2', 'a << 3', ... and 'a >> 1', 'a >> 2', 'a >> 3', ...). Other strength reductions which replace power expressions such as 'a squared', 'a cube',... with 'a * a' or 'a * a * a' are not possible for AST or IR trees derived from the C language, which lack a power operator such as operator '**' in Fortran.

In the model flow, I consider integral-arithmetic null sequences, and a limited number of cases of strength reduction.

3.2.2.4.3 Liveness analysis

Though performing liveness analysis as a part of context handling is optional, all real C compilers must perform liveness analysis at least to guide

the register allocation phase, which takes place in the 'Target code generation' step.

One of the common techniques to perform liveness analysis is by solving data-flow equations on the control-flow graph [98], and a common technique to generate a control-flow graph from an AST is by node threading. Node threading is a simple algorithm which consists in a tree visit of the AST, with simple pointer-updating operations involved.

Since the abstract version of the register allocation phase is trivial (thanks to the unlimited number of registers in the abstract architecture), it needs no liveness information. As a consequence, the abstract and model flows do not perform any liveness analysis at all.

3.2.2.4.4 Other optimization steps

Many other optimization are possible at this level if also the data-flow between statements is analyzed (e.g., constant propagation [93], dead code elimination [94], inter-procedural constant propagation [95]), which lead to smaller and faster programs, but due to their implementation complexity we choose not to comprise their behavior in the abstract flow.

Another common task which real compilers perform during intermediate code optimization is the elimination of common subexpression [97]. I choose not to model this optimization exactly because of the effort it involves. Nevertheless, my model flow can be easily extended to do account for this effect, either statistically or exactly.

3.2.2.5 Target code generation

During target code generation, a real compiler translates the intermediate representation into a program, written in the target machine's assembly code, which realizes the same semantics. This is usually done by replacing each node in the IR tree with segments of target assembly code which realize the semantics of the replaced node. Then, the IR tree is linearized on the basis of the constraints imposed by the data dependencies among nodes and by the control flow. There are three main issues involved in this phase: instruction selection, register allocation and instruction ordering. There is no general technique which addresses effectively all the problems at the same time; on the other side, most available techniques tackle more than one problem at the same time, nevertheless I present them separately.

3.2.2.5.1 Instruction selection

During the instruction selection phase, the compiler tries to determine a good set of assembly instructions which translates the IR tree. It is convenient to describe the problem of the instruction selection as the determination a good tiling of the IR tree, where each instruction in the target instruction set corresponds to a given tile. This ingenious formulation of the problem is by Cattell [87], who also presented an automatic instruction selection algorithm and built a *code generator generator* to automatically produce an instruction selection function from the description of an instruction set. I said *a good* tiling in place of *the optimum* tiling because in the second form

the problem is NP-complete, and real compilers employ a variety of different approaches to solve the problem, among which the maximal munch [87], bottom-up rewriting systems [88] and dynamic programming [89].

The formulation of the instruction selection problem in terms of tree tiling is quite appropriate for the instruction sets of RISC machines, which usually have many general purpose registers, three-operand instructions, a load/store architecture (i.e., simple addressing modes) and exactly one effect per instruction. When these hypotheses are verified, the matching between instruction and tiles is tight and convenient. When compilation for older, RISC machines is concerned, things complicate dramatically. CISC machines have few, dedicated registers, complex addressing modes, two-operand instructions and instructions with multiple side-effects. Modeling their instructions with tiles is tricky. These architectures demand special care and ad-hoc solutions during the design of the instruction selection and register allocation routines. The resulting routines are even more complex to describe and model than the approaches described above. For these architectures, also a revolutionary approach, called super-compilation [90, 91] was recently proposed. Compilers which employ this technique use precalculated optimal code sequences, determined by exhaustive searching the solution space for common, critical IR node patterns.

The complexity and variety of the numerous above approaches is out of the reach of any possible high-level model. For the abstract flow, I have no other choice than providing an abstract version of the instruction selection phase which disregards the approach-dependent low-level details, still preserving the consistency with abstract architecture and assembly language, and allowing the construction of a corresponding high-level model. The abstract translation phase, architecture and assembly needs to be designed in such a way that the corresponding high-level model can be made statistically consistent with the behavior of the real code generator via appropriate statistical tuning.

The solution I propose relies on the the minimalistic choices made for the abstract instruction set. I have deliberately designed the abstract instruction set (especially its cardinality, architecture class, and addressing modes) in such a way that instruction selection is trivial. I formally describe abstract instruction selection as part of the abstract translation model, in the form of a grammar attribute, as defined in Section 4.3 (page 117), and I model it with translation rules from atoms to abstract instructions.

3.2.2.5.2 Register allocation

The intermediate code generated by the algorithms described in the previous sections (3.2.2.3 and 3.2.2.4) may use an infinite number of registers to hold the temporary values, and may use as many 'move' instructions as desired. On the other hand, all real CPUs have a limited number of registers, and 'move' instructions are expensive. The register allocator is responsible for assigning the temporaries to a small number of machine registers, and assigning the source and destination of move instructions to the same register wherever possible, in such a way that the corresponding 'move' instructions can be avoided.

A common approach to perform this task is by constructing an *interference graph* and *coloring* it [92]. The interference graph is a graph whose nodes are temporaries and variables, and whose edges connect couple of nodes which cannot be assigned to the same register. The existence of an edge between two nodes may be a consequence of the liveness analysis (i.e. the two nodes are live at the same time) or may represent structural constraint (no instructions exist in the current architecture which can produce the result in the register already assigned to some node). The task of register allocation coincides now with coloring the interference graph, where no pair of nodes connected by an edge may be assigned the same color. Each color represents a different register. If the target machine has n registers, then any valid n-coloring of the graph is a valid register assignment for the given interference graph. When such a n-coloring does not exist, some temporaries must be selected for being kept in memory instead of a register: this operation is called spilling. When there is no edge between the source and destination node of a move instruction, the two nodes can be coalesced and the corresponding move instruction is eliminated, as anticipated above.

Register allocation is trivial task in the abstract compiler, thanks to the hypothesis that the abstract architecture has an unlimited number of registers (see Section 3.2.1 (page 67)). The generation of unique register names and their use as temporary names during the abstract translation is a valid register allocation. The cost of accessing spilled temporaries must be accounted for statistically in the model.

3.2.2.6 Target code optimization

In real compilers, this phase dominates the others in complexity, and it is the one in which they exhibit the greatest diversity, especially for pipelined, VLIW, superscalar and dynamic scheduling architectures, not counting coarse-grained parallel architectures. It may include some peephole optimization, loop unrolling [99, 118], code motion [100], hoisting, induction variable elimination, code replication [101], loop pipelining (with the Aiken-Nicolau scheduling [102], or with Rau's iterative modulo scheduling [103, 104], trace scheduling [105]. It may also include static branch prediction [106, 71, 109], possibly profile-guided [108, 110], predicated execution [111, 112, 113], profile-guided code positioning [121] and function-inlining, either statically determined [123, 122, 124] or profile-guided [125].

The memory hierarchy-oriented optimizations may include data prefetching [114], scalar replacement [120], loop interchange [115], loop tiling [116], loop blocking [117] and loop fusion.

My fundamental approach, described in Section 1.4 (page 29), does not set any constraints on how to handle this complexity. Each of the above optimization steps in the modeled compiler can be neglected, approximated statistically or modeled exactly. The decision I take for the instance of the technique I describe in this chapter, is to model all these phases statistically. The reasons are the following:

- most of the above phases are not even applicable or not beneficial for single-issue executors, to which the technique instance described here

is targeted;

- many of the above phases have been investigated in research and exhaustively described and evaluated in literature, still they have not been implemented in available compilers;
- the amount of knowledge on the actual modeled compiler required to model these steps correctly is so high, that writing such an accurate model does not involve significant less effort than writing the real compiler.

The above decisions depend on the target and context set for this instance of the technique. In different contexts, other decisions may be more appropriate: for a VLIW architecture with a trace-based compiler, integrating a the compiler in the source-level estimation technique could be the only viable approach to ensure acceptable accuracy.

3.2.2.7 Machine code generation

During this phase, the assembly instructions generated in the previous step are written in an encoded format, such that they can be fetched and executed sequentially by the processor. This task is either carried out internally by the compiler, or by invoking an external assembler. The task it involves encoding the instructions as required by the machine specifications, including the opcode and operand fields, calculating jump target locations, and updating the offsets in those jump instructions which jump there.

This is a trivial operation in the abstract assembly language, and we do not need to specify the details of the abstract machine code. The only significant detail in the model associated to machine instruction decoding is the tuning of the space cost contribution for each abstract assembly instruction.

3.3 The model flow

3.3.1 Analytical cost model

In this section I define the cost model on which the technique instance presented in this thesis relies.

Given a program execution run (the program and its input data), let C_e, C_t and C_s be the energy consumed by that execution (μJ), the execution time (clock cycles), and the memory space occupied by the binary code of the program (bytes) respectively[1].

Let N be the number of nodes in the AST of the given source code. Instrumenting all the nodes is unnecessary and highly inefficient, since for each group of nodes executed the same number of times, only one of them needs be instrumented; see Section 3.3.2.2 (page 82) for details. Thus, a representative set of P nodes is chosen for instrumentation: the pivot nodes, or *pivots*. $\mathbf{R}$ is the a $(P \times N)$ matrix, called the *representation matrix* and defined as:

$$\mathbf{R} = \begin{bmatrix} R_{1,1} & R_{1,2} & ... & R_{1,N} \\ R_{2,1} & R_{2,2} & ... & R_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ R_{P,1} & R_{P,2} & ... & R_{P,N} \end{bmatrix}$$

The matrix indicates which nodes are represented by each pivot. An element $R_{i,j}$ is equal to 1 when the j-th node s_j is represented by the i-th pivot, otherwise $R_{i,j} = 0$. Each column contains exactly one element with value 1, since each node has exactly one pivot.

Once the instrumented program is run, the profile for the pivots is available:

$$p = [p_1\ p_2\ ...\ p_P].$$

The element p_i is the execution count of the i-th pivot. The generic element p_i is the execution count of the i-th pivot. From p and $\mathbf{R}$, it is possible to calculate n, the vector of execution counts of all symbols as:

$$n = p \cdot \mathbf{R} = [n_1\ n_2\ ...\ n_N]$$

where n_i is the execution count of the i-th source symbol.

As anticipated, each node i has a single-execution cost c_i, expressed as a linear combination of atoms, as expressed below:

$$c_i = M_{i,1} \cdot \mathrm{Atom}_1 + M_{i,2} \cdot \mathrm{Atom}_2 + ... = \sum_{j=1}^{A} M_{i,j} \cdot \mathrm{Atom}_j$$

The association of costs to nodes in terms of atoms is described by the *mapping matrix* $\mathbf{M}$ defined as:

[1] The model presented here also allows to determine the executable object size, although this feature is not discussed further in the rest of the thesis.

$$\mathbf{M} = \begin{bmatrix} M_{1,1} & M_{1,2} & ... & M_{1,A} \\ M_{2,1} & M_{2,2} & ... & M_{2,A} \\ \vdots & \vdots & \ddots & \vdots \\ M_{N,1} & M_{N,2} & ... & M_{N,A} \end{bmatrix}$$

where A is the number of atoms. The mapping matrix is a $(N \times A)$ matrix indicating how many occurrences of each kind of atom are associated to each and every symbol.

The cost of each atom is, in turn, composed of the costs of the kernel instructions that the atom translates to. K is the cardinality of the classes of the kernel instruction set. Then we define the *translation matrix* $\mathbf{T}$ as the following $(A \times K)$ matrix:

$$\mathbf{T} = \begin{bmatrix} T_{1,1} & T_{1,2} & ... & T_{1,K} \\ T_{2,1} & T_{2,2} & ... & T_{2,K} \\ \vdots & \vdots & \ddots & \vdots \\ T_{A,1} & T_{A,2} & ... & T_{A,K} \end{bmatrix}$$

This matrix accounts for the behavior of a the modeled compiler. Each compiler leads to a different $\mathbf{T}$.

To reduce the number of measurements which needs to be performed on the target architecture, I partition the entire abstract instruction set in a small set of classes, assuming that instructions in the same class have similar cost in time, energy and space.

The i-th row defines the cost of the atom i-th atom in terms of classes of abstract instructions.

$$\text{Atom}_i = \sum_{j=1}^{K} T_{i,j} \cdot k_j$$

Statistical measurements performed on the target architecture provide energy, time and space costs for each of the classes of kernel instruction:

$$\begin{array}{rcl} k_e & = & \begin{bmatrix} k_{e1} & k_{e2} & ... & k_{eK} \end{bmatrix} \\ k_t & = & \begin{bmatrix} k_{t1} & k_{t2} & ... & k_{tK} \end{bmatrix} \\ k_s & = & \begin{bmatrix} k_{s1} & k_{s2} & ... & k_{sK} \end{bmatrix} \end{array}$$

Based on the above definitions, it is possible to express the estimates of energy, time and space costs of a given program execution in a very compact way:

$$C_e = p \cdot \mathbf{R} \cdot \mathbf{M} \cdot \mathbf{T} \cdot k_e^T \quad (3.1)$$

$$C_t = p \cdot \mathbf{R} \cdot \mathbf{M} \cdot \mathbf{T} \cdot k_t^T \quad (3.2)$$

$$C_s = \mathbf{1}_A \cdot \mathbf{M} \cdot \mathbf{T} \cdot k_s^T \quad (3.3)$$

(where $\mathbf{1}_A$ is a vector of all '1' elements).

Since

$$C_i = n_i \cdot c_i,$$

	Type	Size	Available after	Comment
C_e	scalar		post-processing	final output
C_t	scalar		post-processing	final output
C_s	scalar		post-processing	*idem* (instr./profiling not required)
N	scalar		parsing	depends on source code
n	vector	$1 \times N$	parsing	depends on source code
P	scalar		instrumenting	*idem* and on pivot choice strategy
p	vector	$1 \times P$	profiling	depends on source code, pivot choice strategy and input data
A	scalar		method	fixed in this technique instance
K	scalar		method	fixed in this technique instance
k_e	vector	$1 \times K$	modeling	statistical model of the architecture
k_t	vector	$1 \times K$	modeling	statistical model of the architecture
k_s	vector	$1 \times K$	modeling	*idem* and of the compiler
$\mathbf{R}$	matrix	$P \times N$	instrumenting	describes the choice of pivot symbols
$\mathbf{M}$	matrix	$N \times A$	method	fixed in this technique instance
$\mathbf{T}$	matrix	$A \times K$	tuning	statistical model of the compiler

Table 3.1: Summary of the symbols introduced.

and the execution counts are contained in vector

$$n = p \cdot \mathbf{R},$$

while single-execution costs are in vector

$$\mathbf{M} \cdot \mathbf{T} \cdot k_v^T$$

(where $v \in \{e, t, s\}$), then the cost of each node in terms of energy, time and space can be expressed as:

$$C_{i,e} = (p \cdot \mathbf{R})_i \cdot (\mathbf{M} \cdot \mathbf{T} \cdot k_e^T)_i \tag{3.4}$$

$$C_{i,t} = (p \cdot \mathbf{R})_i \cdot (\mathbf{M} \cdot \mathbf{T} \cdot k_t^T)_i \tag{3.5}$$

$$C_{i,s} = (\mathbf{M} \cdot \mathbf{T} \cdot k_s^T)_i \tag{3.6}$$

The above equations, along with the definition given above of the matrices involved, summarize the proposed technique.

Table 3.1 lists the symbols introduced in this section.

3.3.2 Model application

The technique I propose is composed of the following 6 steps, as depicted in Figure 3.2:

1. Analyze:
   in this step, the tool determines the AST of the input source code by parsing it; then it annotates the AST with costs expressed as atoms. It selects a minimal subset of the nodes (*pivots*) for profiling, such that the execution counts of all the other nodes can be derived from the pivots' execution counts.

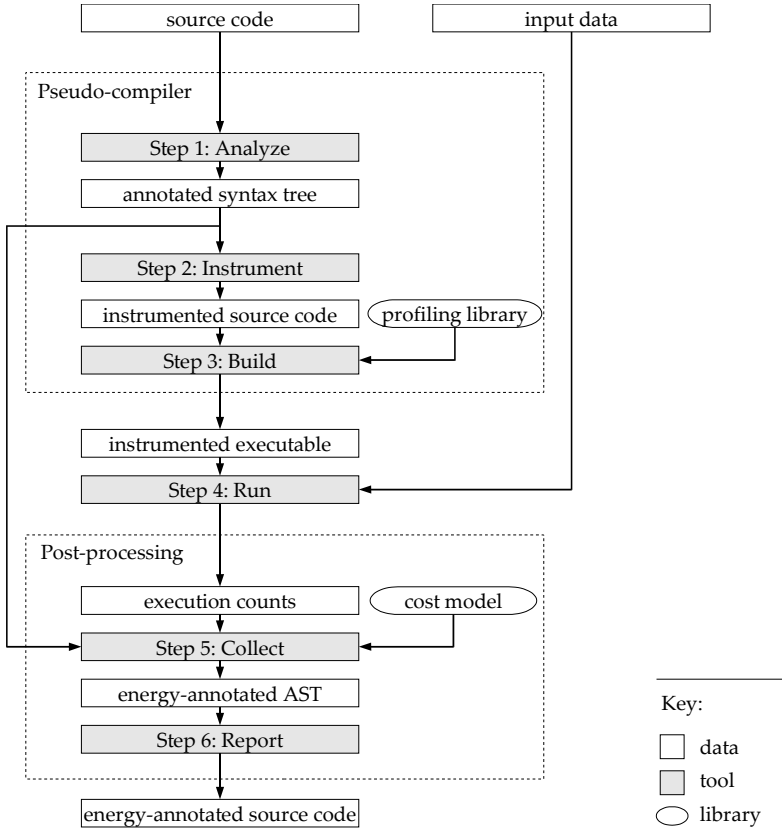


Figure 3.2: The steps which compose the estimation flow I propose.

2. Instrument:
   in this step the tool generates an instrumented version of the code, inserting a profiling call per each pivot.

3. Build:
   the tool employs a regular compiler (e.g. GNU GCC) to build the instrumented source code, also linking it against the profiling library, which contains the implementation of the profiling function. The result is an executable program which is functionally equivalent to the original, non-instrumented application, except that it dumps execution counts (profiles) at the end of its execution.

4. Run:
   the tool runs the instrumented executable over real input data. After the execution, the execution counts per each pivot are available.

5. Collect:
   the tool translates the single-execution costs from atoms, to abstract assembly instructions, and then to physical quantities (time and energy). It determines the profiles of all the nodes starting from the pivot profiles. Finally, it determines the total cost for each AST node;

6. Report:
   the tool generates a report, in the form of a browsable, annotated version of the original source code, indicating how much energy and time are spent by each syntax element.

The next paragraphs discuss the theory and tools on which the above steps are based.

3.3.2.1 Step 1: Analyze

The first task involved in Step 1 is parsing a generic source code written in the C programming language. As I have motivated in Section 3.2.2.1 (page 69), the model flow needs a complete, 'real world' front-end for the C language. The necessary concepts for the construction of a parser for an artificial language belong to a well-consolidated portion of the theory of languages, and a number of good books [76, 77, 78, 79] exist, which cover the topic exhaustively. This thesis offers no original contribution in this field.

As far as the choice of which specific 'flavor' of the programming language to parse is concerned, for sake of compliance with the standards, I choose to strictly adhere to the specifications of the C language, as published in the standard named ISO/IEC 9899:1990 [80] and in the two subsequent associated Technical Corrigenda. The standard is also available is an annotated version [81] which covers a number of pitfalls and technicalities for which the naked standard is not an appropriate learning aid.

To design the C parser I have employed the well known scanner generator 'flex' [145] and parser generator 'bison' [143, 144] from the Free Software Foundation.

The actual bison specification of the syntax of the C language I employed in the design of my parser is by Jeff Lee, republished by Jutta Degener [147].

A parser prepared according to the above grammar operates on the basis of its syntax rules, recognizing a complete concrete syntax tree. Concrete syntax trees are impractical to handle (for reasons which we justify in Section 4.1.2 (page 102)), and it is common practice to transform them into abstract syntax trees. The transformation presents no conceptual difficulties but its actual realization presents some degrees of freedom, therefore, if not specified in detail, it is ambiguous. For this reason, I choose to specify it completely in Section 4.1.2 (page 102).

Once the AST of the input source code is available, the technique associates to each syntax entity (i.e. each node in the AST) its single-execution cost, expressed as a linear combination of atoms (source-level, architecture-independent cost contributions). This task relies on an original analysis of the execution cost of source-level entities. This technique, and the theory which provides its foundations are the subject of Chapter 4. The technique is

expressed in the form of an attribute grammar, whose final goal is the determination of attribute c for each of the node in the AST. Attribute c of a node accounts for the consumption caused by a single execution of that node.

The last task involved in step is the selection of the *pivot* nodes, i.e., a minimal set of nodes which will be subject to instrumentation via profiling calls. The purpose of each profiling call is to determine the exact execution count of the node to which it was attached. The set of pivot nodes must be selected carefully, in such a way that it is minimal (to minimize the overhead of the instrumentation code) but sufficient to derive the execution counts for all the other nodes in the AST. The choice of an appropriate pivot selection strategy is the topic of the next section.

3.3.2.2 Step 2: Instrument

In Step 2, the technique rewrites an instrumented version of the original input source code, with instrumentation code added to the pivot nodes. Though not being technically trivial, the inner workings of this tool show little scientific novelty.

Profiling nodes via source instrumentation requires a sufficient set of transparent probe-inserting transformations, able to reach any desired type of node. The set of transformation chosen is illustrated in Table 3.2. Expressions are included in parenthesized *comma expressions*, statements are wrapped in *compound statements*. The same happens to function bodies, with additional code added at every return point in order to allow call stack simulation, and per-invocation cost calculation. I have called these transformations 'transparent' because they do not modify the input-output behavior of the original program.

Instrumenting all the nodes is unnecessary, highly inefficient and, in some cases, syntactically impossible. Instead, probes should be inserted in the minimum number of points sufficient to determine the execution count of every node in the program: therefore, an efficient instrumentation strategy is required.

Given a set of nodes $A = \{a_1, a_2, ...\}$, I define an *instrumentation* I over A as a partition of A where, for each subset A_i, one node p_i is chosen as pivot. More formally,

$$I = \{(p_1, A_1), (p_2, A_2), \ldots, (p_n, A_n)\}$$

such that $\{A_1, A_2, \ldots, A_n\}$ is a partition of A, and $\forall i \in \{1, 2, \ldots, n\}\ p_i \in A_i$. An instrumentation is *safe* when all nodes inside a given A_i execute the same number of times. The *trivial* instrumentation is the one in which every node is pivot:

$$I_T = \{(a_1, \{a_1\}), (a_2, \{a_2\}), ..., (a_n, \{a_n\})\}.$$

A convenient formal way to define a *instrumentation strategy* is as an equivalence relation $\sim$ over elements of A. In fact, any equivalence relation over A induces a partition $A/\sim = \{A_1, A_2, \ldots\}$. Then, any arbitrary choice of the pivot for each A_i is irrelevant for correctness or for efficiency.

Therefore, determining a good instrumentation strategy is determining a good node equivalence relation $\sim$. Very efficient $\sim$-relations induce large

Syntax element	Instrumentation technique
expression	(profile_function(#node_id), expression)
statement	{ profile_function(#node_id); statement }

type function_name(args)
{
body
}
	type function_name(args)
	{ profile_function(#node_id1); {
	body
	} profile_function(#node_id2); }

Table 3.2: Instrumentation syntactic techniques

equivalence classes, but could be complex to evaluate. For example, in the following excerpt:

```
/* code section 1 */
a = 3;
if (a % 2) {
  /* code section 2 */
}
/* code section 3 */
```

nodes in code section 1,2 and 3 should be all equivalent since the `if` condition is always true[2]. However, the determination of cases like above require a complex symbolic evaluation engine, which is beyond cost-effectiveness in the context of this work.

A traditional solution is basic block instrumentation[71]: given nodes a and b, it considers $a \sim b$ iff a, b belong to the same basic block. This is a good starting point, but at source-level further optimizations are possible. Figure 3.3 illustrates such a case: sections 1 and 4 are evidently always executed together, but assembly-level tools cannot recognize this and would insert 4 probes, when only 3 are actually needed. The gain is evident in case of 'if' chains.

The technique I propose is called *generalized basic block instrumentation*. It is based on the following definition of $\sim$ −relation:

> *Given two nodes a, b, I say that a is* equivalent *to b (or $a \sim b$) iff a and b belong to the same function,* and *it can be stated that a and b are always executed the same number of times without examining any conditional expressions.*

Hence, a generalized basic block is a maximum set of nodes that are executed the same number of times, and it is not required to examine conditional expressions to say that. The above definition can be proved to be safe; additionally, it is simple to check at parse time, and still very efficient.

My implemented instrumentation engine finds optimal instrumentations according to the above definition, and statistics over real code show that it generates instrumentations with equivalence classes containing on the average 100 nodes. This means that on the average, only 1 probe is executed every 100 nodes, resulting in run-times only 2 times slower than the original

[2]provided that `a` is non-`volatile` and no `goto`s involve sections 1–3.

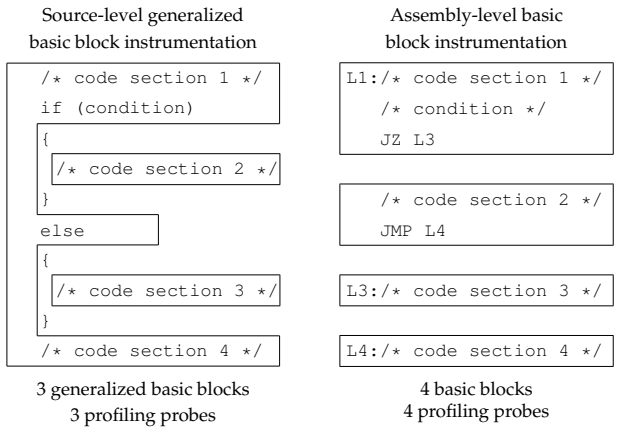


Figure 3.3: A sample section of code where generalized basic block instrumentation is more efficient than basic block instrumentation.

code and approximately 11,000 times faster than basic-block instrumentation.

3.3.2.3 Step 3: Build

In Steps 3 I employ a regular C compiler to compile the instrumented source code. In the implementation I relied upon GNU GCC. This choice is completely uninfluential on the generality and applicability of the technique. The use of the estimation flow with a different compiler would require no or minor modifications in the implementation details of the tools. I chose GCC because of its availability on a large number of platforms and operating systems.

In the figure above, I depicted Steps 1–4 grouped together into a *pseudo-compiler* box. This is to symbolize that the tasks involved in those steps are carried out, in my implementation, by a a single script which externally behaves like a regular compiler. Therefore they are executed in a fully automated way, and they are transparent to the programmer. I discuss the remaining details on the topic in .

The flow intervenes in the regular task of linking the compiled objects, in order to add a library which contains the run-time support for instrumentation. This library provides the implementation for all the function calls which the instrumentation engine may add, either in order to profile the program syntax elements or upon user request.

3.3.2.4 Step 4: Run

In this step the user runs the instrumented executable. It is the user responsibility to provide realistic input data to the instrumented program. The in-

strumented program will behave in a functionally equivalent manner to the original program, with the only difference of the overhead required to maintain profiles and other user-requested measurements. At program termination, profiles will be written to disk.

3.3.2.5 Step 5: Collect

During Step 5, a tool processes profiles and determines the desired statistics, according to the formal cost presented.

3.3.2.6 Step 6: Report

During Step 6, a tool represents the statistics collected and calculated during the previous phase, and it relates these cost data back to the original entities in the program which caused them. The output is a browsable copy of the original code, which each significant entity has been annotated with cost statistics.

3.3.3 People and activities

The activities that a generic actor can undertake with respect to the technique are three: founding the technique, targeting the technique and using the technique. The actors of these three activities are respectively its authors, the target specialists and the users.

3.3.3.1 Founding the technique

This activity consists in the construction of the very technique: laying its theoretical foundations, deriving the cost association rules which are its core, verifying its correctness, devising efficient profiling strategies, designing automated tools which allow the application of the technique, carrying out experiments to measure its estimation accuracy. All the tasks involved in this thesis were carried out by its authors. The outcome of the tasks is the technique itself. All these tasks need to be completed once for all: their results are architecture- and compiler-independent.

There is a significant portion of the ideas and concepts conceived in this activity, which the actors of the other activities need not to be aware of (e.g., the abstract assembly translation model described in Section 4.3 (page 117)). These ideas and concepts serve as the foundations of the technique, but do not appear in its outcome.

Activity	creating the technique
Actor	the author of this thesis
Input	the C language standard
Outcome	the map from source code to atoms, the map from atoms to kernel instructions,
Reiteration	just do once

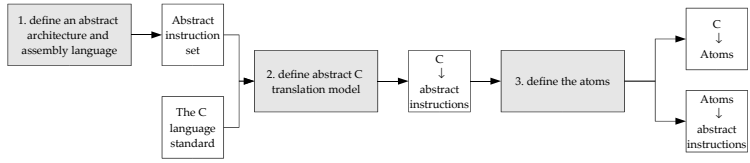


Figure 3.4: Tasks and artifacts involved in activity 1: “founding the technique”.

3.3.3.2 Targeting the technique

This activity consists in preparing the architecture-dependent details of the technique for use with a given target platform. This activity requires the

target platform, either in the form of an instrumented board or of an energy-accurate simulator. The expected actor of this activity is the *target specialist*, i.e., someone who has an extensive knowledge of the target platform. The target specialist does not need to know how the high-level details of the technique, i.e. the abstract translation model and how atoms are associated to nodes. On the other hand, he needs to know how to model the specific target architecture and its instruction set in terms of abstract instructions.

Activity	preparing the technique for use with a given architecture
Actor	the target specialist (e.g. silicon vendor)
Input	the kernel instruction set, the target platform
Outcome	the cost of each kernel instruction, the correction coefficients
Reiteration	do once for each target platform

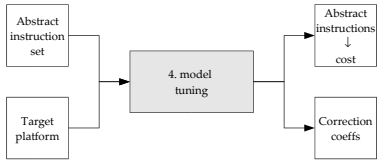


Figure 3.5: Tasks and artifacts involved in activity 2: "targeting the technique".

3.3.3.3 Using the technique

This activity consists in using the technique. The actor is the final user. The final user does not need any knowledge on the target or on the methodology.

Activity	using the technique
Actor	the final user
Input	the application source code the input data
Outcome	the estimated execution cost
Reiteration	do once for each application and set of input data

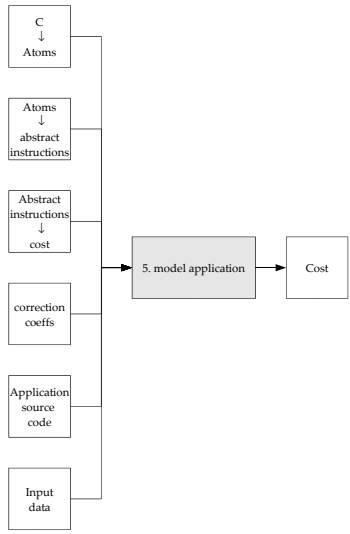


Figure 3.6: Tasks and artifacts involved in activity 3: “using the technique”.

3.3.4 Overall scheme

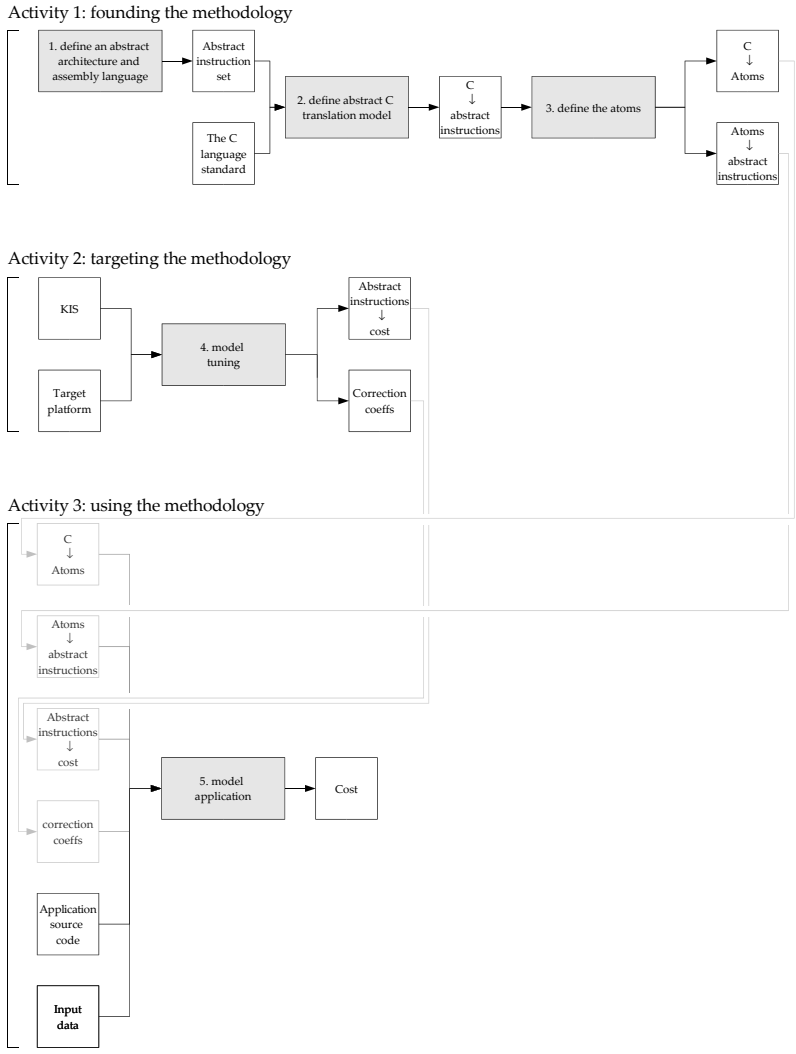


Figure 3.7: The tasks and artifacts involved in all the activities related with the technique.

3.4 The optimization flow

Once the source-level estimation flow has completed his execution, thus providing estimates for execution time and energy absorption of each syntax element, the optimization phase can be initiated.

In this phase, the amount of source code to be considered in the next tasks is reduced by selecting only the most critical portions. In this context, a portion of code is either a whole function or a set of lines. To perform such a selection I sort energy costs of functions and code lines in decreasing order, and select the first N_f functions and N_l lines such that their contribution (calculated independently for functions and for code lines) accounts for a user-defined fraction of the overall energy. It is clear, in fact, that optimizing the most critical portions of the code will potentially result in the highest benefit. A portion of code is uniquely identified by the triple:

$$\langle file, function, [l_s, l_e] \rangle$$

where $[l_s, l_e]$ identifies a range of consecutive lines. I call such a triple a *scope*. Scopes can be incomplete triples, i.e. one or more of their elements can be replaced by the symbol $*$, having the meaning of a wild-card. For example, the scope $\langle file, function, * \rangle$ indicates a whole function, while $\mathcal{A} = \langle *, *, * \rangle$ indicates the whole source code of the application being considered.

The result of critical portions selection is represented by two lists of scopes: the list of functions:

$$\mathcal{C}_f = \{ \langle file, function, * \rangle_i \} \quad 0 \leq i < N_f$$

and the list of sections:

$$\mathcal{C}_l = \{ \langle file, function, [l_s, l_e] \rangle_j \} \quad 0 \leq j < N_l$$

The optimization engine is based on a set of basic fuzzy rules characterized by:

- an input scope;
- a fitness function, measuring a specific feature on the code and returning a fitness value $f_k \in [0; 1]$;
- a threshold, indicating the minimum value of the fitness for which the rule is triggered;
- a guideline, indicating the optimization to perform.

plus a number of compound rules obtained by combination of the above basic rules. The final output of the optimization engine is the list of rules that fired, the corresponding scopes and a score between 0 and 1 indicating how much the suggested optimization is expected to be beneficial.

Now, I now describe in detail the steps composing the transformation steering flow. Briefly said, the transformation steering flow adds two more steps to Figure 3.2, Step 7 and Step 8. Step 7 selects the most critical sections

of code, and Steps 8 selects which transformations to apply on it. Step 7 is a fairly simple task, which mainly groups statistics by their relevant syntax nodes and sorts them by decreasing impact. Step 8 is the very transformation steering operation, carried out by a network of fuzzy rules.

Steps 7 accepts as an input the statistics produced at Step 6. It detects and sorts the critical sections by decreasing execution time, selecting the top ones which cause collectively a user-selected portion (e.g. 90% or 95%) of the total execution time or energy. The other sections are neglected. Sections of code are denoted as *scopes*, as explained before.

At the end of Step 7, a lists of critical scopes is available. Step 8 will generate the optimization directives, determining which transformations are beneficial on these scopes. Step 8 is performed by a modified artificial neural network that I call *Network of Fuzzy Rules* (NFR). NFRs are very similar to traditional Artificial Neural Networks (ANN), except for several details explained below, which have been adapted for this application domain. An important difference is that, while in most ANNs weights and connections are estimated during an error-minimization learning process (as in, e.g. back-propagation), in NFRs connections and weights are individually and intentionally designed to model precise concepts, e.g. to detect the suitability of a scope to benefit from loop unrolling. Each NFR rule (the corresponding concept of a neuron) is an object as depicted in Figure 3.8.

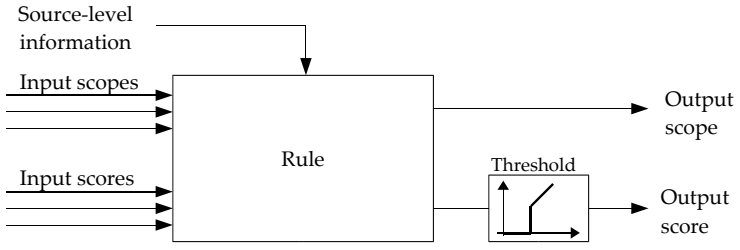


Figure 3.8: A NFR rule.

A NFR rule is a function (as complex as desired) taking as inputs one or more scopes and their associated scores, and returning exactly one scope and one score. A *score* is a value $\in [0,1]$ which express a fuzzy truth value, e.g. how much a given feature is present in the given input scope or how much a transformation is beneficial to the given input scope. Unlike neurons, each NFR rule has an extended set of inputs, which includes the complete data of the program under analysis: its syntax tree annotated with profiles and costs, its symbol tables, etc.. Like neurons, NFR rules have an activation function. I have employed the following simple activation function:

$$\text{output} = f(x) = \begin{cases} 0 & \text{when } x < \text{threshold} \\ x & \text{else} \end{cases}$$

i.e., each rule has a given threshold, and when the output is below threshold,

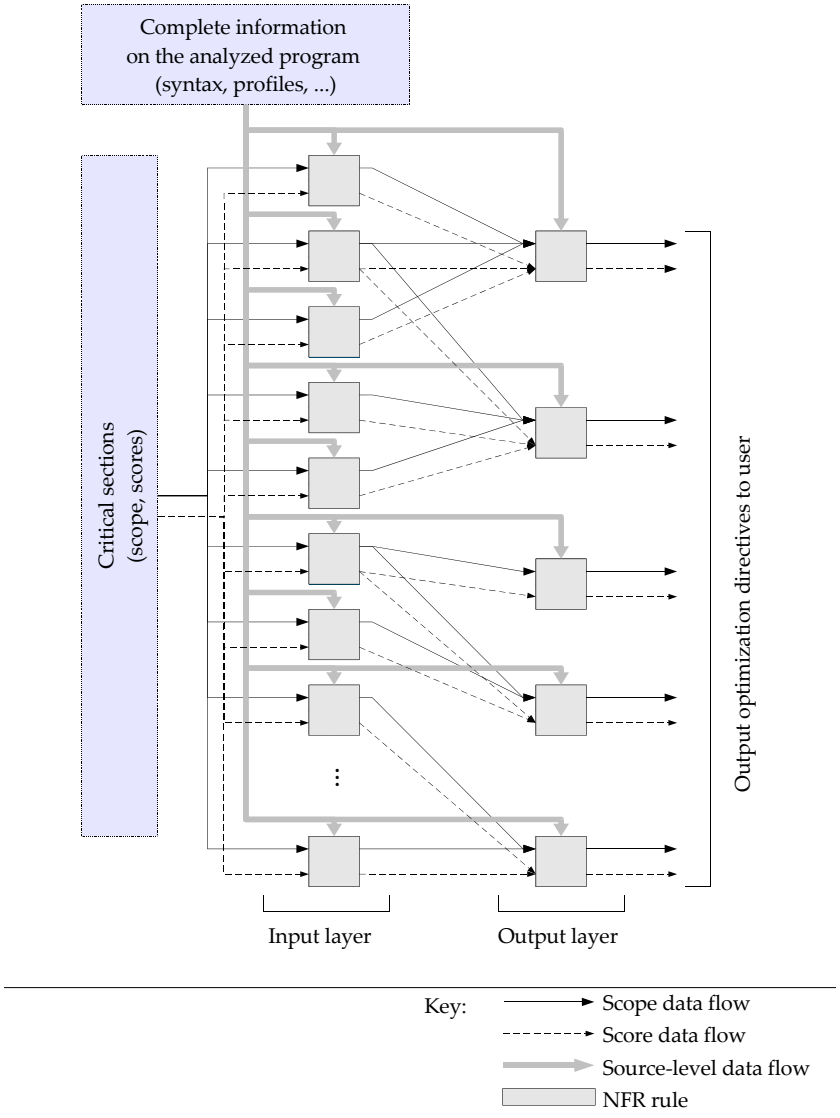


Figure 3.9: The structure of the Network of Fuzzy Rules (NFR) I employ for transformation steering.

zero is returned. When the output score of a rule is above the threshold, I say that the rule *fires*.

As ANNs are composed of multiple layers of neurons, NFRs can be composed of multiple layers of NFR rules. In the following paragraphs, I will employ the concepts of 'input layer', 'output layer', 'hidden layer', 'input

rule', 'output rule' by analogy with the corresponding concepts in ANNs. I skip the corresponding formal definitions because they are lengthy and trivial. In my experimental setup, I propose a NFR composed by two layers, an input and an output layer (see Figure 3.9).

Rules in the input layer have a single scope input, taken from the list of critical scope. They have also a single score input, which is the energy or time cost of that scope, normalized over $[0, 1]$. With each input rule I model a given feature of a section of code: e.g. "this section is a small bounded for loop" or "this section is a floating-point intensive computational kernel", or "this sections allocates small constant-sized chunks of memory" and its score represents the truth of this feature.

Each output rule corresponds to a given transformation. When an output rule fires, the corresponding transformation is beneficial over the given scope. The output score gives an empirical measure of how beneficial it is. The collection of all the rules which fired, together with the input scope which caused them to fire, are recorded to form the optimization directives returned to the user.

The output scope of input rules is a subset or equal to their input scope. The output scope of output rules is a set operation of the input scopes, depending on the case (usually an intersection).

To summarize, Step 8 proceeds as follows:

- each of the critical scopes determined in Step 7 is presented as a scope input to all the NFR rules in the input layers; and each of the associated energy/time weights of the above scopes (normalized to [0;1]) is presented as score input to all the NFR rules;
- the outputs of all the other NFR rules is calculated according to connections;
- if any of the output neuron has fired, it is added to the output optimization directives, together with its associated scope.

3.4.1 Modularity of the algorithm

A relevant advantage of the proposed algorithm is its modularity. NFR rules are objects with a well defined interface. They accept a set of scopes and associated scores as inputs, and they return one scope and its score. Plus, they can access as an input the entire results of the analysis of Steps 1–6, including syntax trees, atom- and abstract assembly characterizations, profiles and symbol tables, which all have a well defined format.

As a consequence, the approach allows easily the augmentation of the NFR with new rules, designed to apply new transformations. Each NFR rule can be developed separately by different entities, in different programming languages. In my implementation, NFR rules are separate programs, some implemented in C and some implemented in scripting languages (mainly Tcl and Python). As a consequence, NFR rules can be also shipped in binary format without disclosing intellectual properties.

At this time, I have modeled NFR rules for the following transformations:

LU loop unrolling (well known);

FI function inlining (well known);

FM function replacement with a macro (well known);

SE common subexpression elimination (well known);

SR strength reduction (well known);

CE type conversion elimination:
this transformation avoids a frequent type conversion (either implicit or explicit) by changing the declared type of one or more variables involved in the operation, for example promoting an integer variable to a floating point type;

SF standard library function call factorization:
this transformation is triggered by standard library function calls such as 'printf' or 'scanf' inside critical loops; it is often possible to unroll these loops n times and replace n calls with a single call. For example, calls like `printf("%i␣",array[i]);` can be factored into a single call `printf("%i␣%i␣...␣%i", array[i], array[i+1], ...);` ;

MF memory allocation factorization:
this transformation is triggered by memory-allocating function calls inside critical loops; it is often possible to unroll these loops n times and replace n calls with a single call, which allocates n times the memory of the original call;

AP argument passing via pointers;
this transformation is triggered when large object are passed as function arguments on the stack; after the transformation, the same objects are passed by pointer;

FS function specialization;
this transformation is triggered by constant arguments in function calls; the original function is replaced by a new version with less arguments, and with an internal algorithm which is replaced with a specialized version.

The above two-letter acronyms associated to each transformation will be used to describe the experimental results in Table 5.2.

As shown above, it is not only possible to model the traditional loop and function call transformations, but also novel transformations, at a higher level of abstraction, for example the ones involving the semantics of operating system calls and library functions. This is possible because NFR rules have complete access to the entire static and dynamic information available on the program.

3.4.2 Scalability of the algorithm

Given the growing size of several embedded applications (e.g. video players), the performance scalability of the entire flow with respect to size is of critical importance. I will show that this flow is composed by steps which scale well with the size of the application under analysis.

There are two important dimensions of size: a static dimension (how complex is the description of the program under analysis) and a dynamic dimension (how complex is the runtime behavior of the program). A rough estimator of the static dimension of a program is its size, measured in lines of code, or better in abstract syntax tree nodes. A rough estimator of the dynamic dimension of a program is its simulation time. A scalable technique must exhibit a time complexity which is not worse than polynomial in both dimensions.

I prove that all the steps comply with this requirement:

- Step 1 is based on a LALR(1) parser (polynomial over static size), with semantic actions that may involve searching or sorting symbol table (which are also polynomial over static size), and is therefore polynomial over static size.
- Step 2 is linear with the static size of the program, and independent from the dynamic size.
- Step 3 and 4 are performed by a regular compiler, whose complexity analysis is beyond reach. Anyway, compilers are designed to exhibit acceptable run times.
- The profiling operation between Steps 4 and 5 are linear with the dynamic size of the program, and independent from the static size.
- Steps 5 and 6 are linear with the static size of the program and independent from its dynamic size.
- Step 7 is a list sort, which can be performed in $O(n \log n)$ time where n indicates the static size of the program. It is independent from the dynamic size.
- Step 8 deserves some discussion, because it involves modules (NFR rules) which can be provided by third parties and whose complexity is not known. Step 7 involves a linear scan of all the scopes. The step is independent from the dynamic size, and has a complexity over the static size equal to $O(n \cdot Q)$ where Q is the complexity of the most complex NFR rule. Therefore, Step 8 is polynomial if all the NFR rules are polynomial.

3.4.3 Current limitations

Employing a structure like the NFR network leads to the advantages in modularity and scalability which I have discussed above. Now, I discuss the limitations of this approach.

I have been able to model successfully transformations which have local, non-interacting effects. The method is well suited for sets of transformations which have no (or, at least, non-degrading) effects on the applicability of one another. Modeling the effects of global transformations (e.g. floating-point to fixed-point arithmetic switch), or transformations which show dependences

(e.g. one is enabled by another, or two transformations are alternative) is possible in this flow, but the NFR network is not designed to evaluate all the possibilities in a single run.

In this case, it is the responsibility of the user to explore the tree of possible transformation choices, by employing my tools in multiple iterations. The construction of a method to automate this loop is possible, and left to future developments.

3.5 Tool implementation

I have implemented the estimation and the optimization flows described in this chapter.

The source-level software estimation flow acts externally as the GNU GCC compiler, thus allowing current projects to be compiled with their own unmodified makefiles. Steps 1–4 are performed by my pseudo-compiler tool, which pre-processes, parses and instruments the input code, finally compiling and linking it with the real GCC compiler. Step 1 is performed by a parser, designed in bison and flex starting from a free ANSI C syntax [147]. It is the most complex tool of the suite: it builds a complete type system for the input, it attributes atoms to nodes and selects pivots. Then, a pseudo-linker stealthily adds a library containing the implementation of the probe function. The resulting executable is equivalent to its non-instrumented version, only it generates profiling information. That information is processed by the post-processing tool, which actually performs the numerical calculation of estimates according to the model presented in Section 3.3.1 (page 77).

The transformation steering flow takes as an input the decorated parse trees generated by the pseudo-compiler, together with execution counts and consumption statistics. A critical section detector selects the regions where the most energy and time is spent, and an inferential engine determines and suggests to the user the optimizations which are likely to produce the highest gains. The engine performs this task on the basis of library containing a set of fuzzy-rules, able to detect specific features in the code; and a set of rules, which determine how much each optimization technique is suitable on a portion of code, on the basis of the degrees at which each of the above features are present. These rules are encapsulated in a which implements the interconnectivity and supervision of the NFR network, plus a separate module for each NFR rule. The final output is a report where each entry indicates an optimization to apply, the section where to apply it, and the degree of confidence in its beneficiality.

All the documentation and manuals regarding the implemented tools is available as separate documents. Due to volume and update reasons, it is not meaningful to report the above documentation in this thesis.

Chapter 4

The cost of syntax elements

In the overview of this thesis, I divided the estimation problem in two subproblems: estimating the single-execution counts of nodes, and profiling their execution counts. This chapter is entirely devoted to the first subproblem.

The solution I propose is an attribute grammar, which determines the single-execution costs of each node in the abstract syntax tree corresponding to the input source code. The cost attribution rules of this attribute are founded on an architecture-independent abstract translation model, which can approximate the behavior of an arbitrary compiler.

I present the topic in the following steps:

1. first, I introduce the concepts which this chapter uses (grammars, syntax trees, costs, atoms, abstract assembly instructions) and unambiguous notations to denote each of them: Section 4.1 (page 100);

2. then, I give an overview of all the factors (at the language level) which influence the cost of a syntax element: Section 4.2 (page 110);

3. then, I present an abstract model which describes how to translate C code into abstract assembly instructions, taking into account all the issues presented above: Section 4.3 (page 117);

4. finally, I present an attribute grammar which allows to determine the single-execution cost of nodes; this grammar is founded on the above translation model, and it allows cost determination without actually performing the abstract translation: Section 4.4 (page 136);

For the convenience of the reader, Section 4.5 (page 231) presents a quick reference summary the syntax rules used in the previous sections.

4.1 Notation

The determination of single-execution costs relies on cost attribution rules which are founded on an appropriate abstract translation model. In this chapter I present such an abstract translation model, and derive the corresponding cost attribution rules. In this sense, this chapter is fundamental for the rest of the methodology presented in this thesis. These cost attribution rules allow to associate an architecture-independent cost to each syntax entity in the original source code.

I achieve this goal via a multi-visit grammar attribute [13] based on the syntax of the C language. This grammar defines, for each applicable symbol, an attribute *c* which represents its single-execution cost. The actual calculation of the value of attribute *c* for all the symbols appearing in a given source code relies on a number of other attributes, on which *c* depends, whose purpose and features will be motivated in the next sections.

In this section I introduce the assumptions and the notation I will use to describe syntax rules, semantic rules and abstract assembly translations.

4.1.1 Denoting syntax and semantic rules

Since I will define the cost determination process as a multi-visit attribute grammar over an abstract syntax tree, a formal notation is needed to specify grammar rules: both syntax rules and semantic rules. Syntax rules describe how symbols derive more complex sentences; all the syntax rules together define the language. Semantic rules describe how to calculate the symbol's attributes. In this section, I describe the notation used in this thesis to specify grammar rules.

Syntax rules will be typeset in a usual style, as in the following example:

⟨*multiplicative_expression*⟩ ::= ⟨*cast_expression*⟩
| ⟨*multiplicative_expression*⟩ '*' ⟨*cast_expression*⟩
| ⟨*multiplicative_expression*⟩ '/' ⟨*cast_expression*⟩
| ⟨*multiplicative_expression*⟩ '%' ⟨*cast_expression*⟩

Nonterminal symbols are enclosed in angular parentheses. Terminal symbols are written in capitals (like 'IDENTIFIER') or single-quoted (such as '+', '>', '++', '!=', 'sizeof', 'if', ...).

I employ two meta-symbols:

- '::=', which separates the left-hand side of each grammar rule from its possible right-hand sides;
- '|', which separate alternative right-hand sides corresponding to the same left-hand side.

No meta-symbol is used to mark the end of a rule. No meta-symbol is used to denote an empty string. The two conditions will be marked by vertical and horizontal whitespace respectively. Meta-symbols are neither quoted nor typeset in a special style. Confusion between one-character terminals and meta-symbols is not possible, since the former ones are quoted whereas the latter ones are not. Indentation and line-breaking are purely cosmetic and bear no semantics.

Whenever the same non-terminal symbol appears multiple times in a rule, its occurrences in the right-hand sides will be explicitly numbered to avoid confusion: ⟨*statement-1*⟩, ⟨*statement-2*⟩, ... The left-hand side symbols are never numbered (e.g., ⟨*statement*⟩). That bears no risk of ambiguity because in this thesis I will deal only with the grammar of the C language, which is context-free, therefore there will always be exactly one non-terminal symbol on the left-hand side.

Attributes are marked in a dotted notation; e.g., attribute *cc* of symbol ⟨*assignment_expression*⟩ is denoted as ⟨*assignment_expression*⟩.*cc*. Depending on what is more convenient in each case, attribute evaluation will be expressed in one of the two following forms:

- in the form of a semantic rule, associated to exactly one syntax rule;
- in the form of (usually recursive) pseudo-program working on an abstract syntax tree.

Note that the two forms operate on different models: the first on syntax rules, which determine the Concrete Syntax Tree (CST), and the second on the Abstract Syntax Tree (AST). Nevertheless, there is no possible ambiguity or conflict, since for each CST there exists exactly one corresponding AST; and all the nodes of a CST which correspond to the same AST node will share the same attributes. Section 4.1.2 details these claims.

A notation is required for semantic rules (the first of the two forms listed above). Semantic rules specify how to calculate a given symbol's attribute, possibly as a function of one or more attributes, either belonging to the same symbols or other symbols in a given grammar rule. For this reason they are always accompanied by an associated syntax rule. The two rules will be marked as 'Syntax:' and 'Semantics:', as in the following example:

Syntax:

⟨*assignment_expression*⟩ ::= ⟨*unary_expression*⟩ '=' ⟨*assignment_expression-1*⟩

Semantics:

⟨*assignment_expression*⟩.*cc* = *CC*(⟨*assignment_expression-1*⟩.*t*, ⟨*unary_expression*⟩.*t*);

(where $CC(\cdot,\cdot)$ is a function described separately).

I will often distinguish between statements and expressions. I will define for statements a different set of attributes than expressions. All the non-terminals to which a cost must be assigned fall into one of these categories. These two categories cover approximately 70% of the symbols in the C grammar. The remaining part is dedicated to declarations. Declarations are determinant for the construction of symbol tables, on which the type resolution mechanism relies, and types influence cost. But declarations, have no cost *per se* and will not be discussed here.

However, an unambiguous definition of 'statement' and 'expression' is needed. With reference to the grammar of the C language, of which an excerpt regarding statements and expressions is reported in Section 4.5.2 (page 232), statement symbols are the non-terminal symbols which belong to the transitive closure of the

derivation from symbol ⟨*statement*⟩ with copy rules only, namely: ⟨*statement*⟩, ⟨*labeled_statement*⟩, ⟨*compound_statement*⟩, ⟨*expression_statement*⟩, ⟨*selection_statement*⟩, ⟨*iteration_statement*⟩, ⟨*jump_statement*⟩.

On the other hand, expression symbols are all the non-terminal symbols which belong to the transitive closure of the derivation from symbol ⟨*expression*⟩, according to the grammar (see Section 4.5.1), for example ⟨*expression*⟩, ⟨*assignment_expression*⟩, ⟨*logical_or_expression*⟩, ⟨*logical_and_expression*⟩, ⟨*inclusive_or_expression*⟩, ⟨*exclusive_or_expression*⟩, ⟨*and_expression*⟩, and others.

4.1.2 Concrete and abstract syntax trees

The cost determination algorithm proposed in this thesis operates on Abstract Syntax Trees (ASTs), instead of Concrete Syntax Trees (CSTs) because ASTs are much simpler to handle than CSTs. Therefore, a transformation is needed to obtain the AST of a source code from its CST. This section motivates the above claims and describes such a transformation.

Concrete syntax trees are the parse trees as obtained from parsing the input source code according to the grammar of the C language; determining the CST of an input is indeed the essence of parsing. CST are poorly suitable for semantic reasoning, because they are overly complex, and impractical to handle. In fact, a large portion of the information they contain is relevant during parsing but not for semantic analysis. Due to their complexity, writing semantic actions which navigate concrete syntax trees is an awkward task.

> «The goal of an abstract syntax is to describe the structural essence of a language. Syntax trees are operators – describing the important concepts in a language, applied to typed operands – describing the important components associated with the concept. Each operand is named with the role it plays in the concept. Trees are classified in a tree of types known as phyla – describing inheritance relationships between concepts. Abstract syntax trees terminate in a pre-specified set of primitive types, such as identifier and integer.
>
> For example, an if-then-else tree will have a structure that contains three slots, one for each of the predicate, the then branch and the else branch. Labeled trees may be used to model abstract syntax trees; the various slots of the trees will correspond positionally to sequence elements. It will not matter which positions correspond, but merely that the correspondence be consistent throughout the tree. » [15].

A CST can be much larger than an AST, even for simple expressions. Consider the following expression:

```
a = (b + c) * d
```

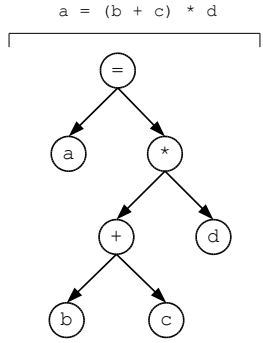


Figure 4.1: The Abstract Syntax Tree (AST) for a simple expression.

its AST, which is represented in Figure 4.1, comprises 7 nodes. On the other hand, its CST (in Figure 4.2) comprises 57 nodes.

While ASTs describe only the hierarchical relation between the symbols of a program (e.g. a function definition contains given statements, and they may be compound statements and contain, in turn, other statements), CSTs describe the complete derivation tree from the grammar's axiom down to the terminals, and this information is irrelevant during semantic analysis. There are two major causes of the semantically irrelevant information: delimiters and copy rules.

Delimiters are terminal symbols (e.g., the comma ',', the semicolon ';', and the various parentheses, brackets and braces, in the C language) which denote the beginning and the end of sequences, and separate elements. Delimiters are useful for human readability, and parsing relies on them; but once parsing is complete, there is no need to keep them in the syntax tree.

The second source of semantically-irrelevant information is the way in which C grammars model the precedence among operators. They use a large number of copy rule alternatives (copy rules are those whose right-hand side is composed of just one nonterminal symbol), which generate long linear chains of derivation. These chains have no meaning during semantic analysis, and they unduly complicate the code which traverses the tree for attribute calculation.

Now, an algorithm is needed to obtain the AST of a C source code, starting from its CST. Defining such algorithm involves no conceptual difficulties, nevertheless I prefer to describe it fully, to avoid ambiguities.

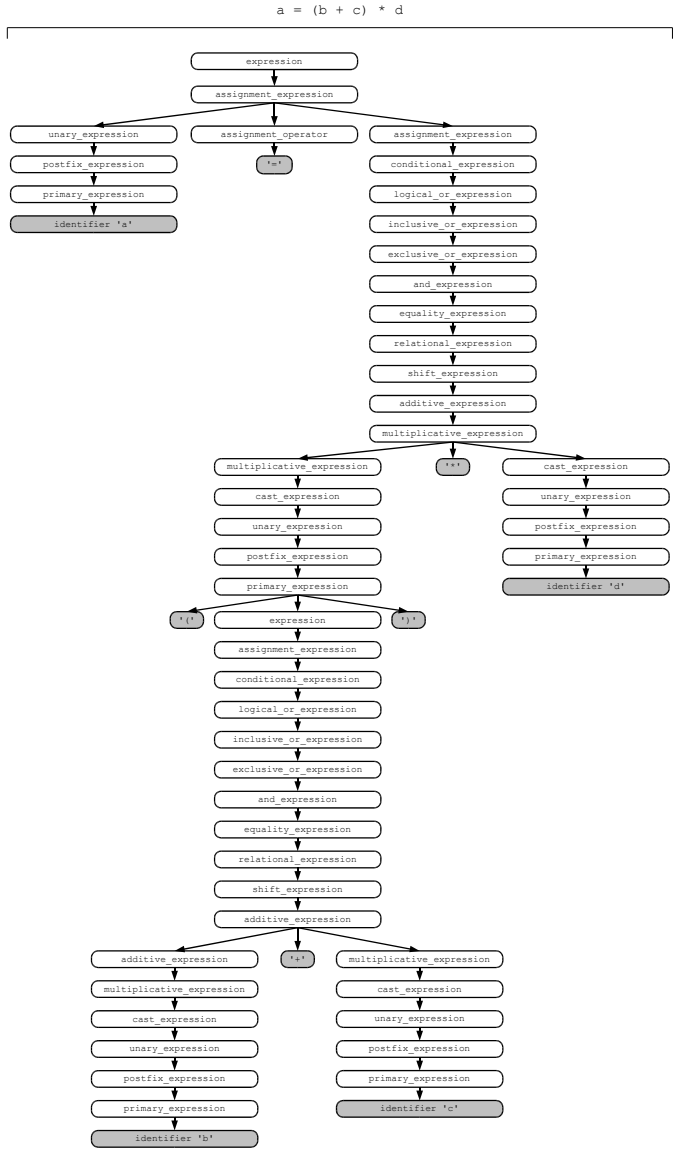


Figure 4.2: The Concrete Syntax Tree (CST) for the same expression.

The algorithm I propose comprises the following steps:

1. Remove all the nodes containing the following terminals: '(', ')', '[', ']', '', '', ':', ',', ';'; with the following exceptions:
   - "comma expressions": nodes ',' when they appear in rule
     $\langle expression \rangle$ ::= $\langle expression \rangle$ ',' $\langle assignment\_expression \rangle$
   - "null statements" : nodes ';' when they appear in rule
     $\langle expression\_statement \rangle$ ::= ';'
2. when a node has one single, non-terminal child (i.e. it corresponds to the recognition of a *copy rule*, merge the two nodes together; repeat this operation until the fixed point is reached;
3. "promote" any operator terminal node, untouched by the previous rules, by moving its contents into its immediate father node.

Figures 4.3, 4.4, 4.5 illustrate how the above algorithm works: they represent how the CST of the expression from the previous example is transformed respectively after Step 1, 2 and 3 of the algorithm above.

The practical task of obtaining an AST by specifying appropriate semantic actions in a parser generator such as yacc [144, 143], SableCC [148] or ANTLR [149] is simple. These tools generate bottom-up parsers, and each semantic action is also executed bottom-up, immediately after the associated syntax rule is recognized. A common approach is to associate a synthesized attribute s to each nonterminal $\langle A \rangle$, which contains the abstract syntax subtree rooted at the node corresponding to that nonterminal. Then, for each syntax rule $\langle A \rangle ::= \langle B \rangle$, provide a corresponding semantic rule which determines $\langle A \rangle.s$, usually by connecting subtrees $\langle B \rangle.s$, $\langle C \rangle.s$, ... This technique is thoroughly explained in Section 2.2.5.9 of [77]. The above semantic rules are designed to discard and merge nodes as prescribed by the Steps 1, 2 and 3 above. For example, all copy rules such as $\langle A \rangle ::= \langle B \rangle$ have semantic actions which set $\langle A \rangle.s = \langle B \rangle.s$, which realizes Step 2.

4.1.3 Describing semantic attribute evaluation

As anticipated in Section 4.1.1 (page 100), I adopt two different styles to indicate how to derive attributes, for economy of formulation and convenience. In this section, I informally motivate why these styles are equivalent and compatible.

For synthesized attributes, I preferably specify their evaluation in the form of semantic rules, as the previous example:

$\langle assignment\_expression \rangle.cc = CC(\langle assignment\_expression\text{-}1 \rangle.t, \langle unary\_expression \rangle.t)$;

Each of these semantic rules is associated to exactly one syntax rule, and it defines some attribute of the left-hand side non-terminal symbol in terms of attributes of the right-hand sides symbols. Therefore, these rules operate on the CST.

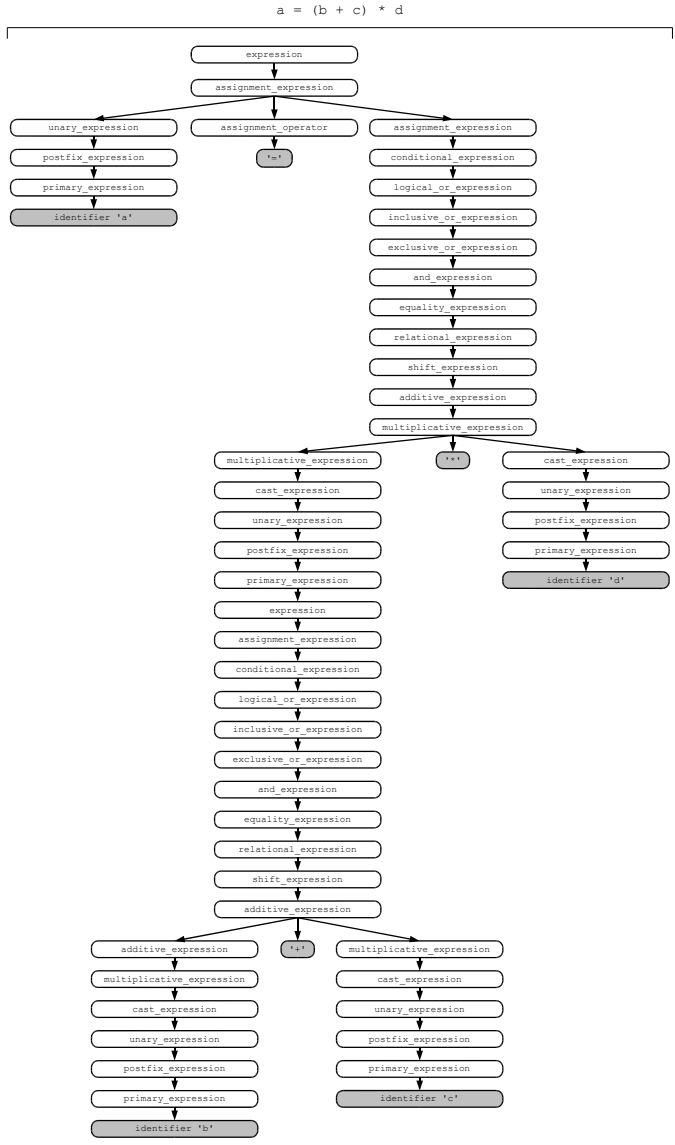


Figure 4.3: Example of transformation from CST to AST. The figure shows how the CST presented before is transformed after Step 1.

I will also describe the evaluation of some of the inherited attributes via the same semantic rules. Alas, for some of them this approach is too cumbersome. It is more convenient to specify their evaluation by means of an im-

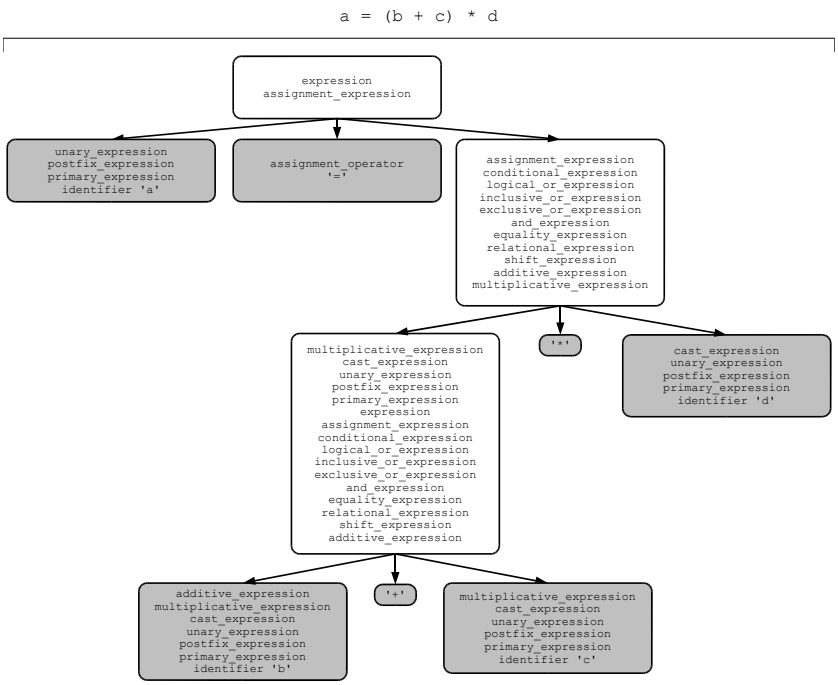


Figure 4.4: Example of transformation from CST to AST. The figure shows how the CST presented before is transformed after Step 2.

perative algorithm which operates on the attributes of the AST nodes. AST nodes, not symbols. I will specify these algorithms in the form of recursive, pseudo-C functions. For an actual example of such an algorithm, see Section 4.4.4 (page 176).

Since the two description styles operate on the CST and on the AST respectively, the issue could be raised whether they are equivalent and consistent. Equivalence means that all the attribute calculation algorithms which can be expressed in one manner can be expressed also in the other and vice versa. Consistency means that the same attribute must have the same value in all the CST nodes which correspond to the same AST node.

It is easy to prove that the two forms are equivalent; changing form may require the introduction of additional attributes to store the results of intermediate evaluation steps.

Consistency is, on the other hand, responsibility of the semantic algorithm designer. For the grammars introduced in this thesis, an informal proof that consistency is guaranteed is given by the fact that whenever node collapsing happens between non-terminal nodes, the two non-terminals belong to the same category (where category is “expression” or “statement”).

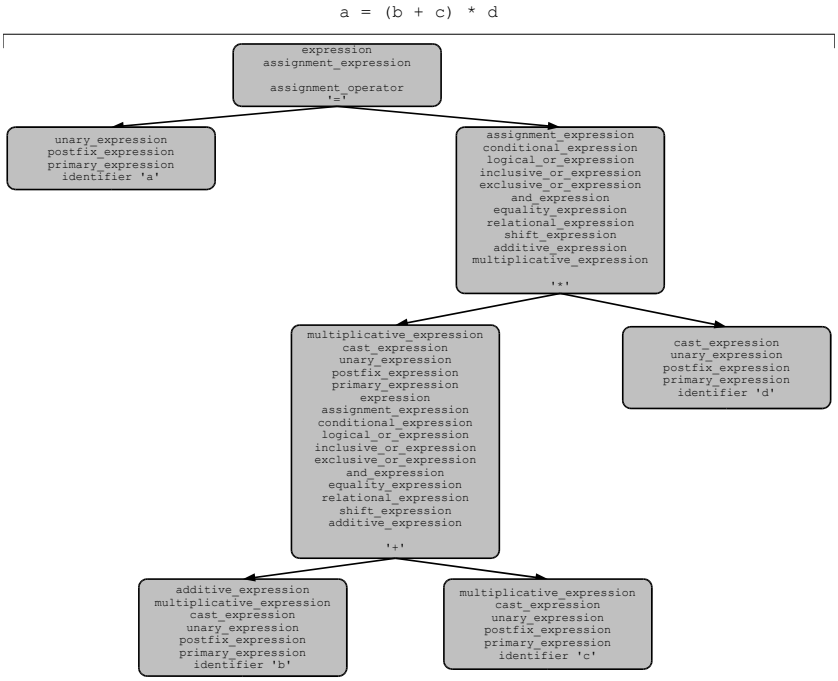


Figure 4.5: Example of transformation from CST to AST. The figure shows how the CST presented before is transformed after Step 3.

4.1.4 Denoting assembly translations

In Sections 4.3 and 4.4 I will have to describe the translation of syntax elements. The way I choose to do so is an architecture-independent abstract assembly language. For my convenience, I include in this language specific traits which greatly simplify the description of the abstract translation process. In this section I present and motivate the details of this language.

The single-execution cost of a syntax node is the cost which the executor undertakes when executing its translation. Determining this cost without performing the actual translation requires a model of the translation process, such as the one described in Section 4.3 (page 117).

This translation model defines, for each symbol in the grammar, one or more attributes containing translations of that symbol. Therefore, this model requires a way to express this translations in an abstract and convenient way. The solution I propose is an assembly language whose instruction set is composed by the abstract instructions introduced in Section 3.2.1 (page 67).

Additionally, for my convenience, I now extend this language to allow nested blocks. A block is a section of assembly code surrounded by braces (‘{’ and ‘}’, as in C). Blocks can nest. Labels declared in a given block are

visible in the block in which they appear, and in all the blocks included in it.

Labels in blocks are subject to the same scoping rules of the C programming language: a 'jump else' instruction causes the control to be transferred to a given instruction with label 'else:'. That instruction might either be located in the same block where the jump instruction appears, or in an outer block. There could be multiple instances of the same label: the targeted one is the first which is found by starting the search in the current block and proceeding up the hierarchy of nested blocks.

The following section of assembly code illustrates an example usage of labels in nested blocks:

```
1   {
2           add a, b, #-1
3           {
4                   cmp  a, #0
5                   jeq  end
6                   add  a, b, #1
7                   j    else
8             end:
9           }
10          add  b, b, #1
11    else: call f1
12    end:
13  }
```

The jump instruction in line 5 targets the label 'end' which appears in line 8, and not the one in line 12. The jump instruction in line 7 targets the 'else' label on line 11.

Blocks provide a convenient mechanism to abstract the label patching phase, a task that a real compiler must undertake in some way (e.g. see Section 11.5 and 7.2 of [78]), whereas I just need to model it. Under my chosen assumptions, this phase only causes changes in the offset field of encoded instructions. It does not affect in any way the number, quality or length of instructions; therefore it has no effect on the cost of the translation. As a consequence, I can safely derive cost rules on the basis of assembly translations with block nesting, and hold the guarantee of their validity on its corresponding flat (nesting-free) assembly rewriting, and the corresponding machine encoding.

The reason why I introduce block nesting in my abstract assembly is that label scoping (as just described) greatly simplifies the task of describing the abstract translation model. For example, it makes it possible to specify a jump in the translation of a children node, which targets a label in the translation of one of its ancestor, whose translation will be calculated later. Also thanks to this feature, my abstract translation model can be described as a simple attribute grammar, made of synthesized attributes only, possible to evaluate in a single bottom-up visit.

4.2 Which factors affect the cost of syntax elements

It is evident that the same syntax element can have different translations, depending on the circumstances in which it appears. For example an assignment operator '=' is translated differently, depending on the type and size of the operands, and other factors. Different translations have different cost. This section gives an overview of all these factors which influence the translation –and therefore the cost, of syntax elements. These considerations are at the basis of the attribute grammar which I will present in Section 4.4 (page 136).

I distinguish the following three categories of cost:

- inherent cost,
- conversion cost,
- flow control cost.

Statement nodes only have flow control costs. Expression nodes may have all the categories of costs. Declarations have no cost. Each category of cost is influenced by different factors.

The inherent cost of a syntax node is the cost of executing the data transfer or manipulation associated to the node. Examples of inherent costs are the cost of performing a floating point addition, of accessing the contents of an array, or of assigning the value of a variable to another.

The conversion cost of a syntax node is the cost of converting the data between different representations (e.g., from integer to floating-point or vice versa). Conversions may be triggered by many circumstances, just to name a few: a binary operation or an assignment between different types (one of the two operand is promoted), an explicit *casting*, a function call.

The flow control cost is the cost of transferring the execution flow to another portion of the program. In case of conditional code, this cost also includes determining whether the jump must be taken or not. Obviously control-flow statements such as 'if', 'for' and 'switch' have non-zero flow control cost, but also some of the expression may have, due to their conditional behavior or to the short-circuit evaluation mechanism.

In the following paragraphs, I examine which circumstances influence the above costs.

4.2.1 The valueness affects the inherent cost

The inherent cost of an expression depends in general on whether its R-value or L-value is used. I will refine later this claim into a formal and unambiguous form. In the meanwhile I will clarify this idea by example. The reader please indulge me with my need for informality.

Given the following assumptions on the type of variables 'a' and 'b',

```
int  a;
int * b;
```

I consider the following statement:

```
a = *b;
```

The statement copies the contents of the memory word whose address is contained in the register associated with variable 'b' into the register associated with variable 'a'. The abstract assembly rendition of this expression is 1 mvld instruction. This suggests the preliminary idea that, in general, the cost of a '*' operator is 1 mvld.

This belief is disproved by the following example:

```
*b = a;
```

The statement requests the copy of the word contained in variable 'a' into a memory word whose address is contained in variable 'b'. Its abstract assembly translation is composed of 1 mvst instruction.

Therefore, the cost of a unary '$*$' operator is not unique, and it depends on the context in which it appears. This difference is due to the fact that in the first example the R-value of expression '$*$b' is used, whereas in the second one, the L-value of the same expression is used.

Traditionally, the L-value of a variable is the storage area bound to a variable during execution whereas the R-value of a variable is the encoded value stored in the location associated with the variable (i.e., its L-value) [17]. The names R-value and L-value derive from the right and left position of operands with respect to the assignment operator '=', whose R-value and L-value are used respectively.

I call *valueness* the property of a given expression of its R- or L- value being used in a given source code. I say that the valueness of an expression whose R-value is taken is R; and the valueness of an expression whose L-value is used is L. The cost of a '$*$' operator is affected by the valueness of the expression which it forms.

The definitions from [17] reported above suggest the idea that valueness is defined only for identifier-type terminal symbols. This is false. Expressions such as '$*$(A[4]$->$p + 2)', which are significantly more complex than just a terminal, may represent a variable and their cost is influenced by their valueness.

An expression may have R and L valueness at the same time. This happens to the first operand of a compound assignment operator (e.g., '+=', '$-$=', '$*$=', ...). It has first its R-value and then its the L-value used. I say that the valueness of such an expression, whose R-value and L-value are used, is RL.

Multiple choices could be done in order to attribute costs depending on valueness. To avoid ambiguities, I define what I mean by "using" an R-value or an L-value, and I introduce the additional concept of "taking" an L-value:

- **using the R-value of an expression** means executing the assembly code which is required to bring the encoded value of the result of the given expression into a register or a bank of contiguous registers whose name or names are known (whether the value fits a single word or multiple words. If the expression consists of a simple variable identifier, using its R-value has zero cost, because the compiler will allocate that variable in a register or register bank whose name is known, and

no data need to be transferred. Always mind the perfect overlapping of memory and register space, and the distinction between "known by name" or "known by address". If the expression is more complex, e.g. it comprises an array subscript operators, such as 'a[i]', using its R-value means executing the instructions required to copy into a register or register bank the contents of the 'i-th' cell of array 'a', which includes at least one address calculation instruction, and at least one mvld instruction;

- **using the L-value of an expression** means executing the assembly code which is required to transfer into the memory locations associated to the given expression an encoded value, present at this time into a register or register banks, whose name or names are known. Again, if the expression is a simple variable identifier, using its L-value has no cost, since –due to the way the abstract assembly is designed– the results of expressions are always left in registers, and no data need to be transferred. Using the R-value of a more complex expression, such as the one in example 'a[i]' described above, means executing the instructions required to copy the contents of a register or register bank into the 'i-th' cell of array 'a', whose address or addresses need to be calculated;

- **taking the L-value of an expression** is something conceptually different from using it. This is a case I have not mentioned before; the emphasis is on the difference between *using* and *taking*. The L-value of an expression is the set of memory locations where that expression is allocated, and using that L-value, according to my definition, means storing data at that address. Using an L-value may mean to execute instructions to calculate addresses, and instructions to transfer one or more words to those addresses, as just said. Taking an L-value of an expression means determining the address of the first location where an expression is allocated. In the C programming language, to take the L-value of an expression one must use the '&' operator. C has no implicit address-taking mechanisms, such as parameter passing by reference (such as VAR arguments in Pascal) or reference semantics (such as in C++). By convention, when the L-value of an expression is taken, I say that it has zero valueness, or Z-valueness for short, or "its Z-value is used".

The above considerations motivate the need to introduce in my grammar an attribute "valueness", for symbols of the category "expressions". Section 4.4.4 (page 176) presents an algorithm to determine the valueness of a given expression in a given source code.

4.2.2 The operand type affects the inherent costs

The type of operands affects the inherent cost of the operators which manipulate them.

Firstly, the cost of arithmetic operators is evidently dependent on the operand types. For example, the cost of integer arithmetics is different from floating-point arithmetics, and the cost of single-precision arithmetics is different from double precision or long double precision.

Secondly, the cost of an operator which involves the access to data is affected by the size of that data, which is a function of its type. For example, the assignment between an array cell and a variable, such as:

```
a = b[i];
```

is rendered as 1 mvld instruction if the type of a (which is the base type of array b) has a representation which fits in a single word, depending on the architecture; this is usually true for the types **char**, **int**, **long int**, **float** and for all pointer types on 32-bit target platforms. Other types, such as **long long int**, **double**, **long double**, and large **struct**s do not fit in a word. Access to variables and expressions occupying multiple words require multiple data transfer instructions, and possibly address resolution instructions between each transfer and the next one. If t is the type of an expression (in some appropriate representation, for example the one described in Section 4.4.1 (page 140)), I define $W(\langle e \rangle)$ as the function which returns the number of words occupied by $\langle e \rangle$, formally:

$$W(\langle e \rangle) = \left\lceil \frac{\texttt{sizeof}(\langle e \rangle)}{wordsize} \right\rceil ,$$

where *wordsize* is the size of each register expressed in bytes, supposed equal to the size of the memory word.

Thirdly, the *alignability* of a type affects the cost of array access and pointer arithmetic operators which involve it. In fact, accessing the i-th element of an array requires calculating its offset, by multiplying the array base type size and the index. This multiplication can be avoided when the base type has size equal to 1 byte, and it can be optimized to a bitwise left shift operation when the base type has a size which is a power of 2. Given a type t, I say that its alignability $A(t)$ is:

- Byte, if $W(t) = 1$;
  the calculation of the offset needs no multiplications;

- Aligned, if $W(t) = 2^i$ for some positive integer i;
  the offset of objects having a size which is a power of two requires a multiplication by a power of 2, which is translated as a right shift;

- Misaligned, otherwise;
  the offset of objects having a size which is not a power of two requires a true integer multiplication.

Section 4.4.8 (page 216) discusses the choice of a good representation for type, and the associated evaluation rules, and additional issues which it is not convenient to report here.

4.2.3 The operand type affects the conversion costs

The C language has a complex type system, which causes a large number of type conversions to take place stealthily when user code is translated and executed.

Arithmetics between operands of different types requires one of the two to be promoted. Function calls require the actual arguments to be converted, one by one, into the respective formal argument types. Return statements require the operand expression to be converted to the current function's declared return type. These are just examples; the compound assignment operators and the function call operator exhibit more exotic behaviors.

I discuss exhaustively the determination of the conversion costs in Section 4.4.8 (page 216).

4.2.4 The constancy affects all the costs

If an expressions has a constant result, the result can be determined statically. If that operation has no other effects than determining its result, it can be eliminated by the compiler.

Therefore, the translation of constant expressions which do not have side effects does not appear in the assembly translation of the source code, or in the corresponding binary executable. Consequently, they have no cost.

On the other hand, operations on constant operands which do produce side effects (e.g., function calls and assignments) cannot be eliminated by the compiler, so they have a non-empty translation and a corresponding non-zero cost.

Nevertheless, the larger expression in which they appear may be constant-value, and therefore the operator which manipulates them has no translation and zero cost.

The following examples illustrate these cases for inherent costs:

- in the following expression:

  offset = **sizeof**(**struct** pnode) * 2 + 57;

  the '*' and '+' have no cost, because their operands are recursively constant expressions, and they have no side effects; on the other hand, the '=' operator has non-zero cost, because it has a side effect even though his operands are constant;

- in the following expression:

  padding = **sizeof**(**struct** pnode) * 2 − (offset = **sizeof**(**struct** pnode) * 2 + 57) − 1;

  which includes the previous one in parentheses, all the '*' and '−' operators have no cost, because all their operands are constant, including the one in the parentheses. On the other hand, the two '=' operators have non-zero cost.

The same applies to type conversion: if a conversion prescribed by the language involves a constant operand, it can be performed at static time. It needs no translation and causes no cost.

Finally, the same applies to flow control costs also. For example, if the condition expression of a selection statement is known statically, the selection statement can be suppressed, and only one of the two branches kept.

Due to short-circuit evaluation and conditional expressions, the constancy of an expression may actually depend on the value of its subexpressions. This is why, in order to determine constancy, also the value of its constant subexpressions must be considered at static time. Additionally, due the presence of the '**sizeof**' operator, this constant value depends in turn on type.

Section 4.4.3 (page 169) propose an evaluation scheme for constancy, which is in turn used to determine cost attributes consistently with the issues just described.

4.2.5 The translation flavor affects the control-flow and inherent costs

In the C language, the same expression may translate to different versions depending on the context where it appears. The abstract translation model I will propose derives the translation of a node by composing chosen versions of the translation of children nodes, possibly with additional code. It is important to determine which version actually appears in the translation of the source code, since different versions have different costs. This section motivates and clarifies this need.

In the C language, the same expression may translate to different versions, which I call *flavors*, depending on whether its result value is needed in a numerical or logical sense. For example:

- when an expression 'E' appears in a larger arithmetic expression, such as 'a + E', its numerical value is needed;
- when it appears as a condition in a conditional statement, such as in '**if** (E) ... ' or '**while** (E) ... ' statements, or in logical expressions such as 'a && E', then its logical value is needed.

In the first case, a *single-entry, single-exit* translation of 'E' is used; in the second case, one of the possible *single-entry, double-exit* translations of 'E' is used. In my model, each translation is either single-entry, single-exit or single-entry, double-exit. For short, I just call them single-exit and double-exit translations, respectively.

A *single-exit* translation has a single entry point and a single exit point. Whenever a single-exit translation is executed, the execution flow starts at its beginning and finishes at its end. Single-exit translations are not, in general, basic blocks. A basic block may not contain any labels or jump instructions; on the other hand a single-exit translation may contain arbitrary jumps and labels, provided that all the jump instructions jump to internal labels, and no internal label is the target of a jump instruction which is outside the translation. In my translation model, the translation of an expression which is used as an operand of an assignment, arithmetic or bitwise operator is single-exit. I call a single-exit translation a *T-flavor* translation.

A *double-exit* translation has a single entry point and two exit points. One exit point is just after the last instruction of the translation, the other is a jump to an external label. As I show in Section 4.3 (page 117), it is possible to build an entire translation scheme for all the control flow structures and expressions of the C language with just two types of double-exit translation: a *TT-flavor* and a *TF-flavor* translation. I postpone any further explanation about translation flavors to Section 4.3.

Since different flavors of the same expression have different costs and features, a set of rules is needed to determine what flavor is actually selected to appear in the final translation of a given node. This requires a complete (though simplified) translation model for the grammar of the part of the C syntax describing statements and expressions. Then, cost attribution rules must be based on that model. Describing this model is the purpose of Section 4.3 (page 117).

4.2.6 The register boundedness affects the inherent cost

The translation of assignment expressions (and therefore their cost), depends on whether the result of the right operand is bound to a register or not. This section motivates and details these claims.

The single-exit translation flavor of a given expression (as just explained) leaves its result in a register or bank of registers, depending on its size. The name of these registers may appear in the translation as specified in one of the following ways:

- names of a physical registers in the architecture: R1, R2, ...;
- conventional names such as ‘a’, ‘b’, ... (where ‘a’, ‘b’, ... are identifiers in the C source code associated to the translation) which are, in fact, a shorthand notation to express “the physical register where variable ‘a’ was allocated”, “the physical register where variable ‘b’ was allocated”, ...
- the conventional name ‘free’, which means “some arbitrarily chosen register such that it does not conflict with the other register allocations of variables”.

In all the cases but the last one, the translation is said to be bound to a register, or *register-bound*.

The cost of an assignment operator depends on the register-boundedness of its operand expressions. In detail, the assignment of an expression which has a register-unbound single-exit translation has no cost, because it can be realized by replacing ‘free’ in its translation with the target register in the assignment. On the other hand, the assignment of an expression which has a register-bound translation requires executing instructions to physically move data from the registers where the translation leaves its results to the assignment target registers. The topic will be covered exhaustively in Section 4.4.7.14 (page 212).

4.3 An abstract translation model

In this section I present an abstract translation model on which the cost modeling decisions will be founded. From this model I will directly derive the evaluation rules for flavor and register-boundedness. Moreover, this model indirectly helps the definition of the other attributes by removing ambiguities and unnecessary degrees of freedom. It is in the form of an attribute grammar. The model is simple: it just comprises 4 synthesized attributes, which can be evaluated in a single bottom-up pass. The section introduces first some assumptions which allow a simpler, easier to understand model; then its details; finally some examples.

Avoid confusion between this attribute grammar and the one presented in Section 4.4 (page 136). The former is mainly designed to support reasoning on translation flavors and register boundedness, while the latter is designed for the broader task of determining node single-execution costs. The relevance of the former is limited to this section only, whereas the latter is discussed throughout the entire chapter. The two grammars share the same syntax –expression and statements of the C language, as in Section 4.5 (page 231), but they are distinct, and the former is, in the end, not part of our cost-estimation methodology.

4.3.1 I privilege understandability

Ideally, the translation model I present should be complete, correct and easy to understand. Clearly these objectives cannot be achieved at the same time. Here, I assume that the reader is interested in getting a broad idea of the topic, therefore I privilege understandability at the expenses of correctness and completeness. Readers interested in a complete, correct, and free-from-ambiguity version of this model should refer to the source code of the tools which implement this thesis. Definitely, that description is not suitable to be presented here.

Precisely, I avoid now to deal with all those factors which can be treated separately, and I also neglect a number of constructs.

It is safe to ignore here all the issues related to type, size, valueness and constancy, because these factors cause effects on the assembly translation which are independent and additive with respect to effects of caused by flavors and boundedness. Therefore, without any loss of generality, I assume that all the involved types have single-word size, integer type, and are not constant.

Consistently, I also postpone any considerations on aggregate, pointer and array access operators, and on compound assignment operators, conditional operators and others.

4.3.2 Attributes

As anticipated, a given symbol may have multiple translations, which are either single-exit or double-exit.

Statements have exactly one translation, and it is a single-exit (a.k.a. a T-flavor) translation. In the attribute grammar which composes the abstract translation model, I associate to each statement symbol an attribute T, which contains the monolithic translation of that symbol.

Expressions have one single-exit translation and two double-exit translations:

- their single-exit translation is relevant when the *numerical* value of the expression is required in the context where it appears. As just done for statements, I associate to each expression symbol an attribute T, which contains its T-flavor translation. Expressions have a result, and their single-exit translations leave it in a register. It the translation is register-bound, the name of this register is indicated in attribute R. Otherwise R assumes the value 'free'.

- the double-exit translations of an expression are relevant when the *logical* value of the expression is required in the context where it appears. In double-exit translations, the logical value of the expression is checked and, depending on the outcome, a conditional jump is taken or not. Therefore, the control flow can leave the translation either by reaching the end of the translation, or by jumping to an external label. Double-exit translations are available in two flavors:

  - the TT flavor, which jumps to an external 'then' label when the condition is true, and
  - the TF translation, which jumps to an external 'else' label when the condition is false.

  For each expression, I define a couple of attributes TF and TT which respectively contain the TT and the TF translation flavors.

The composition of TT and TF flavors allows to model accurately and easily the short-circuit evaluation mechanism. The TT translation is required for all but the last operands of a logical 'or' expression: this way, the first sub-expression which evaluates to true determines the truth of the entire expression, and causes a jump to the 'then' branch, with no need to evaluate the remaining sub-expressions. The TF translation is required for all but the last operands of a logical 'and' expression: this way, the first sub-expression which evaluates to false determines the falsity of the entire expression, and causes a jump to the 'else' branch, with no need to evaluate the remaining sub-expressions. The last operand expressions of a logical 'or' or 'and' operator can be translated indifferently in a jump-if-true or jump-if-false fashion.

I summarize the attributes just introduced for statements and expressions in Table 4.1.

4.3.3 Some useful functions

I introduce a simple function called $replace(\cdot, \cdot, \cdot)$. Given an abstract translation t, and two symbols a and b (which can be assembly mnemonics or

Symbol class	Attributes	Description
statements	T	single-exit assembly translation
expressions	T	single-exit assembly translation
	R	register where the result is left by the single-exit translation
	TT	double-exit assembly translation, *jump-if-true* flavor
	TF	double-exit assembly translation, *jump-if-false* flavor

Table 4.1: The attributes in the grammar attribute which constitutes my abstract translation model.

register names), $replace(t, a, b)$ returns the translation obtained by replacing all the occurrences of a with b in the most external block of t. I emphasize: without modifying any nested blocks in t.

Given a register-unbound translation t, a new translation bound to register r is given by: $replace(t, \text{free}, r)$.

I also introduce the *binding operator*, which I denote with an exclamation mark ('!'). It denotes the register-bound rewriting of a register-unbound translation.

Given an expression N such that $N.T$ is defined, and $N.R =$free, I define $N.T!$ and $N.R!$ as follows: $N.T! = replace(N.T, free, Ru)$; $N.R! = Ru$, with Ru which is some unique register name. A unique register name, for our purposes, is the name of a register such that it is never used by any other variable or temporary. Since in my abstract architecture there are infinite registers, it is always possible to allocate a new register without spilling.

I define $invertjump(\cdot)$ as a function which replaces the top outermost jump instruction in its argument with another jump instruction, having the opposite conditions. It replaces 'jne' with 'jeq' and vice versa; it replaces 'jgt' with 'jle' and vice versa; it replaces 'jlt' with 'jge' and vice versa.

4.3.4 The attribute grammar which is the model

The details of the attribute grammar which realize the proposed abstract assembly translation of an arbitrary C program follow.

---

1. '**if** (...) ... ' statement:
   Syntax:
   ⟨*selection_statement*⟩ ::= '`if`' '(' ⟨*expression*⟩ ')' ⟨*statement-1*⟩
   Semantics:
   ⟨*selection_statement*⟩.T =

   {

```
        ⟨ expression ⟩ .TF
        ⟨statement−1⟩.T
  else:
}
```

2. '**if** (...) ... **else** ... ' statement:
   Syntax:
   ⟨*selection_statement*⟩ ::= '`if`' '`(`' ⟨*expression*⟩ '`)`' ⟨*statement-1*⟩ '`else`' ⟨*statement-2*⟩
   Semantics:
   ⟨*selection_statement*⟩.*T* = one of the following:

```
{                                   {
        ⟨ expression ⟩ .TF                  ⟨ expression ⟩ .TT
        ⟨statement−1⟩.T                     ⟨statement−2⟩.T
        j end                               j end
  else: ⟨statement−2⟩.T             then:   ⟨statement−1⟩.T
  end:                              end:
}                                   }
```

3. '**while**' statement:
   Syntax:
   ⟨*iteration_statement*⟩ ::= '`while`' '(' ⟨*expression*⟩ ')' ⟨*statement*⟩
   Semantics:
   ⟨*iteration_statement*⟩.*T* =

```
{
  loop:
        ⟨ expression ⟩ .TF
        ⟨statement⟩ .T
        j loop
  else:
}
```

4. '**do** ... **while**' statement:
   Syntax:
   ⟨*iteration_statement*⟩ ::= '`do`' ⟨*statement*⟩ '`while`' '`(`' ⟨*expression*⟩ '`)`' '`;`'
   Semantics:
   ⟨*iteration_statement*⟩.*T* =

```
{
  then:
        ⟨statement⟩ .T
        ⟨ expression ⟩ .TT
}
```

5. '**for**' statement:
   Syntax:
   ⟨*iteration_statement*⟩ ::= '`for`' '`(`' ⟨*optional_expression-1*⟩ '`;`' ⟨*optional_expression-2*⟩ '`;`' ⟨*optional_expression-3*⟩ '`)`' ⟨*statement*⟩
   Semantics:
   ⟨*iteration_statement*⟩.*T* =

```
{
        ⟨optional_expression−1⟩ .T
  loop: ⟨optional_expression−2⟩ .TF
        ⟨statement⟩ .T
```

```
            ⟨optional_expression−3⟩.T
            j    loop
  else:
}
```

6. logical 'and' operator:
   Syntax:
   ⟨*logical_and_expression*⟩ ::= ⟨*logical_and_expression-1*⟩ '&&' ⟨*inclusive_or_expression*⟩
   Semantics:

```
⟨logical_and_expression⟩.TF =

⟨logical_and_expression−1⟩.TF
⟨ inclusive_or_expression ⟩.TF
```

```
⟨logical_and_expression⟩.TT =

{
        ⟨logical_and_expression−1⟩.TF
        ⟨ inclusive_or_expression ⟩.TT
  else:
}
```

   ⟨*logical_and_expression*⟩.*R* = free;
   ⟨*logical_and_expression*⟩.*T* = one of the following:

```
{
        ⟨ logical_and_expression ⟩.TF
        mov free, #1
        j    end
 else:  mov free, #0
 end:
}
```

```
{
        ⟨ logical_and_expression ⟩.TT
        mov free, #0
        j    end
 then:  mov free, #1
 end:
}
```

7. logical 'or' operator:
   Syntax:
   ⟨*logical_or_expression*⟩ ::= ⟨*logical_or_expression-1*⟩ '||' ⟨*logical_and_expression*⟩
   Semantics:

```
⟨logical_or_expression⟩.TT =

⟨ logical_or_expression−1 ⟩.TT
⟨ logical_and_expression ⟩.TT
```

```
⟨logical_or_expression⟩.TF =

{
        ⟨ logical_or_expression−1 ⟩.TT
        ⟨ logical_and_expression ⟩.TF
  then:
}
```

   ⟨*logical_or_expression*⟩.*R* = free;
   ⟨*logical_or_expression*⟩.*T* = one of the following:

```
{
        ⟨ logical_or_expression ⟩.TF
        mov free, #1
        j    end
 else:  mov free, #0
 end:
}
```

```
{
        ⟨ logical_or_expression ⟩.TT
        mov free, #0
        j    end
 then:  mov free, #1
 end:
}
```

8. logical 'not' operator:

Syntax:
⟨*unary_expression*⟩ ::= '!' ⟨*cast_expression*⟩
Semantics:
⟨*unary_expression*⟩.*TT* = *invertjump*(⟨*cast_expression*⟩.*TT*);
⟨*unary_expression*⟩.*TF* = *invertjump*(⟨*cast_expression*⟩.*TF*);
⟨*logical_or_expression*⟩.*R* = free;
⟨*unary_expression*⟩.*T* = one of the following:

```
{                                   {
        ⟨ cast_expression ⟩.TF              ⟨ logical_or_expression ⟩.TT
        mov free, #0                        mov free, #1
        j     end                           j     end
  else: mov free, #1                  then: mov free, #0
  end:                                end:
}                                   }
```

9. relational operators;
Syntax:
⟨*re_expression*⟩ ::= ⟨*re_expression-1*⟩ ⟨*re_op*⟩ ⟨*re_expression-2*⟩
where ⟨*re_op*⟩ ::= '==' | '!=' | '>' | '<' | '>=' | '<=' ;
(this is a generalized syntax, see observations below)
Semantics:

```
⟨re_expression⟩.TT =                ⟨re_expression⟩.TF =

⟨re_expression−1⟩.T!                ⟨re_expression−1⟩.T!
⟨re_expression−2⟩.T!                ⟨re_expression−2⟩.T!
cmp   ⟨re_expression−1⟩.R!,         cmp   ⟨re_expression−1⟩.R!,
      ⟨re_expression−2⟩.R!                ⟨re_expression−2⟩.R!
j??   then                          j??   else
```

where 'j??' is a placeholder for a jump instruction chosen in the following table:

	'=='	'!='	'>'	'<'	'>='	'<='
TT	jeq	jne	jgt	jlt	jge	jle
TF	jne	jeq	jle	jge	jlt	jgt

⟨*re_expression*⟩.*R* = free;
⟨*re_expression*⟩.*T* = one of the following:

```
{                                   {
        ⟨ re_expression ⟩.TF                ⟨ re_expression ⟩.TT
        mov free, #1                        mov free, #0
        j     end                           j     end
  else: mov free, #0                  then: mov free, #1
  end:                                end:
}                                   }
```

10. comma expression:
⟨*expression*⟩ ::= ⟨*expression-1*⟩ ',' ⟨*assignment_expression*⟩
Semantics:
⟨*expression*⟩.*R* = ⟨*assignment_expression*⟩.*R*
⟨*expression*⟩.*T* =

⟨*expression−1*⟩.*T*
⟨*assignment_expression*⟩.*T*

⟨*expression*⟩.*TT* =	⟨*expression*⟩.*TF* =
⟨*expression−1*⟩.*T* ⟨*assignment_expression*⟩.*TT*	⟨*expression−1*⟩.*T* ⟨*assignment_expression*⟩.*TF*

11. assignment:
⟨*assignment_expression*⟩ ::= ⟨*unary_expression*⟩ '=' ⟨*assignment_expression-1*⟩
Semantics:
⟨*assignment_expression*⟩.*R* = ⟨*unary_expression*⟩.*R*
⟨*assignment_expression*⟩.*T* =

    - if ⟨*assignment_expression-1*⟩.*R* = free;

      ⟨*unary_expression*⟩.*T*
      replace(⟨*assignment_expression−1*⟩.*T*, **free**, ⟨*unary_expression*⟩.*R*)

    - else:

      ⟨*unary_expression*⟩.*T*
      ⟨*assignment_expression−1*⟩.*T*
      mov ⟨*unary_expression*⟩.*R*, ⟨*assignment_expression−1*⟩.*R*

⟨*assignment_expression*⟩.*TT* =	⟨*assignment_expression*⟩.*TF* =
⟨*assignment_expression*⟩.*T* **jne** then	⟨*assignment_expression*⟩.*T* **jeq** else

(Note that I have defined the *TT* and *TF* attributes in terms of *T* for brevity, which is perfectly legitimate.)

12. arithmetic expressions:
⟨*m_expression*⟩ ::= ⟨*m_expression-1*⟩ ⟨*m_op*⟩ ⟨*m_expression-2*⟩
where ⟨*m_op*⟩ ::= '&' | '|' | '«' | '»' | '+' | '−' | ... ;
(this is generalized syntax, see observations below)
Semantics:
⟨*m_expression*⟩.*R* = *free*;
⟨*m_expression*⟩.*T* =

    - if ⟨*m_expression-1*⟩.*R* ≠ free ∧⟨*m_expression-2*⟩.*R* ≠ free:

      ⟨*m_expression−1*⟩.*T!*
      ⟨*m_expression−2*⟩.*T!*
      ? **free**, ⟨*m_expression−1*⟩.*R!*, ⟨*m_expression−2*⟩.*R!*

⟨*m_expression*⟩.*TT* =	⟨*m_expression*⟩.*TF* =
⟨*m_expression*⟩.*T* **jeq** then	⟨*m_expression*⟩.*T* **jne** else

where '?' is a placeholder for an instruction chosen in the following table:

'&'	'|'	'<<'	'>>'	'+'	'−'	'*'	...
and	or	shl	shr	add	sub	mul	...

13. identifiers:
    Semantics:
    $IDENTIFIER.T = (empty)$;
    $IDENTIFIER.R = IDENTIFIER$;

    $IDENTIFIER.TF =$

    ```
    cmp  IDENTIFIER, #0
    jeq  else
    ```

    $IDENTIFIER.TT =$

    ```
    cmp  IDENTIFIER, #0
    jne  then
    ```

14. expression statements;
    ⟨*expression_statement*⟩ ::= ⟨*expression*⟩ ';'
    Semantics:
    ⟨*expression_statement*⟩.*T* = the least expensive among ⟨*expression*⟩.*T*, ⟨*expression*⟩.*TF* and ⟨*expression*⟩.*TT*.

15. copy rules:
    Syntax:
    any copy rule in Section 4.5 (page 231);
    Semantics:
    attributes of the left-hand side non-terminal assume the same values as in the right-hand side non-terminal.

4.3.5 Observations

As far as the logical 'not' operator is concerned, note how its double-exit translations have no added cost with respect to the double-exit translations of its operand. The following example illustrates the case:

```
if (a) ... :

{
        cmp a, #0       ; IDENTIFIER.TF
        jeq else        ;
        ⟨statement⟩.T
 else:
}
```

```
if (!a) ... :

{
        cmp a, #0       ; IDENTIFIER.TF
        jne else        ;
        ⟨statement⟩.T
 else:
}
```

Because of short-circuit evaluation, all the translations of logical and operator '&&' employ the TF translation of their first operand. Similarly, all the translations of logical or operator '||' employ the TT translation of their first operand. Depending on the translation flavor of the logical expression, the TT or the TF flavor of the second operand is used. The above considerations are illustrated in all the examples which follow.

In certain cases, more alternatives are possible for the same translation flavor. This is indicated by sentences like "... = one of the following: ...". In those cases, the compiler can equivalently translate the node in multiple

ways. Without other knowledge available, the two ways are equivalent, and which is chosen is indeterminate. The flavor attribute evaluation rules will take this indetermination into account. The example in Figure 4.6 (page 125) shows how the two TF and TT flavors of a logical 'or' expression contribute to creating two possible alternative for the T flavor of an 'if' statement. Without profiles or jump probability estimates, both alternatives are meaningful and could be selected. The cost model I propose will have to be aware of this indetermination.

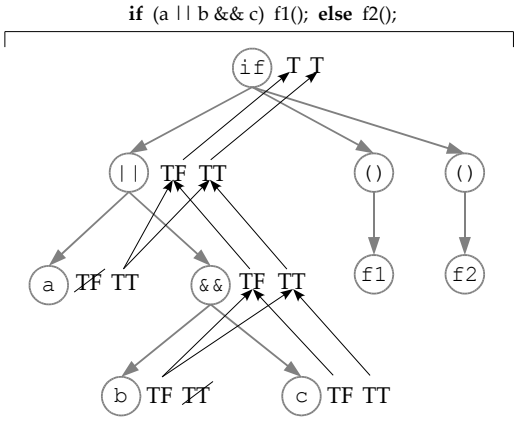


Figure 4.6: Multiple alternatives could be possible for the same translation flavor.

4.3.6 Examples

The first example shows how complex logical expressions are translated. The example considers the following C statement:

if ((a&&b || c || d&&e) && f) f1() **else** f2();

Its abstract syntax tree is reported in Figure 4.7. In this AST, each node is accompanied with a label which indicates which translation flavor appears in the final translation, consistently with the above observations. Figure 4.8 reports graphically how each AST node maps to its respective translation in the abstract assembly translation of the entire statement.

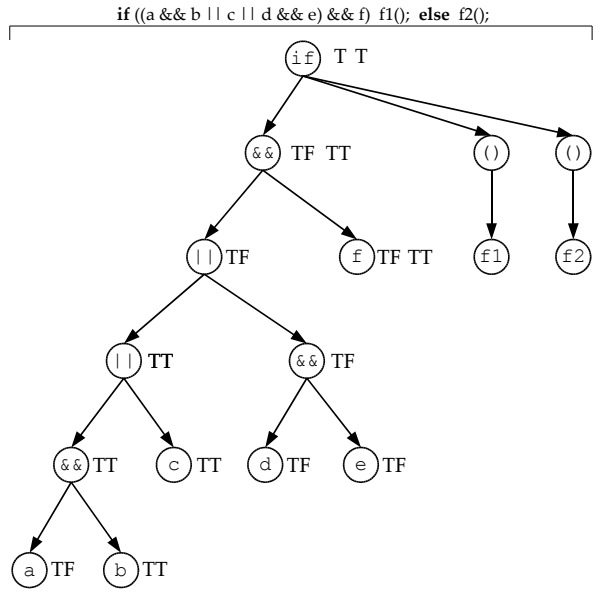


Figure 4.7: The parse tree for an 'if' statement involving a complex logical expression. For each node, the flavor which actually appears in the translation is shown. Two T's are annotated next to the 'if' node to indicate that two distinct translations are possible.

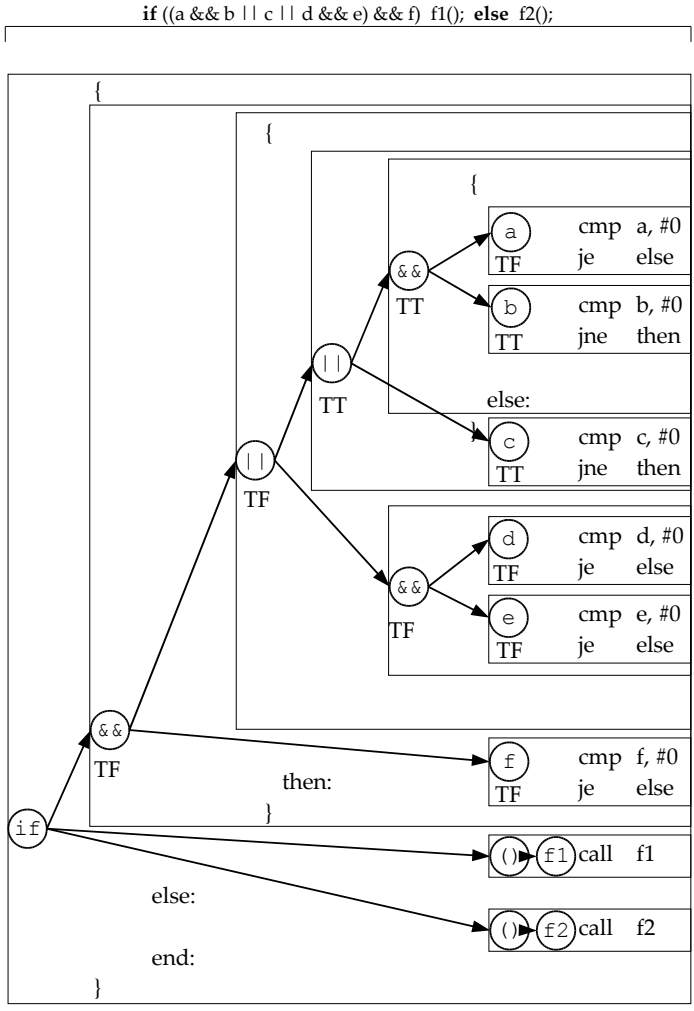


Figure 4.8: Example illustrating the application of the abstract translation scheme to a sample statement.

The following example shows how the translation scheme works on a more complex statement. In addition to the above issues, the example also illustrates relational expressions and assignments, both with register-bound and register-unbound translations. The example considers the following C statement:

```
if (( a && (b < c+d) || e || g&&(h || i) ) && j)
      d = (a == b+c);
else
      g = e = f << 2;
```

Figure 4.9 reports the AST of the statement, where nodes have been numbered from N_1 to N_32 in a post-order visit for clarification purposes. The corresponding translation for each of the same nodes, in the same order, follows below. When multiple translations for a given symbol were possible, I made an arbitrary choice and reported just one of them.

- *N*1.*TF* =

  cmp a, #0
  jeq else

- *N*2.*T* = (empty);
  *N*2.*R* = b;

- *N*3.*T* = (empty);
  *N*3.*R* = c;

- *N*4.*T* = (empty);
  *N*4.*R* = d;

- *N*5.*T* =

	=	**add free**, c, d
N3.T! *N4.T!* ? **free**, *N3.R!*, *N4.R!*		

  *N*5.*R* = free;

- *N*6.*TT* =

	=	**add** R5, c, d **cmp** b, R5 **jlt** then
N2.T! *N5.T!* **cmp** *N2.R!*, *N5.R!* **j**?? then		

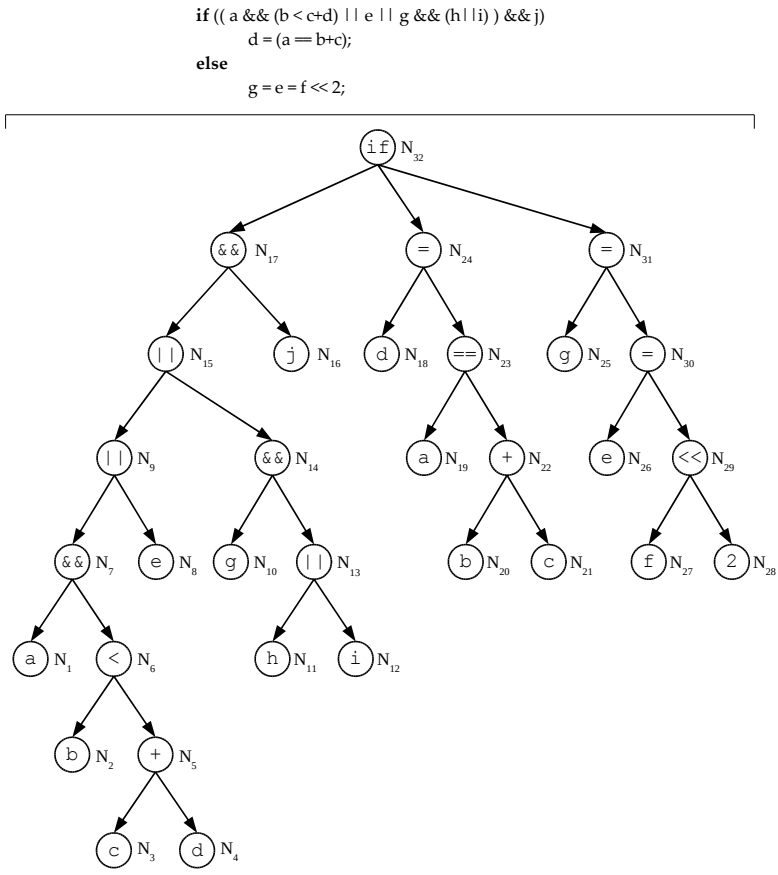


Figure 4.9: Abstract syntax tree of a more complex example statement, used to illustrate the abstract translation scheme.

- *N7.TT* =

```
{
        N1.TF
        N6.TT
  else:
}
```

=

```
{
        cmp   a, #0
        jeq   else
        add   R5, c, d
        cmp   b, R5
        jlt   then
  else:
}
```

- *N*8.*TT* =

```
cmp  e, #0
jne  then
```

- *N*9.*TT* =

```
N7.TT
N8.TT
```

=

```
{
        cmp   a, #0
        jeq   else
        add   R5, c, d
        cmp   b, R5
        jlt   then
  else:
}
cmp  e, #0
jne  then
```

- *N*10.*TF* =

```
cmp  g, #0
jeq  else
```

- *N*11.*TT* =

```
cmp  h, #0
jne  then
```

- *N*12.*TF* =

```
cmp  i, #0
jeq  else
```

- *N*13.*TF* =

```
{
        N11.TT
        N12.TF
  then:
}
```

=

```
{
        cmp   h, #0
        jne   then
        cmp   i, #0
        jeq   else
  then:
}
```

- *N*14.*TF* =

```
N10.TF
N13.TF
```

=

```
cmp  g, #0
jeq  else
{
        cmp   h, #0
        jne   then
        cmp   i, #0
        jeq   else
  then:
}
```

- *N*15.*TF* =

```
{
        N9.TT
        N14.TF
  then:
}
```

=

```
{
    {
            cmp   a, #0
            jeq   else
            add   R5, c, d
            cmp   b, R5
            jlt   then
      else:
    }
    cmp  e, #0
    jne  then
    cmp  g, #0
    jeq  else
    {
            cmp   h, #0
            jne   then
            cmp   i, #0
            jeq   else
      then:
    }
  then:
}
```

- *N*16.*TF* =

```
cmp   j, #0
jeq   else
```

- *N*17.*TF* (chosen arbitrarily) =

  N15.TF
  N16.TF

  =

```
{
        {
                cmp   a, #0
                jeq   else
                add   R5, c, d
                cmp   b, R5
                jlt   then
          else:
        }
        cmp  e, #0
        jne  then
        cmp  g, #0
        jeq  else
        {
                cmp   h, #0
                jne   then
                cmp   i, #0
                jeq   else
          then:
        }
  then:
}
cmp   j, #0
jeq   else
```

- *N*18.*T* = (empty);
  *N*18.*R* = d;

- *N*19.*T* = (empty);
  *N*19.*R* = a;

- *N*20.*T* = (empty);
  *N*20.*R* = b;

- *N*21.*T* = (empty);
  *N*21.*R* = c;

- *N*22.*T* =

  N20.T!
  N21.T!
  ? **free**, *N20.R!*, *N21.R!*

  =

```
add free, b, c
```

  *N*22.*R* = free;

- *N*23.*T* =

(chosen arbitrarily *T* including *N*23.*TF*)	=	{
{		**add** R22, b, c
N19.T!		**cmp** a, R22
N22.T!		**jne** else
cmp *N19.R!*,*N22.R!*		**mov** **free**, #1
j?? else		**j** end
mov **free**, #1		else: **mov** **free**, #0
j end		end:
else: **mov** **free**, #0		}
end:		
}		

*N*23.*R* = free;

- *N*24.*T* =

	=	{
N18.T		**add** R22, b, c
replace(*N23.T*, **free**, *N18.R*)		**cmp** a, R22
		jne else
		mov d, #1
		j end
		else: **mov** d, #0
		end:
		}

- *N*25.*T* = (empty);
  *N*25.*R* = g;
- *N*26.*T* = (empty);
  *N*26.*R* = e;
- *N*27.*T* = (empty);
  *N*27.*R* = f;
- *N*28.*T* = (empty);
  *N*28.*R* = #2;

- *N*29.*T* =

	=	**shl** **free**, f, #2
N27.T!		
N28.T!		
? **free**, *N27.R!*, *N28.R!*		

*N*29.*R* = free;

- *N*30.*T* =

	=	**shl** e, f, #2
N26.T		
replace(*N29.T*, **free**, *N26.R*)		

*N*30.*R* = e;

- *N*31.*T* =

	=	
N25.T		**shl** e, f, #2
N30.T		**mov** g, e
mov *N25.R*,*N30.R*		

- *N32.T* =

(chosen arbitrarily) =

```
{
        N17.TF
        N24.T
        j end
  else: N31.T
  end:
}
```

```
{
        {
                {
                        cmp   a, #0
                        jeq   else
                        add   R5, c, d
                        cmp   b, R5
                        jlt   then
                 else:
                }
                cmp  e, #0
                jne  then
                cmp  g, #0
                jeq  else
                {
                        cmp   h, #0
                        jne   then
                        cmp   i, #0
                        jeq   else
                 then:
                }
         then:
        }
        cmp   j, #0
        jeq   else
        {
                add  R22, b, c
                cmp a, R22
                jne  else
                mov d, #1
                j    end
         else:  mov d, #0
         end:
        }
        j end
  else:
        shl  e, f, #2
        mov g, e
  end:
}
```

The following fragment illustrates the same code as above, rendered in 'flat' assembly language, i.e. without nested block. The task of translating assembly code with block nesting into flat assembly is rather straightforward and just requires a simple label renaming mechanism which I will not discuss here.

```
        cmp   a, #0
        jeq   else1
        add   R5, c, d
        cmp   b, R5
        jlt   then2
else1:
        cmp   e, #0
        jne   then2
        cmp   g, #0
        jeq   else3
        cmp   h, #0
        jne   then1
        cmp   i, #0
        jeq   else3
then1:
then2:
        cmp   j, #0
        jeq   else3
        add   R22, b, c
        cmp   a, R22
        jne   else2
        mov   d, #1
        j     end1
else2:  mov   d, #0
end1:
        j     end2
else3:
        shl   e, f, #2
        mov   g, e
end2:
```

4.4 An attribute grammar to determine the cost of syntax elements

This section describes an algorithm to attribute a single-execution cost to each syntax element of a C program (i.e., each nodes in its abstract syntax tree). The algorithm is consistent with the abstract translation model of the previous section, and takes into account all the factors which impact on cost, presented in Section 4.2 (page 110).

I describe this algorithm in the form of a multi-visit attribute grammar, operating on the abstract syntax tree, with the following 11 attributes:

- attribute t, 'real result type':
  this attribute represents the type of an expression. It is a synthesized attribute. It can assume values whose meaning is "pointer to integer" or "array of functions returning double", expressed in an appropriate representation. It affects the conversion cost. For a node N, $N.t$ depends on t of its child nodes. I will discuss an appropriate representation for types, and all the rules to calculate t in Section 4.4.1 (page 140);

- attribute r, 'restricted type':
  this attribute represents the type of the data which are actually involved in the computation or a transfer. The value of r can be different from t when the dot operator is involved. It affects the inherent cost. It can assume the same set of values as t can. It is an inherited attribute and it depends on t: for a node N, $N.r$ depends on $N.t$. I will discuss the reasons why this attribute is needed and how to derive it in Section 4.4.2 (page 158);

- attribute e, 'constant value':
  when an expression has a constant value, I define attribute e to hold that value. e is a synthesized attribute and can assume any integer or floating-point value. For terminal symbols, e depends on their lexical value (e.g. the value of literal constant '4' is 4). Because of the '**sizeof**' operator, e may depend on t. I will discuss e in Section 4.4.3 (page 169);

- attribute k, 'constancy':
  this is a boolean attribute, which tells whether the expression assumes a constant value or not. It is a synthesized attribute. It affects all the costs. Because of conditional and logical expressions, k may depends on e. For a given node, k depends on k and e of its child nodes. I will discuss k in Section 4.4.3 (page 169);

- attribute v, 'valueness:
  this attribute is also defined for expressions only; it and assumes the values R, L, RL and Z as introduced in the previous section. It is an inherited attribute. It affects inherent costs. Attribute v of a node depends entirely on v of its father node. It does not depend on any other attributes. I discuss its evaluation in Section 4.4.4 (page 176);

- attribute b, 'register-boundedness':
  this is a boolean attribute, which tells whether the expression is bound to a register or not. It is a synthesized attribute. It affects the inherent cost, in detail of assignment operators. For a given node, b depends on the values of b of its child nodes. I will discuss b in Section 4.4.5 (page 178);

- attribute f, 'translation flavor':
  this attribute is defined for expressions and statements. It can assume the value T, TF, TT, or indeterminate. It is an inherited attribute, and f of a node depends on f of its father. f has already been introduced in Section 4.2.5 (page 115). I will summarize its evaluation rules in Section 4.4.6 (page 180);

- attribute ci, 'inherent cost':
  this attribute expresses the single-execution cost of carrying out the data manipulation and data-transfer operations expressed by the semantics of a node. It is defined for expressions and statements. It is a synthesized attribute, and depends on a number of attributes in the same node. It also depends on k, e of the children, to model certain strength reduction optimization. Like any other cost, I express ci as a summation of atoms. I will summarize its evaluation rules in Section 4.4.7 (page 183);

- attribute cc, 'conversion cost':
  this attribute expresses the single-execution cost of carrying out the implicit or explicit data conversion operations prescribed by the C language for a given node, depending on where it appears. It is defined for expressions only. I express this cost as a summation of atoms. It is an synthesized attribute; $N.cc$ depends in general on t of its children. I will summarize its evaluation rules in Section 4.4.8 (page 216);

- attribute cf, 'control flow cost':
  this attribute expresses the single-execution cost of determining whether to transfer the control flow to another point in the program, and possibly transferring it when needed. It is defined for statements. It is an inherited attribute; $N.cf$ depends in general on $N.f$ and on the syntax of its father. I will summarize its evaluation rules in Section 4.4.9 (page 225);

- attribute c, 'cost':
  this attribute is the single-execution cost which is the final goal of this entire chapter. It is the summation of all the costs which affect a node. It is a synthesized attribute. $N.c$ is evaluated by summing together whichever of the $N.ci$, $N.cc$ and $N.cf$ are defined. I will define its evaluation rules in Section 4.4.10 (page 230);

The table below summarizes the attributes just introduced:

Attribute		Name	Defined for
t	synthesized	result type	expressions
r	inherited	restricted result type	expressions
e	synthesized	constant result value	expressions
k	synthesized	constancy	expressions
v	inherited	valueness	expressions
b	synthesized	register-boundedness	expressions
f	inherited	translation flavor	expressions and statements
ci	synthesized	inherent cost	expressions and statements
cc	synthesized	conversion cost	expressions
cf	inherited	flow control cost	statements
c	synthesized	total cost	expressions and statements

Finally, I introduce an attribute n ('name') which is defined for terminal symbols only. Its value is the lexical value of the symbol itself.

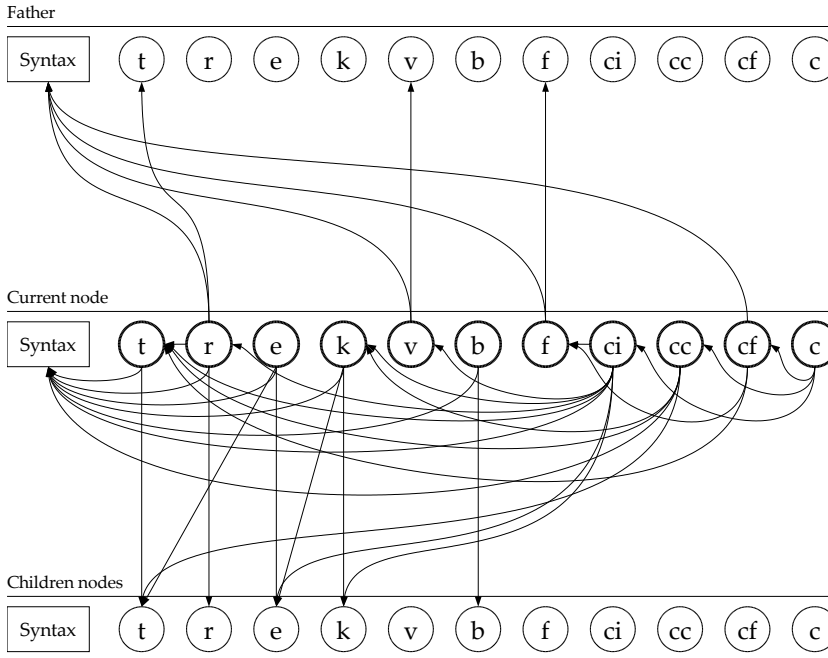


Figure 4.10: Dependences between attributes.

A complete representation of the dependences between attributes is given in Figure 4.10. All the edges start from a node x, representing some attribute x of the current symbol, and end in a node y, representing an attribute which may belong to the same symbol, to the father of the current symbol, or to one of its children. An edge going from attribute $N.x$ to $M.y$

implies that attribute $M.y$ must be evaluated before $N.x$.

The attributes can be evaluated in any order that respects their mutual dependences. As far as the actual implementation of an evaluation algorithm for this grammar is concerned, the order in which attributes are evaluated and the number of visits of the AST required to evaluate them are important, especially for realistic-size inputs. Any further discussion on the topic is beyond the scope of this document. The readers interested to learn more on this are invited to refer to the source code of the project which implements this thesis.

4.4.1 Attribute 't', result type

Simply said, attribute t of an expression is the type of that expression, according to the rules of the C language. The type of an expression can be determined unambiguously on the basis of variable declarations in the current scope, operator semantics, and type assumption and conversion rules of the C language. All these are described unambiguously in the C standard [81]. The purpose of this section is to provide a set of non-ambiguous rules which allow to determine the type of any AST node according to those rules.

Note: the reader should avoid confusion between the *result type* (attribute t) and the *restricted result type* (attribute r) which is introduced in Section 4.4.2 (page 158).

Intuitively, the needed type evaluation rules are those who evaluate the type of `a.m` to **int** if `a` was declared as **struct** { ...; **int** m; ...} a;, the type of `*p` as **char** if `p` was declared as **char** * p; or as **char** p[N], and the type of `sin(a)` as **double** if `sin` was declared as **double** sin(**double** a), and so on.

The reason why t is a synthesized attribute depending on the value of t in the child nodes of the current one is that the standard defines the type of an expression in terms of the types of its subexpressions. Therefore, all the evaluation rules of this section express $N.t$ as a function of t of the child nodes of N.

In order to express this rules easily, I need an appropriate representation for t. The very C declarations as they appear in the code are not a good candidate for this purpose, because they are difficult to manipulate, and the following example shows why. Imagine that you want to determine the type of expression '`*strchr(s,c)`', where t of subexpression 'strchr' is represented as the following string:

```
char *strchr(const char *s, int c);
```

First you need to express a general rule such that the evaluated result type of the '`()`' sub expression is the string:

```
char *
```

then, that operator '`*`' over this last type yields the string:

```
char
```

Describing the above evaluation in generic terms with this representation is cumbersome for the trivial example above. Things get much worse when real-life declarations are considered. The type system of the C language is rich and complex. Approximately one third of the entire grammar of the C language (i.e. approximately 70 syntax rules out of 213) is devoted the structure of declarations. The syntax of several C declaration is rather complex and counterintuitive, an even experienced programmers find it troublesome (see [83], Chapter 3). Qualifiers (**const** and **volatile**) may appear in different orders, type specifiers like **long** or **unsigned** may modify another type specifier or come alone (then **int** is implied). Declarations of pointers to functions are poorly readable. And much more. A real-life example of a non-trivial declaration is:

```
void (*signal(int sig, void (*func)(int)) ) (int);
```

Definitely, manipulating the above type representation is impractical.

I propose an alternative, stack-based representation, such that the type of strchr would look like:
[function][pointer][char]
This way, the type of a '()' operator expression is simply the type of its first operand without the initial '[function]', i.e.:
[pointer][char]
and the type of a unary '$*$' operator expression is the type of its operand deprived from its initial '[pointer]', i.e.:
[char].
The remainder of this chapter is devoted to detail these simple ideas.

In the context of this thesis, a type is a **normalized stack of type-definition records**. The next paragraphs explain what type-definition records are and what normalization is.

I assume that notion of stack is intuitive and given. I represent each element of a stack in square bracket, such as '[a]', and I denote the contents of a stack by listing its element from the top of the stack down to the bottom. For example, the expression '[a][b][c]' denotes a stack containing three elements 'a','b' and 'c', such that 'a' is the top of the stack and 'c' is the bottom. Given a stack s and an element e, I define the usual stack operations in the following way:

- $top(s)$
  returns element e if e is the top of stack s;
- $pop(s)$
  returns a stack where the top element from s was removed;
- $push(e, s)$
  returns a new stack t such that $top(t) = e$ and $pop(t) = s$.

For a non-empty stack s, the following property holds: $push(top(s), pop(s)) = s$. The definitions given for top, pop and push are isomorphic to 'first', 'rest' and 'cons' primitives in the Common Lisp programming language. I will use often stack operators when defining the evaluation rules for t.

I will not explain in detail how to design semantic actions which implement the declaration semantics of the C language. This detail level is beyond the scope of this document. I assume that a set of semantic actions is available, associated to the syntax rules which govern declarations in the C language. This rules will set attribute t of identifiers, constants, string literals and $\langle type\_name \rangle$ in compliance with the above explanations, by stacking type-definition records.

The type-definition records (type records, for short) are the following: [char], [long], [int], [short], [double], [signed], [unsigned], [float], [const], [volatile], [array], [function], [enum], [struct], [union], [pointer], [void], [symbol], [user type]. Type records may have attributes like: the number of elements for an array, qualifiers, specifiers like **long** or **signed**, a symbol

table representing the members for unions and structs, a symbol table representing the arguments for functions. Attributes are denoted as subscripts, e.g. [int_u] denotes an integer without sign, and [pointer_c] a constant pointer.

The above type stacks are not normalized. Normalization is an operation in which:

- records corresponding to qualifiers (**const** and **volatile**) are removed from the stack and applied to their associated type record, e.g. a [int][const] becomes a [int_c];
- records which modify other records are merged with the records they modify, e.g. a [int][unsigned] becomes [int_u];
- the same records as above, when used alone, are merged with an implied int, e.g. a [long] becomes [int_l] and a [const] becomes an [int_c];
- user types are expanded with their definitions.

Normalization is required to allow proper type comparison, which is needed in this context to determine conversion costs.

The following example illustrate C declarations and their associated non-normalized and normalized type stack. For the help of the reader, I also report their natural language description.

- **long int** quot;
  [int] [long]
  [int_l]
  "quot is a long integer"

- **const char** $*$ format;
  [pointer] [const] [char]
  [pointer] [char_c]
  "format is a pointer to a constant character"

- FILE $*$ file ;
  [pointer] [user type FILE]
  [pointer] [struct _IO_FILE]
  "file is a pointer to an object of user type FILE"

- **void const** $*$ s;
  [pointer] [const] [void]
  [pointer] [void_c]
  "s is a pointer to a constant, untyped memory location"

- **unsigned short int** a[3];
  [array[3]] [int] [short] [unsigned]
  [array[3]] [int_{us}]
  "a is an array of three short unsigned integers"

- **const char** $*$**const** sys_errlist [];
  [array[]] [const] [pointer] [const] [char]
  [array[]] [pointer_c] [char_c]
  "sys_err_list is an unsized array of constant pointers to constant characters"

- **int** (∗__compar_fn_t) (**const void** ∗, **const void** ∗);
  [pointer] [function $_{@0x8177f04}$] [int]
  [pointer] [function $_{@0x8177f04}$] [int]
  "__compar_fn_t is a pointer to a function returning an integer; the arguments are stored in a separate symbol table, located at @0x8177f04"

- **unsigned short int** ∗seed48 (**unsigned short int** seed16v[3]);
  [function @0x815ca94] [pointer] [int_{us}]
  "seed48 is function returning a pointer to a short unsigned integer"

- (where seed16v in symbol table 0x815ca94 is:)
  [array[3]] [int] [short] [unsigned]
  [array[3]] [int_{us}]
  "the function takes as a parameter an array of short unsigned integers"

As far as the calculation of attribute t for each possible expression is concerned, the operators of the C language fall in a number of classes, such that all the operators in the same class share the semantic rules to calculate t.

I summarize these classes in Table 4.2, and discuss them individually below. The table presents an enumerated row for each class: each row indicates the arity of the operators which are part of that class, then a name for the class, the complete list of operators which belong to it, and a short description of how the operators behave as far as the result type is concerned, expressed in natural language.

Class	Arity	Informal description, members and behavior
1	1	The 'sizeof' operator '**sizeof**' Behavior: return type is an unsigned integral type.
2	1	Integer-type unary operators '!' Behavior: return type is [int].
3	1	Operand-type unary operators prefix or postfix '++', prefix or postfix '−−' Behavior: return type is the same as the operand.
4	1	Integral promotion operators '+', '−', '~' Behavior: return type is the *integer promotion* of the type of the operand.
5	1	The referencing operator '&' Behavior: return type is pointer to the type of the operand.
6	1	The dereferencing operator '∗' Behavior: return type same of the operand except for the initial [pointer] or [array] element
7	2	The cast operator '(type)' Behavior: return type is the first operand.
8	2	Integer-type binary operators '==', '!=', '<', '>', '<=', '>=', '&&', '||' Behavior: return type is [int].
9	2	First-operand type binary operators '=', '<<', '>>', '+=', '−=', '∗=', '/=', '%=', '&=', '|=', '^=', '<<=', '>>=' Behavior: return type is the same as first operand.
10	2	Second-operand type binary operators ',' Behavior: return type is the same as second operand.
11	2	The arithmetic binary operators '+', '−', '∗', '/', '%', '&', '|', '^' Behavior: return type determined via *usual arithmetic conversions* and *pointer arithmetic*, as in the C standard.
12	2	Access operators '.', '−>', '[]' Behavior: return type is as declared.
13	3	Conditional operator '?:' Behavior: return type is the same as the type of value of at most 2 of the 3 child expressions.
14	1..n	Function call function call ' ...(...) ' Behavior: result type is as declared.

Table 4.2: The operators of the C language, classified on the basis of how their result type (attribute *t*) is determined.

4.4.1.1 The 'sizeof' operator

The '**sizeof**' operator returns the number of bytes occupied by its operand, which may be an expression or a parenthesized name of a type. According to the C standard [81], Section 6.3.3.4, the result value of the '**sizeof**' operator «is implementation-defined, and its type (an unsigned integral type) is 'size_t' defined in the '<stddef.h>' header». It is therefore legitimate to associate to a '**sizeof**' expression a type $[\text{int}_u]$.

Syntax:

⟨*unary_expression*⟩ ::= 'sizeof' ⟨*unary_expression-1*⟩
| 'sizeof' '(' ⟨*type_name*⟩ ')'

Semantics:

⟨*unary_expression*⟩.*t* = $[\text{int}_u]$;

4.4.1.2 Integer-type unary operators

Despite the plural, this class contains only the '!' operator. According to the standard, this operator returns an integer value.

Syntax:

⟨*unary_expression*⟩ ::= '!' ⟨*cast_expression*⟩

Semantics:

⟨*unary_expression*⟩.*t* = [int] ;

4.4.1.3 Operand-type unary operators

As specified in Sections 6.3.2.4 and 6.3.3.1 of the C standard, operators in this class return the same type as their operand, and operands must be scalar (i.e. integral or floating-point).

Syntax:

⟨*unary_expression*⟩ ::= '++' ⟨*unary_expression-1*⟩
| '--' ⟨*unary_expression-1*⟩

⟨*postfix_expression*⟩ ::= ⟨*postfix_expression-1*⟩ '++'
| ⟨*postfix_expression-1*⟩ '--'

Generalized syntax:

⟨*father_expression*⟩ ::= ⟨*child_expression*⟩ ⟨*auto_op*⟩
| ⟨*auto_op*⟩ ⟨*child_expression*⟩

⟨*auto_op*⟩ ::= '++' | '--'

Semantics:

⟨*father_expression*⟩.*t* = ⟨*child_expression*⟩.*t*

Note: expression '++i' is not equivalent to 'i= i + 1'. In fact, the unary operator '++' does not perform integer type promotion, whereas binary operator '+' does. I report further details on this technicality in Section 4.4.1.11 (page 149).

4.4.1.4 Integral promotion operators

Operators in this class yield the same type as their operand, except they perform *integral promotion* when required. According to the standard, «A **char**, a **short int** or an **int** bit-field, or their signed and unsigned varieties, or an enumeration type, may be used in an expression wherever an **int** or **unsigned int** may be used. If an **int** can represent all the values of the original type, the value is converted to an **int**; otherwise it is converted to an **unsigned int**. These are called the *integral promotions*. All other arithmetic types are unchanged by the integral promotions».

If t is a type, I denote with $I(t)$ its integral promotion. The above constraints are strict enough to indicate a unique $I(\cdot)$, provided that the size and representation of each basic type is chosen. The following examples should be enough for the user to understand how types are integer-promoted:

$$\begin{array}{lcl} I([\text{char}]) & = & [\text{int}]; \\ I([\text{int}_s]) & = & [\text{int}]; \\ I([\text{int}]) & = & [\text{int}]; \\ I([\text{int}_l]) & = & [\text{int}_l]; \\ I([\text{float}]) & = & [\text{float}]; \\ I([\text{double}]) & = & [\text{double}]. \end{array}$$

The cost of integer promotion, under my architectural assumptions is null except for the integer promotion of [char] to [int], whose cost is one IntToInt atom.

Syntax:

$\langle unary\_expression \rangle$ ::= '+' $\langle cast\_expression \rangle$
| '−' $\langle cast\_expression \rangle$
| '~' $\langle cast\_expression \rangle$

Semantics:

$$\langle unary\_expression \rangle.t = I(F(\langle cast\_expression \rangle.t))$$

4.4.1.5 The referencing operator

According to the standard, «the result of the unary '&' operator is a pointer to the object or function designated by its operand. If the operand has type "type", the result has type "pointer to type"» ([81], Section 6.3.3.2). In my type representation, this means that the type of '& expr' is obtained by pushing [pointer] onto the type of 'expr'.

The above sentence describes completely the behavior of the '&' operator, also in the case where the argument is a function designator. In fact, other operators perform the *function designator conversion* on the type of their operands, thus implicitly converting "function returning *type*" types to

"pointer to function returning *type*", i.e. taking the address of the function. There is no need to do that here, since the address of the function is taken explicitly.

Syntax:

⟨*unary_expression*⟩ ::= '&' ⟨*cast_expression*⟩

Semantics:

$$\langle unary\_expression \rangle.t = push([\text{pointer}], \langle cast\_expression \rangle.t)$$

Please note that this semantic action depends directly on attribute t of the children node, unlike the actions of most other operators, which depend on $F(t)$. This is due to the peculiarity of the '**sizeof**' and '&' operators, which prevents function designator conversions. For a complete discussion on the topic, see Section 4.4.1.14 (page 153).

4.4.1.6 The dereferencing operator

As the dereferencing operator is concerned: «if the operand points to a function, the result is a function designator; if it points to an object, the result is an lvalue designating the object. If the operand has type "pointer to type", the result has type "type"».

In my representation, the unary star pops the first type-definition record from the type stack of the operand, when this operator is either [pointer] or [array] (in fact, array names used without subscripting must be considered as pointers).

Please note that, in case the operand is a function designator, the automatic conversion from type "function returning *type*" to type "pointer to function returning *type*" is performed via function $F(\cdot)$, according to the behavior described in Section 4.4.1.14 (page 153). Therefore if attribute t of ⟨*cast_expression*⟩ (according to the syntax rule below) is in the form "[function]...", then $F(t)$ is in the form "[pointer][function]...". The final result type is $pop(F(t))$, therefore "[function]..." again.

Syntax:

⟨*unary_expression*⟩ ::= '*' ⟨*cast_expression*⟩ ;

Semantics:

$$\langle unary\_expression \rangle.t = pop(F(\langle cast\_expression \rangle.t))$$

Please note that the following condition must hold: $top(F(\langle cast\_expression \rangle.t)) \in \{[\text{pointer}], [\text{array}]\}$. If it does not, then the source code contains type errors.

4.4.1.7 The cast operator

Syntax:

⟨*cast_expression*⟩ ::= '(' ⟨*type_name*⟩ ')' ⟨*cast_expression-1*⟩

Semantics:

⟨*cast_expression*⟩.*t* = value of ⟨*type_name*⟩;

Note that ⟨*type_name*⟩ is neither an expression nor a statement symbol. Its value must be a type stack representing the corresponding type. Discussing how to determine this type stack while parsing is beyond the scope of this document.

4.4.1.8 Integer-type binary operators

This class comprises the relational and the logical operators. According to the C standard, all the operators in this class return an integer value. Interested readers should refer to the source code of the tools which implement this thesis.

Syntax:

⟨*equality_expression*⟩ ::= ⟨*equality_expression*⟩ '==' ⟨*relational_expression*⟩
| ⟨*equality_expression*⟩ '!=' ⟨*relational_expression*⟩

⟨*relational_expression*⟩ ::= ⟨*relational_expression*⟩ '<' ⟨*shift_expression*⟩
| ⟨*relational_expression*⟩ '>' ⟨*shift_expression*⟩
| ⟨*relational_expression*⟩ '<=' ⟨*shift_expression*⟩
| ⟨*relational_expression*⟩ '>=' ⟨*shift_expression*⟩

⟨*logical_or_expression*⟩ ::= ⟨*logical_or_expression*⟩ '||' ⟨*logical_and_expression*⟩

⟨*logical_and_expression*⟩ ::= ⟨*logical_and_expression*⟩ '&&' ⟨*inclusive_or_expression*⟩

Generalized syntax:

⟨*father_expression*⟩ ::= ⟨*child_expression-1*⟩ ⟨*operator*⟩ ⟨*child_expression-2*⟩

Semantics:

⟨*father_expression*⟩.*t* = [int] ;

4.4.1.9 First-operand type binary operators

Operators in this class return the same type as their first operand.

Syntax:

⟨*assignment_expression*⟩ ::= ⟨*unary_expression*⟩ '=' ⟨*assignment_expression-1*⟩
| ⟨*unary_expression*⟩ '*=' ⟨*assignment_expression-1*⟩
| ⟨*unary_expression*⟩ '/=' ⟨*assignment_expression-1*⟩
| ⟨*unary_expression*⟩ '%=' ⟨*assignment_expression-1*⟩
| ⟨*unary_expression*⟩ '+=' ⟨*assignment_expression-1*⟩
| ⟨*unary_expression*⟩ '−=' ⟨*assignment_expression-1*⟩
| ⟨*unary_expression*⟩ '«=' ⟨*assignment_expression-1*⟩
| ⟨*unary_expression*⟩ '»=' ⟨*assignment_expression-1*⟩
| ⟨*unary_expression*⟩ '&=' ⟨*assignment_expression-1*⟩
| ⟨*unary_expression*⟩ '|=' ⟨*assignment_expression-1*⟩
| ⟨*unary_expression*⟩ '^=' ⟨*assignment_expression-1*⟩

⟨*shift_expression*⟩ ::= ⟨*additive_expression*⟩
| ⟨*shift_expression*⟩ '«' ⟨*additive_expression*⟩
| ⟨*shift_expression*⟩ '»' ⟨*additive_expression*⟩

Generalized syntax:
⟨*father_expression*⟩ ::= ⟨*child_expression-1*⟩ ⟨*operator*⟩ ⟨*child_expression-2*⟩ ;
Semantics:
⟨*father_expression*⟩.*t* = ⟨*child_expression-1*⟩.*t*;

4.4.1.10 Second-operand type binary operators

Operators in this class return the same type as their second operand.

Syntax:
⟨*expression*⟩ ::= ⟨*expression*⟩ ',' ⟨*assignment_expression*⟩
Generalized syntax (for uniformity with the previous case):
⟨*father_expression*⟩ ::= ⟨*child_expression-1*⟩ operator ⟨*child_expression-2*⟩
Semantics:
⟨*father_expression*⟩.*t* = ⟨*child_expression-2*⟩.*t*;

4.4.1.11 The arithmetic binary operators

For these operators, when only arithmetic-type operands are involved, the result type is determined according to the *usual arithmetic conversions* [ipsissimis verbis], a set of rules which, despite the informal name, is specified rigorously in the C standard, in Section 6.2.1.5.

When arithmetic and pointer types are involved, the result type is determined according *pointer arithmetics*, specified in Section 6.3.6 of the standard.

For sake of simplicity, I define function $U(t_1, t_2)$ that returns the type of the result of the operation between a first operand which has type t_1 and a second operand which has type t_2. Function $U(\cdot, \cdot)$ is also aware of the function designator conversion rule described in Section 6.2.2.1 of the standard and in Section 4.4.1.14 (page 153) of this document.

Function $U(t_1, t_2)$ determines the type of the result of an expression composed by a first operand of type t_1, a binary operator, and a second operand of class t_2, when the operator belongs to the class of "arithmetic binary operators".

Function $U(\cdot, \cdot)$ applies, if required, the *function designator conversion*, replacing t_1 and t_2 with $F(t_1)$ and $F(t_2)$, then one of the following conversion rules:

1. if t_1 and t_2 are both arithmetic, the *usual arithmetic conversions* standard pattern, described below is applied;

2. if at least one of t_1 and t_2 is a pointer, the *pointer arithmetic* rules are applied.

From the standard, «This pattern is called the *usual arithmetic conversions*:
First, if either operand has type **long double**, the other operand is converted to **long double**.
Otherwise, if either operand has type **double**, the other operand is converted to **double**.
Otherwise, if either operand has type **float**, the other operand is converted to type **float**.
Otherwise, the integral promotions are first applied to both operands and then the following rules are applied.
If either operand has type **unsigned long int**, the other operand is converted to **unsigned long int**.
Otherwise, if one operand has type **long int** and the other has type **unsigned int**, if a **long int** can represent all values of an **unsigned int**, the operand of type **unsigned int** is converted to **long int**; if a **long int** cannot represent all the values of an **unsigned int**, both operands are converted to **unsigned long int**.
Otherwise, if either operand has type **long int**, the other operand is converted to **long int**.
Otherwise, if either operand has type **unsigned int**, the other operand is converted to **unsigned int**.
Otherwise, both operands have type **int**. »

Syntax:

⟨*additive_expression*⟩ ::= ⟨*additive_expression-1*⟩ '+' ⟨*multiplicative_expression*⟩
| ⟨*additive_expression-1*⟩ '−' ⟨*multiplicative_expression*⟩

⟨*multiplicative_expression*⟩ ::= ⟨*multiplicative_expression-1*⟩ '∗' ⟨*cast_expression*⟩
| ⟨*multiplicative_expression-1*⟩ '/' ⟨*cast_expression*⟩
| ⟨*multiplicative_expression-1*⟩ '%' ⟨*cast_expression*⟩

⟨*inclusive_or_expression*⟩ ::= ⟨*inclusive_or_expression-1*⟩ '|' ⟨*exclusive_or_expression*⟩

⟨*exclusive_or_expression*⟩ ::= ⟨*exclusive_or_expression-1*⟩ '^' ⟨*and_expression*⟩

⟨*and_expression*⟩ ::= ⟨*and_expression-1*⟩ '&' ⟨*equality_expression*⟩

Generalized syntax:

⟨*father_expression*⟩ ::= ⟨*child_expression-1*⟩ operator ⟨*child_expression-2*⟩

Semantics:

$$\langle \textit{father\_expression} \rangle.t = U(\langle \textit{child\_expression-1} \rangle.t, \langle \textit{child\_expression-2} \rangle.t);$$

Technical digression: in Section 4.4.1.3 (page 145), I anticipated that expression '++i' is not equivalent to 'i = i + 1' as far as types are concerned, because arithmetic binary operators perform integral type promotion, while unary '++' and '−−' operators don't. An example which clearly displays this unexpected behavior follows: if 'i' is declared as **char**, then type of expression '++i' is **char**. On the other hand, the type of expression 'i+1' is the integral promotion of type **char**, which is **int**. This can be practically verified, for

example, by measuring the size occupied by the value of expressions '++i' and 'i + 1' respectively, with the '**sizeof**' operator and on any 32 bit architecture. The following fragment:

```
char i;
printf("%i␣%i␣%i\n", sizeof i, sizeof(++i), sizeof(i++));
```

outputs `1 1 1`, proving that if 'i' is **char**, then '++i' and 'i++' are still **char**s. On the other hand, the following fragment:

```
char i;
printf("%i␣%i␣%i", sizeof i, sizeof(i+1), sizeof(1+i));
```

outputs `1 4 4`, proving that '1+i' and 'i+1' are promoted to '**int**'s. Note that **sizeof**(**char**) must be 1 by standard, whereas **sizeof**(**int**)==4 is an architecture-dependent detail.

4.4.1.12 The access operators

For the subscript operator, most of the considerations already made for the dereferencing operator '$*$' apply.

Syntax:

⟨*postfix_expression*⟩ ::= ⟨*postfix_expression-1*⟩ '[' ⟨*expression-1*⟩ ']'

Semantics:

$$\langle \textit{postfix\_expression} \rangle.t = pop(F(\langle \textit{postfix\_expression-1} \rangle.t))$$

For the '.' and '−>' operators, I assume that the '[struct]' and '[union]' type-definition records have an attribute which contains the full symbol table associated with the struct/union as declared. For brevity, I do not denote this attribute in the typesetting.

Additionally, I define a function $lookup(\cdot, \cdot)$ which determines the type of a members of a struct or union. Given an type-definition record r and a name n, $lookup(t, n)$ yields a type stack corresponding to the type of name n as declared in the symbol table associated with the type-definition record.

Any further discussion regarding $lookup$ is out of the scope of this document. The readers interested to learn more on this topic are invited to refer to the source code of the project which implements this thesis.

Syntax:

⟨*postfix_expression*⟩ ::= ⟨*postfix_expression-1*⟩ '.' IDENTIFIER

Semantics:

$$\langle \textit{postfix\_expression} \rangle.t = lookup(top(\langle \textit{postfix\_expression-1} \rangle.t), IDENTIFIER.n)$$

Syntax:

⟨*postfix_expression*⟩ ::= lookup⟨*postfix_expression-1*⟩ '−>' IDENTIFIER

Semantics:

$$\langle \textit{postfix\_expression} \rangle.t = lookup(top(pop(\langle \textit{postfix\_expression-1} \rangle.t)), IDENTIFIER.n)$$

4.4.1.13 The conditional operator

The result type of a conditional operator is the calculated according to the usual arithmetic conversions, applied on the second and third operand.

Syntax:

⟨*conditional_expression*⟩ ::= ⟨*logical_or_expression*⟩ '?' ⟨*expression*⟩ ':' ⟨*conditional_expression-1*⟩

Semantics:

$$\langle \textit{conditional\_expression} \rangle.t = U(\langle \textit{expression} \rangle.t, \langle \textit{conditional\_expression-1} \rangle.t);$$

4.4.1.14 The function call operator

A function call is «a postfix expression followed by parentheses containing a possibly empty, comma-separated list of expressions», as from the C standard, Section 6.3.2.2. I consider the function call to have arity $1 + n$ if the argument list contains n elements. The first child of the operator is always an expression denoting which function is called (symbol ⟨*postfix_expression-1*⟩ in the syntax rules below). The remaining children are the nodes respectively corresponding to the arguments, if any.

The result type of a function call node is the declared return type of the function. Moreover «the expression that denotes the called function shall have type pointer to function returning **void** or returning an object type other than an array type». In my notation:

$$\langle \textit{postfix\_expression-1} \rangle.t = \text{[pointer][function]}u$$

with $top(u) \neq$[array]. Clearly the result type for the function call expression (u) is obtained by popping t twice.

The above rule is straightforward when its first operator is of type "pointer to function", but it makes no exception when the left operand is a function designator. If ⟨*postfix_expression-1*⟩ denotes a function returning type u, then attribute ⟨*postfix_expression-1*⟩.t=[function]u. According to type conversion rules for function designators, it is automatically converted to "pointer to function" type (i.e. to [pointer][function]u). Also in this case the above "pop twice" rule correctly obtains the type of the function call result value (i.e. u). This is perfectly compliant with the type mangling convention I choose to apply for function designators, as described below.

Consider the following example fragment of C code:

```
int * myfunction(char a, char b) {
  return ...;
}

int main() {
  char a, b;
  int * q;

  ...
```

```
  q = myfunction(a,b);
  ...
}
```

As the determination of the type of expression 'myfunction(a,b)' is concerned, the following steps occur:

1. $\langle$*postfix_expression-1*$\rangle$.t = [function][pointer][int];
   (as determined by symbol table lookup)

2. then, the function designator promotion is calculated:
   $F(\langle$*postfix_expression-1*$\rangle.t)$ = [pointer][function][pointer][int];
   (according to the function designator conversion rule, explained below);

3. finally, $\langle$*postfix_expression*$\rangle.t$ = [pointer][int];
   therefore $\langle$*postfix_expression*$\rangle.t = pop(pop(\langle$*postfix_expression-1*$\rangle.t'))$;

The two type-mangling behaviors just described translate formally into the semantic actions below.

Syntax:

$\langle$*postfix_expression*$\rangle$ ::= $\langle$*postfix_expression-1*$\rangle$ '(' ')'
| $\langle$*postfix_expression-1*$\rangle$ '(' $\langle$*argument_expression_list*$\rangle$ ')'

Semantics:

$$\langle\textit{postfix\_expression}\rangle.t = pop(pop(F(\langle\textit{postfix\_expression-1}\rangle.t)));$$

Please note that, according to the rules, it must hold that: $\langle$*postfix_expression-1*$\rangle.t$ = [pointer][function]u for some u.

Function designators are subject to conversions which may be counter-intuitive, and which introduce additional complexity in the type evaluation mechanism presented here. Now I detail and motivate these claims.

According to the C standard, «a function designator is an expression that has function type.» ([81], Section 6.2.2.1). Functions designators are subject to the following conversions (*ibidem*): «Except when it is the operand of the **sizeof** operator or the unary &, a function designator with type "function returning *type*" is converted to an expression that has type "pointer to function returning *type*"».

As a consequence of the above requirement, if 'myfunction' is the name of a function, then expressions 'myfunction' and '& myfunction' denote the same object. Moreover, they are both of type "pointer to function", and they are pointers which point to the same function. I illustrate this claim in the following example: if function 'myfunction' is declared as follows:

```
int myfunction() {
  ...
}
```

then the following expression is always true:

myfunction == &myfunction

i.e., 'myfunction' and '& myfunction' point to the same object. Additionally, if 'myfunction' is automatically converted to a function to pointer, it can be dereferenced with the star operator, and expression '∗myfunction' is an expression of type "function" which is, in turn subject to automatic conversion to pointer to function. Therefore it is always true that:

myfunction == &myfunction == ∗myfunction

Moreover, the above three expression have the same type:

- expression 'myfunction' is automatically converted to type "pointer to function";
- expression '&myfunction' takes the address of a function and is therefore a pointer to a function;
- expression '∗myfunction' dereferences a function name which is converted to pointer to function, thus obtaining a function, which is again promoted to pointer to a function;

as it is illustrated in the annotated ASTs reported in Figures 4.11, 4.12 and 4.13.

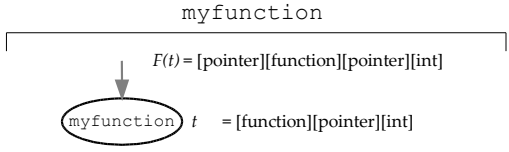


Figure 4.11: Example of function designator conversion, when the expression is a simple function name.

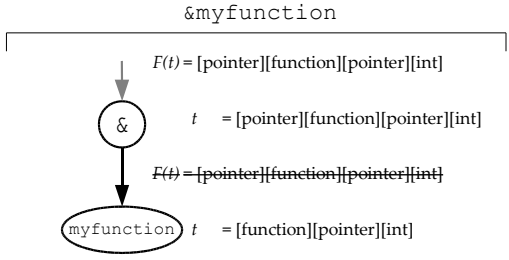


Figure 4.12: Example of function designator conversion, when a function designator is the operand of a referencing '&' operator.

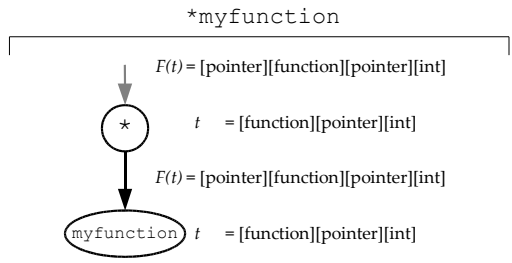


Figure 4.13: Example of function designator conversion, when a function designator is the operand of a dereferencing '$*$' operator.

According to its definition, operator '()', the function call operator, expects on its left an expression of type "pointer to function". The three expressions below, which are respectively the usual and two less-usual ways to call a function whose name is given, satisfy this requirement:

```
i = myfunction();          /* type conversion done once */
i = (&myfunction)();       /* type conversion never done */
i = (*myfunction)();       /* type conversion done twice */
```

in the first statement, the expression at the left of the parentheses is automatically converted to "pointer to function". In the second statement, a pointer to the function is explicitly taken via the '&' operator. In the third statement, subexpression 'myfunction' is converted to pointer to function, and then dereferenced. Then,'*myfunction' is again a function designator and is converted again to pointer to function.

Finally, if I declare a variable 'funcptr' which is of type "pointer to function returning int" (precisely [pointer][function][int]), then 'funcptr' is assignment-compatible with both 'myfunction' and '& myfunction', therefore the assignments below are all type-correct:

```
int (*funcptr)(); // funcptr is a pointer to a function returning int

funcptr =     myfunction;
funcptr = &   myfunction;
funcptr = *   myfunction;

funcptr = *funcptr;
```

Then, once this semantics of interchangeability between functions and pointer to functions was chosen by the language designers, it became necessary to prevent the automatic conversion of function designators to "pointer to function" when they are operands of '&', otherwise this semantics would not be enforced.

I define function $F(\cdot)$ such that it performs the required conversion if the argument is a function designator type, otherwise it leaves it unchanged. If t is a function designator type, $F(t)$ is the converted type, according to the function designator conversion just described.

This function performs the function designator conversion as required by the standard. This consists in converting a "function returning type" to a "pointer to function returning type".

$$F(t) = \begin{cases} \text{push([pointer],}t\text{)} & \text{when top}(t)\text{=[function]} \\ t & \text{else} \end{cases}$$

A technical digression follows on the choice of an appropriate representation technique for t in presence of function designator conversion.

In designing this attribute grammar, I had a choice whether to set attribute t of a node to its real type or to its possibly converted type according to the above rules. Provided that a given semantic action is rendered in the following way:

$$\langle expr \rangle.t = function(\langle sub\_expr\text{-}1 \rangle.t, \langle sub\_expr\text{-}2 \rangle.t, ...);$$

for some $function$ which depends on the individual syntax rule, then:

1. the first solution would require to change the above rule as follows:

   $$\langle expr \rangle.t = function(F(\langle sub\_expr\text{-}1 \rangle.t), F(\langle sub\_expr\text{-}2 \rangle.t, ...)$$

   which converts the children's types immediately before considering them in their father. Attribute t is stored unconverted. The above amendment should be applied to all the operators except for '**sizeof**' and '&', as required by the standard.

2. the second solution would require to change the rule as follows:

   $$\langle expr \rangle.t = F(function(\langle sub\_expr\text{-}1 \rangle.t, \langle sub\_expr\text{-}2 \rangle.t, ...));$$

   which assumes the children's type to be already converted, it calculates the result type and it converts the father's type before saving it into t. If this solution is chosen, appropriate actions must be taken in the semantic actions of '**sizeof**' and '&' to reverse the effects of the conversion which was undue.

Only the first solution is viable, because reversing the effects of the $F()$ conversion is not, in general, a type-correct task. I prove this claim by assuming that all the semantic actions store the type-converted t, as in rule (2) above, then amending the semantic actions associated to the '&' operator, and finally showing that those actions are incorrect.

Under these assumptions, the semantic actions for the type determination of the referencing operator (in Section 4.4.1.5 (page 146)) should be modified as follows:

Semantic rules (hypothetical):

- $\langle unary\_expression \rangle.t = \langle cast\_expression \rangle.t$;
  if $\langle cast\_expression \rangle.t$ = [pointer][function]u, for some u;
- $\langle unary\_expression \rangle.t = push($[pointer], $\langle cast\_expression \rangle.t)$;
  otherwise;

In the above semantic rules, the second rule applies to "usual" types, while the first one applies when the operand has type "pointer to function". The rule assumes that the type is the result of a function designator conversion, and avoids adding another [pointer] type-definition operator on top of its type. Ill, the above rule incorrectly resolves types in the following program:

```
int  myfunction();
int  (*funcptr )();
int  (**funcptrptr )();
```

```
int main()
{
   ...
  funcptrptr = & funcptr;
   ...
}
```

In the above program, variable funcptr is of type [pointer][function][int]. It is therefore perfectly legal to take its address, which is of type [pointer][pointer][function][int] and to store it in variable funcptrptr, which is exactly declared as type [pointer][pointer][function][int]. Unfortunately, the above semantic rules incorrectly calculates the type of expression & funcptr as [pointer][function][int], which is wrong.

Therefore, I choose to adopt the first solution, i.e. associate to each symbol its type before as it is before the function designator conversion.

4.4.2 Attribute 'r', restricted result type

Attribute r is required to evaluate correctly the inherent cost of expressions involving the dot operator. In this section I explain how.

The *restricted result type* of an expression is the type of the value which is actually transferred for use in the super-expression of the current expression, as opposed to the *result type* already defined in Section 4.4.1 (page 140) as attribute t. The restricted result type assumes, almost everywhere, the same value as the result type. That is, in the vast majority of cases, $\langle expr \rangle.r = \langle expr \rangle.t$. Differences emerge in nodes which are children of a dot operator expression.

The dot operator allows to access members of a structure or of a union. It is defined by the following syntax rule:

⟨*postfix_expression*⟩ ::= ⟨*postfix_expression-1*⟩ '.' IDENTIFIER

I consider it a binary operator, where the first operand type is structure or union, and second operand is an identifier among the ones declared in the symbol table associated with the structure or union.

The dot operator demands a peculiar analysis, because it exhibits anomalies which are unique among all the operators of the C language. These anomalies reside in the semantics of its evaluation order, in the way it affects the valueness of the nodes in its left operand subtree, and in the way it affects the restricted type of the nodes in its left operand subtree. In the end, these anomalies require special care in the design of the algorithm for the determination of attribute v, and also require the introduction of attribute r.

I illustrate these claims with the following example. Assume that variables 'a' and 'b' are declared as follows:

```
struct tag {
   ...
  type m;
   ...
} * a;
type b;
```

where 'type' can be any type allowed by the C language, whose size in words is given by $W(\text{type}) = w$. Then, consider the following statement:

```
(*a).m = b;
```

I now determine the AST of the assignment expression above, and the valueness of each AST node. For the moment, I do not take special care of the dot operator node. I just apply the usual rule "the valueness of the immediate left child node of a simple assignment operator node must is L". This leads to the AST in Figure 4.14.

According to the above tree, the following claims can be expressed:

1. Claim 1: first 'a' is dereferenced to obtain the structure pointed by it, then its member 'm' is resolved, i.e., the operations take place in the following order: first the dereference, then the member address resolution;

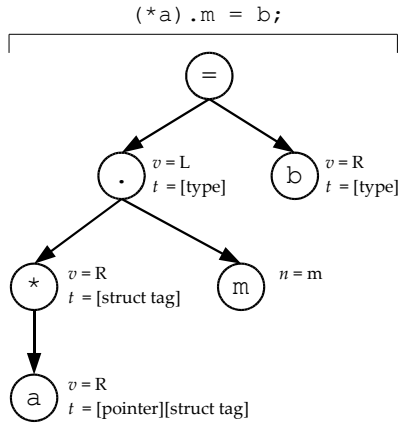


Figure 4.14: A partially decorated AST for the example expression, determined without special care for dot operators.

2. Claim 2: the valueness of expression '(∗a)' is R;
3. Claim 3: the type of the information transferred by the '∗' operator is "struct tag".

I will prove that, with respect to the assembly translation of the expression, **all the above claims are false**. First I explain the reasons why, then I provide an amended version of the above annotated AST, finally I devise appropriate rules are to be associated with the '.' operator in order to obtain correct attribute and cost determination.

4.4.2.1 Anomaly affecting the precedence

From a purely conceptual point of view, Claim 1 is correct: the order of execution of operators '∗' and '.' is the one induced by a bottom-up visit of the AST, as with any other operator: first the unary dereferencing operator '∗' is applied on 'a', which is of type "pointer to struct", obtaining an expression of type "struct"; then the member access operator '.' is applied on this last expression, obtaining the member m, whose L-value is used in such a way that it receives the value currently assigned to 'b'. Briefly, from a conceptual point of view, first '∗' is evaluated, then '.'. But the assembly translation of the above code, which is as follows:

```
add    t_0, a, ⟨offset of m⟩     ; calculates the address of the first word

mvst   t_0, b                    ; transfers the first word

add    t_1, t_0, #4              ; calculates the address of the second word
mvst   t_1, b+1                  ; transfers the second word
 ...                             ; ...
```

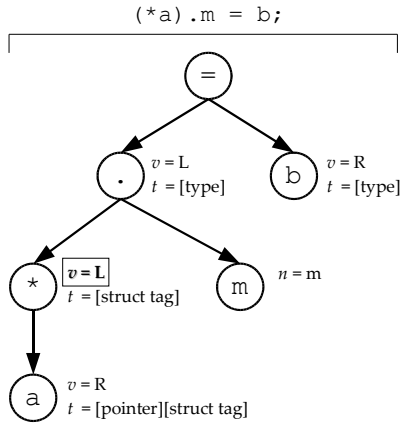


Figure 4.15: The same expression as in the previous example, reworked with amended valueness determination rules. The valueness of node '∗' has been now determined as 'L'.

```
add     t_w−1, t_w−2, #4        ; calculates the address of the last word
mvst    t_w−1, b+w−1            ; transfers the last word
```

contains a block of mvst and add instructions (which are a pointer dereferencing, the translation of the unary '∗', atom LValueStar) appearing *after* an add instruction (used to calculate the address of member 'm', temporarily stored in register 't_0'). Therefore, from the point of view of the assembly translation, operator '.' is evaluated *before* operator '∗' operator. This is the reverse of the order one implied by the AST.

Since this anomaly affects only the order in which operations take place but not their quantity or nature, it has no consequences on the cost of the expressions. I report it just for sake of completeness.

4.4.2.2 Anomaly affecting valueness

As far as Claim 2 is concerned, by looking at the above abstract translation of statement '(∗a).m = b;', notice that its transfers comprise 'mvst' instructions in place of the 'mvld' instructions I expect to find in the translation of an expression with R valueness. The above assembly code is, in fact, the translation of an L-value expression. Therefore, the '∗' node should be definitely marked with L valueness even though it is not an immediately left child of a simple assignment node. The correct valueness values are illustrated in Figure 4.15.

Extending the analysis of the translation of sub-expressions rooted at a dot nodes with all the possible valuenesses, I derive the following general rule: **a dot node propagates its valueness to his immediate left child node**. This rule corresponds to following lines of pseudo-code:

```
if (p->operator_code == code("."))
  children[0]->valueness = p->valueness;
```

which are part of the valueness evaluation algorithm I will present completely in Section 4.4.4 (page 176).

The reader is invited to notice that, thanks to the recursive nature of the algorithm, the valueness is correctly attributed via valueness propagation also in cases which involve nested dot expressions. To help the reader in better understanding this case, I illustrate the following example. Assume that variables 'a' and 'b' are declared as follows:

```
struct tag_outer {
   ...
  struct tag_inner {
     ...
    type m;
     ...
  } s;
   ...
} * a;
type b;
```

and consider the following statement:

```
(*a).s.m = b;
```

The AST for the above statement is reported in Figure 4.16 (page 162), with the correct valuenesses, as determined by the algorithm described.

4.4.2.3 Anomaly affecting the transferred size

As far as Claim 3 is concerned, I prove its incorrectness by considering again the previous example concerning the C statement '(*a).m = b;', with the annotated AST in Figure 4.17 (where nodes have been numbered $N_1, N_2, ..., N_6$ for ease of reference).

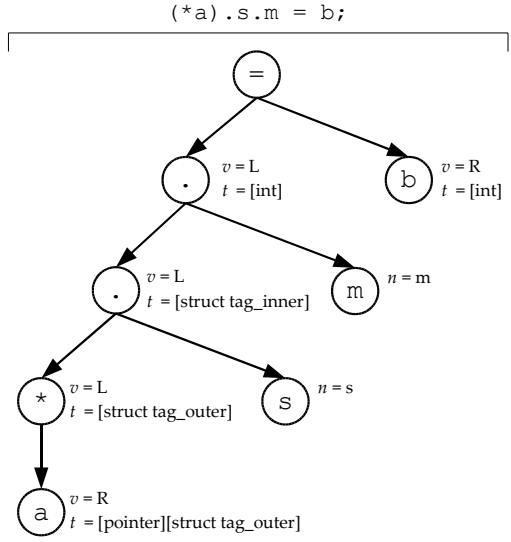


Figure 4.16: Example of application of the amended valueness determination rules on an expression including nested instances of the '.' operator.

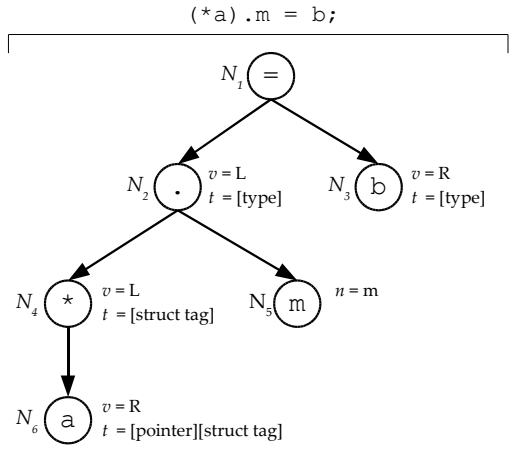


Figure 4.17: Example expression.

So far, I have correctly determined that the valueness of the expression

associated to operator '$*$' is L, that is, $N_4.v$ =L, which means that operator '$*$' writes register contents to the memory. The actual cost of that operator clearly depends on how many words have to be written, and I expect that number to be equal to $W(N_4.t)$. **This is false.** In fact, the number of transferred words is less than that figure. $W(N_4.t)$ yields the number of words occupied by the entire '**struct** tag', since $N_4.t$ =[struct tag]. Instead, the assembly code translation, shown above, proves that the number of transferred words is less than the word size of the structure, and equal to the word size of just member 'm'. That number is $W(N_2.t)$. If I used $W(N_4.t)$ as a measure of the words to transfer in place of $W(N_2.t)$ I would overestimate the actual cost of the star operator. In this case, attribute $N_4.c$ depends on $N_2.t$. Therefore, the assumption "the cost of a node is function of the word size of the return type of the same node", which was always verified so far is **false** for node N_4.

In this case, for $n = N_4$, the transferred words are given $W(father(n).t)$. Anyway, this rule is not general. In fact, in the second example above (statement '(*a).s.m = b;'), with two nested dot operators, the number of transferred words for node '$*$' is given by $W(father(father(n)).t)$.

In order to rectify the above state of things, in such a way that $n.c$ depends only on attributes of a node with a fixed relationship with respect to n, possibly n itself, I introduce attribute r. I call r the *restricted type* to express the idea that the dot operator restricts the scope of its left subtree operator in such a way that it operates not on the full structure or union, but only on a member.

In the following examples I illustrate how the dot operator affects the restricted type of the star '$*$', array subscript '[]', and arrow '$->$' operators.

Assume the following type declarations for variables 'a', 'b' and 's'.

```
struct tag {
   ...
  type m;
   ...
} s, * a;
type b;
```

Consider the following statements:

```
*a = s;
(*a).m = b;
```

The first statement is an assignment which transfers an entire **struct** from the bank of registers pointed by the value stored in register 'a' to the bank of registers associated to 's'. The type t for node '$*$' is [struct tag], and it coincides with r, the type of the actual transferred data, that is, the entire struct. In the second statement, the dot operator restricts the effects of the dereference operator in such a way that only member 'm' is transferred. Therefore, the restricted type r of the same subexpression considered before, the '$*$' node, is not anymore equal to [struct tag], instead it is [type], even though the considered subtree is identical as in the previous statement. The decorated subtree in Figure 4.18 illustrates what I have just said.

The next example shows how the dot operator affects the restricted type

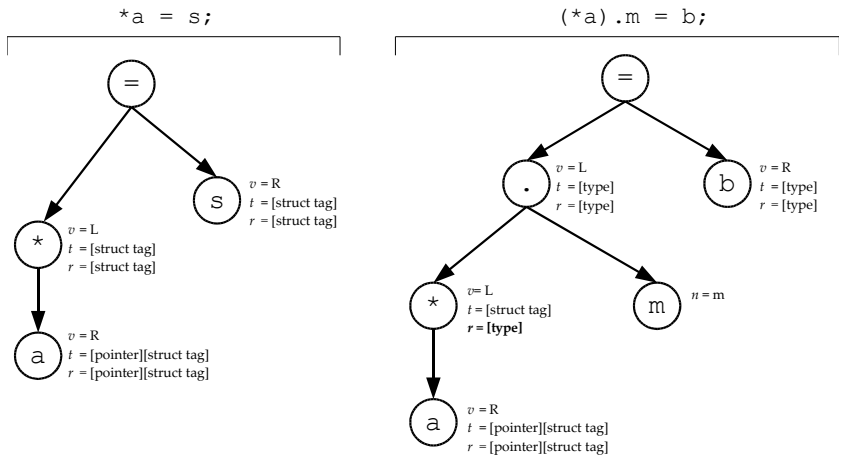


Figure 4.18: Attribute *r* models appropriately how '.' operators restrict the type of information transferred by '∗' operators.

of array subscript operators. Assume variables 'a', 'i', 's' and 'm' as declared below:

```
struct tag {
   ...
  type m;
   ...
} s, a[N];
type  b;
int   i;
```

(where 'N' is an appropriate integer constant value expression). Consider the following statements:

```
a[i] = s;
a[i].m = b;
```

The first statement is an assignment which transfers an entire **struct** from the bank of registers pointed by the value stored in a cell of array 'a' to the bank of registers associated to 's'. The type *t* for node '[]' is [struct tag], and it coincides with *r*, the type of the actual transferred data, that is, the entire struct. In the second statement, the dot operator restricts the effects of the array subscript operator in such a way that only member 'm' is transferred. Again, the restricted type *r* of the same subexpression considered before, the '[]' node, is not anymore equal to [struct tag], instead it is [type], even though the considered subtree is identical as in the previous statement. The decorated subtree in Figure 4.19 (page 165) illustrates what just said.

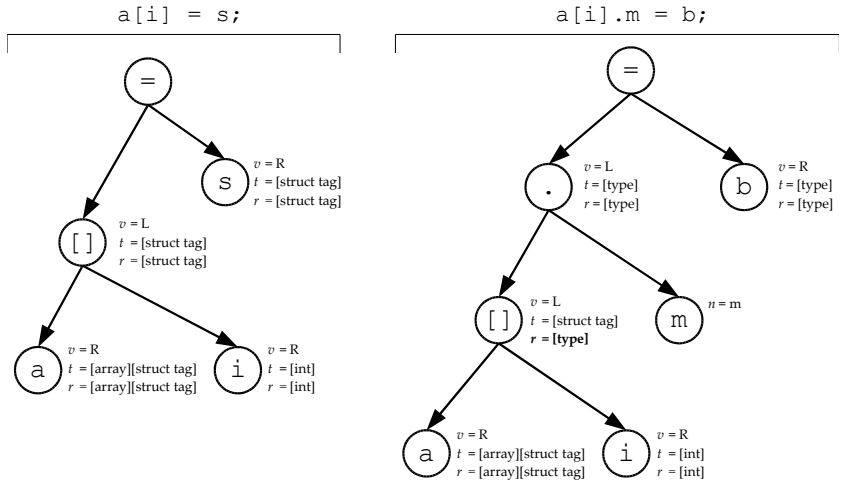


Figure 4.19: Attribute *r* models appropriately how '.' operators restrict the type of information transferred by '[]' operators.

In the last example I show how the dot operator affects the restricted type of arrow operators. Assume the variables 'a', 'n', 's' and 'm' as declared below:

```
struct tag2 {
   ...
  type m;
   ...
} s;

struct tag1 {
   ...
  struct tag2 n;
   ...
} a;

type b;
```

Consider the following statements:

```
a−>n = s;
a−>n.m = b;
```

The first statement is an assignment which transfers an entire **struct** from the bank of registers associated to member 'n' of a struct pointed by 'a' to the bank of registers associated to 's'. The type *t* for node '−>' is [struct tag2], and it coincides with *r*, the type of the actual transferred data, that is, the entire struct tag2. In the second statement, the dot operator restricts the ef-

fects of the array subscript operator in such a way that only member 'm' is transferred. Again, the restricted type *r* of the same subexpression considered before, the '−>' node, is not anymore equal to [struct tag2], instead it is [type], even though the considered subtree is identical as in the previous statement. The decorated subtree in Figure 4.20 (page 166) illustrates what I have just said.

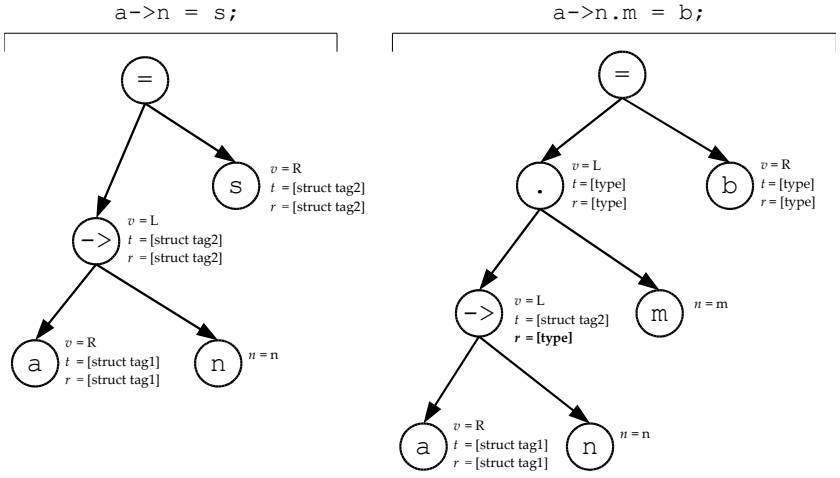


Figure 4.20: Attribute *r* models appropriately how '.' operators restrict the type of information transferred by '−>' operators.

On the basis of the above three examples, I could tentatively derive the following principle which describes the influence exerted by the dot operator onto operators '∗', '[]' and '−>': «if the left immediate children of the current dot node is a dereference, subscript or arrow operator node, its restricted type is set equal to the type of the current node». This tentative rule behaves correctly in all the above cases, but not with nested dot operator nodes. I justify this claim with the following example, where variables 'a', 'n', 's' and 'm' as declared below:

```
struct tag_outer {
  ...
  struct tag_inner {
    ...
    type m;
    ...
  } n;
  ...
} * a;

type b;
```

Consider the following statement:

(∗a).n.m = b;

The correct value of attribute *t* of subexpression '(∗a)' is [type], as depicted in the annotated AST in Figure 4.21.

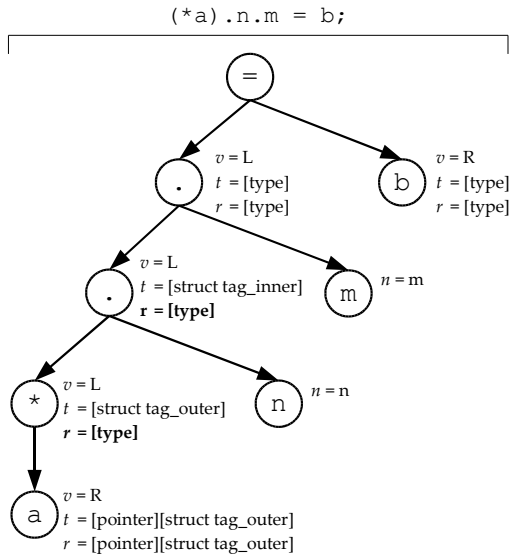


Figure 4.21: Attribute *r* models appropriately the type of information transferred (e.g. by '∗' operators) even in presence of nested '.' operators.

The analysis of cases involving chains of nested dot operators show that the restricted type of the left child of bottommost dot operator node in the chain must be set equal to the (simply said) type of the topmost dot operator node.

Although all the cases considered so far involved **struct**s and not **union**s, the above principle describes completely the correct determination criterion for attribute *t*, irrespectively of whether the dots resolve **union** or **struct** members.

4.4.2.4 Attribute 'r' calculation rules

In this section, I describe the general algorithm for the determination of attribute *r*. Attribute *r* is inherited, and its value for a given node depends, in general on the value of *t* in the same node and on the value of *t* of nodes which are ancestors of the current node.

I choose not to describe the evaluation of attribute *r* with individual semantic rules associated with one syntax rules each, as I usually do for syn-

thesized attributes. The reason is that attribute *r* of a node may depend not only on the father but also on farther ancestor nodes. Therefore, decisions on *r* may span multiple derivation steps, and involve numerous syntax rules. Writing these rules would entail great effort and produce complex networks of semantic actions which are difficult to understand, explain and verify, and which require additional attributes.

Instead, without less of formality, I describe the evaluation of attribute *r* in the form of a recursive algorithm, written as a C function.

The algorithm assumes that all the parse nodes have their *r* attribute initially set to the same value as *t*. Then the algorithm changes the restricted type to the right value in all the nodes where it is required.

The following pseudo-C code indicates how to determine attribute *r*:

```
1  struct parse_node {
2    enum {expression, statement, terminal}   symbol;
3    int                                      operator_code;
4    struct parse_node                     ** children;
5    int                                      children_count;
6    enum {R, L, RL, Z}                       v;
7    type_representation                      t;
8    type_representation                      r;
9  };
10
11 void evaluate_r(struct parse_node * p)
12 {
13   struct parse_node * pN = p;
14   int i;
15
16   if (p->symbol==expression && p->operator_code == code("."))
17   {
18     while(pN->symbol==expression && pN->operator_code==code("."))
19     {
20       pN = pN->children[0];
21       pN->r = p->t;
22     }
23     p = pN;
24   }
25
26   for (i=0; i<p->children_count; i++)
27     evaluate_r(p->children[i]);
28 }
```

In the above algorithm, '**struct** parse_node' contains the data associated to an AST node. If 'p' points to a node, 'p->v' denotes its attribute $\langle expression \rangle.v$, 'p->children_count' indicates how many children it has (its *arity*), and expressions 'p->children[0]', 'p->children[1]', ..., 'p->children[n-1]' denote the first, second, ..., last child.

4.4.3 Attribute 'k' and 'e': constancy and constant value

The attribute k of an expression is a boolean value, which indicates whether the value of the current expression is constant or not. If attribute k for a given expression assumes the value true (the expression has a constant value), then attribute e is defined and it contains the value of the expression.

Attributes k and e are synthesized. Attribute k is, in general, function of attributes k and e of the children nodes. Attribute e is, in general, function of attributes k, e and t of the children nodes. Whenever it can be inferred from the value of k and e (constancy and result, respectively) of the children nodes that the current node is constant, k assumes the value true, otherwise it assumes the value false. In the first case, e assumes the value of the operation applied on the values of the children, otherwise e is undefined.

For many expressions composed by a binary operator (e.g. '+'), k is true if both the children have k true: the sum is constant only if both operands are constant.

Two simple examples, which illustrate the calculation of attributes k and e for an ⟨*additive_expression*⟩ node are reported below:

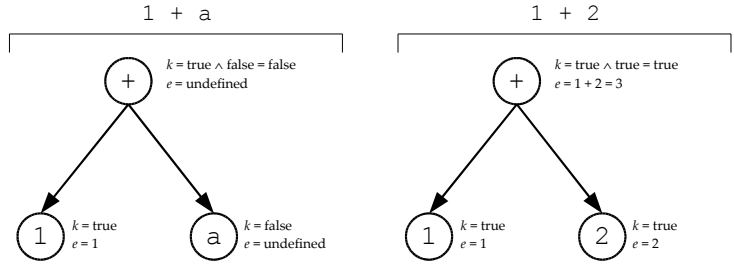


Figure 4.22: Trivial example illustrating the evaluation of attributes k and e.

As far as the evaluation of k and e is concerned, the operators of the C language fall in the nine distinct classes described in Table 4.3.

Class	Arity	Informal description, members and behavior
1	1	Sizeof operator '**sizeof**' Behavior: constancy is always true.
2	1	Simple unary operators '+', '−', '~', '!', postfix '++', postfix '−−' Behavior: constancy is same as child node.
3	1	Other unary operators '∗', '&', prefix '++', prefix '−−' Behavior: constancy is always false.
4	2	Simple binary operators '+', '−', '∗', '/', '%', '&', '|', '^' '<<', '>>', '==', '!=', '<', '>' '<=', '>=' Behavior: constancy depends on constancy and value of both children.
5	2	Logical binary operators '&&', '||' Behavior: constancy and value may depend on constancy and value of one or both children, according to properties of logic *and* and *or* operations.
6	2	Access operators and Compound assignment operators '.', '−>', '[]', '+=', '−=', '∗=', '/=', '%=', '&=', '|=', '^=', '<<=', '>>=' Behavior: constancy is always false.
7	2	Assignment and comma '=', ',', the cast operator ' (...) ' Behavior: constancy and value are the same as the second child node.
8	3	Conditional operator '?:' Behavior: constancy and value depends on constancy and value of at most 2 of the 3 child expressions.
9	1..n	Function call function call ' ... (...) ' Behavior: constancy is always false.

Table 4.3: The operators of the C language classified on the basis of their behavior with respect to the determination of their constancy and constant value (attributes k, e).

4.4.3.1 The 'sizeof' operator

The '**sizeof**' operator always returns a constant value, therefore the constancy of a '**sizeof**' expression is always true. The result value of the expression is given by applying function $S(t)$ on the type (attribute t) of the operand. Given a type attribute t associated to an expression n, $S(t)$ is the function which determines the result of the expression '**sizeof** n', that is, the number of bytes occupied by type t. The C standard [81] specifies in Section 6.3.3.4 a number of constraints on how the types should be allocated, and therefore on function $S(\cdot)$.

Syntax rules:

⟨*unary_expression*⟩ ::= '`sizeof`' ⟨*unary_expression-1*⟩
| '`sizeof`' '(' ⟨*type_name*⟩ ')'

Semantics:

- ⟨*unary_expression*⟩.k = true;
- ⟨*unary_expression*⟩.e = S(⟨*child_node*⟩.t);

---

where ⟨*child_node*⟩ indicates the only child node (either on the concrete or abstract syntax tree, without ambiguity), which is either symbol ⟨*unary_expression*⟩ or ⟨*type_name*⟩. Note that attribute t is defined for the child node.

Note: attribute ⟨*unary_expression*⟩.e is calculated as $S(\langle child\_node\rangle.t)$ and not as $S(F(\langle child\_node\rangle.t))$. Thus, function designator conversion is not performed, as requested by the standard and motivated in Section 4.4.1.14 (page 153).

4.4.3.2 Simple unary operators

The unary operators in this class preserve the constancy of their operands. Their constant value, if defined, results from changing the operand value according to the operator's semantics.

Syntax rules:

⟨*unary_expression*⟩ ::= operator ⟨*cast_expression*⟩
where operator is one of '+', '−', '~', '!'; or
⟨*postfix_expression*⟩ ::= ⟨*postfix_expression*⟩ operator
where operator is one of '++', postfix '−−'.

Generic rule:

⟨*father_expression*⟩ ::= ⟨*child_expression*⟩ operator
| operator ⟨*child_expression*⟩

Semantics:

- k of ⟨*father_expression*⟩ = k of ⟨*child_expression*⟩;
- e of ⟨*father_expression*⟩ = (operator applied on e of ⟨*child_expression*⟩);

---

where 'operator applied on e' indicates respectively e, $-e$, e, $\neg e$, e, e depending on what the operator is, respectively '+', '−', '~', '!', postfix '++',

postfix '−−'. Operators '+', postfix '++', postfix '−−' do not change the value of the operand.

Note: the operators postfix '++' and postfix '−−' could be indifferently categorized in class 2 or 3, since they can be applied only to lvalues, and lvalues are never constant expressions. Nevertheless the above syntax rules are correct, and I choose to put them in this class since, from a conceptual point of view, the return value of the expression is the same, unchanged value of the operand.

4.4.3.3 Other unary operators

This class comprises all the unary operators which never constitute a constant expression. Therefore, their attribute *k* is always false, and attribute *e* is always undefined.

Syntax rules:

⟨*unary_expression*⟩ ::= '++' ⟨*unary_expression*⟩
| '−−' ⟨*unary_expression*⟩
| '*' ⟨*cast_expression*⟩
| '&' ⟨*cast_expression*⟩

Semantics:

- ⟨*unary_expression*⟩.*k* = false;
- ⟨*unary_expression*⟩.*e* = undefined;

4.4.3.4 Simple binary operators

The expressions formed by the binary operators in this class are constant if both their operands are constant.

Generalized syntax:

⟨*expression*⟩ ::= ⟨*child_expression-1*⟩ ⟨*class_3_operator*⟩ ⟨*child_expression-2*⟩

Semantics:

- ⟨*expression*⟩.*k* = ⟨*child_expression-1*⟩.*k* ∧ ⟨*child_expression-2*⟩.*k*;
- ⟨*expression*⟩.*e* = ⟨*child_expression-1*⟩.*e* ⟨*class_3_operator*⟩.*n* ⟨*child_expression-2*⟩.*e*;

4.4.3.5 Logical binary operators

For the logical operators '&&' and '||', the properties of the logic operations allow to state that an expression is constant (and to determine its value) even in certain cases in which one of the two operands is undefined.

Syntax:

⟨*logical_and_expression*⟩ ::= ⟨*logical_and_expression-1*⟩ '&&' ⟨*inclusive_or_expression*⟩

Semantics:

- $\langle logical\_and\_expression \rangle.k =$
  $\langle logical\_and\_expression\text{-}1 \rangle.k \wedge \langle inclusive\_or\_expression \rangle.k \quad \vee$
  $\langle logical\_and\_expression\text{-}1 \rangle.k \wedge \langle logical\_and\_expression\text{-}1 \rangle.e = 0 \quad \vee$
  $\langle inclusive\_or\_expression \rangle.k \wedge \langle inclusive\_or\_expression \rangle.e = 0$

- $\langle logical\_and\_expression \rangle.e =$
  $\begin{cases} \langle logical\_and\_expression\text{-}1 \rangle.e \wedge \langle inclusive\_or\_expression \rangle.e & \\ \qquad \text{when } \langle logical\_and\_expression\text{-}1 \rangle.k \wedge \langle inclusive\_or\_expression \rangle.k & \\ 0 \qquad \text{when } \langle logical\_and\_expression\text{-}1 \rangle.k \wedge \langle logical\_and\_expression\text{-}1 \rangle.e = 0 & \\ 0 \qquad \text{when } \langle inclusive\_or\_expression \rangle.k \wedge \langle inclusive\_or\_expression \rangle.e = 0 & \\ \text{undefined} \quad \text{else} & \end{cases}$

Similar semantic rules apply for the logical *or* operator.

Syntax:

$\langle logical\_or\_expression \rangle$ ::= $\langle logical\_or\_expression\text{-}1 \rangle$ '||' $\langle logical\_and\_expression \rangle$

Semantics:

- $\langle logical\_or\_expression \rangle.k =$
  $\langle logical\_or\_expression\text{-}1 \rangle.k \wedge \langle logical\_and\_expression \rangle.k \quad \vee$
  $\langle logical\_or\_expression\text{-}1 \rangle.k \wedge \langle logical\_or\_expression\text{-}1 \rangle.e \neq 0 \quad \vee$
  $\langle logical\_and\_expression \rangle.k \wedge \langle logical\_and\_expression \rangle.e \neq 0$

- $\langle logical\_or\_expression \rangle.e =$
  $\begin{cases} \langle logical\_or\_expression\text{-}1 \rangle.e \vee \langle logical\_and\_expression \rangle.e & \\ \qquad \text{when } \langle logical\_or\_expression\text{-}1 \rangle.k \wedge \langle logical\_and\_expression \rangle.k & \\ 1 \qquad \text{when } \langle logical\_or\_expression\text{-}1 \rangle.k \wedge \langle logical\_or\_expression\text{-}1 \rangle.e \neq 0 & \\ 1 \qquad \text{when } \langle logical\_and\_expression \rangle.k \wedge \langle logical\_and\_expression \rangle.e \neq 0 & \\ \text{undefined} & \\ \qquad \text{else} & \end{cases}$

4.4.3.6 Access and compound assignment operators

All this operators return non-constant expressions.

Syntax rules:

$\langle assignment\_expression \rangle$ ::= $\langle unary\_expression \rangle$ operator $\langle assignment\_expression \rangle$

(where 'operator' is one of '+=', '−=', '∗=', '/=', '%=', '&=', '|=', '^=', '<<=', '>>=')

or

$\langle postfix\_expression \rangle$ ::= $\langle postfix\_expression \rangle$ '[' $\langle expression \rangle$ ']'
| $\langle postfix\_expression \rangle$ '.' IDENTIFIER
| $\langle postfix\_expression \rangle$ '->' IDENTIFIER

Generalized Syntax:

$\langle father\_expression \rangle$ ::= $\langle ... \rangle$

Semantics:

- $\langle father\_expression \rangle.k =$ false;
- $\langle father\_expression \rangle.e =$ undefined;

4.4.3.7 Simple assignment, comma and cast operators

These operators have the property to preserve the same constancy and value as their second operand.

Syntax:

⟨*assignment_expression*⟩ ::= ⟨*unary_expression*⟩ '=' ⟨*assignment_expression-1*⟩

⟨*expression*⟩ ::= ⟨*expression-1*⟩ ',' ⟨*assignment_expression*⟩

⟨*cast_expression*⟩ ::= '(' ⟨*type_name*⟩ ')' ⟨*cast_expression-1*⟩

Generalized syntax:

⟨*father_expression*⟩ ::= ⟨*op1*⟩ ⟨*child_expression-1*⟩ ⟨*op2*⟩ ⟨*child_expression-2*⟩

Semantics:

- ⟨*father_expression*⟩.*k* = ⟨*child_expression-2*⟩.*k*;
- ⟨*father_expression*⟩.*e* = ⟨*child_expression-2*⟩.*e*;

4.4.3.8 The conditional operator

The conditional operator is the only ternary operator of the C language. If the first child expression of the conditional expression is not constant, then the constancy of the entire expression is false, and its value is undefined. Otherwise, depending on whether attribute *e* of the first child node is non-zero or zero, the constancy and value of the entire expression are the same as the second or third child node respectively.

Syntax:

⟨*conditional_expression*⟩ ::= ⟨*logical_or_expression*⟩ '?' ⟨*expression*⟩ ':' ⟨*conditional_expression-1*⟩

Semantics:

- (⟨*conditional_expression*⟩.*k*, ⟨*conditional_expression*⟩.*e*) of =

$$\begin{cases} (false, \text{undefined}) & \text{when } \langle logical\_or\_expression \rangle.k = \text{false} \\ (\langle expression \rangle.k, \langle expression \rangle.e) & \text{when } \langle logical\_or\_expression \rangle.e \neq 0 \\ (\langle conditional\_expression\text{-}1 \rangle.k, \langle conditional\_expression\text{-}1 \rangle.e) & \text{when } \langle logical\_or\_expression \rangle.e = 0 \end{cases}$$

4.4.3.9 The function call operator

An expression involving a function call is never constant, and its value can never be determined at run time.

Syntax:

⟨*postfix_expression*⟩ ::= ⟨*postfix_expression-1*⟩ '(' ')'
| ⟨*postfix_expression-1*⟩ '(' ⟨*argument_expression_list*⟩ ')'

Semantics:

- $\langle postfix\_expression \rangle.k = \text{false};$
- $\langle postfix\_expression \rangle.e = \text{undefined};$

4.4.4 Attribute 'v', valueness

Attribute v of a node is the valueness of that node, according to the definitions already given in Section 4.2.1 (page 110). It is defined for expressions only, and it may assume one of the following values: Z, R, L, RL. This attribute does not depend on any other attributes; it depends only on the syntax. Attribute v is inherited.

For attribute v, the same considerations as for r apply: providing a determination algorithm in the form of semantic rules would be impractical and counterintuitive, and would require much more effort than describing the same criterion in the form of an imperative program. Therefore, I choose the latter form.

The algorithm assumes that each expression node has attribute v initially set to R. The algorithm follows.

```
1  struct parse_node {
2    enum {expression, statement /* ... */ }      symbol;
3    int                                          operator_code;
4    struct parse_node                         ** children;
5    int                                          children_count;
6    enum {R, L, RL, Z}                           v;
7  };
8
9  void evaluate_v(struct parse_node * p)
10 {
11   int i;
12
13   if (p->symbol==expression)
14   {
15     /* unary '&' operators cause Z-valueness */
16     if (p->operator_code == code("&") && p->children_count==1)
17       p->children[0]->v = Z;
18
19     /* assignments cause L-valueness */
20     if (p->operator_code == code("=") )
21       p->children[0]->v = L;
22
23     /* compound assignments cause RL-valueness */
24     if (p->operator_code == code("+=") ||
25         p->operator_code == code("-=") ||
26         p->operator_code == code("*=") ||
27         p->operator_code == code("/=") ||
28         p->operator_code == code("&=") ||
29                 ... )
30       p->children[0]->v = RL;
31
32     /* '.' operators propagate their valueness to left child;
33        this will be explained later */
34     if (p->operator_code == code("."))
35       p->children[0]->v = p->v;
36   }
37
38   for (i=0; i<p->children_count; i++)
39     evaluate_v(p->children[i]);
40 }
```

In the above algorithm, variables have the same meaning as in the algorithm to determine attribute *r* described in Section 4.4.2.4 (page 167).

Before the algorithm is started, all the parse nodes have valueness set to R. The algorithm changes the valueness to Z, L, or LR in all the nodes where it is required. More precisely:

- it changes the valueness to Z in all the nodes which are the immediate child of a referencing unary '&' operator (code fragment lines 16–17);
- it changes the valueness to L in all the nodes which appear as the immediate left child (the first child, which is pointed by 'children[0]') of a simple assignment operator '=' (code fragment lines 20–21);
- it changes the valueness to RL in all the nodes which appear as the immediate left child of a compound assignment operator, i.e., +=, −=, ∗=, ... (code fragment lines 24–30);
- it recursively propagates the valueness of the current node to the left child (whatever value it assumes) if the operator associated to the current node is a dot '.'. This behavior is chosen to properly model an anomaly in the dot operator, which was already completely discussed in Section 4.4.2.2 (page 160).

4.4.5 Attribute 'b', register boundedness

Attribute b indicates whether a node has its translation register-bound or not, according to the definition already given in Section 4.2.6 (page 116).

Attribute b is synthesized and depends only on the value of b of its children. Therefore, I could describe how to determine it in the form of semantic rules, as already done for t. For sake of brevity, I prefer to describe this task in natural language, in Table 4.4. In this table, all the operators who can assume different arities are intended as binary, except when otherwise explicitly indicated. All the cases listed above are possible when examining nodes in the right operand of the the assignment, while part or them is meaningless or not standard-compliant when the left operand of the assignment is concerned.

Node	Register bound?
identifier, '**sizeof**', literal constant	yes
unary '∗', '[]', '−>' '==', '!=', '<', '>', '<=', '>=', '&&', '||', '+', '−', '∗', '/', '%', '&', '|', '^'	no
unary '−', unary '~', unary '!'	no
unary '+', unary '++', unary '−−'	same as operand
'.'	same as first operand
cast, comma	same as second operand
'=' '+=', '−=', '∗=', '/=' '%=', '&=', '|=', '^=', '<<=', '>>='	recurse on first and second operands, yes, it at least one of them is
'?:'	recurse on second and third operands, yes, it at least one of them is

Table 4.4: Summary of the rules for the determination of the register-boundedness of a given AST node.

The above scheme models correctly the expected behavior of a compiler even in presence of nested assignments like in the following example. Assume the following variable declarations:

```
int   a;
int  *b;
int  *c;
```

Consider the following statement:

```
a = *b = *c;
```

In this a case, the inherent cost of both assignment operators is zero. In fact, expressions '∗b' and '∗c' are register-unbound, and so is expression '∗b = ∗c'. Therefore its translation can be bound to the register bank where 'a' is allocated, with zero cost. Also the assignment between structures or structured data are correctly modeled.

4.4.6 Attribute 'f', translation flavor

In Section 4.3 (page 117) I presented an abstract assembly translation model for the statements and expressions of the C language. That model selects specific translation flavors for each expression, depending on the context where it appears. It is now time to model the effects of this behavior in my attribute grammar. In order to do this, I define attribute f for each expression symbol, which indicates which translation flavor is used for that expression. Attribute f in inherited. In a given node, attribute f depends only on the syntax, on f in the father node, and k in the same node.

Note that statements always have a single, single-exit translation (T).
Attribute f may assume one of the following values:

- N : none
  the current expression is constant; its value is determined at compile time, and its translation is empty;
- T: single-exit flavor;
  the single-entry, single-exit translation is used for the current expression;
- TF: double-exit, jump-if-false flavor;
  the jump-if-false translation is used for the current expression;
- TT: double-exit, jump-if-true flavor;
  the jump-if-true translation is used for the current expression;
- TI: indeterminate flavor;
  in this case, both the jump-if-true and the jump-if-false could be used. Since it is not possible to forecast the behavior of the compiler in this case, nothing better can be done than assuming an average of the two cases.

The attribute is evaluated according to the following rules.

---

1. for all the cases below:
   for each expression child node ⟨*expression-i*⟩ of the current node:
   $\langle \textit{expression-i} \rangle.f = N$; when $\langle \textit{expression-i} \rangle.k = \textit{true}$;

2. '**if** (...) ... ' statement:
   Syntax:
   ⟨*selection_statement*⟩ ::= '`if`' '`(`' ⟨*expression*⟩ '`)`' ⟨*statement*⟩
   Semantics:
   $\langle \textit{expression} \rangle.f = TF$;
   $\langle \textit{statement} \rangle.f = T$;

3. '**if** (...) ... **else** ... ' statement:
   Syntax:
   ⟨*selection_statement*⟩ ::= '`if`' '`(`' ⟨*expression*⟩ '`)`' ⟨*statement-1*⟩ '`else`' ⟨*statement-2*⟩
   Semantics:

⟨*expression*⟩.$f = TI$;
⟨*statement-1*⟩.$f = T$;
⟨*statement-2*⟩.$f = T$;

4. '**while**' statement:
   Syntax:
   ⟨*iteration_statement*⟩ ::= 'while' '(' ⟨*expression*⟩ ')' ⟨*statement*⟩
   Semantics:
   ⟨*expression*⟩.$f = TF$;
   ⟨*statement*⟩.$f = T$;

5. '**do** ... **while**' statement:
   Syntax:
   ⟨*iteration_statement*⟩ ::= '`do`' ⟨*statement*⟩ '`while`' '`(`' ⟨*expression*⟩ '`)`' '**;**'
   Semantics:
   ⟨*statement*⟩.$f = T$;
   ⟨*expression*⟩.$f = TT$;

6. '**for**' statement:
   Syntax:
   ⟨*iteration_statement*⟩ ::= '`for`' '`(`' ⟨*optional_expression-1*⟩ '**;**' ⟨*optional_expression-2*⟩ '**;**' ⟨*optional_expression-3*⟩ '`)`' ⟨*statement*⟩
   Semantics:
   ⟨*optional_expression-1*⟩.$f = T$;
   ⟨*optional_expression-2*⟩.$f = TF$;
   ⟨*optional_expression-3*⟩.$f = T$;
   ⟨*statement*⟩.$f = T$;

7. '**switch**' statement:
   Syntax:
   ⟨*selection_statement*⟩ ::= '`switch`' '(' ⟨*expression*⟩ ')' ⟨*statement*⟩
   Semantics:
   ⟨*expression*⟩.$f = T$;
   ⟨*statement*⟩.$f = T$;

8. comma expression:
   Syntax:
   ⟨*expression*⟩ ::= ⟨*expression-1*⟩ ',' ⟨*assignment_expression*⟩
   Semantics:
   ⟨*expression-1*⟩.$f = T$;
   ⟨*assignment_expression*⟩.f = ⟨*expression*⟩.f;

9. assignment:
   Syntax:
   ⟨*assignment_expression*⟩ ::= ⟨*unary_expression*⟩ '=' ⟨*assignment_expression-1*⟩
   Semantics:
   ⟨*unary_expression*⟩.$f = T$;
   ⟨*assignment_expression-1*⟩.$f = T$;

10. logical 'and' operator:
    Syntax:
    ⟨*logical_and_expression*⟩ ::= ⟨*logical_and_expression-1*⟩ '`&&`' ⟨*inclusive_or_expression*⟩
    Semantics:
    ⟨*logical_and_expression-1*⟩.$f = TF$;

$\langle inclusive\_or\_expression \rangle.f =$
$$\begin{cases} TI & \text{when } \langle logical\_and\_expression \rangle.f = T; \\ \langle logical\_and\_expression \rangle.f & \text{else} \end{cases}$$

11. logical 'or' operator:
Syntax:
$\langle logical\_or\_expression \rangle$ ::= $\langle logical\_or\_expression\text{-}1 \rangle$ '||' $\langle logical\_and\_expression \rangle$
Semantics:
$\langle logical\_or\_expression\text{-}1 \rangle.f = TT;$
$\langle logical\_and\_expression \rangle.f =$
$$\begin{cases} TI & \text{when } \langle logical\_or\_expression \rangle.f = T; \\ \langle logical\_or\_expression \rangle.f & \text{else} \end{cases}$$

12. logical 'not operator:
Syntax:
$\langle unary\_expression \rangle$ ::= '!' $\langle cast\_expression \rangle$
Semantics:
$\langle cast\_expression \rangle.f =$
$$\begin{cases} TF & \text{when } \langle unary\_expression \rangle.f = TT; \\ TT & \text{when } \langle unary\_expression \rangle.f = TF; \\ TI & \text{when } \langle unary\_expression \rangle.f = T; \end{cases}$$

13. relational operators:
Syntax:
$\langle re\_expression \rangle$ ::= $\langle re\_expression\text{-}1 \rangle$ $\langle re\_op \rangle$ $\langle re\_expression\text{-}2 \rangle$
where $\langle re\_op \rangle$ ::= '==' | '!=' | '>' | '<' | '>=' | '<=' ;
(generalized syntax).
Semantics:
$\langle re\_expression\text{-}1 \rangle.f = T;$
$\langle re\_expression\text{-}2 \rangle.f = T;$

14. arithmetic expressions:
Syntax:
$\langle m\_expression \rangle$ ::= $\langle m\_expression\text{-}1 \rangle$ $\langle m\_op \rangle$ $\langle m\_expression\text{-}2 \rangle$
where $\langle m\_op \rangle$ ::= '&' | '|' | '«' | '»' | '+' | '−' | ... ;
(generalized syntax).
Semantics:
$\langle m\_expression\text{-}1 \rangle.f = T;$
$\langle m\_expression\text{-}2 \rangle.f = T;$

15. expression statements;
Syntax:
$\langle expression\_statement \rangle$ ::= $\langle expression \rangle$ ';'
Semantics:
$\langle expression \rangle.f =$ the least expensive among T, TF, TT.

16. copy rules for expressions:
Syntax:
any copy rule in Section 4.5.1 (page 231), generalized as:
$\langle r\_expression \rangle$ ::= $\langle l\_expression \rangle$
Semantics:
$\langle l\_expression \rangle.f = \langle r\_expression \rangle.f;$

4.4.7 Attribute 'ci', inherent cost

The attribute *ci* of a given expression represents the cost of the abstract assembly instructions in the translation of the given expression or statement, which are strictly required to produce the result. They include data manipulation and data transfer. They do not include any type conversion or execution flow control task.

Attribute *ci* is synthesized, and it is function of attributes t, v, r, k and f in the same node. Attribute *ci* is zero for a vast majority of the statement symbols, and non-zero for a vast majority of the expression symbols.

4.4.7.1 The 'sizeof' operator

The value of an expression composed by a '**sizeof**' operator is always determined at compile time. Therefore its runtime cost is zero.

Syntax:

⟨*unary_expression*⟩ ::= 'sizeof' ⟨*unary_expression-1*⟩
| 'sizeof' '(' ⟨*type_name*⟩ ')'

Semantics:

⟨*unary_expression*⟩.*ci* = 0;

4.4.7.2 Comma operator

The comma operator has two operand expressions. The left operand is evaluated as a void expression (i.e., its result is discarded), then the right operand is evaluated, and its return value is used as return value for the comma expression.

The assembly translation of a comma expression is therefore the mere sequential concatenation of the translations of its left and right operands respectively, without any other code added. Therefore the inherent (and also flow) cost of the comma operator itself is zero.

Syntax:

⟨*expression*⟩ ::= ⟨*expression-1*⟩ ',' ⟨*assignment_expression*⟩

Semantics:

⟨*expression*⟩.*ci* = 0;

4.4.7.3 The cast operator

The cast operator explicitly requests a type conversion. If this conversion has any cost, it is accounted for in attribute *cc* according to the rules is Section 4.4.8.4 (page 220).

Syntax:

⟨*cast_expression*⟩ ::= '(' ⟨*type_name*⟩ ')' ⟨*cast_expression*⟩

Semantics:

$$\langle \textit{cast\_expression} \rangle.ci = 0;$$

4.4.7.4 The logical 'and' and 'or' operators

Syntax:

⟨*logical_and_expression*⟩ ::= ⟨*logical_and_expression-1*⟩ '&&' ⟨*inclusive_or_expression*⟩

⟨*logical_or_expression*⟩ ::= ⟨*logical_or_expression-1*⟩ '||' ⟨*logical_and_expression*⟩

Generalized syntax:

⟨*logical_expression*⟩ ::= ⟨*logical_expression-1*⟩ ⟨*logical_operator*⟩ ⟨*logical_expression-2*⟩

Semantics:

$$\langle \textit{logical\_expression} \rangle.ci = \begin{cases} 1 \text{ LogicTop} & \text{if } \langle \textit{expression} \rangle.f = T \\ 0 & \text{else} \end{cases}$$

4.4.7.5 The logical 'not' operator

Syntax:

⟨*unary_expression*⟩ ::= '!' ⟨*cast_expression*⟩

Semantics:

$$\langle \textit{unary\_expression} \rangle.ci = \begin{cases} 1 \text{ LogicTop} & \text{if } \langle \textit{unary\_expression} \rangle.f = T \\ & \quad \wedge \neg \langle \textit{cast\_expression} \rangle.k; \\ 0 & \text{else} \end{cases}$$

Note that the operand type is not influential. The cost of zero-testing the operand is already accounted for in the cost of the operand node, depending on its flavor. A LogicTop atom cost is 1 mov + 0.5 jump.

4.4.7.6 Unary arithmetic operators

Syntax:

⟨*postfix_expression*⟩ ::= ⟨*postfix_expression*⟩ '++'
| ⟨*postfix_expression*⟩ '−−'

⟨*unary_expression*⟩ ::= '++' ⟨*unary_expression*⟩
| '−−' ⟨*unary_expression*⟩
| '+' ⟨*cast_expression*⟩
| '−' ⟨*cast_expression*⟩

Generalized syntax:

⟨*postfix_expression*⟩ ::= ...

Semantics:

$$\langle expression \rangle.ci = \begin{cases} 1 \text{ IntCompare} & \text{if } \langle expression \rangle.t \text{ is integral} \\ 1 \text{ FloatCompare} & \text{if } \langle expression \rangle.t \text{ is floating-point} \end{cases}$$

Exception: if the operator is '+', the cost is zero.

4.4.7.7 Identifiers

Syntax:

⟨*primary_expression*⟩ ::= IDENTIFIER

Semantics:

$$\langle primary\_expression \rangle.ci = \begin{cases} 1\text{IntCompare}+1\text{LogicLeaf} & \text{if } \langle primary\_expression \rangle.t \text{ is integral} \wedge \\ & \langle primary\_expression \rangle.f \in \{TF, TT, TI\} \\ 1\text{FloatCompare}+1\text{LogicLeaf} & \text{if } \langle primary\_expression \rangle.t \text{ is floating-point} \wedge \\ & \langle primary\_expression \rangle.f \in \{TF, TT, TI\} \\ 0 & \text{else;} \end{cases}$$

4.4.7.8 Arithmetical and bitwise expressions

Syntax

⟨*inclusive_or_expression*⟩ ::= ⟨*inclusive_or_expression*⟩ '|' ⟨*exclusive_or_expression*⟩

⟨*exclusive_or_expression*⟩ ::= ⟨*exclusive_or_expression*⟩ '^' ⟨*and_expression*⟩

⟨*and_expression*⟩ ::= ⟨*and_expression*⟩ '&' ⟨*equality_expression*⟩

⟨*unary_expression*⟩ ::= '~' ⟨*cast_expression*⟩

⟨*shift_expression*⟩ ::= ⟨*shift_expression*⟩ '«' ⟨*additive_expression*⟩
| ⟨*shift_expression*⟩ '»' ⟨*additive_expression*⟩

⟨*multiplicative_expression*⟩ ::= ⟨*multiplicative_expression-1*⟩ '*' ⟨*cast_expression*⟩
| ⟨*multiplicative_expression-1*⟩ '/' ⟨*cast_expression*⟩
| ⟨*multiplicative_expression-1*⟩ '%' ⟨*cast_expression*⟩

⟨*additive_expression*⟩ ::= ⟨*additive_expression*⟩ '+' ⟨*multiplicative_expression*⟩
| ⟨*additive_expression*⟩ '−' ⟨*multiplicative_expression*⟩

Generalized syntax:

⟨*expression*⟩ ::= ⟨*expression-1*⟩ ⟨*op*⟩ ⟨*expression-2*⟩

Semantics:

$$\langle expression \rangle.ci == \begin{cases} \text{see Table 4.5} & \text{if none of the operands has pointer type} \\ \text{see Table 4.6} & \text{else;} \end{cases}$$

Please note that the choice to distinguish the cost in atoms of the various operations between integer an floating-point operations is arbitrary. I make this choice here for sake of simplicity, although the implemented tools which accompany this thesis allow much more refined classification of costs.

Also notice that for atoms corresponding to floating point operations, it could be impossible to derive their cost analytically in terms of abstract instructions when the target platform has no floating-point unit (FPU) and the floating-point operations are emulated. The entire Appendix A.

⟨*op*⟩	⟨*expression*⟩.*t*	⟨*expression*⟩.*ci*
'|', '^', '&' or '~'	is integral	1 BitwiseOperation
'«', '»'	is integral	1 BitwiseShift
'*'	is integral	1 IntMul
'*'	is floating-point	1 FloatMul
'/'	is integral	1 IntDiv
'/'	is floating-point	1 FloatDiv
'%'	is integral	1 IntModulo
'%'	is floating-point	(illegal)
'+'	is integral	1 IntAdd
'+'	is floating-point	1 FloatAdd
'−'	is integral	1 IntSub
'−'	is floating-point	1 FloatSub

Table 4.5: The inherent cost of arithmetical and bitwise operators, depending on the resulting type.

Also note that Table 4.6 reports all the possible cases of arithmetic operators which involve pointers. No other possibilities (e.g. pointer + pointer) are legal.

Alignment	Operation		
	ptr + arith	ptr - arith	ptr - ptr
Byte	PtrIntAddByte = 1 add = 1 alul	PtrIntSubByte = 1 sub = 1 alul	PtrPtrSubByte = 1 sub = 1 alul
Aligned	PtrIntAddAligned = 1 shr + 1 add = 2 alul	PtrIntSubAligned = 1 shr + 1 sub = 2 alul	PtrPtrSubAligned = 1 sub + 1 shl = 2 alul
Misaligned	PtrIntAddMisaligned = 1 IntMul + 1 add = 1 alul + 1 IntMul	PtrIntSubMisaligned = 1 IntMul + 1 sub = 1 alul + 1 IntMul	PtrPtrSubAligned = 1 sub + IntDiv = 1 alul + IntDiv

Table 4.6: The inherent costs of pointer arithmetic expressions, expressed in atoms, abstract assembly instructions and corresponding classes of instructions.

4.4.7.9 The unary dereferencing operator, '*'

The unary operator '*' takes as an operand an expression of type pointer. In order to determine its cost in terms of kernel instructions, and then atoms, I will examine a number of cases, and find a general form in the end.

The grammar rule corresponding to this operator is:

⟨*unary_expression*⟩ ::= '$*$' ⟨*cast_expression*⟩

the factors affecting the cost of this operator are:

- the valueness of the entire expression, i.e. ⟨*unary_expression*⟩.v;
- the word size of the transferred data, i.e. $W($⟨*unary_expression*⟩.$r)$.

I analyze some remarkable cases, in order of increasing complexity and generality:

1. **single-word value, R-value used**:
   Assume the following declarations:

   ```
   int   a;
   int * b
   ```

   and consider the following statement:

   ```
   a = *b;
   ```

   The abstract assembly translation of the above statement consists in a single mvld instruction, which transfers into register 'a' the single word at the location stored in register 'b'. For this use, I define the atom RValueStar = 1 mvld. The cost determination process decorates the expression '$*$b' with 1 RValueStar atom.

2. **single-word value, L-value used**: Assume the same declarations as above, and consider the statement:

   ```
   *b = a;
   ```

   Its translation consists in a single mvst instruction, which transfers the single-word contents of register 'a' into the location currently stored in register 'b'. For this use, I define the atom LValueStar = 1 mvst. The cost determination process decorates the expression '$*$b' with 1 LValueStar atom.

3. **multiple-word value, R-value used**:
   Assume the following declarations:

   ```
   type   a;
   type * b
   ```

   where 'type' is such that $W($type$) \geq 1$. Consider again the statement:

```
a = *b;
```

The translation of the above statement consists in $W(\text{type}) = w$ mvld instructions, which transfer the contents of multiple consecutive registers, the first of which is contained in the register associated with b, into the multiple consecutive registers associated with a, plus the increments instructions required to advance pointer b :

```
mvld   a,      b      ; transfers the first word
add    t, b, #4       ; stores the address of the second word in 't'
mvld   (a+1), t       ; transfers the second word
add    t, t, #4       ; stores the address of the third word in 't'
mvld   (a+2), t       ; transfers the third word
 ...
add    t, t, #4       ; stores the address of the last word in 't'
mvld   (a+w-1), t     ; transfers the last word
```

Please note that 'a+1', 'a+2', ..., 'a+w-1' are the names of the w registers associated to variable 'a' by the compiler, known at compile-time. They therefore do not imply any displacement calculation at runtime. On the other hand, 'b' needs to be incremented by the word size after each transfer. The cost of the construct is therefore given by the cost of w mvld instructions plus $w - 1$ integer increment instructions, of class alul. I render this expression in terms of atoms by defining atom RValueStarNext = 1 mvld + 1 alul, and I decorate expression '*b' with 1 RValueStar atom, and $(w - 1)$ RValueStarNext atoms.

4. **multiple-word value, L-value used**:
Assume the same declarations as above and consider the statement:

```
*b = a;
```

Its translation consists in $W(\texttt{type}) = w$ mvst instructions, which transfer the contents of the multiple consecutive registers associated with variable 'a' into the multiple consecutive registers whose addresses start at the location pointed by 'b', plus the increments instructions required to advance pointer 'b:

```
mvst   b, a           ; transfers the first word
add    t, b, #4       ; stores the address of the second word in 't'
mvst   t, (a+1)       ; transfers the second word
add    t, t, #4       ; stores the address of the third word in 't'
mvst   t, (a+2)       ; transfers the third word
 ...
add    t, t, #4       ; stores the address of the last word in 't'
mvst   t, (a+w-1)     ; transfers the last word
```

Again, 'a+1', 'a+2', ..., 'a+w-1' are the names of the w registers associated to variable 'a' by the compiler, known at compile-time. They therefore do not imply any displacement calculation at runtime. The cost of the construct is given by the cost of w mvst instructions plus $w - 1$ integer increment instructions, of class alul. I render this expression in terms of atoms by defining atom LValueStarNext = 1 mvst + 1

alul, and I decorate expression '∗b' with 1 LValueStar atom $+(w-1)$ LValueStarNext atoms.

5. **single-word value, RL-valueness**:
Assume the following declarations:

```
int   a;
int * b
```

and consider the statement:

```
*b += a;
```

The above statement must be equivalent in cost and semantics to the following one:

```
*b = *b + a;
```

and the abstract assembly translation for both statement is as follows:

```
mvld   t, b
add    t, t, a
mvst   b, t
```

The cost, in terms of abstract instructions is therefore = 1 mvst + 1 mvld + 1 alul. However, the grouping of the above cost in terms of atoms vary in the two statements, since atoms are designed to decorate parse trees, and the parse trees of '∗b += a' and '∗b = ∗b + a' are different, as illustrated in Figure 4.23 (page 190)

The atom decoration corresponding to the parse tree of expression '∗b = ∗b + a', obtained applying the rules determined above, is shown in Figure 4.24 on the left. Now I must devise an appropriate atom decoration for the parse tree of expression '∗b += a', which yields the same cost. In order to do that, the parse node corresponding to subexpression '∗b' must have a cost which equals 1 RValueStar plus 1 LValueStar. I define the new atom RLValueStar = 1 mvld + 1 mvst, and I associate this atom to subexpression '∗b'.

6. **multiple-word value, RL-valueness**:
I discuss now how to extend the above considerations to the case in which 'a' and 'b' are declared as follows:

```
type   a;
type * b
```

where 'type' is an arithmetic multiple-word type (such as double, long double, long long int) of word size $W(\text{type}) = w$. The abstract assembly translation of statement

```
*b += a;
```

in this case is:

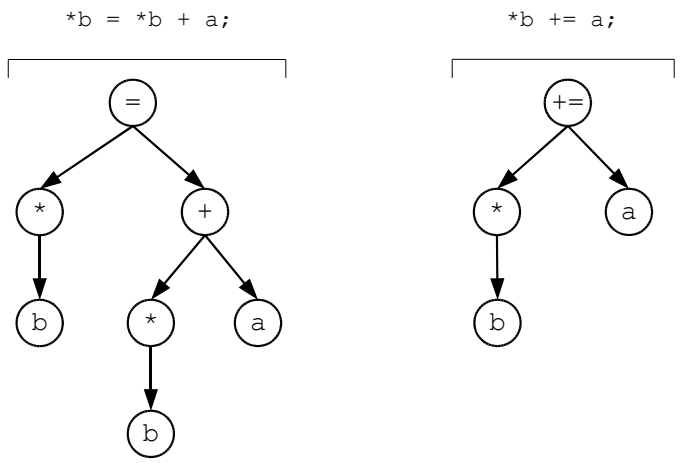


Figure 4.23: ASTs of two example expressions involving the '$*$' operator. The two expressions have the same semantics and effects (as far as inherent cost is concerned), but different parse tree.

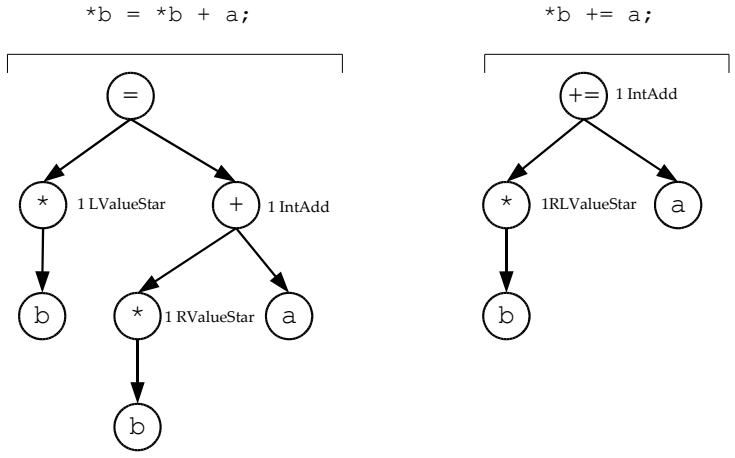


Figure 4.24: Inherent cost decoration for the two previous example expressions. The two expressions have the same cumulative cost, but costs associated to individual nodes may be different.

```
mvld    c,      b                   ; transfers the first word
```

```
add    t_1,   b,    #4          ; calculates address of the second word
add    t_2,   t_1,  #4          ; calculates address of the third word
...                             ; ...
add    t_w−1, t_w−2, #4         ; calculates address of the last word

mvld   c+1,   t_1               ; transfers the second word of 'b'
mvld   c+2,   t_2               ; transfers the third word of 'b'
...                             ; ...
mvld   c+w−1, t_w−1             ; transfers the third word of 'b'

...                             ; code required to sum values in register banks 'c' and 'a'
                                ; result is left in registers 'c' ... 'c+w−1'

mvst   b,     c                 ; writes the first word of result back to 'b'
mvst   t_1,   c+1               ; writes the second word of result back to 'b'
mvst   t_2,   c+2               ; writes the third word of result back to 'b'
...                             ; ...
mvst   t_w=1, c+w−1             ; writes the last word of result back to 'b'
```

and, its cost in terms of abstract assembly instructions is w mvst + w mvld + $(w-1)$ alul, plus the cost of the multiple-word addition, which must be determined according to the rules for the '+' operator described before.

In order to solve the problem in a way that is consistent with the previous cases (thus allowing generalization), I introduce a new atom RLValueStarNext = 1 mvld + 1 mvst + 1 alul. The cost determination process decorates expression '$*$b' with 1 RLValueStar atom and $(w-1)$ RLValueStarNext atoms. Please note that RLValueStarNext $\neq$ RValueStarNext + LValueStarNext.

All the above cases can be expressed in a general form, which is summarized below.

Syntax:

⟨*unary_expression*⟩ ::= '$*$' ⟨*cast_expression*⟩

Semantics:

$$\langle \textit{unary\_expression} \rangle.ci = 1\,.v\text{ValueStar} + (W(.r) - 1)\,.v\text{ValueStarNext}$$

(For sake of brevity, I have abbreviated ⟨*unary_expression*⟩.v with .v and ⟨*unary_expression*⟩.r with .r.)

The following example illustrates how the inherent cost determination works in presence of complex expression involving nested unary '$*$' operators and multiple-word types. Assume the following declarations:

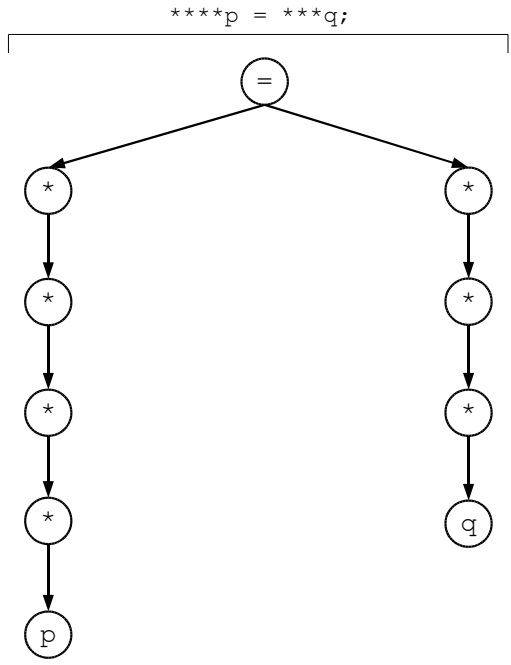


Figure 4.25: AST of an example expression involving multiple nested '∗' operators.

double **** p;
double *** q;

Consider the following statement:

**** p = *** q;

The AST for this expression is given in Figure 4.25. A decorated AST where attributes v, t, r and ci have been determined appears in Figure 4.26. A detailed cost breakdown for one of the operators is shown in 4.27.

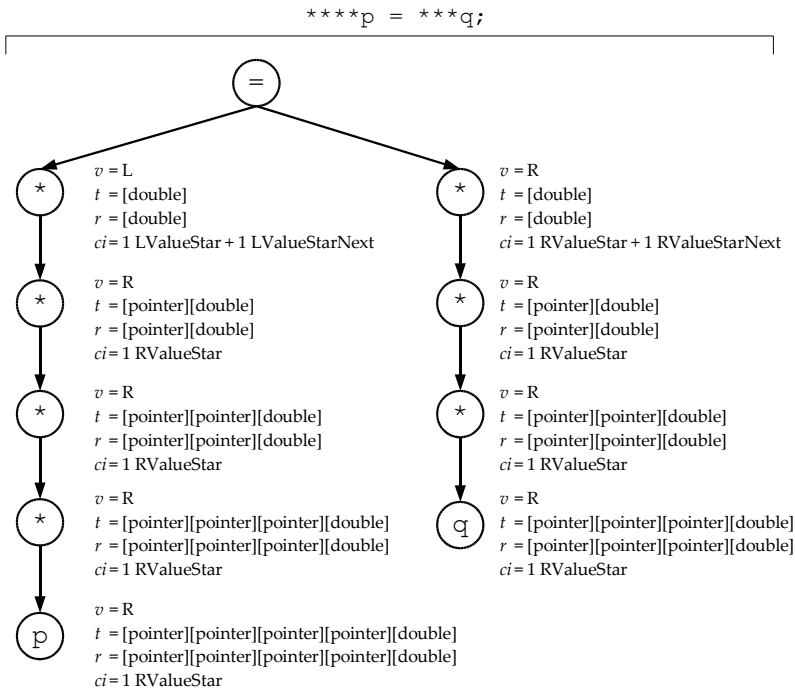


Figure 4.26: Inherent cost determination for an example expression involving multiple instances of '$*$' operators.

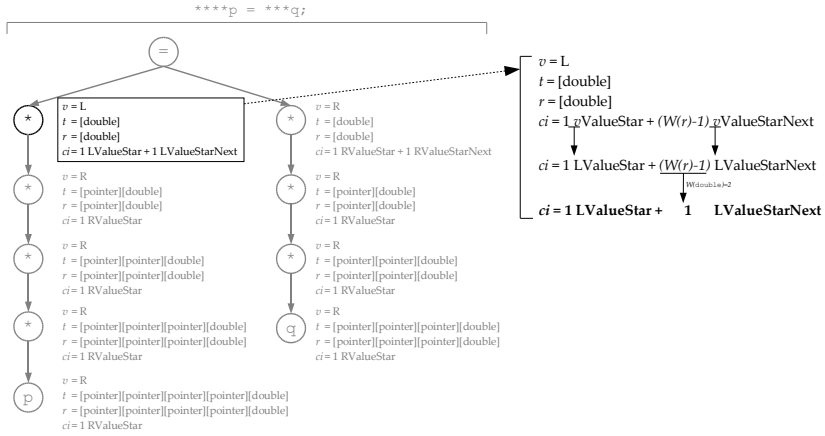


Figure 4.27: Inherent cost determination for an example expression involving multiple instances of '$*$' operators. Detail.

4.4.7.10 The subscript operator '[]'

The subscript operator '[]' allows to access the cells of an array. It has two operands. The left operand must be of pointer or array type, the right operand of integral type. It has the following syntax:

⟨*postfix_expression*⟩ ::= ⟨*postfix_expression-1*⟩ '[' ⟨*expression*⟩ ']'

The factors which affect its cost are:

- the valueness of the entire expression ⟨*postfix_expression*⟩.*v* (not the first child's one, ⟨*postfix_expression-1*⟩.*v*);
- the word size of the transferred value of the entire expression, *W*(⟨*postfix_expression*⟩.*r*);
- the alignability of the byte size of the transferred value of the entire expression, *A*(⟨*postfix_expression*⟩.*r*).

Since the possible valuenesses are 4, the possible cases for word size are 2 (single word or multiple word) and the alignabilities are 3, I should enumerate 24 different cases. This is impractical. It is more practical to split the process of array subscripting in three independent phases, and consider each of them individually:

1. resolving the address of the first word; this phase is affected by alignability only; during this phase I assume that the result of the expression used as an index is ready and available in register 'i', and I leave the result (i.e. the address of the first word of cell a[i] in register 't_0');
2. calculating the addresses of the remaining words; this phase is affected primarily by the word size, and it is skipped completely if valueness is Z. If valueness is Z, there is no point in calculating the addresses of all the words of cell a[i], since those words will not be transferred, and the only useful address was already calculated before. When valueness is not Z, the abstract assembly translation of this phase and its cost are independent of the valueness. At the beginning of this phase I assume that the address of the first word of cell a[i], just calculated in the previous phase, is available in register 't_0', and I calculate the addresses of the following words, leaving them in registers 't_1', 't_2' ... , 't_w−1'.
3. transferring the data; this phase assumes that there are as many 't_n' registers ready with the addresses of the words to transfer. The abstract assembly translation of this phase is composed by a number of transfers which is equal to the word size of the cell type, and each transfer is composed by respectively no instructions, one mvld instruction, one mvst instruction, one mvld and one mvst instructions, if the valueness is respectively Z, R, L, RL.

For all the following analyses, I consider 'a' and 'b' declared as:

```
type a [...];
type b;
```

The inherent cost of the three phases is as follows:

1. **resolving the address of the first word**:
   This operation is always required, disrespectfully of the valueness of the expression, if the valueness is R, the result of this calculation will serve as source address for phase 3; if it is L, it will serve as a destination address for phase 3; if it is RL, it will serve as source and destination address for phase 3; if it is Z, it will be the final result of the expression including the current one. Additionally, this address, and the way it is calculated do not depend on valueness and word size.

   Mind that expressions `a[i]` and `*(a+i)` are always equivalent. Calculating the address of array cell `a[i]` is the same operations as evaluating the '+' subexpression in `*(a+i)`, which is a pointer arithmetic operation. Since I have already discussed the cost of pointer arithmetic in Section 4.4.7.8 (page 185), I will stick here to the same naming conventions and atoms.

   (a) **alignability is byte**: The abstract assembly translation of this phase is:

   ```
   add   t_0, a, i    ; the offset is equal to the index
   ```

   and its cost is 1 add = 1 alul = 1 PtrIntAddByte;

   (b) **alignability is aligned**: The translation of this phase is:

   ```
   shl   t_0, i, ⟨log sizeof type⟩    ; calculates the offset
   add   t_0, a, i                    ; adds the base address
   ```

   and its cost is 1 shl + 1 add = 2 alul = 1 PtrIntAddAligned;

   (c) **alignability is misaligned**: The translation of this phase is:

   ```
   mul   t_0, i, ⟨sizeof type⟩        ; calculates the offset
   add   t_0, a, i                    ; adds the base address
   ```

   and its cost is 1 mul + 1 add = 1 alul + 1 aluh = 1 PtrIntAddMisaligned;

   Summarizing, the cost of this phase is 1 PtrIntAdd$A(r)$;

2. **calculating the addresses of the remaining words**:
   the translation of this phase is empty if the valueness is Z (and its cost is 0), otherwise it is composed by $W(r) - 1$ instructions as follows:

   ```
   add   t_1, t_0, #4          ; calculates the address of the second word
   add   t_2, t_1, #4          ; calculates the address of the third word
   ...                         ; ...
   add   t_w−1, t_w−2, #4      ; calculates the address of the last word
   ```

   Summarizing, the cost of this phase is $(W(r) - 1)$ add = $(W(r) - 1)$ alul instructions.

3. **transferring the data**:
   the translation of this phase is composed as follows:

   (a) **valueness is Z**: The translation of this phase is empty, and its cost is zero.

   (b) **valueness is R**: The translation of this phase is as follows:

```
mvld  c, t_0            ; transfers the first word
mvld  c+1, t_1          ; transfers the second word
 ...
mvld  c+w−1, t_w−1 ; transfers the last word
```

   and its cost is $W(r)$ mvld instructions;

   (c) **valueness is L**: The translation of this phase is as follows:

```
mvst  t_0, c            ; transfers the first word
mvst  t_1, c+1          ; transfers the second word
 ...
mvst  t_w−1, c+w−1 ; transfers the last word
```

   and its cost is $W(r)$ mvst instructions;

   (d) **valueness is RL**: The translation of this composition of the translations for R and L valueness and its cost is $W(r)$ (mvld+mvst);

Obtaining a general formula. Generalizing over $A(r)$ and $W(r)$ is easy. Now I generalize over $W(r)$.

Z:	1 PtrIntAdd$A(r)$	
R:	1 PtrIntAdd$A(r)$	$+(W(r)-1)$ alul $+W(r)$ mvld
L:	1 PtrIntAdd$A(r)$	$+(W(r)-1)$ alul $+W(r)$ mvst
RL:	1 PtrIntAdd$A(r)$	$+(W(r)-1)$ alul $+W(r)$ (mvld+mvst)

Which can be rewritten as:

Z:	1 PtrIntAdd$A(r)$	+1 0	$+(W(r)-1)$ 0
R:	1 PtrIntAdd$A(r)$	+1 mvld	$+(W(r)-1)$ (alul + mvld)
L:	1 PtrIntAdd$A(r)$	+1 mvst	$+(W(r)-1)$ (alul + mvst)
RL:	1 PtrIntAdd$A(r)$	+1 (mvld+mvst)	$+(W(r)-1)$ (alul + mvld + mvst)

Before rewriting the final form, we introduce the following atoms:

ZValueIndex	= 0
RValueIndex	= 1 mvld
LValueIndex	= 1 mvst
RLValueIndex	= 1 mvld + 1 mvst
ZValueIndexNext	= 0
RValueIndexNext	= 1 alul + 1 mvld
LValueIndexNext	= 1 alul + 1 mvst
RLValueIndexNext	= 1 mvld + 1 mvst

The final, general form is given below.

⟨*postfix_expression*⟩ ::= ⟨*postfix_expression-1*⟩ '[' ⟨*expression*⟩ ']'
Semantics:

$$.ci = 1 \text{ PtrIntAdd } A(.r) + 1.v \text{ ValueIndex } + (W(.r) - 1).v \text{ ValueIndexNext}$$

(All the attributes (c, v and r) are attributes of symbol ⟨*postfix_expression*⟩, which has been omitted for brevity.)

The above general rule does not compose correctly when evaluating multiple subscript operators, used to access cells of multi-dimensional arrays. This topic is beyond the scope of this document. To learn more on this, you are invited to refer to the source code of the project which implements this thesis.

4.4.7.11 The access to member of pointed compound operator '->'

The arrow operator allows to access members of a structure or a union pointed by a given pointer. It has the following syntax:

⟨*postfix_expression*⟩ ::= ⟨*postfix_expression-1*⟩ '->' IDENTIFIER

I consider it as a binary operator, whose two operands are: a first (or left) operand, which must be an expression of type pointer to a struct or a union, and a second (or right) operand, which must be an identifier among the ones declared in the symbol table associated with the struct or union. This operator is considered as a postfix unary operator by the C Standard, and the identifier is not considered as a second operand, but as a part of the operator itself. This disagreement is just a matter of taxonomy, and arbitrarily choosing one convention or the other is equivalent.

The inherent cost determination must ensure that expression:

```
a->b
```

has exactly the same cost as:

```
(*a).b
```

I consider the following cases:

1. **access to union, R-value taken, single-word member**:
   Assume the following type declarations:

   ```
   union {
     ...
    int m;
     ...
   } * a;
   int b;
   ```

   (where 'm' and 'b' could be declared as any other single-word type in place of **int**) and consider the following statement:

```
b = a->m;
```

The abstract assembly translation of the above statement is:

```
mvld  b, a
```

i.e., a single mvld instruction, which copies the contents of the memory cell pointed by 'a' into 'b'. There is no need to calculate the address of 'm', since its displacement with respect to the beginning of the union where it belongs is always zero. I define a new atom RValueUnionArrow = 1 mvld, and the cost of symbol ⟨*postfix_expression*⟩ to 1 RValueUnionArrow.

2. **access to union, R-value taken, multiple-word member**:
Assume the following type declarations:

```
union {
  ...
 type m;
  ...
} * a;
type b;
```

(where 'type' is any multiple-word type) and consider the following statement:

```
b = a->m;
```

The translation of the above statement is:

```
mvld   b, a          ; transfers the first word
add    t, a, 4       ; calculates the address of the second word
mvld   (b+1), t      ; transfers the second word
add    t, t, 4       ; calculates the address of the third word
mvld   (b+2), t      ; transfers the third word
 ...
add    t, t, 4       ; calculates the address of the last word
mvld   (b+w-1), t    ; transfers the last word
```

The cost of the translation is composed by 1 initial mvld instruction (equal to 1 RValueUnionArrow), plus a group composed by an add and a mvld instruction, of cost 1 alul + 1 mvld, repeated as many times as the number of words to transfer minus one. Please note that b+1, b+2, ... denote the names of the consecutive registers located after b, whose names are known at static time by the compiler, and do not involve any address calculation operation at runtime. I introduce a new atom, called RValueUnionArrowNext, to capture the cost of the above repeated group, 1 alul + 1 mvld. I associate to symbol ⟨*postfix_expression*⟩ the cost of 1 RValueUnionArrow $+W(r) - 1$ RValueUnionArrowNext, where r is attribute r of ⟨*postfix_expression*⟩.

3. **access to union, L-value taken, single-word member**:
Assume the following type declarations:

```
union {
  ...
 int m;
  ...
} * a;
int b;
```

(where 'm' and 'b' could be declared as any other single-word type in place of **int**) and consider the following statement:

```
a->m = b;
```

The translation of the above statement is:

```
mvst a, b
```

i.e., a single mvst instruction, which copies the contents of register 'b' into the memory cell pointed by 'a'. Again, there is no need to calculate the address of 'm', since its displacement with respect to the beginning of the union where it belongs is always zero. I define a new atom LValueUnionArrow = 1 mvst, and the cost of symbol ⟨*postfix_expression*⟩ to 1 LValueUnionArrow.

4. **access to union, L-value taken, multiple-word member**:
Assume the following type declarations:

```
union {
  ...
 type m;
  ...
} * a;
type b;
```

(where 'type' is any multiple-word type) and consider the following statement:

```
a->m = b;
```

The translation of the above statement is:

```
mvst   a, b          ; transfers the first word
add    t, a, 4       ; calculates the destination address for the second word
mvst   t, (a+1)      ; transfers the second word
add    t, t, 4       ; calculates the destination address for the third word
mvst   t, (a+2)      ; transfers the third word
 ...
add    t, t, 4       ; calculates the destination address for the last word
mvst   t, (a+w-1)    ; transfers the last word
```

The cost of the translation is composed by 1 initial mvst instruction (equal to 1 LValueUnionArrow), plus a group composed by an add and a mvst instruction, of cost 1 alul + 1 mvst, repeated as many times as the number of words to transfer minus one. Please note that 'a+1', 'a+2', ... denote the names of the consecutive registers located after 'a', whose names are known at static time by the compiler, and

do not involve any address calculation operation at runtime. I introduce a new atom, called LValueUnionArrowNext, to capture the cost of the above repeated group, 1 alul + 1 mvst. I associate to symbol ⟨*postfix_expression*⟩ the cost of 1 LValueUnionArrow $+W(r) - 1$ RValueUnionArrowNext, where r is attribute r of ⟨*postfix_expression*⟩.

5. **access to union, RL-value taken, single-word member**:
Assume the following type declarations:

```
union {
  ...
 int m;
  ...
} * a;
int b;
```

(where 'm' and 'b' could be declared as any other single-word type in place of **int**) and consider the following statement:

```
a->m += b;
```

(or any other statement obtained by replacing '+=' with a type-compliant compound assignment operator). The translation of the above statement is:

```
mvld    t, a        ; load 'm' in a temporary register
add     t, t, b     ; perform the addition
mvst    a, t        ; store the result in 'm' again
```

The translation transfers the contents of 'a->m' into a temporary register, then performs the summation and finally transfers it back. Again, there is no need to calculate the address of 'm', since its displacement with respect to the beginning of the union where it belongs is always zero. The cost associated to the arrow operator comprises the mvld and the mvst instructions. The add instruction is, in this case, already accounted for in the cost evaluation for operator '+='. I define a new atom RLValueUnionArrow = 1 mvld + 1 mvst, and the cost of symbol ⟨*postfix_expression*⟩ to 1 RLValueUnionArrow.

6. **access to union, RL-value taken, multiple-word member**:
Assume the following type declarations:

```
union {
  ...
 type m;
  ...
} * a;
type b;
```

(where 'type' is any multiple-word type such that $W(\text{type}) = w$) and consider the following statement:

```
a->m += b;
```

(or any other statement obtained by replacing '+=' with a type-compliant compound assignment operator). The translation of the above statement is:

```
mvld   c, a                ; transfers the first word of 'm' into a temporary register

add    t_1, a, 4           ; calculates the address of the second word
add    t_2, t_1, 4         ; calculates the address of the third word
 ...
add    t_w−1, t_w−2, 4 ;   calculates the address of the last word

mvld   (c+1), t_1          ; transfers the second word of 'm' into a temporary register
mvld   (c+2), t_2          ; transfers the third word of 'm' into a temporary register
 ...
mvld   (c+w−1), t_w−1 ;    transfers the last word of 'm' into a temporary register

 ...                       ; appropriate translation for the operation between
                           ; value in registers c ... c+w−1 and b...?;
                           ; result left in register d ... d+w−1

mvst   a, d                ; transfers the first word of result into 'm'
mvst   t_1, (d+1)          ; transfers the second word of result into 'm'
mvst   t_2, (d+2)          ; transfers the third word of result into 'm'
 ...
mvst   t_w−1, (d+w−1) ;    transfers the last word of result into 'm'
```

Please note that register banks 'c', 'c+1', 'c+2' ... 'c+w-1' and 'd', 'd+1', 'd+2' ... 'd+w-1' are banks of consecutive registers containing multiple word values encoded according to encoding rules for type 'type', whose names are known at static time by the compiler, and do not involve any address calculation operation at runtime. Registers 't_1', 't_2' ... 't_w−1' are temporary registers containing the addresses of the multiple cells containing the value of 'a−>m', from the second one to the last one. They are calculated once, when their value is accessed and they are reused, when the final result is stored.

The cost of the translation is composed by 1 mvld instruction, $W(r) - 1$ add instructions of cost 1 alul each, then $W(r) - 1$ mvld instructions, then $W(r) - 1$ mvst instructions. I group the cost of the 1 mvld and 1 mvst instruction in a RLValueUnionArrow, already defined for the previous case; then I introduce a new atom, called RLValueUnionArrowNext, to capture the cost of the remaining instructions, grouped as 1 alul + 1 mvld + 1 mvst. I associate to symbol $\langle postfix\_expression \rangle$ the cost of 1 RLValueUnionArrow $+W(r) - 1$ RLValueUnionArrowNext, where r is attribute r of $\langle postfix\_expression \rangle$.

Please note that: RLValueUnionArrow = RValueUnionArrow + LValueUnionArrow, but RLValueUnionArrowNext $\neq$ RValueUnionArrowNext + LValueUnionArrowNext, that is, the cost of an access with RL valueness is not, in general, given by the sum of costs for R and L valueness.

7. **access to union, Z-value taken, single- or multiple-word member**: Assume the following type declarations:

```
union {
  ...
 type m;
  ...
} * a;
```

(where 'type' has no restrictions) and consider the following expression:

```
& a->m
```

The abstract assembly translation of the above statement is empty, since the address of 'm' is the same as the address of 'a', and no calculations need to be done.

I define here two new atoms ZValueUnionArrow = 0 and ZValueUnionArrowNext = 0.

8. **access to struct, R-value taken, single-word member**: Assume the following type declarations:

```
struct {
  ...
 int m;
  ...
} * a;
int b;
```

(where 'm' and 'b' could be declared as any other single-word type in place of **int**) and I consider the following statement:

```
b = a->m;
```

The translation of the above statement is:

```
add    t, a, ⟨offset of m⟩    ; calculates the address of m
mvld   b, t                   ; transfers the contents
```

which calculates the address of 'm' by adding the address of 'a' and the offset of 'm' inside 'a' (known by the compiler); and copies the contents of the memory cell at that address into the register associated to variable 'b'. The cost of the translation is equal to 1 alul + 1 mvld. I define a new atom RValueStructArrow = 1 alul + 1 mvld, and set the cost of symbol ⟨*postfix_expression*⟩ to 1 RValueStructArrow.

9. **access to struct, R-value taken, multiple-word member**: Assume the following type declarations:

```
struct {
  ...
 type m;
  ...
} * a;
type b;
```

(where 'type' is any multiple-word type) and consider the following statement:

```
b = a->m;
```

The translation of the above statement is:

```
add   t, a, ⟨offset of m⟩   ; calculates the address of the first word m
mvld  b, t                  ; transfers the first word
add   t, t, 4               ; calculates the address of the second word
mvld  (b+1), t              ; transfers the second word
add   t, t, 4               ; calculates the address of the third word
mvld  (b+2), t              ; transfers the third word
 ...
add   t, t, 4               ; calculates the address of the last word
mvld  (b+w-1), t            ; transfers the last word
```

The translation is composed by an initial group of cost 1 alul + 1 mvld instructions (equal to 1 RValueStructArrow), plus a group composed by an add and a mvld instruction, of cost 1 alul + 1 mvld, repeated as many times as the number of words to transfer minus one. Please note that 'b+1', 'b+2', ... denote the names of the consecutive registers located after 'b', whose names are known at static time by the compiler, and do not involve any address calculation operation at runtime. For sake of consistency and to guarantee the possibility to generalize the formulae, I introduce a new atom, called RValueStructArrowNext, to capture the cost of the above repeated group, 1 alul + 1 mvld, even if its cost is identical to RValueStructArrow. I associate to symbol ⟨*postfix_expression*⟩ the cost of 1 RValueStructArrow $+W(r)-1$ RValueStructArrowNext, where r is attribute r of ⟨*postfix_expression*⟩.

10. **access to struct, L-value taken, single-word member**: Assume the following type declarations:

```
struct {
  ...
 int m;
  ...
} * a;
int b;
```

(where 'm' and 'b' could be declared as any other single-word type in place of **int**) and consider the following statement:

```
a->m = b;
```

The translation of the above statement is:

```
add   t, a, ⟨offset of m⟩   ; calculates the address of m
mvst  t, b                  ; transfers the contents
```

which calculates the address of 'm' by adding the address of 'a' and the offset of 'm' inside 'a' (known by the compiler); and copies the contents of the register associated to variable 'b' into the memory cell

at that address. The cost of the translation is equal to 1 alul + 1 mvst. I define a new atom LValueStructArrow = 1 alul + 1 mvst, and the cost of symbol ⟨*postfix_expression*⟩ to 1 LValueStructArrow.

11. **access to struct, L-value taken, multiple-word member**: Assume the following type declarations:

```
struct {
  ...
 type m;
  ...
} * a;
type b;
```

(where 'type' is any multiple-word type) and consider the following statement:

```
a->m = b;
```

The abstract assembly translation of the above statement is:

```
add   t, a, ⟨offset of m⟩   ; calculates the destination address for the first word
mvst  t, b                  ; transfers the first word
add   t, t, 4               ; calculates the destination address for the second word
mvst  t, (b+1)              ; transfers the second word
add   t, t, 4               ; calculates the destination address for the third word
mvst  t, (b+2)              ; transfers the third word
 ...
add   t, t, 4               ; calculates the destination address for the last word
mvst  t, (b+w-1)            ; transfers the last word
```

The cost of the translation is composed by an initial group of cost 1 alul + 1 mvst instruction (equal to 1 LValueStructArrow), plus a group composed by an add and a mvst instruction, of cost 1 alul + 1 mvst, repeated as many times as the number of words to transfer minus one. Please note that 'b+1', 'b+2', ... denote the names of the consecutive registers located after 'b', whose names are known at static time by the compiler, and do not involve any address calculation operation at run-time. I introduce a new atom, called LValueStructArrowNext, to capture the cost of the above repeated group, 1 alul + 1 mvst. I associate to symbol ⟨*postfix_expression*⟩ the cost of 1 LValueStructArrow $+W(r)-1$ RValueStructArrowNext, where r is attribute r of ⟨*postfix_expression*⟩.

12. **access to struct, RL-value taken, single-word member**: Assume the following type declarations:

```
struct {
  ...
 int m;
  ...
} * a;
int b;
```

(where 'm' and 'b' could be declared as any other single-word type in place of **int**) and I consider the following statement:

```
a->m += b;
```

(or any other statement obtained by replacing '+=' with a type-compliant compound assignment operator). The translation of the above statement is:

```
add   t, a, ⟨offset of m⟩   ; calculates the address of 'm'
mvld  c, t                  ; load its contents into a temporary register
add   c, c, b               ; perform the addition
mvst  t, c                  ; store the result in 'm' again
```

The cost of the translation is equal to 1 alul + 1 mvld + 1 mvst. I define a new atom RLValueStructArrow = 1 alul + 1 mvld + 1 mvst, and the cost of symbol ⟨*postfix_expression*⟩ to 1 RLValueStructArrow.

13. **access to struct, RL-value taken, multiple-word member**: Assume the following type declarations:

```
struct {
  ...
 type m;
  ...
} * a;
type b;
```

(where 'type' is any multiple-word type) and I consider the following statement:

```
a->m += b;
```

(or any other statement obtained by replacing '+=' with a type-compliant compound assignment operator). The translation of the above statement is:

```
add   t_0, a, ⟨offset of m⟩   ; calculates the address of the first word of 'm'
add   t_1, t_0, 4             ; calculates the address of the second word of 'm'
...                           ; ...
add   t_w-1, t_w-2, 4         ; calculates the address of the last word of 'm'

mvld  c,     t_0              ; loads the first word to a temporary register 'c'
mvld  c+1,   t_1              ; loads the second word to a temporary register 'c'
...                           ; ...
mvld  c+w-1, t_w-1            ; loads the last word to a temporary register 'c'

...                           ; appropriate translation for the operation between
                              ; value in registers c ... c+w-1 and b...?;
                              ; result left in registers d ... d+w-1

mvst  t_0, d                  ; transfers the first word of result into 'm'
mvst  t_1, (d+1)              ; transfers the second word of result into 'm'
...                           ; ...
mvst  t_w-1, (d+w-1)          ; transfers the last word of result into 'm'
```

The cost of the translation is composed by *w* add instructions, needed to calculate the addresses of the cells containing the encoded value of

a−>m, then w mvld instructions needed to load their value in a bank of consecutive registers, then w mvst instructions needed to write the result back to the registers whose addresses were calculated before.

Please note that c, c+1, c+2, ..., c+w−1 denote the names of the consecutive registers, whose names are known at static time by the compiler, and do not involve any address calculation operation at runtime.

For sake of consistency and generalizability, I introduce two new atoms, called RLValueStructArrow and RLValueStructArrowNext, whose costs are both equal to 1 alul + 1 mvld + 1 mvst. I associate to symbol ⟨*postfix_expression*⟩ the cost of 1 RLValueStructArrow $+W(r) - 1$ RLValueStructArrowNext, where r is attribute r of ⟨*postfix_expression*⟩.

14. **access to struct, Z-value taken, single- or multiple-word member**: I assume the following type declarations:

```
struct {
  ...
 type m;
  ...
} * a;
```

(where 'type' has no restrictions) and consider the following expression:

```
& a−>m
```

The translation of the above statement is

```
add   t, a, ⟨offset of m⟩   ; calculates the address of a−⟩m
```

Its cost is 1 add = 1 alul. I define here two new atoms ZValueStructArrow = 1 alul and ZValueUnionStructNext = 0.

Summary of the above cases: (generalization over w is complete)

$SU(t)$	v	Cost	
union	Z	0	+ 0
union	R	1 mvld	+ $(W(r) - 1)$ (mvld + alul)
union	L	1 mvst	+ $(W(r)w - 1)$ (mvst + alul)
union	RL	1 (mvld + mvst)	+ $(W(r) - 1)$ (alul + mvld + mvst)
struct	Z	1 alul	+ 0
struct	R	1 (alul + mvld)	+ $(W(r) - 1)$ (alul + mvld)
struct	L	1 (alul + mvst)	+ $(W(r) - 1)$ (alul + mvst)
struct	RL	1 (alul + mvld + mvst)	+ $(W(r) - 1)$ (alul + mvld + mvst)

Expressed in terms of atoms:

$SU(t)$	v	Cost	
union	Z	1 ZValueUnionArrow	$+(W(r)-1)$ ZValueUnionArrowNext
union	R	1 RValueUnionArrow	$+(W(r)-1)$ RValueUnionArrowNext
union	L	1 LValueUnionArrow	$+(W(r)-1)$ LValueUnionArrowNext
union	RL	1 RLValueUnionArrow	$+(W(r)-1)$ RLValueUnionArrowNext
struct	Z	1 ZValueStructArrow	$+(W(r)-1)$ ZValueStructArrowNext
struct	R	1 RValueStructArrow	$+(W(r)-1)$ RValueStructArrowNext
struct	L	1 LValueStructArrow	$+(W(r)-1)$ LValueStructArrowNext
struct	RL	1 RLValueStructArrow	$+(W(r)-1)$ RLValueStructArrowNext

The above formulae can be summarized as follows.

Syntax

⟨*postfix_expression*⟩ ::= ⟨*postfix_expression-1*⟩ '->' IDENTIFIER

Semantics:

$$.ci = 1\ .v\text{Value}SU(.t)\text{Arrow} + (W(.r)-1).v\text{Value}SU(.t)\text{ArrowNext}$$

For sake of brevity, I have not indicated which the symbol of the above attributes. For all of them, the omitted symbol name is ⟨*postfix_expression*⟩.

$SU(\cdot)$ is a utility function, defined as follows. Given a type stack t, $SU(t)$ is a function which yields the name "Struct" if $top(t)$ is a struct type-definition record, and "Union" if $top(t)$ is a union type-definition record.

4.4.7.12 The member access operator '.'

The dot operator allows to access members of a structure or of a union. It is defined by the following syntax rule:

⟨*postfix_expression*⟩ ::= ⟨*postfix_expression-1*⟩ '.' IDENTIFIER

I consider it as a binary operator because it requires two operands: the first (or left) operand type must be structure or union, and second (or right) operand type must be an identifier, among the ones declared in the symbol table associated with the structure or union. Like for the arrow operator '−>', the C standard considers the dot operator as a postfix unary operator, and the identifier is not considered as an operand, but as a part of the operator itself. Again, this disagreement is just a matter of taxonomy, and arbitrarily choosing one convention or the other is equivalent.

As I have already motivated in Section 4.4.2 (page 158), the dot operator exhibits anomalies which affect the valueness and the restricted type of their left descendants. I have already described completely the effects of those anomalies on the cost determination process of the affected nodes. Now, I determine what is the cost of the dot operator itself.

The dot operator, depending on the circumstances, has zero cost, or the cost of an offset calculation (i.e., an alul instruction).

Factors affecting cost:

- access to structure vs. access to union member: if a structure is referenced by address, accessing its member `m` requires calculating the address of `m` by summing the address of the structure and the displacement of `m`; no such address addition is required when accessing a member of a union, since all the members have zero offset;

- left operand is dereference or subscript operator: with these such expressions (e.g., '(∗pa).m' or 'a[i].m'), the accessed structure is indicated by address, and determining the position of the accessed member requires summing the position of the structure and the displacement of 'm' in it. This is not required with accesses to simple variables such as 'a.m', where the final position of 'm' is known at static time;

The cost of the '.' operator, when attributed, is the cost of an integer summation, denoted with atom DotOffset = 1 alul.

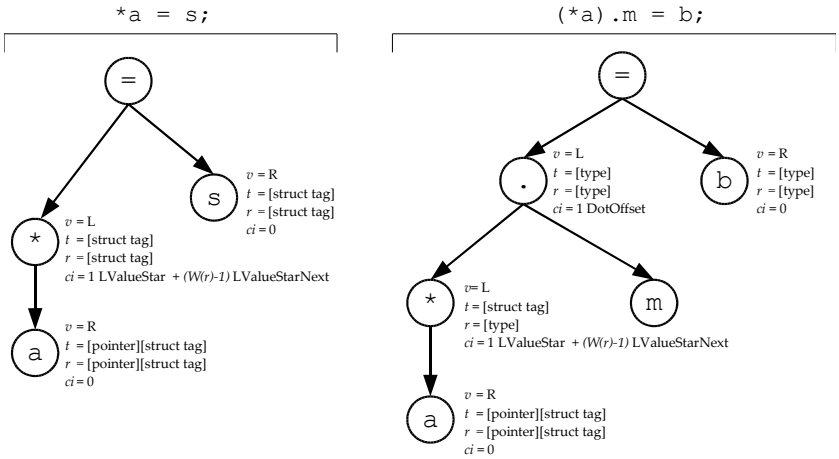


Figure 4.28: Determining *ci* for a '.' operator, when it is father of a '∗' operator (right). The operator has non-zero cost because its offset calculation instruction cannot be merged with the translation of any node. An expression without the '.' is reported for comparison (left).

Syntax:

⟨*postfix_expression*⟩ ::= ⟨*postfix_expression-1*⟩ '.' IDENTIFIER

Semantics:

- ⟨*postfix_expression*⟩.*ci* = = 1 DotOffset;
  if *top*(⟨*postfix_expression-1*⟩.*t*) = [struct] ∧
  (⟨*postfix_expression-1*⟩ ⇒ ⟨*postfix_expression*⟩ '[' ⟨*expression*⟩ ']' ∨
  ⟨*postfix_expression-1*⟩ ⇒ ⟨*primary_expression*⟩ ⇒ '(' ⟨*expression*⟩ ')' $\stackrel{*}{\Rightarrow}$ '(' '*' ⟨*cast_expression*⟩ ')'
)

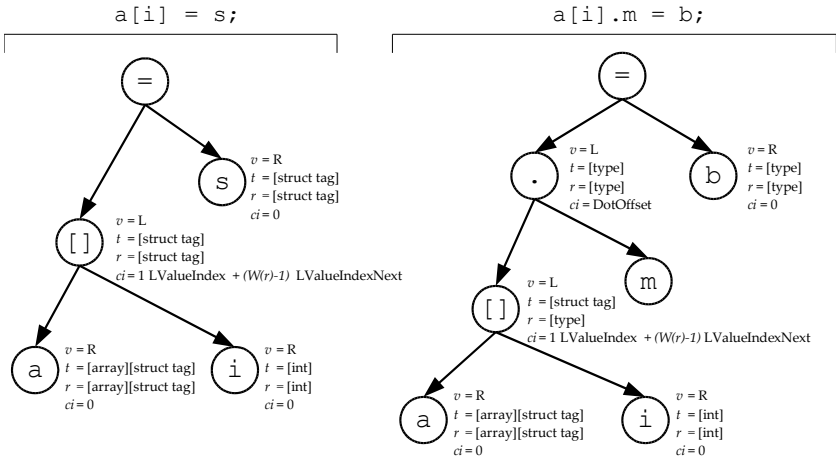


Figure 4.29: Determining *ci* for a '.' operator, when it is father of a '[]' operator (right). The operator has non-zero cost because its offset calculation instruction cannot be merged with the translation of any node. An expression without the '.' is reported for comparison (left).

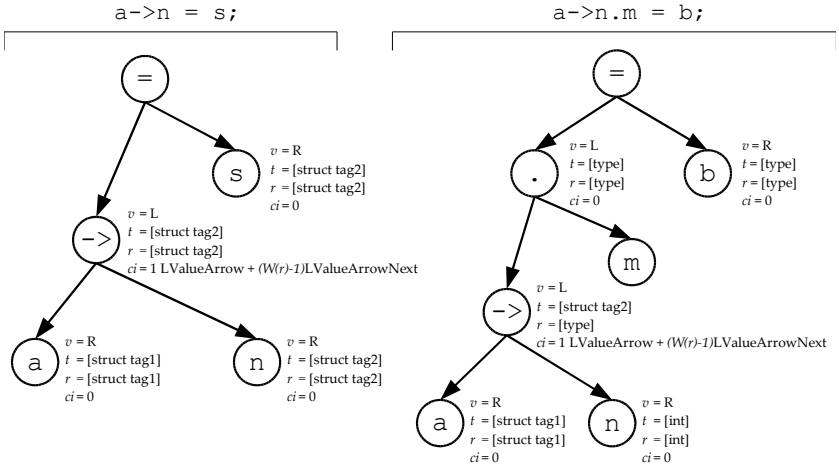


Figure 4.30: Determining *ci* for a '.' operator, when it is father of a '−>' operator (right). The '.' operator has no cost, because the offset calculation can be merged into the translation of '−>'. An expression without the '.' is reported for comparison (left).

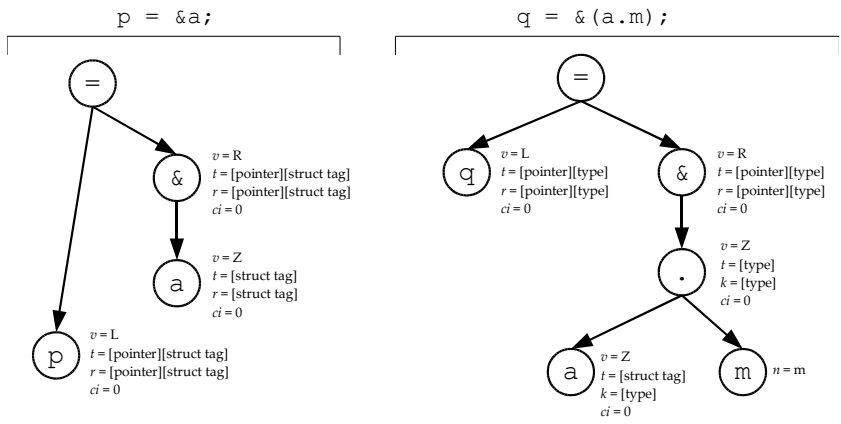


Figure 4.31: Determining *ci* for a '.' operator, when it is child of a '&' operator (right). The '.' operator has no cost, because the address of any member field of 'a' is known at compile time. An expression without the '.' is reported for comparison (left).

- $\langle postfix\_expression \rangle.ci = 0$;
  otherwise.

4.4.7.13 The function call operator

The cost of a function call includes the cost of the instruction which actually transfers execution to the called function, and the copy instructions to transfer the arguments and the return value.

Note that the creation of a virtual 'return' AST node, associated with a Return atom and supported by appropriate instrumentation is required for functions returning 'void' and lacking a final 'return;. Any further discussion on the topic is beyond the scope of this document. The readers interested to learn more on this are invited to refer to the source code of the project which implements this thesis.

Syntax:

$\langle postfix\_expression \rangle$ ::= $\langle postfix\_expression\text{-}1 \rangle$ '(' ')'
| $\langle postfix\_expression\text{-}1 \rangle$ '(' $\langle argument\_expression\_list \rangle$ ')'

Semantics:

$$\langle postfix\_expression \rangle.ci = \begin{array}{l} 1\,\text{FunctionCall} + \\ \text{Argument} \cdot \sum_{i=1}^{d} W(\langle parameter\_declaration \rangle_i.t) + \\ \text{Argument} \cdot \sum_{i=d+1}^{p} W(D(\langle assignment\_expression \rangle_i.t)) \end{array}$$

where:

- d is the number of declared parameters, not counting the ellipsis if present; zero if the function has no prototype;
- p is the number of the passed arguments;
- $\langle assignment\_expression \rangle_i$ is the i-th argument, present in the argument expression list of the function call expression; all the passed arguments are symbols $\langle assignment\_expression \rangle_1$ ... $\langle assignment\_expression \rangle_p$;
- $\langle parameter\_declaration \rangle_i$ is the i-th parameter as declared in the function declaration prototype; I assume that they also have an attribute t; all the parameters declared in the prototype are symbols $\langle parameter\_declaration \rangle_1$ ... $\langle parameter\_declaration \rangle_d$; if there is no prototype for the called function, then $d = 0$;

4.4.7.14 The simple assignment operator

Expression containing a simple assignment operator are governed by the following syntax rule:

$\langle assignment\_expression \rangle$ ::= $\langle unary\_expression \rangle$ '=' $\langle assignment\_expression\text{-}1 \rangle$

If node N is a simple assignment node in an AST, its inherent cost $N.ci$ depends on its left and right child nodes, and it can be either zero, or $W(N.r) \cdot$ Assignment.

In this section I will introduce and motivate the rule to distinguish the two cases. This rule is a more complete rendition of the consequences of translation rules already reported in Section 4.3.

In natural language, the cost of an assignment is zero when at least one of the operands are not register-bound, else the cost is as many 'Assignment' atoms as the word size of the restricted type.

I now motivate the above rule, then I give rules to determine the register-boundedness of the left or right child of an assignment expression.

First, I remind that in the construction of cost rules for expressions, I assumed that the cost of an expression is the cost of translation required to transfer the result to or from a bank of register of given name (according to the expression's valueness).

Therefore,

- in a case like the following one:

  ```
  *a = *b;
  ```

  where the assumptions on the type of variables 'a' and 'b' follow:

  ```
  int * a;
  int * b;
  ```

  the translations of subexpressions '*a' and '*b' are both register-unbound. In fact, the target register of the mvld instruction which

is the translation of '∗b' is free, and so is the the source register of the mvst instruction which is part of the translation of '∗a'. The translation of the entire assignment expression need just to rewrite the two translations, after binding all the occurrences of 'free' to the same, unique register. There is no need for new instructions. The inherent cost of the assignment is therefore zero;

- in a case where exactly one of the right or left child expressions of the assignment operator are register bound, such as

  ```
  a = ∗b;
  ```

  where the assumptions on the type of variables 'a' and 'b' follow:

  ```
  int   a;
  int ∗ b;
  ```

  again, the inherent cost of the assignment is zero. In fact, the translation of the assignment operator is just the composition of the translation of the children, where the 'free' occurrences in the translation of the right child have been bound to the register associated to 'a', without additional instructions and, therefore no additional cost. This is exactly the case illustrated in the translation rules and in the example of Section 4.3 (page 117).

  The same considerations apply when the roles are exchanged, and the register-unbound child is the left one. Although this case is not described in Section 4.3 for simplicity, the rationale is the same.

- in a case where both translations corresponding to the right and left children of the assignment operator are register-bound, such as the following:

  ```
  a = b;
  ```

  where the assumptions on the type of variables 'a' and 'b' follow:

  ```
  int   a;
  int   b;
  ```

  the translation of the assignment needs to actually include mov instructions to move data from the register corresponding to attribute R of the left child to R of the right child. Therefore, as many as $W(N.r)$ mov instructions are required. No displacement update instructions are required because all the registers involved in the data transfer are known by name. I define a new atom, named Assignment, whose cost is exactly one alul instruction (the class of the mov instruction).

The task of determining the register-boundedness for each of the right and left child nodes of the assignment node is guided by the rules already described in Section 4.4.5 (page 178).

Syntax:

⟨*assignment_expression*⟩ ::= ⟨*unary_expression*⟩ '=' ⟨*assignment_expression-1*⟩

Semantics:

$$\langle assignment\_expression \rangle.ci = \begin{cases} W(\langle assignment\_expression \rangle.r) & \text{Assignment} \\ & \text{if both operands are both register-bound} \\ 0 & \\ & \text{else} \end{cases}$$

4.4.7.15 The compound assignment operators

Syntax:

⟨*assignment_expression*⟩ ::= ⟨*unary_expression*⟩ '*=' ⟨*assignment_expression*⟩
| ⟨*unary_expression*⟩ '/=' ⟨*assignment_expression*⟩

⟨*assignment_expression*⟩ ::= ⟨*unary_expression*⟩ '+=' ⟨*assignment_expression*⟩
| ⟨*unary_expression*⟩ '−=' ⟨*assignment_expression*⟩

⟨*assignment_expression*⟩ ::= ⟨*unary_expression*⟩ '«=' ⟨*assignment_expression*⟩
| ⟨*unary_expression*⟩ '»=' ⟨*assignment_expression*⟩
| ⟨*unary_expression*⟩ '&=' ⟨*assignment_expression*⟩
| ⟨*unary_expression*⟩ '|=' ⟨*assignment_expression*⟩
| ⟨*unary_expression*⟩ '^=' ⟨*assignment_expression*⟩

Semantics:
the cost is the same as the arithmetic or bitwise operator without the '='.

4.4.7.16 Equality and relational operators

The equality and relational operator expressions are governed by the following syntax:

Syntax:

⟨*equality_expression*⟩ ::= ⟨*equality_expression-1*⟩ '==' ⟨*relational_expression-1*⟩
| ⟨*equality_expression-1*⟩ '!=' ⟨*relational_expression-1*⟩

⟨*relational_expression*⟩ ::= ⟨*relational_expression-1*⟩ '<' ⟨*shift_expression*⟩
| ⟨*relational_expression-1*⟩ '>' ⟨*shift_expression*⟩
| ⟨*relational_expression-1*⟩ '<=' ⟨*shift_expression*⟩
| ⟨*relational_expression-1*⟩ '>=' ⟨*shift_expression*⟩

which I generalized in the following rule:

⟨*re_expression*⟩ ::= ⟨*re_expression-1*⟩ ⟨*re_op*⟩ ⟨*re_expression-2*⟩

where ⟨*re_op*⟩ ::= '==' | '!=' | '>' | '<' | '>=' | '<=' ;
Semantics:

$$\begin{array}{lcl} t' & = & U(\langle \textit{re\_expression-1} \rangle.t, \langle \textit{re\_expression-2} \rangle.t) \\ \langle \textit{re\_expression} \rangle.ci & = & \begin{cases} \text{1 FloatCompare} & \text{if } t' \text{ is floating-point} \\ \text{1 IntCompare} & \text{if } t' \text{ is integral} \end{cases} \end{array}$$

The generalized rule is not correct from the point of view of operator precedence modeling, nevertheless I assume that parsing is done according to the real untouched grammar (as in Section 4.5.1 (page 231)), and the generalized rule is used only for cost determination and AST decoration purposes.

Legal operand types: arithmetic and pointer. Note that it is legal to assign structures but it is not to perform equality checks.

4.4.7.17 The 'return' statement

Syntax:
⟨*jump_statement*⟩ ::= 'return' ⟨*expression*⟩ ';'
Semantics:

$$\langle \textit{jump\_statement} \rangle.ci = W(CC(\langle \textit{expression} \rangle.t, t_0)) \text{ Argument}$$

where t_0 is the type of the current function containing the statement.

4.4.8 Attribute 'cc', conversion cost

Attribute *cc* expresses the cost of executing the type conversions associated with a given symbol. A type conversion is required to convert the value associated at run-time to a given expression, from the representation corresponding to the type of the expression, to the representation corresponding to the type needed in the larger expression where this value is used. To avoid every ambiguities in the cost attribution process, I define conversions as follows:

---

a **conversion** from type t_1 to type t_2 is the assembly code required to convert an encoded value, stored in a register or register bank (of appropriate size, according to type t_1) in such a way that at the end of that assembly code, the converted value is available in some register or register bank (of appropriate size and encoding, according to type t_2). All the registers involved in the operation are known by name or they are free.

Please note that my notion of conversion is not perfectly overlapping to the notion of conversion used in the C standard. More precisely, whenever there is a conversion according to my definition, there is also a conversion according to the C standard, but the *vice versa* is not true. For example, the so-called *function designator conversion* (as described in Section 4.4.1.14 (page 153)) is not a conversion in my sense, since it just means to take the address of a function as a result instead of taking the function itself (whose value is not defined); there are no values stored in registers which have their encoding changed.

In the C programming language, type conversions are possible either explicitly (via casting operations) or implicitly (e.g. during an arithmetic operation, an assignment or a function call). Whichever is the reason for the type conversion, the cost of the conversion is completely determined once the starting and destination type are known. It is therefore convenient to define, once and forever, a table which contains the cost of conversion for each possible couple of types. An example of this table could be Table 4.7. Be aware of the fact that the order in which the above rules are listed in the table is significant. For example, a conversion from 'float' to 'float' matches the first rule and has cost equal to zero, whereas a conversion from 'float' to 'double' matches the floating-point to floating-point rule, thus costing 1 FloatToFloat atom.

The above table is a simplification of the reality: it treats all integral types in the save way, even though two conversions involving different two different couples of integral types may have different costs, depending on the architecture. The same happens for floating point types.

Please note that, in my terminology, a conversion from type t to the same type t is valid, and it is empty. The cost of an empty conversion is trivially zero. Note that lookups in the table are done discarding type qualifiers

	Source type	Target type	Resulting cost
1	same as ⟨*type_name*⟩.*t*	whatever	0
2	pointer or array	pointer	0
3	integral (word-size)	pointer	0
4	integral	pointer	1 IntToInt
5	integral (different size)	integral	1 IntToInt
6	pointer	integral (non word-size)	1 IntToInt
7	integral	integral	0
8	floating-point	integral	1 FloatToInt
9	integral	floating-point	1 IntToFloat
10	floating-point	floating-point	1 FloatToFloat

Table 4.7: Summary of the rules which determine the cost of a type conversion. The order of the rules is meaningful: multiple rules could match a given case, the first matching one is applied.

(**const**, **volatile**). The **const** qualifier is used only for type checking at compile time, and for the **volatile** qualifier is used only to inhibit certain optimizations. They cannot influence type conversion.

As far as the calculation of attribute *cc* is concerned for each possible expression, the operators of the C language fall in a number of classes such that, in each class, the semantic rules to calculate attribute *cc* follow the same pattern. I summarize these classes in Table 4.8, and I discuss them individually in the next sections. The table presents an enumerated row for each class: each row indicates the arity of the operators which are part of that class, then a name for the class, the complete list of operators which belong to it, and a short description of how the operators behave as far as the result type is concerned, expressed in natural language.

Class	Arity	Informal description, members and behavior
1	1	The no-conversion unary operators '**sizeof**', '!', '&', '∗' prefix or postfix '++', prefix or postfix '−−' Behavior: the operand does not undergo any conversions.
2	1	The integral promotion unary operators '+', '−', '~' Behavior: the operand undergoes integral promotion.
3	2	The no-conversion binary operators ',', '[]', '.', '−>', '&&', '||' Behavior: the operands do not undergo any conversions.
4	2	The cast operator '(type)' Behavior: the right operand is converted to the type indicated by the left operand.
5	2	Integral promotion binary operators '<<', '>>' Behavior: the operands undergo integral promotion.
6	2	The usual arithmetic conversions operators '==', '!=', '<', '>', '<=', '>=' '+', '−', '∗', '/', '%', '&', '|', '^' Behavior: each of the operands is converted to the type determined according to the usual arithmetic conversions (as by C standard).
7	2	The simple assignment operator '=' Behavior: the right operand is converted to the type of the left operand.
8	2	The compound assignment operators '+=', '−=', '∗=', '/=', '%=', '&=', '|=', '^=', '<<=', '>>=' Behavior: the left and right operands undergo the usual arithmetic conversions, then the result is converted to the type of the left operand.
9	3	The conditional operator '?:' Behavior: the second and third operands are converted to the result type, determined according to the usual arithmetic conversion rules.
10	1..n	The function call operator Behavior: each argument is either converted to the type of the corresponding formal parameters, or by *default argument conversion* as by C standard.
11	1	The '**return**' statement Behavior: the operand is converted to the function declared return type.

Table 4.8: The operators of the C language (and the return statement) classified by conversion behavior.

4.4.8.1 The no-conversion unary operators

The operator expressions belonging to this class do not cause any conversions.

Syntax:

⟨*unary_expression*⟩ ::= '`sizeof`' ⟨*unary_expression-1*⟩
| '`sizeof`' '(' ⟨*type_name*⟩ ')'
| '`!`' ⟨*cast_expression*⟩
| '`&`' ⟨*cast_expression*⟩
| '`*`' ⟨*cast_expression*⟩
| '`++`' ⟨*unary_expression-1*⟩
| '`--`' ⟨*unary_expression-1*⟩

⟨*postfix_expression*⟩ ::= ⟨*postfix_expression-1*⟩ '`++`'
| ⟨*postfix_expression-1*⟩ '`--`'

Semantics:

$$\langle unary\_expression \rangle.cc = 0;$$

or

$$\langle postfix\_expression \rangle.cc = 0;$$

Some more detailed annotations follow. As far as the '!' operator is concerned, according to the standard, this operator returns an **int** value. Nevertheless, no conversion is done, ever. The operation may require a comparison which is type dependent. If the operand has floating-point type, the cost of floating point comparison is already accounted for in ⟨*unary_expression*⟩.*ci*.

As specified in Sections 6.3.2.4 and 6.3.3.1 of the C standard [81], prefix and postfix increment and decrement operators in this class return the same type as their operand, and operands must be scalar (i.e. integral or floating-point). Therefore no operand conversion is needed.

The same applies to referencing and dereferencing operators.

4.4.8.2 The integral promotion unary operators

Operators in this class perform an *integral promotion* on their operand. I have already presented integral promotion in Section 4.4.1.4 (page 146).

The cost of integer promotion, under my architectural assumptions is always null except for the integer promotion of [char] to [int], whose cost is one IntToInt atom.

Syntax:

⟨*unary_expression*⟩ ::= '+' ⟨*cast_expression*⟩
| '−' ⟨*cast_expression*⟩
| '~' ⟨*cast_expression*⟩

Semantics:

$$\langle unary\_expression \rangle.cc = CC(\langle cast\_expression \rangle.t, I(\langle cast\_expression \rangle.t))$$

Practically, under my assumptions, the cost of this conversion is always zero except when the type of ⟨*cast_expression*⟩ is a 'char' (signed or unsigned). In that case, the cost assumes the value of 1 IntToInt atom.

4.4.8.3 The no-conversion binary operators

Operators in this class do not cause any conversion.

Syntax:

⟨*expression*⟩ ::= ⟨*expression-1*⟩ ‘`,`’ ⟨*assignment_expression*⟩

⟨*postfix_expression*⟩ ::= ⟨*postfix_expression-1*⟩ ‘`[`’ ⟨*expression-1*⟩ ‘`]`’
| ⟨*postfix_expression-1*⟩ ‘`.`’ IDENTIFIER
| ⟨*postfix_expression-1*⟩ ‘`->`’ IDENTIFIER

⟨*logical_or_expression*⟩ ::= ⟨*logical_or_expression-1*⟩ ‘`||`’ ⟨*logical_and_expression-1*⟩

⟨*logical_and_expression*⟩ ::= ⟨*logical_and_expression-1*⟩ ‘`&&`’ ⟨*inclusive_or_expression-1*⟩

Semantics:

⟨*expression*⟩.*cc* = 0; or
⟨*postfix_expression*⟩.*cc* = 0; or
⟨*logical_or_expression*⟩.*cc* = 0; or
⟨*logical_and_expression*⟩.*cc* = 0;

The dot, arrow operators simply indicate the access to members of objects which are of type ‘struct’ or ‘union’. The accessed values are already in their respective locations, encoded according to their declared type. No conversion is needed.

4.4.8.4 The cast operator

The cast operator simply converts the right operand to the type indicated by the left operand. The conversion is explicit, and its cost is determined by function $CC(\cdot,\cdot)$.

Syntax:

⟨*cast_expression*⟩ ::= ‘`(`’ ⟨*type_name*⟩ ‘`)`’ ⟨*cast_expression-1*⟩

Semantics:

$$\langle \textit{cast\_expression} \rangle.cc = CC(\langle \textit{cast\_expression-1} \rangle.t, \langle \textit{type-name} \rangle.t)$$

4.4.8.5 The integral promotion binary operators

Only the bitwise shift operators belong to this class. These operators perform integral promotion on each of their two operands, individually.

Syntax:

⟨*shift_expression*⟩ ::= ⟨*shift_expression-1*⟩ ‘«’ ⟨*additive_expression*⟩
| ⟨*shift_expression-1*⟩ ‘»’ ⟨*additive_expression*⟩

Semantics:

$\langle shift\_expression \rangle.cc = CC(\langle shift\_expression\text{-}1 \rangle.t, I(\langle shift\_expression\text{-}1 \rangle.t)) +$
$CC(\langle additive\_expression \rangle.t, I(\langle additive\_expression \rangle.t));$

4.4.8.6 The usual arithmetic conversions operators

Operators in this class determine their result type according to the usual arithmetic conversions (see Section 4.4.1.11 (page 149)), then they convert each of their operands to the result type.

Syntax:

$\langle equality\_expression \rangle$ ::= $\langle equality\_expression \rangle$ '==' $\langle relational\_expression \rangle$
| $\langle equality\_expression \rangle$ '!=' $\langle relational\_expression \rangle$

$\langle relational\_expression \rangle$ ::= $\langle relational\_expression \rangle$ '<' $\langle shift\_expression \rangle$
| $\langle relational\_expression \rangle$ '>' $\langle shift\_expression \rangle$
| $\langle relational\_expression \rangle$ '<=' $\langle shift\_expression \rangle$
| $\langle relational\_expression \rangle$ '>=' $\langle shift\_expression \rangle$

$\langle additive\_expression \rangle$ ::= $\langle additive\_expression\text{-}1 \rangle$ '+' $\langle multiplicative\_expression \rangle$
| $\langle additive\_expression\text{-}1 \rangle$ '−' $\langle multiplicative\_expression \rangle$

$\langle multiplicative\_expression \rangle$ ::= $\langle multiplicative\_expression\text{-}1 \rangle$ '*' $\langle cast\_expression \rangle$
| $\langle multiplicative\_expression\text{-}1 \rangle$ '/' $\langle cast\_expression \rangle$
| $\langle multiplicative\_expression\text{-}1 \rangle$ '%' $\langle cast\_expression \rangle$

$\langle inclusive\_or\_expression \rangle$ ::= $\langle inclusive\_or\_expression\text{-}1 \rangle$ '|' $\langle exclusive\_or\_expression \rangle$

$\langle exclusive\_or\_expression \rangle$ ::= $\langle exclusive\_or\_expression\text{-}1 \rangle$ '^' $\langle and\_expression \rangle$

$\langle and\_expression \rangle$ ::= $\langle and\_expression\text{-}1 \rangle$ '&' $\langle equality\_expression \rangle$

Generalized syntax:

$\langle father\_expression \rangle$ ::= $\langle child\_expression\text{-}1 \rangle$ $\langle operator \rangle$ $\langle child\_expression\text{-}2 \rangle$

Semantics:

$\langle father\_expression \rangle.cc = CC(\langle child\_expression\text{-}1 \rangle.t, \langle father\_expression \rangle.t) +$
$CC(\langle child\_expression\text{-}2 \rangle.t, \langle father\_expression \rangle.t);$

Please remember from Section 4.4.1.11 (page 149) that $\langle father\_expression \rangle.t = U(\langle child\_expression\text{-}1 \rangle.t, \langle child\_expression\text{-}2 \rangle.t)$. Therefore, at least one of $CC(\langle child\_expression\text{-}1 \rangle.t, \langle father\_expression \rangle.t)$ or $CC(\langle child\_expression\text{-}2 \rangle.t, \langle father\_expression \rangle.t)$ is zero.

4.4.8.7 The simple assignment operator

In a simple assignment expression, the right operand is converted to the type of the left operand.

Syntax:

$\langle assignment\_expression \rangle$ ::= $\langle unary\_expression \rangle$ '=' $\langle assignment\_expression\text{-}1 \rangle$

Semantics:

$\langle assignment\_expression \rangle.cc = CC(\langle assignment\_expression\text{-}1 \rangle.t, \langle unary\_expression \rangle.t);$

4.4.8.8 The compound assignment operators

Compound assignment operators determine the type of their 'intermediate result' (which I call t') according to the usual arithmetic conversions as described in Section 4.4.1.11 (page 149)). Then they convert each operand to type t', they calculate the result and they store it back by converting it from type t' to the type of the left operand.

Syntax:

$$
\begin{array}{lcl}
\langle assignment\_expression \rangle & ::= & \langle unary\_expression \rangle \text{ '*=' } \langle assignment\_expression\text{-}1 \rangle \\
& | & \langle unary\_expression \rangle \text{ '/=' } \langle assignment\_expression\text{-}1 \rangle \\
& | & \langle unary\_expression \rangle \text{ '\%=' } \langle assignment\_expression\text{-}1 \rangle \\
& | & \langle unary\_expression \rangle \text{ '+=' } \langle assignment\_expression\text{-}1 \rangle \\
& | & \langle unary\_expression \rangle \text{ '-=' } \langle assignment\_expression\text{-}1 \rangle \\
& | & \langle unary\_expression \rangle \text{ '<<=' } \langle assignment\_expression\text{-}1 \rangle \\
& | & \langle unary\_expression \rangle \text{ '>>=' } \langle assignment\_expression\text{-}1 \rangle \\
& | & \langle unary\_expression \rangle \text{ '\&=' } \langle assignment\_expression\text{-}1 \rangle \\
& | & \langle unary\_expression \rangle \text{ '|=' } \langle assignment\_expression\text{-}1 \rangle \\
& | & \langle unary\_expression \rangle \text{ '\^{}=' } \langle assignment\_expression\text{-}1 \rangle
\end{array}
$$

Semantics:

$$
\begin{array}{lcl}
t' & = & U(\langle unary\_expression \rangle.t, \langle assignment\_expression\text{-}1 \rangle.t); \\
\langle assignment\_expression \rangle.cc & = & CC(\langle unary\_expression \rangle.t, t') + \\
& & CC(\langle assignment\_expression\text{-}1 \rangle.t, t') + \\
& & CC(t', \langle assignment\_expression \rangle.t);
\end{array}
$$

I emphasize that the semantics described above is the correct one. The reader is invited to abandon more naive beliefs, such as the one according which compound assignment operators cause the right operand to be converted to the type of the left operand, and then the operation to be performed directly between the left and the converted right operand, in the encoding of the left operand. This belief is incorrect. Please consider the following two examples:

```
int i = 12;
i /= 2.7;
```

```
int i = 12;
i /= (int)2.7;
```

At the end of the left fragment, the value of variable 'i' is 4. At the end of the right fragment, the value of variable 'i' is 6. If the above belief was true, both fragments should yield 6.

The cost rules in this section apply provided that the operation is admitted by the type constraints of the operators involved (e.g. the expression 'i <<= 2.0' is not allowed). Depending on the cases one or more of the above three $CC(...,...)$ contributions may be zero.

4.4.8.9 The conditional operator

The result type of a conditional operator is determined according to the usual arithmetic conversions, applied on the second and third operand. The conversions associated to the operator involve the conversion of the second and third argument to the result type.

Syntax:

⟨*conditional_expression*⟩ ::= ⟨*logical_or_expression*⟩ '?' ⟨*expression*⟩ ':' ⟨*conditional_expression-1*⟩

Semantics:

⟨*conditional_expression*⟩.*cc* =
CC(⟨*expression*⟩.*t*, ⟨*conditional_expression*⟩.*t*)+
CC(⟨*conditional_expression-1*⟩.*t*, ⟨*conditional_expression*⟩.*t*)

Please note that one or more of the above $CC(...)$ contributions are zero.

4.4.8.10 The function call operator

The conversions undergone by the actual arguments of a function call are not completely trivial. They are fully specified in Section 6.3.2.2 of the C standard [81]. I report the salient specifications in the following two statements:[1]:

- «If the expression that denotes the called function has a type that does not include a prototype, the integral promotions are performed on each argument, and arguments that have type '**float**' are promoted to '**double**'. These are called the *default argument promotions*.»
- «If the expression that denotes the called function has a type that includes a prototype, the arguments are implicitly converted, as if by assignment, to the types of the corresponding parameters. The ellipsis notation in a function prototype declarator causes argument type conversion to stop after the last declared parameter. The default argument promotions are performed on trailing arguments».

In synthesis, when a prototype is available for the called function, arguments are converted to the respective parameter types, as if by assignment, therefore following the same rules and costs already described in Section 4.4.8.7 (page 221). When a prototype is not available for the called function,

[1] Be aware of the following terminological issue. In the standard, the terms *parameter* and *argument* are not interchangeable, and have distinct meaning. Informally, parameter and arguments are respectively close to the concepts called *formal parameters* and *actual parameters*. Formally, a *parameter* is an element (nonterminal ⟨*parameter_declaration*⟩ in the grammar) of a parameter type list (nonterminal ⟨*parameter_type_list*⟩ in the grammar). «A parameter type list specifies the types of, and may declare identifiers for, the parameters of a function» (see Section 6.5.4.3 of the C standard [81]). Simply said, parameters are symbols which appear in function declarators, and include a type and possibly an identifier. On the other hand, an *argument* is an expression (as complex as desired, nonterminal ⟨*assignment_expression*⟩ in the grammar) which appears in an argument expression list (nonterminal ⟨*argument_expression_list*⟩ in the grammar).

the *default argument promotions* are used. The default argument promotions are used also when a prototype is available and it is a variable-argument prototype, for those arguments corresponding to the ellipsis '...'.

If t is a type, I denote with $D(t)$ the type resulting from the default argument promotion of type t, as prescribed by the C standard.

Syntax:

⟨*postfix_expression*⟩ ::= ⟨*postfix_expression-1*⟩ '(' ')'
| ⟨*postfix_expression-1*⟩ '(' ⟨*argument_expression_list*⟩ ')'

where:

⟨*argument_expression_list*⟩ ::= ⟨*argument_expression_list*⟩ ',' ⟨*assignment_expression*⟩

Semantics:

$$\langle \textit{postfix\_expression} \rangle.cc = \sum_{i=1}^{d} CC(\langle \textit{assignment\_expression} \rangle_i.t, \langle \textit{parameter\_declaration} \rangle_i.t) + \sum_{i=d+1}^{p} CC(\langle \textit{assignment\_expression} \rangle_i.t, D(\langle \textit{assignment\_expression} \rangle_i.t))$$

where:

- d is the number of declared parameters, not counting the ellipsis if present; zero if the function has no prototype;
- p is the number of the passed arguments;
- $\langle \textit{assignment\_expression} \rangle_i$ is the i-th argument, present in the argument expression list of the function call expression; all the passed arguments are symbols $\langle \textit{assignment\_expression} \rangle_1$ ... $\langle \textit{assignment\_expression} \rangle_p$;
- $\langle \textit{parameter\_declaration} \rangle_i$ is the i-th parameter as declared in the function declaration prototype; I assume that they also have an attribute t; all the parameters declared in the prototype are symbols $\langle \textit{parameter\_declaration} \rangle_1$ ... $\langle \textit{parameter\_declaration} \rangle_d$.

4.4.8.11 The 'return' statement

Syntax:

⟨*jump_statement*⟩ ::= 'return' ⟨*expression*⟩ ';'

Semantics:

$$\langle \textit{jump\_statement} \rangle.cc = CC(\langle \textit{expression} \rangle.t, t_0)$$

where t_0 is the type of the current function containing the statement.

4.4.9 Attribute 'cf', flow control cost

Attribute cf expresses the single-execution cost of determining whether to transfer the control flow to another point in the program, and possibly transferring it when needed. It is defined for statements. It is an inherited attribute; $N.cf$ depends in general on $N.f$, on the syntax of its father node.

4.4.9.1 Iteration statements

4.4.9.1.1 'while' statements

Syntax:

⟨*iteration_statement*⟩ ::= 'while' '(' ⟨*expression*⟩ ')' ⟨*statement*⟩

Semantics:

⟨*expression*⟩.*cf* = 0;
⟨*statement*⟩.*cf* = 1 While = 1 jump;

4.4.9.1.2 'do ... while (...)' statements

Syntax:

⟨*iteration_statement*⟩ ::= 'do' ⟨*statement*⟩ 'while' '(' ⟨*expression*⟩ ')' ';'

Semantics:

⟨*expression*⟩.*cf* = 0;
⟨*statement*⟩.*cf* = 0;

(Costs are zero. There are no jump instructions to associate. Compare with Section 4.3 (page 117).)

4.4.9.1.3 'for' statements

The only additional instruction introduced by the translation of a '**for**' statement is a an unconditional jump instruction 'j', appearing after the translation of ⟨*optional_expression-3*⟩ in the translation rule for '**for**' statements (Section 4.3 (page 117)).

Its cost is of class jump, and to encapsulate it, I define a new atom named 'For'. This atom must be assigned as an inherited cost to one of symbols ⟨*statement*⟩ or ⟨*optional_expression-3*⟩, since the 'j' instruction is guaranteed to execute exactly the same number of times as ⟨*statement*⟩.T and ⟨*optional_expression-3*⟩.T, since both translations are monolithic, and they are interested by exactly the same execution flows. The choice is completely arbitrary. I choose to attribute it to ⟨*statement*⟩.

Syntax:

⟨*iteration_statement*⟩ ::= 'for' '(' ⟨*optional_expression-1*⟩ ';' ⟨*optional_expression-2*⟩ ';' ⟨*optional_expression-3*⟩ ')' ⟨*statement*⟩

Semantics:

$$\begin{array}{lcl} \langle \textit{optional\_expression-1} \rangle.cf & = & 0 \\ \langle \textit{optional\_expression-2} \rangle.cf & = & 0 \\ \langle \textit{optional\_expression-3} \rangle.cf & = & 0 \\ \langle \textit{statement} \rangle.cf & = & 1 \text{ For} \end{array}$$

4.4.9.2 Selection statements

4.4.9.2.1 'if (...) ...' statements

Syntax:

⟨*selection_statement*⟩ ::= 'if' '(' ⟨*expression*⟩ ')' ⟨*statement*⟩

Semantics:

$$\begin{array}{lcl} \langle \textit{expression} \rangle.cf & = & 0 \\ \langle \textit{statement} \rangle.cf & = & 0 \end{array}$$

(Costs are zero. There are no jump instructions to associate. Compare with Section 4.3 (page 117).)

4.4.9.2.2 'if (...) ... else ...' statements

Syntax:

⟨*selection_statement*⟩ ::= `'if' '(' ⟨expression⟩ ')' ⟨statement-1⟩ 'else' ⟨statement-2⟩`

Semantics:

$$\begin{array}{lcl} \langle \textit{statement-1} \rangle.cf & = & \frac{1}{2} \text{ If} \\ \langle \textit{statement-2} \rangle.cf & = & \frac{1}{2} \text{ If} \end{array}$$

Note that it is impossible to determine in advance whether the flavor of the translation of ⟨*expression*⟩ will be TF or TT. Therefore, it is not possible to attribute in advance the cost of the additional jump ('j end') to one of the two branches. For this reason, this cost is attributes to both of them in half. Therefore, both statements receive a cost equal to half If, and 1 If = 1 jump.

4.4.9.2.2.0.7. *»Wovon man nicht sprechen kann, darüber muß man schweigen.«* [18]

4.4.9.2.3 'switch' statements

The most convenient and realistic way to account for the translation of a switch statement is to assume that the compiler translates it in the form of a jump table. This translation style is very common, and significantly easier to model than a chain of comparisons. I illustrate it with the following example:

```
switch (⟨expression⟩)
{
        case ⟨constant_expression-1⟩ :
             ⟨statement_list-1⟩

        case ⟨constant_expression-2⟩ :
             ⟨statement_list-2⟩


         ...
        default:
             ⟨statement_list-n⟩
}
```

The corresponding translation based on a jump-table is as follows:

```
{
              ⟨ expression ⟩.T
              shl    free, ⟨ expression ⟩.R, #2
              add    free, free, #jumptable
              j      free

jumptable:    DATA default  ;      cell 0
              DATA default  ;      cell 1
               ...
              DATA case−2   ;    cell ⟨constant_expression−2⟩.e
               ...
              DATA default  ;      other cells
               ...
              DATA case−1   ;    cell ⟨constant_expression−1⟩.e
               ...

case−1:       ⟨statement_list−1⟩.T
case−2:       ⟨statement_list−2⟩.T
    ...
default:      ⟨statement_list−n⟩.T

end:
}
```

Note that:

- the translation is correct with respect to '**break**' statements and fall-through cases. Break instructions are simply translated to a 'j end';
- the purpose of the 'shl' instruction is to multiply by four the result of the selection expression, thus obtaining the offset of the cell containing the case address in the jump table. This is required because in my architectural assumptions, addresses are 32 bit (i.e. 4 byte) long;
- 'DATA' is a directive to reserve a word in the program memory, whose value is initialized to the value of the label given as an argument; I employ the 'DATA' directives to create the jump table;
- note that jump addresses in the jump table are ordered by increasing value of the case key, while code in the 'case-n' sections is ordered in

the same way as in the original source code, to respect fall-through correctness;

- cell ⟨*constant_expression-2*⟩.*e* is reported before ⟨*constant_expression-1*⟩.*e* just to give the idea that cases in the jumptable are ordered depending on the numerical value of their respective constant expressions, which is not the same order as the cases appear in the source code.

Syntax:

⟨*selection_statement*⟩ ::= 'switch' '(' ⟨*expression*⟩ ')' ⟨*statement*⟩

Semantics:

$$\langle expression \rangle.cf = 1 \text{ Switch;}$$

A Switch atom is therefore 2 alul + 1 jump. More accurate models of the compiler, modeling more efficient techniques in a more accurate way are left to the implementation of this thesis.

4.4.9.3 Labeled statements

Syntax:

⟨*labeled_statement*⟩ ::= IDENTIFIER ':' ⟨*statement*⟩
| 'case' ⟨*constant_expression*⟩ ':' ⟨*statement*⟩
| 'default' ':' ⟨*statement*⟩

Semantics:

$$\langle statement \rangle.cf = 0;$$

No cost.

4.4.9.4 Jump statements

Syntax:

⟨*jump_statement*⟩ ::= 'goto' IDENTIFIER ';'
| 'continue' ';'
| 'break' ';'
| 'return' ';'
| 'return' ⟨*expression*⟩ ';'

Generalized syntax:

⟨*jump_statement*⟩ ::= ⟨*jump_instruction*⟩ ⟨*optional_identifier_or_expression*⟩ ';'

Semantics:

$$\langle jump\_instruction \rangle.cf = \begin{cases} 1 \text{ Goto} \\ 1 \text{ Continue} \\ 1 \text{ Break} \\ 1 \text{ Return} \\ 1 \text{ Return} \end{cases}$$

respectively, depending on which of the above syntax rules was recognized.

Trivial. All the above atoms have cost equal to 1 jump. I introduce separate atom names to distinguish the cause of the consumption.

4.4.10 Attribute 'c', total single-execution cost

The total cost of each node is the sum of the cost contributions presented in the previous sections: the inherent cost, the conversion cost and the flow control cost.

In other words, for each symbol $\langle symbol \rangle$:

Semantics:

$$\langle symbol \rangle.c = \langle symbol \rangle.ci + \langle symbol \rangle.cc + \langle symbol \rangle.cf$$

4.5 Grammar reference

This section contains the two portions of the specifications of the C language (according to the ANSI standard) describing respectively expressions and statements.

4.5.1 Expressions

⟨*expression*⟩ ::= ⟨*assignment_expression*⟩
| ⟨*expression*⟩ ‘`,`’ ⟨*assignment_expression*⟩

⟨*assignment_expression*⟩ ::= ⟨*conditional_expression*⟩
| ⟨*unary_expression*⟩ ‘`=`’ ⟨*assignment_expression*⟩
| ⟨*unary_expression*⟩ ‘`*=`’ ⟨*assignment_expression*⟩
| ⟨*unary_expression*⟩ ‘`/=`’ ⟨*assignment_expression*⟩
| ⟨*unary_expression*⟩ ‘`%=`’ ⟨*assignment_expression*⟩
| ⟨*unary_expression*⟩ ‘`+=`’ ⟨*assignment_expression*⟩
| ⟨*unary_expression*⟩ ‘`-=`’ ⟨*assignment_expression*⟩
| ⟨*unary_expression*⟩ ‘`«=`’ ⟨*assignment_expression*⟩
| ⟨*unary_expression*⟩ ‘`»=`’ ⟨*assignment_expression*⟩
| ⟨*unary_expression*⟩ ‘`&=`’ ⟨*assignment_expression*⟩
| ⟨*unary_expression*⟩ ‘`|=`’ ⟨*assignment_expression*⟩
| ⟨*unary_expression*⟩ ‘`^=`’ ⟨*assignment_expression*⟩

⟨*conditional_expression*⟩ ::= ⟨*logical_or_expression*⟩
| ⟨*logical_or_expression*⟩ ‘`?`’ ⟨*expression*⟩ ‘`:`’ ⟨*conditional_expression*⟩

⟨*logical_or_expression*⟩ ::= ⟨*logical_and_expression*⟩
| ⟨*logical_or_expression*⟩ ‘`||`’ ⟨*logical_and_expression*⟩

⟨*logical_and_expression*⟩ ::= ⟨*inclusive_or_expression*⟩
| ⟨*logical_and_expression*⟩ ‘`&&`’ ⟨*inclusive_or_expression*⟩

⟨*inclusive_or_expression*⟩ ::= ⟨*exclusive_or_expression*⟩
| ⟨*inclusive_or_expression*⟩ ‘`|`’ ⟨*exclusive_or_expression*⟩

⟨*exclusive_or_expression*⟩ ::= ⟨*and_expression*⟩
| ⟨*exclusive_or_expression*⟩ ‘`^`’ ⟨*and_expression*⟩

⟨*and_expression*⟩ ::= ⟨*equality_expression*⟩
| ⟨*and_expression*⟩ ‘`&`’ ⟨*equality_expression*⟩

⟨*equality_expression*⟩ ::= ⟨*relational_expression*⟩
| ⟨*equality_expression*⟩ ‘`==`’ ⟨*relational_expression*⟩
| ⟨*equality_expression*⟩ ‘`!=`’ ⟨*relational_expression*⟩

⟨*relational_expression*⟩ ::= ⟨*shift_expression*⟩
| ⟨*relational_expression*⟩ ‘`<`’ ⟨*shift_expression*⟩
| ⟨*relational_expression*⟩ ‘`>`’ ⟨*shift_expression*⟩
| ⟨*relational_expression*⟩ ‘`<=`’ ⟨*shift_expression*⟩
| ⟨*relational_expression*⟩ ‘`>=`’ ⟨*shift_expression*⟩

⟨*shift_expression*⟩ ::= ⟨*additive_expression*⟩
| ⟨*shift_expression*⟩ ‘`«`’ ⟨*additive_expression*⟩
| ⟨*shift_expression*⟩ ‘`»`’ ⟨*additive_expression*⟩

⟨*additive_expression*⟩ ::= ⟨*multiplicative_expression*⟩
| ⟨*additive_expression*⟩ '+' ⟨*multiplicative_expression*⟩
| ⟨*additive_expression*⟩ '−' ⟨*multiplicative_expression*⟩

⟨*multiplicative_expression*⟩ ::= ⟨*cast_expression*⟩
| ⟨*multiplicative_expression*⟩ '*' ⟨*cast_expression*⟩
| ⟨*multiplicative_expression*⟩ '/' ⟨*cast_expression*⟩
| ⟨*multiplicative_expression*⟩ '%' ⟨*cast_expression*⟩

⟨*cast_expression*⟩ ::= ⟨*unary_expression*⟩
| '(' ⟨*type_name*⟩ ')' ⟨*cast_expression*⟩

⟨*unary_expression*⟩ ::= ⟨*postfix_expression*⟩
| '++' ⟨*unary_expression*⟩
| '−−' ⟨*unary_expression*⟩
| '&' ⟨*cast_expression*⟩
| '*' ⟨*cast_expression*⟩
| '+' ⟨*cast_expression*⟩
| '−' ⟨*cast_expression*⟩
| '~' ⟨*cast_expression*⟩
| '!' ⟨*cast_expression*⟩
| 'sizeof' ⟨*unary_expression*⟩
| 'sizeof' '(' ⟨*type_name*⟩ ')'

⟨*postfix_expression*⟩ ::= ⟨*primary_expression*⟩
| ⟨*postfix_expression*⟩ '[' expression ']'
| ⟨*postfix_expression*⟩ '(' ')'
| ⟨*postfix_expression*⟩ '(' ⟨*argument_expression_list*⟩ ')'
| ⟨*postfix_expression*⟩ '.' IDENTIFIER
| ⟨*postfix_expression*⟩ '->' IDENTIFIER
| ⟨*postfix_expression*⟩ '++'
| ⟨*postfix_expression*⟩ '−−'

⟨*primary_expression*⟩ ::= IDENTIFIER
| CONSTANT
| STRING_LITERAL
| '(' ⟨*expression*⟩ ')'

4.5.2 Statements

⟨*statement*⟩ ::= ⟨*labeled_statement*⟩
| ⟨*compound_statement*⟩
| ⟨*expression_statement*⟩
| ⟨*selection_statement*⟩
| ⟨*iteration_statement*⟩
| ⟨*jump_statement*⟩

⟨*labeled_statement*⟩ ::= IDENTIFIER ':' ⟨*statement*⟩
| 'case' ⟨*constant_expression*⟩ ':' ⟨*statement*⟩
| 'default' ':' ⟨*statement*⟩

⟨*compound_statement*⟩ ::= '{' ⟨*optional_declaration_list*⟩ ⟨*optional_statement_list*⟩ '}'

⟨*optional_statement_list*⟩ ::=
| ⟨*statement_list*⟩

⟨*statement_list*⟩ ::= ⟨*statement*⟩
| ⟨*statement_list*⟩ ⟨*statement*⟩

⟨*expression_statement*⟩ ::= ';'
| ⟨*expression*⟩ ';'

⟨*selection_statement*⟩ ::= 'if' '(' ⟨*expression*⟩ ')' ⟨*statement*⟩
| 'if' '(' expression ')' ⟨*statement*⟩ 'else' ⟨*statement*⟩
| 'switch' '(' ⟨*expression*⟩ ')' ⟨*statement*⟩

⟨*iteration_statement*⟩ ::= 'while' '(' ⟨*expression*⟩ ')' ⟨*statement*⟩
| 'do' ⟨*statement*⟩ 'while' '(' ⟨*expression*⟩ ')' ';'
| 'for' '(' ⟨*optional_expression*⟩ ';' ⟨*optional_expression*⟩ ';' ⟨*optional_expression*⟩ ')' ⟨*statement*⟩

⟨*jump_statement*⟩ ::= 'goto' IDENTIFIER ';'
| 'continue' ';'
| 'break' ';'
| 'return' ';'
| 'return' ⟨*expression*⟩ ';'

⟨*translation_unit*⟩ ::= ⟨*external_declaration*⟩
| ⟨*translation_unit*⟩ ⟨*external_declaration*⟩

⟨*external_declaration*⟩ ::= ⟨*function_definition*⟩
| ⟨*declaration*⟩

⟨*function_definition*⟩ ::= ⟨*declaration_specifiers*⟩ ⟨*declarator*⟩ ⟨*declaration_list*⟩ ⟨*compound_statement*⟩
| ⟨*declaration_specifiers*⟩ ⟨*declarator*⟩ ⟨*compound_statement*⟩
| ⟨*declarator*⟩ ⟨*declaration_list*⟩ ⟨*compound_statement*⟩
| ⟨*declarator*⟩ ⟨*compound_statement*⟩

Chapter 5

Results, conclusions, developments

THIS chapter reports experimental results which show the viability and effectiveness of the estimation and optimization techniques presented in this thesis. Then, it draws the conclusion of the work, and it discusses some directions where the research could extend.

5.1 Results

5.1.1 Estimation

The major advantages of my estimation technique are the high level of information provided and the speed, at the expenses of a slightly reduced accuracy. As far as the high level is concerned, a quantitative comparison with other approaches is difficult to setup and bears little meaning. As far as speed and accuracy is concerned, I have compared the technique against instruction set simulation on a suite of industrial benchmarks including automotive, consumer, network, office, security and telecommunications applications.

Although the technique is general and completely independent from any architectural detail, I chose one specific simulator and, therefore, a specific architecture to model. As an instruction set simulator, I have employed SimIt-Arm [35], because it is currently the common reference point in the embedded design community. Actually, SimIt itself provides the latencies and the counts of the executed instructions, but not their energies, so I augmented SimIt with absorbed current data. The most convenient way to do so was to reuse the same platform information available from JouleTrack [42], therefore I refer to a platform hosting a StrongArm SA-1100 processor with a

clock running at 206 MHz and a supply voltage of 1.5 V. The memory hierarchy comprises separate instruction and data caches, each with associativity equal to 32 and sized 512 and 256 blocks respectively, where each block is formed by 32 bytes. Cache hits complete in 1 clock cycle; cache misses cause read and write penalties of 33 and 22 clock cycles respectively.

As a benchmark suite, I used MiBench [150], which is, in turn, the reference point for benchmarks in the embedded systems community. I ran 17 of the 24 benchmarks on both SimIt-Arm and in our tool flow. I was unable to include the remaining 7 benchmarks in the tests because they did not compile successfully under the `arm-linux-gcc` tool chain, which is required for SimIt-Arm.

		SimIt		My estimates		Estimation error	
#	Benchmark	E(mJ)	T(ms)	E(mJ)	T(ms)	ϵ_E(%)	ϵ_T(%)
1	adpcm-enc	37.1	191.6	40.4	281.9	9.3	47.6
2	adpcm-dec	35.3	171.3	29.1	148.2	-16.6	-13.3
3	bitcount	57.2	318.7	58.5	234.9	2.8	-26.1
4	blowfish-enc	75.4	483.2	74.1	517.9	-1.2	7.2
5	blowfish-dec	75.8	479.1	74.1	517.9	-1.2	8.1
6	crc32	91.0	637.0	72.4	511.0	-20.3	-19.8
7	dijkstra	32.4	176.6	41.4	198.9	27.9	12.6
8	fft	89.9	336.6	82.9	391.9	-7.8	16.5
9	ispell	10.1	80.5	11.7	89.2	16.8	10.9
10	jpeg	60.9	504.7	75.1	489.5	23.2	-3.0
11	qsort	120.9	687.4	83.5	466.4	-30.9	-32.1
12	rijndael-enc	42.0	245.0	40.9	242.3	-2.4	-1.1
13	rijndael-dec	41.2	238.8	39.6	237.6	-3.4	-0.2
14	sha	18.1	74.7	22.4	102.0	24.9	37.8
15	stringsearch	4.8	44.4	4.6	40.9	-3.5	-6.8
16	susan	40.5	180.1	41.7	191.4	4.4	6.3
17	tiff	3.1	16.6	1.8	9.6	-42.8	-41.9

Table 5.1: The comparison between energy and execution time estimates provided by SimIt-Arm and our tool shows that our methodology is quite accurate.

I report in Table 5.1 the results, with the full list of the benchmarks I employed, together with their consumed energy and execution time estimated respectively with SimIt-Arm and our methodology.

Figures show that there is close correspondence between reference and estimated data. The estimates exhibit good quality-of-result indicators: the coefficients of correlation between real and estimated data are 0.978 and 0.960 for the consumed energy and the execution time estimates respectively. Moreover, the average of the absolute value of the relative errors equals 14.1% for energy and 17.1% for execution time. Work is in progress to refine the implementation of our methodology, especially as far as statistical corrections are concerned, and even higher accuracy is expected when it will be complete.

As far as speed is concerned, statistics over the same benchmarks show, for our source-level flow, simulation times which are, on the average 10,350

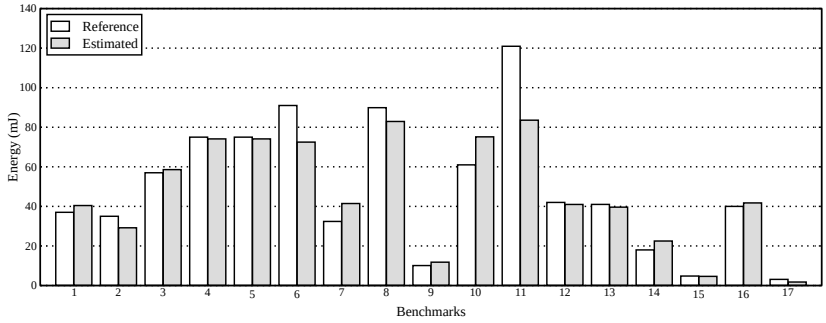


Figure 5.1: Comparison between reference and estimated energy in the experimental benchmarks.

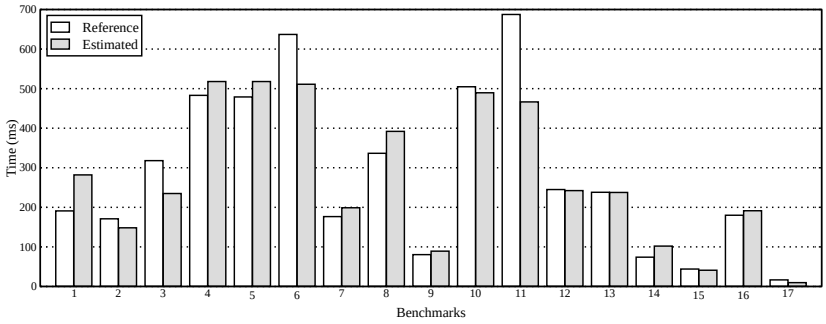


Figure 5.2: Comparison between reference and estimated execution time in the experimental benchmarks.

times shorter than SimIt-Arm. In short, 1 hour of instruction-set simulation can be replaced with 0.348 seconds of source-level simulation. This is good advance in the exploration possibilities made available to developers: with ISS tools, in one hour of simulation over a modern workstation, a developer can analyze a single run of a complex application (such as an MP3 encoder) over a limited number of samples. Since the same analysis requires less than half second with our source-level approach, in the same simulation time the developer could analyze the algorithm over sequences which are 10,000 times longer, or explore many different optimization alternatives of the algorithm over the same sequence.

5.1.2 Optimization

I have conducted preliminary experiments to show the viability and the effectiveness of the approach. In order to do so, I have submitted four different

benchmark applications to our flow. For each application, we obtained the list of proposed optimizations. Then, I applied those optimizations manually, verifying the effect after each of them.

The results are that: (1) the transformations suggested by the flow cause significant improvements in consumed energy and execution times, and (2) the optimization loop is very quick, requiring a few seconds per iteration. I do not consider here the time required for the application of the transformations. I chose not to perform automatic transformation application because it involves considerable implementation effort, and little scientific novelty. An automatic transformation technique was proposed in [127], and there are no conceptual difficulties to integrate that technique in our approach.

In the remainder of this section, I describe the benchmarks, the experimental setup, and the detailed results.

I have employed four different benchmarks: (1) Acfilter, (2) Hough Transform, (3) Dijkstra, (4) FFT. The first benchmark, 'Acfilter', is a psychoacoustic audio filter, composed by three stages: gammatone filter bank, equalization and phase-accurate recombination. It was provided to me by an industrial partner, who defined it as "already optimized". The second benchmark is a computer-vision application, which employs the Hough transform to detect lines in panoramic images captured from conical mirrors. This code was embedded on a moving robot. The third benchmark is Dijkstra's algorithm to find the shortest path in a graph, an algorithm frequently used in networking applications. It comes from MiBench [150]. The fourth benchmark is a usual FFT implementation, also from [150].

The experimental setup included SimIt-Arm [35], an instruction set simulator, used to verify *a posteriori* the effect of the transformations. Please note that an ISS is not part of our proposed flow: I used it here only for independent validation. I have employed SimIt-Arm rather than another ISS, because it is currently the reference ISS in the embedded community. The experimental configuration is the same as in the experiments described in the previous section.

I now report briefly the transformations suggested by our optimization flow, in order of decreasing score. For benchmark 1, it suggested 7 optimizations: a strength reduction, a function inlining, a function macro replacement, a common subexpression elimination, and 3 loop unrollings. For benchmark 2, the flow suggested 7 optimizations: 6 loop unrollings and a strength reduction. For benchmark 3, the flow suggested 5 optimizations: 2 loop unrollings, 1 malloc factoring, 1 loop unrolling, 1 fscanf factorization. For benchmark 4, the flow suggested: 1 function specialization, 1 loop unrolling, 2 standard library call factorization.

I applied all the above optimizations, one at a time, and ran the instruction-level simulators after each step. The results are reported in Table 5.2 and in the charts in Figure 5.3. The optimization flow achieves reductions between 7.8% and 22% in execution time, and between 5% and 22% in consumed energy.

The optimization flow proves to be effective, suggesting transformations that cause significant gains both in energy and execution time.

Benchmark (1): AcFilter

	Time (ms)	Time gain	Energy (mJ)	Energy gain
Original	331.18		390.70	
1: FI	329.05	0.64%	388.14	0.65%
2: LU	326.91	1.29%	385.63	1.30%
3: LU	318.69	3.77%	382.76	2.03%
4: LU	318.69	3.77%	382.76	2.03%
5: FM	298.71	9.80%	379.62	2.83%
6: FM	297.95	10.03%	371.51	4.91%
7: LU	285.86	13.68%	367.39	5.96%

Benchmark (2): Hough Transform

	Time (ms)	Time gain	Energy (mJ)	Energy gain
Original	1756.32		240.25	
1: LU	1602.46	8.76%	224.10	6.72%
2: LU	1544.46	12.06%	230.77	3.94%
3: LU	1446.64	17.63%	217.60	9.43%
4: LU	1402.14	20.17%	214.81	10.59%
5: SR	1401.86	20.18%	214.83	10.58%

Benchmark (3): Dijkstra

	Time (ms)	Time gain	Energy (mJ)	Energy gain
Original	237.08		41.458	
1: LU	236.61	0.20%	41.423	0.08%
2: LU	236.51	0.24%	41.412	0.11%
3: MF	227.00	4.25%	40.713	1.80%
4: LU	226.60	4.42%	40.639	1.98%
5: SF	218.39	7.88%	39.366	5.05%

Benchmark (4): FFT

	Time (ms)	Time gain	Energy (mJ)	Energy gain
Original	214.13		39.081	
1: FS	169.06	21.05%	30.874	21.00%
2: LU	167.68	21.69%	30.665	21.53%
3: SF	167.03	22.00%	30.575	21.77%
4: SF	166.43	22.27%	30.481	22.01%

Table 5.2: Results: which transformations are selected by our flow for each benchmark, and how much execution time and energy gains they cause. For a key to the acronyms (FS, LU, ...) see Section 3.4.1 (page 93).

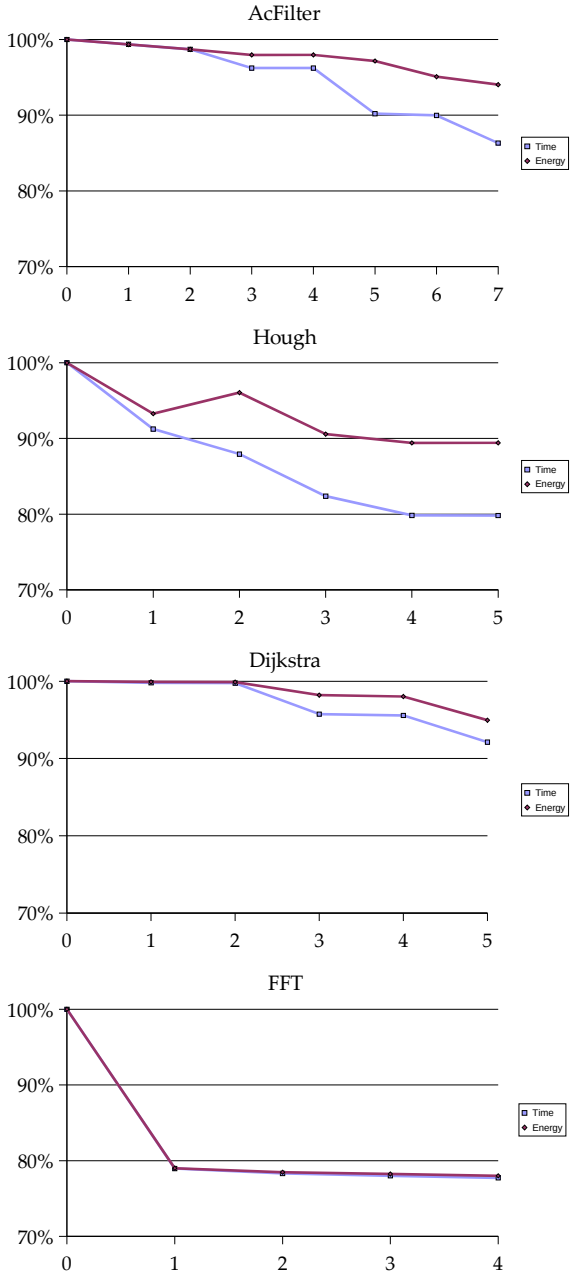


Figure 5.3: Results: how much execution time and energy are gained after applying each of the transformations proposed by our flow.

5.2 Conclusions

5.2.1 Estimation

This thesis describes a methodology to determine the cost of executing a C program, providing estimates for source-level entities. The method is formally founded and supported by a complete set of tools. It proves to be accurate and remarkably faster (10,000 times) than instruction-set simulation. Thanks to its optimized instrumentation strategy, instrumented programs run only 2.2 slower than original programs. Most important, our methodology offers to the programmer an unprecedented insight in the source-level causes of energy and time consumption, by allowing consumption estimation of individual syntax elements. Additionally, it externally operates as regular C compiler, therefore it can be applied transparently to most existing projects with no changes. Finally, it does not require the target onto which the code will be targeted but rather relies on a pre-characterization of that platform.

5.2.2 Optimization

I have also described a technique to select and target automatically the source code transformations to optimize a given program. The method is formally founded, and supported by a set of tools. It relies on an estimation engine which is accurate, and faster than instruction-set simulation. It improved execution time and energy consumption significantly over four industrial benchmarks. Thanks to this technique, it is possible to speed up the optimization process for embedded software, thanks to a short optimization loop, free from slow steps, such as instruction-set simulation.

5.3 Developments

5.3.1 Extending the methodology to C++

Given the important and increasing relevance of the C++ programming language in the embedded design community, the extensibility of the methodology proposed here to C++ is highly desirable. This section shows that this extension is possible, and it involves mainly a technical effort, while the conceptual difficulties are quite moderate.

The following paragraphs discuss the issues, and their associated required effort, involved in adapting the methodology to each of the new features that C++ introduces with respect to C (with reference to [84]):

- lexical scanning:
  the language adds a new style of comments '// ... ', 28 new keywords[1], and minor extensions in the notation of string literals. Updating the implementation of the flow to comply is trivial;

- parsing:
  the complexity of the syntax of the C++ programming language is significantly higher than ANSI C. Willink [85] publishes a grammar for standard C++ which comprises 560 syntax rules, whereas the C grammar used in this thesis reported by Degener [147] comprises 213. The very effort required to extend the syntax is significant, nevertheless it is very likely that the ready, open-source rules by Willink can be reused for the purpose. Very significant technical effort is required, on the other hand to extend the structure of the type system[2] and the symbol tables, which are required to perform scoping. A complete support for the C++ scoping rules is required for correct lexical tie-ins. Significant technical effort is required to extend the AST generation for the new rules. C++ presents some syntactical ambiguities which need to be resolved semantically. Non-negligible technical effort is required to implement the following parse-time disambiguation mechanisms:

  - declarations may now appear in all the places where a statement may appear; a disambiguation technique is required to distinguish expression statements from declarations; in cases, this requires examining the entire parse sub-tree candidate (Section 6.8 in [84]);
  - a similar disambiguation technique is needed to distinguish function-style casts from declarations (similar as above) inside declarators (see Section 8.1.1 in [84]);

[1] asm, catch, class, const_cast, delete, dynamic_cast, explicit, false, friend, inline, mutable, namespace, new, operator, private, protected, public, reinterpret_cast, static_cast, template, this, throw, true, try, typeid, typename, virtual, wchar_t.

[2] scoping rules which are consistent with namespaces and classes (potentially nested) require a symbol table structure and contents which are far more complex than the corresponding data structures for the C language.

  - access to base class members can also be ambiguous; although it can be assumed that the input code is correct and unambiguous, effort is needed to resolve which class member in the hierarchy is accessed.

- declarations and definitions:
  new distinction between what is considered a declaration and what a definition, with impact on its repeatability; new 'class'-type scope and scoping rules; namespaces. Significant technical effort involved, but no conceptual difficulties;

- standard conversions:
  no significant additions, except to pointer to members. Initialization, assignment, access, and comparison of pointer to members require some theoretical modeling;

- expressions:
  there are 5 new operators: 'new', 'delete', 'delete[]', and pointer-to-member operators '$->*$' and '$.*$'. Some technical effort is required to insure appropriate instrumentation of the new operators. Some theoretical modeling effort is required to model the inherent costs of the new operators in a consistent way. The instrumentation of the 'new' operator may require installing an instrumental new handler via set_new_handler. Due to 'virtual' methods and late binding, a method call could have a significantly different inherent cost than the current inherent cost of a function call. Some theoretical effort is involved in the preparation of a translation model and of an associated cost model which is consistent with the modeled compiler. Candidate implementation models are proposed in Section 10.7c to 10.10c in [84];

- declarations:
  C++ introduces 'inline' functions and methods. The inherent cost of an inlined function must be set to zero. A fundamental difficulty is in determining whether a function or a method marked as 'inline' was actually inlined or not. There are circumstances of different kinds which may prevent inlining appropriate metrics, compiler assistance or other hacks needed to determine whether the function was actually inlined. Section 7.1.2 in [84] discusses them thoroughly;

- references:
  the model requires a theoretical revision of non-negligible effort to model the cost of accessing, initializing, comparing and passing references;

- derived classes:
  inheritance (even multiple and virtual inheritance) does not cause any difficulties to the current instrumentation style chosen. The instrumentation relies on the ability (of the actual compiler and language runtime libraries) to correctly perform the determination of the called methods;

- member access control:
  no effort involved; programs are assumed correct; it is not the responsibility of our methodology to guarantee appropriate member access;

- overloading:
  significant effort required to determine the actual called function or operator in case of overloaded functions/operator. Note that this is just required to calculate cost of conversion for the arguments, not to calculate the costs of the code inside the called function. Evaluation of this cost is not an issue, since the current instrumentation technique already instruments functions internally. Some technical effort is required to determine whether each individual operator instance is overloaded or not. If it is not, the associated ci, cc and cf evaluation rules must be suspended[3]. Some technical effort is required to determine which one among the many possible overloaded operator definitions apply to the current case: this is required to determine if there is any argument conversion cost;

- templates:
  templates offer a way to provide a way to define a type-unbound set of classes or functions with a single definition. This is a potential cause of indeterminism. It introduces difficulties for appropriate cost analysis and instrumentation. More precisely: the control-flow cost of the statements and expressions inside a template function or a method of a template class are independent from the template parameters. Inherent and conversion costs present some difficulties. In general, for each operator instance, it is not easy to determine in advance whether an overloaded or a standard implementation was called. In cases, it requires runtime support. Technical effort is required to perform this cost disambiguation. Thanks to complete instrumentation of the overloading definitions, it is always possible to determine what was called, although it is not clear how to do that with the least run-time overhead. It is likely that RTTI may help for the purpose;

- exception handling:
  the current instrumentation technique allows to determine whether an exception was thrown and caught. Some theoretical effort is required to model the exception handling process, also considering the inherent and conversion costs associated to creating and passing the exception object.

As a conclusion, I estimate the effort required to update the current implementation of the methodology in such a way that it accepts C++ as an input language. In the light of the above considerations, my estimate for this effort is less than one man-year.

[3] imagine the consequences of applying the current cost evaluation mechanism to an overloaded instance of '<<', thus attributing a bitwise operation cost to an operation which is significantly more complex.

5.3.2 Modeling more complex hardware

The original version of the methodology presented here is designed to estimate the energy consumed by the core of a single-issue processor, with single-instruction single-data instructions. During the preparation of this document, the author has been working on an extension of this technique, aimed at estimating the consumption of the datapath, scratchpad data memory and instruction memory hierarchy of a single-core or multi-core architecture, with data-level parallel instructions.

The extended version of the tools allows the definition of intrinsics, for which the programmer provides:

- a C implementation, which is used only during source-level simulation to ensure functionally-correct behavior;
- a cost definition in terms of abstract instructions; an intrinsic defined in this way is, at the same time, a new atom, for which the programmer has to provide an abstract cost.

Preliminary work is also ongoing on source-level estimation for VLIW architectures.

The design and implementation of these extended techniques and tools will be discussed by the author in other publications.

Appendix A

The cost of floating-point emulated operations

THE source-level estimation methodology proposed in this thesis relies on abstract translation and execution models. Unfortunately, these models do not provide deterministic estimates for the execution of floating-point arithmetic operations on platforms which do not include a floating point unit.

Therefore, appropriate models are required which account for the cost of executing emulated floating-point operations. In this appendix, I determine statistical models specifically designed for this purpose.

I show how to obtain estimates for the emulation routines included in soft-float (the emulation library included in the popular GCC compiler) on a popular architecture (ARM). To do that, I have performed experiments on the reference instruction-set simulator for that architecture (SimIt-ARM).

A.1 Motivation

Source-level estimation relies on an abstract compilation model from source code to an abstract instruction set, and on a statistical characterization of the execution time and energy consumption of the instruction set. The translations of purely arithmetical expressions generated by that translation models are short, simple basic blocks. Therefore, only one execution flow is possible in such translations, and this makes the determination of their execution cost trivial and independent from the operands.

Unfortunately, when a compiler generates the translation of a floating-point arithmetical expression for architectures which do not include a floating-point unit (FPU), the result is not a short, simple basic block. Instead, the compiler maps floating-point operations to calls to routines in a

floating-point emulation library. The cost of executing these routines cannot be evaluated deterministically at static time. In fact, these routines loops and conditions, therefore multiple execution flows are possible, each with different costs.

The above limitation is not negligible, because a large number of processors designed for use within embedded systems do not feature an FPU, for example the ARM7TDMI processor. To overcome this limitation, I provide statistically-accurate estimates for the cost of executing floating point operations.

I have determined these estimates by preparing an appropriate experimental setup, including an instruction-set simulator, a complete C compilation and build toolchain for that target, and a set of script for the automated test set generation and evaluation.

I will show that the actual values assumed by operands affect significantly the cost of operations. For example, summing two numbers which are similar in magnitude costs more than summing two numbers which are far away from each other. In fact, in the first case the mantissa of the result must be really calculated, while in the second case the result is just one of the two operands.

With the approach I show, a developer can also tune his own statistical models for emulated FP operations according to the actual statistical distributions of the operand values that his application will process.

A.2 Experimental setup

For the experiments, I chose the ARM7TDMI microprocessor, which is a general purpose micro-processor very commonly used in the design of embedded systems, and which does not contain a floating-point unit. I chose to employ SimIt-Arm as an instruction-set simulator for ARM because it is the reference point in the embedded design community. SimIt-Arm comes with a build toolchain for the C/C++ programming languages based on GNU GCC version 2.95, which includes the *soft-float* library for floating point emulated operations.

The entire list of the emulation functions which is included in *soft-float* for the ARM architecture is reported in Table A.1, together with the C operator they map (e.g. arithmetic operators such as '+', '−', ... or relational operators such as '>=', '<', ...) and the floating-point data type which they accept as arguments (one of **float**, **double** or **long double**). The complete list of prototypes for the above functions is also reported, in Figure A.2. This list is important in the construction of any benchmark for reasons that I will explain later.

The information reported here is compliant with the reference information for the chosen compiler, as reported in Section 4.2 "Routines for floating point emulation" of the "GCC Internals" manual [86].

Data types **float**, **double** and **long double** are composed as indicated in Table A.2, according to the respective IEEE standard.

Operator	Data type float	double	long double
+	__addsf3	__adddf3	__addtf3
−	__subsf3	__subdf3	__subtf3
*	__mulsf3	__muldf3	__multf3
/	__divsf3	__divdf3	__divtf3
==	__eqsf2	__eqdf2	__eqtf2
!=	__nesf2	__nedf2	__netf2
>=	__gesf2	__gedf2	__getf2
<	__ltsf2	__ltdf2	__lttf2
<=	__lesf2	__ledf2	__letf2
>	__gtsf2	__gtdf2	__gttf2

Table A.1: Map of the emulation functions used in the GCC compiler [86].

data type	Standard	Size	sign	mantissa	exponent	unused
float	IEEE 754	32 bits	1	23	8	0
double	IEEE 754	64 bits	1	52	11	0
long double	IEEE 854	96 bits	1	64	15	16

Table A.2: Bit composition of the floating-point data types.

A.3 Benchmark construction

The task of constructing accurate benchmarks to measure the above functions is a tricky operation, which demands special attention in order to filter out (or, when not possible, at least minimize) a number of parasitic effects which perturb the measurements. In the following paragraphs, I describe and motivate the design choices I made for the purpose.

- I have written the benchmarks in C rather than in assembly language for portability. This way, the same set of benchmarks can be easily reused with no modifications for measurements on any other target for which the GCC compiler is available, and with minor modifications on those targets which have no GCC but some other C compiler available;
- I have chosen to I directly call the soft-float routines, in order to have strict control over the assembly translation of the benchmarks. Therefore, I place invocations to them in source codes (__addsf3,__subsf3, ...) rather than the corresponding C operators (+,−, ...). This allows complete control to which routine is actually invoked, making sure that no unexpected type conversions take place. This choice has the additional beneficial effect of avoiding the effects compiler optimizations (e.g. constant subexpression elimination) which may happen in arithmetic expressions but not across function call boundaries;
- I am interested in determining the entire statistical distribution of exe-

```
float          __addsf3     (float a, float b);
double         __adddf3     (double a, double b);
long double    __addtf3     (long double a, long double b);

float          __subsf3     (float a, float b);
double         __subdf3     (double a, double b);
long double    __subtf3     (long double a, long double b);

float          __mulsf3     (float a, float b);
double         __muldf3     (double a, double b);
long double    __multf3     (long double a, long double b);

float          __divsf3     (float a, float b);
double         __divdf3     (double a, double b);
long double    __divtf3     (long double a, long double b);

int            __eqsf2      (float a, float b);
int            __eqdf2      (double a, double b);
int            __eqtf2      (long double a, long double b);

int            __nesf2      (float a, float b);
int            __nedf2      (double a, double b);
int            __netf2      (long double a, long double b);

int            __gesf2      (float a, float b);
int            __gedf2      (double a, double b);
int            __getf2      (long double a, long double b);

int            __ltsf2      (float a, float b);
int            __ltdf2      (double a, double b);
int            __lttf2      (long double a, long double b);

int            __lesf2      (float a, float b);
int            __ledf2      (double a, double b);
int            __letf2      (long double a, long double b);

int            __gtsf2      (float a, float b);
int            __gtdf2      (double a, double b);
int            __gttf2      (long double a, long double b);
```

Figure A.1: Prototypes of the emulation functions belonging to the *soft-float* library.

cution times of each emulation routines. I derive these distributions by performing tests over samples. These samples come themselves from some populations with given statistical properties, and these properties influence the outcome of the measurement. In order for the measurements not to be biased, the choice of these properties must be conscious and consistent with reality. It is perfectly sound to assume that in a given user application variables assume values in some ranges more frequently than the rest of representable values, nevertheless there are no simple, application-independent motivations to assume

that the entire population of variables and application privileges a given subset of the floating-point representable field. Therefore, in the experiments I select operands by extracting independent, identically-distributed random samples from a population with uniform distributions, covering exactly all the representable range for each of the floating-point data types. The choice to determine complete distributions rather than average values has impacts on the construction of experiments, especially since the instruction-set simulator does not allow to extract statistics on individual portions of a benchmark. For example, a benchmark consisting of a loop including the generation of random operands and a call to an emulation routine is not acceptable, since the benchmark execution time gives information on the average execution time of the single call, but not its statistical distribution;

- in general, the cost of executing a benchmark as just explained is much larger than the cost of executing the single emulated operation I am interested in. It includes a large overhead due to program start, command-line argument parsing and conversion. A way to remove this overhead is needed. I do this by comparison against a *dummy* benchmark. A dummy benchmark is a program designed to have the same translation and execution flow of the original benchmark, except for the calls of the routines under estimation, which are excluded from the dummy. Then, I simulate execution of the benchmark and of the corresponding dummy in the same conditions and with the same arguments. Eventually, I subtract the cost of executing the dummy from the cost of executing the benchmark. The design of accurate dummies in C requires care to avoid undesired compiler optimizations. For example, if the result of the operation under measurement is not used, the compiler could cancel the entire operation. Benchmarks and dummies must be carefully designed together, with an eye on their data-flow structure. The benchmark designer must ensure that the compiler applies the same optimizations on the benchmark and on the dummy, with special attention to common subexpression elimination, dead code elimination, constant folding and arithmetic optimizations. A couple of examples of common techniques which I have employed to achieve these results are:

  - literal constants must be replaced with calls to conversion functions, for example the definition:

    ```
    float f = 2.5;
    ```

    must be replaced with:

    ```
    float f = atof("2.5");
    ```

    While in the first case the compiler is able to determine the value of variable 'f' for all the subsequent operations which use it, and even to suppress dead code depending on conditional expressions such as: `if (!f) { ... }`, this cannot happen anymore in the

second case, where the compiler must actually generate the code for the call to 'atof()', and it cannot predict its outcome;

- to ensure that an operation (not including a function call) is actually performed, its result must be used later. If the result is not used, the compiler may eliminate the entire operation. A common way to do that is to sum intermediate results and return the sum as the 'main' function result value. The benchmark source code and associated dummy must be designed in such a way that this 'use' code is present in both in the same form.

- in the design of benchmarks, especially when long sequences of operations are present, or the use of arrays are involved, the effects on caches must be taken into account. The loop bodies should be long enough to make the effect of the jump at the end of the body negligible, but short enough to fit entirely in the instruction cache. After the first loop iteration, no more instruction cache misses should occur. The same should apply for data.

- the benchmark designer must be aware of the technicalities of the C language which could affect the measurements, and which are often unknown to programmers: one of these is how calls are generated by ANSI C compilers to functions whose prototypes are not given. Since I am measuring routines which are not designed to be called by the user, their prototype is not available in standard header files. It is the responsibility of the benchmark designer to declare these routines, as in Figure A.2. Without them, the C compiler creates prototypes with the default argument data types (i.e., 'double' when floating-point argument are provided). This makes it impossible to perform correct argument passing and perturbs the results by adding unrequested argument conversion overhead.

A.4 Arithmetic operations

This section describes an experiment to determine the statistical distribution of the latencies of the four arithmetic operations ($+$, $-$, $*$,/) when applied on floating-point data. To comply with all the requirements for parasitic effect removal described in the previous section, I have designed a script and a template for the generation of benchmark codes.

I prepared multiple executable versions of the benchmarks by compiling the template in Figure A.2 with different values for TYPE and FUNCTION. TYPE assumes all the possible data types (float, double, long double) and FUNCTION is replaced with all the emulation functions (__addsf3, __subsf3, ...) or is left undefined, for the dummy benchmark. The multiple compilations are performed by the script in Figure A.3.

At the end of benchmark generation, the following executable benchmarks are available:

```
#include "../soft-float-prototypes.h"
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char * argv[])
{
  TYPE f1, f2, f3;

  f1 = atof(argv[1]);
  f2 = atof(argv[2]);
  f3 = atof("12345.6789");

#ifdef FUNCTION
  f3 = FUNCTION(f1,f2);
  f3 = FUNCTION(f1,f2);
  f3 = FUNCTION(f1,f2);
                        /* ...
                           total 100 calls to the desired function
                           ...
                         */
  f3 = FUNCTION(f1,f2);
#endif

  return (int)(f1 + f2 + f3);
}
```

Figure A.2: Parametric source code template for the generation of arithmetic operation benchmarks.

test-float-add	test-double-add	test-long-double-add
test-float-sub	test-double-sub	test-long-double-sub
test-float-mul	test-double-mul	test-long-double-mul
test-float-div	test-double-div	test-long-double-div
test-float-dummy	test-double-dummy	test-long-double-dummy

Another script, which I do not report here for sake of brevity, generates 1000 random numbers for each data type (3000 tests in total), extracted from a uniform distribution over the representable field of each data type, and runs each benchmark with these numbers as arguments. In each benchmark, the operation is repeated 100 times in order to minimize border effects.

The results of the benchmark are reported in the following tables.

```
#!/usr/bin/tclsh

set precisions      {s d t}
set precname(s)     "float"
set precname(d)     "double"
set precname(t)     "long double"

set operators       { + - * / }
set opname(+)       add
set opname(-)       sub
set opname(*)       mul
set opname(/)       div

foreach precision $precisions {
  set type      $precname($precision)
  foreach operator $operators {
      set function      __$opname($operator)${precision}f3
      set file          test-[string map {" " "_"} $type]-$opname($operator)

      exec arm-linux-gcc -static -DTYPE=$type -DFUNCTION=$function prototype.c -o $file
      exec arm-linux-gcc -S -DTYPE=$type -DFUNCTION=$function prototype.c -o $file.s
  }
  set   file      test-[string map {" " "_"} $type]-dummy
  exec arm-linux-gcc -static -DTYPE=$type prototype.c -o $file
  exec arm-linux-gcc -S -DTYPE=$type prototype.c -o $file.s
}
```

Figure A.3: The Tcl script used to generate benchmark source files for the arithmetic operations benchmarks.

	Mean value		
	Float	Double	Long Double
Addition	165.53	259.50	546.07
Subtraction	175.70	279.11	624.86
Multiplication	164.00	404.08	1095.00
Division	673.30	1396.10	3538.90

Table A.3: Average measured cost of floating-point operators on operands extracted from uniformly-distributed populations, expressed in clock cycles.

	Standard deviation		
	Float	Double	Long Double
Addition	18.54	20.98	13.54
Subtraction	18.35	20.80	29.20
Multiplication	3.80	3.14	24.69
Division	13.25	26.23	153.48

Table A.4: Standard deviation of the measured cost of floating-point arithmetic operators on operands extracted from uniformly-distributed populations, expressed in clock cycles.

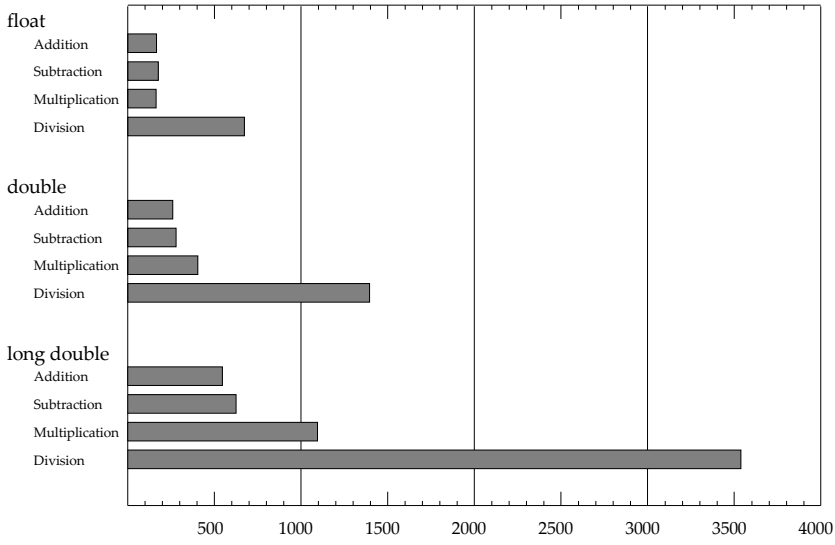
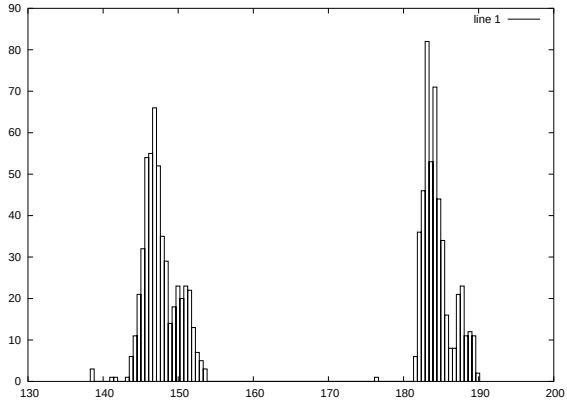
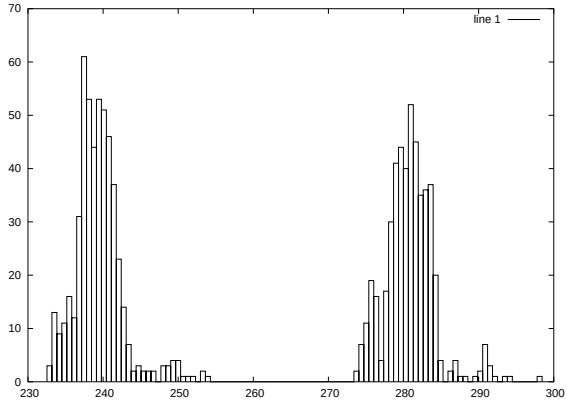


Figure A.4: Average cost of arithmetic operators between operands of the various floating-point types.

float



double



long double

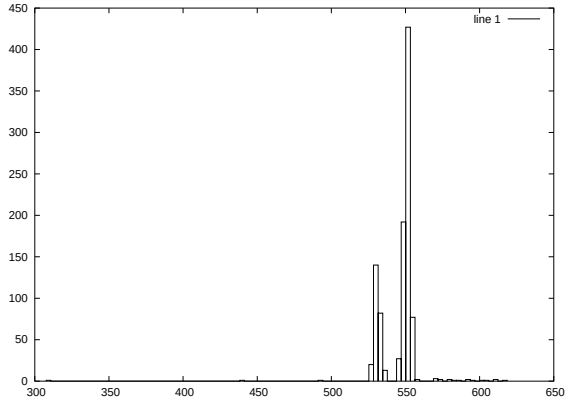
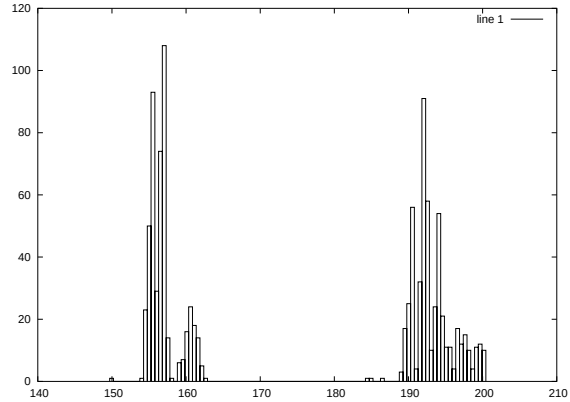
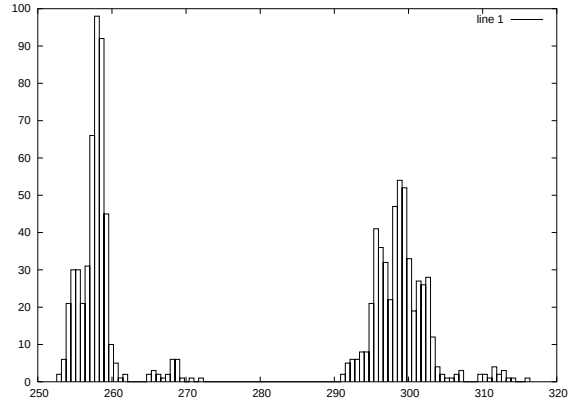


Figure A.5: Statistical distribution of the latency of emulation routine for operator '+'. Horizontal axis is latency in cycles, vertical is frequency over 1000 cases.

float



double



long double

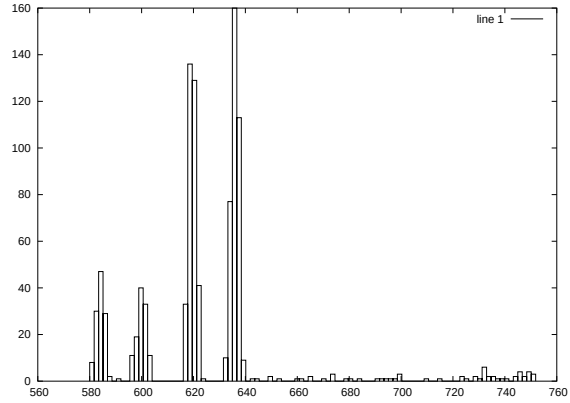
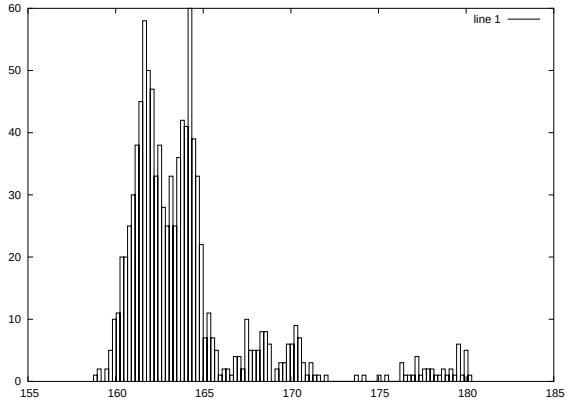
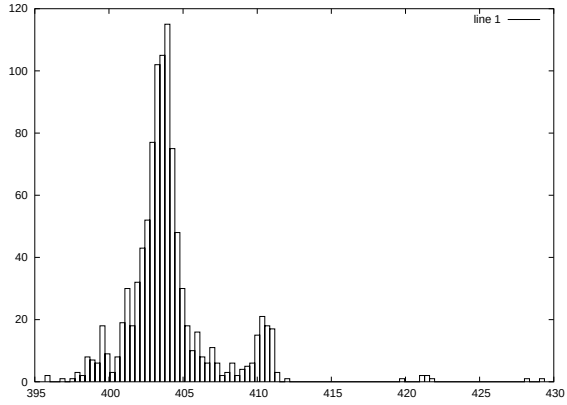


Figure A.6: Statistical distribution of the latency of emulation routine for operator '−'. Horizontal axis is latency in cycles, vertical is frequency over 1000 cases.

float



double



long double

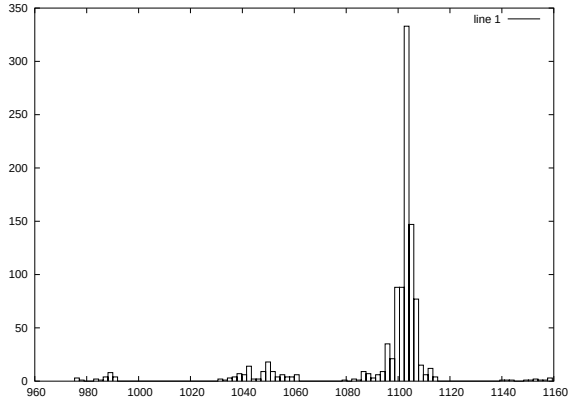
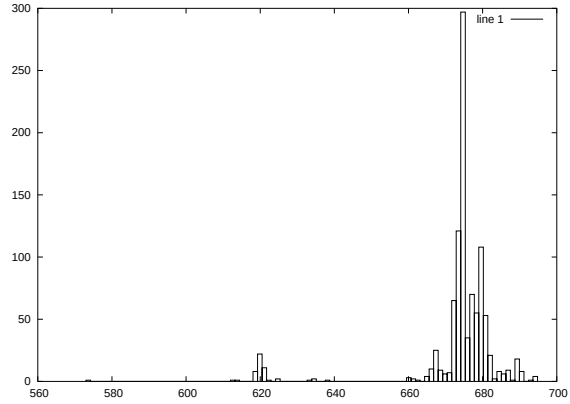
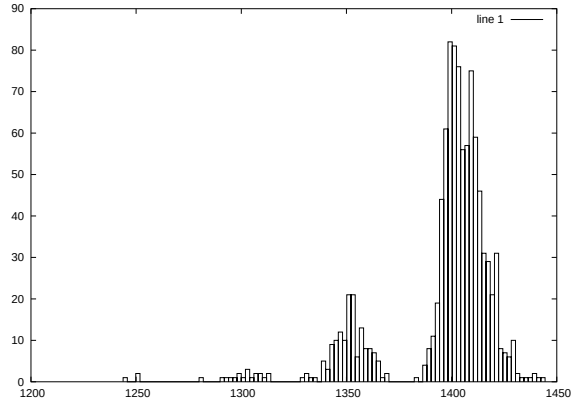


Figure A.7: Statistical distribution of the latency of emulation routine for operator '$*$'. Horizontal axis is latency in cycles, vertical is frequency over 1000 cases.

float



double



long double

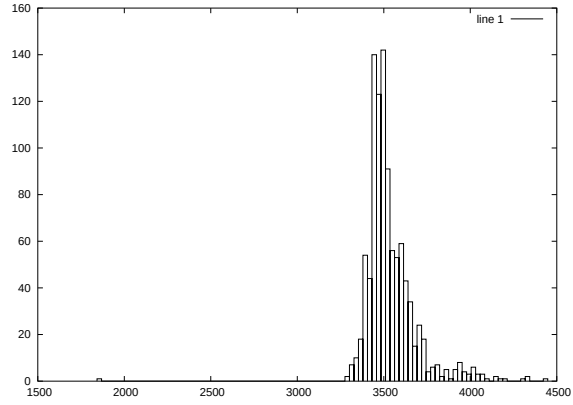


Figure A.8: Statistical distribution of the latency of emulation routine for operator '/'. Horizontal axis is latency in cycles, vertical is frequency over 1000 cases.

A.5 Relational operators

This section describes an experiment to determine the statistical distribution of the latencies of relational operators (==,!=, >=, < , <= and >) when applied on floating-point data. To comply with all the requirements for parasitic effect removal described in the previous section, I employed scripts and templates which are similar the ones already described in the previous section. The corresponding results follow.

Mean value			
	Float	Double	Long Double
==	87.44	126.39	169.46
!=	99.69	141.62	208.83
>=	104.38	126.76	201.78
<	116.76	148.79	253.72
<=	101.12	128.98	202.97
>	116.72	148.85	254.58

Table A.5: Average cost of floating-point relational operators, on operands extracted from uniform populations, expressed in clock cycles. Values with an asterisk (*) were not measurable due to deficiencies in the soft-float library implementation, and have been interpolated.

Standard deviation			
	Float	Double	Long Double
==	35.44	73.28	73.71
!=	35.91	73.59	* 73.64
>=	35.79	71.26	72.91
<	38.89	73.55	* 73.64
<=	35.66	73.26	73.98
>	35.89	73.54	* 73.63

Table A.6: Standard deviation of the cost of floating-point relational operators, on operands extracted from uniform populations, expressed in clock cycles. Values marked with an asterisk (*) were not measurable due to deficiencies in the soft-float library implementation, and have been interpolated.

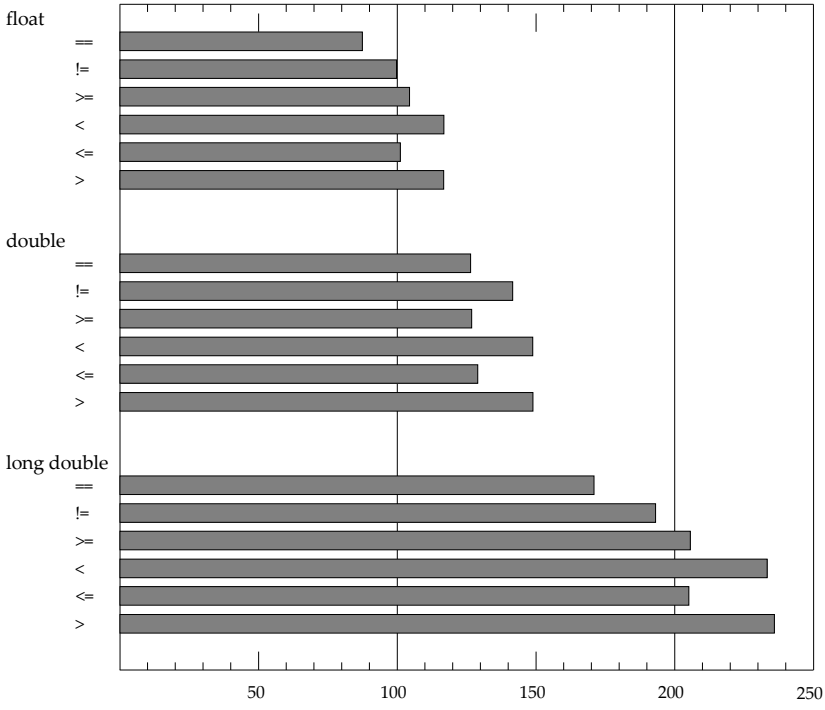
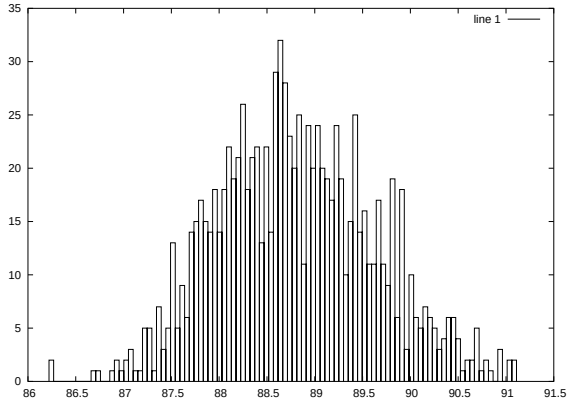
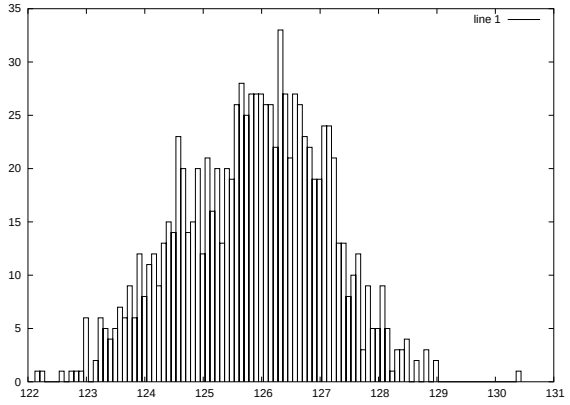


Figure A.9: Average cost of relational operators between operands of the various floating-point types.

float



double



long double

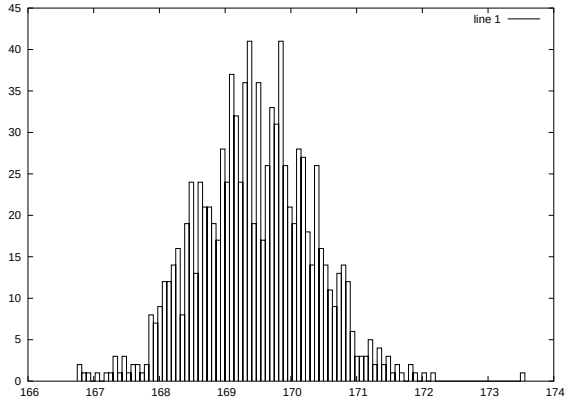
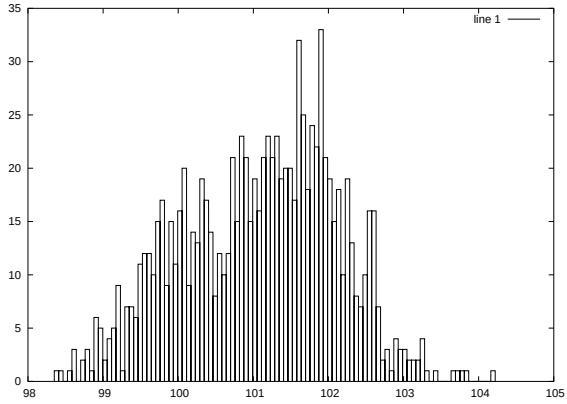
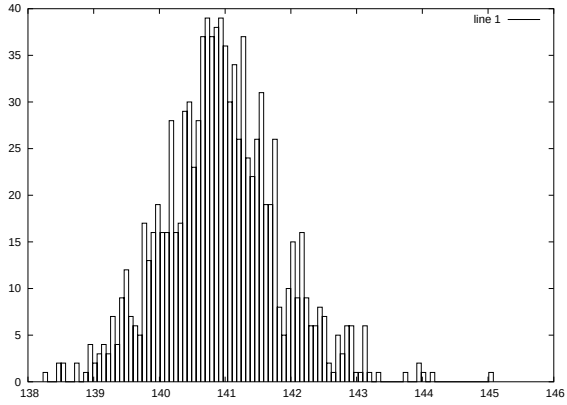


Figure A.10: Statistical distribution of the latency of emulation routine for operator '=='. Horizontal axis is latency in cycles, vertical is frequency over 1000 cases.

float



double



long double

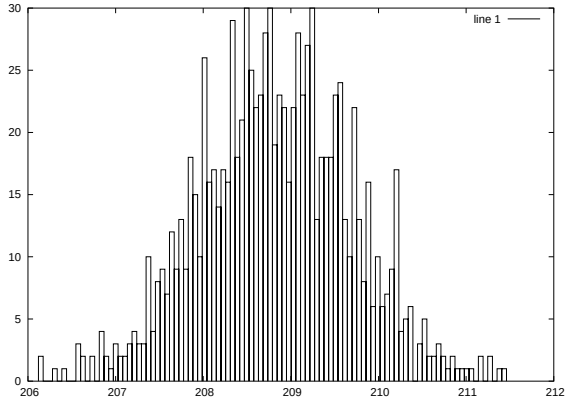
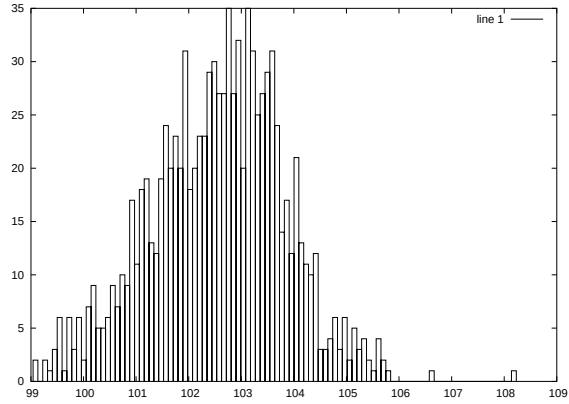
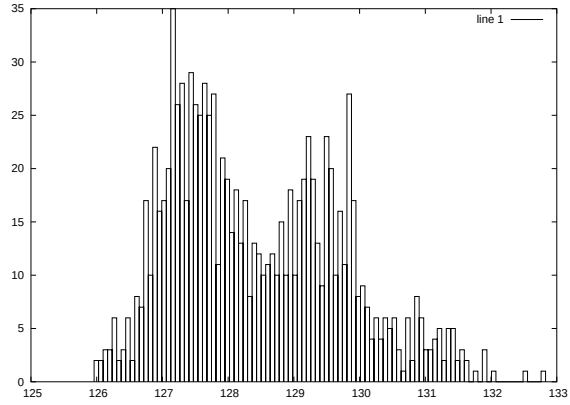


Figure A.11: Statistical distribution of the latency of emulation routine for operator '!='. Horizontal axis is latency in cycles, vertical is frequency over 1000 cases.

float



double



long double

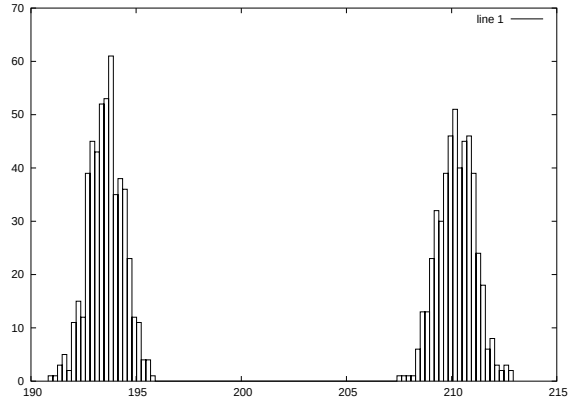
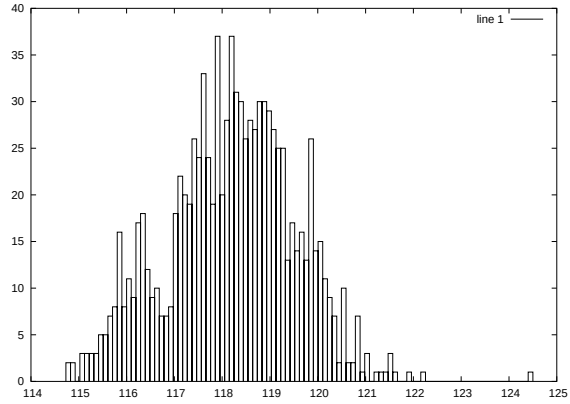
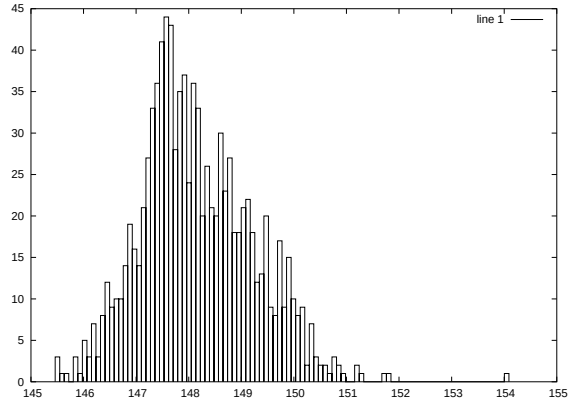


Figure A.12: Statistical distribution of the latency of emulation routine for operator '>='. Horizontal axis is latency in cycles, vertical is frequency over 1000 cases.

float



double



long double

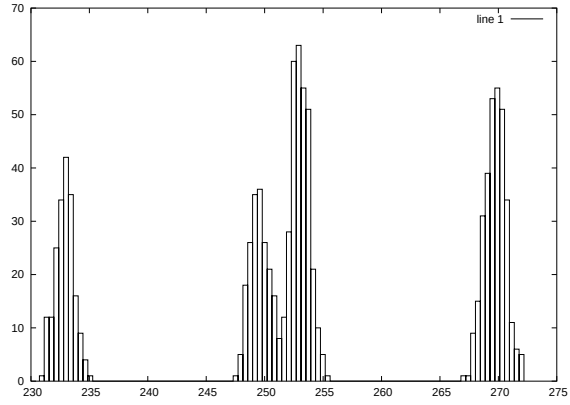
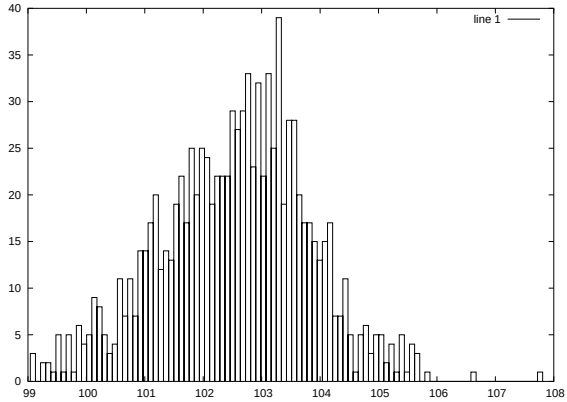
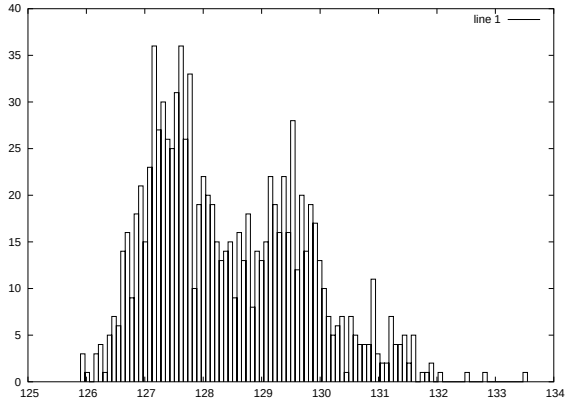


Figure A.13: Statistical distribution of the latency of emulation routine for operator '<'. Horizontal axis is latency in cycles, vertical is frequency over 1000 cases.

float



double



long double

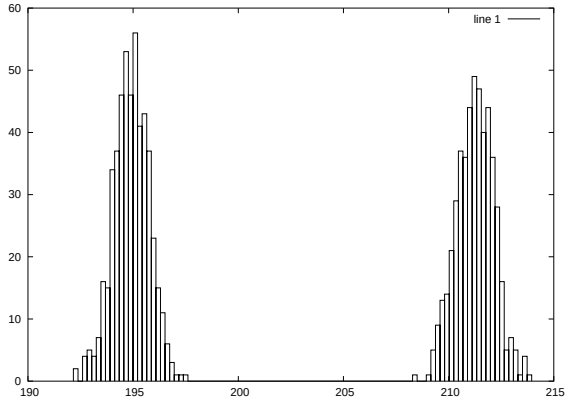
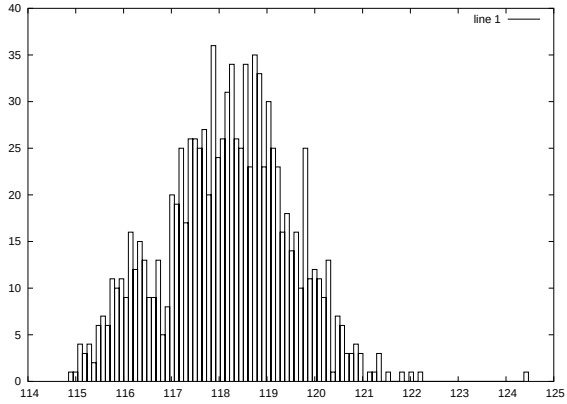
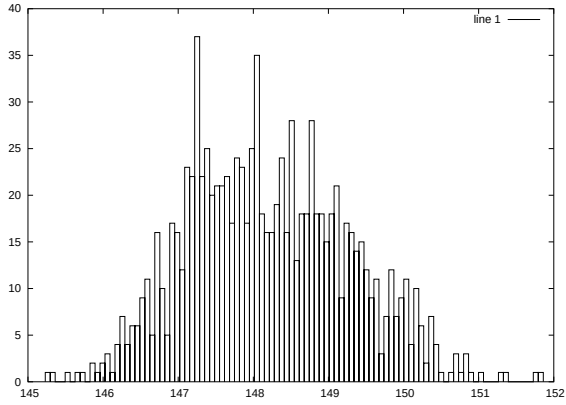


Figure A.14: Statistical distribution of the latency of emulation routine for operator '<='. Horizontal axis is latency in cycles, vertical is frequency over 1000 cases.

float



double



long double

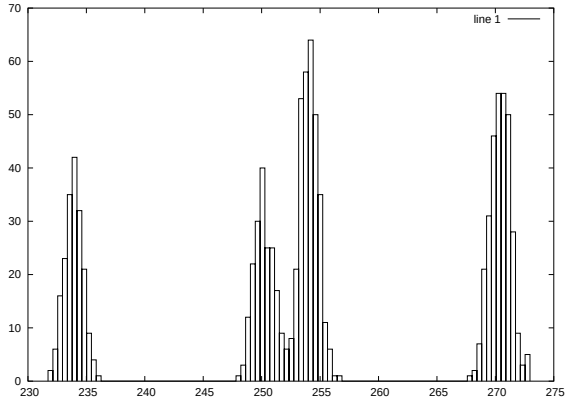


Figure A.15: Statistical distribution of the latency of emulation routine for operator '>'. Horizontal axis is latency in cycles, vertical is frequency over 1000 cases.

A.6 Dependence on data for arithmetic operations

In this section I examine how the cost of arithmetic operations is influenced by the value assumed by the arguments, especially by the exponent. To evaluate this, I divided the representable field of the three data types in 30 equally-spaced subfields, and for each of the 900 (30 *times* 30) subfields in which a couple of numbers could fall, I generated 30 different random floating-point values. For each of those subfields, I obtained the mean value and standard deviation of the execution time of the operation expressed in clock cycles.

The purpose of this investigation is to determine zones where the mean value differs significantly from the global average, and possibly determine its cause. For example, if in case of overflow the cost is significantly higher or lower than when overflow does not occur, and I know that in the application under estimation overflow never occurs, then it is reasonable to provide my application with an estimate for the execution of emulated routines that is different from the global average already calculated, and which only takes account of cases without overflow.

The result of the experiments are shown in the next pages. In all the cases, the standard deviation has not shown any significant irregularity. On the other hand, the mean value of the execution time exhibits significant changes. Apart from the long double data type, the following observations apply:

- all the plots show symmetry with respect to the main diagonal, which is the proof that the algorithms are cost-invariant with respect to the order of operands (e.g. the cost of $a + b$ is the same as $b + a$, as it is reasonable to expect);
- addition and subtraction show the same behavior as expected, since the two operations differ in their implementation only for sign exchange operations, which have negligible cost;
- in addition and subtraction, there is a "white band" along the main diagonal, which represents cases where the two operands are close enough to each other. In these cases, a and b are close enough (roughly $|log_{10}|a| - log_{10}|b|| <= 9$ for floats and $|log_{10}|a| - log_{10}|b|| <= 20$ for doubles) that the mantissa and exponent of the result must be actually calculated; in all other cases, they are far enough that the result is equal to one of the operands;
- for multiplication and division, there are a "white band" and a "light gray triangle". The plot for multiplication shows the same features as the division, except for a reflection on the horizontal axis. I attempt explanation for the above features on the multiplication plot; specular version of the same explanations apply for the division. The white band represents multiplication between numbers which produce very small results (in absolute values), which are represented in

non-normalized form, and it represents the additional costs involved in calculating the non-normalized form. Whatever is below this white stripe leads to underflow. The "light gray triangle" is associated with overflow. It is not clear why all the triangle above overflow leads to higher costs.

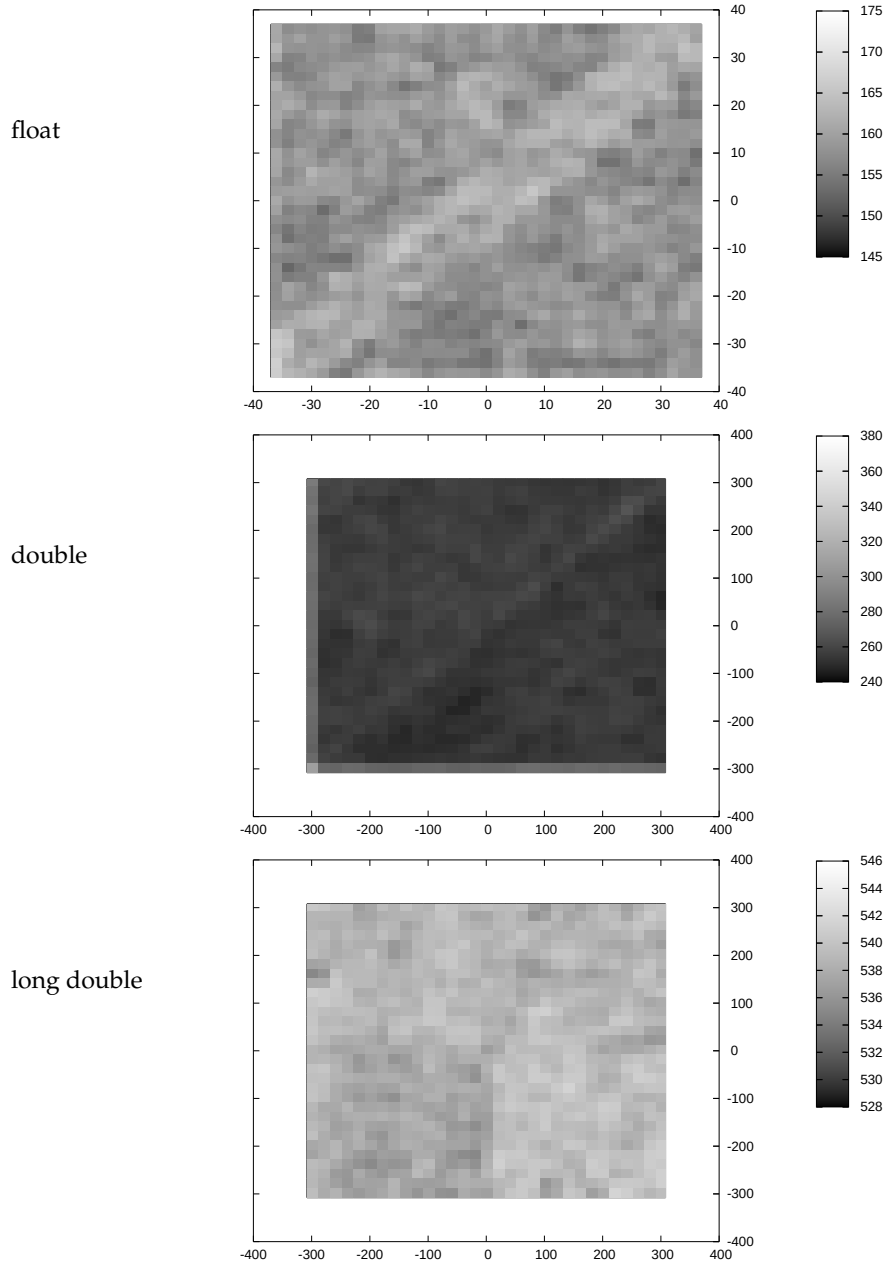


Figure A.16: Average cost of operator '+' depending on arguments. Horizontal and vertical axes are operand exponents, color is average cost over 30 random cases.

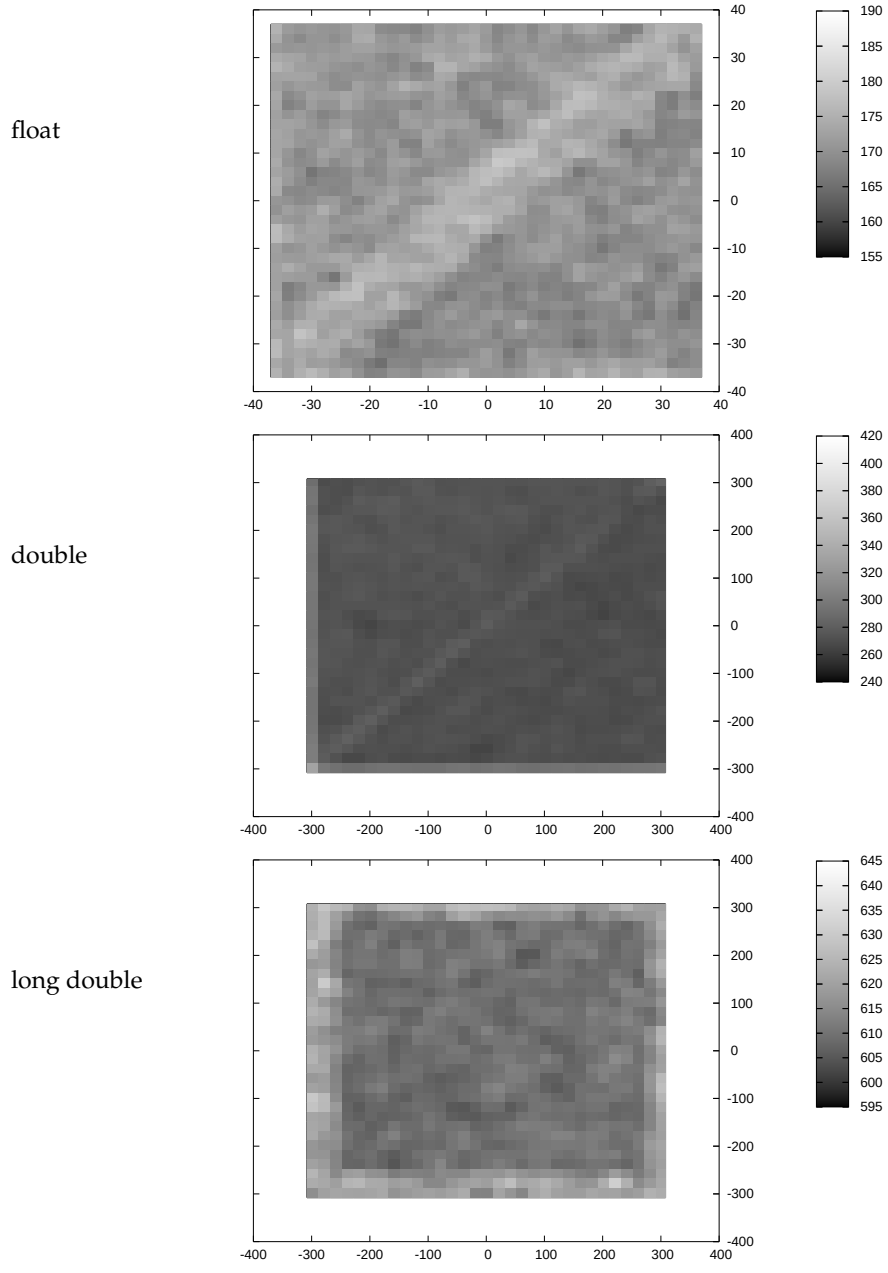


Figure A.17: Average cost of operator '−' depending on arguments. Horizontal and vertical axes are operand exponents, color is average cost over 30 random cases.

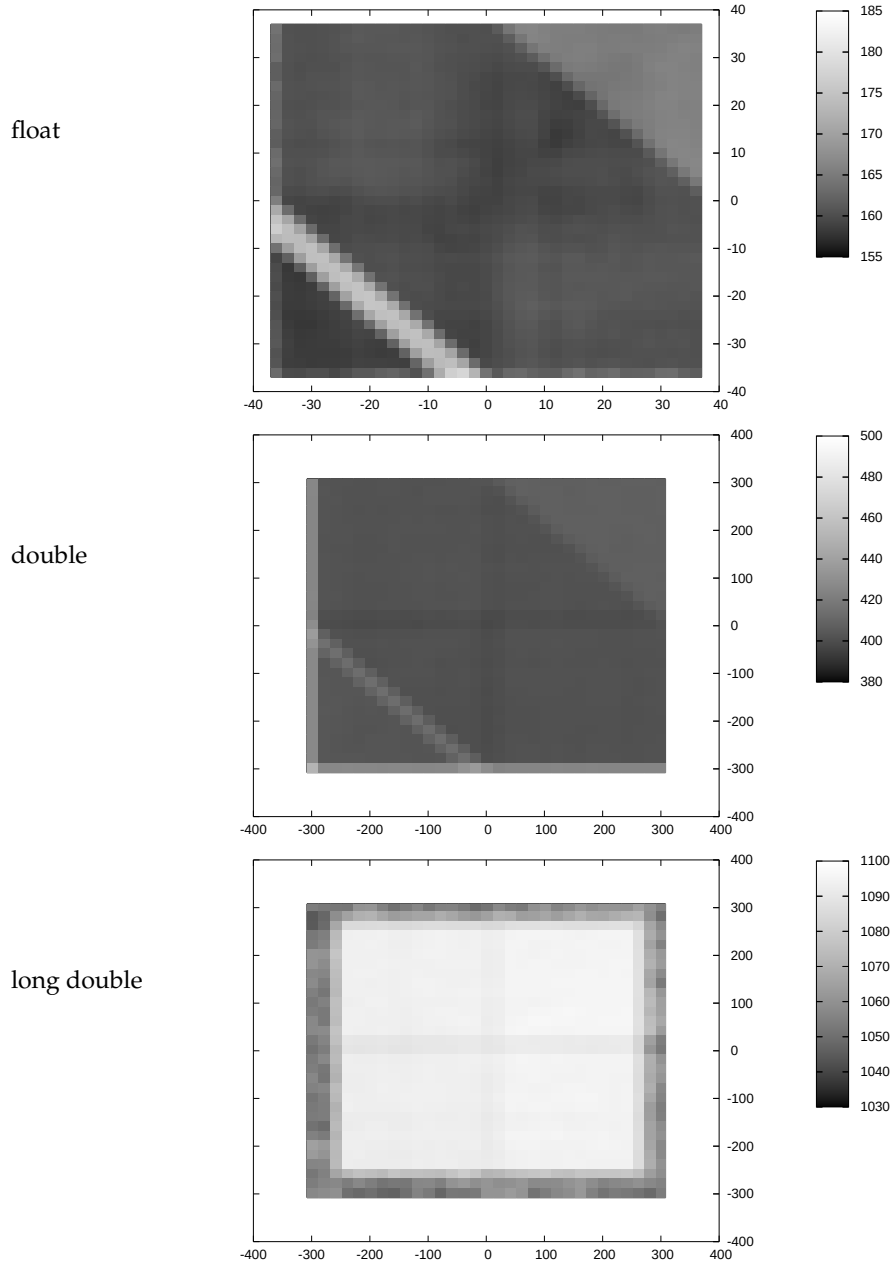


Figure A.18: Average cost of operator '$*$' depending on arguments. Horizontal and vertical axes are operand exponents, color is average cost over 30 random cases.

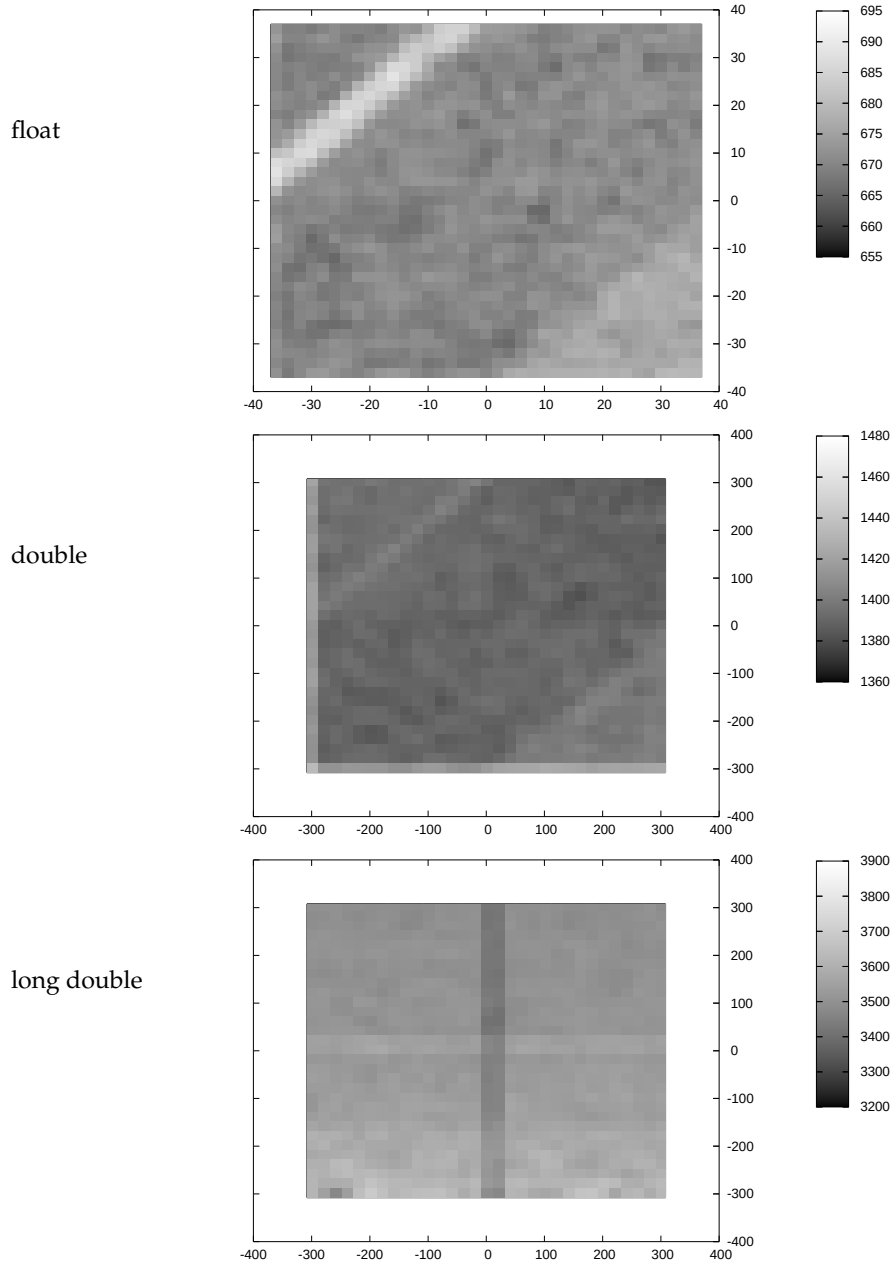


Figure A.19: Average cost of operator '/' depending on arguments. Horizontal and vertical axes are operand exponents, color is average cost over 30 random cases.

A.7 Dependence on data for relational operators

I am also interested in knowing whether the cost of relational expressions changes when portions of the encoded values of the two operands match. I distinguish the following 7 cases:

1. operand1 $>$ operand2 (different sign)
2. operand1 $>$ operand2 (same sign, different exponent)
3. operand1 $>$ operand2 (same sign and same exponent)
4. operand1 $=$ operand2
5. operand1 $<$ operand2 (same sign and same exponent)
6. operand1 $<$ operand2 (same sign, different exponent)
7. operand1 $<$ operand2 (different sign)

It is reasonable to expect how the above circumstances influence the cost of comparisons. For example the cost of $<$ and $>$ operators is lower when the operands have different sign, because in such a case, there is no need to examine their mantissa and exponent. The cost is slightly higher when they have the same sign but different exponent, and the operation involves exponent comparison. The cost is even higher when the two operands have same sign and exponent: in this case the operation must also compare the two mantissas. All these considerations are visually expressed by the plots in Figure A.20.

		Mean value						
Op.	Data type	Case						
		1	2	3	4	5	6	7
==	float	83.47	83.20	83.70	73.93	83.81	83.74	83.45
	double	121.44	122.08	125.00	115.18	124.22	122.20	121.56
	long double	170.39	169.26	169.35	177.12	169.14	169.42	170.13
!=	float	102.04	102.02	101.47	92.14	102.31	102.11	102.00
	double	137.18	137.20	139.31	130.21	139.05	137.51	137.25
	long double	209.34	208.25	208.31	216.07	208.54	209.21	209.11
>	float	118.28	120.38	122.12	121.16	122.00	119.39	118.33
	double	144.46	146.57	153.38	159.16	148.16	145.55	144.52
	long double	257.06	259.02	256.32	274.48	259.36	254.00	256.04
<=	float	105.33	107.13	109.15	107.53	108.20	106.18	105.30
	double	128.20	130.18	137.41	143.27	132.10	129.44	128.06
	long double	201.01	206.53	203.34	217.42	204.31	203.39	202.57
<	float	118.43	120.05	122.20	121.28	122.11	119.50	118.34
	double	144.15	146.35	153.27	159.17	148.12	145.06	144.05
	long double	256.35	258.28	255.52	274.14	258.51	253.17	255.10
>=	float	105.02	106.31	108.42	107.19	107.51	106.12	105.34
	double	128.23	130.12	137.40	143.31	132.38	129.40	128.14
	long double	200.19	205.45	202.14	216.41	203.15	202.23	201.38

Table A.7: Average costs of floating-point comparison operations depending on which parts of the operands' encoded value match.

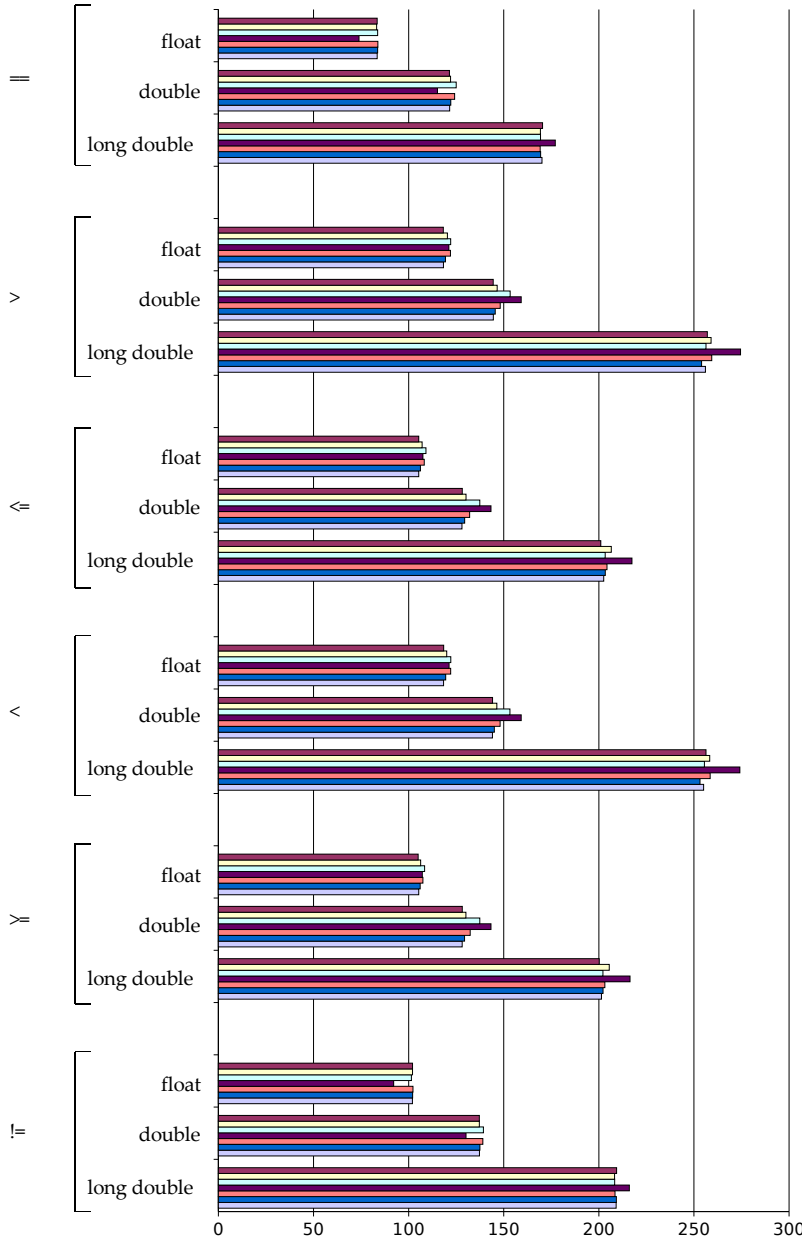


Figure A.20: Average costs of floating-point comparison operations depending on which parts of the operands' encoded value match.

Bibliography

Context

[1] G. E. Moore, Cramming More Components Onto Integrated Circuits, in *Electronics*, April 19, 1965, republished at: http://www.intel.com/technology/mooreslaw/;

[2] G. D. Hutcheson, Moore's Law: The History and Economics of an Observation that Changed the World, in *Electrochemical Society Interface*, March 2005, pages: 17–21, http://electrochem.org;

[3] S. Hamilton, Taking Moore's Law Into the Next Century, in *IEEE Computer*, Vol. 32, No. 1, pages: 43–48, January 1999;

[4] *The International Technology Roadmap for Semiconductors* ITRS 2004 Update, http://public.itrs.net, January 10, 2005,

[5] R.W. Keyes, Fundamental limits of silicon technology, in *Proceedings of the IEEE*, Vol. 89, No. 3, pages: 227–239, March 2001;

[6] M. Ajmone Marsan, S. Marano, C. Mastroianni and M. Meo, Performance Analysis of Cellular Mobile Communication Networks Supporting Multimedia Services, in *Mobile Networks and Applications*, Vol. 5, No. 3, pages: 167–177, September 2000;

[7] K. W. Richardson, UMTS overview in *Electronics and Communication Engineering Journal*, Vol. 12, No. 3, pages: 93–100, June 2000;

[8] International Organization for Standardisation, ISO/IEC JTC1/SC29/WG11 (MPEG), Coding of Moving Pictures and Audio, N4668, *MPEG-4 Overview*, March 2002;

[9] P. K. Doenges, T. K. Capin, F. Lavagetto, J. Ostermann, Igor S. Pandzic and E. D. Petajan, MPEG-4: Audio/Video & Synthetic Graphics/Audio for Mixed Media in *Image Communication Journal*, Special Issue on MPEG-4, Vol. 9, No. 4, May 1997;

[10] International Organization for Standardisation, IEEE Standard for Information technology – Telecommunications and information exchange between systems – Local and metropolitan area networks – Specific requirements, ISO/IEC 8802-11: 1999(E), ANSI/IEEE Std 802.11, 1999 Edition (R2003), Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications, available at: http://standards.ieee.org/getieee802/download/802.11-1999.pdf, 2003;

[11] IEEE Standard for Information technology – Telecommunications and information exchange between systems – Local and metropolitan area networks – Specific requirements, Part 15.1: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Wireless Personal Area Networks (WPANs), available at: http://standards.ieee.org/getieee802/download/802.15.1-2002.pdf, 2002;

[12] D. A. Wheeler, More Than a Gigabuck: Estimating GNU/Linux's Size, Version 1.07, June 30, 2001;

Artificial language issues

[13] D. E. Knuth, Semantics of Context-Free Languages, in *Theory of Computing Systems*, Vol. 2, No. 2, pages: 127–145, Springer-Verlag, New York, USA, June 1968;

[14] D. S. Wile, Abstract syntax from concrete syntax, in *Proc. International Conference on Software engineering (ICSE'97)*, pages: 472–480, Boston, MA, USA, 1997;

[15] D. S. Wile, Toward a Calculus for Abstract Syntax Trees, in *Proc. IFIP TC 2 WG 2.1 Intl. Workshop on Algorithmic Languages and Calculi*, IFIP Working Group 2.1, pages: 324–353, Strasbourg, France, 1997;

[16] L. Cardelli and A. D. Gordon, Mobile Ambients *Theoretical Computer Science*, Special Issue on Coordination, D. Le Métayer Editor. Vol 240/1, pages: 177–213, June 2000;

[17] C. Ghezzi and M. Jazayeri, *Programming Language Concepts*, Third Edition, John Wiley and Sons, 1998, ISBN 0-471-10426-4;

[18] L. J. J. Wittgenstein, *Logisch-Philosophische Abhandlung*, Annalen der Naturphilosophie, Vol. 14, Wilhelm Ostwald (ed.), 1921;

Performance estimation for embedded systems

[19] P. Puschner and Ch. Koza, Calculating the Maximum Execution Time of Real-Time Programs, in *Real-Time Systems*, Kluwer Academic Publishers, Norwell, MA, USA, Vol. 1, No. 2, pages: 159–176, September 1989;

[20] E. Kligerman and D. Stoyenko, Real-time Euclid: A language for reliable real time systems, in *IEEE Transactions on Software Engineering*, vol. SE-12, pages: 941-949, September 1986;

[21] K. Suzuki and A. Sangiovanni-Vincentelli, Efficient Software Performance Estimation Methods for Hardware/Software Codesign in *Proc. Design Automation Conference (DAC'96)*, pages: 605–610, June 1996;

[22] S. Malik, M. Martonosi and Y. T. S. Li, Static Timing Analysis for Embedded Software, in *Proc. Design Automation Conference (DAC'97)*, pages: 147–152, June 1997;

[23] K. Chen, S. Malik and D. I. August, Retargetable Static Timing Analysis for Embedded Software, in *Proc. International Symposium on Systems Synthesis (ISSS'01)*, pages: 39–44, Montréal, P.Q., Canada, 2001;

[24] A. Hergenhan and W. Rosenstiel, Static Timing Analysis of Embedded Software on Advanced Processor Architectures, in *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE'00)*, pages: 552–559, Paris, France, March 2000;

[25] P. Puschner and R. Nossal, Testing the Results of Static Worst-Case Execution-Time Analysis, in *Proc. IEEE Real-Time Systems Symposium (RTSS'98)*, pages: 134–143, 1998;

[26] S. Mallat and F. Falzon, Analysis of low bit rate image transform coding, in *IEEE Transactions on Signal Processing*, vol. 46, pages: 1027–1042, April 1998;

[27] M. Mattavelli and S. Brunetton, *A statistical study of MPEG-4 VM texture decoding complexity*, Technical Report M924, ISO-IEC/JTC1/SC29/WG11 MPEG-4, Tampere, Finland, July 1996;

[28] V. Tiwari, S. Malik and A. Wolfe, Power Analysis of Embedded Software: A First Step Towards Software Power Optimization, in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Vol. 2, No. 4, pages: 437–445, 1994;

[29] V. Tiwari, S. Malik, and A. Wolfe, Compilation Techniques for Low Energy: An Overview, in *Proc. 1994 Symposium on Low-Power Electronics*, San Diego, CA, USA, 1994;

[30] V. Tiwari, S. Malik, A. Wolfe and M. T.-C. Lee Instruction Level Power Analysis and Optimization of Software, in *Journal of VLSI Signal Processing*, pages: 1–18, Kluwer Academic Publishing, Boston, MA, USA, 1996;

[31] D. Burger and T. Austin, in *The SimpleScalar Tool Set, Version 2.0*, in Technical Report 1342, Computer Science Department, University of Wisconsin, Madison, WI, USA, 1997;

[32] W. Ye, N. Vijaykrishnan, M. Kandemir and M. J. Irwin, The Design and Use of SimplePower: A Cycle Accurate Energy Estimation Tool, in *Proc. Design Automation Conference (DAC'00)*, pages: 340–345, 2000;

[33] D. Brooks, V. Tiwari and M. Martonosi, Wattch: A Framework for Architectural Level Power Analysis and Optimizations, in *Proc. International Symposium on Computer Architecture (ISCA'00)*, pages: 83–94, June 2000;

[34] T. Simunic, L. Benini and G. De Micheli, Cycle-accurate simulation of energy consumption in embedded systems, in *Proc. Design Automation Conference (DAC'99)*, pages: 867–872, June 1999;

[35] W. Qin and S. Malik, Automated Synthesis of Efficient Binary Decoders for Retargetable Software Toolkits, in *Proc. Design Automation Conference (DAC'03)*, pages: 220–225, June 2003;

[36] W. Qin and S. Malik, Flexible and Formal Modeling of Microprocessors with Application to Retargetable Simulation, in *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE'03)*, p. 10556, Munich, Germany, 2003;

[37] D. C. Suresh, W. A. Najjar, F. Vahid, J. R. Villarreal, G. Stitt, Profiling tools for hardware/software partitioning of embedded applications, in *Proc. ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'03)*, San Diego, CA, USA, 2003;

[38] A. Stammermann, L. Kruse, W. Nebel, A. Pratsch, E. Schmidt, M. Schulte and A. Schulz, System Level Optimization and Design Space Exploration for Low Power, in *Proc. International Symposium on Systems Synthesis (ISSS '01)*, pages: 142–146, Montréal, P.Q., Canada, 2001;

[39] K. V. Seshu Kumar, Value reuse optimization: reuse of evaluated math library function calls through compiler generated cache, in *ACM SIGPLAN Notices Archive*, Vol. 38, No. 8, pages: 60–66, August 2003;

[40] S. Wilton and N. Jouppi, CACTI: An enhanced cache access and cycle time model, in *IEEE Journal of Solid-State Circuits*, Vol. 31, No. 5, p. 677–688, May 1996;

[41] R. A. Uhlig and T. N. Mudge, Trace-driven Memory Simulation: A Survey, in *ACM Computing Surveys*, Vol. 29, No. 2, p. 128–170, 1997;

[42] A. Sinha and A. P. Chandrakasan, JouleTrack: a Web-based Tool for Software Energy Profiling, in *Proc. Design Automation Conference (DAC'01)*, June 2001;

[43] T. Simunic, L. Benini and G. De Micheli, Energy Efficient Design of Battery-Powered Embedded Systems in Special Issue of *IEEE Transactions on VLSI Systems*, May 2001;

[44] M. Zhao, B. Childers and M. L. Soffa, Predicting the impact of optimizations for embedded systems, *Proc. ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'03)*, pages: 1–11, San Diego, CA, USA, 2003;

[45] E. Senn, N. Julien, J. Laurent and E. Martin, Power Consumption Estimation of a C Program for Data-Intensive Applications, in *Proc. Workshop on Complexity-Effective Design (WCED'02)*, May 2002;

[46] N. Julien, E. Senn, J. Laurent and E. Martin, Power Estimation of a C algorithm on a VLIW Processor, in *Proc. ISHPC-IV*, Kansai Science City, Japan, May 2002;

[47] M. Ravasi and M. Mattavelli, High-level algorithmic complexity evaluation for system design, in *Journal of Systems Architecture*, No. 48, pages: 403–427, 2003;

[48] M. Ravasi, *An Automatic C-code Instrumentation Framework for High Level Algorithmic Complexity Analysis and System Design* Ph. D. dissertation, document 2839 (2003), École Polytechnique Fédérale de Lausanne, Switzerland, September 2003;

[49] M. Ravasi, M. Mattavelli1, P. Schumacher and R. Turney, High-Level Algorithmic Complexity Analysis for the Implementation of a Motion-JPEG2000 Encoder, in *Lecture Notes in Computer Science*, Springer-Verlag GmbH, Vol. 2799, pages: 440–450, 2003;

[50] T. K. Tan, A. K. Raghunathan, G. Lakishminarayana and N. K. Jha, High-level software energy macro-modeling, in *Proc. Design Automation Conference (DAC'01)*, Las Vegas, Nevada, USA, pages: 605–610, 2001;

[51] A. Muttreja, A. Raghunathan, S. Ravi and N. K. Jha, Automated energy/performance macromodeling of embedded software, in *Proc. Design Automation Conference (DAC'04)*, pages: 99–102, June 2004;

[52] J. T. Russell and M. F. Jacome, Software Power Estimation and Optimization for High Performance, 32-bit Embedded Processors, in *Proc. International Conference on Computer Design (ICCD'98)*, pages: 328–333, 1998;

[53] M. Reshadi, P. Mishra and N. Dutt, Instruction Set Compiled Simulation: A Technique for Fast and Flexible Instruction Set Simulation, in *Design Automation Conference (DAC'03)*, pages: 758–763, Anaheim, CA, USA, June 2003;

[54] Y. Li and J. Henkel, A framework for estimating and minimizing energy dissipation of embedded HW/SW systems, In *Proc. Design Automation Conference (DAC'98)*, pages: 188–193, June 1998;

[55] J. Henkel and Y. Li, Avalanche: An Environment for Design Space Exploration and Optimization of Low-Power Embedded Systems *IEEE Trans. VLSI Systems*, Vol. 10, No. 4, p. 454–468, August 2002;

[56] C. Brandolese, W. Fornaciari, F. Salice and D. Sciuto, Energy Estimation for 32-bit Microprocessors in *Proc. International Workshop on Hardware/Software Codesign (CODES 2000)*, Mission Bay, San Diego, May 2000;

[57] C. Brandolese, W. Fornaciari, F. Salice and D. Sciuto, Source-Level Execution Time Estimation of C Programs, in *Proc. International Workshop on Hardware/Software Codesign (CODES 2001)*, Copenhagen, Denmark, 2001;

[58] C. Brandolese, W. Fornaciari, F. Salice and D. Sciuto, An Instruction-level Functionality-based Energy Estimation Model for 32-bit Microprocessors, in *Proc. Design Automation Conference (DAC'00)*, Los Angeles, CA, USA, June 2000;

[59] C. Brandolese, W. Fornaciari, F. Salice and D. Sciuto, Library Functions Timing Characterization for Source–Level Analysis, in *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE'03)*, Munich, Germany, pages: 1132–1133, March 2003;

Instrumentation techniques

[60] J. K. Hollingsworth, B. P. Miller, and J. Cargille. Dynamic program instrumentation for scalable performance tools. in *Proc. of Scalable High-Performance Computing Conference (SHPCC'94)*, pages: 841–850, May 1994.

[61] M. D. Smith, Tracing with Pixie, in *Technical Report CSL-TR-91-497*, Computer Systems Laboratory, Stanford University, Stanford, CA, USA, November 1991;

[62] A. Srivastava and A. Eustace, ATOM: A System for Building Customized Program Analysis Tools, in *Proc. ACM SIGPLAN Conference on Programming Language Design an Implementation (PLDI'94)*, Orlando, FL, USA, pages: 196–205, June 1994;

[63] J. R. Larus and E. Schnarr, EEL: Machine-Independent Executable Editing, in *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI'95)*, pages: 291–300, June 1995;

[64] J. R. Larus, Efficient Program Tracing, in *IEEE Computer*, May 1993, pages: 291–300, 1993;

[65] T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy and B. Bershad, Instrumentation and Optimization of Win32/Intel Executables Using Etch in *Proc. USENIX Windows NT Workshop*, pages: 1–8, Seattle, Washington, August 1997;

[66] J. R. Larus, Abstract Execution: A Technique for Efficiently Tracing Programs, in *Software Practice and Experience*, vol. 20, n. 12, pages: 1241–1258, December 1990;

[67] M. Lajolo, M. Lazarescu and A. Sangiovanni-Vincentelli, A Compilation-based Software Estimation Scheme for Hardware/Software Co-simulation, in *Proc. International Workshop on Hardware/Software Codesign (CODES'99)*, Rome, Italy, pages: 85–89, 1999;

[68] K. Templer and C. L. Jeffery, A configurable automatic instrumentation tool for ANSI C, in *Proc. of the Automated Software Engineering Conference*, pages: 249–253, 1998;

[69] L. DeRose, D. Reed, SvPablo: A Multi-Language Architecture-Independent Performance Analysis System, in *Proc. International Conference on Parallel Processing (ICPP'99)*, Fukushima, Japan, September 1999;

[70] R. F. Cmelik, *SpixTools introduction and user's manual*, Technical Report TR-93-6, Sun Microsystems Laboratories, Palo Alto, CA, 1993;

[71] T. Ball and J. R. Larus, Optimally Profiling and Tracing Programs, in *ACM Transactions on Programming Languages & Systems*, Vol. 16, pages: 1319–1360, July 1994;

[72] M. Ducassé and J. Noye, Tracing Prolog programs by source instrumentation is efficient enough in *Journal of Logic Programming*, Vol. 43, No. 2, pages: 157–172, May 2000;

[73] E. Jahier and M. Ducassé, Generic Program Monitoring by Trace Analysis, in *Theory and Practice of Logic Programming Journal*, Special Issue on Program Development, Cambridge University Press, Vol. 2 parts 4 and 5, pages: 613–645, September 2002;

[74] C. X. Zhang, Z. Wang, N. C. Gloy, J. B. Chen and M. D. Smith, System Support for Automated Profiling and Optimization, in *Proc. Symposium on Operating Systems Principles*, pages: 15–26, 1997;

[75] P. T. Devanbu, GENOA - A Customizable, front-end retargetable Source Code Analysis Framework in *Proc. International Conference on Software engineering (ICSE '92)*, pages: 307–317, Melbourne, Australia, 1992;

Compiler design

[76] A.V. Aho, R. Sethi and J. D. Ullman, *Compilers: Principles, Techniques and Tools*, Addison Wesley, ISBN: 0-201100-88-6, January 1986;

[77] D. Grune, H. E. Bal, C. J. H. Jacobs and K. G. Langendoen, *Modern Compiler Design*, John Wiley and Sons, ISBN 0-471976-97-0, August 2000;

[78] A. Appel, *Modern Compiler Implementation in C*, Cambridge University Press, ISBN 0-521-60765-5, 1998;

[79] C. W. Fraser and D. R. Hanson, *A Retargetable C Compiler: Design and Implementation*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, ISBN: 0805316701, 1995;

The C and C++ programming languages

[80] International Standards Organization, Technical committee JTC 1 / SC 22, "Programming languages, their environments and system software interfaces", *Programming languages – C*, standard ISO/IEC 9899:1990, 1990;

[81] H. Schildt, *The Annotated ANSI C Standard*, Osborne McGraw-Hill, Berkeley, CA, USA, ISBN: 0-07-881952-0, 1990;

[82] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Second Edition, Prentice-Hall, ISBN 0-13-110362-8, 0-13-110370-9, 1988;

[83] P. van den Linden, *Expert C Programming – Deep C Secrets*, Prentice Hall Professional Technical Reference, ISBN: 0-13-177429-8, June 1994;

[84] M. A. Ellis and B. Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley, ISBN: 0-20-151459-1, April 2001;

[85] E. Willink, *"Meta-compilation for C++"*, Ph.D. Thesis, Department of Computing, University of Surrey, UK, 1999;

[86] The Free Software Foundation, *The GNU Compiler Collection Internals*, http://gcc.gnu.org/onlinedocs/gccint;

Compiler optimizations

[87] R. G. Cattell, Automatic Derivation of Code Generators from Machine Descriptions, in *ACM Transactions on Programming Languages and Systems (TOPLAS)*, Vol. 2, No. 2, pages: 173–190, 1980;

[88] A. V. Aho and S. C. Johnson, Optimal code generation for expression trees, in *Journal of the ACM*, Vol. 23, No. 3, pages: 488–501, July 1976;

[89] A. V. Aho, M. Ganapathi and S. W. Tjiang, Code generation using tree matching and dynamic programming, in *ACM Transactions on Programming Languages and Systems (TOPLAS)*, Vol. 11, No. 4, pages: 491–516, October 1989;

[90] H. Massalin, Superoptimizer: A Look At The Smallest Program, in *Proc. International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS'87)*, Vol. 22, No. 10, pages: 122–127, New York, NY, 1987;

[91] T. Granlund and R. Kenner, Eliminating branches using a superoptimizer and the GNU C compiler. in *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'92)*, San Francisco, CA, pages: 341–352, June 1992;

[92] G. J. Chaitin, Register allocation and spilling via graph coloring. in *Proc. ACM SIGPLAN Symposium on Compiler Construction (SIGPLAN '82)*, Boston, MA, USA, June 23-25, pages: 98–101;

[93] M. N. Wegman and F. K. Zadeck, Constant propagation with conditional branches, in *ACM Transactions on Programming Languages and Systems (TOPLAS)*, Vol. 13, No. 2, 1991, pages: 181–210, 1991;

[94] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman and F. K. Zadeck, Efficiently Computing Static Single Assignment Form and the Control Dependence Graph, in *ACM Transactions on Programming Languages and Systems (TOPLAS)*, Vol. 13, No. 4, pages: 451–490, October 1991;

[95] R. Metzger and S. Stroud, Interprocedural Constant Propagation: An Empirical Study, in *ACM Letters on Programming Languages and Systems*, Vol. 1, No. 2, pages: 213–232, June 1992;

[96] K. D. Cooper, L. T. Simpson and C. A. Vick, Operator strength reduction, in *ACM Transactions on Programming Languages and Systems (TOPLAS)*, Vol. 23, No. 5, pages: 603–625, September 2001;

[97] J. Cocke, Global Common Subexpression Elimination, in *Proc. Symposium on Compiler Construction*, SIGPLAN Notices, Vol.5, No. 7, pages: 20–24, July 1970;

[98] S. S. Muchnick and N. D. Jones, *Program Flow Analysis: Theory and Application*, Prentice Hall Professional Technical Reference, ISBN: 0137296819, 1981;

[99] F. E. Allen and J. Cocke, A Catalogue of Optimizing Transformations, in *Design and Optimization of Compilers*, pages: 1–30, Prentice Hall, 1972;

[100] J. Knoop, O. Ruething and B. Steffen, Lazy code motion, in *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'92)*, SIGPLAN Notices, Vol. 27, No. 7, pages: 224–234, June 1992;

[101] F. Müller and D. B. Whalley, Avoiding unconditional jumps by code replication, *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'92)*, SIGPLAN Notices, Vol. 27, No. 7, pages: 322–330, July 1992;

[102] A. Aiken and A. Nicolau, Optimal Loop Parallelization, in *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'88)*, pages: 308–317, Atlanta, Georgia, USA, 1988;

[103] B. R. Rau, Iterative Modulo Scheduling: an Algorithm for Software Pipelining Loops, in *Proc. IEEE International Symposium on Microarchitecture (MICRO 27)*, pages: 63–74, San Jose, California, USA, 1994;

[104] B. R. Rau, Iterative Modulo Scheduling: an Algorithm for Software Pipelining Loops, in *International Journal of Parallel Programming*, Vol. 24, No. 1, pages: 3–64, 1996;

[105] J. A. Fisher, Trace scheduling: a technique for global microcode compaction, in *Instruction-level parallel processors*, IEEE Computer Society Press, Los Alamitos, CA, USA, ISBN: 0-8186-6527-0, pages: 186–198, 1995;

[106] J. E. Smith, A study of branch predition strategies, in *Proc. 8th Annual International Symposium on Computer Architecture (ISCA'81)*, pages: 135–148, June 1981;

[107] T. Ball, J. R. Larus, Branch Prediction for Free, in *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '93)*, Albuquerque, New Mexico, USA, pages: 300–313, June 1993;

[108] Y. Wu and J.R. Larus, Static branch frequency and program profile analysis, in *Proc. IEEE International Symposium on Microarchitecture (MICRO 27)*, San Jose, CA, USA, pages: 1–11, November 1994;

[109] J. R. C. Patterson, Accurate Static Branch Prediction by Value Range Propagation", in *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'95)*, pages: 67–78, La Jolla, San Diego, CA, USA, June 1995;

[110] Andreas Krall, Improving Semi-static Branch Prediction by Code Replication, in *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'94)*, Vol. 29, No. 7, pages: 97–106, Orlando, USA, June 1994;

[111] R. A. Johnson and M. S. Schlansker, Analysis Techniques for Predicated Code, in *Proc. IEEE International Symposium on Microarchitecture (MICRO 29)*, pages: 100–113, San Jose, California, USA, 1996;

[112] J. C. H. Park and M. S. Schlansker, *On predicated execution*, Technical Report HPL-91-58, Hewlett-Packard Laboratories, Palo Alto CA, May 1991;

[113] R. A. Johnson and M. S. Schlansker, *Analysis of Predicated Code*, Technical Report HPL-96-119, Hewlett-Packard Laboratories, Palo Alto CA, December 1996;

[114] T. C. Mowry, M. S. Lam and A. Gupta, Design and Evaluation of a Compiler Algorithm for Prefetching, in *Proc. Intl. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'92)*, SIGPLAN Notices, Vol. 27, No. 9, pages: 62–73, 1992;

[115] J. R. Allen and K. Kennedy, Automatic Loop Interchange, in *Proc. ACM SIGPLAN Symposium on Compiler Construction (SIGPLAN '84)*, Montréal, Canada, pages: 233–246, 1984;

[116] F. Irigoin and R. Triolet, Supernode Partitioning. in *Proc. ACM Symposium on the Principles of Programming Languages (POPL'88)*, pages: 319–329, San Diego, CA, USA, January 1988;

[117] K. S. McKinley, S. Carr and C.-W. Tseng, Improving Data Locality with Loop Transformations, in ACM Transactions on Programming Languages and Systems, pages: 424–453, Vol. 18, No. 4, July 1996;

[118] V. Sarkar, Optimized unrolling of nested loops, in *Proc. International Conference on Supercomputing (ICS'00)*, pages: 153–166, 2000;

[119] J. Lu and K. D. Cooper, Register promotion in C programs, in *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '97)*, pages: 308–319, Las Vegas, Nevada, USA, 1997;

[120] S. Carr and K. Kennedy, Scalar replacement in the presence of conditional control flow, in *Software – Practice & Experience*, Vol. 24, No. 1, pages: 51–77, John Wiley & Sons, Inc., New York, NY, USA, January 1994;

[121] K. Pettis and R.C. Hansen, Profile Guided Code Positioning, in *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'90)*, SIGPLAN Notices, Vol. 25, No.6, pages: 16–27, White Plains, NY, USA, June 1990;

[122] J. W. Davidson, A. Holler, A Study of a C Function Inliner, in *Software–Practice and Experience*, Vol. 19, No. 1, pages: 79–97, January 1988;

[123] K. D. Cooper, M. W. Hall, L. Torczon, An Experiment with Inline Substitution, in *Software–Practice and Experience*, Vol. 21, No. 6, pages: 581–601, June 1991;

[124] Jack W. Davidson and Anne Holler, Subprogram Inlining: A Study of Its Effects on Program Execution Time, in *IEEE Transactions on Software Engineering*, Vol. 18, No. 2, pages: 89–102, 1992;

[125] P. P. Chang, S. A. Mahlke, W. Y. Chen and W. W. Hwu, Profile-guided Automatic Inline Expansion for C Programs, in *Software–Practice and Experience*, Vol. 22, No. 5, pages: 349–369, 1992;

Source-level optimizations

[126] C. Brandolese, Analysis and modeling of energy reducing source code transformations in *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE'04)*, Vol. 3, pages: 306-311, Paris, France, February 2004;

[127] The Stanford Compiler Group, *The SUIF Library: a set of core routines for manipulating SUIF data structures*, Stanford University Press, 1994;

[128] T. Simunic, L. Benini and G. De Micheli, Energy Efficient Design of Battery-Powered Embedded Systems in Special Issue of *IEEE Trans. on VLSI Systems*, May 2001;

[129] C. Brandolese, W. Fornaciari, F. Salice and D. Sciuto, The Impact of Source Code Transformations on Software Power And Energy Consumption, in *Journal of Circuits, Systems, and Computers (JCSC)*, Vol. 11, No. 5, 2002, pages. 477–502;

[130] B. Franke, M. O'Boyle, J. Thomson, G. Fursin, Probabilistic Source-level Optimisation of Embedded Programs, in *Proc. ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'05)*, pages: 78–86, Chicago, Illinois, USA, 2005;

[131] S. Triantafyllis, M. Vachharajani, N. Vachharajani and D. I. August, Compiler Optimization-Space Exploration, in *Proc. International Symposium on Code Generation and Optimization (CGO'03)*, p. 204–215, March 2003;

[132] Y. Fei, S. Ravi, A. Raghunathan and Niraj K. Jha, Energy-Optimizing Source Code Transformations for OS-driven Embedded Software, in *Proc. International Conference on VLSI Design* (VLSI Design 2004), pages: 261–266, Mumbai, India, January 2004;

[133] E.-Y. Chung, L. Benini and G. De Micheli, Source code transformation based on software cost analysis, in *Proc. International Symposium on Systems Synthesis (ISSS '01)*, pages: 153–158, Montréal, P.Q., Canada, 2001;

[134] S. E. Richardson, Caching Function Results: Faster Arithmetic by Avoiding Unnecessary Computation, in *Sun Microsystems Technical Report SMLI TR-92-1*, September 1992;

[135] E.-Y. Chung, G. De Micheli, M. Carilli, L. Benini and G. Luculli, Value-based Source Code Specialization for Energy Reduction in *ST Journal of System Research*, Vol 3, No. 1, April 2002;

[136] J. Bormans, K. Denolf, S. Wuytack, L. Nachtergaele and I. Bolsens, Integrating System-level Low Power Methodologies into a Real-life Design Flow, *IEEE PATMOS'99*, pages: 19–28, 1999;

[137] G. Arnout, *PowerEscape, Maximizing Data Efficiency for Power and Performance*, White paper, http://www.powerescape.com/technology/papers, 2005;

[138] G. Agosta, G. Palermo and C. Silvano, Multi-Objective Co-Exploration of Source Code Transformations and Design Space Architectures for Low-Power Embedded Systems, *Proc. ACM Symposium on Applied Computing*, pages: 891–896, Nicosia, Cyprus, 2004;

[139] S. Triantafyllis, M. Vachharajani, N. Vachharajani and D. I. August, Compiler Optimization-Space Exploration, *Proc. Intl. Symposium on Code Generation and Optimization*, pages: 204–215, March 2003;

[140] E. Zitzler, J. Teich and S. S. Bhattacharyya, Multidimensional Exploration of Software Implementation for DSP Algorithms, in *Journal of VLSI Signal Processing Systems*, Vol. 24, No. 1, Kluwer Academic Publishers, 1999;

[141] H. Blume, H. Hübert, H. T. Feldkämper and T. G. Noll, Model-based Exploration of the Design Space for Heterogeneous Systems on Chip *Journal of VLSI Signal Processing Systems*, Vol. 40 No. 1, p. 19–34, May 2005;

[142] A. Peymandoust, T. Simunic and G. De Micheli, Low Power Embedded Software Optimization using Symbolic Algebra *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE'02)*, Paris, France, 2002;

Compiler design tools

[143] C. Donnelly and R. M. Stallman, *Bison Manual: Using the Yacc-compatible Parser Generator* GNU Press, Free Software Foundation, Boston, MA, USA, September 2003, ISBN 1-882114-23-X;

[144] C. Donnelly and R. M. Stallman, *Bison: the Yacc-compatible Parser Generator* http://www.gnu.org/software/bison/manual/, September 2005;

[145] V. Paxson, *Flex, version 2.5: a fast scanner generator* http://www.gnu.org/software/flex/manual/, March 1995;

[146] J. Fenlason, R. Stallman, *GNU gprof, The GNU Profiler*, http://www.gnu.org/software/flex/manual/, September 1997;

[147] J. Lee and J. Degener, ANSI C Yacc grammar, http://www.lysator.liu.se/c/ANSI-C-grammar-y.html;

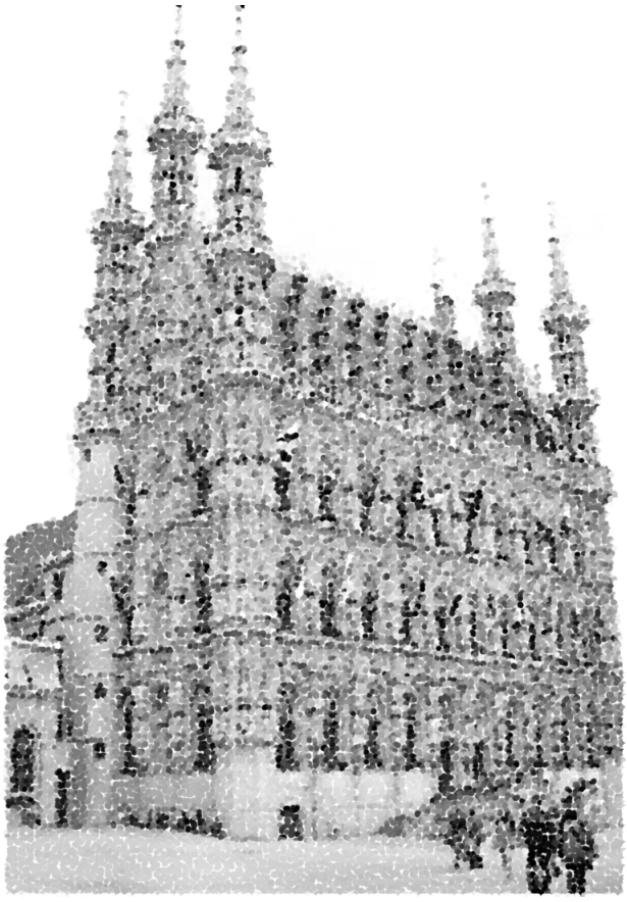
[148] E. M. Gagnon and L. J. Hendren, SableCC, an Object-Oriented Compiler Framework, in *Proc. Technology of Object-Oriented Languages and Systems (TOOLS'98)*, 1998, pages: 140–154;

[149] T. J. Parr and R. W. Quong, ANTLR: A Predicated-LL(k) Parser Generator, in *Software – Practice & Experience*, Vol. 25 , No. 7, pages: 789–810, John Wiley & Sons, York, NY, USA, July 1995;

Benchmarks

[150] M. R. Guthaus et al., MiBench: A free, commercially representative embedded benchmark suite, in *Proc. IEEE 4th Annual Workshop on Workload Characterization*, Austin, TX, USA, Dec. 2001;

[151] Telenor R & D, Digital Video Coding at Telenor R & D, *Telenor's H.263 Software*, http://www.nta.no/brukere/DVC/h263 software/



Typeset by the author in Palatino Linotype with PDFLaTeX.
Printed by ACCO (Academische Coöperatief) Drukkerij,
Kaboutermanstraat 30, 3000 Leuven, België.

POLITECNICO DI MILANO
Dipartimento di Elettronica e Informazione
Piazza Leonardo da Vinci, 32 I-20133 Milano, Italia