On the Side-Effects of Code Abstraction

Bjorn De Sutter
brdsutte@elis.ugent.be

Hans Vandierendonck
hvdieren@elis.ugent.be

Bruno De Bus
bdebus@elis.ugent.be

Koen De Bosschere
kdb@elis.ugent.be

Electronics and Information Systems (ELIS) Department
Ghent University, Sint-Pietersnieuwstraat 41
9000 Gent, Belgium

ABSTRACT

More and more devices contain computers with limited amounts of memory. As a result, code compaction techniques are gaining popularity, especially when they also improve performance and power consumption, or at least not degrade it. This paper quantifies the side-effects of code abstraction on performance using extensive measurements and simulations on the SPECint2000 benchmark suite and some additional C++ programs. We show how to use profile information in order to obtain almost all the code size reduction benefits of code abstraction, yet experience almost none of its disadvantages.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*code generation;compilers;optimization*; E.4 [**Coding and Information Theory**]: Data Compaction and Compression—*program representation*

General Terms

Experimentation, Performance

Keywords

performance, code abstraction, code compaction

1. INTRODUCTION

More and more devices contain computers with limited amounts of memory. Examples are PDAs, set top boxes, wearables, mobile and embedded systems in general. The limitations on memory size result from considerations such as space, weight, power consumption and production cost. As a result, the last decade has witnessed a growing research into the automated generation of smaller programs using compaction and compression techniques.

As the field of program compaction and compression matures, side-effects of these techniques are more and more taken into account. The goals for code compaction are gradually shifting. Whereas reducing program size was for a long time the main objective, the focus has now changed to reducing program size while maintaining or even improving performance and lowering power consumption.

Code abstraction is a technique by which a program fragment that occurs multiple times in a program is abstracted into a separate procedure. The original occurrences of the fragment are replaced by a call to the abstracted procedure. Code abstraction techniques have proven very efficient in reducing code sizes of programs in general, and of C++ programs in particular, leading to code size reductions of up to 35% and more [9].

Unfortunately, code abstraction also introduces a significant amount of run-time overhead: more instructions will be executed, in particular more procedure calls and returns. This influences performance-related design criteria such as instruction counts and instruction cache hit rates. As a result, embedded system developers in practice often discard code abstraction because design criteria like performance and power consumption are even more important than code size.

In this paper we present the first extensive empirical study of the side-effect of code abstraction. With this study, we refute the performance-related arguments for not applying code abstraction. We discuss and extensively quantify the side-effects of code abstraction on execution speed, instruction cache behavior, code schedule quality and branch prediction. Previously we suggested to avoid performance degradation by using profile information while still obtaining significant code size reductions [9]. The main contribution of this paper is the rigid validation of that suggestion.

2. CODE ABSTRACTION

Code abstraction is the replacement of a multiple occurring code fragment by a single copy. The latter forms the body of a new procedure, and each original occurrence of the fragment is replaced by a call to that procedure. This technique can be seen as the inverse of inlining.

In general, code abstraction techniques consist of three tightly connected phases:

1. First of all, multiple occurring code fragments need to be detected. While one can chose to detect only truly identical code fragments, most detection algorithms are able to detect functionally equivalent code fragments that are not identical. The most complex algorithms even detect fragments that are not functionally equivalent, but which can be made equivalent by parameterization.
2. For functionally equivalent but non-identical code fragments, it might be necessary to apply transformations that enable the abstraction of the fragments. The most common such transformations are register renaming, code rescheduling and parameterization.
3. Once abstractable fragments are detected or created, they must actually be abstracted.

In this paper, we do not discuss the detection phase of abstraction techniques. In fact we never delve into the technical details of any specific abstraction technique. For those interested in detailed discussions, a number of references can be found in Section 5. In this paper we focus on the conceptual program transformations that are involved in code abstraction, and their effects on program behavior.

Note that we so far have only spoken about code fragments, and not, e.g., about code sequences. The discussion in this section is mostly orthogonal to the type of the abstracted code fragment. Examples of fragments might be basic blocks, subblock instruction sequences, groups of basic blocks, procedures, etc.

We consider two basic forms of code abstraction: simple abstraction, that does not involve additional parameters, and parametrized abstraction.

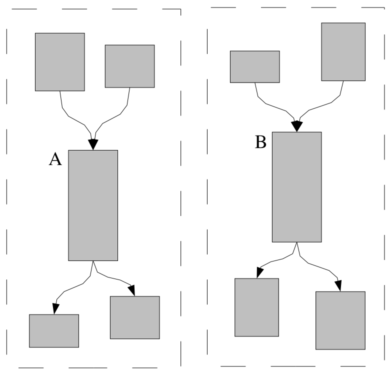
2.1 Simple Abstraction

Simple abstraction applies to code fragments that are functionally equivalent, i.e. code fragments that perform the same computations. How these computations are performed may differ however: the order of the computations might be different, source and destination operands might be different, temporary results might be stored in different locations, etc.

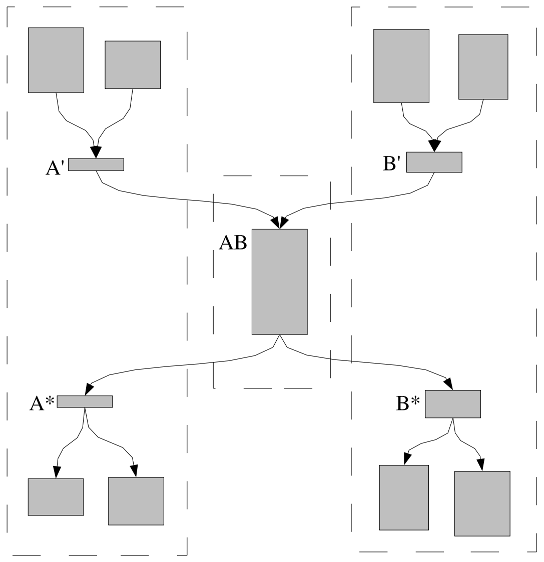
Consider the program fragment in Figure 1(a). To ease the discussion, we assume that all boxes in this figure represent basic blocks, and we assume that the two labeled blocks A and B are functionally equivalent.

In Figure 1(b), A and B have been abstracted into a procedure consisting of the single block AB. In order to do so, we might have to insert four other blocks in the graphs, for the following reasons:

- As the abstracted block AB is put in a new, separate procedure, blocks A' and B' at least contain the call instructions to call the new procedure. To return, block AB ends with a newly inserted return instruction.
- The control flow transfer instructions at the end of A and B are put at the continuation points of the inserted calls, i.e. in A* and B*. These blocks might be empty in case A or B did not end with a control flow transfer, or in case they ended with a return that can be eliminated with tail-call optimization.



(a) Before abstraction



(b) After simple abstraction

Figure 1: Example code fragment for simple abstraction.

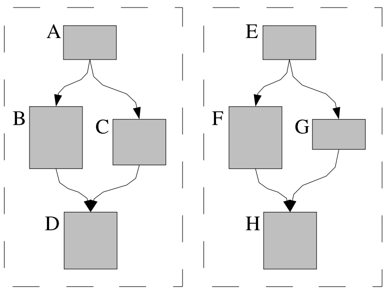
- As AB is executed in two different contexts, it might experience more register pressure than A or B separately. To abstract it, some spill code might need to be inserted in any of the newly created blocks.
- While we assumed that A and B perform equivalent computations, these do not necessarily operate on the same data. So it might be necessary to add additional code in A', B', A* and/or B* to move data consumed or produced in AB into the right operand or location.

Furthermore, as AB is supposed to perform at least the computations of A and B, AB will contain at least as many instructions as the larger of A and B.

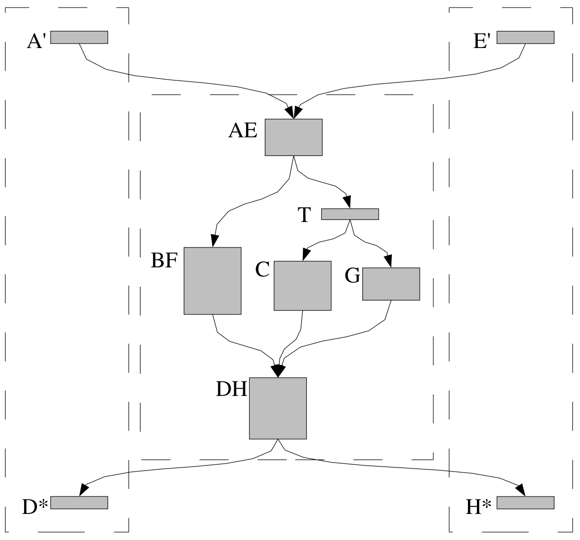
A very similar discussion holds for all other types of code fragments that can be abstracted without parameterization.

2.2 Parametrized Abstraction

Simple abstraction is typically used for program fragments that provide equivalent functionality in different contexts. In some case however, the opposite occurs: almost similar program fragments differ only very locally. An example is depicted in Figure 2(a). Assume the two code fragments



(a) Before abstraction



(b) After parametrized abstraction

Figure 2: Example code fragment for parametrized abstraction.

have the same structure, and all the basic blocks are pairwise equal, except for blocks `C` and `G`.

In such a case, we could apply simple abstraction on all three pairs of identical blocks. That would result in a high run-time overhead however.

Another option is parametrized abstraction, which is depicted in Figure 2(b). The two code fragments are merged and abstracted into a new procedure that takes an additional parameter. This parameter is set just prior to the calls that replace the original fragments (in blocks `A'` and `E'`). The parameter is tested in a newly inserted block (`T`) in the merged procedure to decide which of the alternatives (`C` or `G`) has to be executed.

Apart from setting and testing the parameter, other additional operations that might be inserted to perform parametrized abstraction are of the same nature as those inserted for simple abstraction.

3. EFFECTS ON PERFORMANCE

In this section, performance-related side-effects of code abstraction are discussed.

3.1 Effects on Dynamic Instruction Counts

Following the discussion in section 2, it is clear that more instructions will be executed when code has been abstracted.

Looking back at Figure 1, the procedure calls in blocks `A'` and `B'` are an unavoidable run-time overhead, as is the return instruction in `AB` (except when tail-call optimization is possible). Other possible run-time overhead, depending on the details of the abstraction techniques used, are all other instructions in `AB`, `A'`, `B'`, `A*` and `B*` that were inserted in order to enable the abstraction of similar code fragments.

3.2 Effects on Cache Behavior

While the number of executed instructions can only increase as a result of code abstraction, the effects on instruction cache behavior are not that straightforward.

To get some insight into the possible effects on instruction cache behavior, we consider the effect on the size of the hot code in a program. This is the frequently executed code. The link between the two is as follows: if more code is hot, cache pressure will be higher, and cache behavior will degrade if the cache is too small.

Consider the abstraction example of Figure 1 again. If blocks `A` and `B` are both hot, all the named blocks in Figure 1(b) are hot, and the number of hot instructions has decreased after abstraction. If only one of `A` and `B` are hot however, the hot code size has increased. And in the case no blocks are hot, the hot code size stays zero. It is clear that the hot code size, and accordingly the cache pressure, can either increase or decrease, depending on the execution counts of the code fragments abstracted.

Moreover, when considering hot code size in the decision to include or exclude some fragment from a group of identical fragments to be abstracted, one clearly needs to consider the execution counts of all involved code fragments, and not just the execution count of the fragment under consideration. In the example of Figure 1, whether the abstraction of block `A` will increase the cache pressure or not depends on the execution frequency of block `B`.

It is obvious that taking the precise effect on hot code size into account to guide the abstraction decision process will complicate that process significantly. And this will only get worse if more accurate and more subtle measures than the hot code size are taken into account to minimize the number of instruction cache misses.

Code abstraction can also influence instruction cache behavior through its indirect effects on code layout. Abstracted code is connected to more than one (calling) context in the whole-program control flow graph. It replaces identical code fragments that were connected to a single context in the graph. In the latter case, the different occurrences and their contexts are usually layed out in memory to optimize spatial locality. This is much more difficult after abstraction has taken place and as a result, depending on the code layout algorithm used, the number of instruction cache misses can increase significantly.

3.3 Effects on Branch Prediction

It is obvious that all forms of abstraction come with additional control flow transfers. While most of them, such as the procedure calls and returns, are ideally suited for branch prediction, all of them occupy space in the branch prediction tables. As a result of the increasing pressure on these tables, performance can degrade. This effect will be most outspoken for the return address stack predictions: the number of return addresses that can be stored on this stack is often very small.

On the other hand, conditional branches in procedures that are abstracted with or without parameterization, are merged into a single conditional branch, thus decreasing the number of branches that need to be stored in the prediction tables. Of course, the merged branches might not be as easily predictable as their original occurrences.

The influence on branch prediction is therefore not easily determinable.

3.4 Effects on Code Schedule

In general, code abstraction will introduce relatively more control flow transfers than other operations. As a result, the average size of basic blocks decreases, and it becomes more difficult to schedule the code efficiently. As a result, pipeline usage will be worse on RISC and CISC processors, and instruction slots will be less filled on VLIW processors.

Other effects can play as well, such as the grouping of certain kinds of instructions. A typical example of this is the factoring of procedure prologues/epilogues. These code sequences typically consist of a sequence of register stores/restores, to spill callee-saved registers to the stack. Such sequences are excellent candidates for abstraction: they occur frequently and are cheap to detect, as they occur at easily identifiable locations: procedure entry and return points.

In compiler-generated instruction schedules, the prologue and epilogue will be scheduled in between the code of the procedure body. In order to abstract the prologues/epilogues, they need to be separated from the procedure bodies however. This results in abstracted prologues/epilogues that consist of store/load instruction sequences that will almost never lead to efficient instruction schedules.

4. EXPERIMENTAL EVALUATION

To quantify the side-effects described in the previous section, we've compiled the whole SPECint2000 benchmark suite [20] and three additional C++ programs with the vendor-supplied compilers for the Alpha Tru64 Unix 5.1 platform, and we've compacted all binaries with Squeeze++.

The reason we added the C++ programs to the suite is that C++ programs are often better candidates for code abstraction. As discussed and evaluated in [9], they contain a lot more multiple occurring code fragments, especially when a lot of templates are used. The SPECint2000 programs (who's name begins with a number) are all C programs, except `252.eon` which is a C++ program as well. The other C++ programs are `lcom` (a hardware description language compiler), `fpt` (a mixed C/C++ program that parallelizes Fortran and translates it into High Performance Fortran code) and `LyX` (a WYSIWYG word processor). `lcom` uses no C++ templates at all. `252.eon` and `fpt` use a rather small amount of templates, and the `LyX` code consists for the most part of template code.

While these benchmarks are not typical for the embedded world, we think there are some good reasons for using them. First of all, we want to use benchmarks that resemble future embedded applications. Today it is very hard to find embedded applications written in C++. Yet one of the ultimate goals of program compaction research is to enable the use of higher level programming languages for embedded systems. Therefore we included the C++ benchmarks.

Also, we opted for the SPECint2000 benchmark suite, rather than, e.g., MediaBench or MiBench programs, because the SPECint2000 programs come with well-studied training and reference input data sets. These are engineered to avoid tainting evaluation results with the use of profile input data that is either not at all representative of, or too resembling to the input data used for the actual measurements. The SPECint2000 input sets incorporate the fact that it is not always possible to guarantee the representativeness of profile data.

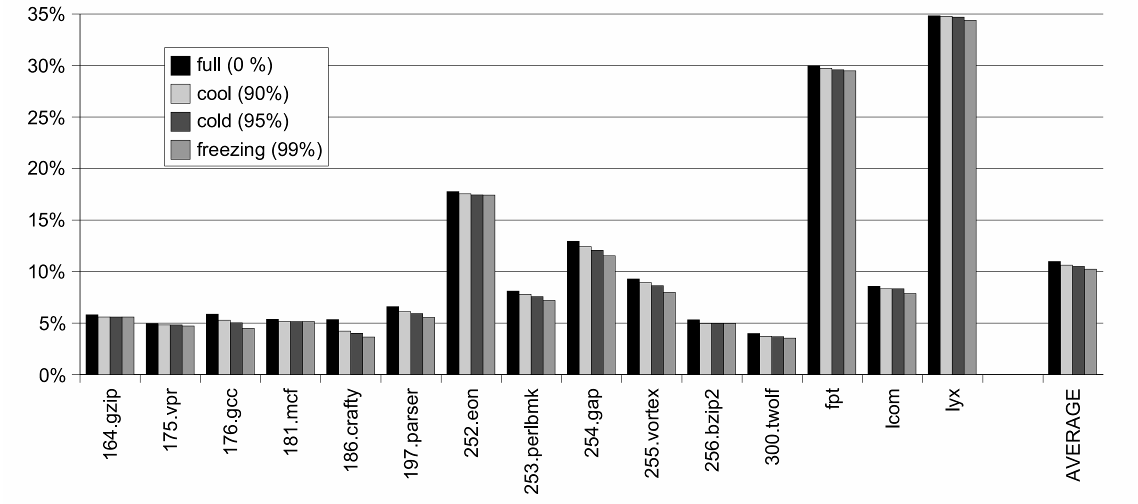
4.1 Squeeze++

The code abstraction techniques whose side-effects are studied in this paper have been implemented in Squeeze++, a link-time binary rewriter aimed at program compaction. The compaction techniques implemented in Squeeze++ are two-fold:

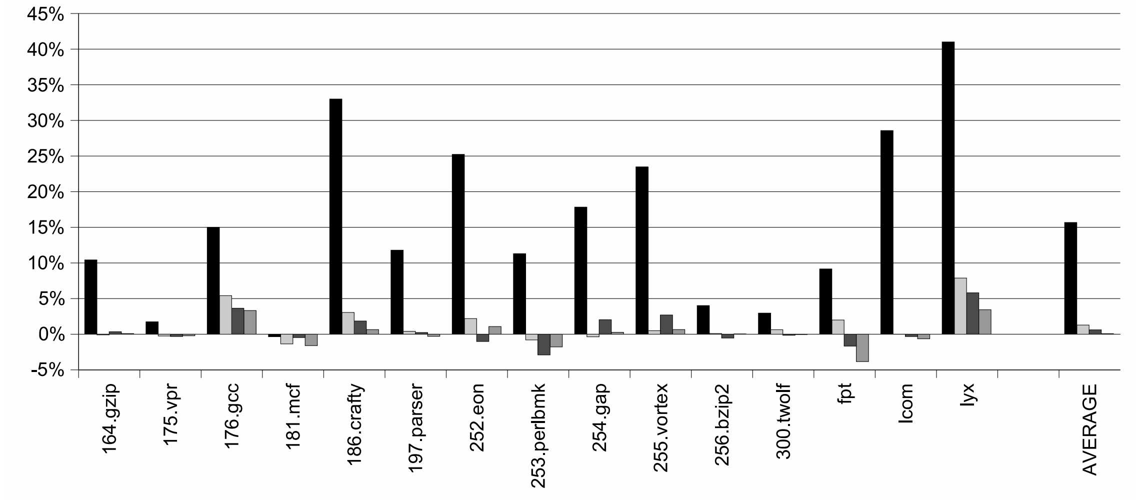
1. Whole-program analyses and optimizations, including liveness analysis, constant propagation, useless and unreachable code elimination, dead data elimination, profile-guided code layout, etc. [10, 11].
2. A range of code abstraction techniques [9] on different types of program fragments. The most important types of abstracted program fragments are:
    - *whole procedures* — Of two or more identical procedures, all but one are eliminated. This frequently occurs for template instantiations, because different instances at the source code can result in identical instances at the assembly level. Often however, procedures will be almost identical, with very local differences only. In such cases, Squeeze++ tries to abstract the procedures using parameterization.
    - *whole basic blocks* — Identical and functionally equivalent whole basic blocks are abstracted. Register renaming and adding spill code are the only transformations applied in order to create abstractable blocks. Renaming registers is done by inserting the necessary copy operations before and after the blocks to be abstracted. In Figure 1(b), such copy instructions would show up in blocks `A'`, `B'`, `A*` and `B*`. Spill code is only inserted in those same blocks, and it must be said that registers are not spilled to the stack, but rather to otherwise unused registers. As such, this spill code consists of copy operations as well. A copy elimination optimization phase follows the abstraction of basic blocks in Squeeze++ to eliminate some of the inserted copy operations.
    - *procedure prologues and epilogues* — As discussed in section 3.4, procedure prologues and epilogues are good candidates for abstraction.
    - *subblock instruction sequences* — General subblock instruction sequences are abstracted as well. As the search space for general instruction sequences is potentially extremely large, their abstraction in Squeeze++ is limited to mostly identical sequences. No rescheduling is tried, and register renaming is very limited.

More details on these techniques can be found in [9].

Given the possible negative side-effects of code abstraction on performance, we adapted Squeeze++ to limit all



(a) code size reduction



(b) slowdown

Figure 3: Code size reductions and program slowdown obtained with full abstraction and profile-guided abstraction.

code abstraction techniques that involve run-time overhead to cold code only.[1] Using different options for Squeeze++, five versions of the binaries were generated:

- No abstraction techniques at all were applied for the basic versions of the binaries used for comparison.
- In *full* mode, all abstraction techniques are applied, without taking profile information into account.
- Three versions are produced with the profile-guided mode, in which basic block counts are used to differentiate between hot and cold code and for which three different hot/cold thresholds were used. A threshold of 90% means that the most frequently executed basic blocks that account for 90% of the dynamic instruction count are considered hot. These blocks are excluded from abstraction. Using different thresholds of 90, 95, and 99%, we had Squeeze++ generate three different versions of the programs (named *cool*, *cold*, and *freezing*).

[1]In this mode, only the abstraction of whole identical procedures is applied on hot code.

The fact that all binaries are generated with Squeeze++ assures that the same compiler back-end is used (same scheduler, same code layout algorithm, etc.) and thus assures a fair comparison between different versions of the binaries.

The basic block counts were obtained using the training data sets of the SPECint2000 benchmark and small input files for the additional C++ benchmarks. These input data differ considerably from the input data we used to measure the effects on performance. For those measurements, we used the reference data inputs of the SPECint2000 benchmarks, and larger input files for the C++ benchmarks.

4.2 Code Size Reductions

In Figure 3(a) we have depicted the code size reductions we obtained with full and profile-guided abstraction. In general, the obtained code size reductions are modest. It is only when templates are used in C++ programs that really high reductions are obtained. On average the size of our benchmark programs is reduced with about 10%. For a lot of C programs, this is merely 5%.

The importance of the bars in Figure 3(a) lays in the fact that profile-guided abstraction results in almost the same code size reduction as full abstraction. There are two reasons for this. The main reason is that according to the conventional wisdom of the 10/90 rule, a very small fraction of the code is usually responsible for a high fraction of the dynamic instruction count.

On top of that comes a more subtle effect, related to the trade-off between optimization and code abstraction. Optimization is closely related to specialization, and the more some code fragment is specialized for its (important) execution contexts, the less likely will other identical or functionally equivalent code fragments be found. Programmers optimize the hot code in their programs. Whereas they might be using some generic template container class from the C++ Standard Template Library to store cold data, they will often use customized, self-written container classes for hot data. Likewise, hot loop bodies are less likely to contain procedure calls, as programmers try to avoid the performance degradation they cause.

As a result of these algorithmic and source code level optimizations by the programmer, relatively fewer abstractable code fragments are found in hot code than in cold code.

4.3 Execution Speed

All generated binaries were executed on a lightly loaded dual 667 MHz Alpha 21264 EV67 machine running Compaq Tru64 Unix 5.1. The 4-way superscalar processors each have a split four-way associative L1 data and instruction cache of 64KB and a unified L2 cache of 2MB. The main memory is 2.5 GB large.

The execution slowdown caused by full and profile-guided abstraction is shown in Figure 3(b). Full code abstraction results in significant slowdowns. On average, it is about 15%. For some programs however, it is more than 30%.

The slowdown caused by profile-guided abstraction is much lower however. On average, it is less than 1% and the maximal slowdown observed is around 8% for a threshold of 90%, and as small as 3% when only freezing code is abstracted.

Note that for some benchmarks, profile-guided code abstraction seems to result in a speed-up instead of a slowdown, which is at first sight counter-intuitive. These observed speedups relate to non-determinism in the quality of the generated code schedules. On the Alpha processor we used for our experiments, the execution time of an instruction sequence depends on its alignment. As Squeeze++ is a compaction tool, it normally does not insert no-ops to optimize code alignment. The alignment of the hot code therefore is not very deterministic, and execution times can significantly vary, even if no really hot code fragments are abstracted. While we have adapted Squeeze++ to insert a very limited amount of no-ops to avoid this non-determinism, we have not been able to avoid it completely. Depending on the heuristics used to insert no-ops, there were always some benchmarks for which this effect occurred.

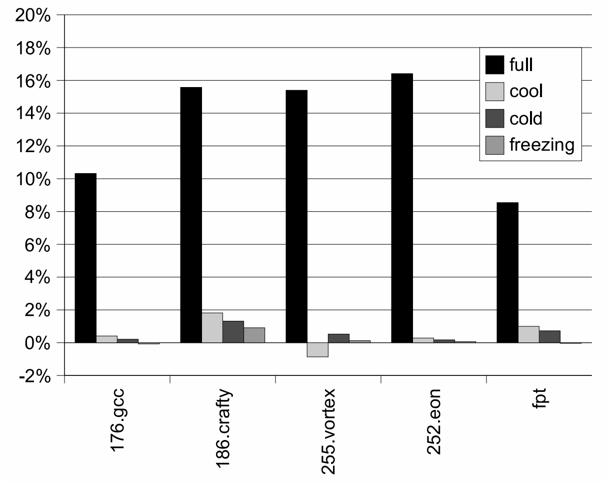


Figure 4: Increase of the dynamic instruction count resulting from code abstraction.

4.4 Dynamic Instruction Counts and Code Schedules

Figure 4 depicts dynamic instruction counts relative to the counts of the unabstracted program versions. These instruction counts were measured with the SimpleScalar simulation toolset [4]. Because of the extremely long simulation time, only part of the reference input data was used for the SPEC programs. Still, the simulations range from 4G instructions for fpt to 132G instructions for 252.eon.

While the dynamic instruction counts for the fully abstracted programs show a strong correlation with the slowdowns observed, the slowdowns are much higher than the increases in executed instructions.

This is caused by the type of the instructions that are inserted during code abstraction. These additional instructions are mostly branches, which causes the number of instructions per basic block to decrease significantly. This is shown in Figure 5. For each benchmark, the basic block size when no abstraction is applied is shown between brackets. The basic block size is decreased with up to 28%, which significantly impacts both the compiler and the processor. The compiler is hampered to generate a good code schedule, as it cannot re-order instructions easily across control transfers. The processor suffers from additional control stalls and has a harder time to exploit instruction level parallelism, as the scheduler already did a worse job.

The strong increase in branch instructions alone does not fully explain the difference between the increase in execution time and the increase in dynamic instruction count. In the next sections we show that code abstraction also leads to an unproportionally large increase in instruction cache misses and branch mispredictions.

Figures 4 and 5 confirm that limiting the code abstraction to cold code can avoid most of the overhead that comes with code abstraction. As with some of the execution measurements, the instruction count decrease observed for 255.vortex results from the insertion and execution of no-ops to align the hottest code.

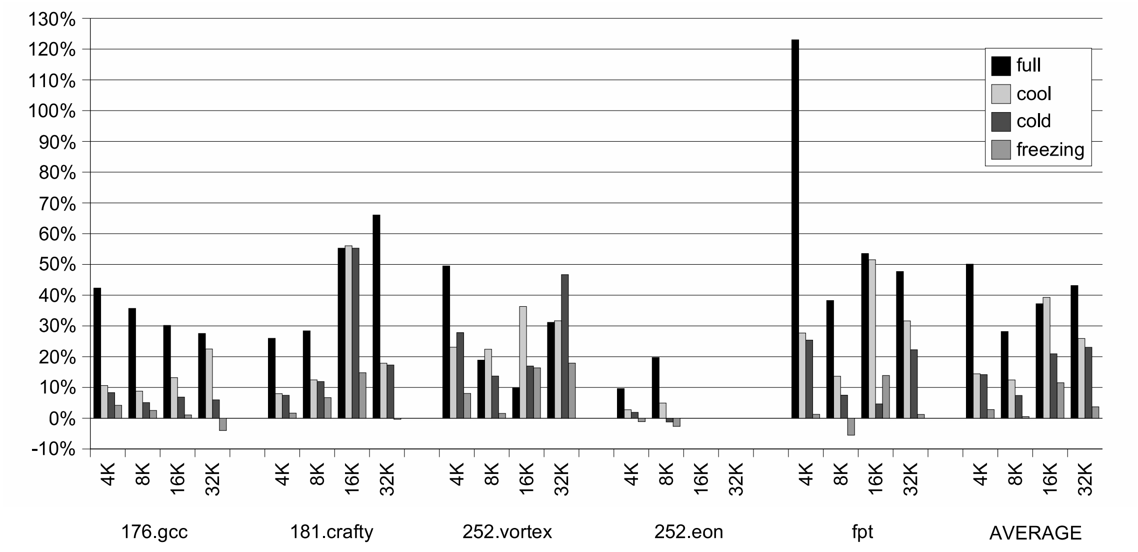


Figure 6: Increase in the number of instruction cache misses for 5 benchmarks.

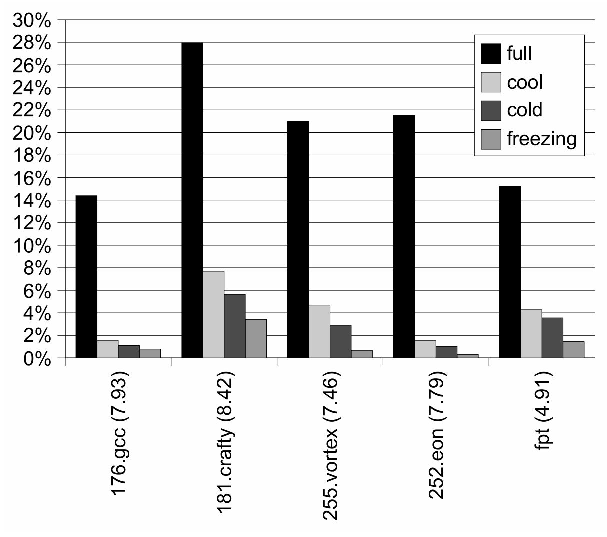


Figure 5: Reduction in the number of instructions per basic block resulting from code abstraction.

4.5 Instruction Cache Behavior

With the SimpleScalar toolset we also measured the number of level 1 instruction cache misses for all versions of five benchmarks. To get representative numbers that do not depend on any single cache organization, we have simulated direct-mapped, 2-way and 4-way set associative caches of different sizes. Because of space concerns however, Figure 6 only depicts the increase in the number of instruction cache misses for 2-way set-associative caches.[2] For other configurations, the results were along similar lines. In all simulations the cache line width was set to 64 bytes. Also, it is important to note that we used a close variant on the Pettis and Hansen profile-guided algorithm to determine code layout [19]. This, together with the fact that we checked the results for numerous cache configurations, guarantees that the presented cache miss numbers are not the result of random code layout or cache configuration anomalies, as discussed in [3].

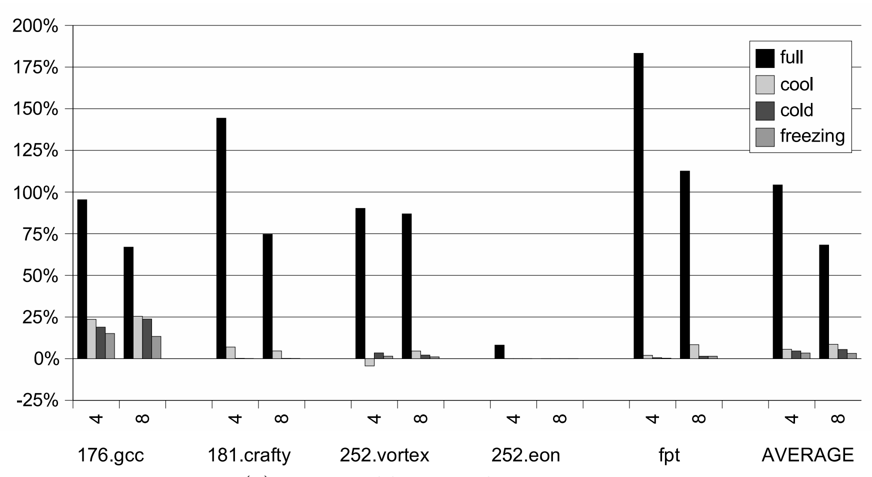
[2]For caches of 16K and larger, the number of instruction caches misses for 252.eon was too small to be significant. We have therefore not included those numbers in the chart.

Note that all depicted increases in instruction cache misses for a cache configuration are relative to the number of misses for the program without abstraction executed on that same configuration. In other words, the depicted results do not at all allow to compare the performance of the different cache configurations.

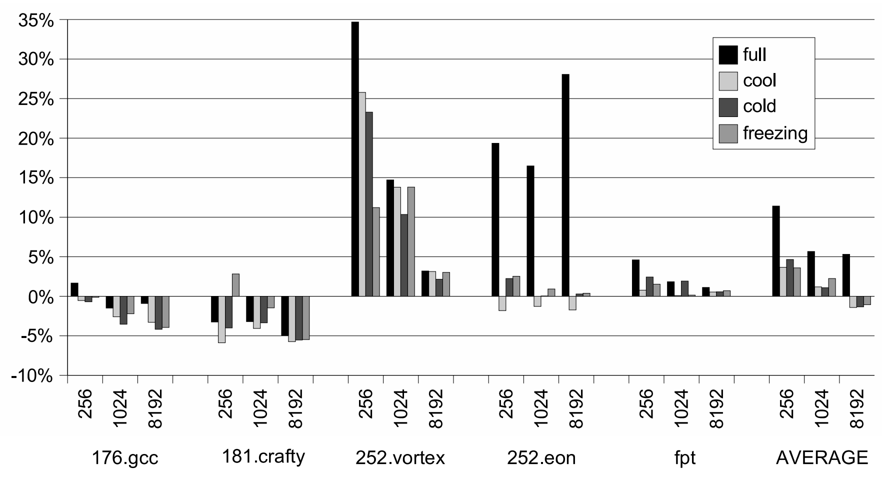
As can be seen from Figure 6, code abstraction can result in a significantly increased number of instruction cache misses. The main reason is the increase of the hot code size when hot code is abstracted together with cold code. For very small caches, most of this can be avoided by limiting the code abstraction to cold code only. The one important result from this graph is that, unlike what we've seen for instruction counts or execution speeds, the move to the "freezing" version seems to be worthwhile not only for some programs, but also on average.

4.6 Branch Prediction

In order to quantify the effect of abstraction on return address prediction, we have simulated five benchmarks on architectures with return address stacks (RAS) of 4, 8 and 12 elements. For all of these benchmarks, the number of misses with 12-element stacks was insignificant. Therefore Figure 7(a) only shows the increase in the number of misses for tables with 4 or 8 elements. As can be seen, return address prediction suffers heavily from full abstraction, with increases in the number of misses of up to 180%. As the stack becomes larger, the negative effect of code abstraction diminishes, as the stack is large enough to also predict the additional returns. Note that for 252.eon, even the very small stack suffices to predict almost all the return addresses. The reason is that all versions of 252.eon show very flat



(a) return address prediction misses



(b) branch direction prediction misses

Figure 7: Increase in the number of missed predictions resulting from full abstraction and profile-guided abstraction.

procedure stack traces. For the other benchmarks, most of the degradation can be avoided by limiting the abstraction to cold code only however.

The influence on conditional branch prediction is quantified in Figure 7(b). Again, to get results that are not tied to one particular organization, we have simulated hybrid branch predictors consisting of 128, 1024 and 8192-entry bimodal and meta-predictors and 256, 2048 and 16384-entry gshare predictors. Conditional branch prediction proves not to be as sensitive to code abstraction as the RAS, and also not as straightforward. A first reason for the latter is that, due to the variations in code layout, correlating branches are mapped to the same entries in the predictor when using one code layout (i.e. one abstraction threshold) but not in another. This is the same cause of small variations in the instruction cache misses. A second reason is that sometimes the branches that are abstracted together are correlated and sometimes they are not. In 252.vortex they were not. The third reason is that the branches inserted for parameterization can also be or not be well predictable. In 252.eon and fpt, a relatively high number of conditional branches was inserted for parameterization. For fpt, these proved to be very well predictable, but for 252.eon this is clearly not the case. In cases such as 252.eon and 255.vortex, limiting the abstraction to cold code again severely limits the experienced performance degradation.

Finally, we want to note that there is no need to study the branch target buffer miss rates. As all calls to abstracted procedures are encoded as direct calls, code abstraction does not impact the number of indirect, unconditional non-RAS branches.

5. RELATED WORK

5.1 Code Abstraction

Most of the previous work on code abstraction to yield smaller executables treats an executable program as a simple linear sequence of instructions [2, 7, 13]. They use suffix trees to identify repeated instructions in the program and abstract them into procedures. The size reductions they report are modest, averaging about 4–7%.

In contrast, our previous work on code abstraction [9, 11] works on control flow graphs, and it considers different granularities of program fragments, such as instruction sequences, basic blocks or procedures.

Chen *et al.* [5] study the compaction of single-entry multiple-exit regions. Instead of using abstraction, they use tail-merging and parameterization to avoid run-time overhead. No run-time overhead measurements are presented however, and neither are code size reductions for whole programs.

Clausen et al. [6] applied minor modifications to the Java Virtual Machine to allow it to decode macros that combine frequently recurring bytecode instruction sequences. They report code size reductions of 15% on average.

Fraser and Proebsting [14] look for repeated patterns in the intermediate program representation used by the compiler. Frequently occurring so called super-operators are detected and used to extend an interpretable code, for which the program is compiled and an interpreter is generated. They report an average code size reduction of 50%, albeit with an undesirable large impact on execution speed. Evans and Fraser [12] further propose compact encodings for interpreted programs, based on transformations of the grammar of the interpreted language.

In a totally different computer science field, that of software engineering, code duplication detection has been studied to measure (and improve) the quality of software. Komondoor and Horwitz [16] and Krinke [17] describe slice-based approaches to detect duplicated code fragments of all possible kinds and shapes. These fragments are not necessarily candidates for abstraction or tail-merging however.

5.2 Relation with Inlining

Code abstraction is the inverse operation of inlining, and as such, it is sometimes called outlining. It is therefore no surprise that there are similarities between the findings of this paper and the conclusions of research into inlining.

Possible advantages of inlining include the elimination of procedure calls and code to implement calling conventions, the possible optimization of the inlined callee in the context of the caller, and the optimization of the caller around the inlined callee. Inlining, because of the duplication of code, can also result in more opportunities to exploit spatial locality during code layout. As a result, inlining can result in significant speedups [1].

These advantages correspond to the most important disadvantages of code abstraction. Code abstraction involves additional control flow, in particular procedure calls and returns. While abstracted procedures do not need to adhere to calling conventions (since their callers are all known at abstraction time), the actual abstraction most often involves the insertion of copy operations or parameter settings to allow the actual abstraction, as discussed in section 2. Also, the optimization of inlined procedure bodies is the inverse of the insertion of conditional branches during parameterization.

The main drawback of inlining, when applied blindly, is code growth because of the code duplication. If the result of inlining is that the size of the working set exceeds the size of the instruction cache during program execution, performance can suffer. Therefore most research into inlining focuses on maximizing the inlining benefits under limited code growth constraints [15]. For profile-guided techniques [18], that only inline procedures at frequently executed call-sites, limiting the code growth is equivalent to limiting the growth of the hot code.

Sometimes however, because of the optimizations that inlining allows, inlining can actually reduce the (hot) code size [8]. This is trivially so when procedures with only one (hot) call-site are inlined, or when the body of an inlined procedure is smaller than the code needed for implementing the call and return. While these trivial case have no corresponding case for code abstraction, we similarly noticed that code abstraction, if applied blindly, can result in an increased hot code size.

Whereas most inlining techniques try to balance between code growth and program optimization, we have opted to completely abandon the abstraction of hot code in our profile-guided approach. Given the disadvantages of abstracting hot code as discussed in sections 4.4, and the fact that the results in section 4.2 show that we should not expect significant code size reductions of abstracting hot code, we believe there is no need to try more complex schemes.

6. CONCLUSIONS

We have shown that blindly applying code abstraction can reduce program sizes significantly. Often however, a high price is payed on almost all performance criteria. We quantified this price for execution time (15% average slowdown), instruction cache misses (on average 30-50% more misses), instruction counts (8-17% increases), code schedule quality (up to 28% less instructions per basic block) and branch prediction (on average 2 times more return address mispredictions for small return address stacks).

Results were also presented that prove that having the code abstraction guided by a very simple form of profile information, namely basic block counts, suffices to obtain almost all the code size reduction benefits of code abstraction, yet experience almost none of its disadvantages.

While the exact numerical results depend on the target platform and the specific details of the used abstraction algorithms, this study gives a very good indication of the performance degradation one can expect from code abstraction, and more importantly, how it can easily be avoided.

Acknowledgement

Bjorn De Sutter, a post-doc research Fellow of the Fund for Scientific Research - Flanders (FWO), is grateful for their support. Hans Vandierendonck and Bruno De Bus are supported by the Flemish Institute for the Promotion of Scientific-Technological Research in the Industry (IWT).

7. REFERENCES

[1] A. Ayers, R. Schooler, and R. Gottlieb. Aggressive inlining. In *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 134–145, 1997.

[2] B. S. Baker and U. Manber. Deducing similarities in Java sources from bytecodes. In *USENIX Annual Technical Conference*, pages 179–190, June 1998.

[3] J. P. Bradford and R. Quong. An empirical study on how program layout affects cache miss rates. *ACM SIGMETRICS Performance Evaluation Review*, 27(3):28–42, 1999.

[4] D. Burger, T. M. Austin, and S. Bennett. Evaluating future microprocessors: The SimpleScalar tool set. Technical report, Computer Sciences Department, University of Wisconsin-Madison, July 1996.

[5] W.-K. Chen, R. Gupta, and B. Li. Code compaction of matching single-entry multiple-exit regions. In *Proceedings of the the 10th Annual International Static Analysis Symposium*, June 2003. To appear.

[6] L. Clausen, U. Schultz, C. Consel, and G. Muller. Java bytecode compression for low-end embedded systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(3):471–489, 2000.

[7] K. Cooper and N. McIntosh. Enhanced code compression for embedded RISC processors. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 139–149, 1999.

[8] K. D. Cooper, M. W. Hall, and L. Torczon. Unexpected side effects of inline substitution: a case study. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(1):22–32, 1992.

[9] B. De Sutter, B. De Bus, and K. De Bosschere. Sifting out the mud: Low level c++ code reuse. In *Proceedings of the 2002 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 275–291, 2002.

[10] B. De Sutter, B. De Bus, K. De Bosschere, and S. Debray. Combining global code and data compaction. In *Proceedings of the 2001 ACM SIGPLAN Workshop on languages, compilers and tools for embedded systems (LCTES)*, pages 29–38, 2001.

[11] S. Debray, W. Evans, R. Muth, and B. De Sutter. Compiler techniques for code compaction. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(2):378–415, 2000.

[12] W. Evans and C. Fraser. Bytecode compression via profiled grammar rewriting. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 148–155, 2001.

[13] C. Fraser, E. Myers, and A. Wendt. Analyzing and compressing assembly code. In *Proceedings of the 1984 ACM Symposium on Compiler Construction*, pages 117–121, 1984.

[14] C. Fraser and T. Proebsting. Custom instruction sets for code compression. http://research.microsoft.com/~toddpro, 1995.

[15] O. Kaser and C. Ramakrishnan. Evaluating inlining techniques. *Computer Languages*, 24:55–72, 1998.

[16] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *Proceedings of the 8th Static Analysis Symposium (SAS)*, 2001.

[17] J. Krinke. Identifying similar code with program dependence graphs. In *Proceedings of the 8th Working Conference on Reverse Engineering*, pages 301–309, 2001.

[18] R. Leupers and P. Marwedel. Function inlining under code size constraints for embedded processors. In *Proceedings of the 1999 IEEE/ACM International Conference on Computer-aided Design*, pages 253–256, 1999.

[19] K. Pettis and R. Hansen. Profile-guided code positioning. In *Proceedings of the 1995 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 16–27, 1995.

[20] http://www.spec.org.