Retargetable Static Timing Analysis for Embedded Software

Kaiyu Chen
Department of Electrical Engineering
Princeton University
Princeton, NJ 08544
Kchen@ee.princeton.edu

Sharad Malik
Department of Electrical Engineering
Princeton University
Princeton, NJ 08544
Sharad@ee.princeton.edu

David I. August
Department of Computer Science
Princeton University
Princeton, NJ 08544
August@cs.princeton.edu

ABSTRACT

This paper presents a novel approach for retargetable static software timing analysis. Specifically, we target the problem of determining bounds on the execution time of a program on modern processors, *and solve this problem in a retargetable software development environment.* Another contribution of this paper is the modeling of important features in contemporary architectures, such as branch prediction, predication, and instruction pre-fetching, which have great impact on system performance, and have been rarely handled thus far. These ideas allow to build a timing analysis tool that is efficient, accurate, modular and retargetable. We present preliminary results for sample embedded programs to demonstrate the applicability of the proposed approach.

1. INTRODUCTION

Static timing analysis is essential in real-time systems development, where the schedulability analysis [1] of programs with hard real-time constraints depends on the estimated extreme case performance. It is also useful in verification of timing critical systems, hardware/software co-design of embedded systems, and early design space exploration.

Because of its importance, many researchers have studied the problem and proposed various solutions. However, two problems remain largely unresolved. First, many analysis techniques are highly architecture specific, which makes them expensive to migrate to other architectures. Second, many advanced features in modern processors are still difficult to model. Since they have great impact on system performance, it is overly pessimistic to simply ignore their presence.

In this paper we describe a general approach for solving these problems. We address the first difficulty by integrating a machine independent timing analysis scheme with our programmable platform-based software development infrastructure (called MESCAL) [2], which allows automatic retargetability by using a unified machine description language. We address the second one by modeling the effects of several, hitherto largely ignored, important features in modern processors.

To be suitable for static timing analysis, the program must be statically predictable, e.g. it cannot have infinite loops or dynamic function calls. In general, the analysis is performed at task-level, in which we do not consider the effects of preemption or interrupts.

This paper is organized as follows. In Section 2 we survey related work. Section 3 provides a review of a powerful integer linear programming (ILP) based software performance bounding technique, which forms the foundation of our tool. We present our retargetable design methodology in Section 4. In Section 5, we describe how to extend the analysis scheme to model the effects of branch prediction, predication, and instruction pre-fetching. Some preliminary experimental results are presented in Section 6 to demonstrate the applicability of the proposed approach. We conclude our work in section 7.

2. RELATED WORK

Early work in the field of timing analysis relies on measuring the program runtimes on the target machine [3]. However, this is known to be unsafe and inaccurate, since it is often impossible to identify the worst-case scenario. Most recent researches focus on analytical methods, in which the analysis is based on information collected before or at compile time.

In order to get fairly accurate estimation results, both program flow and system resource utilization must be analyzed. Since the number of feasible program execution paths can be exponential, explicit [8] and implicit [10] path enumeration techniques are proposed to identify the worst-case program execution sequence efficiently. Micro-architecture affects program execution time by changing instruction timing and program flow. Commonly used mechanisms in modern processors for the purpose of boosting system performance often introduce uncertainty about dynamic program behavior. In particular, the effects of pipelines [4] and caches [5, 6] have been studied extensively. However, most of these approaches only handle some aspect of the problem.

More recently, there have been researches trying to integrate multiple efforts. In [7] the author proposes an integrated path and timing analysis method based on cycle-level symbolic execution. The work in [8] focus on straight line-code and covers aspects on the assembly instruction level as well as on the programming language level. The authors in [9] present a prototype for worst-case execution time (WCET) analysis system, which is similar to ours, but in their work the level of retargetability is unclear. In general, due to the dependency on built-in-house development tools (e.g. simulator/compiler) or target specific data schema, the portability of most existing works are rather limited.

To our knowledge, there has been little work on modeling of predication and software instruction pre-fetching for timing estimation. The effect of branch prediction is studied in [11]. However, it's based on branch target buffer modeling by considering *all* possible program execution paths.

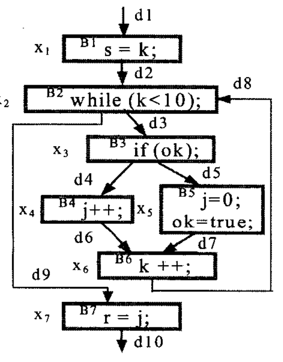
3. INTEGER LINEAR PROGRAMMING (ILP) BASED TIMING ANALYSIS SCHEME

In this section we briefly review the ILP based bounding technique used in Cinderella [10], which is adapted as the core analysis scheme in our work. Cinderella is a timing analysis tool for simple embedded processors. It consists of two components for solving the WCET problem: program flow analysis and micro-architecture modeling.

Program flow analysis characterizes the program flow by two types of linear constraints, namely *structural constraints* and *functionality constraints*. The construction of these constraints is illustrated in Figure 1, where the label on the node/edge denotes its execution/flow count.

```
/* k >= 0 */
s = k;
while (k < 10) {
   if (ok)
      j++;
   else {
      j = 0;
      ok = true;
   }
   k++;
}
r = j;
```

(a) Code



(b) Control flow graph

Figure 1. Code segment example and its CFG.

Structural constraints can be automatically derived from the Control Flow Graph (CFG) based on the flow constraints at each node, i.e. the basic block execution count is equal to the number of times that the controls enters the node (inflow), and is also equal to the number of times that the control exists the node (outflow). For example: $x_1=d_1=d_2$. *Functionality constraints* can be derived by performing data flow analysis, or more effectively specified by the user. They provide two types of logical flow information. The first type gives loop bounds in the program. For example, we can see in the code fragment above that the loop body will be executed between 0 and 10 times each time the loop entered, this information can be specified as: $0x_1 \leq x_3 \leq 10x_1$. The second type specifies other path information that may help tighten the analysis results. For example, due to the correlation between basic blocks B_4 and B_5, the *else* statement can be executed at most once inside the loop, this information can be specified as: $x_5 \leq 1x_1$.

If we assume that the execution time of a basic block B_i of the program is a constant c_i (e.g. each instruction has fixed timing), let x_i be the execution count, then the total execution time of the program can be expressed as $\boldsymbol{\sum c_i x_i}$. Since all the constraints on x_i are expressed as integer linear formulas, we can solve the WCET problem by maximizing the cost function of total execution time under the constraints derived above using ILP techniques.

Nevertheless, in modern architectures instruction timing may vary depending on factors such as operand value and machine state. Micro-architecture modeling is used to account for these effects on dynamic program behavior. For example, a simulator can be used to obtain the cost coefficient of a basic block by modeling pipeline interaction within a known sequence of instructions. Cache behavior is hard to model because it depends potentially on a long trace in history. To tackle this problem Cinderella uses a refined solution based on the same ILP technique. The analysis for a direct mapped Icache is illustrated as follows:

Define an l-block as a contiguous sequence of instructions within the same basic block that are mapped to the same set in the Icache, and thus have identical hit/miss behavior. Denote an l-block inside a basic block as Bi.j. It is associated with two variables $x_{i.j}^{hit}$ and $x_{i.j}^{miss}$, which represent its hit and miss counts. Let $c_{i.j}^{hit}$ and $c_{i.j}^{miss}$ represent the hit and miss execution time of Bi.j respectively, then the new cost function for total execution time can be expressed as: $\boldsymbol{\sum_i \sum_j (c_{i.j}^{hit} x_{i.j}^{hit} + c_{i.j}^{miss} x_{i.j}^{miss})}$. Since the total execution count of Bi.j is equal to that of the basic block containing it, we have: $x_i = x_{i.j}^{hit} + x_{i.j}^{miss}$.

For any two l-blocks mapped to the same cache set, they *conflict* with each other if their address tags are different. Otherwise they are said to be *non-conflicting*. If there are less than one conflicting l-blocks mapped to a cache set, *the sum of their miss counts must be at most 1*. For each cache set containing two or more conflicting l-blocks, a *cache conflict graph (CCG)* is constructed. It contains a start node 's' (representing the start of the program), an end node 'e' (representing the end of the program), and a node '$B_{k.l}$' for every l-block $B_{k.l}$ mapped to the same cache set. There is a directed edge drawn from node $B_{k.l}$ to $B_{m.n}$, if there exists a path in the CFG from basic block B_k to basic block B_m without passing through any other l-blocks of the same cache set. We assign a new variable p(k.l, m.n) to count the number of times that control pass the edge. Then we can derive a new set of linear constraints from the CCG as follows.

Similar to the CFG analysis, for each node $B_{i.j}$ the sum of inflow must be equal to its execution count and the sum of the outflow. Therefore, we have: $x_i = \sum_{u.v} p(u.v,\ i.j) = \sum_{u.v} p(i.j,\ u.v)$, where "u.v" denotes some other node $B_{u.v}$ that is connected with $B_{i.j}$. In particular, for the start node 's', *the sum of outflow is equal to the number of times the program segment is executed.* Note that the constraints link the new variables to those we used in CFG analysis, therefore allow the effect of all parts to be considered in a global way. Besides, according to the definition of the CCG, the number of the cache hits can be determined by: $p(i.j,\ i.j) \leq x_{i.j}^{hit} \leq p(s,\ i.j) + p(i.j, i.j)$. These constraints, together with the structural constraints and functionality constraints, are used to solve the ILP problem for the new cost function. The same technique can be extended to model set associative caches and data caches. More details can be found in [10].

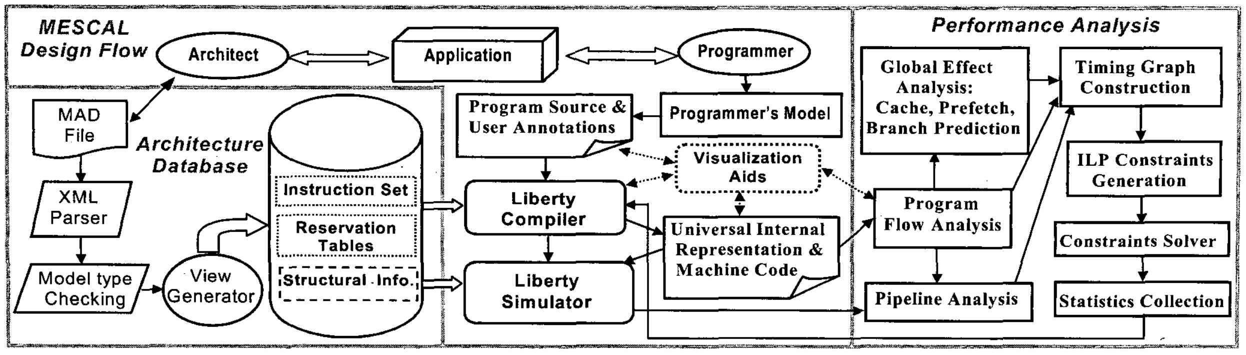


Figure 2. System overview.

4. DESIGN METHODOLOGY

As previously mentioned, both program flow and micro-architecture need to be modeled for accurate timing analysis. In general both depend heavily on target specific information, which varies a lot for different machines. For example, in program flow analysis we need to understand the instruction set architecture, while in micro-architecture modeling we need to know the system resource utilization and their effects on program's dynamic behavior. Traditionally, such information is hard-coded into the development tools, which makes it hard to retarget to different architectures. In this paper we show how we can reduce the overall design efforts and complexity of the problem, through the adoption of an approach that minimizes the overhead in doing so. Figure 2 gives the overview of our approach.

4.1 MESCAL Software Design Environment

The goal of MESCAL [2] project is to provide a programmer's model and software development environment that allows for efficient implementation of an interesting set of applications onto a family of fully-programmable architectures/micro-architectures. As depicted in Figure 2, software development tools involve the compiler, simulator, performance analyzer, debugger, and visualization aids. Despite their functionality division, they are closely related and often share common information about target platform. For example, both the compiler and the simulator need to know the information contained in the reservation table for a VLIW processor. These common dependencies motivate us to use a unified architecture description to drive the whole development system.

There have been a lot of research efforts in the area of machine description, such as MDES used in IMPACT compiler [12]. Our approach is different from traditional ones in that MESCAL Architecture Description (MAD) aims to provide both simulator view and compiler view for the target processor. MAD is a mixed level description that has combined instruction semantics and structural specification, which provides the flexibility to express the variety of architectural features found in modern processors. It is written using XML as syntax. We chose XML because of its flexibility and its popularity in public domain. Currently the description consists of five sections: The **Declaration section** specifies abstract machine parameters. For example: whether interlocking is supported or not, in-order or out-of-order instruction issue, etc. The **Unit section** describes pipeline stages and special function units, such as register files, memory, branch predictor, etc. The **Operand section** corresponds to the layer between the storage units and the operation, where different addressing modes are enumerated. The type of operand can be immediate, register, or composite. The **Operation section** has two elements. *Operation* specifies the attributes of individual instructions, such as instruction format, encoding, and behavior. *Opergroup* groups instruction into instruction sets for the convenience of later reference. The **Connection section** defines two types of connection: *pipe* and *link*. The former specifies the pipeline flow in the architecture and the latter specifies the data flow.

4.2 Implementation Strategy

As described in section 3, our core analysis scheme is machine independent and based on ILP model. However, in Cinderella's case the user still needs to develop a handcrafted backend support (including disassembler and simulator) for each different target processor, which is difficult and inefficient for complicated architectures. In our implementation, the program flow analysis is performed on the compiler universal intermediate representation of the program [16]. In this way we can have a unified interface and processing mechanism for the compiled "program", while the differences between architectures are captured by the machine description and handled by the front end of compiler.

Other information required by flow analysis can be derived from the architecture description. For example, to build the program's flow control graph we need to identify the instructions that may change program flow. We can build a table from the architecture description files to store this kind of ISA information (i.e., the mapping between instructions and their types - conditional jump, function call, return, etc.). The data structures for such records are machine independent and manipulated by the core analysis algorithm. Note that although some information is not explicitly presented in the architecture description, it can be inferred from the associated attributes (e.g. function call has a special register PC as its arguments).

As integrated in the same framework, the tools can benefit from each other. The compiler may provide other useful information, such as the profiling results, to assist the analysis. With the help of visualization aids, the user can add annotation and additional constraints to help tighten the estimation results [10], which can be then used as feedback to help tune the design performance.

The modeling of the micro-architecture consists of two parts. The first models the timing of program's execution in a restricted scope. The simulator can be used for this purpose. It accurately models the effect of the pipeline and gives the cost coefficients of basic blocks. The overhead of crafting a simulator from scratch can be avoided as the simulator is automatically built from the machine description. The second one is the global effect analysis. One example is the cache modeling described in section 3, in which the required information, such as the cache size and set associativity, can also be automatically extracted from the architecture description. Hence the core analysis module is kept machine independent and the analysis framework can be easily retargeted to new architectures.

5. ADVANCED MICRO-ARCHITECTURE MODELING

Similar to cache, there are other features in modern processors that affect program's execution in a global way, such as branch prediction, predication, and instruction pre-fetching. These features dramatically reduce the program execution time and an inability to model them will result in a severely pessimistic analysis. In this section we demonstrate how to extend the analysis scheme to model their effects on program timing.

5.1 Branch Prediction Analysis

To reduce the penalty caused by control hazards in pipeline processors, it is common to predict the outcome of a branch without interrupting the instruction flow. In modern processors, both *static* and *dynamic* branch prediction schemes are employed. The analysis for commonly used static prediction schemes can easily fit into our ILP based solution. The dynamic schemes are much harder to model statically, due to the fact that the prediction results are based on program behavior at runtime. Existing methods rely on static simulation techniques, such as BTB modeling used in [11], to identify worst case branching cost and then combine it with high-level analysis for WCET estimation.

Three commonly used static prediction schemes are: *predict-taken*, *predict-untaken*, and *delayed-branch-with-canceling* [13]. It is the compiler's responsibility to determine the predicted outcome and schedule code appropriately. In many cases the decision is based on examination of program flow behavior. This makes it suitable for static timing analysis. An example is shown in Figure 3.

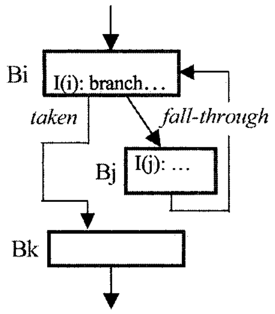


Figure 3. Control flow example.

Suppose instruction I(i) inside the loop is predicted as not taken, which is the case in most execution times. At runtime, the processor will check the result after the branch is resolved. If it is the same as predicted, then no penalty is involved. Otherwise, the instruction I[j] scheduled after the branch instruction is turned into a no-op and the program will continue to execute in the correct direction. To model this effect, we can count the execution time of I[j] into that of the basic block B_i. Then its corresponding term $c_i x_i$ in the original cost function of the program's total execution time would be replaced by: $(c_i^{miss\text{-}predict} x_i^{miss\text{-}predict} + c_i^{correct\text{-}predict} x_i^{correct\text{-}predict})$, where $x_i = x_i^{miss\text{-}predict} + x_i^{correct\text{-}predict}$. Suppose the shown program fragment is executed only once, then there would be only one miss-prediction, which corresponds to: $x_i^{miss\text{-}predict} = 1$. These constraints link the new variables with those we used in program flow analysis and cache analysis, and are sufficient to bound the observed case. We can model similar cases and other schemes using the same approach.

Nevertheless, there are other cases where the branch does not come from loops, and where the compiler depends on profiled information to predict the outcome. In these cases, we can still represent the variances in execution time by linear constraints. The worst-case scenario may be assumed to have the longest execution path (e.g. always predicted wrong), or we can rely on data flow analysis and user annotation for the bounds.

5.2 Predication Analysis

Modern compilers use a more aggressive optimization technique to transform the control flow into predication, and thus eliminate inefficiencies in handling branch instructions. In this approach, most instructions can be tagged with some guarding predicate, a special 1-bit register. The execution of the instruction may be nullified at runtime if the predicate value is FALSE (0). As a result the instruction latency may vary depending on the results of the predicate define instructions. To model this optimization effect, we can partition the instructions within a basic block into smaller groups based on their predicates. Thus the total execution time would be the sum of cost terms of each group of instructions, which is the product of its normal execution time and count plus the product of its nullified execution time and count. In the worst case we may have one cost term for each instruction. An example using IA-64 architecture [14] is shown in figure 4. The default semantics of "cmp.eq" is to set p1 and p2 to complementary values depending on the comparison results. Therefore we can derive that instruction B and instruction D are mutually exclusive and cannot be executed normally at the same time. The dependence among the predicates (e.g. dominance, mutual exclusion, exhaustion, etc.) can be derived from a BDD package [15] generated when the compiler performs the optimization, or provided by user annotations.

```
      cmp.eq  p1, p2 = r5,r4 ;: // cycle 0: instr A
(p1)  ld8       r1 = [r3] ;:      // cycle 1: instr B
      add       r3 = r1, r2 ;:    // cycle ?: instr C
(p2)  ld8       r4 = [r6] ;:      // cycle ?: instr  D
      add       r6 = r4, r5 ;:    // cycle ?: instr E
```

(a) Basic block Xi

$$\begin{aligned} Xi &= X_{iA} = X_{iC} = X_{iE} && \text{//execution count equation} \\ &= X_{iB}^{N} + X_{iB}^{n} && \text{//'N' denotes normal case} \\ &= X_{iD}^{N} + X_{iD}^{n} && \text{//'n' denotes nullified case} \\ X_{iB}^{N} &= X_{iD}^{n} \quad (X_{iB}^{n} = X_{iD}^{N}) \end{aligned}$$

(b) Linear constraints for execution counts

$$\begin{aligned} \text{Total\_execution\_time} = {} & X_{iA} C_{iA} + X_{iC} C_{iC} + X_{iE} C_{iE} \\ & + X_{iB}^{N} C_{iB}^{N} + X_{iB}^{n} C_{iB}^{n} \\ & + X_{iD}^{N} C_{iD}^{N} + X_{iD}^{n} C_{iD}^{n} \end{aligned}$$

(c) Modified cost function

Figure 4. Predication modeling example.

5.3 Instruction Pre-fetching Analysis

Instruction pre-fetching [17] is another important scheme to hide the relatively slow speed of memory system. In software controlled methods, since pre-fetching instructions can change the state of cache, their effects must be modeled in the cache analysis. Based on the observation that, in the case of pre-fetching, the program flow is still characterized by the execution of the pre-fetching l-block, while the cache content is determined by the pre-fetched data, we can modify our cache analysis scheme described earlier to factor in the effect of pre-fetching instructions.

An example using a directed mapped Icache is shown in Figure 5, in which $B_{5.3}$ and $B_{6.1}$ are non-conflicting l-blocks. They both conflict with $B_{4.1}$, which is pre-fetched by $B_{1.1}$. The modified cache conflict graph CCG' that corresponds the cache set they are mapped to is constructed as follows: first we add in the l-block that contains instructions pre-fetching to the same cache set, in this case $B_{1.1}$. The 'pre-fetch' nodes are distinguished from the normal nodes by attached attributes. The edges for the newly added nodes are drawn in the same way as for constructing the CCG. We can then determine if an edge represents a transition that results in a cache miss (denoted as 'C' in the graph, otherwise denoted as 'N') by looking at the cache content of the nodes it connects. Note that for a 'pre-fetch' node, we need to consider its pre-fetched data when we decide the hit/miss property of its connected edge. For example, since $B_{1.1}$ pre-fetches $B_{4.1}$, and $B_{4.1}$ conflict with $B_{5.3}$, the edge from $B_{1.1}$ to $B_{4.1}$ is counted as a hit, while the edge from $B_{1.1}$ to $B_{5.3}$ is counted as miss. Therefore, we can generate the constraints for bounding cache hits of the l-blocks, e.g., $x_{4.1}^{hit} = p(4.1,4.1) + p(1.1,4.1)$. Other types of constraints can be generated as we have described for the CCG, with the modification that when generating the flow related constraints, we need to count the execution of the pre-fetching l-block associated with a 'pre-fetch' node. For example, the constraints for node $B_{1.1}$ would be: $p(s,1.1) = p(1.1,e) + p(1.1,4.1) + p(1.1,5.3) = x_1$. Similarly we can extend the scheme to model other cases and set associative caches, which have been omitted for sake of brevity.

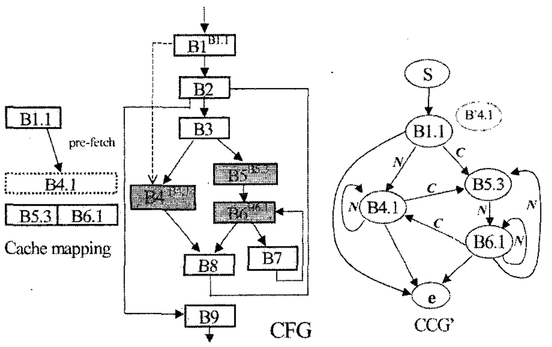


Figure 5. Example of cache analysis with pre-fetching.

6. EXPERIMENTAL RESULTS

As an initial proof-of-concept experiment to verify the applicability of our approach, we have incorporated and tested the core analysis scheme as described in section 3 within the Liberty compiler infrastructure [16]. The target architectures we experiment with belong to the IMPACT EPIC [12] class, which allows effective exploitation of instruction-level parallelism and has configurable aspects such as memory characteristics, register file sizes, number of functional units, and specialized functional units. The architecture configuration from which we obtained the test data are IMPACT EPIC-8G-1BL [12], which is an 8-issue 1-branch-slot VLIW processor that supports general speculation, and EPIC-1G-1BL, which is an 1-issue processor. Table 1 gives a brief description of the programs we tested. The CPU time is measured on a server running Linux with Pentium III 800Mhz CPU and 2 Gig memory.

Table 1. Description of tested programs.

Program Name	Description
Jfdctint	JPEG slow-but-accurate implementation of DCT
Stats	Calculate the sum, mean and variance of two arrays
FFT	1024-point Fast Fourier Transform
Matcnt	Summation of 2 100*100 matrices
Sort	Sort an array using BubbleSort

Table 2 shows the flow analysis results. The cost coefficients of the basic blocks are derived from the instruction scheduling information, which is feasible as the compiler has fully knowledge of pipeline resource usage for VLIW processors. The estimation results are compared with the those obtained from compiler profiling, in which we have to manually set the worst case scenario input data, therefore may not get the *exact* actual bound. The unit of the data listed for WCET is 'cycle'. The last column shows the ILP solver (lp_solve) runtime in 'second'.

Table 2. Experimental results of flow analysis.

(a) Using configuration EPIC-1G-1BL

Program	Estimated WCET	Profiled WCET	Time
Jfdctint	1471	1471	0.00
Stats	151202	151202	0.03
FFT	198272	198272	0.03
Matcnt	491063	481063	0.01
Sort	2626261	2626261	0.02

(b) Using configuration EPIC-8G-1BL

Program	Estimated WCET	Profiled WCET	Time
Jfdctint	1434	1434	0.00
Stats	126113	126113	0.03
FFT	195708	195708	0.03
Matcnt	380620	380620	0.01
Sort	1625752	1625752	0.02

Table 3 shows the Icache analysis results. Comparisons are presented with the results obtained from the Dinero IV cache simulator. The input to Dinero IV is the program's execution trace generated by Liberty emulator. Note that to verify the cache analysis accuracy, we explicitly set the basic block execution counts so that they match the execution trace. The units of the data shown are 'total_access_count / total_miss_count'.

We can see that the estimated results are very close to those given by other standard tools. As analyzed in [10] the ILP technique is efficient compared with other methods. In general, the problem size will grow with the number of variables (corresponding to vertices and edges in the constructed timing graphs), program structure complexity (represented by the generated linear constraints), and architecture configuration (e.g. cache associativity). Furthermore, as indicated in table 3-c, the overhead of static analysis can be much lower than simulation based scheme, especially for programs with long execution trace but unsophisticated structures.

Table 3. Experimental results of cache analysis.

(a) Cache configuration: Size = 512; Line Size = 32; Assoc. = 1

Program Name	Cache Analysis	Dinero IV
Jfdctint	1583 / 38	1583 / 38
Stats	209394 / 2072	209394 / 2072
FFT	167068 / 1023	167068 / 1023
Matcnt	721196 / 39	721196 / 39
Sort	4996587 / 20	4996587 / 20

(b)Cache configuration: Size = 512; Line Size = 16; Assoc. = 2

Program Name	Cache Analysis	Dinero IV
Jfdctint	1583 / 73	1583 / 73
Stats	209394 / 145	209394 / 145
FFT	167068 / 60	167068 / 60
Matcnt	721196 / 76	721196 / 76
Sort	4996587 / 37	4996587 / 37

(c) Measured runtime in experiment (a)

Program Name	Cache Analysis	Dinero IV
Jfdctint	0.06	0.02
Stats	0.19	0.38
FFT	0.08	0.28
Matcnt	0.09	1.19
Sort	0.03	8.25

7. CONCLUSION

In this paper we have described an approach for retargetable static software timing analysis. We propose to integrate a machine independent timing analysis scheme with our programmable platform-based software development environment.

The advantages of this approach are twofold. First, by separating machine dependent and independent parts, and carefully designing the interface, we can make the timing analysis tool retargetable and modular. Second, due to the close interaction, the modules in the infrastructure can benefit from each other. Compared with other methods, in our approach significant design efforts can be leveraged among the modules in the same development infrastructure, including the highly optimizing compiler and automatically built simulator. The core analysis scheme can handle both software/hardware aspects in a uniform ILP framework and remains primarily unchanged. Furthermore, as we have demonstrated, the core analysis scheme can be easily extended to model several of the most important features in contemporary architectures, and thus produce tighter bounds.

8. ACKNOWLEDGEMENT

The authors would like to acknowledge the support from GSRC, and suggestions from all colleagues, in particular Yau-Tsun Steven Li. This work is partly sponsored by Hangseng Overseas Scholarship.

9. REFERENCES

[1] J. Stankovic, M. Spuri, M. Di Natale, and G. Buttazzo. "Implications of classical scheduling results for real-time scheduling". *IEEE Computer,* pages 16--25, June 1995.

[2] K. Keutzer, S. Malik, A. R. Newton, J. Rabaey and A. Sangiovanni-Vincentelli. "System Level Design: Orthogonolization of Concerns and Platform-Based Design". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(12), December 2000.

[3] P. Gopinath and R. Gupta. "Applying compiler techniques to scheduling in Real-time systems". In *Proceedings of 1990 IEEE Real-Time Systems Symposium*, pages 247--256, 1990.

[4] S. Lim, Y. H. Bae, G. Tae Jang, B. Rhee, S. Lyul Min, C. Y. Park, H. Shin, K. Park, S. Moon, C. S. Kim, "An Accurate Worst Case Timing Analysis Technique for RISC Processors," In *Proceedings of the 15th Real-Time Systems Symposium*, 1994.

[5] R. T. White, F. Mueller, C. A. Healy, D. B. Whalley, and M. G. Harmon, "Timing Analysis for Data Caches and Set-Associative Caches". In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, Montreal, Canada, pp. 192—202, June 1997.

[6] F. Mueller. "Timing predictions for multi-level caches". In *ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, pages 29--36, 1997.

[7] T. Lundqvist and P. Stenstrom. "An integrated path and timing analysis method based on cyclelevel symbolic execution". *Journal of Real-Time Systems*, November 1999.

[8] F. Stappert and P. Altenbernd. "Complete Worst-Case Execution Time Analysis of Straight-line Hard Real-Time Programs." *Journal of Systems Architecture*, 46(4), 2000.

[9] J. Engblom, A. Ermedahl, M. Sjodin, J. Gustafsson, and H. Hansson, "Worst-Case Execution-Time Analysis for Embedded Real-Time Systems". *Unpublished technique report*, Department of Computer Systems, Uppsala University, Sweden, 2000.

[10] Y. S. Li, S. Malik and A. Wolfe, "Cache Modeling for Real-Time Software: Beyond Direct Mapped Instruction Caches". In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1996.

[11] A. Colin and I. Puaut. "Worst case execution time analysis for a processor with branch prediction". In *Real-Time Systems*, 18(2-3): 249--274, May 2000.

[12] "The IMPACT Research Group", http://www.crhc.uiuc.edu/IMPACT

[13] J. Hennessy, J. L. Hennessy, D. Goldberg, D. A. Patterson. "Computer Architecture : A Quantitative Approach". Morgan Kaufmann Publishers, 1996.

[14] Itanium^{TM} Processor Microarchitecture Reference. *Technical report*, Intel Corp., August 2000. http://developer.intel.com/design/ia-64/

[15] J. W. Sias, D. I. August, and W. W. Hwu. "Accurate and Efficient Predicate Analysis with Binary Decision Diagrams". In *Proceedings of the 33rd International Symposium on Microrchitecture*, December 2000.

[16] "The Liberty Computer Architecture Research Group", http://liberty.cs.princeton.edu

[17] C. Luk and T. Mowry. "Cooperative Prefetching: Compiler and Hardware Support for Effective Instruction Prefetching in Modern Processors". In *Proceedings of 31st International Symposium on Microarchitecture*, pages 182--193, December 1998.