Static Cache Simulation and its Applications

by

Frank Mueller

Dept. of Computer Science
Florida State University
Tallahassee, FL 32306-4019
e-mail: mueller@cs.fsu.edu
phone: (904) 644-3441

July 12, 1994

A Dissertation submitted to the
Department of Computer Science
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Degree Awarded:
Summer Semester, 1994

Abstract

This work takes a fresh look at the simulation of cache memories. It introduces the technique of *static cache simulation* that statically predicts a large portion of cache references. To efficiently utilize this technique, a method to perform efficient on-the-fly analysis of programs in general is developed and proved correct. This method is combined with static cache simulation for a number of applications. The application of fast instruction cache analysis provides a new framework to evaluate instruction cache memories that outperforms even the fastest techniques published. Static cache simulation is shown to address the issue of predicting cache behavior, contrary to the belief that cache memories introduce unpredictability to real-time systems that cannot be efficiently analyzed. Static cache simulation for instruction caches provides a large degree of predictability for real-time systems. In addition, an architectural modification through bit-encoding is introduced that provides fully predictable caching behavior. Even for regular instruction caches without architectural modifications, tight bounds for the execution time of real-time programs can be derived from the information provided by the static cache simulator. Finally, the debugging of real-time applications can be enhanced by displaying the timing information of the debugged program at breakpoints. The timing information is determined by simulating the instruction cache behavior during program execution and can be used, for example, to detect missed deadlines and locate time-consuming code portions. Overall, the technique of static cache simulation provides a novel approach to analyze cache memories and has been shown to be very efficient for numerous applications.

Acknowledgements

I would like to to express my gratitude to Dr. David Whalley, my major professor, for his guidance, support, patience, and promptness during my work on this dissertation. He was always available to discuss new problems and exchange ideas. I would also like to thank the other committee members, Dr. Ted Baker, Dr. Gregory Riccardi, and Dr. Steve Bellenot, for their kind support. Dr. Baker introduced me to the area of real-time systems, one of the application areas of the dissertation work. His comments on early drafts of the formalizations in this dissertation were most valuable. Dr. Riccardi helped me to organize my thoughts and improve the presentation style of this dissertation. Last but not least, Dr. Bellenot's input on the formalization and proofs of the graph-theoretical aspects were invaluable.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This dissertation addresses the issue of providing a fast framework for cache performance evaluation to determine the number of cache hits and misses during a program execution. Cache performance measurements are commonly used to evaluate new cache designs, *i.e.* to determine the cache configuration that best fits a new processor design. In addition, new compiler optimization techniques are often analyzed with regard to their impact on cache performance.

Furthermore, this dissertation challenges the claim that cache memories introduce unpredictability to execution-time predictions for real-time applications. This common belief has forced real-time designers to predict the worst-case execution time of program assuming that caches are not present. A schedulability analysis based on such simplifying assumptions often results in a gross underutilization of a processor, the selective enabling and disabling of caches for the most critical task, or even the disabling for caches at all times.

This dissertation addresses these issues by a technique to statically simulate a large portion of the caching behavior of programs. The technique, called *static cache simulation*, is formally defined in this dissertation. Furthermore, the technique is shown to provide considerable speed-up over traditional cache performance analysis techniques. It also provides a framework to statically predict a large number of the caching behavior that is shown to produce tight execution time bounds when combined with a timing tool. Each of these issues is addressed in a separate chapter in this dissertation and is supported by measurements.

The approach taken by static cache simulation is quite different from traditional methods. It does not rely on tracing but rather combines compile-time analysis and code instrumentation within the environment of a compiler back-end. The simulator attempts to determine statically whether a given program line will result in a cache hit or miss during program execution. This is achieved by the analysis of both the call graph of the program and control-flow graph for each function. A set of instructions executed in sequence is called a *unique path* if it can be distinguished from all other paths by at least one (unique) control-flow component. To better predict the cache behavior, functions are further distinguished by *function instances* that depend on the call site and call sequence.

During program execution, extensive use of frequency counters suffices for cache simulation when instruction references are statically determined to be always cache hits or always cache misses. For the remaining instruction references, state information is associated with code portions and is updated dynamically. This state information represents a localized view of the cache and is used to determine whether the remaining program lines of a code portion are or are not cached. These localized states are in contrast to a comprehensive global view of the cache state as employed in conventional trace-driven simulation. The total hits and misses can be inferred from the state-dependent frequency counts after running the program.

In summary, the cheaper method (frequency counters) is used when references are statically known, and the remaining references are determined by local states that also impose less execution overhead than one global cache state. The improvements of fast instruction cache analysis using static cache simulation over traditional trace-driven cache simulation are summarized below.

Unique Paths: The code is instrumented at unique paths (UPs) rather than at basic blocks. This reduces the number of *instrumentation points* (also called *measurement points*), *i.e.* the places where instrumentation code is inserted into the regular code generated during program compilation. The set of unique paths is shown to provide a small set of measurement points for on-the-fly analysis methods in general.

Static Cache Simulation: A large percentage of the instruction references are statically identified as always hits and always misses. These references will not have to be simulated at all during program execution.

Function Instances: The static cache simulation is refined by taking the call site of a function invocation into account. Thus, the simulation overhead required at run time is further reduced since the behavior of more instruction references can be statically identified.

Inline Code Instrumentation and Frequency Counters: The remaining instruction references are simulated at run time by inlining short sequences of instrumentation code for each UP rather than calling a tracing routine. The compiler identifies the live registers at the instrumentation point. A set of unused registers is provided for the instrumented code to avoid unnecessary saves and restores. If all registers are used, then some registers will be temporarily spilled around the instrumentation point. The instrumentation consists of incrementing simple frequency counters and state transitions.

Figure 1.1 depicts an overview of the programs and interfaces involved in static cache simulation. The set of source files of a program are translated by a compiler. The compiler

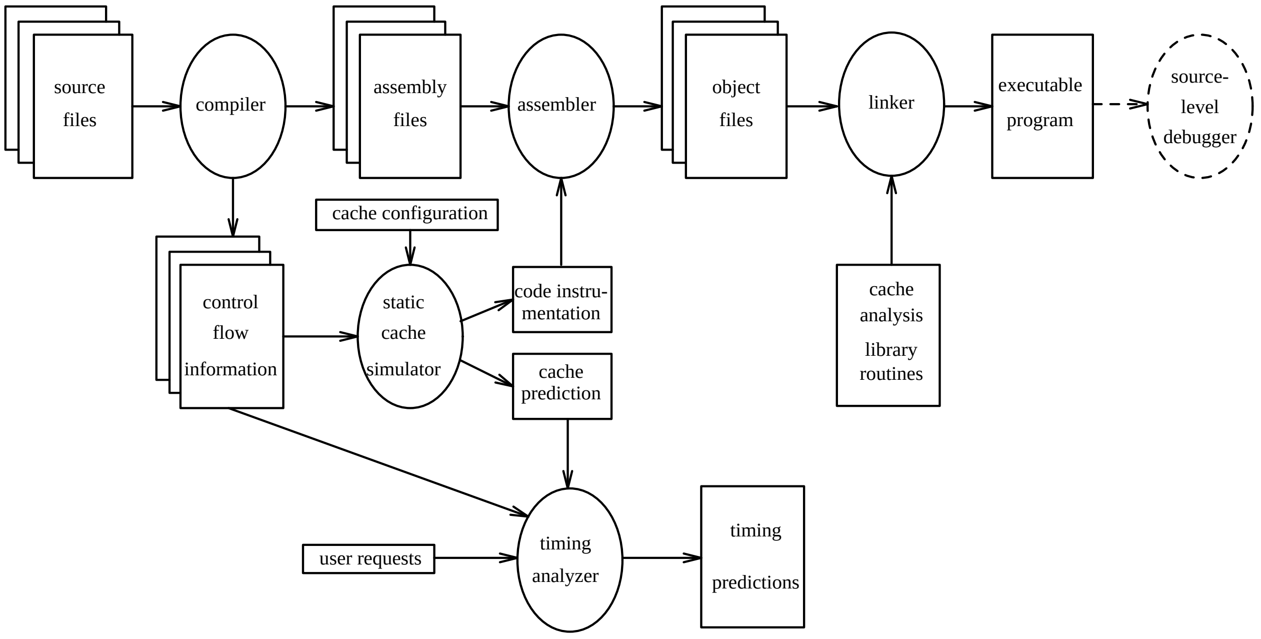


Figure 1.1: Overview of Static Simulation

generates assembly code with macro entries for instrumentation and passes information about the control flow of each source file to the static cache simulator. The simulator constructs the call graph of the program and the control-flow graph of each function based on the information provided by the compiler. The cache behavior is then simulated for a given cache configuration. The output of the static cache simulator depends on the intended application and will either describe the predicted cache behavior of each instruction or emit macro code together with tables to store cache information for on-the-fly cache analysis. In the former case, further analysis can be employed, for example to analytically bound the timing of code portions. In the latter case, the output of the simulator is passed to the assembler, which translates the code generated by the compiler into instrumented object code. The linker combines these object files to an executable program and links in library routines that may provide the final cache analysis results or support the output of intermediate results for debugging.

This dissertation is structured as follows. Chapter 2 introduces the reader to measurement techniques related to cache analysis. In Chapter 3, a method is introduced to determine a small set of measurement points for on-the-fly analysis. Chapter 4 details the method of static cache simulation for instruction caching. Chapter 5 illustrates the instrumentation of programs with measurement code. Chapter 6 presents the application of this work to fast instruction cache analysis. Chapter 7 describes the predictability of instruction caching in the context of real-time systems. Chapter 8 shows the benefits of this work for bounding the execution time of real-time applications in the presence of instruction caches. Chapter 9 provides a description of its application for real-time debugging. Chapter 10 discusses some future work. Chapter 11 summarizes the results of this work.

Chapter 2
Motivation and Prior Work

Cache memories have become an important part of recent microprocessor design. While the clock speed of processors has increased dramatically, the access time to main memory is lagging behind, causing a bottleneck. This bottleneck is dealt with by primary caches on the microprocessor and stand-alone secondary caches. Primary caches are a major contributor to the speed-up of memory access. Unified caches have been designed where instructions and data are not separated (Von-Neumann Architecture). For primary caches separate instruction caches and data caches are more popular (Harvard Architecture) [31, 29]. This is due to the trend of modern processors to pipeline the instruction execution. A pipelined architecture overlaps the different stages of instruction execution to effectively achieve a throughput of one instruction per clock cycle. Typical pipeline stages are instruction fetch, decode, load operands, execute, and store result. To achieve a throughput of one instruction per cycle, primary caches are generally split into instruction caches and data caches. The former feed the instruction pipeline and the latter provide access to program data via load or store instructions in a single cycle, provided that the information is cached. Thus, an instruction cache hit and a data cache hit can be served in the same cycle. This would not be possible if a primary cache was a unified cache.

The organization of caches varies from fully-associative caches to direct-mapped caches. Recent results have shown that direct-mapped caches tend to match, if not exceed, the speed of associative caches for large cache sizes [31]. Due to the slightly more complex design, the access time for hits of associative caches is generally slower (by about 10%) than the access time for direct-mapped caches. For large caches, the benefit of a higher hit ratio for set-associative caches is generally outweighed by the faster access for direct-mapped caches.

The simulation of caches has played an important role in the design of cache memories. Different design issues such as the number of cache sets, line size, level of associativity, and unified or split data and instruction caches have been investigated by cache simulation. In the following, a summary of different measurement techniques is given.

2.1 Measurement Techniques

Program analysis through profiling and tracing has long been used to evaluate new hardware and software designs. For instance, an early reference to profiling by Knuth can be found in [37, 39]. Measurement techniques can be distinguished by the provided level of detail of the program analysis. For example, the *ordering of events* allows to distinguish between a first execution of some code and the second execution of the same code portion, as well as those code portions executed in between. Depending on the intended analysis, the ordering of events may or may not be relevant. Below is a list of a variety of techniques used to gather measurements about program executions.

- **Sampling:** During execution, the program counter is sampled at regular intervals to determine which portion of the code is currently executing. This approach provides a statistical sampling method that yields approximate measurements indicating the estimated portion of the total execution time spent in certain portions of the program (typically at the scope of functions). To perform the sampling, common tools, such as `prof` [65] and `gprof` [25, 26], rely on the availability of hardware timers and the corresponding operating system interface to activate these timers, catch the timer interrupt, and sample the program counter at the interrupt point. It is neither possible to collect accurate measurements with this method, nor is it possible to deduce the order of events from the sample after program execution [40]. Yet, if the interrupt handler is used to collect and record trace data (see below), the order of events can be reconstructed [56].

- **Tracing:** This method involves the generation of a partial or full sequence of the instruction and data references encountered during program execution. Trace data is generated during program execution but analyzed at a later point in time. Thus, the trace data is commonly stored in a file. The technique produces accurate measurements and preserves the order of events during execution for later analyses. A common problem with this method is presented by the large size of trace data. It is therefore important to include only the (partial) information that is essential to reconstruct a full trace after program execution. In the following, different tracing techniques are described and their overhead is reported based on previous work [63].

  - **Hardware Simulation** of the execution of a program can be used to generate the trace for prototyped architectures. This technique is known to be very slow (100x to over 1000x slower than the original execution) but provides accurate and very detailed measurements [61].

  - **Single-Stepping** is a processor mode that interrupts the execution of a program after each instruction. The interrupt handler can be used to gather the trace data. This technique is just slightly faster the hardware simulation (100x – 1000x slow down) and works only for existing architectures [70, 21], though sometimes traces from an existing architecture are used to project the speed of prototyped architectures [56].

  - **Inline Tracing** is a technique where the program is instrumented before execution such that the trace data is generated by the instrumentation code as a side effect of the program execution. This technique requires a careful analysis of the program to ensure that the instrumentation does not affect data or code references. This technique is faster than the above techniques (about 10x slow down) and is currently regarded as the preferred method to collect trace data. A minimal set of instrumentation points can be determined by analyzing the control-flow graph [7].

    A number of different inline tracing techniques have been used to instrument the code and to process the trace data. Stunkel and Fuchs [62] instrumented the code during compilation and analyzed the trace on-the-fly as part of the program execution. Eggers *et. al.* [22] instrumented code during compilation and saved the trace data in secondary storage for later analysis. Borg *et. al.* [10] instrumented the program at link time and processed the trace data in parallel to the program execution by using buffers shared between the program and the analysis tool. The pros and cons of these approaches can be summarized as follows. Compile-time instrumentation has the advantage that unused registers can be utilized for the instrumentation code. Link-time instrumentation provides the means to instrument library code, which is not available as source code. Saving the trace data in a file allows later analysis with varying cache configurations but limits the trace size due to storage capacity and disk access overhead. Immediate processing of the trace data allows the analysis of longer traces but can only be performed for one cache configuration per program execution.

  - **Abstract Execution** is a variation of inline tracing [41]. This technique relies on a modified compiler that instruments the original program to generate a trace of "significant events" for a subset of basic blocks and a given program input. An abstract program, a scaled-down version of the original program, uses the significant events to generate a complete address trace by recording taken conditional branches. The abstract program is derived from the original program by omitting computations that do not influence the address trace, such as certain file I/O. Nevertheless, the execution time to obtain significant events might still be quite long.

  - **Microprogramming Instrumentation** provides a technique to modify the microcoded instruction set of a processor such that trace data is produced as a side effect of a program's execution. The technique is about as fast as inline tracing (about 20x slow down) due to the fact that the execution is still slowed down moderately by the additional microcode instructions [2]. Furthermore, the technique is generally not portable, and modern architectures, such as RISC processors, do not have microcode or do not provide the ability to reprogram microcode anymore.

  - **Hardware Monitoring** can be used to generate traces by probing the pins of a processor with dedicated instruments, *e.g.* a logic analyzer. The probes can be stored in a trace file. This technique requires additional, expensive hardware and some expertise to use this hardware. The technique is very fast since the program executes at its original speed and does not need to be modified [16, 15]. Yet, the method hides on-chip activities such as instruction or data references accessing primary caches.

- **Frequency Counting:** Similar to inline tracing, the program is modified to include instrumentation code. But rather than generating a program trace, the execution frequency of code portions is recorded for later analysis. Frequency measurements can be obtained very efficiently by inserting instructions into a program that increment frequency counters. The counters are typically associated with basic blocks and are incremented each time the basic block executes. The number of measurement points can be reduced from all basic blocks to a minimal set of control-flow transitions, which guarantees optimal profiling for most programs as explained by Ball and Larus [7]. Their reported overhead is a factor of 1.3-3.0 for basic block frequency accounting and 1.1-1.5 for optimal frequency accounting. The resulting measurements are accurate but it is not possible to reconstruct the order of events from frequency counts.

- **Inline On-the-fly Analysis:** This technique performs inline tracing and the analysis of trace data as part of the program execution. Instead of buffering trace data for a concurrent analysis tool, the program is modified to include instrumentation code that performs the trace *and* the analysis "on-the-fly" during program execution. This method requires a prior static analysis that performs the code instrumentation. This static analysis depends on the measurements requested by the user, *i.e.* cache performance analysis requires a specific static analysis and instrumentation code for this purpose. Several variations on the generation of trace data are possible. Stunkel and Fuchs [62] generated a full trace, Whalley [68, 69] only generated a partial trace for some highly tuned methods, and this dissertation discusses a method that does not generate any address trace at all during program execution. The overhead is about 1.2 to 2.2 for the method described in this dissertation, 10-30 for Stunkel and Fuchs, and about 2-15 for Whalley's most highly tuned method. The measurements are accurate and the order of events is preserved for the analysis. The program execution has to be repeated if simulation parameters change. With some additional effort, this method can even be used for prototyped architectures [19]. In this dissertation, inline on-the-fly analysis will be simply referred to as "on-the-fly analysis".

2.2 Cache Simulation

The task of cache simulation is to ascertain the number of cache hits and misses for a program execution. To determine if an instruction or data reference was a hit or miss, the cache simulator must be able to deduce the order of events, *i.e.* the order in which references occur. Simple sampling techniques and counting the frequency of blocks do not preserve the order of events and thus cannot be used for cache simulation. Tracing methods preserve the order of events but often require a hidden overhead for reading and writing the trace data file, even when the fastest methods are used. Processing trace data while it is generated eliminates the overhead of storing the entire trace but requires that each reference be interpreted to determine cache hits and misses. Some of the references have to be reconstructed to determine the full trace when only a partial trace is generated for a minimal set of instrumentation points.

The on-the-fly analysis techniques described by Whalley [68, 69] do not require the interpretation of each reference. Consecutive references are passed to the simulator as one block and only the first reference of a program line is simulated. Further improvements are based on the observation that many references in a loop result in misses during the first iteration but in hits for subsequent iterations if the loop fits in cache. In such cases, the trace action can be simplified for each subsequent iteration. But the performance of these techniques suggests that they do not scale well for small caches.

Unfortunately, on-the-fly analysis cannot be performed on the minimal set of measurement points used by inline tracing and frequency counting [7]. The minimal set of measurement points does not immediately provide a full set of events and their execution order. It is one of the objectives of this work to determine a small set of measurement points that still covers all events and preserves their execution order. This objective is addresses by the partitioning of the control flow into *unique paths*.

Another objective of this work is to take on-the-fly cache analysis one step further. Rather than simulating the entire cache behavior during program execution, a *static cache simulator* predicts a large portion of the references prior to program execution. If a reference is statically determined to be a cache hit or miss, simple frequency counters associated with the region of the reference suffice to account for its cache behavior at execution time. If the behavior of a reference is not known statically, it still has to be simulated during execution. Yet, instead of interpreting the addresses of references, localized state transitions are used to perform the dynamic analysis efficiently. There are several applications of static cache simulation that will be discussed to show to benefits of this approach.

Chapter 3
Control-Flow and Call-Graph Analysis

In this chapter, terms and methods are introduced to analyze the call graph of a program and the control-flow graph of each function. The analysis is performed to find a small set of measurement points suitable for on-the-fly analysis. The analysis provides a general framework to reduce the overhead of event-ordered profiling and tracing during program execution. Excerpts of this chapter can be found in [46].

This chapter precedes the central part of the dissertation, static cache simulation. In this chapter, the terms of a unique path, a unique path partitioning, and the function-instance graph are defined. These terms are used throughout this dissertation. For example, the static cache simulation is performed on a function-instance graph and the control-flow graph of each function. The control-flow graph can be represented in the traditional notion of basic blocks as vertices and control-flow transitions as edges, or it can be represented as a partitioning of unique paths. The choice depends on the application of static cache simulation.

3.1 Introduction

Program analysis through profiling and tracing has long been used to evaluate new hardware and software designs. In this chapter, a technique for efficient on-the-fly analysis of programs is presented.

Traditional tracing techniques rely on generating a program trace during execution, which is analyzed later by a tool. The problem of generating a minimal trace, which can later be expanded to a full event-ordered trace, can be regarded as solved. A near-optimal (often even optimal) solution to the problem for a control-flow graph G can be found by determining a maximum spanning tree $max(G)$ for the control-flow graph and inserting code on the edges of $G - max(G)$ [38, 7].

Recently, tracing and analyzing programs has been combined using inline tracing [10] and on-the-fly analysis [68, 69]. Both techniques require that events are analyzed as they occur. Traditional inline tracing performs the analysis separate from the generation of trace information.

On-the-fly analysis integrates the program analysis into its execution. The analysis is specialized for a certain application (*e.g.*, counting hits and misses for cache performance evaluation). The results of the analysis are available at program termination such that no concurrent analysis or post-execution analysis by any tool is required. If the application or the configuration changes, the program has to be executed again, sometimes even instrumented and then executed. In contrast, trace data can be analyzed by several tools and for several configurations once the data is generated. But the generation and analysis of trace data is typically slow and space consuming since the data is written to a file and later read again by a tool.

On-the-fly analysis requires that the program be instrumented with code, which performs the analysis. Many applications, including cache simulation, require that all events are simulated in the order in which they occur. In the past, each basic block was instrumented with code to support event-ordered analysis [62]. Inserting code based on the maximum spanning tree (or, to be more precise, on its complement) does not cover all events since instrumentation points are placed on a subset of the control-flow graph. It is therefore not applicable to on-the-fly analysis.

This chapter is structured as follows: First, a formal approach to reduce code instrumentation to a small number of places is introduced. This general framework supports efficient on-the-fly analysis of program behavior with regard to path partitioning. The focus is restricted to the analysis of the control-flow graph of a single function. Next, the formal model and the analysis are extended to the analysis of the entire program by transforming a call graph into a function-instance graph. Furthermore, a quantitative analysis is presented to show that the new methods reduce the number of measurement points by one third over traditional methods. Finally, future work is outlined, related work is discussed, and the results are summarized.

3.2 Control-Flow Partitioning into Unique Paths

The control flow of each function is partitioned into unique paths (UPs) to provide a small set of measurement points. The motivation for restructuring the control flow into UPs is twofold.

1. Each UP has a unique vertex or edge that provides the insertion point for instrumentation code at a later stage. This code may perform arbitrary on-the-fly analysis, *e.g.* simple profiling or more complex cache performance analysis.

2. Each UP is comprised of a range of instructions that are executed in sequence *if and only if* the unique vertex or edge is executed. This range of instructions does not have to be contiguous in the address space. The range of instructions provides a scope for static analysis to determine the instrumentation code for dynamic on-the-fly analysis, which preserves the order of events.

The first aspect, the strategy of instrumenting edges (or vertices where possible), is also fundamental to the aforementioned work on optimal profiling and tracing by Ball and Larus [7]. It is the second aspect that distinguishes this new approach from their work. The option of performing static analysis on the control flow to determine and optimize the instrumentation code for order-dependent on-the-fly analysis requires the definition of ranges for the analysis. Naively, one could choose basic blocks to comprise these ranges. But it has been demonstrated for profiling and tracing that fewer instrumentation points can be obtained by a more selective instrumentation technique. UPs provide such a framework supporting efficient instrumentation for on-the-fly analysis.

The set of UPs is called a unique path partitioning (UPPA) and is defined as follows: Let $G(V, E)$ be the control-flow graph (directed graph) of a function with a set of edges (transitions) E and a set of vertices (basic blocks) V.

Let p be a path

$$p = \nu_0, \epsilon_1, \nu_1, ..., \epsilon_n, \nu_n$$

with the ordered set of edges $\epsilon_p = \{\epsilon_1, ..., \epsilon_n\} \subseteq E$ and the ordered set of vertices $\nu_p = \{\nu_0, ..., \nu_n\} \subseteq V$, *i.e.*, a sequence of distinct vertices connected by edges [13]. The edge ϵ_i may also be denoted as $\nu_{i-1} \rightarrow \nu_i$. Vertex ν_0 is called an *head vertex* and vertex ν_n a *tail vertex*, while all other ν_i are *internal vertices*. Let H be the set of all head vertices and T be the set of all tail vertices.

Definition 1 (UPPA) *A unique path partitioning, $UPPA$, for a control-flow graph G(V,E) is a set of paths with the following properties:*

1. *all vertices are covered by paths:*

$$\forall_{v \in V} \quad \exists_{p \in UPPA} \quad v \in \nu_p$$

2. *each edge is either on a path or it connects a tail vertex to a head vertex, but not both:*

$$\forall_{e=(v \to w) \in E} \quad \exists_{p \in UPPA} \quad e \in \epsilon_p \oplus v \in T \wedge w \in H$$

3. *each path has a feature f, an edge or a vertex, which is* **globally** *unique,* i.e. *f is in no other path:*

$$\forall_{p \in UPPA} \quad (\exists_{e \in E} \quad e \in \epsilon_p \wedge \forall_{q \in UPPA \setminus \{p\}} \quad e \notin \epsilon_q) \vee (\exists_{v \in V} \quad v \in \nu_p \wedge \forall_{q \in UPPA \setminus \{p\}} \quad v \notin \nu_q)$$

4. *overlapping paths only share an initial or final subpath:*

$$\forall_{p,q \in UPPA} \quad \nu_p \cap \nu_q = \alpha \cup \beta$$

   where α and β denote the vertices of a common initial and final subpath, respectively. In other words, let $\nu_p = \{\nu_0, ..., \nu_m\}$ and $\nu_q = \{\omega_0, ..., \omega_n\}$ be the ordered sets of vertices for paths p and q. Then, $\alpha = \phi$ or $\alpha = \{\nu_0 = \omega_0, ..., \nu_i = \omega_i\}$ and $\beta = \phi$ or $\beta = \{\nu_k = \omega_l, ..., \nu_m = \omega_n\}$ for $i < k$ and $i < l$.

5. *proper path chaining:*

$$\forall_{p,q \in UPPA} \quad \forall_{v \in \nu_p, w \in \nu_q} \quad e = (v \to w) \in E \wedge e \notin \epsilon_p \cup \epsilon_q \Rightarrow v \in T \wedge w \in H$$

6. *break at calls: Let $C \subseteq V$ be the set of vertices (basic blocks) terminated by a call instruction.*

$$\forall_{v \in C, p \in UPPA} \quad v \in \nu_p \Rightarrow v \in T$$

7. *break at loop boundaries: Let L_i be the set of vertices in loop (cycle) i and let L be the set of all L_i.*

$$\forall_{e=(v \to w) \in E,\ p \in UPPA,\ L_i \in L} \quad e \in \epsilon_p \Rightarrow (v \in L_i \Leftrightarrow w \in L_i)$$

The properties 6 and 7 are operational restrictions motivated by the application of the partitioning for on-the-fly analysis of program behavior. The break at calls allows the insertion of instrumentation code for separate compilation. Thus, the compiler is not required to perform interprocedural analysis. The break at loop boundaries ensures that the frequency of events can be identified. The frequency of events outside a loop differs from the frequency inside loops (unless the loop was iterated only once). Thus, a UP associated with an event should not cross loop boundaries.
Example: Paths 1 and 2 in Figure 3.1 have two unique transitions each. They comprise an if-then-else structure. Paths 3 and 4 are generated because the loop is entered after basic block 4. Path 3 only has one unique transition while path 4 has two. Basic block 8 is outside the loop and therefore lies in a new path. ■

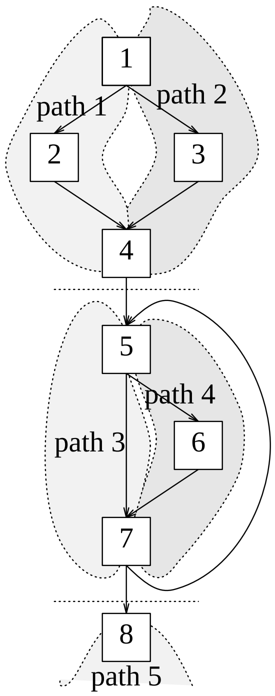


Figure 3.1: Unique Paths in the Control-Flow Graph

Theorem 1 (Existence of a UPPA) *Any control-flow graph G(V,E) has a UPPA.*

Proof: Let G(V,E) be a control-flow graph. Then, $UPPA_b = \{\{v_0\}, ..., \{v_n\}\}$ is a unique path partitioning, *i.e* each vertex (basic block) constitutes a UP. Each property of Definition 1 is satisfied:

1. Any vertex v_i is part of a UP $p_i = \{v_i\}$ by choice of the partitioning.

2. $\underset{p \in UPPA_b}{\forall} \quad \epsilon_p = \phi$ since all edges connect paths.

3. $\underset{p,q \in UPPA_b}{\forall} \quad p \neq q \Rightarrow \nu_p \cap \nu_q = \phi.$

   None of the UPs overlap in any vertex as shown for the previous property.

4. $\forall_{p_i=\{v_i\}\in UPPA_b} \quad v_i \in H \land v_i \in T$

5. The proof for the previous property suffices to prove this property as well.

6. $\forall_{e\in E,\ p\in UPPA_b} \quad e \notin \epsilon_p$, so the premise of can never be satisfied. Thus, the property is preserved. ■

Definition 2 (Ordering of UPPAs) *For a control-flow graph $G(V, E)$, a partitioning $UPPA_a$ is smaller than a partitioning $UPPA_b$ if $UPPA_a$ contains fewer paths than $UPPA_b$.*

The significance of the ordering is related to the number of measurement points for on-the-fly analysis. A smaller partitioning yields fewer measurement points, which improves the performance of on-the-fly analysis. The following algorithm provides a method to find a small partitioning. The algorithm uses the terminology of a loop header for a vertex with an incoming edges from outside the loop. A loop exit is a vertex with an outgoing edge leaving the loop. This is illustrated in Figure 3.2(a).

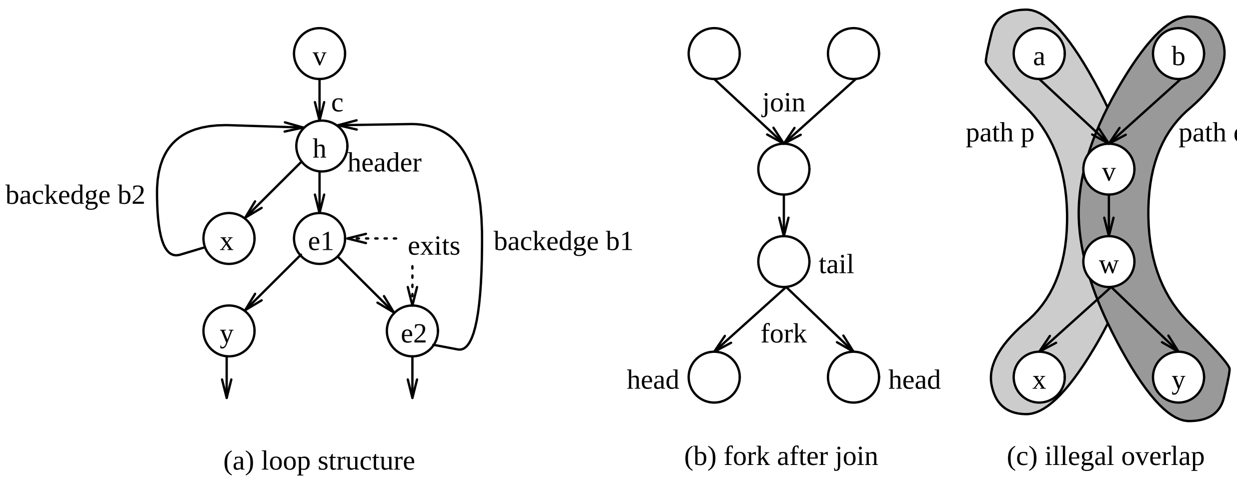


Figure 3.2: Sample Graphs

Algorithm 1 (Computation of a Small UPPA)
Input: Control-flow graph G(V,E).
Output: A small partitioning $UPPA$.
Algorithm: Let C be the set of vertices containing a call, let L_i be the set of vertices in loop i, and let L be the set of all L_i as in Definition 1. The algorithm then determines the beginning of paths (*heads*) and the end of paths (*tails*), for example at loop boundaries. In addition, a vertex is a tail if the path leading to this vertex *joins* with other paths and *forks* at the current vertex (see Figure 3.2(b)). Once the heads and tails have been determined, a path comprises a sequence of vertices and edges from a head to a tail in the control flow.

```
BEGIN
   FOR each $v \in V$ without any predecessor DO
      mark $v$ as *head*; /* entry blocks to the function */
   FOR each $v \in V$ without any successor DO
      mark $v$ as *tail*; /* return blocks from the function */
   FOR each $v \in C$ DO
      mark $v$ as *tail*; /* calls */
   FOR each $e = (v \rightarrow w) \in E$ WITH $v \notin L_i$ AND $w \in L_i$ DO
      mark $w$ as *head*; /* loop headers */
   FOR each $e = (v \rightarrow w) \in E$ WITH $v \in L_i$ AND $w \notin L_i$ DO
      mark $v$ as *tail*; /* loop exits */
   FOR each $v \in V$ DO
      mark $v$ as not *done*;

   WHILE change DO
      change:= False;
      propagate_heads_and_tails;
      FOR each $v \in V$ WITH $v$ marked as *head* AND
            not marked as *done* AND not marked as *tail* DO
         change:= True;
         mark $v$ as *done*;
         FOR each $e = (v \rightarrow w) \in E$ DO
            recursive_find_fork_after_join($w$, False);

   $UPPA = \phi$
   FOR each $v \in V$ WITH $v$ marked as *head* DO
      recursive_find_paths($v$, $\{v\}$);
END;

PROCEDURE propagate_heads_and_tails IS
   WHILE local_change DO
      local_change:= False;
      FOR each $v \in V$ DO
         IF $v$ marked as *head* THEN
            FOR each $e = (w \rightarrow v) \in E$ DO
               IF $w$ not marked as *tail* THEN
                  local_change:= True;
                  mark $w$ as *tail*;
         IF $v$ marked as *tail* THEN
            FOR each $e = (v \rightarrow w) \in E$ DO
               IF $w$ not marked as *head* THEN
                  local_change:= True;
                  mark $w$ as *head*;
END propagate_heads_and_tails;

PROCEDURE recursive_find_fork_after_join($v$, joined) IS
   IF $v$ marked as *tail* THEN
      return;
   IF $v$ joins, *i.e.* $v$ has more than once predecessor THEN
```

```
        joined:= True;
    IF joined AND $v$ forks, *i.e.* $v$ has more than once successor THEN
        mark $v$ as *tail*;
        return;
    FOR each $e = (v \rightarrow w) \in E$ DO
        recursive_find_fork_after_join($w$, joined);
END recursive_find_fork_after_join;

PROCEDURE recursive_find_paths($v$, $p$) IS
    IF $v$ marked as *tail* THEN
        $UPPA = UPPA \cup \{p\}$;
    ELSE FOR each $e = (v \rightarrow w) \in E$ DO
        recursive_find_paths($w$, $p \cup \{v \rightarrow w, w\}$);
END recursive_find_paths;
```

Example: Figure 3.3 illustrates two examples of the construction of a small UPPA using Algorithm 1. For the first example (upper part of Figure 3.3), vertices without predecessor (successor) are marked as head (tail). In addition, loop headers are heads and loop exits are tails. The second picture shows the same graph after **propagate_heads_and_tails** has been applied. Block 1 is marked as a tail since block 2 is a head. Conversely, block 7 is marked as a head since block 6 is a tail. The last picture depicts the graph after path partitioning through **recursive_find_paths**. Each head is connected to the next tail by one or more paths, depending on the number of different ways to reach the tail. The resulting UPPA has 5 paths.

The second example (lower part of Figure 3.1(b)) initially shows a graph whose vertices without predecessor (successor) are marked as heads (tails). The second picture shows an additional tail found by **recursive_find_fork_after_join** since there is a possible traversal for the head block 1 to, for example, the tail block 6, which encounters a join followed by a fork in block 4. The final graph depicts the effect of **propagate_heads_and_tails**. Blocks 5 and 6 are a head since 4 was a tail. Block 2 is a tail since block 5 is now a head. Thus, block 4 becomes a head. This causes block 3 to be marked as a tail. Finally, **recursive_find_paths** partitions the graph resulting in a UPPA with 5 paths. ■

Theorem 2 (Correctness of Algorithm 1) *Algorithm 1 constructs a UPPA for a control-flow graph $G(V, E)$.*

Proof:

Termination: It suffices to show that the WHILE loops and the recursive routines terminate. Both WHILE loops terminate since one more vertex is marked as head or tail during each iteration. This process terminates either when all vertices are marked as heads and tails or when none of the conditions for marking vertices are satisfied any longer. The recursive routine **recursive_find_fork_after_join** terminates for the following reasons. Initially, all loop headers are marked as heads. The propagation of heads and tails ensures that all predecessors of loop headers are marked as tails, in particular the vertices preceding a backedge in a loop. Since **recursive_find_fork_after_join** terminates when a tail is encountered, it will stop at a tail vertex with an outgoing backedge or at a tail vertex without any successor since it can only traverse forward edges in the control-flow graph. This also applies for **recursive_find_paths**.

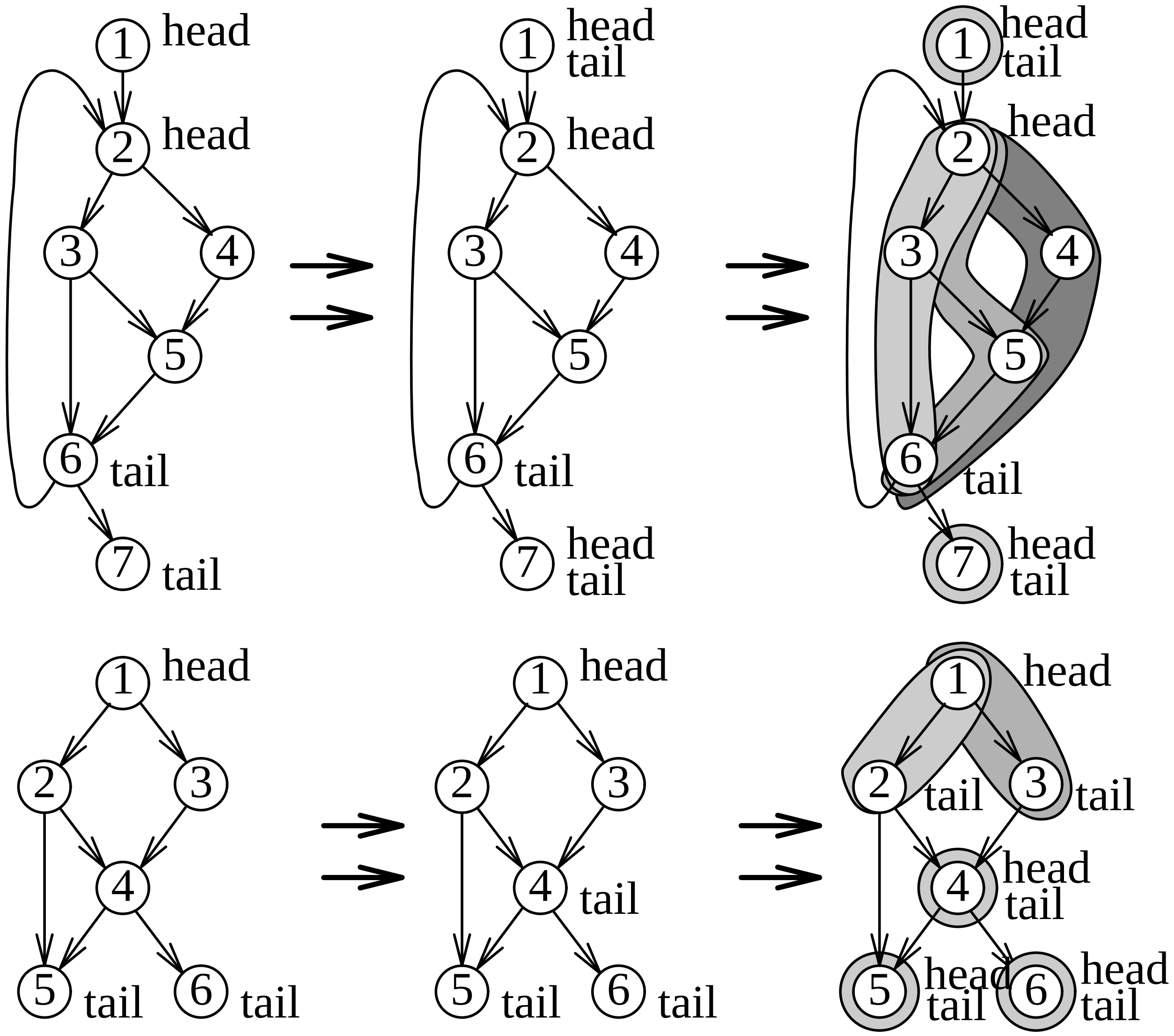


Figure 3.3: Algorithmic Construction of Two Small UPPAs

Output is a UPPA: It has to be shown that the properties of a UPPA as stated in Definition 1 hold for Algorithm 1.

1. All vertices are covered since **recursive_find_paths** includes all vertices between a head and a tail in some path. Due to **propagate_heads_and_tails**, an outgoing edge of a tail vertex always leads to a head vertex, *i.e.* there cannot be any intermediate vertices between a tail and a head. Furthermore, at least the initial vertex (without predecessor) is a head and the final vertices (without successors) are tails.

2. Consider any edges between a head and a tail. These edges are included in some path by **recursive_find_paths**, and these are all edges on paths. The remaining edges are those connecting tails to heads and are not in any path.

3. The following cases have to be distinguished for construction of paths by **recursive_find_paths**: If there are no forks between a head h and the next tail, then there will only be one path starting at h, and h is a unique vertex for this path. If there are forks after a head h but no joins, then the tail vertex will be unique for each path starting in h. If there are forks after a head h, followed by the first join at vertex v along some path starting in h, then the edge immediately preceding v on this path will be unique (since no other path has joined yet). Notice that there cannot be another fork after the join in v within the path since any forking vertex would have been marked as a tail by **recursive_find_fork_after_join**.

4. Property 3 ensures that any two overlapping paths differ in at least an edge. (Notice that a unique vertex implies a unique edge for non-trivial paths with multiple vertices.) Assume there exist two paths p, q that overlap in a subpath $\{v, ..., w\}$ (see Figure 3.2(c)) and v is preceded by distinct vertices a and b in p and q, respectively. Also, w is succeeded by distinct vertices x and y in p and q, respectively. In other words, p and q overlap somewhere in the middle of their paths. Then, two edges join in vertex v and two edges fork from vertex w, *i.e.* a join is followed by a fork. Thus, w should have been mark as a tail by **recursive_find_fork_after_join**. Therefore, w should have been the last vertex of paths p and q. Contradiction.

5. All edges between a head and the next tail are covered by paths, as shown for property 2. Thus, it suffices to observe that edges connecting a tail t to a head h always connect all paths ending with vertex t to the paths starting with vertex h. It is guaranteed by **recursive_find_paths** that a path starts with a head vertex and ends in a tail vertex.

6. Each vertex v containing a call is initially marked as a tail vertex. Thus, vertex v must be the final vertex for any path containing v by construction of the paths (**recursive_find_paths**).

7. Each loop header vertex is initially marked as a head and each loop exit is marked as a tail. Thus, the vertices preceding a loop header are marked as a tail and the vertices succeeding a loop exit are marked as heads by **propagate_heads_and_tails**. Furthermore, the edges crossing loop boundaries connect the paths ending in the tail vertex to the paths starting with the head vertex. As already shown for property 2, edges between a tail and a head cannot be covered by any path. ■

In terms of the ordering of UPPAs, the basic block partitioning $UPPA_b$ is the partitioning with the largest number of measurement points. Algorithm 1 constructs a partitioning that has an equal or smaller number of measurement points. It was found that the algorithm produces a much smaller UPPA if possible. The algorithm may in fact produce a minimal UPPA (with the smallest possible number of measurement points). Attempts to prove the minimality have not yet succeeded due to the fact the a given graph may have more than one minimal UPPA.

In summary, the control-flow graph can be transformed into a small UPPA by Algorithm 1. The small set of measurement points is given by a unique vertex or unique edge of each UP. This provides the framework for efficient on-the-fly analysis with regard to the definition of UPPAs.

Another short example for a small UPPA construction shall be given, which is used to discuss the possibility of letting paths begin and end in edges as well as vertices.
Example: Consider the subgraph of Figure 3.2(a) that is inside the loop. A corresponding $UPPA_s$ can be constructed by Algorithm 1 resulting in the following partitioning:

$$UPPA_s = \{\{h, h \to x, x\}, \{h, h \to e1, e1\}, \{e2\}\}$$

■

In general, the method may still be further tuned with regard to the dynamic behavior. Currently, a path has to begin and end in a vertex. Consider the notion of *open paths* that can start and end in a vertex *or* an edge. Then, another small UPPA of the loop in Figure 3.2(a) would be:

$$UPPA_t = \{\{h, h \to x, x\}, \{h, h \to e1, e1, e1 \to y\}, \{h, h \to e1, e1, e1 \to e2, e2\}\}$$

Consider the number of measurement points executed during each loop iteration. For $UPPA_s$, there are two measurement points for an iteration reaching $b1$, one each in paths 2 and 3. For $UPPA_t$, there is only one measurement point on $b1$ in path $3'$. The definition of UPPAs does not take dynamic properties into account.

3.3 From Call Graph to Function-Instance Graph

The small set of measurement points provides the location for inserting measurement code that records the order of events. While the actual measurement code depends on the intended analysis of the program, the amount of the measurement code may be further reduced by distinguishing between different call sites of a function. For an event-ordered analysis, the first invocation of a function may trigger certain initialization events. The analysis of subsequent calls to the same function are simplified by the assumption that these initialization events have already occurred. Such an example will be illustrated later in the context of instruction cache analysis.

A program may be composed of a number of functions. The possible sequence of calls between these functions is depicted in a call graph [3]. Functions can be further distinguished by function instances. An instance depends on the call sequence, *i.e.* on the immediate call site of its caller, the caller's call site, etc. The function instances of a call graph are defined below. The definition excludes recursive calls that require special handling and are discussed later. Indirect calls through function pointers are not handled since the callee cannot be statically determined.

Definition 3 (Function Instances) *Let $G(V, EC)$ be a call graph where V is the set of functions including an initial function "main" and EC is a set of pairs (e, c). The edge*

$e = v \rightarrow w$ denotes a call to w within v (excluding recursive and indirect calls). The vertex c is a vertex of the control-flow graph of v that contains a call site to w. Then, the set of function instances is defined recursively:

1. *The function (vertex) "main" has a single instance $main_0$.*
2. *Let $(f \rightarrow g, c) \in EC$ and f_i be an instance of f. Then, g_{c,f_i} is an instance.*
3. *These are all the function instances.*

The call graph of a program without recursion (*i.e.*, a directed acyclic graph) can be transformed into a tree of function instances by a depth-first search traversal of the call graph. Function instances can then be uniquely identified by their index, where f_i denotes the ith occurrence of function f within the depth-first search.

Backedges in the call graph corresponding to recursive calls can be detected by marking vertices as visited during the depth-first traversal. If an already visited edge is encountered again, the last edge in the current traversal is due to recursion. The depth-first search will then backtrack and retain this backedge as a special edge in the function-instance graph (see Algorithm 3 in Appendix A).
Example: In Figure 3.4, function f contains three calls: a call to g and two calls to h.

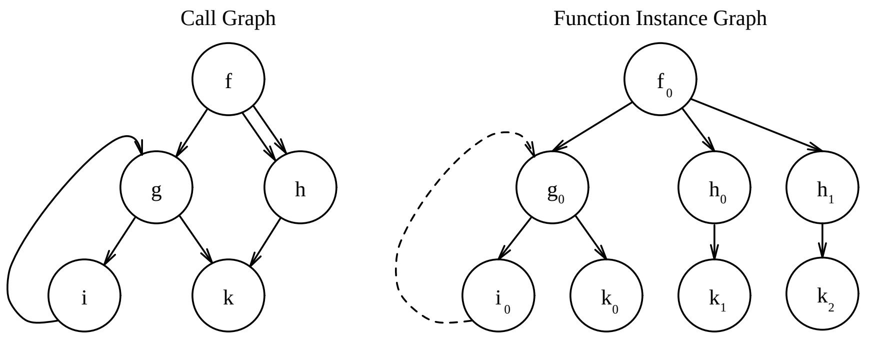


Figure 3.4: Construction of Function-Instance Graph

Function g calls i and k. Function h calls k. Function i calls g, which is an indirect recursive call. The corresponding function-instance graph contains two instances of h (for each call from f_0) and three instances of k (for the calls from g_0, h_0, h_1). The backedge $i \rightarrow g$ due to indirect recursion is retained as a special edge in the function-instance graph. ■

The construction of a function-instance graph does not result in inlining, partial evaluation, or any other form of code replication. It is merely a decomposition that facilitates cache analysis. But the code instrumentation includes information to identify a function instance during execution.

3.4 Performance Evaluation

This chapter evaluates the benefits of control-flow partitioning and function-instance graphs to reduce the number of measurement points. Table 3.1 summarizes the performance tests for user programs, benchmarks, and UNIX utilities. The numbers were produced by modifying the back-end of an optimizing compiler VPO (Very Portable Optimizer) [8] to determine measurement points by partitioning the control flow and by creating the function-instance graph.

Table 3.1: Results for Measurement Overhead

		Size	Instructions		Measure Pts.	
Name	Description	[bytes]	exec.	in FIG	static	exec.
cachesim	Cache Simulator	8,460	2,995,817	13,776	73.38%	60.56%
cb	C Program Beautifier	4,968	3,974,882	12,735	89.62%	65.61%
compact	Huffman Code Compression	5,912	13,349,997	3,226	68.89%	56.56%
copt	Rule-Based Peephole Optimizer	4,148	2,342,143	1,309	84.19%	74.88%
dhrystone	Integer Benchmark	1,916	19,050,093	644	81.61%	72.73%
fft	Fast Fourier Transform	1,968	4,094,244	536	78.43%	74.08%
genreport	Execution Report Generator	17,720	2,275,814	8,968	71.58%	81.31%
mincost	VLSI Circuit Partitioning	4,448	2,994,275	2,198	83.19%	76.27%
sched	Instruction Scheduler	8,272	1,091,755	5,410	73.16%	58.29%
sdiff	Side-by-side File Differences	7,288	2,138,501	16,463	72.13%	77.82%
tsp	Traveling Salesman	4,724	3,004,145	1,548	64.08%	58.67%
whetstone	Floating point benchmark	4,816	8,520,241	1,667	70.49%	68.25%
average		6,220	5,485,992	5,707	75.90%	68.75%

The size of the programs varied between about 2kB and 18kB (see column 3). The number of instructions executed for each program comprised a range of 1 to 19 million using realistic input data for each program (see column 4). Column 5 shows the static number of instructions in the program after expanding the call graph into a function instance graph and is used by subsequent chapters for comparison. Column 6 indicates the percentage of measurement points required for the new UPPA method versus the number of measurement points inserted in conventional on-the-fly analysis (*i.e.*, one measurement point per basic block). The new method requires only 76% of the measurement points required for the traditional trace-driven analysis, *i.e.* about 24% fewer measurement points statically. The run-time savings (column 7) are even higher, requiring only about 69% of the measurement points executed under traditional trace-driven analyses. The additional dynamic savings are due to reducing sequences of basic blocks inside loops to fewer UPs, sometimes just to a single UP.

3.5 Future Work

As discussed previously, it may be possible to guarantee more efficient results for on-the-fly analysis in the general case by extending paths to open paths. Also, it still remains an open question if the Algorithm 1 could be proved to produce a minimal UPPA.

3.6 Related Work

Traditional profiling and tracing is often performed by collecting trace information during program execution that is analyzed afterwards by a separate tool, which reconstructs the order of events. It has been well established that a small set of measurement points for this traditional approach can be provided by the edges of $G - max(G)$, where G is the control-flow graph and $max(G)$ is its maximum spanning tree [38, 7]. The resulting placement is optimal for a large class of control-flow graphs, in particular reducible graphs resulting from structured programming constructs, and it is near-optimal for most other cases.

It shall be noted that placing measurement code on an edge may involve the creation of new basic blocks and unconditional jumps. Samples [58] challenges the claim that the maximum spanning tree approach is optimal. He argues that the overhead of control-flow transformations should be taken into account. He develops a heuristic model to assess the approximate cost of control-flow transformations for code instrumentation and conjectures that an optimal solution may be NP-complete. In practice, the overhead of control-flow transformations for placing instrumentation code on edges is generally small and therefore mostly neglected.

Lately, on-the-fly analysis has been performed for collecting all measurements for a certain analysis during program execution and generally results in a lower overall overhead than traditional tracing methods. In the past, on-the-fly analysis was performed at the level of basic blocks [19].

Independent research by Emami *et. al.* [23] defines an *invocation graph* that has properties similar to the function-instance graph. Their intention lies in interprocedural data-flow and alias analysis. The handling of recursion in a function-instance graph was inspired by their work but realized differently due to the different application.

3.7 Conclusion

In this chapter, a formal method was developed to perform efficient on-the-fly analysis of program behavior with regard to path partitioning. The method partitions the control-flow graph into a small set of unique paths, each of which contain a unique edge or vertex where instrumentation code can be placed. Furthermore, the construction of the function-instance graph from a program's call graph refines the analysis. Performance evaluations show that the number of dynamic measurement points can be reduced by one third using these methods.

Chapter 4

Static Cache Simulation

This chapter introduces the method of static cache simulation that provides the means to predict the behavior of a large number of cache references prior to execution time of a program. The method is based on a variation of an iterative data-flow algorithm commonly used in optimizing compilers. It utilizes control-flow partitioning and function-instance graphs for predicting the caching behavior of each instruction. No prior work on predicting caching behavior statically could be found in the literature. Excerpts of this chapter can be found in [46, 50].

4.1 Introduction

In the last chapter, a framework for efficient on-the-fly analysis was developed. One application for on-the-fly program analysis is cache performance evaluation. Different cache configurations can be evaluated by determining the number of cache hits and misses for a set of programs. Cache analysis can be performed on-the-fly or by analyzing stored trace data, though faster results have been reported for the former approach [69].

This chapter introduces the method of static cache simulation, which predicts the caching behavior of a large number of instruction references prior to execution time[1]. The method employs a novel view of cache memories that seems to be unprecedented. The method is based on a variation of an iterative data-flow algorithm commonly used in optimizing compilers. It can be used to reduce the amount of instrumentation code inserted into a program for on-the-fly analysis. It can also be used to enable a program timing tool to take the effects of caching into account. These and other applications of static cache simulation are discussed in later chapters.

This chapter is structured as follows: First, the categorization of instructions for cache analysis is formalized. Next, an algorithm is presented to calculate the information required for instruction categorization within one function. The algorithm is then extended to interprocedural analysis. Furthermore, measurements of a simple program are discussed. Finally, future work, related work, and conclusions are presented.

4.2 Instruction Categorization

Static cache simulation calculates the abstract cache states associated with UPs. The calculation is performed by repeated traversal of the function-instance graph and the UPPA of each function.

Definition 4 (Potentially Cached) *A program line l can potentially be cached if there exists a sequence of transitions in the combined UPPAs and function-instance graph such that l is cached when it is reached in the UP.*

Definition 5 (Abstract Cache State) *The abstract cache state of a program line l within a UP and a function instance is the set of program lines that can potentially be cached prior to the execution of l within the UP and the function instance.*

[1]Data cache references could be predicted in a similar manner but are not discussed here.

The notion of an abstract cache state is a compromise between a feasible storage complexity of the static cache simulation and the alternative of an exhaustive set of all cache states that may occur at execution time with an exponential storage complexity.

Based on the abstract cache state, it becomes possible to statically predict the caching behavior of each instruction of a program. Instructions may be categorized as *always-hit*, *always-miss*, *first-miss*, or *conflict*. The semantics for each category is as follows. Always-hit (always-miss) instructions will always result in a cache hit (miss) during program execution. First-miss instructions will result in a cache miss on the first reference to the instruction and in a cache hit for any consecutive references. Conflict instructions may result in a cache hit or a cache miss during program execution, *i.e.* their behavior cannot be predicted statically through this simulation method. The different categories are defined below after introducing the notion of a reaching state.

Definition 6 (Reaching State) *The reaching state of a UP within a function instance is the set of program lines that can be reached through control-flow transitions from the UP of the function instance.*

Definition 7 (Instruction Categorization) *Let i_k be an instruction within a UP and a function instance. Let $l = i_0..i_{n-1}$ be the program line containing i_k and let i_{first} be the first instruction of l within the UP. Let s be the abstract cache state for l within the UP. Let l map into cache line c, denoted by $l \rightarrow c$. Let t be the reaching state for the UP. Then, the instruction categorization is defined as*

$$\text{category } (i_k) = \begin{cases} \text{always-miss if } k = \mathit{first} \wedge l \notin s \\ \text{always-hit} \quad \text{if } k \neq \mathit{first} \vee (l \in s \wedge \underset{m \rightarrow c, m \neq l}{\forall} \; m \notin s) \\ \text{first-miss} \quad \text{if } k = \mathit{first} \wedge l \in s \wedge \underset{m \rightarrow c, m \neq l}{\exists} \; m \in s \wedge \underset{m \rightarrow c, m \neq l}{\forall} \; m \in s \Rightarrow m \notin t \wedge \\ \qquad\qquad \underset{0 \leq x < n}{\forall} \; \mathit{category}(i_x) \in \{\text{always-hit, first-miss}\} \\ \text{conflict} \qquad \text{otherwise} \end{cases}$$

An always miss occurs when instruction i_k is the first instruction encountered in program line l and l is not in the abstract cache state s. An always hit occurs either if i_k is not the first instruction in l or l is the only program line in s mapping into c. A first miss occurs if the following conditions are met. First, i_k is first in l, and l and at least one other program line m (which maps into c) are in s. Second, if one such line m is in s, then the line must not be reachable anymore from the current UP. Third, all other instructions in the program line have to be either always hits or first misses. A conflict occurs in all other cases.

This categorization results in some interesting properties. If the size of the program does not exceed the size of the cache, hardly any instructions will be categorized as conflicts. Thus, the cache behavior can mostly be statically predicted.[2] As the program becomes much larger than the cache, the number of conflicts increases to a certain point. This point depends on the ratio between program size and cache size. After this point, conflicts start to decrease again while first misses increase.

The definition for instruction categorization is refined according to the application. This is discussed in later chapters in the context of a number of applications.

[2]The adaptations of the definition for different applications in later chapters will provide static predictability of all instructions if the program fits into cache, *i.e.* no instruction will be categorized as a conflict in this case. Since the adaptation depends on the application it could not be incorporated in the original definition.

4.3 Calculation of Abstract Cache States

Algorithm 2 (Calculation of Abstract Cache States)
Input: Function-Instance Graph of the program and UPPA for each function.
Output: Abstract Cache State for each UP.
Algorithm: Let `conf_lines(UP)` be the set of program lines (excluding the program lines of UP), which map into the same cache line as any program line within the UP.

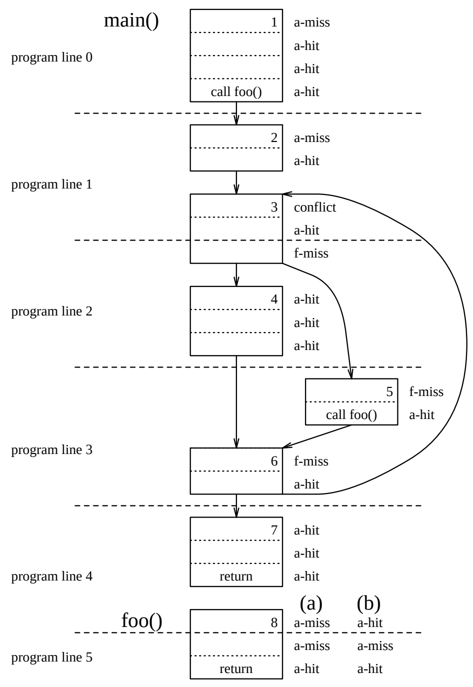
```
input_state(main):= all invalid lines;
WHILE any change DO
   FOR each instance of a UP in the program DO
      input_state(UP):= $\phi$;
      FOR each immediate predecessor P of UP DO
         input_state(UP):= input_state(UP) $\cup$ output_state(P);
      output_state(UP):= [input_state(UP) $\cup$ prog_lines(UP)] $\setminus$ conf_lines(UP);
   propagate_states
```

The iterative Algorithm 2 calculates the abstract cache states. In the algorithm, the abstract cache state of the program line of a UP that is referenced first is referred to as `input_state`. Conversely, the abstract cache state after the program line of a UP that is referenced last is referred to as `output_state`. The set of vertices (basic blocks) in a UP provides the scope of program lines to transform an input_state into an output_state. The interprocedural propagation of states, `propagate_states`, is explained in the next section.

The algorithm is a variation of an iterative data-flow analysis algorithm commonly used in optimizing compilers. Thus, the time overhead of the algorithm is comparable to that of data-flow analysis and the space overhead is $O(pl * UPs * fi)$, where pl is the number of program lines, UPs is the number of paths, and fi the number of function instances. The correctness of the algorithm for data-flow analysis is discussed in [3]. The calculation can be performed for an arbitrary control-flow graph, even if it is irreducible. In addition, the order of processing basic blocks is irrelevant for the correctness of the algorithm. The reaching states can be calculated using the same base algorithm with `input_state(main) = conf_lines(UP) =` ϕ.
Example: Figure 4.1 depicts the calculation of input and output states. The chosen UPPA is $UPPA_b$, the basic block partitioning.[3] In the example, there are 4 cache lines and the line size is 16 bytes (4 instructions). Thus, program line 0 and 4 map into cache line 0, program line 1 and 5 map into cache line 1, program line 2 maps into cache line 2, and program line 3 maps into cache line 3. The immediate successor of a block with a call is the first block in that instance of the called function. Block 8a corresponds to the first instance of foo() called from block 1 and block 8b corresponds to the second instance of foo() called from block 5. Two passes are required to calculate the input and output states of the blocks, given that the blocks are processed in the order shown in Figure 4.1. Only the states of blocks inside the loop (except for blocks 6 and 8b) change on the second pass. Pass 3 results in no more changes.

After determining the input states of all blocks, each instruction is categorized based on its abstract cache state (derived from the input state) and the reaching state shown in the figure. By inspecting the input states of each block, one can make some observations that may not have been detected by a naive inspection of only physically contiguous sequences of

[3]Algorithm 2 operates on any UPPA, and the categorization is not influenced by the choice of a UPPA. The $UPPA_b$ simplifies the example but would result in more measurement overhead during on-the-fly analysis than a smaller UPPA constructed by Algorithm 1.



```
 "I" = invalid

cache   0 1 2 3 0 1 2 3 0 1 cache ln. 0 1 2 3 0 1 2 3 0 1 cache line
program I I I I 0 1 2 3 4 5 prog. ln. I I I I 0 1 2 3 4 5 program line

PASS 1
------
 in(1)=[I I I I            ]  out(1)=[  I I I 0          ]
in(8a)=[  I I I 0          ] out(8a)=[    I I         4 5]
 in(2)=[    I I         4 5]  out(2)=[    I I   1     4  ]
 in(3)=[    I I   1     4  ]  out(3)=[      I   1 2   4  ]
 in(4)=[      I   1 2   4  ]  out(4)=[      I   1 2   4  ]
 in(5)=[      I   1 2   4  ]  out(5)=[          1 2 3 4  ]
in(8b)=[          1 2 3 4  ] out(8b)=[            2 3 4 5]
 in(6)=[      I   1 2 3 4 5]  out(6)=[          1 2 3 4 5]
 in(7)=[          1 2 3 4 5]  out(7)=[          1 2 3 4 5]

PASS 2
------
 in(1)=[I I I I            ]  out(1)=[  I I I 0          ]  reach(1)=[  1 2 3 4 5]
in(8a)=[  I I I 0          ] out(8a)=[    I I         4 5] reach(8a)=[  1 2 3 4 5]
 in(2)=[    I I         4 5]  out(2)=[    I I   1     4  ]  reach(2)=[  1 2 3 4 5]
 in(3)=[    I I   1 2 3 4 5]  out(3)=[      I   1 2 3 4  ]  reach(3)=[  1 2 3 4 5]
 in(4)=[      I   1 2 3 4  ]  out(4)=[      I   1 2 3 4  ]  reach(4)=[  1 2 3 4 5]
 in(5)=[      I   1 2 3 4  ]  out(5)=[          1 2 3 4  ]  reach(5)=[  1 2 3 4 5]
in(8b)=[          1 2 3 4  ] out(8b)=[            2 3 4 5] reach(8b)=[  1 2 3 4 5]
 in(6)=[      I   1 2 3 4 5]  out(6)=[          1 2 3 4 5]  reach(6)=[  1 2 3 4 5]
 in(7)=[          1 2 3 4 5]  out(7)=[          1 2 3 4 5]  reach(7)=[           ]
```

Figure 4.1: Example with Flow Graph

references. For instance, the static simulation determined that the first instruction in block 7 will always be in cache (always hit) due to spatial locality since program line 4 is in `in(7)` and no conflicting program line is in `in(7)`. It was also determined that the first instruction in basic block 8b will always be in cache (always hit) due to temporal locality. The static simulation determined that the last instruction in block 3 will not be in cache on its first reference, but will always be in cache on subsequent references (first miss). This is indicated by `in(3)`, which includes program line 2 but also a conflicting program line "invalid" for cache line 3. Yet, the conflicting program line cannot be reached. This is also true for the first instructions of block 5 and 6 though a miss will only occur on the first reference of either one of the instructions. This is termed a *group first miss* and is discussed later. Finally, the first instruction in block 3 is classified as a conflict since it could either be a hit or a miss (due to the conditional call to foo). This is indicated by `in(3)`, which includes program line 1 and a conflicting program line 5 that can still be reached. ■

4.4 Interprocedural Propagation

The notion of function instances reduces the complexity of the cache states propagated across functions. Consider the calls from function f to h in Figure 3.4. If there was no distinction between the instances h_0 and h_1, it could not be determined if a program line in h was cached, *i.e.* most lines would be considered conflicts. Using function instances, it is known that the first call h_0 will result in many cache misses to bring the program lines of h into cache while the second call h_1 results in many hits (assuming that the lines of h were retained in cache between the calls).

Algorithm 2 illustrates the calculation of abstract cache states. But it does not show how the states are propagated across function instances. The pseudo code in Figure 4.2 fills this

```
PROCEDURE propagate_states IS
  FOR each function F instance I DO
    FOR each path P in F with a call to function G instance K DO
      FOR each entry path E in G DO
        input_state(E,K):= output_state(P,I);
      FOR each path Q that is a successor path of P DO
        input_state(Q,I):= $\phi$;
        FOR each exit path E in G DO
          input_state(Q,I):= input_state(Q,I) cup output_state(E,K);
END propagate_states;
```

Figure 4.2: Pseudo Code for Propagation of Abstract Cache States

gap. Notice that a function instance may have multiple entry paths and exit paths due to the definition of UPs. Informally, the output states of the UP at the call site (of the caller's instance) are propagated into the input states of the entry blocks of the callee's instance. Conversely, the union of the output states of the callee's instance are propagated into the input state of the single UP that succeeds the call site (of the caller's instance).

4.5 Measurements

Some of the characteristics of the instruction categorization have already been discussed. Figure 4.3 shows the distribution of each instruction category for varying cache sizes of a sample program. The sample program performs a fast Fourier transformation and has a code size of slightly less than 2kB. The numbers correspond to the static prediction by the static cache simulation for a cache line size of 16 bytes (4 instructions).

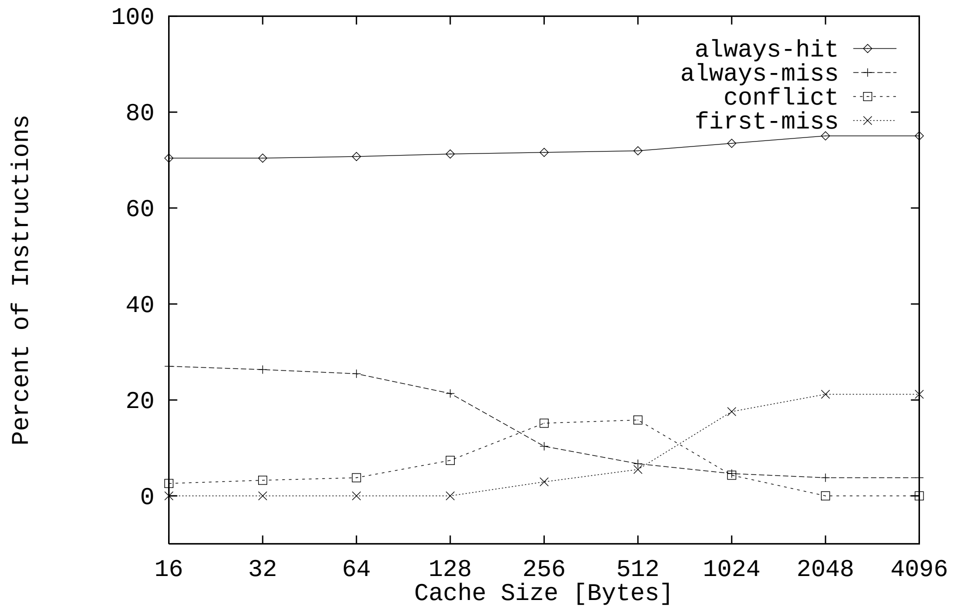


Figure 4.3: Distribution of Static Cache Prediction for fft

The number of always hits increases slightly with the cache size but, overall, 70-75% of the instructions are predicted as always hits. This number is affected by the size of a cache line. In this case, the first instruction of each line mostly does not result in a hit but once the line is brought into cache, the remaining 3 instructions are hits. This explains the static approximation of 75% of hits for large cache sizes.

The number of always misses is large (about 27%) for small cache sizes due to capacity misses of small caches. But as the cache size increases, misses are reduced to compulsory misses due to bringing a program line into cache for the first time and stays constant (at about 4% here) once the whole program fits into cache (at 2kB cache size).

The number of conflicts starts out relatively low (about 2%), reaches a peak (at about 15% here) when the cache size is about a quarter of the program size, and reaches zero once the entire program fits into cache. For small cache sizes, program lines that map into the same cache line are often certain to be capacity misses as discussed before. As the cache size increases, it can no longer be determined statically whether a program line always replaces another or not. Once the program fits into cache, only one program line maps into a cache line and conflicts are complete eliminated.

The number of first misses is zero for small cache sizes, gradually increases and stabilizes (at 21% here) once the entire program fits into cache. The following conditions have to be met for first misses. First, the cache has to be large enough to hold a program line of a loop. Second, other program lines mapping into the same cache line must either not exist or must not be reachable anymore.

Overall, the large number of always misses for small caches is first replaced by mostly conflicts as the cache size increases, then conflicts and always misses are replaced by first misses (and by a few always hits). A more comprehensive analysis of the effects of static cache simulation are given later in the context of various applications.

4.6 Future Work

So far, only instruction caching has been simulated. Current work includes the application of static cache simulation to data caches under certain restrictions, such as the absence of pointers and dynamic memory allocation (which are feasible assumptions for the design of predictable real-time applications). However, many addresses of data references are known statically. For instance, static or global data references retain the same addresses during the execution of a program. Addresses of run-time stack references can be statically determined as well in the absence of recursion. Compiler flow analysis can be used to detect the pattern of many calculated references, such as indexing through an array. Previous work has shown improvements by balancing the number of instructions placed behind loads where the memory latency was uncertain [34]. By predicting the memory latency of a large portion of loads, instruction scheduling could be performed more effectively. For example, the number of instructions the scheduler would place between a load instruction and the first instruction referencing the loaded register should be greater for a data reference classified as an always miss than an always hit.

The current implementation of the static simulator imposes the restriction that only direct-mapped cache configurations are allowed. Recent results have shown that direct-mapped caches have a faster access time for hits, which outweighs the benefit of a higher hit ratio in set-associative caches for large cache sizes [31]. Yet, current micro-processors are still designed with set-associative caches [12]. A modified algorithm and data structure could be designed to handle set-associative caches within the framework of static cache simulation.

The implementation of the static cache simulator currently rejects the analysis of recursive functions. This restriction can be lifted by denoting recursion as described in the context of the function-instance graph and by applying the described algorithm to calculate abstract cache states repeatedly for backedges due to recursion.

Furthermore, static cache simulation can only be applied accurately to split data and instruction caches. This is due to the limited information about data references that can be inferred statically. In a unified cache design the interference between data caching and instruction caching may not always be known statically. Also, the current design only covers primary (on-chip) caches. The simulation of secondary caches would be possible by taking the cache behavior of a primary cache into account. Yet, most secondary caches are unified caches and cannot be accurately simulated by this method as of now.

Finally, indirect calls are not handled since the static simulator must be able to generate an explicit call graph. It may be possible for the compiler to determine some values of function pointers but this does not seem to be possible for the general case of function pointers resulting from arithmetic expressions. The same applies to non-local transfers of control such as `setjmp()` and `longjmp()`.

4.7 Related Work

The idea to statically simulate a portion of the cache behavior seems to be unprecedented in research. Conventional methods for cache analysis use hardware simulators, inline tracing, or on-the-fly analysis. Hardware simulators are reportedly much slower than any other technique mentioned here. For inline tracing, the cache behavior is analyzed based on trace data that is generated during program execution. The fastest results have been reported for on-the-fly analysis, a method that simulates the entire cache during program execution by instrumenting the program with calls to a trace routine. None of these methods analyze the cache prior to program execution.

4.8 Conclusion

The method of static cache simulation is introduced, which allows the prediction of a large number of cache references prior to program execution by taking advantage of path partitionings and the function-instance graph. A number of applications for this method are discussed in later chapters, ranging from faster instruction cache analysis to the analytical bounding of execution time by static analysis for real-time tasks. The benefit of static cache simulation for fast and accurate cache analysis is illustrated in the context of the applications.

Chapter 5
Code Instrumentation for Instruction Cache Analysis

This chapter discusses the generation of instrumentation code for on-the-fly analysis of instruction cache performance evaluation. This code is generated during the second phase of the static simulator based on the information of the first phase, the instruction categorization. While this work describes the generation of code for the purpose of cache analysis, any other on-the-fly analysis could be performed in its place by emitting different instrumentation code.

The code emitted by the compiler back end includes macro calls for each UP and for each call site. The simulator generates the corresponding macro bodies, produces tables to store local path states and frequency counters at run time, and provides other constant data structures for the final calculation of hits and misses. The code instrumentation includes the insertion of instructions at the unique transition of each UP to keep track of local state information and to record the frequency of executed instructions for this path and state. The generated code is later inserted into the assembly code of the compiled program.

When the instrumented program executes, the counters are incremented to provide the execution frequency of portions of code. In addition, the cache behavior is simulated for references that could not be predicted statically (so-called conflicts). The dynamic simulation employs a novel view of the cache by updating local state information associated with code portions. At the exit points of the program, an epilogue is inserted to call a library routine that calculates the total hits and misses from the gathered state-dependent frequencies. It will be shown in later chapters that this new method speeds up cache analysis over conventional trace-driven methods by an order of a magnitude. Excerpts of this chapter can be found in [47].

5.1 Introduction

Statistical sampling methods are often employed by profiling tools such as `prof` [65] or `gprof` [25, 26]. Yet, these tools only provide approximate measurements. On the other hand, code instrumentation results in accurate profiling measurements. For example, instruction frequency measurements can be obtained by inserting instructions that increment frequency counters into a program. The counters are typically associated with a basic block and incremented each time the basic block executes. The overhead induced by frequency measurements is less than a factor of two in execution time. This much lower overhead can be attributed to the fact that the execution order of instructions is irrelevant.

Conventionally, cache analysis is either based on a trace data file generated during program execution or by on-the-fly tracing, a method where the trace analysis is performed during program execution. The method discussed here is an on-the-fly analysis technique that employs short sequences of instrumentation code (inlined as macro calls) for each UP and avoids the generation of trace addresses all together. The compiler identifies the live registers at the instrumentation point. A set of unused registers is provided for the instrumentation code to avoid unnecessary saves and restores. If all registers are allocated, register spill code will be emitted around an instrumentation point.

The code instrumentation for cache analysis discussed in this chapter makes extensive use of frequency counters when instruction references are statically determined to be always cache hits, always cache misses, or first misses. For the remaining instruction references, state information is associated with code portions and is updated dynamically. The total hits and misses can be inferred from the state-dependent frequency counts after running the program.

This chapter is structured as follows: First, the merging of states associated with UPs is presented. The main portion of this chapter describes the code instrumentation step-by-step, first describing data structures such as shared path states and frequency counters, then code macros for calls and paths, and finally the first miss table. Afterwards, the calculation of hits and misses for instruction cache analysis is presented, followed by future work, related work, and conclusions.

5.2 Merging States

After decomposing the program into function instances and UPs, there still remain many lines that are analyzed to be in conflict with another line. It is inevitable to maintain information at run time to determine which line is currently cached and to update this information dynamically. This is achieved by maintaining a *path state*. A path state only reflects the conflicts local to the current path while a cache state comprises the global state of a cache memory.

Naively, a path state may be kept on the most specialized level (for every function instance and path). But this may require a considerable amount of interaction between UPs. In the worst case, the execution of a UP of some function instance would not only have to update its path state but every other path state conflicting with a line of this path and any function instance.

The number of function instances may grow exponentially with the dynamic nesting depth as can be seen in the representation as a function-instance graph in Figure 3.4. For programs with a call graph whose average branching factor is greater or equal to two, this can be infeasible if the height of the call graph becomes large and a leaf function (or a function close to a leaf) is called from many places.

Cache state information is therefore merged after simulation in two stages to comprise path states. First, the conflicts of the cache states of a UP of all instances of a function are merged into one *local path state*. Second, local path states of neighboring UPs, which share at least one instruction, are merged into one *shared path state* (SPS), to better utilize the storage and without any loss of information.

The former merging allows uniform instrumentation of code rather than distinguishing instances dynamically at every instrumentation point or replicating code for each instance. In both cases, the amount of dynamic simulation of conflicts is reduced. While an SPS only needs to maintain one state to keep track of conflicts dynamically, the state may comprise a wider range of values to combine all possible conflicts of the local path states. The local path states to be merged are therefore chosen with regard to their locality.

5.3 Shared Path States

For each SPS, a state field is generated in the state table (see Figure C.1 in Appendix C). These states are modified at run time by the macro code of UPs. The value of such a state denotes which lines are cached out of a set of conflicting lines. The initial value denotes the set of lines cached prior to the first execution of any corresponding UP. The value can be

used as an index into the frequency counter array of the current UP. Thus, state-dependent frequency counting can be performed by using the SPS as an index into the counter array and incrementing the corresponding counter. Furthermore, if an SPS is constant at run time (no conflicting lines), then the state field is omitted from the state table.
Example: In Figure 5.1, paths 1 and 2 have a shared path state, which is used to simulate the hits and misses of program lines a and b. The lines conflict with the reachable program lines x and y, which explains why they are categorized as conflicts. The SPS for path 1 and 2 has two bits (due to two conflicting program lines) to hold the possible encoding of cached program lines of the SPS (as shown in the figure). The state is updated on the execution of path 1 to include program line a. The execution of path 2 includes both a and b in the state, the execution of path 3 excludes b, and the execution of path 4 excludes both a and b. Simple bit manipulations suffice for these updates, as indicted by the pseudo code in the figure. The frequency counter, indexed by the incoming SPS, is incremented. Path 2 has an array of four frequency counters, corresponding to each possible value of the SPS. An increment of the first counter element corresponds to misses on line a and b, an increment of the second counter element indicates a miss on a and a hit on b, etc. The separate counter array for path 2 is incremented in the same manner. Neither the frequency counter increments for paths 3 and 4 nor their SPS are shown to simplify the example. ■

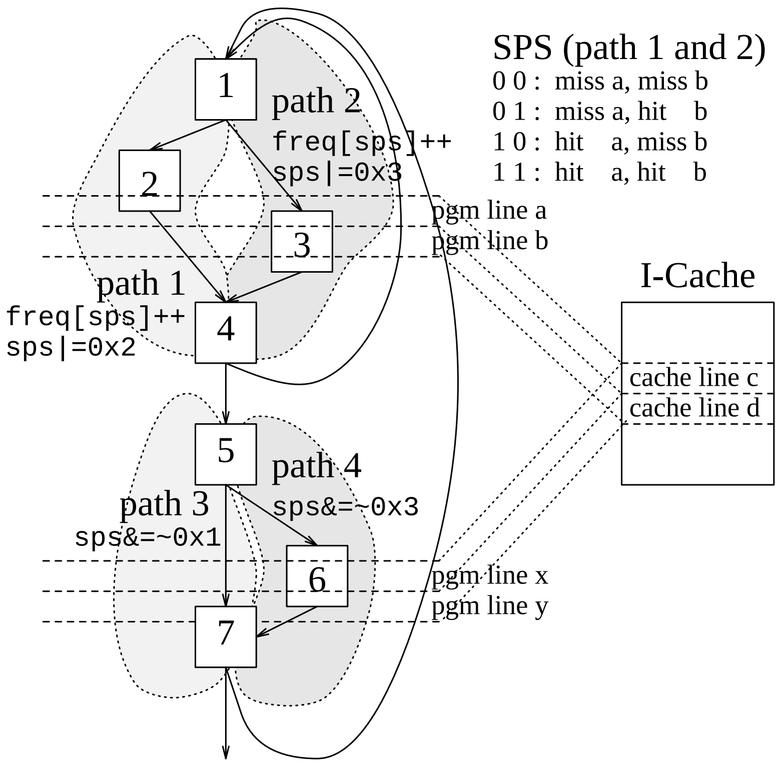


Figure 5.1: Frequency Counters Indexed by the SPS

5.4 Frequency Counters

For each UP of every function instance, an array of frequency counters is used to keep track of the execution frequency of the UP (see Figure C.2 in Appendix C). The size of the array is determined by the number of permutations of conflicting lines for an SPS. Since the size is growing exponentially with the number of conflicting lines, an alternate counter array with a constant size of two entries is provided for large numbers of conflicting lines in the SPS. There is a time/space trade-off between the two alternatives, which is discussed in the context of the path macros. In general, alternate methods of code instrumentation optimize special cases to reduce the instrumentation overhead.

5.5 Macros for Calls

Macro code is generated at call sites to pass the base address of the counter table for the callee's function instance as an additional parameter of the call. The function instance can thereby be identified by path macros (see Figure C.4 in Appendix C).

5.6 Macros for Paths

The code emitted for path macros increments the frequency counter indexed by the SPS, updates the SPS to reflect that the lines of the current path are now cached, and updates any other SPS of conflicting paths that can still be reached. If a different path *shares* a line (but not the SPS) with the current path, the line is marked as cached in the SPS of the conflicting path. Conversely, if a different path *conflicts* with the current SPS in a line, the line is marked as uncached in the SPS of the conflicting path.

Alternately, code is emitted to increment a general frequency counter for large SPSs. Since no counter array is generated for large SPSs, indexing into an array becomes obsolete. Rather, the SPS is first combined with an AND mask to single out the conflict lines of only the current path. Then, the number of remaining on-bits is counted and added to a second counter that accumulates references to conflicting lines resulting in misses. This alternate method requires less counter space but increases execution time by determining the number of set bits in a loop[1]. Figures C.5 and C.6 in Appendix C depicts examples of path macro code.

5.7 First Miss Table

If a path of a function instance contains a line that is classified as a first miss, an entry for this line is created in the first miss table (see Figure C.3 in Appendix C). If another path shares the same line and also categorizes this line as a first miss, this path's instance is also included in the same table entry. This table is used to adjust the total number of hits and misses as explained in the next section.

5.8 Calculation of Hits and Misses

The total number of hits and misses can be inferred from the state-dependent frequency counters and from the first miss table. This calculation is performed after running the instrumented program as part of its exit code. The calculation is independent from the number of SPSs or any other code generation parameters and can thus be hidden in a library routine that is linked with the instrumented program.

[1]RISC architectures as well as most CISC architectures do not provide a special bit-counting instruction.

5.8.1 Hits and Misses based on Frequency Counters

For each path of each function instance, the product of a frequency count and the number of always hits (misses) is added to the total number of hits (misses). First misses, weighted by the frequency, are also added to the total number of hits at this point.

The index into the counter array indicates the number of hits and misses for conflicting lines, which are then also multiplied by the corresponding frequency (see Figure C.2 in Appendix C). A zero index indicates that all conflicting lines are cached while the last index corresponds to misses of all conflicting lines.

Not all cache line configurations may be valid during the execution of the program for a given path and instance. In other words, the frequency count for an index should be zero if the SPS cannot occur. The actual implementation violates this rule to further improve the performance in the following manner. To minimize the amount of state changes during run time, a conflicting SPS is not updated if it can be determined at simulation time that the corresponding cache state cannot occur. This information is provided by the reaching states. Therefore, only a subset of counter indices may actually correspond to a valid cache configuration for a given path and instance. The number of conflicting lines is thus inferred from the array index combined with an AND mask with bits set in the position of valid cache lines. This method ensures that the lines corresponding to impossible SPSs are not counted.

If the number of states in the SPS was large and the alternate counting method was applied, then the always hits (misses) and first misses are still counted based on the frequency counter. The number of misses due to conflicts is readily available in one counter. The number of hits can be calculated as the total frequency times the number of conflict lines less the number of misses due to conflicts.

5.8.2 First Miss Adjustment

Since first misses were exclusively counted as hits with respect to the frequency, the hits and misses have to be adjusted. For each entry in the first miss table (see Figure C.3 in Appendix C), the counters of corresponding paths (and instances) are checked. If the frequency of one of the paths is greater than zero, the total number of hits is decremented while misses are incremented by one.

5.9 Future Work

The current code instrumentation could be improved in several ways. Some performance improvement could be achieved by applying code motion to path macros in the innermost loops when the number of iterations is known before the loop is entered and no alternate execution paths exist inside the loop.

The static analysis has already been extended based on a more detailed picture of the loop structure of the program. A refined notion of first misses refers to a miss of a program line on loop entry and consecutive hits inside the loop. The code instrumentation could take this situation into account during first miss adjustment. Thus, some conflicts can be replaced by first misses and first hits whose simulation requires less overhead during program execution. For a more detailed discussion, see Chapter 8.

5.10 Related Work

A technique called inline tracing can be used to generate the trace of addresses with much less overhead than trapping or simulation. Measurement instructions are inserted in the program to record the addresses that are referenced during the execution in a file or buffer. The program analysis may be performed concurrently on a buffered address trace to reduce the storage requirements to temporary trace buffers. Borg *et. al* [10] and Eggers *et. al.* [22] used this technique to obtain accurate measurements for the simulation of instruction and data caches. Whalley [68, 69] used on-the-fly analysis where a cache simulation function was called during program execution for each basic block. In the next chapter, their work will be discussed in more detail and contrasted with the performance results of this work.

5.11 Conclusion

An outline of inline code instrumentation for instruction cache analysis was presented. The instrumentation is based on the instruction categorization provided by static cache simulation. By reducing code instrumentation to simple frequency counting in many places and locally shared path states in other places, the overhead of the instrumentation code is kept surprisingly low at program execution time. This will be shown in more detail in the next chapter.

Chapter 6

Fast Instruction Cache Performance Analysis

This chapter evaluates the method of static cache simulation in conjunction with code instrumentation for instruction cache performance analysis. Measurements taken from a variety of programs show that static cache simulation speeds up cache analysis over conventional trace-driven methods by an order of a magnitude. Thus, cache analysis with static cache simulation makes it possible to analyze the instruction cache behavior of longer and more realistic program executions. Excerpts of this chapter can be found in [47].

6.1 Introduction

The method for instruction cache analysis discussed in this chapter uses static cache simulation to statically predict the cache behavior of a large number of instruction references. The method also uses the techniques for code instrumentation described in the last chapter. Thus, dynamic simulation is reduced to simple frequency counting for always hits, always misses, and first misses. Conflicts are simulated by updating local state information.

This chapter is structured as follows: First, related work in the area is reviewed. Then, the adaptation of static cache simulation for instruction cache performance analysis is discussed. Next, a quantitative analysis of this method is provided. Finally, future work and conclusions are presented.

6.2 Related Work

Evaluating cache performance has long been recognized as a challenging task to be performed in an efficient manner. Traces of the actual addresses referenced during the execution of programs have to be used to perform a realistic evaluation. The problem is that a realistic trace typically consists of millions of references. Evaluation of these traces can require excessive amounts of space and time when using simple approaches. For instance, a traditional approach is to generate the trace via trapping or simulation, write each address generated in the trace on disk, and analyze the trace via a separate program that reads the trace from disk and simulates the cache. Such an approach can easily slow the execution by a factor of a 1000 or more [55, 70, 33].

A technique called inline tracing can be used to generate the trace of addresses with much less overhead than trapping or simulation. Measurement instructions are inserted in the program to record the addresses that are referenced during the execution in a buffer. The program analysis is performed either concurrently on the buffered address trace to reduce the storage requirements to temporary trace buffers or it is performed after program execution on trace file data. Borg, Kessler, and Wall [10] modified programs at link time to write addresses to a trace buffer and these addresses were analyzed by a separate process. The time required to generate the trace of addresses was reduced by reserving five of the general purpose registers to avoid memory references in the trace generation code. Overhead rates of 8 to 12 times of the normal execution time were reported for the trace generation. Analysis of the trace was stated to require at least 10 times of the overhead of the generation of the trace (or about 100 times slower than normal execution time).

Eggers *et. al.* [22] also used the technique of inline tracing to generate a trace of addresses to a trace buffer, which was copied to disk by a separate process. They used several strategies for minimizing the overhead of generating the trace. First, they produced a subset of the addresses from which the other addresses could be inferred during a postprocessing pass. For instance, they only stored the first address in a sequence of contiguous basic blocks with a single entry point and multiple exit points. Rather than reserving a set of registers to be used for the trace generation code, they identified which registers were available and thus avoided executing many save and restore instructions. The trace generation overhead was accomplished in less than 3 times of the normal execution time. In addition, writing the buffers to disk required a factor of 10 times of the normal execution time. The postprocessing pass, which generates the complete trace from the subset of addresses stored, was much slower and produced about 3,000 addresses per second. No information was given on the overhead required to actually analyze the cache performance.

Ball and Larus [7, 41] also reduced the overhead of the trace generation by storing a portion of the trace from which the complete trace can be generated. They optimized the placement of the instrumentation code to produce the reduced trace with respect to a weighting of the control-flow graph. They showed that the placements are optimal for a large class of graphs. The overhead for the trace generation was less than a factor of 5. However, the postprocessing pass to regenerate the full trace required 19-60 times of the normal execution time.

Whalley [68, 69] evaluated a set of on-the-fly analysis techniques to reduce the time required to evaluate instruction cache performance. He linked a cache simulator to the programs, which were instrumented with measurement code to evaluate the instruction cache performance during the program's execution. The techniques he evaluated avoided making calls to the cache simulator when it could be determined in a less expensive manner that the reference was a hit. The overhead time for the faster techniques was highly dependent upon the hit ratio of the programs. He reported 15 times of the normal execution time for average hit ratios of 96% and 2 times of the normal execution time for hit ratios exceeding 99%. These faster techniques also required recompilation of the program when the cache configuration was altered.

6.3 Adaptation of Static Cache Simulation

Figure 6.1 depicts an overview of the tools and interfaces involved in instruction cache analysis using static cache simulation. The set of source files of a program are translated by a compiler. The compiler generates assembly code with macro entries for instrumentation and passes information about the control flow (*i.e.* a set of unique paths) of each source file to the static cache simulator. The static cache simulator performs the task of determining which instruction references can be predicted prior to execution time. It constructs the call graph of the program and the control-flow graph of each function based on the information provided by the compiler. The cache behavior is then simulated for a given cache configuration as described in Chapter 4. Furthermore, macro code for instrumenting the executable is generated together with tables to store cache information at run time. This output of the simulator is passed to the assembler, which translates the code generated by the compiler into instrumented object code. The linker combines these object files to an executable program and links in library routines that produce the final report of the cache analysis at run time (see Chapter 5). When the cache configuration changes, no recompilation is needed; only the static cache simulator, assembler, and linker have to be reinvoked.

In the current implementation, the instruction categorization has been slightly relaxed

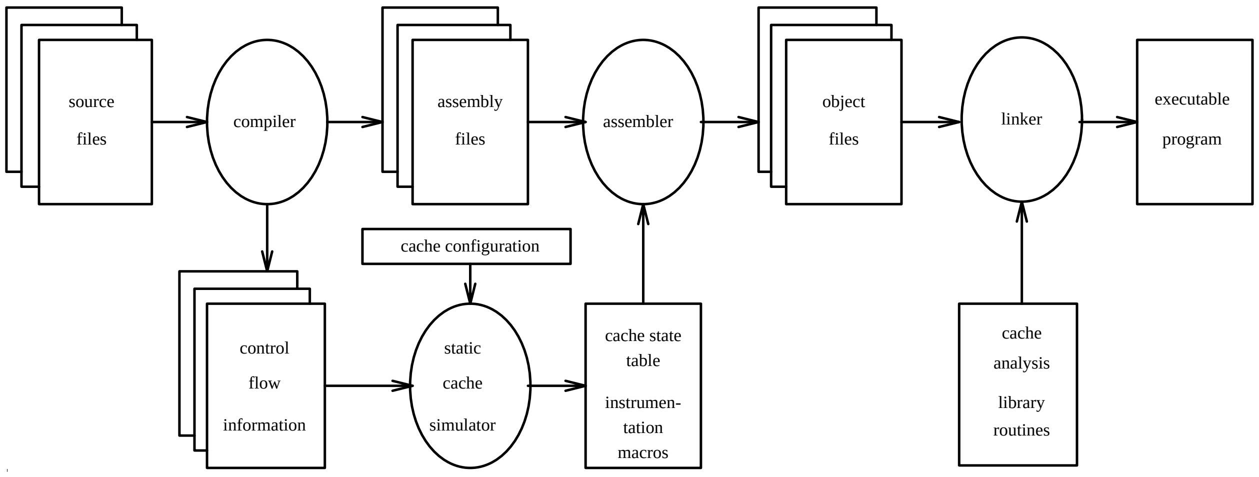
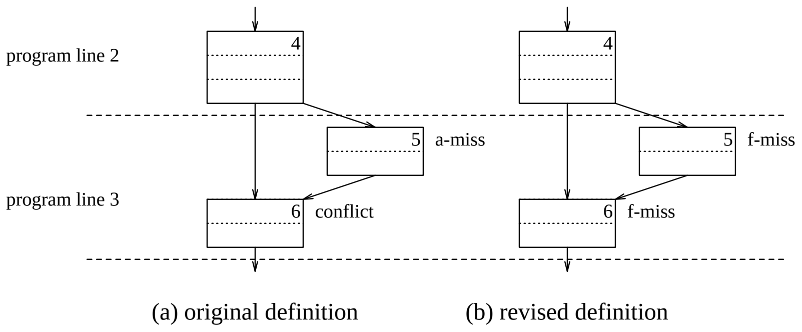


Figure 6.1: Overview of Cache Performance Analysis

to further reduce instructions categorized as conflicts. If an instruction was categorized as a conflict according to Definition 7 and the only other conflicting line m (where $m \to c$ and $m \in s$) is the invalid cache line "I", then all first references to the program line shall become first misses. The measurements of the next section show that this relaxation eliminates conflicts all together when the entire program fits into cache. For such cases, the cache simulation during program execution is reduced to only increments of frequency counters.
Example: Consider the example in Figure 6.2. In (a), the categorization is shown according



```
 "I" = invalid

  cache line 0 1 2 3 0 1 2 3 0 1
program line I I I I 0 1 2 3 4 5

 input(5) = [      I     2      ] reach(5) = [      3 4 5]
 input(6) = [      I     2 3    ] reach(6) = [        4 5]
```

Figure 6.2: Example for Adaptation of Categorization

to the original definition. Notice that the conflict instruction occurred as a result of the uncertainty introduced by the conditional control flow. If block 5 is not executed, a miss will occur for the first instruction of block 6. If block 5 is executed, the miss will occur in block 5 and the first instruction in block 6 will be a hit. The relaxed categorization shown in (b) recognizes both references as a group first miss since only the invalid line is conflicting with line 3 in `input(6)`. ■

6.4 Measurements

This section evaluates the benefits of instruction cache analysis via static cache simulation. Cache measurements were obtained for user programs, benchmarks, and UNIX utilities listed in Table 3.1 of Chapter 3. The table has already been discussed except for column 5. Column 5, instructions in the tree, refers to the static number of instructions in the program after expanding the call graph to a function instance tree. The measurements in the next section are based on this number.

All measurements were produced by modifying the back-end of an optimizing compiler VPO (Very Portable Optimizer) [8] and by performing static cache simulation. The simulation was performed for the Sun SPARC instruction set, a RISC architecture with a uniform instruction size of one word (four bytes). The parameters for cache simulation included direct-mapped caches with sizes of 1kB, 2kB, 4kB, and 8kB. The cache line size was fixed at 16 bytes (4 instructions). No context switches were simulated.

6.4.1 Static Analysis

This section describes the analysis that was performed statically on the test programs. Table 6.1 shows the percentage of always hits, always misses, first misses, and conflicts out of the

Table 6.1: Static Results for Cache Performance Evaluation

	1kB cache				2kB cache			
Name	Hit	Miss	Firstmiss	Conflict	Hit	Miss	Firstmiss	Conflict
cachesim	70.83%	6.99%	0.70%	21.48%	71.54%	5.18%	1.84%	21.44%
cb	79.02%	2.35%	0.00%	18.63%	80.56%	0.95%	0.00%	18.49%
compact	70.06%	4.96%	0.12%	24.86%	70.15%	1.95%	0.12%	27.77%
copt	70.82%	7.41%	7.03%	14.74%	71.28%	5.73%	15.28%	7.72%
dhrystone	70.03%	10.71%	7.30%	11.96%	70.81%	3.73%	25.47%	0.00%
fft	74.07%	4.85%	16.42%	4.66%	75.75%	3.92%	20.34%	0.00%
genreport	71.63%	9.75%	5.64%	12.98%	72.45%	9.02%	6.67%	11.86%
mincost	72.75%	9.96%	1.14%	16.15%	74.66%	5.91%	4.60%	14.83%
sched	67.52%	5.06%	0.09%	27.32%	67.76%	2.48%	0.09%	29.67%
sdiff	68.93%	12.06%	0.90%	18.11%	69.34%	9.81%	1.88%	18.97%
tsp	72.61%	13.50%	3.88%	10.01%	73.00%	9.95%	10.21%	6.85%
whetstone	75.70%	12.84%	0.24%	11.22%	76.60%	8.94%	0.24%	14.22%
average	72.00%	8.37%	3.62%	16.01%	72.83%	5.63%	7.23%	14.32%

	4kB cache				8kB cache			
Name	Hit	Miss	Firstmiss	Conflict	Hit	Miss	Firstmiss	Conflict
cachesim	72.05%	4.52%	13.78%	9.65%	72.61%	3.54%	23.78%	0.07%
cb	80.81%	0.35%	15.81%	3.04%	80.85%	0.03%	19.12%	0.00%
compact	70.27%	0.46%	10.26%	19.00%	70.71%	0.46%	28.83%	0.00%
copt	71.81%	5.19%	22.99%	0.00%	71.81%	5.19%	22.99%	0.00%
dhrystone	70.81%	3.73%	25.47%	0.00%	70.81%	3.73%	25.47%	0.00%
fft	75.75%	3.92%	20.34%	0.00%	75.75%	3.92%	20.34%	0.00%
genreport	72.62%	8.57%	8.36%	10.44%	72.88%	8.12%	10.87%	8.13%
mincost	75.48%	2.91%	21.43%	0.18%	75.52%	2.91%	21.57%	0.00%
sched	67.80%	1.92%	2.35%	27.93%	67.95%	1.24%	30.81%	0.00%
sdiff	69.41%	9.55%	18.28%	2.75%	69.44%	9.44%	21.12%	0.00%
tsp	73.32%	5.10%	21.45%	0.13%	73.39%	5.10%	21.51%	0.00%
whetstone	77.08%	1.44%	17.10%	4.38%	77.98%	0.48%	21.54%	0.00%
average	73.10%	3.97%	16.47%	6.46%	73.31%	3.68%	22.33%	0.68%

total number of instructions in the function instance tree. It can be seen that a large number of hits and misses can be predicted statically.

The number of always hits is slightly above 70% in average and does not change significantly as the cache size is varied. Always hits occur mostly due to subsequent references within a program line (past the first reference) that do not depend on the cache size. The slight variation of always hits is mainly due to multiple function instances. Consider a first call to a function that will cache the function's program lines. Subsequent calls always result in hits for these program lines, given a sufficiently large cache capacity such that no conflicting lines could be executed between the two function calls.

The number of first misses increases for larger caches while conflicts and misses decrease at the same time. This can be explained as follows. First misses occur when a program line without any conflicts is placed in cache on its first reference and remains in cache thereafter. For smaller caches, program lines tend to conflict with one another more frequently. As the programs begin to fit into cache, fewer program lines are in conflict. In the worst case, only every sixth instruction is statically predicted as a conflict and will have to be simulated at execution time. At best, there are virtually no conflicts and almost the entire runtime simulation can be performed using efficient frequency counters.

6.4.2 Dynamic Analysis

Table 6.2 summarizes the dynamic measurements taken for the test programs. For each given cache size and program execution, our method produced exactly the same number of hits and misses that were obtained by traditional trace-driven cache analysis. As the cache size increases, the hit ratio (column 2) increases as well. Column 3 and 4 represent the quotient of the execution time of a program with instrumentation over the execution time for the same program without instrumentation. Column 3 refers to a trace-driven method that has been optimized such that the cache simulator is only called once per basic block[1]. Column 4 refers to the analysis via static cache simulation. The percentage of conflicts (out of all instruction references) simulated at execution time is shown in the last column.

With the traditional trace-driven method, the execution time of instrumented programs averages about 17 times slower than the execution time of regular programs without instrumentation. The overhead for the new method using static cache simulation is much lower, only a factor of 1.2 to 2.2.[2] This overhead depends slightly on the ratio of program size and cache size. The variation can be explained as follows.

Let the *conflict degree* be the number of program lines that map into the same cache line. This is a useful term to characterize the size of shared path states (SPSs) and the execution overhead due to order-dependent simulation. For small cache sizes, the conflict degree is larger and there are more always misses due to capacity misses. As the cache size increases, capacity misses and the conflict degree of program line decrease. They are replaced by first misses. With a diminishing number of conflicts for large caches, the size of SPSs decreases as the cache size increases. In other words, fewer and fewer conflicting program lines map into the same cache lines. Consequently, less instrumentation code to update conflicting SPSs is needed. Finally, for large caches, hardly any conflicts remain. Thus, the cache simulation at execution time can be reduced to simple frequency counting, which imposes a much lower

[1]We used a traditional trace-driven method similar to "Technique B" in [69] but the new method was probably finer tuned.

[2]Sometimes, the instrumented code ran faster than the uninstrumented program, *i.e.* the ratio was smaller than 1. These results were reproducible. They may be caused by the different placement of code due to instrumentation, resulting in fewer misses for frequently executed loops.

overhead than conventional cache simulation. To summarize this discussion, it is observed that the new method requires slightly more execution overhead for small caches than for large caches since more SPSs have to be updated dynamically.

Table 6.2: Dynamic Results for Cache Performance Evaluation

	1kB cache				2kB cache			
Name	Hit Ratio	Trace	SSim	Conflict	Hit Ratio	Trace	SSim	Conflict
cachesim	77.19%	8.52	1.52	34.12%	86.08%	7.98	1.33	38.01%
cb	93.84%	35.18	3.65	30.67%	99.06%	32.63	2.85	31.91%
compact	92.90%	22.42	2.33	21.34%	96.79%	22.05	1.87	20.94%
copt	93.64%	16.04	1.59	30.00%	98.10%	16.46	1.24	10.88%
dhrystone	83.73%	19.35	1.28	16.01%	100.00%	14.88	0.91	0.00%
fft	99.95%	5.76	0.93	8.80%	100.00%	5.63	0.92	0.00%
genreport	97.49%	14.39	2.08	25.74%	98.10%	12.66	1.70	24.32%
mincost	89.08%	22.32	2.06	30.67%	95.68%	22.46	1.80	25.41%
sched	96.41%	31.38	4.49	41.99%	99.75%	24.59	2.67	42.67%
sdiff	97.61%	27.69	3.60	28.40%	99.38%	28.45	2.77	28.27%
tsp	86.98%	5.62	1.12	17.63%	96.94%	5.16	1.04	13.24%
whetstone	100.00%	13.50	1.35	23.56%	100.00%	13.20	1.52	25.39%
average	92.40%	18.51	2.17	25.74%	97.49%	17.18	1.72	21.75%

	4kB cache				8kB cache			
Name	Hit Ratio	Trace	SSim	Conflict	Hit Ratio	Trace	SSim	Conflict
cachesim	99.22%	7.45	1.15	14.02%	99.98%	6.45	1.04	1.67%
cb	99.83%	29.36	2.15	7.50%	99.99%	28.09	1.68	0.00%
compact	99.82%	20.99	1.43	12.22%	100.00%	18.40	0.88	0.00%
copt	99.99%	13.37	1.03	0.00%	99.99%	13.08	0.98	0.00%
dhrystone	100.00%	14.96	0.92	0.00%	100.00%	15.03	0.92	0.00%
fft	100.00%	5.76	0.89	0.00%	100.00%	5.75	0.91	0.00%
genreport	98.11%	12.52	1.72	24.18%	99.93%	12.67	1.62	19.41%
mincost	99.99%	17.49	1.10	0.05%	99.99%	17.07	1.10	0.00%
sched	99.90%	20.62	1.85	36.82%	99.96%	22.16	1.42	0.00%
sdiff	99.99%	25.13	1.36	4.30%	99.99%	23.42	1.30	0.00%
tsp	99.99%	4.43	0.99	0.00%	99.99%	4.26	0.96	0.00%
whetstone	100.00%	11.11	1.10	12.15%	100.00%	11.10	1.01	0.00%
average	99.74%	15.27	1.31	9.27%	99.99%	14.79	1.15	1.76%

In general, the new method outperforms conventional trace-driven cache simulation by almost an order of a magnitude without compromising the accuracy of measurements. Even the best results published in [69] required an overhead factor of 4-15 over uninstrumented code for hit ratios between 96% and 99%. This highly tuned traditional method required a recompilation pass for better instrumentation. Under all conditions, the new method using static cache simulation outperforms the best traditional trace-driven methods published.

6.5 Future Work

The definition of first misses can still be improved. The static analysis could be extended based on a more detailed picture of the loop structure of the program. A refined notion of first misses could then refer to a miss of a program line on loop entry and consecutive hits inside the loop. This would be valuable when a program line will not be cached at loop entry and no other conflicting program line is part of the loop.

6.6 Conclusion

A new method to evaluate instruction cache performance was designed and implemented. The cache performance of programs for various cache configurations can be obtained without recompiling the analyzed program. No special operating system support or dedicated registers are required. The new method outperforms conventional trace-driven cache simulation by an order of a magnitude without any loss of accuracy of the measurements. By making extensive use of static cache simulation and reducing code instrumentation to simple frequency counting in many places, this method reduces the execution overhead of analyzed programs to a factor of 1.2 to 2.2. In addition, different cache sizes and resulting hit ratios have little influence on the overhead. In summary, instruction cache analysis via static cache simulation is a general method to quickly obtain accurate measurements.

Chapter 7
Predicting Instruction Cache Behavior

It has been claimed that the execution time of a program can often be predicted more accurately on an uncached system than on a system with cache memory [27, 60, 43]. Thus, caches are often disabled for critical real-time tasks to ensure the predictability required for scheduling analysis. This work shows that instruction caching can be exploited to gain execution speed without sacrificing predictability. This work takes advantage of static cache simulation to statically predict the caching behavior of a large portion of the instruction cache references of a program. In addition, a *fetch-from-memory* bit is added to the instruction encoding that indicates whether an instruction shall be fetched from the instruction cache or from main memory. This bit-encoding approach provides a significant speedup in execution time (factor 3-8) over systems with a disabled instruction cache without any sacrifice in the predictability of worst-case execution time. The fetch-from-memory bit facilitates the bounding of execution time by conventional timing tools. Even without bit-encoding, the ability to predict the caching behavior of a large percentage of the instruction references is very useful for providing tight worst-case execution time predictions of large code segments on machines with instruction caches but requires more sophisticated analysis by a timing tool. Excerpts of this chapter can be found in [50].

7.1 Introduction

Predicting the execution time of programs or code segments is a difficult task. Yet, in the context of hard real-time systems, it is essential to provide a schedule for tasks with known deadlines. Thus, tasks have to be analyzed to determine their best-case execution time (BET) and worst-case execution time (WET). The following problems have to be addressed to predict the execution time of a task or program:

- The number of loop iterations needs to be known prior to execution. It is often required that the maximum number of iterations be provided by the programmer [36].
- The possible execution paths in the control flow have to be analyzed to predict both BET and WET.
- Architectural features have to be taken into account (*e.g.* pipeline stalls).

In the context of real-time systems, caches have been regarded as a source of unpredictability, which conflicts with the goal of making the execution of tasks deterministic [60]. For a system with an instruction cache as a primary (on-chip) cache, the execution time of an instruction can vary greatly depending on whether the given instruction is in cache or not. In addition, context switches and interrupts may replace the instructions cached by one task with instructions from another task or an interrupt handler. As a result, it has been common practice to simply disable the cache for sections of code when predictability was required [60].

This work shows that it is possible to predict some cache behavior with certain restrictions. Let a task be the portion of code executed between two scheduling points (context switches). When a task starts execution, the cache memory is assumed to be invalidated. During task execution, instructions are gradually brought into cache and often result in many hits and misses that can be predicted by static cache simulation. Furthermore, a slight change in the architecture in conjunction with the simulator's analysis allows, without loss of predictability, significantly faster execution time than on systems with a disabled instruction cache.

This chapter is structured as follows: The next section reviews related work in the area. Afterwards, the bit-encoding approach is introduced, which can exploit caches for real-time systems. In the following, the bit-encoding approach is contrasted with uncached and regular cached systems on an abstract level. Then, the adaptation of static cache simulation for this application is discussed. Furthermore, a quantitative analysis of both static cache simulation and the bit-encoding approach is provided. Finally, future work and conclusions are presented.

7.2 Related Work

The problem of determining the execution time of programs has been the subject of some research in the past. Sarkar [59] suggested a framework to determine both average execution time and its variance. His work was based on the analysis of a program's interval structure and its forward control flow. He calculated a program's execution time for a specific set of input data by using a description of the architecture and the frequency information obtained by incrementing counters during a profiling run. He assumed that the execution order of instructions does not affect this calculation. Thus, his method cannot capture the impact of caching on execution time.

For real-time systems, several tools to predict the execution time of programs have been designed. The analysis has been performed at the level of source code [54], at the level of intermediate code [52], and at the level of machine code [28]. Only Harmon's tool took the impact of instruction caches into account for restrictive circumstances, *i.e.* only for small code segments that entirely fit into cache.

Niehaus outlined how the effects of caching can be taken into account in the prediction of execution time [53]. He suggested that caches be flushed on context switches to provide a consistent cache state at the beginning of each task execution. He provided a rough estimate of the benefit of caches for speedup and tried to determine the percentage of instruction cache references that can be predicted as hits. The level of analysis remained at a very abstract level though, as it only dealt with spatial locality for sequential execution and some temporal locality for simple loops. No general method to analyze the call graph of a task and control flow for each function was given.

A few attempts have been made to improve the predictability of caches by architectural modifications to meet the needs of real-time systems. Kirk [35] outlined such a system that relied on the ability to segment the cache memory into a number of dedicated partitions, each of which can only be accessed by a dedicated task. But this approach introduced new problems such as exhibiting lower hit ratios due to the partitioning and increasing the complexity of scheduling analysis by introducing another resource (cache partitioning) as an additional degree of freedom in the allocation process.

Lee *et. al.* proposed a modified instruction pipeline where instructions are prefetched along the worst-case execution path. They reported a 45% improvement of the predicted worst-case execution time under this scheme. This work shows that better results for the timing prediction can be achieved by using instruction caches, with or without architectural modifications.

Other suggested architectural modifications often dedicate a bit in the instruction encoding that is used by the compiler to affect the cache behavior. McFarling [45] used such an approach to exclude instructions from cache that were not likely to be in cache on subsequent references. Chi and Dietz [14] introduced a data cache bypass bit on load and store instructions, which, when set, indicates that the processor should go directly to memory (without caching the value as a side-effect) or goes to the cache when clear. Their idea is to improve execution speed by keeping data values either in registers or in cache, thus avoiding storage mirroring among the faster components in the memory hierarchy (registers and data caches). Their work emphasizes instruction caches rather than data caches. In contrast to McFarling's study and the work by Chi and Dietz, the work described here is primarily focused on the predictability of instruction caching and secondarily on execution speed.

7.3 Bit-Encoding Approach

Based on the categorization of instruction references introduced in Definition 7, a bit-encoding approach has been formulated. The intention of this approach is to provide better performance than uncached systems (as currently used in real-time systems) and better predictability over conventional caches with a moderate sacrifice in execution speed. The bit-encoding approach allows conventional timing tools to provide tight execution time bounds in the presence of instruction caches with this hypothetical architectural feature. A *fetch-from-memory* bit is encoded into the instruction format by dedicating a single bit position. If the bit is set in an instruction and the instruction is in cache, then the instruction will be fetched from main memory. If the bit is not set, then the instruction will be fetched from cache.

7.3.1 Operational Semantics of the Fetch-from-Memory Bit

During each cache reference, the fetch-from-memory bit is evaluated in parallel with the tag comparison, as shown in Appendix D. The following logic is used to resolve instruction fetch requests:

- If the cache access results in a miss, then the corresponding program line is fetched from main memory taking n cycles and the fetch-from-memory bit is ignored. (The bit would not be available anyway until the instruction is fetched.)
- If the tag comparison matches and the cache line is valid, then the effect depends on the evaluation of the fetch-from-memory bit.
  - If the bit is clear, then the processor is directed to use the instruction without delay.
  - If the bit is set, then the corresponding program line is fetched from main memory taking n cycles.

In the last subcase, a memory fetch is performed although the program line already resides in cache. If the effect of such a memory fetch is only simulated to reduce bus contention, it would be unpredictable whether an actual memory fetch occurs or not. Thus, bus contention may or may not occur. The current semantics forces a memory access such that bus contention can be predicted for any memory reference with a fetch-from-memory bit set if a data reference occurs at the same time.

The fetch-from-memory bit is set whenever the static cache simulation categorizes an instruction as a conflict or an always miss. Otherwise, the bit is cleared. This is straight forward for always hits. For first misses, on the other hand, the cache look-up fails on the first reference and the program line is fetched from main memory. For any subsequent references to this address, the instruction is found in cache with the bit clear resulting in a cache hit and a one cycle access time. Thus, bit-encoding takes advantage of first-miss instructions.

If an instruction is in a function that has multiple instances and the instruction has different categories in the different instances, then the static simulator must decide whether or not to set the fetch-from-memory bit. Currently, the static simulator conservatively decides to fetch from memory if one or more instances categorize the instruction as a miss or a conflict. Otherwise, the bit is cleared[1].

7.3.2 Speedup

In this section, the execution time w.r.t. instruction fetch overhead is analyzed. Other factors, *e.g.* data references to main memory, may add to the execution time but should not be adversely effected by the benefits of instruction caching.

For any uncached system, let the fetch time of one instruction be n cycles. Furthermore, let i be the number of instructions executed. Then, a lower bound for the time for this execution is

$$t_{uncached} = i * n \text{ cycles.} \tag{7.1}$$

For a cached system, let $i = h + m$ be the number of instructions executed where h and m are the number of hits and misses respectively. Assume a cache look-up penalty of one cycle [31, 29]. Since a cache look-up always has to be performed before it can be decided whether the program line associated with an instruction has to be fetched from main memory, the lower bound for an execution in a cached system is

$$t_{cached} = h + m * (n + 1) \text{ cycles.} \tag{7.2}$$

For the bit-encoded cached system, let $i = h' + m'$ be the number of instructions executed where h' and m' are the number of instructions fetched from cache and memory respectively[2]. Then, a lower time bound can be given as

$$t_{bit\_encoded} = h' + m' * (n + 1) \text{ cycles.} \tag{7.3}$$

There is both spatial and temporal locality inherent in the code of almost all programs. For instance, assume that a cache line consists of multiple instructions. The first reference to an instruction in such a line may cause a miss. But if instructions are executed sequentially, consecutive references to instructions of the same line will result in hits. Also, assume that some portion of the code executes in a loop does not conflict with any other program lines accessed by the loop. Subsequent references to this code in the same execution of the loop will also result in hits. Based on this observation, the following inequalities can be assumed for a typical execution:

$m \ll h$, $m' \ll h'$, and $h' < h$.

[1]It is possible in such a situation that the merged instruction could be safely classified as a first miss and have its bit cleared. An example of this situation is the first instruction in block 8 of Figure 4.1. In future work, the control flow could be analyzed to recognize these situations.

[2]$h'$and m' are approximately the same as the number of instructions executed with the fetch-from-memory bit clear and set respectively with the exception of first misses, which are counted as misses on the first reference and hits on subsequent references.

In summary, the following relation holds for an execution on the average.

$$t_{cached} < t_{bit\_encoded} < t_{uncached} \tag{7.4}$$

7.4 Adaptation of Static Cache Simulation

For the performance analysis of instruction caches, the compiler emits information about the control flow of each function at the basic block level rather than at the level of unique paths to facilitate the encoding of the fetch-from-memory bit. The static cache simulator is adapted in the same way as discussed in Chapter 6. The bit-encoding approach uses the same framework but adjusts the categorization to simulate the effect of the fetch-from-memory bit. If the bit is set, then the effect on the timing is equivalent to a cache miss. Thus, a simple remapping of the categorization can be used to determine the performance exhibited by the bit-encoding approach.

$$\text{new-category } (i_k) = \begin{cases} \text{always-miss} & \text{if } category(i_k) \in \{\text{always-miss,conflict}\} \\ \text{always-hit} & \text{if } category(i_k) = \text{always-hit} \\ \text{first-miss} & \text{if } category(i_k) = \text{first-miss} \end{cases}$$

The mapping of conflicts into always misses simulates the overhead of a memory fetch on each conflict instruction, regardless of its original caching behavior (hit or miss at execution time). First misses will still be adjusted as described in Chapter 5. Thus, the first reference to a first miss will be simulated as a cache miss while all consecutive references are counted as cache hits. Since paths are comprised of basic blocks for this application, paths cannot overlap[3]. On the other hand, different categorizations may exist for distinct function instances of the same function. In such a case, the worst-case scenario is assumed:

- if any instance classifies the instruction as an always miss, then the final category will be an always miss; otherwise,
- if any instance classifies the instruction as a first miss, the final category will be a first miss; otherwise,
- the final category will be an always hit.

This remapping of instruction categories provides a framework to evaluate the performance impact of the fetch-from-memory bit. It will be used in the next section to provide a quantitative analysis of a set of test programs.

7.5 Analysis

This section analyzes the benefit of predicting the behavior of instruction cache references. Cache measurements were obtained for user programs, benchmarks, and UNIX utilities listed in Table 3.1. The measurements were produced by modifying the back-end of an optimizing compiler VPO (Very Portable Optimizer) [8] and by performing static cache simulation. The compiler back-end provided the control-flow information for the static simulator. It also produced assembly code with instrumentation points for instruction cache simulation.

[3]Architectural features may present an exception to this general rule. For example, consider RISC architectures with an instruction d in the delay slot of a branch b. Both b and d are part of a basic block. Yet, if d is also the target of another branch instruction, d will also be the first instruction of another basic block.

The cache simulation for traditional caches was based on the instruction categorization by the static simulator and has been validated by comparison with another trace-driven cache simulator. The validity of the bit-encoding approach was derived from mapping the instruction categories into the values for the fetch-from-memory bit. The assembly code was generated for the Sun SPARC instruction set, a RISC architecture with a uniform instruction size of one word (four bytes).

The parameters for cache simulation included direct-mapped caches with sizes of 1kB, 2kB, 4kB, and 8kB (see column 1 in Tables 7.1 and 7.2). The cache line size was fixed at 4 words. The size of the programs varied between 500 and 4500 instructions (5kB – 18kB, see column 3 of Table 7.1). This provided a range of measurements from capacity misses dominating for small cache sizes to entire programs fitting in cache for large cache sizes. The number of instructions executed for each program comprised a range of 1 to 19 million using realistic input data for each program (see column 3 of Table 7.2).

7.5.1 Static Analysis

Static cache simulation classifies instructions into categories based on the predicted cache behavior. Table 7.1 shows the static number of instructions for each program (column 3)

Table 7.1: Static Results: Call Graph (CG) & Function-Instance Graph (FIG)

		number of		cache prediction				
Cache		instructions		CG	FIG			
Size	Name	CG	FIG	memory	always hit	always miss	first miss	conflict
	cachesim	2,115	9,397	28.07%	69.30%	8.23%	0.74%	21.73%
	cb	1,242	7,017	31.08%	75.70%	2.85%	0.00%	21.45%
	compact	1,478	2,173	29.84%	69.67%	5.52%	0.18%	24.62%
	copt	1,037	1,152	21.99%	71.01%	7.73%	7.03%	14.24%
	dhrystone	479	549	23.17%	69.76%	11.29%	6.56%	12.39%
1kB	fft	492	528	9.55%	74.05%	4.92%	16.29%	4.73%
	genreport	4,430	7,060	19.77%	70.64%	11.30%	6.18%	11.88%
	mincost	1,112	1,657	28.33%	72.24%	8.57%	1.39%	17.80%
	sched	2,068	3,378	32.88%	66.55%	5.98%	0.15%	27.32%
	sdiff	1,822	10,407	28.06%	67.44%	12.28%	1.08%	19.21%
	tsp	1,181	1,236	22.95%	72.73%	13.59%	4.37%	9.30%
	whetstone	1,204	1,485	26.62%	75.69%	11.65%	0.27%	12.39%
1kB	average	1,555	3,837	25.19%	71.23%	8.66%	3.69%	16.42%
2kB	average	1,555	3,837	21.18%	72.09%	5.88%	7.28%	14.75%
4kB	average	1,555	3,837	11.35%	72.40%	4.36%	16.64%	6.60%
8kB	average	1,555	3,837	4.73%	72.61%	4.03%	22.77%	0.59%

and the number of instructions associated with all function instances when the call graph is converted into a function-instance graph (column 4). Column 5 denotes the percentage of instructions in the call graph that have the fetch-from-memory bit set. Columns 6 to 9 show the percentage of instructions in the function-instance graph for each category as determined by the static simulator. Notice that the cache behavior could be predicted statically for 84-99% of the instructions, depending on the ratio of program size and cache size. The remaining 1-16% are due to conflicts.

7.5.2 Dynamic Analysis

Table 7.2 illustrates the dynamic behavior of three systems: an uncached system (simulating

Table 7.2: Dynamic Results for Cache Predictability

cache Size	Name	Instructions executed	hit ratio bit-enc.	hit ratio cached	conflicts cached	exec time uncached	% of exec time bit-enc.	% of exec time cached
1kB	cachesim	2,995,817	65.70%	77.19%	28.52%	26,962,353	45.41%	33.92%
	cb	3,974,882	67.24%	93.84%	31.08%	35,773,938	43.87%	17.27%
	compact	13,349,997	67.12%	92.90%	32.45%	120,149,973	43.99%	18.21%
	copt	2,342,143	68.56%	93.64%	28.93%	21,079,287	42.55%	17.47%
	dhrystone	19,050,093	77.95%	83.73%	15.75%	171,450,837	33.16%	27.38%
	fft	4,094,244	91.17%	99.95%	8.80%	36,848,196	19.94%	11.16%
	genreport	2,275,814	74.64%	97.49%	24.58%	20,482,326	36.47%	13.63%
	mincost	2,994,275	67.35%	89.08%	28.06%	26,948,475	43.76%	22.03%
	sched	1,091,755	67.21%	96.41%	32.15%	9,825,795	43.90%	14.70%
	sdiff	2,138,501	71.20%	97.61%	28.40%	19,246,509	39.92%	13.50%
	tsp	3,004,145	72.01%	86.98%	22.06%	27,037,305	39.10%	24.13%
	whetstone	8,520,241	71.57%	100.00%	23.78%	76,682,169	39.54%	11.11%
1kB	average	5,485,992	71.81%	92.40%	25.38%	49,373,930	39.30%	18.71%
2kB	average	5,485,992	77.81%	97.49%	21.14%	49,373,930	33.30%	13.62%
4kB	average	5,485,992	90.73%	99.74%	9.12%	49,373,930	20.38%	11.37%
8kB	average	5,485,992	98.15%	99.99%	1.76%	49,373,930	12.97%	11.13%

a disabled instruction cache), a cached system with the bit-encoding approach, and a conventional cached system. Column 3 indicates the number of instructions executed. The hit ratio (percentage of cache hits of all instruction references) is shown for the bit-encoded system in column 4 and for conventional caches in column 5. Column 6 shows the percentage of executed instruction references that were classified as conflicts on a cached system. Column 7 indicates the estimated execution time in cycles for an uncached system. The percentage of cycles required for a bit-encoded system (column 8) and for a conventional cached system (column 9) are compared to an uncached system. The execution time is calculated based on the equations 7.1, 7.2, and 7.3 for $n = 9$.[4]

The bit-encoding approach results in lower hit ratios (72-98%) than on a conventional cached system (92-99%). Yet, caches are often disabled for critical real-time tasks to provide the predictability required by scheduling analysis. Thus, the bit-encoding approach should be compared to an uncached system. The bit-encoding method requires only 13-39% of the cycles used by the uncached systems. This provides a speedup of programs by a factor of 3-8 without sacrificing the predictability of a program's execution time. The result represents the improvement over critical real-time tasks that require caches to be disabled. The results improve considerably as the cache size increases and entire programs fit into cache. The execution time required for a conventional cached system is only about 14% of an uncached system, but the predictability also decreases to the point where it becomes insufficient for scheduling analysis of critical tasks. This can be explained as follows:

Conflicts correspond to the instructions whose cache behavior could not be predicted prior to execution in a conventional cached system. The dynamic percentage of conflict references is higher than the static percentage given in Table 7.1 since conflicts typically

[4]The latency for a memory fetch is assumed to be $n = 9$ cycles, a cache look-up takes one cycle, and thus a cache hit also consumes one cycle while a miss takes $n + 1 = 10$ cycles. These assumptions are described as realistic by other researchers [31, 29]. A memory fetch in an uncached system fetches exactly one instruction while a memory fetch in a cached system fetches a line of 4 instructions. Fetching a line of multiple instructions is typically accomplished through a wider bus between cache and main memory for a cached system.

occur in loops. Since 25% of the instructions executed were conflicts, the execution time of programs cannot be predicted as tightly in conventional cached systems with traditional timing tools. However, more recent work (combining the static simulator with a timing tool) shows that the instruction categorization of the static simulator may be used by a more sophisticated timing tool to provide tight worst-case execution time predictions with a 4-9 times speedup over uncached system using a conventional instruction cache [5]. This will be discussed in more detail in the next chapter.

7.6 Future Work

Further work focuses on integrating the method of static cache simulation with a tool that estimates a program's best-case execution time (BET) and worst-case execution time (WET) [28, 5]. Using the information provided by static cache simulation, the BET and WET can be based on the categorization of instructions. This relieves the time-estimation tool from having to simulate all possible cache states. The instruction categorization is refined to provide a separate category for each loop level, thereby providing the base for tight execution time predictions.

With the bit-encoding approach, a traditional tool predicting the execution time can perform the same type of analysis and provide estimations for both BET and WET. But the execution time predictions can be tighter since the caching behavior is fully predictable. Instructions classified as always hits can be assumed to require one cycle, and always misses or conflicts can be estimated to take $n+1$ cycles. For a first miss, the tool could distinguish between the first reference ($n+1$ cycles) and any subsequent references (one cycle) by simply tagging first miss instructions that have been encountered. A traditional timing tool should be easily modified to take the effect of bit-encoding into account. The resulting execution time estimate will be as tight as for uncached systems since the estimation of the fetch cost accurately represents the number of cycles taken for an instruction in any category. There is no uncertainty with respect to the effect of an instruction classified as conflict, the fetch will always take $n+1$ cycles.

7.7 Conclusion

Cache memories have often been disabled for critical real-time tasks to provide sufficient predictability for scheduling analysis. This chapter shows that the behavior of instruction cache references can be predicted to a large extent prior to the execution of a program via the method of static cache simulation. The cache simulator uses information provided by the back-end of a compiler to statically predict the cache behavior of 84-99% of the instructions. Furthermore, a fetch-from-memory bit has been proposed that is added to the instruction encoding. This approach provides a speedup in execution time by a factor of 3-8 over uncached systems without sacrificing the predictability of the program's worst-case execution time. The ability to predict the caching behavior of a large percentage of the instruction references (in a conventional cached system) or even all instruction references (using the fetch-from-memory bit) can be used to predict the execution time of large code segments on machines with instruction caches. The advantage of the bit-encoding approach is that it can be used by conventional timing tools to bound execution time while a more sophisticated and more complex timing tool is needed to achieve tight bounds for regular caches. The latter approach is discussed in the next chapter.

In summary, instruction cache behavior is sufficiently predictable to provide worst-case execution time predictions that are tight enough for scheduling analysis in a non-preemptive environment. Thus, the performance advantage of instruction caches can be exploited for critical real-time tasks by enabling either conventional or bit-encoded instruction caches.

Chapter 8

Bounding Execution Time

The use of caches poses a difficult tradeoff for architects of real-time systems. While caches provide significant performance advantages, they have also been viewed as inherently unpredictable since the behavior of a cache reference depends upon the history of previous references to the cache. The use of caches will only be suitable for real-time systems if a reasonable bound on the performance of programs using cache memory can be predicted. This chapter describes an approach for bounding the worst-case instruction cache performance of large code segments. Excerpts of this chapter can be found in [5].

8.1 Introduction

Caches present a dilemma for architects of real-time systems. The use of cache memory in the context of real-time systems introduces a potentially high level of unpredictability. An instruction's execution time can vary greatly depending on whether the instruction causes a cache hit or miss. Whether or not a particular reference is in cache depends on the program's previous dynamic behavior (i.e. the history of its previous references to the cache). As a result, it has been common practice to simply disable the cache for sections of code where predictability is required [60]. Unfortunately, even the use of other architectural features, such as a prefetch buffer, cannot approach the effectiveness of using a cache. Furthermore, as processor speeds continues to increase faster than the speed of accessing memory, the performance advantage of using cache memory becomes more significant. Thus, the performance penalty for not using cache memory in real-time applications will continue to increase.

Bounding instruction cache performance for real-time applications may be quite beneficial. The use of instruction caches has a greater impact on performance than the use of data caches. Code generated for a RISC machine typically results in four times more instruction references than data references [29]. In addition, there tends to be a greater locality for instruction references than data references, typically resulting in higher hit ratios for instruction cache performance. Also, unlike many data references, the address of each instruction remains the same during a program's execution. Thus, instruction caching behavior should be inherently more predictable than data caching behavior.

This chapter shows that, with certain restrictions, it is possible to predict much of the instruction caching behavior of a program. In contrast to previous chapters, the prediction is provided at a finer level of detail, *i.e.* for each loop nesting level. As in the last chapter, assume that a task be the portion of code executed between two scheduling points (context switches) in a system with a non-preemptive scheduling paradigm. When a task starts execution, the cache memory is assumed to be invalidated. During task execution, instructions are gradually brought into cache and often result in many hits and misses that can be predicted statically.

Figure 8.1 depicts an overview of the approach described in this chapter for predicting bounds on instruction cache performance of large code segments. Control-flow information is stored as the side effect of the compilation of a file. This control-flow information could have also been obtained by analyzing assembly or object files. The control-flow information is passed to a static cache simulator, which constructs the control-flow graph of the program

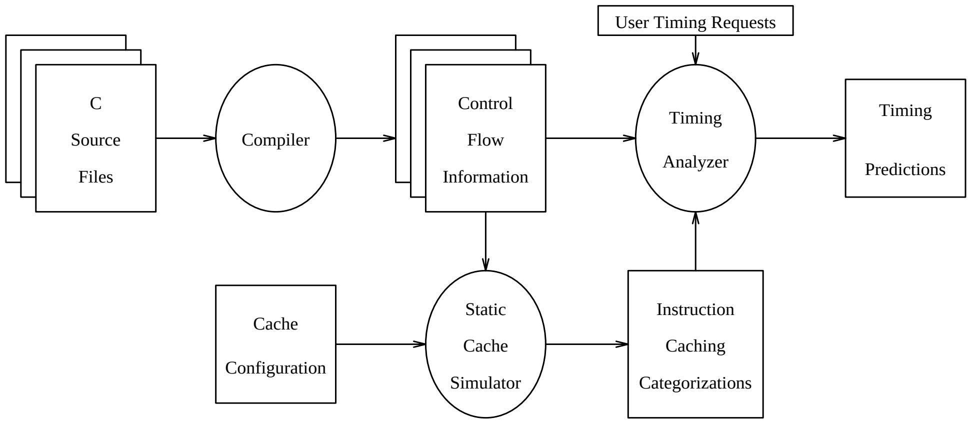


Figure 8.1: Overview of Bounding Instruction Cache Performance

that consists of the call graph and the control flow of each function based on the information provided by the compiler. The program control-flow graph is then analyzed for a given cache configuration and a categorization of each instruction's potential caching behavior is produced. Next, a timing analyzer uses the instruction caching categorizations along with the control-flow information provided by the compiler to estimate the worst-case instruction caching performance for each loop within the program. A user is then allowed to request the instruction cache performance bounds for any function or loop within the program.

This chapter is structured as follows: First, related work is reviewed. Then, the adaptation of the instruction categorization for this application is discussed. A timing analyzer, which uses the instruction categorization information, then estimates the worst-case instruction cache performance for each loop in the program. Finally, future work and conclusions are presented.

8.2 Related Work

As already mentioned in Chapter 7, several tools to predict the execution time of programs have been designed for real-time systems. The analysis has been performed at the level of source code [54], intermediate code [52], and machine code [28]. Only the last tool attempted to estimate the effect of instruction caching and was only able to analyze small code segments that contained no function calls and entirely fit into cache. Thus, this tool was able to assume that at most one miss will occur for each reference.

Niehaus outlined how the effects of caching on execution time can be estimated [53], as discussed in the last chapter. However, no general method was provided to analyze the call graph of a program and the control flow within each function.

Lin and Liou suggested that more frequently executed tasks be placed entirely in cache and other tasks be denied any cache access [44]. While this approach may have some benefit for a few tasks, the performance of the remaining tasks will be significantly decreased. Part of their rationale was that if a task could not entirely fit in cache, then the worst-case execution would be the same as an uncached system since cache hits could not be guaranteed. It will be shown later that a high percentage of instruction cache hits for such programs can be guaranteed and that the worst-case performance with an instruction cache is significantly better than a comparable system with a disabled cache.

There have been attempts to improve the performance and predictability of accessing memory for real-time systems by architectural modifications. One attempt by Kirk via cache partitioning has already been discussed in the last chapter, including some of the problems, such as lower hit ratios and increased complexity of scheduling analysis [35].

Cogswell and Segall suggest a different approach for the MACS architecture [17], which uses no cache memory. Instead, their pipelined processor performs a context swap between threads in a round-robin ordering on each instruction. No thread may have more than one instruction in the pipeline at one time. Assuming that the number of pipeline stages does not exceed the number of threads, the delay of memory fetches is overcome since the next instruction for a given thread will not be scheduled for the pipeline until the current instruction for the task has propagated through all the stages of the pipeline. By using different memory banks for different threads, memory fetches for local data can be issued without causing bus contention. Note that only the overall throughput of the entire set of threads is enhanced. The response time for an individual thread is not improved. Thus, this approach requires that a task be broken up into a number of independent threads, which implies restructuring of conventional real-time programs.

Lee *et. al.* suggested an architecture that prefetches instructions in the direction of the worst-case execution path [43]. The justification for using their approach was that "it is very difficult, if not impossible, to determine the worst-case execution path and, therefore, the worst-case execution time of a task" when instruction caching is employed. Their analysis measured a 45% improvement of the predicted worst-case time compared to no prefetching (and no instruction cache). This improvement is probably quite optimistic since bus contention was not taken into consideration (contention between the three competing memory classes for instructions access, data access, and threads). Furthermore, mispredicted branches may result in an uninterruptible block fetch along the wrong path, which often cannot be aborted. This misprediction penalty may then cause worst-case behavior along the (previously) shorter path. In addition, the ability to improve performance by prefetching a block of instructions is quite limited compared to the potential improvement when using an instruction cache. It will be shown later in the chapter that much better worst-case predictions can be made in the presence of instruction caching than with just a prefetch buffer.

8.3 Timing Analysis Tree

Timing tools generally propagate timing estimates through a timing analysis tree from node to node in a bottom-up fashion, *i.e.* the timing of leaf nodes is calculated first, followed by innermost loops in the control-flow graph, innermost functions in a call sequence, to finally the outermost function `main()`. Intermediate results are stored at each node during the timing analysis pass. This information can be used at a later stage to satisfy user requests for timings of arbitrary code ranges without recalculation of the timing behavior.

The timing analysis tree can easily be constructed from the control-flow graph of each function and the function instance graph assuming the absence of recursion.

Definition 8 (Timing Analysis Tree) *The timing analysis tree of a program is defined as follows:*

1. *The function instance $main_0$ is the root loop node with one iteration.*
2. *Any other function instance is represented as a loop node with one iteration, which is a child of the loop node at its call site.*
3. *The outermost loops of a function are represented as child nodes of the corresponding function instances.*

4. Nested loops are represented as child nodes of the next-outer loops.

A loop node λ with a distance n from the root is said to be at nesting level $nesting(\lambda) = n$. *Example:* Figure 8.3 (discussed in detail later) shows a simple C program in (a) and the corresponding timing analysis tree in (b). The function `main` at level 0 includes `loop1` at nesting level 1. The loop calls `value (a)` and `value (b)`, both at level 2. ■

8.4 Adaptation of Static Cache Simulation

This application requires a number of adaptations of the instruction categorization. First, the static cache simulation is performed on the level of basic blocks rather than unique paths to avoid different categorizations of the same instruction when paths overlap. Furthermore, Definition 7 for instruction categorization is adapted to provide sufficiently fine-grained information for the timing tool. The original definition was based on the notion of abstract cache states and reaching states for the entire program. This global notion of caching behavior is too coarse to provide tight timing estimates at the level of functions, loops, or even basic blocks. Consider the conflict category. For worst-case timings, a conflict would be counted as a miss since this reflects the worst-case memory access time. Conversely, a conflict would be approximated as a hit for the best case. This would result in some predictability of instruction caches but the timing estimates may not be very tight. A tighter timing prediction can be provided if conflict instructions can be categorized separately at each loop level as one of the categories besides conflicts.
Example: In Figure 8.2, instruction `a` is the first instruction that can be executed within the

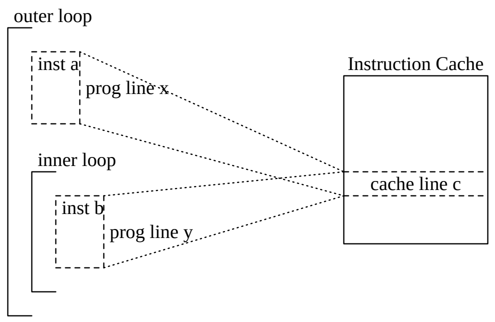


Figure 8.2: Example of Conflicting Lines between Nested Loops

program line `x` in the outer loop. Instruction `b` is the first instruction that can be executed within the program line `y` in the inner loop. Assume program lines `x` and `y` are the only two lines that map to cache line `c` and there are no conditional transfers of control within the two loops. In other words, instructions `a` and `b` will always be executed on each iteration of the outer and inner loops, respectively. How should instruction `b` be classified? With respect to the inner loop, instruction `b` will not be in cache when referenced on the first iteration, but will be in cache when referenced on the remaining iterations. This situation can be ascertained by the static cache simulator since it can determine that there are no other program lines within the inner loop that conflict with program line `y`. In addition, the abstract cache state at the exit point of the basic block preceding the inner loop does not contain program line `y`. With respect to the outer loop, instruction `b` will always cause a miss on each iteration since it will not be in cache as the outer loop initially enters the inner loop.[1] ■

[1]Note that instruction `a` would be classified as an always miss.

Definition 7 needs to be revisited and adapted to represent the caching behavior relative to each loop nesting level. The same terminology as for Definition 7 is assumed for Definition 9 below with the following changes.

- Let i_k be an instruction within a UP, a loop λ, and a function instance.
- Let u be the set of program lines in loop λ.
- Let $child(\lambda)$ be the child loop of λ for this UP and function instance, if such a child loop exists.
- Let $header(\lambda)$ be the set of header paths, $preheader(\lambda)$ be the set of preheader paths and $backedges(\lambda)$ be the set of backedges of loop λ, respectively (see Figure 3.2(a)).[2]
- Let $s(p)$ be the abstract output cache state of path p.[3]
- Let $linear(p)$ be the linear cache state of path p.
- Let $postdom(p)$ be the set of self-reflexive post-dominating programming lines of path p.

The linear cache state of a path represents the hypothetical cache state in the absence of loops. It is calculated by algorithm 2 where backedges in the control flow and recursive edges in the function-instance graph are disregarded.[4]

The post dominator set of a path includes the program lines that are certain to be reached from the path, regardless of the taken paths in the control flow. This information for basic blocks is commonly used in optimizing compilers. A more detailed discussion of post dominators can be found elsewhere [3].

In addition, a new category of *first-hits* is introduced. In analogy to a first miss, a first hit occurs when the first reference to an instruction in a loop results in a cache hit but all subsequent references in the loop result in misses.

Definition 9 (Instruction Categorization for a Loop) *The instruction categorization is defined separately for the worst-case execution timing and the best-case execution timing as*

$$\text{worst }(i_k,\lambda)=\begin{cases}\text{always-hit} & \text{if } k \neq first \vee (l \in s \wedge \underset{m \to c, m \neq l}{\forall} m \notin s)\\ \text{first-hit} & \text{if worst}(i_k, child(\lambda)) = \text{first-hit} \vee \\ & \quad k = first \wedge l \in s \wedge \underset{m \to c, m \neq l}{\exists} m \in (s \cap u) \wedge \\ & \quad [\underset{p \in preheaders(\lambda)}{\forall} l \in s(p) \wedge \underset{m \to c, m \neq l}{\forall} m \notin (s(p) \cap u)] \wedge \\ & \quad \underset{p \in headers(\lambda)}{\forall} l \in postdom(p) \wedge \underset{m \to c, m \neq l}{\forall} m \notin (linear \cap u)\\ \text{first-miss} & \text{if worst}(i_k, child(\lambda)) = \text{first-miss} \wedge k = first \wedge l \in s \wedge \\ & \quad \underset{m \to c, m \neq l}{\exists} m \in s \wedge \underset{m \to c, m \neq l}{\forall} m \notin (s \cap u)\\ \text{always-miss} & \text{otherwise}\end{cases}$$

[2]The common notion of "natural loops" defines a single loop header preceded by a single preheader outside the loop [3]. This work extends this notion to handle more general control flow with unstructured loops. Multiple loop headers occur only for unstructured loops, which are handled by the simulator. Multiple loop preheaders occur when the loop can be entered from more than one path outside the loop, which can occur even for natural loops.

[3]This notation is also used for edges $e = p \to q$ where $s(e) = s(p)$. Thus, abstract cache state $s(e)$ of an edge is the the abstract output cache state of the source path p of the edge.

[4]Linear states are even calculated correctly, yet conservatively, for unstructured loops by disregarding all backedges of an unstructured loop.

$$\text{best }(i_k,\lambda) = \begin{cases} \text{always-miss} & \text{if } k = first \wedge l \notin s \\ \text{first-hit} & \text{if best}(i_k, child(\lambda)) = \text{first-hit} \vee \\ & \quad k = first \wedge l \in s \wedge \underset{m \rightarrow c, m \neq l}{\exists} m \in (s \cap u) \wedge \\ & \quad \underset{p \in preheaders(\lambda)}{\forall} l \in s(p) \wedge \\ & \quad \underset{p \in headers(\lambda)}{\forall} l \in postdom(p) \wedge \underset{b \in backedges(\lambda)}{\forall} l \notin s(b) \\ \text{first-miss} & \text{if best}(i_k, child(\lambda)) \in \{\text{first-miss,always-hit}\} \wedge \\ & \quad k = first \wedge l \in s \wedge l \notin linear \\ \text{always-hit} & \text{otherwise} \end{cases}$$

Unlike the original categories introduced in Definition 7, recognizing first hits requires more data flow information. Nonetheless, it was decided to recognize first hits during the static cache simulation to provide the timing tool with this new category and facilitate its job of timing prediction. The information for recognizing first hits can be obtained by using available information during static cache simulation and by calculating additional information using existing algorithms. First hits can then be used by the timing tool to achieve slightly tighter timing estimates for the worst case and much tighter timing estimates for the best case. Informally, a first hit occurs under the following conditions.

1. The instruction was a first hit for the previous (deeper) loop nesting level or all of the following conditions hold.
2. The instruction is the first reference to the program line in the path.
3. The current line is in the abstract cache state.
4. A conflicting line is in the abstract cache state for this loop.
5. The current line is in the abstract output cache state of all preheaders of this loop.
6. None of the conflicting lines is in the abstract output cache state of the preheaders of this loop (only for worst case).
7. The current line is in the post dominator of the loop's headers, *i.e.* the current line will be referenced during each loop iteration.[5]
8. None of the conflicting lines are in the linear cache state of the current path, *i.e.* for each loop iteration, the current line will be referenced before any conflicting line (only for worst case).
9. The current line is not in the abstract cache state preceding any of the backedges, *i.e.* the current line is replaced by a conflicting line during each loop iteration (only for best case).

Several categorizations depend on the categorization of the previous (deeper) loop nesting level. This ensures the following invariants that are essential for the consistency of timing predictions.

- Once an instruction becomes a first hit, it will remain a first hit for all higher level nestings.

[5]This property is not essential for the categorization but simplifies the implementation of the timing tool since it can be assumed that a first hit is always referenced during a loop iteration, independent of any conditional execution.

- For the worst-case categorization, once an instruction ceases to be a first miss, it will never again be a first miss for any higher nesting level.

- For the best-case categorization, once an instruction ceases to be an always hit, it will never again be an always hit for any higher nesting level.

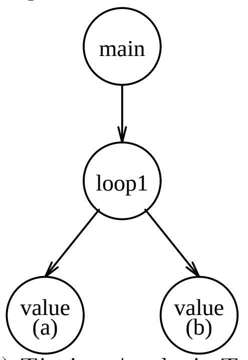
Notice also that the definition of first misses for the best case checks if the current line is not in the linear cache state, thereby allowing that the line be brought into cache during the first iteration.

The static cache simulator categorizes the instructions for each function instance, each loop level, and both the best-case and worst-base categorization.

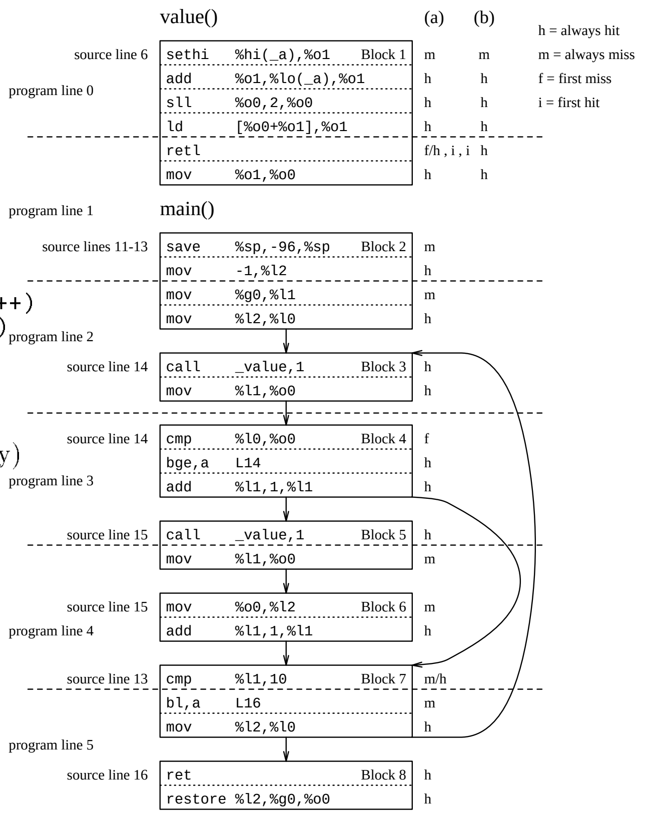
Example: The approach for bounding instruction cache performance is illustrated in Figure 8.3. Part (a) contains the C code for a simple toy program that finds the largest value in

```
 1  extern int a[10];
 2
 3  int value(index)
 4  int index;
 5  {
 6     return a[index];
 7  }
 8
 9  main()
10  {
11     int i, max = -1;
12
13     for (i = 0; i < 10; i++)
14        if (max < value(i))
15           max = value(i);
16     return max;
17  }
```

(a) C Program to find MAX(Array)



(b) Timing Analysis Tree



```
value()                                                  (a)       (b)
source line 6       sethi    %hi(_a),%o1      Block 1    m         m
                    add      %o1,%lo(_a),%o1             h         h
                    sll      %o0,2,%o0                   h         h
                    ld       [%o0+%o1],%o1               h         h
                    retl                                 f/h,i,i   h
                    mov      %o1,%o0                     h         h
main()
source lines 11-13  save     %sp,-96,%sp      Block 2    m
                    mov      -1,%l2                      h
                    mov      %g0,%l1                     m
                    mov      %l2,%l0                     h
source line 14      call     _value,1         Block 3    h
                    mov      %l1,%o0                     h
source line 14      cmp      %l0,%o0          Block 4    f
                    bge,a    L14                         h
                    add      %l1,1,%l1                   h
source line 15      call     _value,1         Block 5    h
                    mov      %l1,%o0                     m
source line 15      mov      %o0,%l2          Block 6    m
                    add      %l1,1,%l1                   h
source line 13      cmp      %l1,10           Block 7    m/h
                    bl,a     L16                         m
                    mov      %l2,%l0                     h
source line 16      ret                       Block 8    h
                    restore  %l2,%g0,%o0                 h
```

(c) Instruction Categorization

Figure 8.3: Sample Analysis by the Static Cache Simulator

an array, part (b) illustrates the corresponding timing analysis tree, and part (c) shows the actual SPARC assembly instructions generated for this program within a control-flow graph of basic blocks. Assume there are 4 cache lines and the line size is 16 bytes (4 SPARC instructions). Note the immediate successor of a block with a call is the first block in that instance of the called function. Block 1a corresponds to the first instance of `value()` called from block 3 and block 1b corresponds to the second instance of `value()` called from block 5. The instruction categorizations are given to the right of each instruction.

Instructions categorizations are separated by a slash if the the worst case differs from the best case (*e.g.* block 7, instruction 1). Categorizations that differ for each loop level are separated by commas from the innermost loop level at the far left to the outermost loop level at the far right (*e.g.* block 1, instruction 5). This example also shows a different categorization for the best case and the worst case, separated by a slash. Recall that a function is considered a loop with a single iteration.

Each instruction is categorized according to the criteria specified in Definition 9 providing information that may not be detected by a naive inspection of only physically contiguous sequences of references. For instance, the static cache simulator determined that the fifth instruction in block 1b will always be in cache (an always hit) due to temporal locality. It detected that the first instruction of block 1, the last instruction of block 5, and the first instruction of block 6 will never be in cache (always misses) since the program lines associated with these instructions map to the same cache line and the execution of block 1 occurs exactly between block 5 and 6. The static cache simulator was also able to predict the caching behavior of instructions that could not be classified as always being a hit or always a miss. It determined that the first instruction in block 4 will miss on its first reference and all subsequent references will be hits. The fifth instruction of block 1a is a first miss for function instance (a) of `value` and the worst-case prediction, *i.e.* if the function value was timed, the instruction would be timed as a miss since there is only one iteration of a function. For the best case and the timing of value (a), the prediction indicates an always hit. This is the optimistic (best case) interpretation of the fact that program line 1 *may* be in cache when value (a) is called. The same instruction is then classified as a first hit for the loop consisting of blocks 3 to 7 and for `main`. This is due to spatial locality, caused by bringing program line 1 into cache when block 2 executes. During the first call to `value (a)`, program line 1 will be in cache but is then replaced by line 5, which is accessed by block 7 and causes subsequent misses for the fifth instruction of block 1a. Thus, the first instruction of block 7 is categorized as a miss for the worst-case prediction since the line is not in cache if the branch in block 4 is taken. Conversely, the instruction is categorized as a hit for the best-case prediction since the line was brought into cache by block 6 if the branch in block 4 was not taken. ■

The instruction categorization is summarized in an interface file as specified in Appendix E. This file is produced by the static cache simulator instead of the code instrumentation discussed in the previous chapters. The interface file is used by the timing tool to identify the caching behavior of instructions in the program, which will be discussed in the next section.

8.5 Timing Analysis

The design and implementation of the timing analysis tool is beyond the scope of this work. A detailed description can be found elsewhere [5, 4]. But a short outline and some preliminary results shall be presented to illustrate the benefit of static cache simulation for timing analysis.

The timing tool constructs a timing analysis tree as discussed earlier. The instruction categorizations are then used to calculate the timings for each node. The timings are expressed in number of processor cycles, which can be easily transformed into seconds for a known processor cycle frequency.

The timings for each node are calculated in a bottom-up fashion in the timing analysis tree. First, both worst-case and best-case timings are calculated for leaf nodes by traversals through the longest and shortest paths. The timing of non-leaf nodes is determined again by path traversals where the timing of child nodes has already been calculated and can simply be added for the corresponding paths. The timing of child nodes is also adjusted at a higher nesting level if the child nodes contained first hits or first misses.[6]

Some preliminary timing predictions are shown in Table 8.1. Currently, only the worst-

Table 8.1: Worst-Case Time Estimation for Four Test Programs

	Static Measurements					Dynamic Worst-Case Measurements			
Name	Size [Bytes]	Num Funcs	Always Hit	Always Miss	First Miss	Hit Ratio	Observed Cycles	Estimated Ratio	Naive Ratio
Matmult	788	7	71.15%	25.28%	3.57%	99.04%	2,917,887	1.00	9.21
Matsum	632	7	69.89%	26.24%	3.87%	87.08%	677,204	1.00	4.63
Matsumcnt	800	8	70.64%	26.70%	2.65%	85.32%	959,064	1.09	4.31
Bubblesort	520	5	68.18%	27.60%	4.22%	84.05%	7,620,684	1.99	8.18

case calculation algorithm has been implemented. The first program, *Matmult*, multiples two 50x50 matrices. The second program, *Matsum*, determines the sum of the non-negative values in a 100x100 matrix. The third program, *Matsumcnt*, is a variation of the second program, *Matsum*, since it also counts the number of elements that were summed. The final program, *Bubblesort*, uses the bubblesort algorithm to sort an array of 500 numbers into ascending order.

For each program, a direct-mapped cache configuration containing 8 lines of 16 bytes was used. Thus, the cache contains 128 bytes. The programs were 4 to 6 times larger than the cache as shown in column 2 of Table 3. Column 3 shows that each program was highly modularized to illustrate the handling of timing predictions across functions. Columns 4-6 show the static percentage of each type of instruction categorization in the function-instance graph. Column 7 indicates the hit ratio for each program. Only *Matmult* had a very high hit ratio. This was due to the program spending most of its cycles in 3 tightly nested loops containing no calls to perform the actual multiplication. Column 8 shows the time in cycles for an execution with worst-case input data. The number of cycles was measured using a traditional cache simulator [20], where a hit required one cycle and a miss required ten cycles (a miss penalty of nine cycles). These assumptions were described as realistic by other researchers [31, 29]. Column 9 shows the ratio of the predicted worst-case instruction cache performance using the timing analyzer to the observed worst-case performance in column 8. Column 10 shows a similar ratio assuming a disabled cache. This naive prediction simply determines the maximum number of instructions that could be executed and assumes that each instruction reference requires a memory fetch of ten cycles (miss time).

For programs without conditional control flow except for looping constructs (*e.g.* Matmult), the timing estimation is exact. Even for simple conditional statements (*e.g.* Matsum), the prediction for worst-case performance estimates is generally very tight. In case of Matsum, it is even exact.

As the conditional control flow becomes more complex (*e.g.* Matsumcnt), the estimates are no longer accurate but remain relatively tight. The analysis of the last program, Bubblesort, depicts a problem faced by any conventional timing analyzer. The *Bubblesort* program contains an inner loop whose number of iterations depends on the counter of an outer loop. Without additional information from the compiler or from the user, the limits of any static analysis method are reached. The loop count will be overestimated for the worst-case timing prediction. This explains why the estimated worst-case time is twice as high as the observed time and indicates the limits of strictly analytical timing prediction in general.

[6]A detailed description of these timing prediction algorithms is beyond the scope of this work.

The user of the timing tool can query the estimated execution time of a range of source lines. This range is approximated as closely as possible by a range of basic blocks. The timing can then be calculated based on the timing analysis tree as explained earlier.

In summary, timing analysis based on static cache simulation for instruction caches can result in tight timing predictions with a small error at the order of traditional timing predictions for uncached systems, contrary to the belief that instruction caches are unpredictable.

8.6 Future Work

Future extensions concern mostly the timing tool and are outside the scope of this dissertation but shall be mentioned briefly. An algorithm has been designed and partially implemented that estimates the best-case instruction cache for each loop within a program. The facility to query timing predictions is currently being extended to provide a user-friendly interface under a window environment. Current work also includes an attempt to predict the execution time of code segments on a MicroSPARC I processor. In order to provide realistic timing predictions, the effect of other architectural features besides instruction caching (*e.g.* pipelining) must be analyzed. A technique called micro-analysis [28] was developed to detect the potential overlap between operations on various CISC processors. This technique is being extended to model the MicroSPARC I processor.

8.7 Conclusion

Predicting the worst-case execution time of a program on a processor that uses cache memory has long been considered an intractable problem [60, 44, 43]. However, this work shows that tight estimations in the presence of instruction caches are feasible, using the fact that the addresses of the instructions within a program and the possible control-flow paths between these instructions are known statically.

This chapter presents a technique for predicting worst-case instruction cache performance in two steps. First, a static cache simulator analyzes the control flow of a program to statically categorize the caching behavior of each instruction within the program. Second, a timing analyzer uses this instruction categorization information to estimate the worst-case instruction cache performance for each loop in the program. The user is allowed to query the timing analyzer for the estimated worst-case performance of any function or loop within the program.

It has been demonstrated that instruction cache behavior is sufficiently predictable for real-time applications. Thus, instruction caches should be enabled, yielding a speedup of four to nine for the predicted worst case compared to disabled caches (depending on the hit ratio and miss penalty). This speedup is a considerable improvement over prior work, such as requiring special architectural modifications for prefetching, which only results in a speedup factor of 2 [43]. As processor speeds continue to increase faster than the speed of accessing memory, the performance benefit of using cache memory in real-time systems will only increase.

Chapter 9
A Real-Time Debugging Tool

Debugging is an integral part of the software development cycle that can account for up to 50% of the development time of an application. This chapter discusses some of the challenges specific to real-time debugging. It explains how developing real-time applications can be supported by an environment that addresses the issues of time deadline monitoring and distortion due to the interference of debugging. The current implementation of this environment provides the elapsed time during debugging on request at breakpoints. This time information corresponds to the elapsed execution time since program initiation. Delays due to the interference of the debugger, for example input delays at breakpoints, are excluded from the time estimates. The environment includes a modified compiler and a static cache simulator that together produce instrumented programs for the purpose of debugging. The instrumented program supports source-level debugging of optimized code and efficient cache simulation to provide timing information at execution time. The overhead in execution time of an instrumented optimized program is only approximately 1 to 4 times slower than the corresponding unoptimized program. Conventional hardware simulators could alternatively be used to obtain the same information but would run much slower. The environment facilitates the debugging of real-time applications. It allows the monitoring of deadlines, helps to locate the first task that misses a deadline, and supports the search for code portions that account for most of the execution time. This facilitates hand-tuning of selected tasks to make a schedule feasible. Excerpts of this chapter can be found in [48].

9.1 Introduction

The issue of debugging real-time applications has received little attention in the past. Yet, in the process of building real-time applications, debugging is commonly performed just as often as in the development of non-real-time software and may account for up to 50% of the development time [66]. The debugging tools used for real-time applications are often ordinary debuggers that do not cater to specific needs of real-time systems listed below.

Time distortion: The notion of real time is central to real-time applications. Hardware timers are commonly used to inquire timing information during program execution to synchronize the application with periodic events. Yet, during debugging the notion of real time should be replaced by the notion of virtual time to compensate for time distortion due to the interference of debugging. External events have to be simulated based on the elapsed (virtual) time of tasks. Thus, values of variables used by the application can be related to the elapsed time, which is essential for debugging real-time applications.

Deadline monitoring: During the implementation phase, deadlines may not always be met. A real-time debugger should display the elapsed time for a task on request. This would facilitate finding the first task that fails to meet a deadline. It could also be used to inquire at which point during the execution a deadline was missed. Furthermore, the elapsed time may help in tuning tasks by locating where most of the execution time is spent.

Uniprocessor vs. multiprocessor: Multiprocessor applications are sometimes simulated on uniprocessors during debugging. In this case, a virtual clock has to be kept for each processor that is shared by a set of tasks running on this processor.

This work concentrates on time distortion and deadline monitoring.

A debugging environment has been developed that permits the user to query the elapsed time. This time corresponds to the virtual time from program initiation to the current breakpoint excluding debugging overhead and is calculated on demand. In contrast, time queries in current debuggers correspond to the wall-clock time and include the delay of user input at breakpoints as well as the debugger trap overhead.

The environment can be used to debug a real-time application whose tasks do not meet their deadline. It facilitates the analyses of the tasks and helps to find out where a task spends most of its execution time or which portion of a task completed execution before missing the deadline. This knowledge can then be utilized to fine-tune the task that is missing its deadlines or any of the previous tasks in the schedule. Thus, this debugging environment assists the process of designing a feasible schedule in a step-by-step fashion.

The elapsed time of a task is estimated based on the caching behavior of the task. The caching information is updated during execution and provides an estimate of the number of elapsed processor cycles.

The dynamic simulation of cache performance necessitates the tracking of events and their ordering to determine a cache miss *vs.* a cache hit. It can be quite a challenge to perform order-dependent events efficiently. This chapter describes the design and implementation of such an environment within the framework of a compiler, a static cache simulator [50], and an arbitrary source-level debugger. The compiler translates a program into assembly code and provides control-flow information to the static cache simulator. The static cache simulator analyzes the caching behavior of the program and produces instrumentation code that is merged into the assembly code. The source program corresponding to the resulting assembly code can then be debugged and the elapsed time can be requested at breakpoints.

The elapsed time is calculated based on the cache simulation up to the current breakpoint, i.e. the number of cache hits and misses are multiplied by the access time for hits and misses, respectively. This provides an estimate of the executed numbers of processor cycles corresponding to the elapsed (virtual) time since program initiation.

It may be argued that the virtual execution time can be provided by the operating system. Notice though that the debugging process affects the execution of the real-time task, *e.g.* the caching behavior. The cache simulation discussed here estimates the timing of the task in an actual real-time environment disregarding the interference of debugging.

Another problem is posed by the debugging of optimized code. Conventional compilers only support source-level debugging of unoptimized code. Clearly, unoptimized code causes further time distortion, which cannot be accepted for real-time systems. Thus, a compiler has been modified to support source-level debugging of optimized code with certain restrictions, which are discussed later in the chapter.

This chapter is structured as follows: In the next section, related work is discussed. Then, a new real-time debugging environment is introduced. In the following, the application of the environment is illustrated. In addition, the feasibility of the environment is demonstrated by presenting performance figures. Finally, future work and conclusions are presented.

9.2 Related Work

Conventional debugging tools, whether at the assembly-level or at the source-level, do not address the specific demands of real-time debugging. The amount of work in the area of real-time debugging has been limited with a few exceptions.

The Remedy debugging tool [57] addresses the customization of the debugging interface for real-time purposes and synchronizes on breakpoints by suspending the execution on all processors. DCT [9] is a tool that allows practically non-intrusive monitoring but requires special hardware for bus access and does not extend to non-intrusive debugging. Both RED [30] and ART [67] provide monitoring and debugging facilities at the price of software instrumentation. RED dedicates a co-processor to collect trace data and send it to the host system. The instrumentation is removed for production code. In ART, a special reporting task sends trace data to a host system for further processing. The instrumentation code is a permanent part of the application. It will never be removed to prevent alteration of the timing. Debugging is limited to forced suspension and resumption of entities, viewing and alteration of variables, and monitoring of communication messages.

The DARTS system [66] approaches the debugging problem in two stages. It first generates a program trace and then allows for debugging based on the trace data that is time-stamped to address the time distortion problem. The debugging is limited to a restricted set of events that is extracted from the control flow. This tool only supports a subset of the functionality of common debuggers, *e.g.* excluding data queries. The high volume of trace information and the associated overhead of trace generation may also limit its application to programs with short execution times. None of the systems make use of the compiler to enhance the debugging process.

In the absence of real-time debuggers, hardware simulators are often used that run considerably slower than the actual application and, consequently, allow only selective and not very extensive testing. In addition, changing the simulated architecture of hardware simulators is typically complicated.

9.3 A Real-Time Debugging Environment

The current work concentrates on monitoring deadlines based on the cache analysis of a task and the corresponding estimate of the elapsed (virtual) execution time. This facility can be used in conjunction with a conventional debugger. The debugger does not need to be modified.

The cache simulation overhead at run time is reduced by analyzing the cache behavior statically. A large number of cache hits and misses can be determined prior to execution time by considering the control flow of each function and the call graph of the program. The remaining references are simulated at execution time.

Figure 9.1 depicts an overview of the environment. A set of source files of a program is translated by a compiler. The compiler generates object code with symbol table entries and passes information about the control flow of each source file to the static cache simulator. The static cache simulator performs the task of determining which instruction references can be predicted prior to execution time. It constructs the call graph of the program and the control-flow graph of each function based on the information provided by the compiler. The cache behavior is then simulated for a given cache configuration. Furthermore, the static simulator produces instruction annotations and passes them to the linker, which modifies the object code according to the annotations and creates an executable program including library routines for the time estimation. The executable may then be run within a source-level debugger. The elapsed time can be inquired at any breakpoint by calling the library routine that estimates the number of processor cycles executed based on the number of cache hits and misses up to that point.

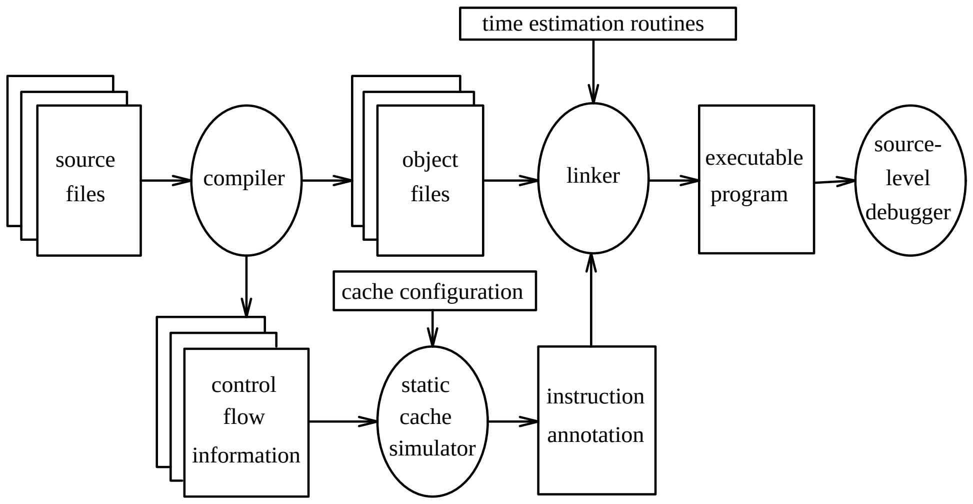


Figure 9.1: Overview of the Debugging Environment

9.3.1 Adaptation of Static Cache Simulation

For real-time debugging support, the static cache simulator is adapted in the same way as discussed in Chapter 6 with one exception: The static cache simulation is performed on the level of basic blocks rather than unique paths to represent the control flow of a function, which provides a finer level of granularity for intermediate timings. The code instrumentation provides the means to calculate the number of cache hits and misses at any given point during program execution. The calculation is based on simple frequency counters and the adjustment of first misses. Thus, the calculation can be repeated and the program can be interrupted at breakpoints without influencing the outcome of this calculation.

9.3.2 Querying the Elapsed Time

The elapsed execution time can be queried at any breakpoint while debugging a program without modifying the debugger. The time is calculated based on the cache analysis. The number of cache hits and misses can be calculated on the fly from the frequency counters. The elapsed time is then calculated as

$$t_{elapsed} = hits * hit\_penalty + misses * miss\_penalty \text{ [cycles]}$$

where the hit penalty is typically one cycle while the miss penalty is ten cycles [31] or even more, depending on the clock rate and the access time of main memory. This time estimate can be converted into seconds by multiplying it by the cycle time. The calculation of hits and misses takes only a short time and can therefore be repeated whenever the program stops at a breakpoint without much overhead. The code performing the calculation is hidden in linked-in library code.

The program being debugged has been compiled with full optimizations to avoid time distortion. The compiler was modified to emit debugging information for unoptimized code *as well as* optimized code. Emitting accurate debugging information for optimized code is a non-trivial task and subject to ongoing research [1, 11, 32]. Contrary to debugging unoptimized code, debugging optimized code typically restricts the scope of breakpoints and the displaying of data structures. In the debugging environment described in this chapter, a breakpoint set on a source line is approximated as a breakpoint at the beginning of the corresponding basic block when code is optimized. In addition, the value of variables assigned to a register will only be displayed if all live ranges of the variable are assigned to the same register [3]. Register-mapped values may still be inconsistent at times due to global optimizations, such as common subexpression elimination, which is a common problem when debugging optimized code.

The fact that optimized code is executed during debugging speeds up the execution over conventional debugging of unoptimized code. The cache simulation, on the other hand, adds to the execution time. A quantitative analysis of the effect of these issues will be given in the measurement section.

9.4 Application of the Debugging Tool

The output shown in Figure 9.2 illustrates a short debugging session of a program performing fast Fourier transformations within the environment using the unmodified source-level debugger *dbx* [64].

```
> dbx fft
Reading symbolic information...
Read 396 symbols
(dbx) stop at 43              /* set breakpoint on line 43                 */
(2) stop at 43
(dbx) stop at 114             /* set breakpoint on line 114                */
(3) stop at 114
(dbx) stop at 123 if elapsed_cycles() > 4000000 /* set cond. breakpoint      */
(4) stop  at 123 if elapsed_cycles() > 4000000
(dbx) display elapsed_cycles() /* display function return value on breakpoint */
elapsed_cycles() = 0          /* 0 cycles since program has not started    */
(dbx) run                     /* start program execution                   */
Running: fft
stopped in main at line 114   /* execution stopped on first breakpoint     */
  114    printf("Objective: measure exec. time of 128 FFT.\n");
elapsed_cycles() = 22         /* 22 cycles executed before first breakpoint */
(dbx) cont                    /* resume execution until next breakpoint    */
Objective: measure exec. time of 128 FFT.      /* program output           */
stopped in four at line 43
   43    mmax=2;
elapsed_cycles() = 29413
(dbx) next                    /* single step to next source line statement  */
stopped in four at line 44
   44    while(n>mmax) {
elapsed_cycles() = 29428
(dbx) print mmax              /* print out value of variable               */
mmax = 2
(dbx) cont
stopped in four at line 43
   43    mmax=2;
elapsed_cycles() = 70547
(dbx) clear                   /* clear current breakpoint (line 43)        */
(dbx) next
stopped in four at line 44
   44    while(n>mmax) {
elapsed_cycles() = 70553
(dbx) cont
stopped in main at line 123   /* execution stopped on conditional breakpoint */
  123      four(tdata,nn,isign);
elapsed_cycles() = 4015629
(dbx) clear
(dbx) cont
K = 100   Time = 0.290000 Seconds              /* program output           */
elapsed cycles() = 4095351    /* total number of executed cycles           */
execution completed, exit code is 1
program exited with 1
(dbx) quit
```

Figure 9.2: Annotated Sample Debugging Session

First, a few breakpoints are set including a conditional breakpoint on a function call that checks on a deadline miss after 4 million cycles. The display command ensures that the elapsed time estimated in cycles is displayed at each breakpoint as seen later during execution. The value of the variable `mmax` can be printed although it has been assigned to a register due to code optimization. Notice that the breakpoint on line 43 is reached twice. The difference in the number of cycles between line 43 and line 44 is 15 cycles during the first iteration but only 6 cycles during the second iteration. A closer investigation reveals that during the first iteration, one of the six instructions in the basic block references a program line that results in a compulsory miss estimated as 10 cycles. On the second iteration, the same reference results in a hit due to temporal locality estimated as 1 cycle. The execution is stopped on line 123 after over 4 million cycles, which indicates that the task could not finish within the given deadline. This conditional breakpoint was placed on a repeatedly executed function call to periodically check this condition. The deadline miss can be narrowed down to an even smaller code portion by setting further conditional breakpoints. At program termination, the final number of processor cycles is displayed.

The timing information can be used during debugging to locate portions of code that consume most of the execution time. This knowledge can be used to hand-tune programs or redesign algorithms.

When a set of real-time tasks is debugged, one can identify the task that is missing a deadline either by checking the elapsed time or by setting a conditional breakpoint dependent on the elapsed time. The schedule can then be fixed in various ways. One can tune the task that missed the deadline. Alternatively, one can tune any of the preceding tasks if this results in a feasible schedule. The latter may be a useful approach when a task overruns its estimated execution time without violating a deadline, thereby causing subsequent tasks to miss their deadlines. The debugger will help to find the culprit in such situations. Another option would be to redesign the task set and the schedule, for example by further partitioning of the tasks [24].

9.5 Measurements

The environment discussed above was implemented for the SPARC architecture. It includes a modified compiler front-end of VPCC (very portable C compiler) [18] and a modified back-end of VPO (very portable optimizer) [8], the static simulator for direct-mapped caches [50], and the regular system linker and source level debugger `dbx` under SunOS 4.1.3. Calling a library routine to query the elapsed time takes a negligible amount of time in the order of one millisecond. Thus, this section focuses on measuring the overhead of cache simulation during program execution. The correctness of the instruction cache simulation was verified by comparison with a traditional trace-driven cache simulator. The execution time was measured for a number of user programs, benchmarks, and UNIX utilities using the built-in timer of the operating system to determine the overhead of cache simulation at run time. Table 9.1 shows programs of varying program size (column 3), the overhead of unoptimized code (column 4), and the support of virtual timing information through dynamic cache simulation as a factor of the execution time of optimized code for cache sizes of 1kB, 2KB, 4kB, and 8kB (column 5-8).

On the average, unoptimized programs ran 1.8 times slower than their optimized version. Running the optimized program and performing cache simulation to provide virtual timing information was on average 2.1 to 7.8 times slower than executing optimized code.[1] In other

[1]The overhead reported here differs from Table 6.2 in Chapter 6 since the former used instrumentation for basic blocks and the latter used instrumentation for UPs. Notice also that the static cache simulation

Table 9.1: Performance Overhead for Debugging

	Size	unopt.	opt. code with time estimates			
Name	[bytes]	code	1kB	2kB	4kB	8kB
cachesim	8,452	1.1	2.0	1.4	1.3	1.2
cb	4,968	1.4	6.8	5.8	3.4	2.6
compact	5,912	2.1	10.3	7.8	6.0	2.7
copt	4,144	1.4	2.5	1.7	1.4	1.4
dhrystone	1,912	1.6	2.7	1.6	1.6	1.6
fft	1,968	1.3	1.4	1.2	1.2	1.2
genreport	17,716	1.4	3.6	2.5	2.4	2.3
mincost	4,492	1.6	5.0	3.2	2.2	1.8
sched	8,352	2.1	22.9	14.6	8.3	4.1
sdiff	7,288	4.1	27.1	8.1	4.0	3.0
whetstone	4,812	1.2	2.0	2.0	1.5	1.2
average	6,365	1.8	7.8	4.5	3.0	2.1

words, the optimized code with cache simulation was roughly 1 to 4 times slower than the unoptimized code typically used for program debugging.

The cache size influences the overhead factor considerably, which can be explained as follows: For small cache sizes, programs do not fit into cache and capacity misses occur frequently, which requires the dynamic overhead of simulating program lines classified as *conflicts*. For larger cache sizes, a larger portion of the program fits into cache reducing capacity misses and thereby reducing the number of conflicts. Once the entire program fits into cache, no conflicts need to be simulated. Rather, frequency counters are sufficient to simulate the cache behavior. This reduces the overhead considerably.

9.6 Future Work

The work is currently being extended to take the effect of pipeline stalls and other machine-specific characteristics into account. The goal is to provide a debugging framework via minimal hardware simulation for the MicroSPARC I processor [49]. The work could be extended to take external events into account. The user will be required to specify the occurrence of events in a time table. The events are then simulated by the debugging environment based on the elapsed time. At program termination, the monitored activities (*e.g.* completion time, deadline) could be summarized in a table.

The interaction of the debugging environment with a compiler provides the means to introduce a compilation `pragma zero_time` that excludes a code portion from virtual time accounting. This can be used to insert conditionally compiled debugging code that does not affect the overall timing.

Furthermore, this environment could also be used for multi-threaded applications where a thread corresponds to a task. The application could be designed for a non-preemptive embedded system[2] but may be debugged on a regular workstation using this environment to simulate the embedded system efficiently.

makes the debugging approach feasible. Traditional trace-driven cache simulation (for each basic block) is reported to slow down the execution time by over one to three orders of a magnitude [68].

[2]A multi-threaded real-time kernel has been designed for such an embedded system based on a SPARC VME bus board [6, 51].

9.7 Conclusion

This work discusses some challenges of real-time debugging that have not yet been addressed adequately. A debugging environment is proposed that addresses the problem of time distortion during debugging. In this environment, the notion of real time is replaced by virtual time based on the estimated number of elapsed processor cycles. The first implementation step has been completed and provides the elapsed time based on instruction cache simulation at any breakpoint during debugging. This time information can be used for deadline monitoring, identifying the task that first misses a deadline, or locating time-consuming code portions to support hand-tuning of tasks until a schedule becomes feasible. To provide this timing information, the execution speed of the application during debugging is 1-4 times slower in average than the speed of the corresponding unoptimized application. In contrast, conventional hardware simulators may provide the same information but are less portable and much slower. The environment facilitates the debugging of real-time programs when timing-related problems occur that have to be reproduced during debugging.

Chapter 10
Future Work

The future work sections of each chapter discuss various extensions of efficient on-the-fly analysis, static cache simulation, and each of its applications, which will not be reiterated. Instead, further potential areas for applications shall be presented here.

The static cache simulator could be used for profiling to efficiently provide more detailed information than traditional profiling tools. Traditional profiles often rely on sampling methods, which are somewhat inaccurate and generally provide profiling data at the level of functions. Static cache simulation tracks the accurate frequency of execution not only at the level of functions but at the level of UPs. The actual frequency of basic blocks (and thereby of any instruction) can be inferred from the frequency of UPs. By annotating the generated code, instructions may be correlated to source-code statements. This can be used to provide the user with timing information in a source-code listing. A tool for precise and detailed profiling can be constructed around static cache simulation.

Static cache simulation also provides the means to produce measurements for prototyped machines. One example has already been given in Chapter 7 for the bit-encoding approach. In general, the cache behavior of non-existent architectures can be tested more efficiently. Traditional tools such as hardware simulators may provide analysis at a higher level of detail with regard to hardware components but are far less efficient than static cache simulation. Even inline tracing for cache simulation is slower than the method of static cache simulation, as shown in Chapter 6.

Finally, the method of static cache simulation does not have to interact with a compiler, although compiler support seems to facilitate the task. Yet, it is possible to analyze the control flow of an arbitrary executable and to modify the binary by inserting instrumentation code (similar to the work in [42, 10]). This would allow library code to be measured as well.

Chapter 11
Summary

This work presents a fresh look at the simulation of cache memories, provides an efficient framework for on-the-fly program analysis in general, and combines this framework with a new cache simulation technique for a number of applications.

On-the-fly program analysis instruments the code of a program to perform a specific analysis of the program during execution. In contrast, the most common tracing methods today separate the analysis from the program execution. While the problem of optimally profiling and tracing programs can be regarded as solved, on-the-fly analysis requires a different approach. A framework for efficient on-the-fly analysis is developed and proved correct as part of this work. This framework can be applied to any type of program analysis. This work discusses its application to cache analysis.

In the past, cache performance has often been analyzed using trace-driven methods. These methods record program traces at execution time and analyze the traces either concurrently or at a later time. Recently, on-the-fly analysis has been used to simulate the cache during program execution by instrumenting the program. To determine cache access hits and misses, it is recorded at run time which program or data line resides in a particular cache line.

This work introduces the technique of *static cache simulation* that statically predicts a large portion of cache references. The technique is formally derived for the simulation of direct-mapped instruction caches. It provides a novel view of cache memories. By analyzing the call graph and control-flow graphs of a program at compile time, some hits and misses can be determined statically. For the remaining program lines, a somewhat unorthodox view of the cache is taken. Rather than examining the contents of the global cache to determine if a program line is currently cached, a local state associated with a path (*i.e.*, a set of instructions) keeps track if the lines of this path have been cached. Instead of updating the hits and misses for each program line, a frequency counter associated with the current path state is incremented. Hits and misses can be inferred from the frequency counters after program termination.

Furthermore, static cache simulation determines the set of values a local state could have during execution. If this set is a singleton, the state is omitted during execution and the hits and misses are inferred statically for each iteration. The simulator decomposes the control-flow graph and the call graph, analyzes the cache behavior at this finer level by finding state transitions, and then recomposes the information to reduce the amount of instrumentation code.

Efficient on-the-fly analysis and static cache simulation are combined in this fashion for a number of applications that have been implemented and evaluated on the Sun SPARC architecture. The application of fast instruction cache analysis provides a new framework to evaluate instruction cache memories that outperforms even the fastest techniques published. Static cache simulation is also used to predict the caching behavior of real-time applications. This result disproves the conjecture that cache memories introduce unpredictability in real-time systems that cannot be efficiently addressed. While static cache simulation for instruction caches provides a certain degree of predictability for real-time systems, an architectural modification through bit-encoding is introduced that provides fully predictable caching behavior. Even for regular instruction caches without architectural modifications, tight bounds for the execution time of real-time programs can be derived from the information provided by the static cache simulator. Finally, the debugging of real-time applications is enhanced by displaying the timing information of the debugged program at breakpoints. The timing information is determined by simulating the instruction cache behavior during program execution and can be used, for example, to detect missed deadlines and locate time-consuming code portions. Overall, the technique of static cache simulation provides a novel approach to analyze cache memories and is shown to be very efficient for numerous applications.

Appendix A

Algorithm to Construct the Function-Instance Graph

Algorithm 3 (Construction of Function-Instance Graph)
Input: Let $G(V, EC)$ be a call graph where V is the set of functions (vertices) including an initial function *main* and EC is a set of pairs (e, c) of edges e and call sites c. The edge $e = v \rightarrow w$ denotes a call to w within v (excluding indirect calls but including recursive calls).
Output: The function-instance graph $FIG(W, F, B)$ where W is a set of function instances (vertices), F is a set of forward edges, and B is a set of backedges (due to recursive calls).
Algorithm:

```
PROCEDURE construct_FIG
  v: vertex;
BEGIN
  FOR all v in V DO
    v.visited:= FALSE;         /* initialize vertices: not visited      */
    v.last_instance:= -1       /*                      no func instance */
  END FOR;
  F:= {};                      /* initialize FIG components: empty sets */
  B:= {};
  W:= {main(0)};
  main.last_instance:= 0;      /* initialize instance of main to be 0   */
  dfs_traverse_CG(main)        /* perform recursive depth-1st-search    */
END construct_FIG;

PROCEDURE dfs_traverse_CG(v: vertex)
  w: vertex;
  i, k: INTEGER;
BEGIN
  i:= v.last_instance;
  v.visited:= TRUE;
  FOR all vertices w
      with (v --> w) in EC DO /* for each edge: v to w  */
    IF (w.visited) THEN       /* if visited then add to recursive edges*/
      k:= w.last_instance;
      B:= B + {v(i) --> w(k)} /* vertex v inst i to vertex w inst k    */
    ELSE                      /* otherwise add to non-recursive edges  */
      w.last_instance:=
        w.last_instance + 1;
      k:= w.last_instance;    /* new function instance of callee w     */
      W:= W + {w(k)};
      F:= F + {v(i) --> w(k)};/* vertex v inst i to vertex w inst k    */
      dfs_traverse_CG(w)
    END IF
  END FOR;
  v.visited:= FALSE
END dfs_traverse_CG;
```

Appendix B
Interface Specification of the PATH File

In Figure B.1, the specification for the interface file generated by the compiler for the control flow and instruction layout of paths is given using a BNF notation.

```
<file>          ::=  <funclist>.

<funclist>      ::=  <func> <funclist> |
                     .

<func>          ::=  -1 <funcname> \n <entrypaths> -1 \n
                     <other_paths> -1 0 \n .

<entrypaths>    ::=  <pathlist>.

<other_paths>   ::=  <pathlist>.

<pathlist>      ::=  <pathinfo> -1 \n <flowinfo> \n .

<pathinfo>      ::=  <thispath> <loopno> <loopfreq> <instlist>.

<instlist>      ::=  <instseq> <instlist> |
                     <instseq>.

<instseq>       ::=  <instoffset> <numinst>.

<flowinfo>      ::=  0 <pathlabelseq> -1 |
                     1 <funccall>.

<pathlabelseq>  ::=  <nextpath> <pathlabelseq> |
                     .

<funccall>      ::=  <funcname> <thispath> <nextpath>.

<thispath>      ::=  <pathlabel>.

<nextpath>      ::=  <pathlabel>.
```

Figure B.1: BNF Interface Specification for PATH File

The syntax and semantics of some of the symbols requires further explanation:

"\n" denotes a new line.

<funcname> is always preceded by an underscore "_".

<entrypaths> is a sequence of paths that can be reached from a call to the current function.

<pathlabel> is the numeric label of the path within the current function.

<**thispath**> refers to the current path.

<**nextpath**> refers to a successor path in the control flow.

<**instoffset**> denotes the byte offset of an instruction sequence. Each function starts with an offset 0.

<**numinst**> denotes the number of instructions of an instruction sequence.

Appendix C
Examples of Code Instrumentation

```
path3_cal_curstate:
        .word   12 ! SPS: 2 uncached lines at startup (1100b)
```

Figure C.1: State Table Entry

```
! typical method based on frequency counter array
path3_cal0_curstate: ! path 3 in function cal instance 0
        .word   0       ! AND mask
        .word   12      ! always hits  \
        .word   5       ! always misses > on each increment
        .word   0       ! first misses /
        .word   4       ! # shared path states
path3_cal0:          ! frequency counts (counter array):
        .word   0       !   hits on line a and b
        .word   0       !   miss line a, hit line b
        .word   0       !   hit line a, miss line b
        .word   0       !   misses on line a and b

! alternative method for large path states
path7_whet1_0_curstate:
        .word   4088    ! AND mask
        .word   23      ! hits
        .word   0       ! misses
        .word   1       ! first misses
        .word   -9      ! -(# shared path states)
path7_whet10:
        .word   0       ! general frequency counter
        .word   0       ! counter for misses due to conflicts
```

Figure C.2: Counter Table Entry

```
conf_table:
        .word   path6_cal0_curstate
        .word   0
        .word   path8_cal1_curstate
        .word   path7_cal1_curstate
        .word   0
        ...
```

Figure C.3: First Miss Table

The code emitted for call macros in Figure C.4 places the callee's base address (instance) in a register designated by the compiler. The compiler either chooses an unused register or spills an allocated register before the call. If the calling function of the macro has only one instance, two instructions suffice to load the register with a fixed address. (Notice that a set instruction is a synonym for the two instructions sethi and or.) In the case of multiple function instances of the caller, the register is loaded with the value of the callee's base address, which is determined by indexing with the caller's instance into a base address array. Figure C.4 depicts examples for both cases.

```
! current function has one function instance
#define CALL6_main(base_in, base_out) \
        set     inst_table_number0,%base_out

! current function has multiple function instances
#define CALL1_cal(base_in, base_out) \
        ld      [%base_in+8],%base_out
```

Figure C.4: Call Macros

The number of instructions generated for a path macro varies. Figures C.5 and C.6 depict examples for three cases. In the first case without conflicts, a frequency counter is

```
#define PATH1_cal(base, temp1, temp2) \
        ld      [%base+path1_cal0-inst_table_cal0],%temp2 ; \
        inc     %temp2 ; \
        st      %temp2,[%base+path1_cal0-inst_table_cal0] ; \

#define PATH4_cal(base, temp1, temp2) \
! increment counter in array index by SPS of current path \
        sethi   %hi(path3_cal_curstate),%temp1 ; \
        ld      [%temp1+%lo(path3_cal_curstate)],%temp2 ; \
        add     %temp2,%base,%temp1 ; \
        ld      [%temp1+path4_cal0-inst_table_cal0],%temp2 ; \
        inc     %temp2 ; \
        st      %temp2,[%temp1+path4_cal0-inst_table_cal0] ; \
! update SPS of current path \
        sethi   %hi(path3_cal_curstate),%temp1 ; \
        ld      [%temp1+%lo(path3_cal_curstate)],%temp2 ; \
        andn    %temp2,8,%temp2 ; \
        st      %temp2,[%temp1+%lo(path3_cal_curstate)] ; \
! update two conflicting SPSs of other paths \
        set     state_table,%temp1 ; \
        ld      [%temp1+path20_main_curstate-state_table],%temp2 ; \
        or      %temp2,4,%temp2 ; \
        st      %temp2,[%temp1+path20_main_curstate-state_table] ; \
        ld      [%temp1+path19_main_curstate-state_table],%temp2 ; \
        or      %temp2,4,%temp2 ; \
        st      %temp2,[%temp1+path19_main_curstate-state_table] ; \
```

Figure C.5: Path Macros (1)

simply incremented in three instructions. In the second case, conflicts are present. The SPS is used as an index into the frequency array, and the indexed counter is incremented. Then, the SPS is updated to reflect changes in the cached program lines. Thus, 12 instructions are required for updating the counter and SPS. In addition, three instructions are required for updating every conflicting SPS in the worst case. In the second example, there are two such conflicting SPSs. In the third case, the alternate code instrumentation is used. A loop counts the number of on-bits in the SPS combined with the AND mask. This number is added to the second counter entry while the first entry, the frequency counter, is simply

```
#define PATH16_whet1(base, temp1, temp2) \
! apply AND mask to SPS of current path \
        sethi   %hi(path6_whet1_curstate),%temp1 ; \
        ld      [%temp1+%lo(path6_whet1_curstate)],%temp2 ; \
        ld      [%base+path16_whet1_0_curstate-inst_table_whet10],%temp1 ; \
        andcc   %temp2,%temp1,%temp1 ; \
        bz      path16_whet1_nocnt;\
! count on-bits \
        mov     %g0,%temp2 ; \
path16_whet1_cnt: ; \
        bz      path16_whet1_ccnt;\
        btst    1,%temp1 ; \
        srl     %temp1,1,%temp1 ; \
        bz      path16_whet1_cnt ;\
        tst     %temp1 ; \
        b       path16_whet1_cnt ; \
        inc     %temp2 ; \
path16_whet1_ccnt: ; \
! add # of on-bits to accumulated # of conflicts which were misses \
        ld      [%base+4+path16_whet10-inst_table_whet10],%temp1 ; \
        add     %temp1,%temp2,%temp2 ; \
        st      %temp2,[%base+4+path16_whet10-inst_table_whet10] ; \
path16_whet1_nocnt: ; \
! increment the general frequency counter \
        ld      [%base+path16_whet10-inst_table_whet10],%temp2 ; \
        inc     %temp2 ; \
        st      %temp2,[%base+path16_whet10-inst_table_whet10] ;
        %\
! update the SPS of the current path \
        sethi   %hi(path6_whet1_curstate),%temp1 ; \
        ld      [%temp1+%lo(path6_whet1_curstate)],%temp2 ; \
        andn    %temp2,12,%temp2 ; \
        st      %temp2,[%temp1+%lo(path6_whet1_curstate)] ; \
```

Figure C.6: Path Macros (2)

incremented. Then, the SPS is updated to reflect changes in the cached program lines. Here, 16 instructions are required plus a maximum of 30 iterations of 7 instructions inside the bit-counting loop.

Appendix D

Cache Access Logic of the Fetch-From-Memory Bit

The access logic for an instruction cache using the proposed bit-encoded approach is illustrated in Figure D.1. The instruction memory contains the cached instructions. It is

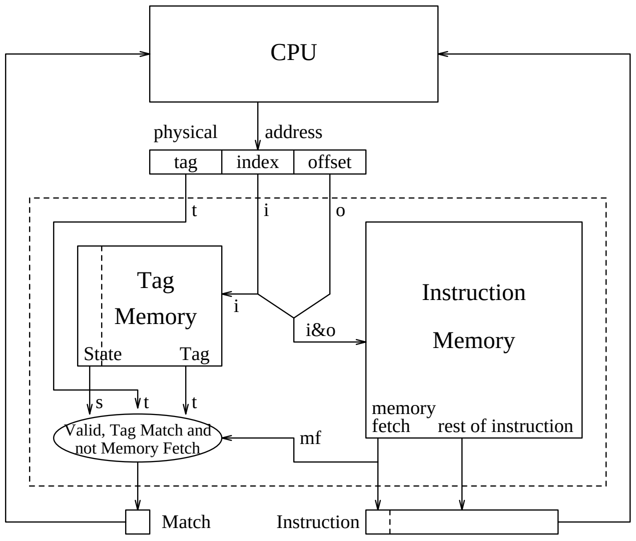


Figure D.1: Access Logic for Bit-Encoded Approach

accessed by using the index field to select the cache line and the offset field to select the instruction within that line. The tag memory contains the state bit and address tag for each cache line and is also accessed by using the index field. The match logic compares the tag of the instruction's physical address to the tag obtained by accessing the tag memory and verifies the state to ensure that the cache line is valid. In parallel, it also checks that the fetch-from-memory bit is clear. If any of these conditions are not met, then it informs the CPU to issue a main memory fetch. The logic to request a main memory fetch is not shown in this figure.

Appendix E
Interface Specification of the IST File

In Figure E.1, the specification for the interface file generated by the static cache simulator for instruction caches is given using a BNF notation.

```
<file>         ::=  <func> <file> |
                    <func>.

<func>         ::=  <func_header> <inst_seq>.

<func_header>  ::=  -1 func <funcname> instance <instno>
                    parent <parent> \n.

<inst_seq>     ::=  <inst_line> <inst_seq> |
                    <inst_line>.

<inst_line>    ::=  0 inst <instno> <loopno> <cache_line>
                    <cat_list> <cont> \n.

<parent>       ::=  <funcname> |
                    0.  /* for _main only */

<cat_list>     ::=  <nesting> <loop_cat>.

<loop_cat>     ::=  <worst_cat>/<best_cat> <loop_cat> |
                    <worst_cat>/<best_cat>.

<worst_cat>    ::=  <category>.

<best_cat>     ::=  <category>.

<category>     ::=  h | /* hit */
                    m | /* miss */
                    f | /* first miss */
                    i .  /* initial miss */

<cont>         ::=  1 \n call <funcname> instance <instno> |
                    0.  /* no function call */
```

Figure E.1: BNF Interface Specification for IST File

The syntax and semantics of some of the symbols requires further explanation:

"\n" denotes a new line.

<funcname> is preceded by an underscore "_", unless it is a static function.

<instno> starts with 0 for each function.

<**loopno**> indicates if an instruction belongs to a loop with <loopno> within a function. A <loopno> of 0 means no loop (outermost level of function).

<**cache_line**> denotes the cache line number (which can be used for handling first misses). First misses of different instructions with the same <cache_line> only cause one miss on the line during program execution.

<**nesting**> denotes the number of loop nesting levels (incl. functions) for which the prediction is listed. In trivial cases (always hit "h/h" and always miss "m/m"), the value will be 1 although the nesting might be deeper.

<**loop_cat**> is a sequence of doubles "worst/best" for each loop nesting level where any function is regarded as a separate loop nesting level.

Example: Assume a loop sequence main(loop1(func1(loop2(loop3())))).

```
0  inst  11  1  49  5   f/h    f/h    f/f    m/f    m/f   0

                        loop3  loop2  func1  loop1  main
```

Subsequent categories are provided for each loop nesting level (including function levels) in an inside-out order with respect to the nesting level. ■

References

[1] A. Adl-Tabatabai and Thomas Gross. Detection and recovery of endangered variables caused by instruction scheduling. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–25, June 1993.

[2] A. Agrawal, R. L Sites, and M. Horowitz. ATUM: A new technique for capturing address traces using microcode. In *International Symposium on Computer Architecture*, pages 119–127, 1986.

[3] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers – Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[4] R. Arnold. Bounding instruction cache performance. Master's thesis, Dept. of CS, Florida State University, December 1994. (to appear).

[5] R. Arnold, F. Mueller, D. B. Whalley, and M. Harmon. Bounding worst-case instruction cache performance. In *IEEE Symposium on Real-Time Systems*, December 1994. (accepted).

[6] T. P. Baker, F. Mueller, and Viresh Rustagi. Experience with a prototype of the POSIX "minimal realtime system profile". In *IEEE Workshop on Real-Time Operating Systems and Software*, pages 12–16, 1994.

[7] T. Ball and J. R. Larus. Optimally profiling and tracing programs. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 59–70, January 1992.

[8] M. E. Benitez and J. W. Davidson. A portable global optimizer and linker. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 329–338, June 1988.

[9] D. Bhatt, A. Ghonami, and R. Ramanujan. An instrumented testbed for real-time distributed systems development. In *IEEE Symposium on Real-Time Systems*, pages 241–250, December 1987.

[10] A. Borg, R. E. Kessler, and D. W. Wall. Generation and analysis of very long address traces. In *International Symposium on Computer Architecture*, pages 270–279, May 1990.

[11] G. Brooks, G. Hansen, and S. Simmons. A new approach to debugging optimized code. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–11, June 1992.

[12] B. Burgess, N. Ullah, P. Van Overen, and D. Ogden. The PowerPC 603 microprocessor. *Communications of the ACM*, 37(6):34–42, June 1994.

[13] G. Chartrand and L. Lesniak. *Graphs & Digraphs*. Wadsworth & Brooks, 2nd edition, 1986.

[14] C.-H. Chi and H. Dietz. Unified management of register and cache using liveness and cache bypass. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 344–355, June 1989.

[15] D. W. Clark. Cache performance in the VAX-11/780. *ACM Transactions on Computer Systems*, 1(1):24–37, February 1983.

[16] D. W. Clark and H. M. Levy. Measurement and analysis of instruction use in the VAX-11/780. In *Architectural Support for Programming Languages and Operating Systems*, pages 9–17, April 1982.

[17] B. Cogswell and Z. Segall. MACS: a predictable architecture for real time systems. In *IEEE Symposium on Real-Time Systems*, pages 296–305, December 1991.

[18] J. W. Davidson and D. B. Whalley. Quick compilers using peephole optimizations. *Software Practice & Experience*, 19(1):195–203, January 1989.

[19] J. W. Davidson and D. B. Whalley. Ease: An environment for architecture study and experimentation. In *SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 259–260, May 1990.

[20] J. W. Davidson and D. B. Whalley. A design environment for addressing architecture and compiler interactions. *Microprocessors and Microsystems*, 15(9):459–472, November 1991.

[21] S. J. Eggers and R. H. Katz. A characterization of sharing in parallel programs and its application to coherency protocol evaluation. In *International Symposium on Computer Architecture*, pages 373–382, 1988.

[22] S. J. Eggers, D. R. Keppel, E. J. Koldinge, and H. M. Levy. Techniques for efficient inline tracing on a shared-memory multiprocessor. In *SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 37–47, 1990.

[23] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 242–256, June 1994.

[24] R. Gerber and S. Hong. Semantics-based compiler transformations for enhanced schedulability. In *IEEE Symposium on Real-Time Systems*, pages 232–242, December 1993.

[25] S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof: A call graph execution profiler. In *Symposium on Compiler Construction*, pages 276–283, June 1982.

[26] S. L. Graham, P. B. Kessler, and M. K. McKusick. An execution profiler for modular programs. *Software Practice & Experience*, 21(11):25–40, December 1988.

[27] T. Hand. Real-time systems need predictability. *Computer Design (RISC Supplement)*, pages 57–59, August 1989.

[28] M. Harmon, T. P. Baker, and D. B. Whalley. A retargetable technique for predicting execution time. In *IEEE Symposium on Real-Time Systems*, pages 68–77, December 1992.

[29] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1990.

[30] C. R. Hill. A real-time microprocessor debugging technique. In *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, pages 145–148, 1983.

[31] M. Hill. A case for direct-mapped caches. *IEEE Computer*, 21(11):25–40, December 1988.

[32] U. Hoelzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 32–43, June 1992.

[33] M. Huguet, T. Lang, and Y. Tamir. A block-and-actions generator as an alternative to a simulator for collecting architecture measurement. In *ACM SIGPLAN Symposium on Interpreters and Interpretive Techniques*, pages 14–25, June 1987.

[34] D. Kerns and S. Eggers. Balanced scheduling: Instruction scheduling when memory latency is uncertain. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 278–289, June 1993.

[35] D. B. Kirk. SMART (strategic memory allocation for real-time) cache design. In *IEEE Symposium on Real-Time Systems*, pages 229–237, December 1989.

[36] E. Kligerman and A. Stoyenko. Real-time euclid: A language for reliable real-time systems. *IEEE Transactions on Software Engineering*, SE-12(9):941–949, September 1986.

[37] D. E. Knuth. An empirical study of FORTRAN programs. *Software Practice & Experience*, 1:105–133, 1971.

[38] D. E. Knuth. *The Art of Computer Programming: Fundamental Algorithms*, volume 2. Addison Wesley, 2 edition, 1973.

[39] D. E. Knuth and F. R. Steverson. Optimal measurement points for program frequency counts. *BIT*, 13:313–322, 1973.

[40] S. Laha, J. H. Patel, and R. K. Iyer. Accurate low-cost methods for performance evaluation of cache memory systems. *IEEE Transactions on Computers*, 37(11):1325–1336, November 1988.

[41] J. R. Larus. Abstract execution: A technique for efficiently tracing programs. *Software Practice & Experience*, 13(8):671–685, August 1983.

[42] J. R. Larus and T. Ball. Rewriting executable files to measure program behavior. TR 1083, University of Wisconsin, March 1992.

[43] M. Lee, S. Min, C. Park, Y. Bae, H. Shin, and C. Kim. A dual-mode instruction prefetch scheme for improved worst case and average program execution times. In *IEEE Symposium on Real-Time Systems*, pages 98–105, December 1993.

[44] T. H. Lin and W. S. Liou. Using cache to improve task scheduling in hard real-time systems. In *IEEE Workshop on Architecture Supports for Real-Time Systems*, pages 81–85, 1991.

[45] S. McFarling. Program optimization for instruction caches. In *Architectural Support for Programming Languages and Operating Systems*, pages 183–191, April 1989.

[46] F. Mueller and D. B. Whalley. Efficient on-the-fly analysis of program behavior and static cache simulation. In *Static Analysis Symposium*, September 1994. (accepted).

[47] F. Mueller and D. B. Whalley. Fast instruction cache analysis via static cache simulation. TR 94-042, Dept. of CS, Florida State University, April 1994.

[48] F. Mueller and D. B. Whalley. On debugging real-time applications. In *ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, June 1994.

[49] F. Mueller and D. B. Whalley. Real-time debugging by minimal hardware simulation. In *PEARL Workshop über Realzeitsysteme*, December 1994. (accepted).

[50] F. Mueller, D. B. Whalley, and M. Harmon. Predicting instruction cache behavior. In *ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, June 1994.

[51] Frank Mueller. A library implementation of POSIX threads under UNIX. In *Proceedings of the USENIX Conference*, pages 29–41, January 1993.

[52] D. Niehaus. Program representation and translation for predictable real-time systems. In *IEEE Symposium on Real-Time Systems*, pages 53–63, December 1991.

[53] D. Niehaus, E. Nahum, and J. A. Stankovic. Predictable real-time caching in the spring system. In *IEEE Workshop on Real-Time Operating Systems and Software*, pages 80–87, 1991.

[54] C. Y. Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Systems*, 5(1):31–61, March 1993.

[55] B. L. Peuto and L. J. Shustek. An instruction timing model of CPU performance. In *International Symposium on Computer Architecture*, pages 165–178, March 1977.

[56] A. Poursepanj. The PowerPC performance modeling methodology. *Communications of the ACM*, 37(6):47–55, June 1994.

[57] P. Rowe and B. Pagurek. Remedy: A real-time, multiprocessor, system level debugger. In *IEEE Symposium on Real-Time Systems*, pages 230–239, December 1987.

[58] A. D. Samples. *Profile-Driven Compilation.* PhD thesis, University of California at Berkley, September 1992.

[59] V. Sarkar. Determining average program execution times and their variance. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 298–312, June 1989.

[60] D. Simpson. Real-time RISCS. *Systems Integration*, pages 35–38, July 1989.

[61] K. So, F. Darema, D. A. George, V. A. Norton, and G. F. Pfister. PSIMUL – a system for parallel execution of parallel programs. *Performance Evaluation of Supercomputers*, pages 187–213, 1988.

[62] C. Stunkel and W. Fuchs. Trapeds: Producing traces for multicomputers via execution driven simulation. In *SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 70–78, May 1989.

[63] C. B. Stunkel. Address tracing for parallel machines. *IEEE Computer*, 24(1):31–38, January 1991.

[64] Sun Microsystems, Inc. *Programmer's Language Guide*, March 1990. Part No. 800-3844-10.

[65] SunSoft, Inc. *SunOS 5.1 User Commands*, March 1992.

[66] M. Timmerman, F. Gielen, and P. Lambix. A knowledge-based approach for the debugging of real-time multiprocessor systems. In *IEEE Workshop on Real-Time Applications*, pages 23–28, 1993.

[67] H. Tokuda, M. Kotera, and C. W. Mercer. A real-time monitor for a distributed real-time operating system. In *ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 68–77, 1988.

[68] D. B. Whalley. Fast instruction cache performance evaluation using compile-time analysis. In *SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 13–22, June 1992.

[69] D. B. Whalley. Techniques for fast instruction cache performance evaluation. *Software Practice & Experience*, 19(1):195–203, January 1993.

[70] C. A. Wiecek. A case study of VAX-11 instruction set usage for compiler execution. In *Architectural Support for Programming Languages and Operating Systems*, pages 177–184, March 1982.