Compositional Static Instruction Cache Simulation *

Kaustubh Patil
VMware, Inc.
3145 Porter Drive
Palo Alto, CA 94304 USA

Kiran Seth
Qualcomm, Inc.
2000 Center Green Way
Cary, NC 27513

Frank Mueller
Department of Computer Science/
Center for Embedded Systems Research
North Carolina State University
452 EGRC, Raleigh, NC 27695-7534

mueller@cs.ncsu.edu, phone: +1.919.515.7889, fax: +1.919.515.7925

ABSTRACT

Scheduling in hard real-time systems requires a priori knowledge of worst-case execution times (WCET). Obtaining the WCET of a task is a difficult problem. Static timing analysis techniques approach this problem via path analysis, pipeline simulation and cache simulation to derive safe WCET bounds. But such analysis has traditionally been constrained to only small programs due to the complexity of simulation, most notably the complexity of static cache simulation, which requires inter-procedural analysis.

This paper describes a novel approach of compositional static cache simulation that alleviates the complexity problem, thereby making static timing analysis feasible for much larger programs than in the past. Specifically, a framework is contributed that facilitates static cache analysis by splitting it into two steps, a module-level analysis and a compositional phase, thus addressing the issue of complexity of inter-procedural analysis for an entire program. The module-level analysis parameterizes the data-flow information in terms of potential evictions from cache due to calls containing conflicting references. The compositional analysis stage uses the result of the parameterized data-flow for each module. Thus, the emphasis here is on handling most of the complexity in the module-level analysis and performing as little analysis as possible at the compositional level. The experimental results for direct-mapped instruction caches show that the compositional analysis framework outperforms prior analysis methods for larger programs by one to two orders of magnitude, depending on the reference for comparison, while providing equally accurate predictions. This novel approach to static cache analysis provides a promising solution to the complexity problem in timing analysis, which, for the first time, makes the analysis of larger programs feasible.

*This work was supported in part by NSF grants CCR-0208581, CCR-0310860 and CCR-0312695. This paper does not necessarily reflect or represent the views of VMware, Inc. or Qualcomm, Inc.

LCTES'04, June 11–13, 2004, Washington, DC, USA.

Categories and Subject Descriptors

D.4.1 [**Operating Systems**]: Process Management—*scheduling*; D.4.7 [**Operating Systems**]: Organization and Design—*real-time systems and embedded systems*

General Terms

Algorithms, Experimentation

Keywords

Real-Time Systems, Caches, Scheduling, Worst-Case Execution Time

1. INTRODUCTION

Timely execution is of essence in real-time systems. Particularly in hard real-time systems, a violation of temporal constraints may have irreparable effects on the controlled system, its environment or both. The theory of real-time systems reasons about the feasiblility of executing a task set, *i.e.*, offline schedulability tests may determine if all deadlines of a set of tasks can be met. To obtain such guarantees, task parameters have to be specified, such as the period of each task and its worst-case execution time (WCET). On the one side, the period is typically derived from the operating environment, such as temporal constraints on sensors and actuators. On the other side, determining the WCET is a non-trivial task due to software complexity, unknown worst-case inputs for even moderately complex computational tasks and hardware complexity with unpredictable execution behavior.

Experimental approaches to determine the WCET are either considered unsafe or constrained to a probabilistic approach [28, 4]. In contrast, static analysis methods have been developed to derive safe WCET bounds. Such static analysis tools model hardware components, *e.g.*, the processor pipeline and caches, and they consider the longest control-flow paths through the inter-procedural program representation to obtain an upper bound on the number of cycles for any execution. The complexity of cycle-level simulations for entire programs currently restricts the feasibility of analysis tools, particularly that of static cache analysis, to small programs. Computationally complex inter-procedural analysis is needed to determine caching effects, which depend on knowledge of data and instruction references.

This paper presents a novel framework to perform worst-case static cache analysis for direct-mapped instruction caches. Instead of a single integrated inter-procedural analysis phase, the new approach combines 1) a module-level analysis and 2) a compositional

step. By constraining the analysis to smaller units in the first step and deferring composition to a second step, the complexity problem of inter-procedural analysis is addressed for an entire program. The module-level analysis parameterizes the data-flow information in terms of potential evictions from cache due to calls containing conflicting references. The compositional analysis stage uses the result of parameterized data-flow for each module. The objective of this approach is to constrain most of the complexity to the module level and, thereby, enable a significant reduction in complexity at the compositional level of the analysis. The experimental results show that the compositional analysis outperforms the integrated inter-procedural approach for larger programs while providing equally accurate predictions.

2. TRADITIONAL STATIC CACHE ANALYSIS

This section introduces a traditional approach to static cache analysis [20] before motivating the need for a compositional approach. Static cache simulation is an analysis technique performed by a tool to assess the worst-case cache behavior of memory references within a program at program analysis (static) time. Static cache analysis integrates into a set of tools, depicted in Figure 1, that provide safe upper bounds on the worst-case execution time (WCET) of programs. These tools range from a modified compiler that, besides code, emits program information for the analysis phases of our tools. The analyzed programs are assumed to have know upper bounds for each loop, bounded recursion while heap allocation and indirect calls should be absent. The static cache simulator classifies references into categories to capture their cache behavior. The timing analyzer combines cache categorizations with program information to simulate the pipeline of an architecture and consider execution through any possible paths in the control flow. The details of path analysis and pipeline simulation can be found elsewhere [2]. As a result of our toolset, an upper bound for the WCET is provided, which is then used in real-time schedulability analysis to determine if a task set is feasible, *i.e.*, if it is guaranteed to meet all deadlines, *e.g.*, with rate-monotone or EDF scheduling [16, 17].

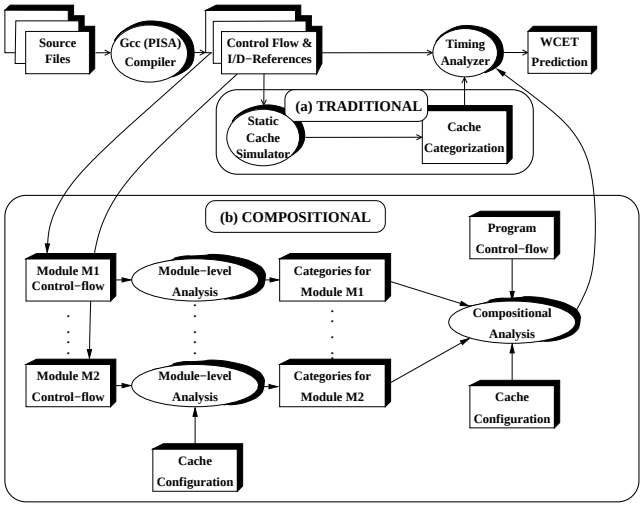

Figure 1: Timing Toolset w/ Static Cache Analysis: (a) Traditional vs. (b) Compositional

The challenge in performing static cache analysis is to represent possible cache states during simulation as succinctly as possible. Even though traditional static cache analysis, depicted in Figure 1(a), provides a cache abstraction for storage purposes, it still requires considerable memory space to perform inter-procedural data-flow analysis. This results in both space and time overhead that has made the analysis of even moderately large codes infeasible in practice. The objective of compositional cache analysis is to overcome the complexity barrier for larger programs while retaining the existing accuracy of traditional cache analysis. In the following, a brief overview of traditional cache analysis is given.

Throughout the rest of this paper, the discussion is constrained to the analysis of instruction caches. Data cache analysis based on data-flow techniques is believed to profit in a similar manner from compositional analysis since the compositional model only affects the data-flow framework, not the memory reference categorizations [29].

The objective of static cache simulation is to categorize every instruction reference before actual program execution for a specific cache configuration. The traditional approach consists of three phases: 1) A control-flow graph for the whole program is constructed. 2) The graph is analyzed to determine the program lines that can possibly be cached before entering each basic block of the program. 3) This information is used to categorize each instruction reference.

Inter-procedural analysis: A function instance graph is constructed from the control-flow graph of each function and the call graph of the program. This function instance graph distinguishes calling contexts as distinct function instances. For example, a function `foo` called from `main` in two distinct basic blocks results in two instances of `foo` in the function instance graph.

Instruction Reference Categorization: An instruction reference is categorized based on the abstract cache state (see below) of the basic block that contains the instruction. For each loop and function nesting level, such a category is derived per instruction based on these terms:

Potentially cached line: A program line can potentially be cached before entering a basic block if there exists a sequence of control-flow such that the line is cached when the basic block is entered along that control-flow.

Abstract cache state (ACS): An ACS of a basic block in a function instance is the subset of all program lines that can potentially be cached before entering that basic block.

Based on these definitions and data-flow information obtained for the function instance graph, each instruction reference is categorized into one of the following categories:

- **Always-Miss**: An instruction is categorized as an *always miss* if it cannot be guaranteed to be in cache for that reference. An *always miss* is predicted when the instruction is the first reference to a program line in the basic block and the program line is not in the ACS of that basic block.
- **Always-Hit**: An instruction is categorized as an *always hit* if it is guaranteed to be in cache for that reference. An *always hit* is predicted when other instructions in the basic block have already accessed the same program line or the program line is in the ACS, and no other conflicting program line is in the ACS.
- **First-Miss**: An instruction is categorized as a *first miss* if it cannot be guaranteed to be in cache for its first reference when the loop is entered, but it is guaranteed to be cached for all later iterations of that loop.
- **First-Hit**: An instruction is categorized as a *first hit* if it is guaranteed to be in cache the first time it is accessed when the

loop is entered, but cannot be guaranteed for later iterations of that loop.

These categorizations have been formalized in the past and were demonstrated to yield tight and safe WCET bounds for relatively small programs [21]. Other work on WCET prediction includes different data-flow frameworks [7] and integrated timing analysis frameworks with cache models [18, 14], all of which reported results for small benchmarks due to analysis overhead.

3. COMPOSITIONAL CACHE ANALYSIS

To overcome the complexity barrier that currently constrains traditional cache analysis to small programs, a compositional approach is proposed that separates inter-procedural data-flow analysis over the function instance graph into two stages, as depicted in Figure 1(b). The first phase, module-level analysis, derives categorizations from the control-flow information for a module in isolation. Traditional static cache simulation is slightly modified for this task. Since absolute addresses of instruction references are unknown at this stage, the cache behavior is captured in four analysis scopes for each module. In a second stage, the absolute address information for each module is combined with the four analysis scopes by performing a limited inter-procedural analysis to derive final instruction categorizations. This approach not only results in significantly less inter-procedural analysis overhead during module composition, it even reduces the total overhead when considering both analysis stages compared to the traditional approach for larger programs.

Assumptions: The compositional approach requires the cache configuration, supplied to both analysis stages, to be identical. Furthermore, worst-case analysis for direct-mapped instruction caches is supported by the current implementation. The design can be extended to set-associative caches similar to previous work [21]. Finally, each module should be aligned at cache line size. This is typically already performed by an `align` directive emitted by the compiler, hence, it is a realistic constraint. If not present, a module can be augmented by an alignment directive. Alignment is not a necessary but rather a simplifying assumption to reduce the overhead involved in the module-level analysis. This reduces the module analysis overhead from n analyses for any possible alignment to just one analysis, where $n = |words\ per\ cache\ line|$ for RISC architectures and $n = |bytes\ per\ cache\ line|$ for CISCs.

3.1 Module-level Analysis

In the first stage, instructions are categorized only on the module level (assuming cache line size alignment). At this level of analysis, no information is available about called modules, if any, except for the name of the modules. Data-flow equations for deriving the abstract cache state are identical for the traditional inter-procedural approach and the modular approach. However, two new states are introduced in the data-flow representation.

C: Within module-level analysis, a must-conflict line is replacing any existing line in an abstract cache state (ACS) prior to a call in the ACS after the call if calls are considered in the analysis.

MC: Within compositional analysis, a may-conflict line is replacing a non-trivial subset of lines in an exit ACS of a module, *i.e.*, when due to conditional execution different program lines may be resident in a specific cache line.

In the following, four analysis scopes are described, which are performed on each module. Each scope differs in its context of references and the resulting cache effects for loops and calls: Scope one captures the absence of loop/callee conflicts, scope two captures conflicts with callees but ignores loop conflict, scope 3 captures loop conflicts but not conflicts with callees, and scope 4 encompasses both loop and callee conflicts. Later on, compositional analysis picks the correct scope analysis to lift intra-module cache analysis to the interprocedural level. A sequence of examples describes each analysis step as it is performed for a function `bar` that a) is called by `main` and b) is calling `foo`, as depicted in Figure 2. The dotted backedge from the exit(s) of `bar` to its entry is optional, as explained later. The instruction cache is assumed to have four cache lines in the example. The 4-entry ACS sets for the modules in the figure are listed in Table 1 for each analysis scope, as discussed below.

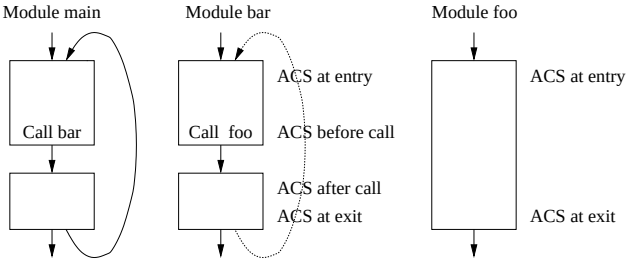

Figure 2: Flow Graphs for Module-level Analysis

Scope 1: Analysis without backedge, ignoring calls: While calculating ACS sets by solving the corresponding data-flow equations iteratively for a module, it is assumed in this scope that a call in a predecessor block will not affect the incoming abstract cache state for the basic block. This is equivalent to assuming that every line present in the ACS preceding the call will be retained in cache until after the call. Hence, the callee has no conflicting cache line. As an example, consider `bar` in Figure 2 without the dashed backedge and recall that calls are ignored. Let `foo` contain a reference to program line 9 (without conflicts). The relevant ACS sets for this first scope of modular analysis are depicted in column 2 of Table 1 and indicate that program line 4 is cached prior and after the call to `foo` inside `bar`. The corresponding traditional analysis for the same calling sequence (without backedge for `bar`) would result in the ACS sets indicated in the third and fourth column of Table 1 for `bar` and `foo`, respectively. The compositional stage, detailed later, explains how to derive the same states as in the traditional analysis from the modular analysis scopes, in this case scope 1.

Scope 2: Analysis without backedge, considering calls: In this type of analysis, it is assumed that a call affects the ACS such that there is a conflicting line brought into cache by the callee. In other words, the callee module caches its own lines and evicts cache lines in the ACS of the caller. Since the absolute address information for the lines of the called module is not known during module-level analysis, we indicate this by a **C** (must-conflict) in the ACS, as defined previously. Consider Figure 2 without backedge for `bar` but with the call to `foo`. Let `foo` contain a reference to program line 8 (conflicting with program oine 4). Table 1 column 5 indicates a must-conflict as C_0 in the ACS after the call, *i.e.*, a must-conflict for a reference mapping to cache line 0. Column 6 and 7 indicate a corresponding situation where traditional analysis evicts line 4 (at entry) in `foo` with line 8 (at exit), which is in `bar` after the call. By parameterizing the module-level ACS with a must-conflict, compositional analysis uses this scope 2 if a callee contains an evictor in its exit state.

Scope 3: Analysis with backedge, ignoring calls: In the fol-

	Scope 1			Scope 2			Scope 3			Scope 4		
ACS	mod. bar	trad. bar	trad. foo	mod. bar	trad. bar	trad. foo	mod. bar	trad. bar	trad. foo	mod. bar	trad. bar	trad. foo
at entry			{4}			{4}	ϕ,{6}		{4}	ϕ,{6}		{4}
before call	{4}	{4}		{4}	{4}		{4}	{4}		{4}	{4}	
after call	{4}	{4,9}		{C_0}	{8}		{4}	{4,9}		{C_0}	{8}	
at exit			{4,9}			{8}	{6}		{4,9}	{6}		{8}

Table 1: Abstract Cache States for Analysis Scopes

lowing two scopes, the module-level control-flow graph is augmented by backedge(s) from exit(s) to the entry block. Such a virtual backedge simulates the effect on data-flow analysis if the module is repeatedly called within a loop. Scope 3 adds this/these backedge(s) but ignores calls. Figure 2 illustrates the backedge as a dotted edge in the control-flow graph of `bar`. Let `foo` contain a reference to program line 9 (without conflicts). Table 1 depicts the ACS for module-level analysis in Column 8. At the first call, the ACS is the empty set (ϕ) while for consecutive calls, program line 6 is cached along the backedge from the last call to `bar`. Line 4 is in cache prior and after the call to `foo` since this analysis disregards calls. Traditional analysis considers this call, which brings program line 9 into cache (Columns 9 and 10 of Table 1). This additional caching effect is considered in the compositional stage for our novel analysis technique.

In this analysis, ACS sets of the exit blocks of the module propagate along the added backedges to the entry block of the module. Hence, the categorizations from this analysis are obtained with the same effect as if this module is being called from a loop in another module, and the control returns to this module from the calling loop for each loop iteration except for the first iteration. Calls are ignored during this analysis. This simulates calls within this module that do not have any effect on the incoming ACS of the successors of the calling basic block.

Scope 4: Analysis with backedge, considering calls: This scope also adds backedges from exits to the entry block. Additionally, effects of calls are considered. The objective of this scope is to simulate a module M being called from a loop of another module, where M has itself a call that affects the ACS following this call. Figure 2 illustrates this situation, where the dotted backedge and calls are considered. Let `foo` contain a reference to program line 8 (conflicting with program line 4). Table 1 further depicts the ACS effects in Columns 11 and 12/13 for modular and traditional analysis, respectively. Due to repeated calls, line 6 may be retained in the module-level analysis and line 4 is replaced by another line due to the call to `foo`. Traditional analysis identifies this replacement as line 8. This effect is again captured by the compositional stage in our approach.

In addition to these four different types of analysis scopes, module-level analysis records for each basic block the loop numbers for any enclosing loop(s) and for each loop the set of basic blocks contained in the loop. This information is subsequently used by the compositional analysis stage.

3.2 Compositional analysis

Compositional analysis is performed on an entire program when the absolute address information for each module is available.

Initial predicted categorizations: For each instruction of each module in the program, it is assumed that the module-level categorizations ignoring calls are the initial categorizations. These are optimistic categorizations in the absence of temporal locality across module boundaries (due to calls). Subsequently, necessary adjustments are applied to these categorizations during the compositional stage. Two adjustments are necessary for each line in each module:

Adjustments as a caller: When a basic block B has a predecessor P and P contains a call, then the ACS in block B is adjusted based on *module-level analysis considering calls*.

Adjustments as a callee: When a module M is being called within a loop L of another module, then during calls to M within iterations $2..n$, lines of M may still be cached (but not necessarily during the first iteration). Hence, for those lines, adjustments are based on *module-level analysis with added backedge(s)*.

Bottom-up processing: The algorithmic details are described in the following. In the compositional analysis stage, a limited amount of analysis is done in order to obtain the inter-procedural relationship in terms of the cached lines passed across the module boundaries during calls and call returns. The modules are processed starting from the leaf module in a call graph and gradually traversing the graph toward the root, *i.e.*, the `main` function. The algorithm for bottom-up processing over basic blocks is given in Figure 3.

The algorithm traverses modules in a bottom-up order exactly once per module. The complexity of the algorithm is $O(nf \times nlf \times nbl \times npb)$ where *nf* is the number of modules, *nlf* is the maximum number of loops in any one module, *nbl* is the maximum number of basic blocks in any one loop and *npb* is the maximum number of predecessors of a basic block. Thus, the complexity of this algorithm that captures the inter-procedural analysis information is constrained in that it does not re-analyze module-intrinsic details, *i.e.*, callees need not be reconsidered after an initial pass. The set of program lines and conflicting program lines of a block are denoted as conf_lines and conf_prog_lines, respectively (see [23] for details).

The algorithm determines the lines of each loop and a unique loop header. The must_exit_lines for a module (line 47 of Fig. 3) represent lines that *are guaranteed* to be in cache when the control of execution returns from the module. Conversely, may_exit_lines are lines of a module that *might* be present in the cache on exit from the module (but their presence cannot be guaranteed), denoted as **MC** (a May-Conflict line, as defined previously).

Consider Figure 4. When control is transitioned to the module `getbit`, line 43 may or may not be cached as an exit line. If a block B in a module `des` calls `getbit`, then for the successor S of block B, it is necessary to consider the fact that, when the control returns to S, line 43 of `getbit` might be in the cache to capture the worst-case scenario. Hence, this line is an **MC** line in the exit ACS of `getbit`.

For each basic block and for each surrounding loop, bottom-up processing gathers the lines that must be in cache before entering the block (*must_in_lines* in Figure 3) and also the lines that might be in cache (*may_in_lines*). Line 15 of the algorithm shows the formula for calculating the *must_out_lines* for a basic block if the block has a call. All the lines in the *must_out_lines* calculated on

```
for (each function F)
  if (isleaf_function || all children processed) { // for bottom-up order
    for (each loop L in F) {
      /* Calculations for in lines of each block of this loop */
      for (each block B in the loop L) {
        inter_out = full set;
        union_out = φ;
        for (each predecessor P of B in loop L) {
          if ( B!=L → loopheader)
            if (backedge(P-to-B)) // P-to-B form a loop inside L.
              continue; // Disregard this predecessor.

12:         must_out_lines(P)=must_in_lines(P) \
              conf_prog_lines(P) ∪ prog_lines(P);
          if (iscaller(P)) {
            function C = P→callee;
15:           must_out_lines(P)=must_out_lines(P) \
                conf_prog_lines(C→may_exit_lines) ∪
                prog_lines(C→must_exit_lines);
          }
          inter_out=must_out_lines(P) ∩ inter_out;
          may_out_lines(P)=may_in_lines(P) \
            conf_prog_lines(P) ∪ prog_lines(P);
          if (iscaller(P)) {
            function C = P→callee;
22:           may_out_lines(P)=may_out_lines(P) \
                conf_prog_lines(C→must_exit_lines) ∪
                prog_lines(C→may_exit_lines);
          }
          union_out=may_out_lines(P) ∪ union_out;
        }
        inter_out=must_in_lines(B);
        union_out=may_in_lines(B);
      } // Processing for block B complete

      /* Adjustments applied as a caller */
31:   for (each block B in L) {
        for (each line l in B)
          if (l conflicts with some line in in_line(B))
            for (each instruction in l)
              normal_category = module-level cat considering call;
      }
      /* Calculations of loopheader and lines within this loop */
      find loopheader of loop L;
      lines(loop L) = cache all the lines of each block of this loop;
      if (loop L has calls)
        for (each callee)
          lines(loop L) = lines(loop L) ∪ lines(callee);

    } // Processing for loop L complete

    // calculate the must and may exit_lines for this function.
47:  F→must_exit_lines = full set;
    F→may_exit_lines = empty set;
    for (each exit block E) {
50:   F→must_exit_lines ∩= must_in_lines(E) \
        conf_prog_lines(E) ∪ prog_lines(E);
      F→may_out_lines ∪= may_in_lines(E) \
        conf_prog_lines(E) ∪ prog_lines(E);
    }
  }
```

Figure 3: Algorithm for Bottom-up Processing

line 12 that are in conflict with any of the *may_exit_lines* of the callee are subtracted. Note that the set *may_exit_lines* of the callee is referenced here to ensure that the set *must_out_lines* calculated is safe and accurate. Then, all the lines in *must_exit_lines* of the callee are added, because these lines must be in cache when the control returns from the callee. The formula on line 22 similarly calculates the *may_out_lines*.

As can be seen from line 31 of this algorithm, this information is used to adjust the categorizations for a module when it acts as

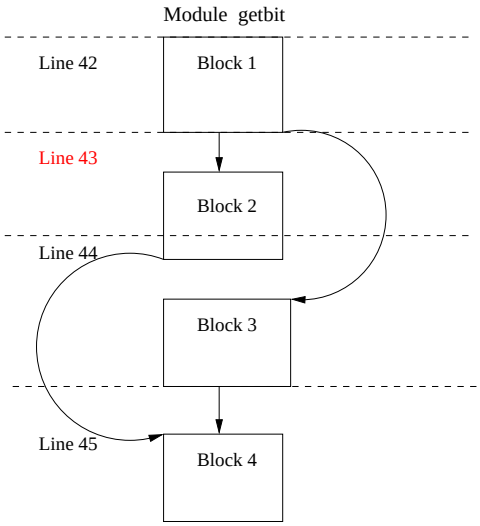

Figure 4: Module with Two Paths in Exit Lines

a caller. When a block has a call to another module, its successor block might have some lines of the calling module as its incoming lines. In that case, the categorizations for those lines of the successor that conflict with incoming lines of the module need to be adjusted. This is done using the *module-level analysis considering calls*. For adjusting the categorizations with added backedge(s), a similar analysis (not described in the figure) is performed for each function and each loop, but with an added backedge from each exit block to the entry block of the function. Thus, the adjustments to the initial categorizations required for a module when a module acts as a caller are applied during bottom-up processing. It is important to understand here that these adjustments will change the categorizations from an overly optimistic to a realistic and safe (conservative) worst-case value.

A module may alternatively represent a callee. In that case, some lines of the module may still be in cache when the flow of control returns to the module on every loop iteration, except for state prior to the first iteration. This will cause subsequent hits for such lines. These adjustments are applied during the second pass when each instruction is categorized *w.r.t.* each loop it lies in and each instance of the module. This is achieved by using information of the incoming lines for each calling block of the module and also the *adjusted module-level analysis with added backedge(s)*, as explained in the following.

Adjustments Inside Callees: During this second and final pass of the compositional analysis, categorizations of each instruction might be adjusted depending upon whether the module is called by some other module or not, algorithmically depicted in Figure 5. This algorithm is used for each line l in the program.

In the *do while();* loop of this algorithm, we determine if line l is still in cache when the control returns to line l of the module from the call within loop L. If so, the line will be a hit, consequently, the categorizations from the simulated analysis with added outer loops is used.

In summary, the novel compositional static cache categorizations approach operates in two stages. In the first stage, analysis is done on module-levels without knowledge of the absolute addresses of the module and those of the called modules. In the second stage, a two-pass approach is used. In the first pass, information is gathered in a bottom-up manner about incoming lines for each basic block and the exit lines of each module. Adjustments to the initial

```
// Algorithm applied to a particular instance of a module.

for (each loop L with header H the line l lies in) {
  l_in_conflict=FALSE;
  calling block C = basic block that contains l;
  do {
    if (l conflicts w/ any line in
        must_in_lines(C) ∪ may_in_lines(C)) {
      l_in_conflict = TRUE;
      break;
    } // next, backtrace by walking up call chain:
    C = caller block of C for current instance of module;
  } while(C → currentmodule != H → currentmodule);

  for (each instruction i of l)
    if (!l_in_conflict)
      i.finalcat = adjusted categorizations obtained
        from module-level analysis with added backedges;
    else
      i.finalcat = adjusted categorizations obtained from
        from module-level analysis without added backedges;
}
```

Figure 5: Adjustments for Final Categorizations

categorizations when the module acts as a caller are applied in this pass. In the second pass, each instruction is categorized for each loop level and each instance of the module. Also, adjustments are applied when the module acts as a callee. The complexity is kept at module level for the compositional approach, *i.e.*, none of the callees have to be re-evaluated.

4. EXAMPLE

The following example shows the difference in calculations and categorizations between the traditional and compositional analysis schemes. The example depicted below has a conditional call to illustrate the effects of alternate execution paths on the analysis schemes.

```
#include<stdio.h>
int a[] = {1,2,3,4,5,6,7,8,9,10};

int value (int i) {
  return a[i];
}

int main() {
  int sum = 0, i;
  for(i=0; i<10; i++)
    if(i)
      sum += value(i);
  return sum;
}
```

The corresponding control-flow graph is depicted in Figure 6. The figure also lists the categorizations based on different calculations. In the calculations, line **I** indicates an invalid line. When the analysis begins, it is assumed that all the lines of the ACS are invalid lines. Also, an ACS denoted as [*a b c d*] indicates that program line *a* is mapped to cache line 0, line *b* is mapped to cache line 1 etc. We consider the ACS at the basic block level. For each basic block, we provide `in` and `out` sets indicating the ACS entering and exiting the block. The cache is assumed to contain four lines.

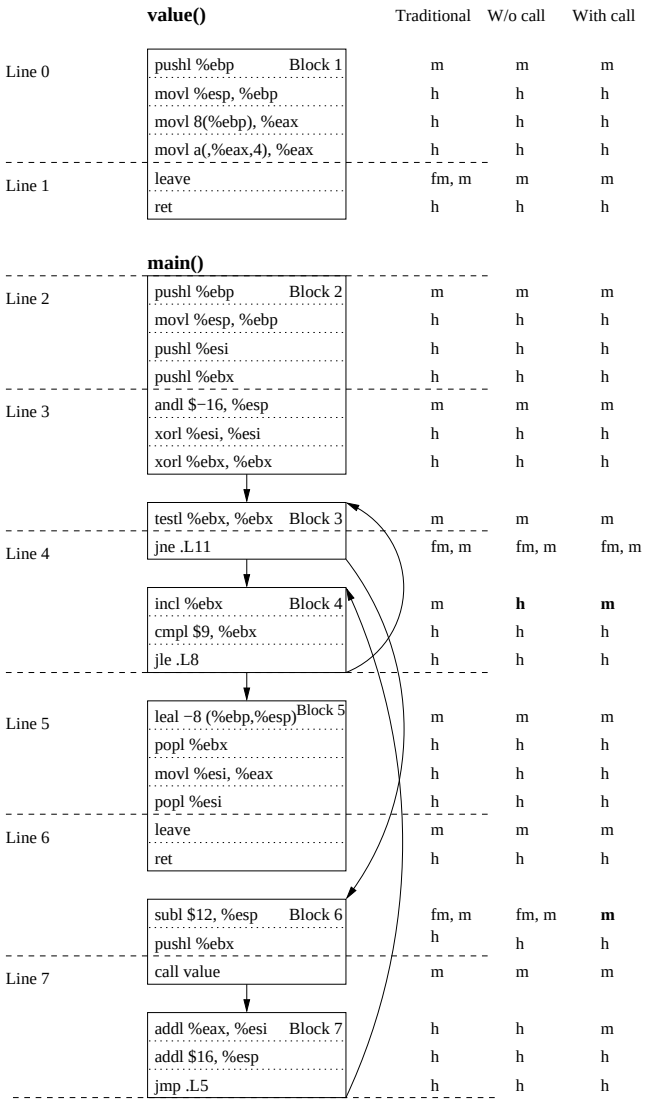

Figure 6: Categorizations by Traditional vs. Module-level Analysis for Always-hits (h), Always-misses (m), First-hits (fh) and First-misses (fm).

The ACS calculations in the traditional approach are made using the inter-procedural data-flow analysis. In contrast, module-level analysis is constrained to only the control-flow graph of a module at a time before compositional analysis combines module-level results. For the `main` function, the ACS calculations are performed in two stages, 1) ignoring calls and 2) considering calls such that the ACS is invalidated on return.

The differences between the categorizations obtained at different stages from the two approaches are depicted in Figure 6. The first column of categorizations is based on calculations in the traditional approach. Our ultimate aim is to achieve the exact same categorizations using the new method. The second column shows the categorizations calculated in the first step of the compositional analysis method, assuming there are no calls in the `main` module. A list of categorizations **a,b** denotes the cache behavior of the inner nest and the outer nest, respectively, where a nesting represents a loop (multiple iterations) or a function instance (one iteration).

There are some differences between the categorizations of the first column and the second column since the call to `value` was

ignored. The possible differences in calculations due to the call to `value` are taken into account in the second step to obtain the categorizations shown in the third column. Comparing column three with column one, we notice that all the categorizations are the *same* or, if different, have the same effect. The differences in the calculations are as follows:

1. The first instruction in basic block 4 was initially categorized as a `hit` by the compositional method. The `in` and `out` sets are shown below.

```
Traditional analysis -
in(4) = [4 I 2 3          out(4) = [4 I 2 3
         0 1 6 7]                   1 6 7]
Compositional analysis -
   a) Ignoring calls -
in(4) = [4 I 2 3          out(4) = [4 I 2 3
           6 7]                     6 7]
   b) Considering calls -
in(4) = [4 I 2 3          out(4) = [4 I 2 3
         C C C 7]                   C 7]
```

   In the traditional approach, the `in` for basic block 4 has a conflict between line 4 (which contains basic block 4) and line 0 (which is in module `value`). Thus, it is categorized as a miss. The same `in` set is shown for the compositional analysis calculations. When ignoring calls, the conflict for line 4 remains unrecognized since the cache effect of the call to `value` is not considered. The categorization is adjusted by the algorithm in Fig. 5 since the conflict between 0 and 4 is recognized resulting in an always-miss, just as in the traditional approach.

2. The first instruction in basic block 6 and the fifth instruction in basic block 1 show different categorizations for the two types of cache analysis. But the categorization produce the same effect during timing analysis since a first-miss for one iteration (for function `value` without a loop) is equivalent to an always-miss. (This is an artifact of instruction categorizations [21]).

5. EXPERIMENTAL FRAMEWORK

This section describes the experiments performed to measure the performance improvement of the new compositional approach over the traditional approach and the accuracy of categorizations predicted by the new approach. Five benchmarks from the C-Lab real-time benchmark suite [6] (`adpcm`, `ndes`, `cnt`, `mm` and `fft`) and one benchmark (`multimedia audio decoder - mad`) from the MiBench [8] embedded benchmark suite were used.

Timing Analysis: We modified the static timing analysis tool from [10], to make it compatible with an architectural simulator based on the SimpleScalar toolset [1]. The static timing anlaysis tool takes program flow information from a GCC with a Portable Instruction Set Architecture (PISA) backend, which also interfaces with the architectural simulator. Both the architectural simulator and the timing analysis tool use a pipeline model with six stages (fetch, decode, issue, execute, memory and write back) featuring static branch prediction (Ball-Larus heuristic [3]). Both tools take into account a 8KB I-cache and a constant memory latency of 100ns.

To calculate the Worst-Case Execution Cycles (WCEC), the Real-time benchmark is compiled using the PISA-GCC compiler and the control-flow information obtained is supplied to the static instruction cache simulator (which may use the traditional integrated approach or the compositional approach) to obtain the caching categorizations. These categorizations obtained for the I-cache are used in the timing analyzer to determine accurate WCET bounds.

Architectural Simulator: The architectural simulator is based on the SimpleScalar toolset [1]. It supports a simple six-stage in-order processor pipeline model. All the Real-time benchmarks are compiled with the SimpleScalar GCC-based compiler for the SimpleScalar ISA (PISA), a MIPS-like ISA [5]. The Architectural Simulator loads actual binaries and executes them on the pipeline model. The actual or observed execution times for the benchmarks are obtained from the architectural simulator.

Experiment 1: Performance improvement: Static cache analysis is performed for the six benchmarks using the traditional integrated approach and the compositional approach for various cache configurations. Different cache configurations are formed by changing the number of cache lines (varying from 4 to 1024 in powers of 2) and the size of each cache line (from 8 bytes to 64 bytes). Execution times of the two analysis procedures are recorded and compared.

Experiment 2: Comparison of WCEC: To determine the accuracy of predictions, the WCEC was measured for all the benchmarks using the timing analyzer discussed earlier. Data on WCEC were collected for categorizations predicted by the traditional approach as well as by the new approach. Subsequently, the WCEC obtained using the two approaches were compared. To obtain the control-flow analysis information, the benchmarks were compiled using `gcc` for PISA.

Experiment 3: Comparison of WCEC with architectural simulator: The WCECs for the six benchmarks were obtained using the timing analyzer for the various cache configurations (as described in the previous sub-section). The benchmarks were also executed on the architectural simulator to obtain the execution times. These numbers were compared to the WCEC obtained from the timing analysis tool. The benchmarks were executed for worst-case input data on the architectural simulator (when available).

6. EXPERIMENTAL RESULTS

In this section, the results for each of the three experiments are described. First, we report timing results for the analysis overhead. Figure 7 shows times required by the traditional approach and the compositional approach for a cache line size of 16 bytes. The graph is shown for three of the six benchmarks, because the patterns in the results obtained for these three benchmarks are representative of the complete set.

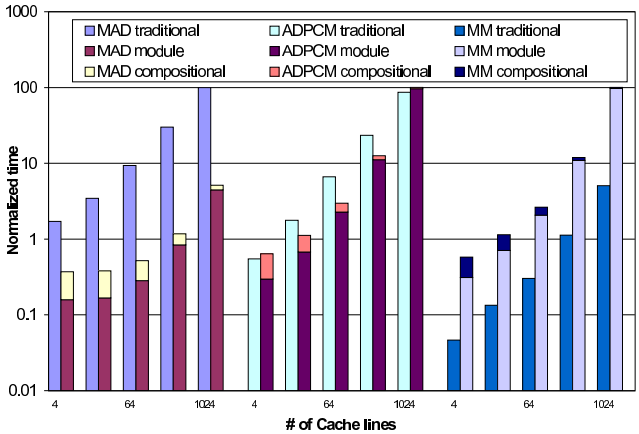

Figure 7: Overhead Traditional vs. Compositional

The first set of ten bars describe the computation times for the mad benchmark for 4, 16, 64, 256, 1024 cache lines respectively. The second set of ten bars represents the adpcm benchmark and the last set represents mm. The odd-numbered bars (single bars) show the time taken by the traditional cache analysis approach. The even-numbered bars (stacked bars) show the total time taken by the compositional analysis approach, taking both the modular and the compositional stages into account. These bars are stacked, separately showing the time taken for the module analysis (lower portion of bar) and the time taken for compositional analysis (upper portion of bar). Note that if module analysis had been performed in advance, the only computation times that should be compared are the traditional and the compositional stage times. Also note that the y-axis shows a logarithmic scale.

mad is a relatively large benchmark with around 9000 instructions. It consists of around 74 functions and the program structure involves complex loop and control-flow constructs. For mad (Figure 7), a significant savings in computation time is observed for all different cache sizes. For a cache with 1024 lines, the new approach takes a total of around 350 seconds compared to the 6600 seconds of the traditional approach, resulting in savings of an order of a magnitude. If only the compositional stage time is compared to the traditional time, as expected, a significant improvement in the compositional approach is observed. For instance, the traditional approach takes about 6600 seconds for 1024 lines compared to only about 45 seconds for the new approach, a savings of two orders of magnitude. Notice that a 16 bytes cache line size requires more time than a 32 byte line size.

Figure 7 shows that for adpcm the total time for the two-stage approach is smaller than that for the traditional approach for cache lines ranging from 8 to 512. However, for a cache with 1024 lines, the time for the modular stage exceeds the traditional time causing the total time to be larger for the new approach. This is easily explained since the modular stage performs 4 different analyses on each module, hence, this overhead increases with larger cache sizes. Absolute analysis times are .1 to 26 seconds.

Let us compare traditional computation time overhead with just the compositional stage overhead (upper portion of even bars). For adpcm, the compositional approach takes less time than the traditional approach for all the cache configurations. Additional results for adpcm also indicate that for larger programs the compositional approach scales slightly better than the traditional approach (see [23] for details). For instance, the increase in traditional time for a change from 256 to 512 lines is about twice, whereas the increase for the compositional time is only about 1.5 times.

For mm (Figure 7), the total time for the new approach is greater than the traditional time for all cache configurations. mm is a small benchmark (around 130 instructions). Its calling structure and the control structure are fairly simple. Hence, it does not provide enough opportunity for the compositional approach to exploit any complexity in the program structure. Even when comparing just traditional times to the compositional times for fewer cache lines (less than 128), the traditional approach outperforms the new one. Absolute analysis times are .001 to 3.4 seconds.

In summary, results for larger benchmarks (adpcm and mad) show that the compositional approach, even when applied just once for all the modules in a program, performs much better than the traditional approach. This is significant because it indicates that for such large programs, the new approach can result in savings in excess of an order of magnitude in computation time.

Next, we compare the resulting WCEC of the two analysis methods. For all benchmarks (except for adpcm) and for all cache configurations ranging from 4 cache lines to 512 cache lines, the WCEC were exactly the same. For adpcm and ndes, the results of timing analysis are summarized in Tables 2 and 3, respectively.

Number of cache lines	Traditional WCEC	Compositional WCEC	% change
512	9,227,240	9,227,240	0
256	9,227,240	9,227,240	0
128	18,590,441	18,590,441	0
64	23,751,141	23,651,041	-0.42

Table 2: Timing Analysis of Adpcm

For adpcm, results for a cache size of 64 lines show a tighter WCEC prediction by the new approach. This is the only difference observed between WCEC for the two approaches. In this case, the traditional approach categorizes one instruction of adpcm pessimistically as an always-miss in a loop. The new approach predicts categories of this instruction more accurately as a first-miss. This is due to the analysis scope where data-flow is retained both considering and ignoring backedges. Comparing the data-flow, one can detect that a conflicting line is absent along the backedge, which indicates a first miss. The traditional analysis does take advantage of such information since it would be computationally intensive, albeit possible, to calculate it. The compositional approach leads to a tighter and still safe WCEC bound using the compositional approach. It is also interesting to note that as adpcm occupies around 230 program lines in memory, it entirely fits in caches with 256 and 512 lines. *I.e.*, once a program line is brought in cache, it will not be evicted later. Hence, the WCEC for these two cache sizes are the same for this benchmark.

For ndes, as can be seen from the table, the estimated WCECs are exactly the same for the two approaches.

Finally, we compare the WCEC with cycles reported by SimpleScalar for a simulated execution under worst-case input. As shown in the above section, WCECs for the six benchmarks were obtained using the timing analyzer for the various cache configurations. These numbers were compared to the number of cycles obtained from the architectural simulator for the SimpleScalar toolset. The benchmarks were executed for worst-case input data on the simulator.

Table 4 shows the WCEC comparison for a direct-mapped cache with 64 lines and a line size of 16 bytes. For other cache configurations, similar results are observed. As already illustrated by the comparison between traditional vs. compositional cache analysis, by maintaining the same level of accuracy, differences between the predicted WCECs of our toolset and the observed WCECs under SimpleScalar, such as for adpcm and ndes, stem from data dependencies. Our path analysis operates at static time and considers combinations of paths that, due to infeasible data value combinations, cannot be executed in sequence. These infeasibilities are only known at runtime and cannot be captured by static analysis (unless manually specified, which is tedious).

The adpcm benchmark has several instructions within conditional execution. Some of these constructs contain loops or are embedded in one. The timing analysis will choose the longest path that

Number of cache lines	Traditional WCEC	Compositional WCEC	% change
512	131,514	131,514	0
64	146,583	146,583	0
16	347,523	347,523	0
8	839,551	839,551	0

Table 3: Timing Analysis of Ndes

is found for every iteration of the loops. In the architectural simulator, the longest path need not be the path that is always taken inside the loops resulting in a difference between the WCEC and observed number of cycles (due to data dependencies). Since the WCEC always uses the longest path, it is a safe bound of the execution time. Similarly, the `ndes` benchmark has several instructions within conditional execution. These instructions may not be executed in the architectural simulator but are always considered in timing analysis since they contribute to the longest path. The WCEC of `mad` could not be obtained due to constraints of path analysis within the timing analysis tool, a problem beyond compositional static cache simulation that we are currently addressing.

Benchmark	Predicted WCEC	Observed WCEC	Ratio
`adpcm`	23,740,141	17,549,967	1.35
`ndes`	146,298	95,147	1.54
`fft`	381,091	369,671	1.03
`cnt`	72,240	71,411	1.01
`mm`	2,037,588	2,034,313	1.01

Table 4: Predicted vs. Observed WCEC

In summary, the validity of the new approach is dependent on two metrics:

- the improvement in the computation time required to perform static cache analysis and
- the accuracy of the predicted categorizations.

The compositional approach certainly leads to a significant improvement in execution time of static cache analysis. It also provides worst-case categorization predictions at least as accurate as those provided by the traditional approach. In some cases, it provides more accurate predictions than the traditional approach that are still safe.

7. RELATED WORK

The purpose of static cache simulation is to bound the WCET of a program. Bounding WCET estimations is important as schedulability analysis for hard real-time systems requires that WCETs be known in order to ensure feasibility of scheduling a task set for a given scheduling policy, such as rate-monotone and earliest-deadline-first scheduling [16].

Many different approaches have been used to obtain the WCET for a program. The combined use of the static cache analysis (Arnold *et al.* [2]) and the static timing analysis (Healy *et al.* [11]) provide fairly accurate WCET bounds. Our approach splits the static cache analysis into two stages, reusing the results from the first (module-level analysis) stage in the second (compositional analysis) stage, thus addressing the issue of computation overhead of cache analysis.

Harmon *et al.* [9] describe a retargetable framework for predicting execution time that transforms machine-level instructions into a sequence of primitive operations, which express the functionality of each instruction in fine-grain detail. This framework, along with Puschner *et al.* [24] and Park [22], proposes ideas for timing analysis of unoptimized programs on simple CISC processors. Zhang *et al.* [31] introduce an analyzable pipeline model considering the effect of pipelined execution on timing analysis. The combined framework of [2] and [11] described above takes instruction cache effects into account. These ideas and the WCET analysis framework in [15] introduce timing analysis for pipelined RISC processors. Li *et al.* [13] use an Integer Linear Programming (ILP) formulation for capturing structural functionality and cache constraints and solve it to bound the WCET. The ILP approach is extended to take cache timing effects into account. Rawat [25], Kim *et al.* [12], Li *et al.* [14] and White *et al.* [30] extend the conventional static timing analysis to data caches.

Some instruction-level dynamic timing analysis methods assume the knowledge of an input set that would trigger the worst-case path during the actual program execution [28]. Finding such an input set can be a non-trivial problem involving very high computational complexity. Even if such an input set is found, the actual execution-time effects of architectural features onto timing analysis might cause a different input set to determine the WCET.

The integrated path and timing analysis method in Lundqvist *et al.* [19] combines dynamic cache analysis and dynamic timing analysis using an instruction-level cycle-accurate simulation technique. The cache state at various points in the timing analysis is calculated on the basis of the actual memory references made. The worst-case state is obtained by analyzing the cache state that will trigger worst-case response time in the remaining execution sequence. Thus, the cache calculations are integrated in the instruction-level simulation. Such approaches would suffer a significant computational overhead involved in cache calculations for fairly large programs that consist of a large number of possible execution paths.

Vera *et al.* [27] describe a technique of data cache locking to achieve predictable worst-case execution time for a program even when the execution environment has data caches. Unpredictability in WCET estimations is eliminated by locking those regions in the code that cannot be analyzed statically. Performance degradation possible due to cache locking is reduced by loading the cache with data likely to be accessed, but some overhead prevails.

From this review of prior work, it is evident that all these ideas deal with the problem of accurately estimating WCET for a program in the presence of architectural features, such as data and instruction caches and instruction pipelines. Our work is different from these as it focuses on the computational overhead involved in static cache analysis (performed separately from timing analysis), while preserving the accuracy of the predicted categorizations.

Rubin *et al.* [26] discusses a profile-analysis framework for data-layout optimizations. It uses the concept of deriving a *data objects trace* to save the re-execution of programs to assess the performance of a newly computed data layout. Each entry in the *data objects trace* contains a symbolic address as well as a unique identifier of the referenced data object. To evaluate a candidate data layout, each data object is assigned its new memory location obtained from the layout. Then, the original data trace is simulated using the *data objects trace* and the new memory addresses to measure the new memory behavior. Thus, the *data objects trace* parameterizes the actual memory address trace, making its reuse possible. Since the *data objects trace* is address-independent, it can be reused in the new memory layout evaluation. In this way, the idea of address parameterization is utilized here, but it addresses the orthogonal problem of optimizing data layout. Our work also parameterizes the data-flow information obtained in the first stage (module-level analysis) as the absolute address information is not available during this stage, and the inter-procedural analysis on modules belonging to a program can be performed irrespective of the actual absolute addresses the modules are mapped to.

8. CONCLUSION

This paper contributes an approach to reduce the computational overhead involved in obtaining cache hit/miss information from static cache simulation for direct-mapped instruction caches. Cat-

egorizations obtained in module-level analysis are reused in compositional analysis, which performs the necessary inter-procedural analysis and derives the final categorizations. Thus, for any concrete cache configuration and for any module, the module-level analysis is performed only once. The compositional stage reuses such analysis for any program containing pre-analyzed modules.

The experimental results show that the compositional approach outperforms the traditional approach by one to two orders of magnitude for large cache sizes and larger benchmarks. For smaller cache sizes and smaller programs, the computational overhead is comparable to the traditional approach. The compositional approach also gives predictions as accurate as the traditional static cache simulation approach. This was evident from the worst-case execution cycles obtained for the benchmarks for the two approaches. This work provides an efficient way for analyzing direct-mapped instruction caches statically, especially for large cache sizes and for programs with complex loop structures as well as complex calling structures.

9. REFERENCES

[1] A. Anantaraman, K. Seth, K. Patil, E. Rotenberg, and F. Mueller. Virtual simple architecture (VISA): Exceeding the complexity limit in safe real-time systems. In *International Symposium on Computer Architecture*, pages 250–261, June 2003.

[2] R. Arnold, F. Mueller, D. B. Whalley, and M. Harmon. Bounding worst-case instruction cache performance. In *IEEE Real-Time Systems Symposium*, pages 172–181, Dec. 1994.

[3] T. Ball and J. R. Larus. Branch prediction for free. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 300–313, June 1993.

[4] G. Bernat, A. Colin, and S. Petters. Wcet analysis of probabilistic hard real-time systems. In *IEEE Real-Time Systems Symposium*, Dec. 2002.

[5] D. Burger, T. M. Austin, and S. Bennett. Evaluating future microprocessors: The simplescalar tool set. Technical Report CS-TR-1996-1308, University of Wisconsin, Madison, July 1996.

[6] C-Lab. Wcet benchmarks. Available from http://www.c-lab.de/home/en/download.html.

[7] C. Ferdinand and R. Wilhelm. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Systems*, 17(2/3):131–181, Nov. 1999.

[8] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Workshop on Workload Characterization*, pages 3–14, Dec. 2001. http://www.eecs.umich.edu/mibench.

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

[10] C. A. Healy, R. D. Arnold, F. Mueller, D. Whalley, and M. G. Harmon. Bounding pipeline and instruction cache performance. *IEEE Transactions on Computers*, 48(1):53–70, Jan. 1999.

[11] C. A. Healy, D. B. Whalley, and M. G. Harmon. Integrating the timing analysis of pipelining and instruction caching. In *IEEE Real-Time Systems Symposium*, pages 288–297, Dec. 1995.

[12] S. Kim, S. Min, and R. Ha. Efficient worst case timing analysis of data caching. In *IEEE Real-Time Embedded Technology and Applications Symposium*, June 1996.

[13] Y.-T. S. Li, S. Malik, and A. Wolfe. Efficient microarchitecture modeling and path analysis for real-time software. In *IEEE Real-Time Systems Symposium*, pages 298–397, Dec. 1995.

[14] Y.-T. S. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software: Beyond direct mapped instruction caches. In *IEEE Real-Time Systems Symposium*, pages 254–263, Dec. 1996.

[15] S.-S. Lim, Y. H. Bae, G. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, and C. S. Kim. An accurate worst case timing analysis for RISC processors. In *IEEE Real-Time Systems Symposium*, pages 97–108, Dec. 1994.

[16] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. of the Association for Computing Machinery*, 20(1):46–61, Jan. 1973.

[17] J. Liu. *Real-Time Systems*. Prentice Hall, 2000.

[18] J.-C. Liu and H.-J. Lee. Deterministic upperbounds on the worst-case execution time of cached programs. In *IEEE Real-Time Systems Symposium*, Dec. 1994.

[19] T. Lundqvist and P. Stenström. An integrated path and timing analysis method based on cycle-level symbolic execution. *Real-Time Systems*, 17(2/3):183–208, Nov. 1999.

[20] F. Mueller. *Static Cache Simulation and its Applications*. PhD thesis, Dept. of CS, Florida State University, July 1994.

[21] F. Mueller. Timing analysis for instruction caches. *Real-Time Systems*, 18(2/3):209–239, May 2000.

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

[23] K. Patil. Compositional static cache analysis using module-level abstraction. Master's thesis, Dept. of CS, North Carolina State University, Aug. 2003.

[24] P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. *Real-Time Systems*, 1(2):159–176, Sept. 1989.

[25] J. Rawat. Static analysis of cache analysis for real-time programming. Master's thesis, Iowa State University, 1995.

[26] S. Rubin, R. Bodik, and T. Chilimbi. An efficient profile-analysis framework for data-layout optimizations. In *Annual Symposium on Principles of Programming Languages*, pages 140–153, 2002.

[27] X. Vera, B. Lisper, and J. Xue. Data caches in multitasking hard real-time systems. In *IEEE Real-Time Systems Symposium*, Dec. 2003.

[28] J. Wegener and F. Mueller. A comparison of static analysis and evolutionary testing for the verification of timing constraints. *Real-Time Systems*, 21(3):241–268, Nov. 2001.

[29] R. White, F. Mueller, C. Healy, D. Whalley, and M. Harmon. Timing analysis for data caches and set-associative caches. In *IEEE Real-Time Embedded Technology and Applications Symposium*, pages 192–202, June 1997.

[30] R. T. White, F. Mueller, C. Healy, D. Whalley, and M. G. Harmon. Timing analysis for data and wrap-around fill caches. *Real-Time Systems*, 17(2/3):209–233, Nov. 1999.

[31] N. Zhang, A. Burns, and M. Nicholson. Pipelined processors and worst case execution times. *Real-Time Systems*, 5(4):319–343, Oct. 1993.