Accurate Timing Analysis by Modeling Caches, Speculation and their Interaction

Xianfeng Li
lixianfe@comp.nus.edu.sg

Tulika Mitra
tulika@comp.nus.edu.sg

Abhik Roychoudhury
abhik@comp.nus.edu.sg

School of Computing
National University of Singapore
Republic of Singapore 117543

ABSTRACT

Schedulability analysis of real-time embedded systems requires worst case timing guarantees of embedded software performance. This involves not only language level program analysis, but also modeling the effects of complex micro-architectural features in modern processors. Speculative execution and caching are very common in current processors. Hence one needs to model the effects of these features on the Worst Case Execution Time (WCET) of a program. Even though the individual effects of these features have been studied recently, their combined effects have not been investigated. We do so in this paper. This is a non-trivial task because speculative execution can indirectly affect cache performance (e.g., speculatively executed blocks can cause additional cache misses). Our technique starts from the control flow graph of the embedded program, and uses integer linear programming to estimate the program's WCET. The accuracy of our modeling is illustrated by tight estimates obtained on realistic benchmarks.

Categories and Subject Descriptors

C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems—*Real-time and Embedded Systems*

General Terms

Measurement, Performance.

Keywords

Worst Case Execution Time, Cache, Branch Prediction.

1. INTRODUCTION

Static timing analysis of software is important for real-time embedded systems. A worst case timing guarantee is part of functional requirements for these systems. Given a software and a micro-architecture, *Worst Case Execution Time (WCET)* analysis finds the maximum execution time of the software for *any possible input*. Static analysis is required because for most software and micro-architectures, it is impossible to guess the input that will generate the worst case execution time.

WCET analysis consists of two steps: (a) path analysis that eliminates infeasible paths and derives upper bounds on loop executions, and (b) micro-architectural modeling of pipeline, caches, branch prediction, etc. to estimate the execution time of feasible paths. A useful approach to WCET analysis is Integer Linear Programming (ILP) formulation that combines the two steps in a single framework [12].

Current generation processors use aggressive speculation to improve performance. In particular, they commonly employ control speculation which is also known as branch prediction. Conditional branch instructions cause pipeline stalls as the processor does not know which way to go till the branch is resolved. Branch prediction solves this problem by fetching and executing instructions along the predicted path. If the prediction is correct, then the processor never stalls. If the prediction is incorrect, then the instructions from the predicted path are flushed resulting in *branch misprediction penalty*. Modeling speculative execution is important for timing analysis as discussed in [2, 3, 14].

Apart from penalty due to instruction flushing, misprediction can indirectly affect instruction cache performance. As the processor caches instructions along the mispredicted path, the instruction cache content changes. This *prefetching* of instructions can have both constructive and destructive effect on the cache performance and hence on WCET. It is also known as *wrong path prefetching* in the computer architecture community [15]. Effect of wrong-path prefetching on cache performance has been quantitatively evaluated in [4, 15]. However, there have been no studies to combine speculative execution and caching for WCET analysis, leave alone the indirect effects such as wrong path prefetching. In fact, it is believed that these indirect effects are hard to capture for WCET analysis [16].

In this paper, we develop an ILP based technique to model the combined effects of speculation, caching and wrong-path instruction prefetching on embedded code performance. We first combine existing micro-architectural modeling of instruction cache and branch prediction in a unified framework. Furthermore, we accurately estimate the constructive

DAC 2003, June 2–6, 2003, Anaheim, California, USA.

and destructive effects of instruction prefetching by branch prediction. Our experimental results demonstrate the accuracy of our WCET estimates.

2. RELATED WORK

Analyzing the WCET of a program has been extensively investigated. Earlier works estimated the WCET of a program by finding the longest execution path via program analysis. The cost of a path is the sum of the costs of the different instructions, assuming that the execution time of each instruction is constant. Presence of performance enhancing micro-architectural features such as pipeline, cache and branch prediction make this cost model inapplicable. Recent works on WCET analysis have therefore modeled micro-architectural features such as pipelined processors [6, 8], superscalar processors [13], cache [7, 12, 18], and branch prediction [2, 3, 14].

Integer Linear Programming (ILP) formulation of WCET analysis problem is a popular technique. In particular, [12] has reduced the WCET analysis of instruction cache behavior into an ILP problem and [14] has modeled various branch prediction schemes using ILP. In [17], ILP has been used for program path analysis subsequent to abstract interpretation based micro-architectural modeling of cache, pipelines etc.

The wrong-path cache effect involves prefetching instructions due to misprediction. Note that instruction prefetching has been previously modeled for WCET analysis in [2, 11]. However, wrong-path prefetch modeling is much more involved as the prefetching is controlled by branch mispredictions (which by itself is difficult to model).

3. TIMING ANALYSIS BACKGROUND

Our formulation is based on Integer Linear Programming (ILP). The WCET is obtained by maximizing a time function subject to linear constraints. The time function will take into account the effects of branch prediction as well as instruction caching.

Path Analysis. The starting point of the ILP analysis is the **Control Flow Graph (CFG)** of the program. The vertices of the control flow graph are basic blocks (a consecutive sequences of instructions with a single entry and exit point) and the edges denote the flow of control from one basic block to another. A separate copy of the CFG of a function f (recursive or otherwise) is created for every distinct call site of f in the program such that each call transfers control to its corresponding copy of CFG.

Let v_i be the execution count of a basic block B_i and $cost_i$ be a constant representing the execution time of B_i assuming zero cache miss and perfect branch prediction. The total execution time of the program is $Time = \sum_{i=1}^{N} cost_i \times v_i$ where N is the number of basic blocks in the program. Let $e_{i\to j}$ be the execution count of the edge $i \to j$ between basic blocks B_i and B_j in CFG. Since inflow equals outflow for each basic block, we have $v_i = \sum_j e_{j\to i} = \sum_j e_{i\to j}$. We provide upper bounds on loops and recursion depths either from user input or via offline data flow analysis.

Instruction Cache. In order to model the instruction cache effect, we use **line-block** or **l-block** as defined by Li, Malik and Wolfe in [12]. An l-block is a sequence of instructions in a basic block that belong to the same instruction cache line. A basic block B_i is partitioned into n_i l-blocks denoted as $B_{i.1}, B_{i.2}, \ldots, B_{i.n_i}$. Let $cm_{i.j}$ be the total cache misses for l-block $B_{i.j}$ and cmp be the constant denoting the cache miss penalty. Then the total execution time becomes

$$Time = \sum_{i=1}^{N} (cost_i \times v_i + \sum_{j=1}^{n_i} cmp \times cm_{i.j}) \quad (1)$$

For simplicity of exposition, let us assume a direct mapped cache. For each cache line c, we construct a **Cache Conflict Graph (CCG)** G_c [12]. The nodes of G_c are the l-blocks mapped to c. An edge $m \to m'$ exists in G_c iff there exists a path in the CFG s.t. control flows from m to m' without going through any other l-block mapped to c. In other words, there is an edge between l-blocks m to m' if m can be present in the cache when control reaches m' resulting in a cache miss for m'.

Let $e_{i.j\to u.v}$ be the execution count of the edge between l-blocks $B_{i.j}$ and $B_{u.v}$ in a CCG. Now execution count of l-block $B_{i.j}$ is equal to the execution count of basic block B_i. Also, at each node of the CCG, inflow is equal to outflow and both are equal to the execution count of the node. Therefore,

$$v_i = \sum_{u.v} e_{i.j\to u.v} = \sum_{u.v} e_{u.v\to i.j} \quad (2)$$

Finally, cache miss count at an l-block is equal to the inflow from other l-blocks in the CCG. Thus,

$$cm_{i.j} = \sum_{\substack{u.v \\ u.v\neq i.j}} e_{u.v\to i.j} \quad (3)$$

Speculative Execution. Most modern processors perform control speculation via dynamic branch prediction. Dynamic prediction schemes use a 2^n entry branch prediction table to store past branch outcomes. When the processor encounters a conditional branch instruction, this prediction table is looked up using some index and the indexed entry is used as prediction. When the branch is resolved, the entry is updated with the actual outcome. Different branch prediction schemes differ in how they compute n-bit index to access this table. In case of simplest prediction scheme, the index is n lower order bits of the branch address [10]. More accurate are the *global branch prediction* schemes [19] where the index also uses the outcome of the neighboring branches to exploit the correlation among consecutive branch outcomes.

Let bm_i be the number of mispredictions of the branch (if one exists) in B_i and bmp is a constant denoting the penalty for a single branch misprediction. Then the total execution time can be modified as

$$Time = \sum_{i=1}^{N} (cost_i \times v_i + bmp \times bm_i + \sum_{j=1}^{n_i} cmp \times cm_{i.j}) \quad (4)$$

In earlier work [14] we have modeled different branch prediction schemes for WCET analysis in ILP framework. Here, we use that modeling to derive constraints on bm_i.

4. SPECULATION + CACHING

The WCET analysis as described in the previous section does not take into account the effect of branch misprediction on instruction cache performance. When a branch is predicted, instructions are fetched and executed from the

predicted path. If all the branches were predicted correctly, then the analysis described in previous section will give accurate results. Now consider a branch that is mispredicted. The processor will fetch and execute instructions along the mispredicted path till the branch is resolved. There can be two scenarios during mispredicted path execution (1) there is no cache miss, and (2) there are one or more cache misses. In the first scenario, the misprediction has no effect on the instruction cache. However, in the second scenario, the instruction cache content has been modified when the processor resumes execution from the correct path. Various studies concluded that depending on the application, this wrong-path prefetching can have constructive or destructive effect on the instruction cache performance [4, 15]. Our goal here is to model this wrong-path cache effect for WCET analysis.

Assumptions. We make two standard assumptions. First, the processor allows only one unresolved branch at any point of time in execution. Thus, speculative execution occurs only when all previous branches have been resolved. We also assume that the instruction cache is blocking (i.e., it can support only one pending cache miss). This is indeed the case in almost all commercial processors.

Wrong Path Prefetching. From now, we will use the shorthand $m \cong m'$ to denote that l-blocks m, m' map to the same cache line. Also, we will use $[m]$ to denote the cache line to which l-block m maps to. Thus $m \cong m'$ iff $[m] = [m']$. We want to capture the following: (1) *destructive effect*: additional cache misses suffered due to speculative execution and (2) *constructive effect*: cache misses avoided due to speculative execution. The cache misses/delays *introduced* due to speculative execution belong to two categories.

Category 1 An l-block m misses during normal execution, since $m' \cong m$ was fetched during speculative execution.

Category 2 An l-block m misses during speculative execution. All or part of this cache miss delay can be masked by the branch misprediction penalty. We call this additional delay due to miss at m as **mp_delay**.

Wrong path prefetching can also *avoid* cache misses as follows. Suppose $m \cong m'$ and there is an edge between m and m' in the original CCG of $[m]$. The edge implies that there are one or more paths from m to m' that do not go through any node in $[m]$. Suppose, all these paths go through a branch b whose mispredicted path contains m'. Now, for any misprediction of b, m' can be fetched along the wrong path and hence m' will not incur cache miss in the correct path. Both the constructive and destructive effects of wrong path prefetching are modeled by changing the Cache Conflict Graph (CCG) as discussed below.

4.1 Changes to Cache Conflict Graph

A brief discussion of Cache Conflict Graph was given in the last section. A more detailed discussion appears in [12]. We add some nodes and edges to the CCG to capture the constructive and destructive effects on instruction cache performance due to branch misprediction.

Additional nodes in CCG. Given a branch b, actual outcome X (non-taken or taken, denoted as $0, 1$) and misprediction penalty bmp, we can identify a totally ordered *sequence* of l-blocks which will be accessed along the mispredicted path. This is denoted as $\mathbf{Spec(b, X)}$. Clearly, the cost of executing the blocks in $Spec(b, X)$ cannot exceed bmp (which is why we need to consider the misprediction penalty while defining $Spec(b, X)$). Also, note that every speculative execution of branch b (where the actual outcome is X and the misprediction is along outcome $\neg X$) may not execute all the l-blocks in $Spec(b, X)$. This is because an l-block along the mispredicted path may incur cache miss which will reduce the total number of l-blocks that can be fetched in branch misprediction penalty duration.

We now add the following set of nodes to the CCG. For convenience, we will call these nodes as **mp-nodes** (nodes to model effect of branch *mis*prediction).

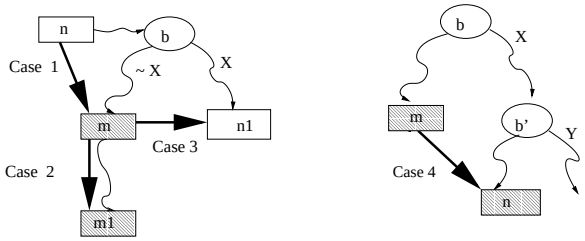
$$\{m^{b,X} \mid b \in Branches(P), X \in \{0,1\}, m \in Spec(b, X)\}$$

$Branches(P)$ denotes the set of branches in the control flow graph of the program P we are considering. In an abuse of notation, we have written $m \in Spec(b, X)$ to denote membership even though $Spec(b, X)$ is a sequence.

Additional edges in CCG. For any mp-node $m^{b,X}$, the following edges are added to the CCG of cache line $[m]$

1. $n \rightarrow m^{b,X}$ if (a) there exists a path in CFG from n to the l-block containing branch b that does not pass through any l-block in $[m]$ (b) m is the first use of cache line $[m]$ in $Spec(b, X)$ and (c) $n \cong m$.
2. $m^{b,X} \rightarrow m_1^{b,X}$ where m_1 is the first l-block appearing after m in $Spec(b, X)$ s.t. $m \cong m_1$.
3. $m^{b,X} \rightarrow n1$ where $n1$ ($n1 \cong m$) is a possible first use of cache line $[m]$ if outcome X occurs at branch b.
4. $m^{b,X} \rightarrow n^{b',Y}$ where n ($n \cong m$, $n \in Spec(b', Y)$) is a possible first use of cache line $[m]$ if outcome X occurs at branch b (b' can be same as b).

The figure below illustrates these cases. The shaded ones are the mp-nodes and the unshaded ones are the normal nodes.



The third and the fourth type of edges require some explanation. If there are multiple l-blocks along the speculative path that map to a particular cache block, then we conservatively add outgoing edges from all of them to the first use of the cache block in the correct path (or another speculative path). This is because any one of these l-blocks can be in the cache when the branch is resolved.

Figure 1 illustrates the modifications to the CCG with an example. The control flow graph is shown in Figure 1(a). Let us assume that l-blocks $B_{0.1}$, $B_{1.2}$ and $B_{3.1}$ belong to the same cache block. Then the original CCG for that cache block is shown in Figure 1(b). A dummy start node and an end node are added to each CCG to make the initial and terminal flow equations correct.

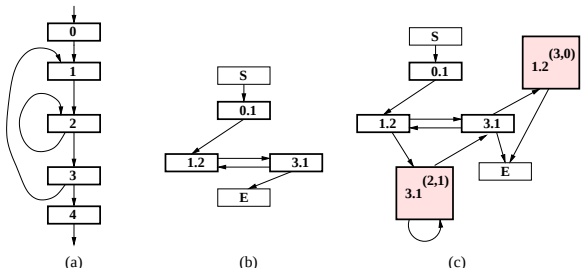


Figure 1: Changes to Cache Conflict Graph (Shaded nodes are mp-nodes)

The modifications to the CCG due to wrong-path prefetching is shown in Figure 1(c). We add two mp-nodes $B_{3.1}^{2,1}$ and $B_{1.2}^{3,0}$ corresponding to the mispredictions at node B_2 and node B_3 respectively. Note that we do not add any node corresponding to 0 outcome at branch B_2 and 1 outcome at branch B_3 respectively. This is because corresponding to 0 outcome at branch B_2, the mispredicted path fetches basic block B_2 which does not contain any l-block that maps to the cache line and similarly for B_3 with outcome 1. Among the additional edges, $B_{1.2} \rightarrow B_{3.1}^{2,1}$ and $B_{3.1} \rightarrow B_{1.2}^{3,0}$ belong to the first type. The edges $B_{3.1}^{2,1} \rightarrow B_{3.1}$ and $B_{3.1}^{2,1} \rightarrow B_{3.1}^{2,1}$ belong to the third and fourth type respectively.

4.2 Objective function

We want to model both constructive/destructive effects: cache misses avoided/introduced due to speculative execution. The constructive effect is modeled by modifying the CCG. Let us illustrate with the example in Figure 1(b). In the original CCG, there is a direct edge $B_{1.2} \rightarrow B_{3.1}$ and that is the only path between the two nodes; every time control reaches from $B_{1.2}$ to $B_{3.1}$, it is a cache miss. In the modified CCG, we have another path via the mp-node $B_{3.1}^{2,1}$. The cache miss at $B_{3.1}^{2,1}$ is (partially) masked by the branch misprediction delay, but this cache miss results in early prefetching of $B_{3.1}$ and hence a subsequent cache hit. Therefore, number of cache misses at $B_{3.1}$ is reduced.

Among the destructive effects, in the modified CCG $cm_{i.j}$ will include the cache misses of $B_{i.j}$ due to normal as well as speculative execution of other memory blocks (i.e., category 1 cache misses described in Section 4). In order to capture the delay introduced due to category 2 misses, we modify the objective function:

$$Time = \sum_{i=1}^{N}(cost_i \times v_i + bmp \times bm_i + \sum_{j=1}^{n_i} cmp \times cm_{i.j}) + \sum_{\substack{b \in Branches(P) \\ X \in 0,1}} mp\_delay(b, X) \quad (5)$$

where $mp\_delay(b, X)$ is the delay imposed due to cache misses in the mispredicted path of branch b with outcome X.

4.3 Additional constraints

We introduce some additional constraints to model the effect of speculation on caching. The execution count of a normal l-block is equal to the execution count of the basic block it belongs to. However, for an mp-node $m^{b,X}$, this count is dependent on the number of mispredictions at branch b with actual outcome X. To derive this execution count, note that the number of mp-nodes missed due to a single misprediction is $\left\lceil \frac{bmp}{cmp} \right\rceil$ where bmp (cmp) denotes branch misprediction penalty (cache miss penalty). In accordance with most modern processors we assume $bmp < cmp$ and therefore $\left\lceil \frac{bmp}{cmp} \right\rceil = 1$. This assumption is however not required and our modeling can be easily extended. Given $bmp < cmp$, a single misprediction can result in only one cache miss along the mispredicted path. Let $Spec(b, X) = \langle m_1, \ldots, m_k \rangle$. Therefore, execution count of the mp-node $m_i^{b,X}$ is

$$bm_b^X - \sum_{l=1}^{i-1} cm_{m_l}^{b,X}$$

where bm_b^X is the number of mispredictions at branch b with outcome X (obtained from the modeling of branch prediction) and $cm_{m_l}^{b,X}$ is the number of cache misses at the mp-node $m_l^{b,X}$. Constraints on $cm_{m_l}^{b,X}$ are obtained from the CCG as shown in Equation 3.

We introduce additional constraints on CCG edges to get tighter WCET estimates. Consider the self-loop $B_{3.1}^{2,1} \rightarrow B_{3.1}^{2,1}$ in Figure 1(c). With constraints only on nodes and not edges, the ILP solver will try to assign a high execution count to this self-referencing edge and thereby maximize the execution count (and cache misses) on the edge $B_{1.2} \rightarrow B_{3.1}$. We can solve this problem by putting an upper bound on the execution count of self-referencing edges. The bounds are obtained from static analysis of the control flow graph for normal nodes and branch misprediction analysis for the mp-nodes. Details are omitted due to space constraint.

Finally, we bound $mp\_delay(b, X)$ in Equation 5. For a delay to result from misprediction at b with actual outcome X we must have a cache miss during mispredicted execution of b. Recall that $Spec(b, X) = \langle m_1, \ldots, m_k \rangle$. Thus

$$mp\_delay(b, X) = \sum_{i=1}^{k}(cm_{m_i}^{b,X} \times delay_{m_i}^{b,X})$$

where $delay_{m_i}^{b,X}$ is the delay introduced due to cache miss of m_i along the mispredicted path of branch b with actual outcome X. This delay is not a constant, as part or all of the cache miss delay cmp can be masked depending on the location of the cache miss in the mispredicted path. Thus

$$delay_{m_i}^{b,X} = cmp - (bmp - \sum_{l=1}^{i-1} cost_{m_l})$$

where $cost_{m_l}$ is the execution time of the l-block m_l assuming zero cache miss and perfect branch prediction.

5. EXPERIMENTAL RESULTS

We select ten different benchmarks for our experiments (refer Table 1). They have been widely used in prior studies on WCET analysis [7, 12, 14]. Many of these benchmarks (such as `bsearch`, `des`, `dhry`) contain large number of *hard-to-predict* conditional branches arising from if-then-else statements within nested loops.

5.1 Methodology

In our experiments we assume a perfect processor pipeline with no stalls due to data dependencies. This makes each in-

Pgm.	WCET			Misprediction		Cache miss		mp_delay	
	Obs.	Est.	Ratio	Obs.	Est.	Obs.	Est.	Obs.	Est.
matsum	105504	105917	1.00	203	203	307	409	613	6
matmul	25155	25679	1.02	204	215	945	975	621	790
isort	48685	48836	1.00	391	400	107	109	495	495
bsearch	506	546	1.07	8	10	33	35	43	53
fdct	8798	8803	1.00	8	8	626	626	25	30
fft	219428	229651	1.04	3094	5139	21	25	26	0
dhry	218684	232523	1.06	2603	2514	8125	9639	9014	8088
des	87436	96437	1.10	574	1460	3255	3497	1839	3732
whet	545544	581557	1.06	3752	10580	765	986	769	432
djpeg	44962565	65184227	1.44	384102	1356349	2431414	3304327	1598854	3962793

Table 2: Observed and estimated WCET (in processor cycles), misprediction count, cache miss count, and additional delay due to partially masked cache misses on mispredicted paths (mp_delay)

Pgm.	Description
`matsum`	Summation of two 100 × 100 matrices
`matmul`	Multiplication of two 10 × 10 matrices
`isort`	Insertion sort of 100-element array
`bsearch`	Binary search of 100 element array
`fft`	1024-point Fast Fourier Transform
`fdct`	Fast Discrete Cosine Transform
`dhry`	Dhrystone benchmark
`des`	Data Encryption Standard
`whet`	Whetstone benchmark
`djpeg`	Decompresses 128 × 96 color JPG image

Table 1: Description of benchmark programs.

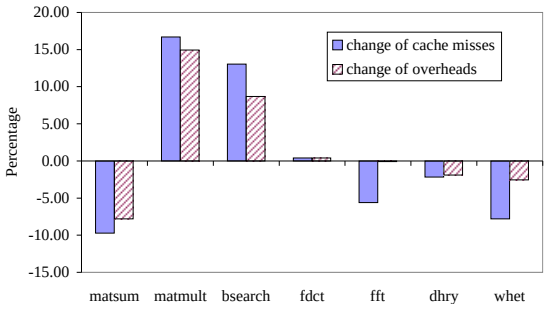


Figure 2: Comparison of combined modeling of cache and speculation with individual modelings.

struction take a fixed number of clock cycles to execute. The only time overhead is introduced by instruction cache misses and branch mispredictions. We assume that the branch misprediction penalty is 5 clock cycles and cache miss penalty is 10 clock cycles. Needless to say, any other choice of penalties can also be used.

We use SimpleScalar architectural simulation platform [1] to evaluate the accuracy of our analysis. SimpleScalar instruction set architecture (ISA) is a superset of MIPS ISA - a popular embedded processor. Given a benchmark program, we attempt to identify the program input that will generate the WCET. Among the benchmarks, `matsum`, `matmult`, `fft`, `fdct`, `dhry` and `whet` have only one possible input and we can get the *actual* WCET. For the other programs, determining the worst-case input can be tedious. Therefore we use human guidance to select a set of inputs (which are suspected to increase execution time). We then simulate the execution of the benchmark programs using SimpleScalar with these selected inputs. We report the profile for the selected input which maximizes the execution time and call it *observed* WCET.

We wrote a prototype analyzer that accepts assembly language code annotated with loop bounds and recursion depths. Various techniques exist currently for offline computation of such loop bound annotations [9]. Our analyzer is parameterized w.r.t. cache configuration, cache miss penalty, predictor table size, choice of prediction schemes and misprediction penalty. The analyzer first disassembles the code, identifies the basic blocks and constructs the the CFG. From the CFG, our analyzer automatically generates the objective function and the linear constraints. These constraints are submitted to an ILP solver to obtain *estimated* WCET. For our experiments, we use CPLEX [5], a commercial ILP solver.

5.2 Results

First, we illustrate the importance of combined modeling of cache and speculation for WCET analysis by comparing it against a naive technique which models both cache and speculation but ignores the cache-speculation interaction. Figure 2 shows this comparison with benchmarks for which we can find the actual WCET and the corresponding cache miss and branch misprediction overheads through simulation. The first group of bars indicate the percentage increase/decrease in cache misses due to the effect of branch prediction on cache behavior. The second group of bars show the percentage change in total timing overhead of cache miss and branch misprediction due to cache-speculation interaction. The timing overhead shows similar behavior as cache misses (i.e., both the bars show similar trends in each benchmark). The results show that if the naive modeling was used (i.e., the effect of branch prediction on cache was not modeled), the WCET can either be overestimated (as the downward bars indicate in `matsum`, `fft`, `dhry` and `whet`), or, more seriously, be underestimated (as the upward bars indicate in `matmult`, `bsearch` and `fdct`).

The accuracy our WCET estimation technique is shown in Table 2. The first four smaller benchmarks are evaluated with a cache of 8 lines (each line is 16-bytes). The other larger benchmarks are evaluated with a cache of 16 lines (each line is 32 bytes). This is because the first four benchmarks are too small to be simulated/estimated against larger cache sizes. For modeling branch prediction, we use the popular *gshare* scheme [19] with a 16 entry prediction ta-

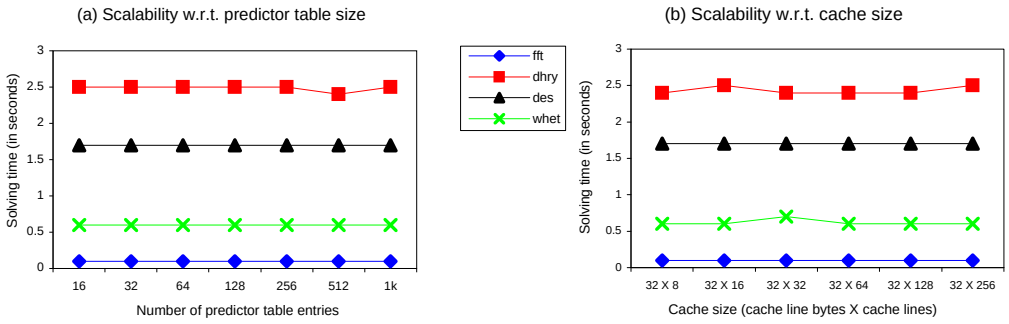


Figure 3: scalability w.r.t increasing hardware complexity

ble. The observed WCET (obtained by SimpleScalar simulation) is a lower bound on the actual WCET. The estimated WCET, which is computed by analysis, is an upper bound on actual WCET. Thus *estimated WCET* $\geq$ *actual WCET* $\geq$ *observed WCET*. The observed (estimated) misprediction count is the total misprediction count that produces the observed (estimated) WCET; similarly for the cache misses. The experiments were performed on a Pentium IV 1.3 GHz workstation with 1 GB of memory.

As we can see from the `Ratio` column, most benchmarks have very tight estimated bounds. Some benchmarks have lower estimated misprediction counts than their observed counterparts, such as `dhry`. This is because the ILP solver may trade lower mispredictions for higher cache misses to maximize the overall WCET. The `djpeg` benchmark has a higher `Est./Obs.` ratio because finding an input close to the actual worst case and thus the actual WCET is extremely hard. In this case, we generate a set of random compressed images to obtain the observed WCET.

We also study the variation of ILP solution time for some benchmarks with larger predictor table sizes (*gshare* scheme) and cache sizes. Figure 3 shows that the ILP solving time for each benchmark does not change significantly with increasing cache and branch prediction table sizes, thereby exhibiting good scalability.

6. DISCUSSION

In this paper, we have modeled speculation, caching and their indirect interactions (prefetching of instructions during speculative execution). Our experimental results indicate that our modeling is accurate, tightly estimating the maximum execution time. In future we plan to extend our micro-architectural modeling to real-life processors involving caches, pipeline and speculation.

7. ACKNOWLEDGMENTS

This work was partially supported by NUS research grant R252-000-088-112.

8. REFERENCES

[1] D. Burger, T. Austin, and S. Bennett. Evaluating Future Microprocessors: The SimpleScalar Toolset. Technical Report CS-TR96-1308, Univ. of Wisconsin - Madison, 1996.

[2] K. Chen, S. Malik, and D.I. August. Retargatable static software timing analysis. In *IEEE/ACM Intl. Symp. on System Synthesis (ISSS)*, 2001.

[3] A. Colin and I. Puaut. Worst case execution time analysis for a processor with branch prediction. *Jnl. of Real time Systems*, May 2000.

[4] J. Combs, C. B. Combs, and J. P. Shen. Mispredicted path cache effects. In *Euro-Par Conference*, 1999.

[5] CPLEX. The ILOG CPLEX optimizer v7.5, 2002. Commercial software, `http://www.ilog.com`.

[6] J. Engblom and B. Jonsson. Processor pipelines and their properties for static WCET analysis. In *Intl. Conf. on Embedded Software (EmSoft), LNCS 2491*, 2002.

[7] C. Ferdinand, F. Martin, and R. Wilhelm. Applying compiler techniques to cache behavior prediction. In *ACM Intl. Workshop on Languages, Compilers and Tools for Real-Time Sys. (LCTRTS)*, 1997.

[8] C. Healy et al. Bounding pipeline and instruction cache performance. *IEEE Trans. on Computers*, 48(1), 1999.

[9] C. Healy, M. Sjodin, V. Rustagi, and D. Whalley. Bounding loop iterations for timing analysis. In *IEEE Real-time Appplications Symposium (RTAS)*, 1998.

[10] J.L. Hennessy and D.A. Patterson. *Computer Architecture-A Quantitative Approach*. Morgan Kaufmann, 1996.

[11] M. Lee, S. L. Min, and C. S. Kim. A worst case timing analysis technique for instruction prefetch buffers. *Microprocessing and Microprogramming*, 1994.

[12] Y-T. S. Li, S. Malik, and A. Wolfe. Performance estimation of embedded software with instruction cache modeling. *ACM Transactions on Design Automation of Electronic Systems*, 4(3), 1999.

[13] S.S. Lim, J.H. Han, J. Kim, and S.L. Min. A worst case timing analysis technique for in-order superscalar processors. Technical report, Seoul Nat. Univ., 1998.

[14] T. Mitra, A. Roychoudhury, and X. Li. Timing analysis of embedded software for speculative processors. In *ACM Intl. Symp. on System Synthesis (ISSS)*, 2002.

[15] J. Pierce and T. Mudge. Wrong-path instruction prefetching. In *Intl. Symp. on Microarchitecture*, 1996.

[16] C. Rochange and P. Sainrat. Difficulties in computing the WCET for processors with speculative execution. In *Intl. Workshop on Worst-Case Execution Time Analysis (WCET)*, 2002.

[17] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise WCET prediction by separated cache and path analysis. *Jnl. of Real Time Systems*, May 2000.

[18] Fabian Wolf, Jan Staschulat, and Rolf Ernst. Associative caches in formal software timing analysis. In *ACM Design Automation Conf. (DAC)*, 2002.

[19] T.Y. Yeh and Y.N. Patt. Alternative implementations of two-level adaptive branch prediction. In *ACM Intl. Symp. on Computer Architecture (ISCA)*, 1992.