A Unified WCET Analysis Framework for Multi-core Platforms

SUDIPTA CHATTOPADHYAY, National University of Singapore
LEE KEE CHONG, National University of Singapore
ABHIK ROYCHOUDHURY, National University of Singapore
TIMON KELTER, Technical University of Dortmund
PETER MARWEDEL, Technical University of Dortmund
HEIKO FALK, Ulm University

With the advent of multi-core architectures, worst case execution time (WCET) analysis has become an increasingly difficult problem. In this paper, we propose a unified WCET analysis framework for multi-core processors featuring both shared cache and shared bus. Compared to other previous works, our work differs by modeling the interaction of shared cache and shared bus with other basic micro-architectural components (*e.g.* pipeline and branch predictor). In addition, our framework does not assume a *timing anomaly free* multi-core architecture for computing the WCET. A detailed experiment methodology suggests that we can obtain reasonably tight WCET estimates in a wide range of benchmark programs.

Categories and Subject Descriptors: C.3 [**Special-purpose and Application-based Systems**]: Real-time and embedded systems

General Terms: Design, Performance, Verification

Additional Key Words and Phrases: WCET, shared cache, shared bus, multi-core

1. INTRODUCTION

Hard real-time systems require absolute guarantees on program execution time. Worst case execution time (WCET) has therefore become an important problem to address. WCET of a program depends on the underlying hardware platform. Therefore, to obtain a safe upper bound on WCET, the underlying hardware need to be modeled. However, performance-enhancing micro-architectural features of a processor (*e.g.* cache, pipeline) make WCET analysis a very challenging task.

With the rapid growth of multi-core architectures, it is quite evident that the multi-core processors are soon going to be adopted for real-time system design. Although multi-core processors are aimed for improving performance, they introduce additional challenges in WCET analysis. Multi-core processors employ shared resources. Two meaningful examples of such shared resources are shared caches and shared buses. The presence of a shared cache requires the modeling of inter-core cache conflicts. On the other hand, the presence of a shared bus introduces variable bus access latency to accesses to shared cache and shared main memory. The delay introduced by shared cache conflict misses and shared bus accesses is propagated by different pipeline stages and affects the overall execution time of a program. WCET analysis is further complicated by a commonly known phenomenon called *timing anomalies* [Lundqvist and Stenström 1999]. In the presence of timing anomalies, a local worst case scenario may not lead to the WCET of the overall program. As an example, a *cache hit* rather than a *cache miss* may lead to the WCET of the entire program. Therefore, we cannot always assume a *cache miss* or *maximum bus delay* as the worst case scenario, as the assumptions are not just *imprecise*, but they may also lead to an *unsound* WCET estimation. A few solutions have been proposed which model the shared cache and/or the shared bus ([Yan and Zhang 2008; Li et al. 2009; Chattopadhyay et al. 2010; Kelter et al. 2011; Lv et al. 2010]) in isolation, but all of these previous solutions ignore the interactions of shared resources with important micro-architectural features such as pipelines and branch predictors.

In this paper, we propose a WCET analysis framework for multi-core platforms featuring both a shared cache and a shared bus. In contrast to previous work, our analysis can efficiently model the interaction of the shared cache and bus with different other micro-architectural features (*e.g.* pipeline, branch prediction). A few such meaningful interactions include the effect of shared cache conflict misses and shared bus delays on the pipeline, the effect of speculative execution on the shared cache etc. Moreover, our analysis framework does not rely on a *timing-anomaly free* archi-

tecture and gives a *sound* WCET estimate even in the presence of timing anomalies. In summary, the central contribution of this paper is to propose a unified analysis framework that features most of the basic micro-architectural components (pipeline, (shared) cache, branch prediction and shared bus) in a multi-core processor.

Our analysis framework deals with timing anomalies by representing the timing of each pipeline stage as an *interval*. The interval covers all possible latencies of the corresponding pipeline stage. The latency of a pipeline stage may depend on cache miss penalties and shared bus delays. On the other hand, cache and shared bus analysis interact with the pipeline stages to compute the possible latencies of a pipeline stage. Our analysis is context sensitive — it takes care of different procedure call contexts and different micro-architectural contexts (*i.e.* cache and bus) when computing the WCET of a single basic block. Finally, WCET of the entire program is formulated as an integer linear program (ILP). The formulated ILP can be solved by any commercial solver (*e.g.* CPLEX) to get the whole program's WCET.

We have implemented our framework in an extended version of Chronos [Li et al. 2007], a freely available, open-source, single-core WCET analysis tool. To evaluate our approach, we have also extended a *cycle-accurate simulator* [Austin et al. 2002] with both shared cache and shared bus support. Our experiments with moderate to large size benchmarks from [Gustafsson et al. 2010] show that we can obtain tight WCET estimates for most of the benchmarks in a wide range of micro-architectural configurations.

2. RELATED WORK

WCET analysis in single core. Research in single-core WCET analysis has started a few decades ago. Initial works used only integer linear programming (ILP) for both micro-architectural modeling and path analysis [Li et al. 1999]. However, the work proposed in [Li et al. 1999] faces scalability problems due to the explosion in number of generated ILP constraints. In [Theiling et al. 2000], a novel approach has been proposed, which employs abstract interpretation for micro-architectural modeling and ILP for path analysis. Subsequently, an iterative fixed-point analysis has been proposed in [Li et al. 2006] for modeling advanced micro-architectural features such as out-of-order and superscalar pipelines. The pipeline analysis proposed in [Li et al. 2006] has later been extended to consider parametric execution context in [Rochange and Sainrat 2009]. An ILP-based modeling of branch predictors has been proposed, among others, in [Li et al. 2005] and the work has later been extended by [Maiza and Rochange 2011] to automatically generate models for different dynamic branch prediction schemes. Our baseline framework is built upon the techniques proposed in [Li et al. 2006; Li et al. 2005].

Timing analysis of shared cache. Although there has been a significant progress in single-core WCET analysis research, little has been done so far in WCET analysis for multi-cores. Multi-core processors employ shared resources (*e.g.* shared cache, shared bus), which gives rise to a new problem for modeling inter-core conflicts. A few solutions have already been proposed for analyzing a shared cache [Yan and Zhang 2008; Li et al. 2009; Hardy et al. 2009]. All of these approaches extend the abstract interpretation based cache analysis proposed in [Theiling et al. 2000]. However, in contrast to our proposed framework, these approaches model the shared cache in isolation, assume a timing-anomaly-free architecture and ignore the interaction of shared cache with different other micro-architectural features (*e.g.* pipeline and branch prediction). A recent approach [Chattopadhyay and Roychoudhury 2011] has enhanced the abstract interpretation based shared cache analysis with a gradual and controlled use of model checking. In [Chattopadhyay and Roychoudhury 2011], abstract interpretation is used as a baseline analysis. Subsequently, a model checking pass is applied to improve the result generated by abstract interpretation. Since abstract interpretation is inherently path insensitive, it generates some spurious cache conflicts due to the presence of infeasible program paths. However, due to the path sensitive search process employed by a model checker, it eliminates certain *spurious* shared cache conflicts that can never be realized in any real execution. [Chattopadhyay and Roychoudhury 2011]

does not model the shared bus and any improvement generated by the approach proposed in [Chattopadhyay and Roychoudhury 2011] will directly improve the precision of WCET prediction using our framework.

Timing analysis of shared bus. Shared bus analysis introduces several difficulties in accurately analyzing the variable bus delay. It has been shown in [Wilhelm et al. 2009] that a time division multiple access (TDMA) scheme would be useful for WCET analysis due to its statically predictable nature. Subsequently, the analysis of TDMA based shared bus was introduced in [Rosen et al. 2007]. In [Rosen et al. 2007], it has been shown that a statement inside a loop may exhibit different bus delays in different iterations. Therefore, all loop iterations are virtually unrolled for accurately computing the bus delays of a memory reference inside loop. As loop unrolling is sometimes undesirable due to its inherent computational complexity, [Chattopadhyay et al. 2010] proposed a TDMA bus analysis technique which analyzes the loop without unrolling it. However, [Chattopadhyay et al. 2010] requires some fixed alignment cost for each loop iteration. Such a loop alignment ensures that a particular memory reference inside the loop suffers exactly the same bus delay in any iteration. The analysis proposed in [Chattopadhyay et al. 2010] is fast, as it avoids loop unrolling, however imprecise due to the alignment cost added for each loop iteration. Finally, [Kelter et al. 2011] proposes an efficient TDMA-based bus analysis technique which avoids full loop unrolling, but it is almost as precise as [Rosen et al. 2007]. The analysis time in [Kelter et al. 2011] significantly improves compared to [Rosen et al. 2007]. However, none of the works ([Rosen et al. 2007; Chattopadhyay et al. 2010; Kelter et al. 2011]) model the interaction of shared bus with pipeline and branch prediction. Additionally, [Rosen et al. 2007] and [Chattopadhyay et al. 2010] assume a *timing-anomaly-free* architecture. A recent approach [Lv et al. 2010] has combined abstract interpretation and model checking for WCET analysis in multi-cores. The micro-architecture analyzed by [Lv et al. 2010] contains a private cache for each core and it has a shared bus connecting all the cores to access main memory. The framework uses abstract interpretation ([Theiling et al. 2000]) for analyzing the private cache and it uses model checking to analyze the shared bus. However, [Lv et al. 2010] ignores the interaction of shared bus with pipeline and branch prediction. It is also unclear whether the proposed framework would remain scalable in the presence of shared cache and other micro-architectural features (*e.g.* pipeline).

Time predictable micro-architecture and execution model. To eliminate the problem of pessimism in multi-core WCET analysis, researchers have proposed predictable multi-core architectures [Paolieri et al. 2009a] and predictable execution models by code transformations [Pellizzoni et al. 2011]. The work in [Paolieri et al. 2009a] proposes several micro-architectural modifications (*e.g.* shared cache partitioning among cores, TDMA round robin bus) so that the existing WCET analysis methodologies for single cores can be adopted for analyzing the hard real-time software running on such system. On the other hand, [Pellizzoni et al. 2011] proposes compiler transformations to partition the original program into several *time-predictable* intervals. Each such interval is further partitioned into *memory phase* (where memory blocks are prefetched into cache) and *execution phase* (where the task does not suffer any last level cache miss and it does not generate any traffic to the shared bus). As a result, any other bus traffic scheduled during the execution phases of all other tasks does not suffer any additional delay due to the bus contention. We argue that the above mentioned approaches are orthogonal to the idea of this paper and our idea in this paper can be used to pinpoint the sources of overestimation in multi-core WCET analysis.

In summary, there has been little progress on multi-core WCET analysis by modeling the different micro-architectural components (*e.g.* shared cache, shared bus) in isolation. Our work differs from all previous works by proposing a unified framework, which is able to analyze the most basic micro-architectural components and their interactions in a multi-core processor.

3. BACKGROUND

In this section, we introduce the basic background behind our WCET analysis framework. Our WCET analysis framework for multi-core is based on the pipeline modeling of [Li et al. 2006].

Pipeline modeling through execution graphs. The central idea of pipeline modeling revolves around the concept of the *execution graph* [Li et al. 2006]. The execution graph is constructed for each basic block in the program control flow graph (CFG). For each instruction in the basic block, the corresponding execution graph contains a node for each of the pipeline stages. We assume a five stage pipeline — *instruction fetch* (IF), *decode* (ID), *execution* (EX), *write back* (WB) and *commit* (CM). Edges in the execution graph capture the dependencies among pipeline stages; either due to resource constraints (instruction fetch queue size, reorder buffer size etc.) or due to data dependency (*read after write hazard*). The timing of each node in the execution graph is represented by an interval, which covers all possible latencies suffered by the corresponding pipeline stage.

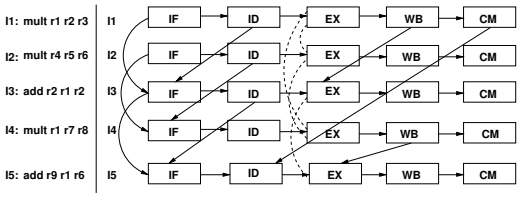


Fig. 1. Execution graph for the example program in a 2-way superscalar processor with 2-entry instruction fetch queue and 4-entry reorder buffer. Solid edges show the dependency between pipeline stages, whereas the dotted edges show the contention relation

Figure 1 shows a snippet of assembly code and the corresponding execution graph. The example assumes a 2-way superscalar processor with 2-entry instruction fetch queue (IFQ) and 4-entry reorder buffer (ROB). Since the processor is a 2-way superscalar, instruction I3 cannot be fetched before the fetch of I1 finishes. This explains the edge between IF nodes of I1 and I3. On the other hand, since IFQ size is 2, IF stage of I3 cannot start before ID stage of I1 finishes (edge between ID stage of I1 and IF stage of I3). Note that I3 is data dependent on I1 and similarly, I5 is data dependent on I4. Therefore, we have edges from WB stage of I1 to EX stage of I3 and also from WB stage of I4 to EX stage of I5. Finally, as ROB size is 4, I1 must be removed from ROB (*i.e.* committed) before I5 can be decoded. This explains the edge from CM stage of I1 to ID stage of I5.

A dotted edge in the execution graph (*e.g.* the edge between EX stage of I2 and I4) represents contention relation (*i.e.* a pair of instructions which may contend for the same functional unit). Since I2 and I4 may contend for the same functional unit (multiplier), they might delay each other due to contention. The pipeline analysis is iterative. Analysis starts without any timing information and assumes that all pairs of instructions which use same functional units and can coexist in the pipeline, may contend with each other. In the example, therefore, the analysis starts with $\{$(I1,I2), (I2,I4), (I1,I4), (I3,I5)$\}$ in the contention relation. After one iteration, the timing information of each pipeline stage is obtained and the analysis may rule out some pairs from the contention relation if their timing intervals do not overlap. With this updated contention relation, the analysis is repeated and subsequently, a refined timing information is obtained for each pipeline stage. Analysis is terminated when no further elements can be removed from the contention relation. WCET of the code snippet is then given by the worst case completion time of the CM node for I5.

4. OVERVIEW OF OUR ANALYSIS

Figure 2 gives an overview of our analysis framework. Each processor core is analyzed at a time by taking care of the *inter-core conflicts* generated by all other cores. Figure 2 shows the analysis flow for some program A running on a dedicated processor core. The overall analysis can broadly be classified into two separate phases: 1) micro-architectural modeling and 2) path analysis. In micro-architectural modeling, the timing behavior of different hardware components is analyzed (as shown by the big dotted box in Figure 2). We use abstract interpretation (AI) based cache analysis [Theiling et al. 2000] to categorize memory references as all-hit (AH) or all-miss (AM) in L1 and L2 cache. A memory reference is categorized AH (AM) if the resulting access is always a *cache hit* (*miss*). If a memory reference cannot be categorized as AH or AM, it is categorized as *unclassified* (NC). In the presence of a shared L2 cache, categorization of a memory reference may change from AH to NC due to the *inter-core conflicts* [Li et al. 2009].

Moreover, as shown in Figure 2, L1 and L2 cache analysis has to consider the effect of *speculative execution* when a branch instruction is mispredicted (refer to Section 7 for details). Similarly, the

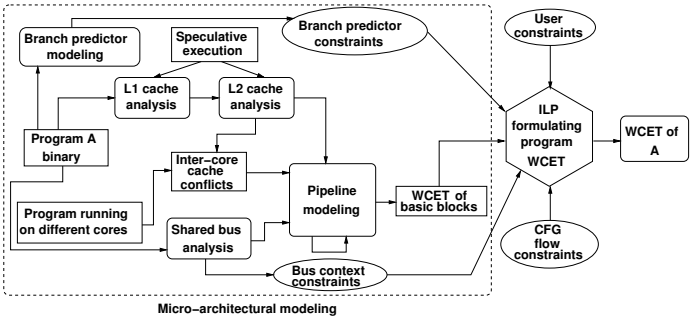


Fig. 2. Overview of our analysis framework

timing effects generated by the mispredicted instructions are also taken into account during the iterative pipeline modeling (refer to [Li et al. 2006] for details). The *shared bus analysis* computes the bus context under which an instruction can execute. The outcome of cache analysis and shared bus analysis is used to compute the latency of different pipeline stages during the analysis of the pipeline (refer to Section 5 for details). Pipeline modeling is iterative and it finally computes the WCET of each basic block. WCET of the entire program is formulated as *maximizing* the objective function of a *single integer linear program* (ILP). WCETs of individual basic blocks are used to construct the objective function of the formulated ILP. The constraints of the ILP are generated from the structure of the program's control flow graph (CFG), micro-architectural modeling (branch predictor and shared bus) and additional user-given constraints (*e.g.* loop bounds). The modeling of the branch predictor generates constraints to bound the execution count of mispredicted branches (for details refer to [Li et al. 2005]). On the other hand, constraints generated for bus contexts bound the execution count of a basic block under different bus contexts (for details, refer to Section 6). *Path analysis* finds the longest feasible program path from the formulated ILP through *implicit path enumeration* (IPET). Any ILP solver (*e.g.* CPLEX) can be used for IPET and for deriving the whole program's WCET.

System and application model. We assume a multi-core processor with each core having a private L1 cache. Additionally, multiple cores share a L2 cache. The extension of our framework for more than two levels of caches is straightforward. If a memory block is not found in L1 or L2 cache, it has to be fetched from the main memory. Any memory transaction to L2 cache or main memory has to go through a shared bus. For shared bus, we assume a *TDMA-based round robin arbitration policy*, where a fixed length bus slot is assigned to each core. We also assume fully separated caches and buses for instruction and data memory. Therefore, the data references do not interfere with the instruction references. In this work, we only model the effect of instruction caches. However, the data cache effects can be considered in a similar fashion. Since we consider only instruction caches, the cache miss penalty (computed from cache analysis) directly affects the *instruction fetch* (IF) stage of the pipeline. We do not consider *self modifying* code and therefore, we do not need to model the coherence traffic. Finally, we consider the *LRU cache replacement policy* and *non-inclusive* caches only. Later in Section 11 and in Section 12, we shall extend our framework for FIFO cache replacement policy and we shall also discuss the extension of our framework for other cache replacement policies (*e.g.* PLRU), other cache hierarchies (*e.g.* inclusive) and data caches.

5. INTERACTION OF SHARED RESOURCES WITH PIPELINE

Let us assume each node i in the execution graph is annotated with the following timing parameters, which are computed iteratively:

— $earliest[t_i^{ready}], earliest[t_i^{start}], earliest[t_i^{finish}]$: Earliest ready, earliest start and earliest finish time of node i, respectively.

— $latest[t_i^{ready}], latest[t_i^{start}], latest[t_i^{finish}]$: Latest ready, latest start and latest finish time of node i, respectively.

For each pipeline stage i, $earliest[t_i^{ready}]$ and $earliest[t_i^{start}]$ are initialized to *zero*, whereas, $earliest[t_i^{finish}]$ is initialized to the *minimum latency* suffered by the pipeline stage i. On the other hand, $latest[t_i^{ready}]$, $latest[t_i^{start}]$ and $latest[t_i^{finish}]$ are all initialized to ∞ for each pipeline stage i. The active time span of node i can be captured by the interval $[earliest[t_i^{ready}], latest[t_i^{finish}]]$. Therefore, each node of the execution graph is initialized with a timing interval $[0, \infty]$.

Pipeline modeling is iterative. The iterative analysis starts with the coarse interval $[0, \infty]$ for each node and subsequently, the interval is tightened in each iteration. The computation of a precise interval takes into account the analysis result of caches and shared bus. The iterative analysis eliminates certain infeasible contention among the pipeline stages in each iteration, thereby leading to a tighter timing interval after each iteration. The iterative analysis starts with a contention relation. Such a contention relation contains pairs of instructions which may potentially delay each other due to contention. Initially, all possible pairs of instructions are included in the contention relation and after each iteration, pairs of instructions whose timing intervals do not overlap, are removed from this relation. If the contention relation does not change in some iteration, the iterative analysis terminates. Since the number of instructions in a basic block is finite, the contention relation contains a finite number of elements and in each iteration, at least one element is removed from the relation. Therefore, this analysis is guaranteed to terminate. Moreover, if the contention relation does not change, the timing interval of each node reaches a fixed-point after the analysis terminates. In the following, we shall discuss how the presence of a shared cache and a shared bus affects the timing information of different pipeline stages.

5.1. Interaction of shared cache with pipeline

Let us assume $CHMC_i^{L1}$ ($CHMC_i^{L2}$) denotes the AH/AM/NC cache *hit-miss* classification of an IF node i in L1 (shared L2) cache. Further assume that E_i denotes the possible latencies of an *IF* node i without considering any shared bus delay. E_i can be defined as follows:

$$E_i = \begin{cases} 1, \text{ if } CHMC_i^{L1} = AH; \\ LAT^{L1} + 1, \text{ if } CHMC_i^{L1} = AM \wedge CHMC_i^{L2} = AH; \\ LAT^{L1} + LAT^{L2} + 1, \text{ if } CHMC_i^{L1} = AM \wedge CHMC_i^{L2} = AM; \\ [LAT^{L1} + 1, LAT^{L1} + LAT^{L2} + 1], \text{ if } CHMC_i^{L1} = AM \wedge CHMC_i^{L2} = NC; \\ [1, LAT^{L1} + 1], \text{ if } CHMC_i^{L1} = NC \wedge CHMC_i^{L2} = AH; \\ [1, LAT^{L1} + LAT^{L2} + 1], \text{ otherwise.} \end{cases} \quad (1)$$

where LAT^{L1} and LAT^{L2} represent the fixed L1 and L2 cache miss latencies respectively. Note that the interval-based representation captures the possibilities of both a *cache hit* and a *cache miss* in case of an NC categorized cache access. Therefore, the computation of E_i can also deal with the architectures that exhibit timing anomalies.

5.2. Interaction of shared bus with pipeline

Let us assume that we have a total of $\mathcal{C}$ cores and the TDMA-based round robin scheme assigns a slot length S_l to each core. Therefore, the length of one complete *round* is $S_l\mathcal{C}$. We begin with the following definitions which are used throughout the paper:

Definition 5.1. (*TDMA offset*) A *TDMA offset* at a particular time T is defined as the relative distance of T from the beginning of the last scheduled *round*. Therefore, at time T, the TDMA offset can be precisely defined as $T \ mod \ S_l\mathcal{C}$.

Definition 5.2. (*Bus context*) A *Bus context* for a particular execution graph node i is defined as the set of TDMA offsets reaching/leaving the corresponding node. For each execution graph node i, we track the incoming bus context (denoted O_i^{in}) and the outgoing bus context (denoted O_i^{out}).

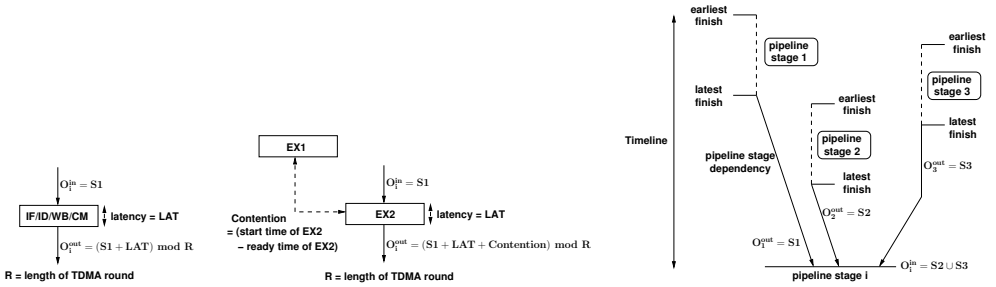


Fig. 3. Computation of outgoing and incoming bus contexts. (a) Computation of outgoing bus context for IF/ID/WB/CM pipeline stages; (b) computation of outgoing bus context for an EX pipeline stage, the outgoing bus context must take into account the resource contention; (c) computation of incoming bus context to pipeline stage i, since pipeline stage 2 always finishes after pipeline stage 1, the outgoing bus context from pipeline stage 1 (*i.e.* $O_1^{out} = S1$) cannot be propagated to pipeline stage i

For a task executing in core p (where $0 \leq p < \mathcal{C}$), $latest[t_i^{finish}]$ and $earliest[t_i^{finish}]$ are computed for an IF execution graph node i as follows:

$$latest[t_i^{finish}] = latest[t_i^{start}] + max\_lat_p(O_i^{in}, E_i) \tag{2}$$

$$earliest[t_i^{finish}] = earliest[t_i^{start}] + min\_lat_p(O_i^{in}, E_i) \tag{3}$$

Note that max_lat_p, min_lat_p are not constants and depend on the incoming bus context (O_i^{in}) and the set of possible latencies of IF node i (E_i) in the absence of a shared bus. max_lat_p and min_lat_p are defined as follows:

$$max\_lat_p(O_i^{in}, E_i) = \begin{cases} 1, & \text{if } CHMC_i^{L1} = AH; \\ \max\limits_{o \in O_i^{in}, t \in E_i} \Delta_p(o,t), & \text{otherwise.} \end{cases} \tag{4}$$

$$min\_lat_p(O_i^{in}, E_i) = \begin{cases} 1, & \text{if } CHMC_i^{L1} \neq AM; \\ \min\limits_{o \in O_i^{in}, t \in E_i} \Delta_p(o,t), & \text{otherwise.} \end{cases} \tag{5}$$

In the above, E_i represents the set of possible latencies of an *IF* node i in the absence of shared bus delay (refer to Equation 1). Given a TDMA offset o and latency t in the absence of shared bus delay, $\Delta_p(o,t)$ computes the total delay (including shared bus delay) faced by the IF stage of the pipeline. $\Delta_p(o,t)$ can be defined as follows (similar to [Chattopadhyay et al. 2010] or [Kelter et al. 2011]):

$$\Delta_p(o,t) = \begin{cases} t, & \text{if } pS_l \leq o + t \leq (p+1)S_l; \\ t + pS_l - o, & \text{if } o < pS_l; \\ t + (\mathcal{C}+p)S_l - o, & \text{otherwise.} \end{cases} \tag{6}$$

In the following, we shall now show the computation of incoming and outgoing bus contexts (*i.e.* O_i^{in} and O_i^{out} respectively) for an execution graph node i.

Computation of O_i^{out} from O_i^{in}. The computation of O_i^{out} depends on O_i^{in}, on the possible latencies of execution graph node i (including shared bus delay) and on the contention suffered by the corresponding pipeline stage. In the modeled pipeline, inorder stages (*i.e.* IF, ID, WB and CM) do not suffer from contention. But the out-of-order stage (*i.e.* EX stage) may experience contention when it is *ready* to execute (*i.e.* operands are available) but cannot *start* execution due to the unavailability of a functional unit. Worst case contention period of an execution graph node i can be

denoted by the term $latest[t_i^{start}] - latest[t_i^{ready}]$. For *best case* computation, we conservatively assume the absence of contention. Therefore, for a particular core p ($0 \leq p < \mathcal{C}$), we compute O_i^{out} from the value of O_i^{in} as follows:

$$O_i^{out} = \begin{cases} u(O_i^{in}, E_i + [0, latest[t_i^{start}] - latest[t_i^{ready}]]), & \text{if } i = EX; \\ u(O_i^{in}, \bigcup_{o \in O_i^{in}, t \in E_i} \Delta_p(o,t)), & \text{if } i = IF; \\ u(O_i^{in}, E_i), & \text{otherwise.} \end{cases} \tag{7}$$

Here, u denotes the update function on TDMA offset set with a set of possible latencies of node i and is defined as follows:

$$u(O, X) = \bigcup_{o \in O, t \in X} \{(o + t) \ mod \ S_l\mathcal{C}\} \tag{8}$$

Figure 3(a) and Figure 3(b) capture the computation of outgoing bus context from a non-EX pipeline stage and an EX pipeline stage, respectively. Note that $E_i + [0, latest[t_i^{start}] - latest[t_i^{ready}]]$ captures all possible latencies suffered by the execution graph node i, taking care of contentions as well. Therefore, O_i^{out} captures all possible TDMA offsets exiting node i, when the same node is entered with bus context O_i^{in}. More precisely, assuming that O_i^{in} represents an over-approximation of the incoming bus context at node i, the computation by Equation 7 ensures that O_i^{out} represents an over-approximation of the outgoing bus context from node i.

Computation of O_i^{in}. The value of O_i^{in} depends on the value of O_j^{out}, where j is a predecessor of node i in the execution graph. If $pred(i)$ denotes all the predecessors of node i, clearly, $\cup_{j \in pred(i)} O_j^{out}$ gives a *sound* approximation of O_i^{in}. However, it is important to observe that not all predecessors in the execution graph can propagate TDMA offsets to node i. Recall that the edges in the execution graph represent dependency (either due to resource constraints or due to true data dependences). Therefore, node i in the execution graph can only *start* when all the nodes in $pred(i)$ have *finished*. Consequently, the TDMA offsets are propagated to node i only from the predecessor j, which finishes *immediately* before i is ready. Nevertheless, our static analyzer may not be able to compute a single predecessor that propagates TDMA offsets to node i. However, for two arbitrary execution graph nodes $j1$ and $j2$, if we can guarantee that $earliest[t_{j2}^{finish}] > latest[t_{j1}^{finish}]$, we can also guarantee that $j2$ finishes *later* than $j1$. Figure 3(c) demonstrates the computation of O_i^{in} from the outgoing bus contexts of three predecessors. Formally, the computation of O_i^{in} captures the following property:

$$O_i^{in} = \bigcup \{O_j^{out} \mid j \in pred(i) \wedge \ earliest[t_{pmax}^{finish}] \leq latest[t_j^{finish}]\} \tag{9}$$

where $pmax$ is a predecessor of i such that $latest[t_{pmax}^{finish}] = \max_{j \in pred(i)} latest[t_j^{finish}]$. Therefore, O_i^{in} captures all possible outgoing TDMA offsets from the predecessor nodes that are *possibly* finished *latest*. Given that the value of O_j^{out} is an over-approximation of the outgoing bus context for each predecessor j of i, Equation 9 gives an over-approximation of the incoming bus context at node i. Finally, Equation 7 and Equation 9 together ensure a sound computation of the bus contexts at the entry and exit of each execution graph node.

6. WCET COMPUTATION UNDER MULTIPLE BUS CONTEXTS

6.1. Execution context of a basic block

Computing bus context without loops. In the previous section, we have discussed the pipeline modeling of a basic block B in isolation. However, to correctly compute the execution time of B, we need to consider 1) contentions (for functional units) and data dependencies among instructions prior to B and instructions in B; 2) contentions among instructions after B and instructions in B. Set of instructions before (after) B which directly affect the execution time of B is called the *prologue*

(*epilogue*) of B [Li et al. 2006]. B may have multiple prologues and epilogues due to the presence of multiple program paths. However, the size of any prologue or epilogue is bounded by the total size of IFQ and ROB. To distinguish the execution contexts of a basic block B, execution graphs are constructed for each possible combination of prologues and epilogues of B. Each execution graph of B contains the instructions from B itself (called *body*) and the instructions from one possible prologue and epilogue. Assume we compute the incoming (outgoing) bus context $O_i^{in}(p,e)$ ($O_i^{out}(p,e)$) at body node i for prologue p and epilogue e (using the technique described in Section 5). After we finish the analysis of B for all possible combinations of prologues and epilogues, we compute an *over-approximation* of O_i^{in} (O_i^{out}) by *merge* operation as follows: $O_i^{in} = \bigcup_{p,e} O_i^{in}(p,e)$ and $O_i^{out} = \bigcup_{p,e} O_i^{out}(p,e)$. Clearly, O_i^{in} (O_i^{out}) captures an over-approximation of the bus context at the entry (exit) of node i, irrespective of any prologue or epilogue of B.

Computing bus context in the presence of loops. In the presence of loops, a basic block can be executed with different bus contexts at different loop iterations. The bus contexts at different iterations depend on the set of instructions which can propagate TDMA offsets across loop iterations.

For each loop l, we compute two sets of nodes — π_l^{in} and π_l^{out}. π_l^{in} are the set of pipeline stages which can propagate TDMA offsets across iterations, whereas, π_l^{out} are the set of pipeline stages which could propagate TDMA offsets outside of the loop. Therefore, π_l^{in} corresponds to the pipeline stages of instructions inside l which resolve *loop carried dependency* (due to resource constraints, pipeline structural constraints or true data dependency). On the other hand, π_l^{out} corresponds to the pipeline stages of instructions inside l which resolve the dependency of instructions outside of l. Figure 4 demonstrates the π_l^{out} and π_l^{in} nodes for a sample execution graph.

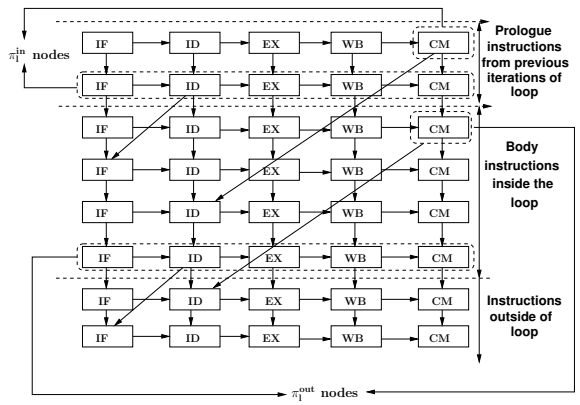


Fig. 4. π_l^{in} and π_l^{out} nodes shown with the example of a sample execution graph. π_l^{in} nodes propagate bus contexts across iterations, whereas, π_l^{out} nodes propagate bus contexts outside of loop.

The bus context at the entry of all *non-first* loop iterations can be captured as $(O_{x1}^{in}, O_{x2}^{in}, \ldots, O_{xn}^{in})$ where $\pi_l^{in} = \{x1, x2, \ldots, xn\}$. The bus context at the first iteration is computed from the bus contexts of instructions prior to l (using the technique described in Section 5). Finally, O_{xi}^{out} for any $xi \in \pi_l^{out}$ can be responsible for affecting the execution time of any basic block outside of l.

6.2. Bounding the execution count of a bus context

Foundation. As discussed in the preceding, a basic block inside some loop may execute under different bus contexts. For all *non-first* iterations, a loop l is entered with bus context $(O_{x1}^{in}, O_{x2}^{in}, \ldots, O_{xn}^{in})$ where $\{x1, x2, \ldots, xn\}$ are the set of π_l^{in} nodes as described in Figure 4. These bus contexts are computed during an iterative analysis of the loop l (described below). On the other hand, the bus context at the first iteration of l is a tuple of *TDMA offsets* propagated from outside of l to some pipeline stage inside l. Note that the bus context at the first iteration of l is computed by following the general procedure as described in Section 5.

In this section, we shall show how the execution count of different bus contexts can be bounded by generating additional ILP constraints. These additional constraints are added to a global ILP formulation to find the WCET of the entire program. We begin with the following notations:

Ω_l. The set of all bus contexts that may reach loop l in any iteration.

Ω_l^s. The set of all bus contexts that may reach loop l at first iteration. Clearly, $\Omega_l^s \subseteq \Omega_l$. Moreover, if l is contained inside some outer loop, l would be invoked more than once. As a result, Ω_l^s may contain more than one element. Note that Ω_l^s can be computed as a tuple of TDMA offsets propagated from outside of l to some pipeline stage inside l. Therefore, Ω_l^s can be computed during the procedure described in Section 5. If l is an inner loop, an element of Ω_l^s is computed (as described in Section 5) for each analysis invocation of the loop immediately enclosing l.

G_l^s. For each $s_0 \in \Omega_l^s$, we build a *flow graph* $G_l^s = (V_l^s, F_l^s)$ where $V_l^s \subseteq \Omega_l$. The graph G_l^s captures the transitions among different bus contexts across loop iterations. An edge $f_{w_1 \rightarrow w_2} = (w_1, w_2) \in F_l^s$ exists (where $w_1, w_2 \in \Omega_l$) if and only if l can be entered with bus context w_1 at some iteration n and with bus context w_2 at iteration $n+1$. Note that G_l^s cannot be infinite, as we have only finitely few bus contexts that are the nodes of G_l^s.

M_l^w. Number of times the body of loop l is entered with bus context $w \in \Omega_l$ in any iteration.

$M_l^{w_1 \rightarrow w_2}$. Number of times l can be entered with bus context w_1 at some iteration n and with bus context w_2 at iteration $n+1$ (where $w_1, w_2 \in \Omega_l$). Clearly, if $f_{w_1 \rightarrow w_2} \notin F_l^s$ for any flow graph G_l^s, $M_l^{w_1 \rightarrow w_2} = 0$.

Construction of G_l^s. For each loop l and for each $s_0 \in \Omega_l^s$, we construct a flow graph G_l^s. Initially, G_l^s contains a single node representing bus context $s_0 \in \Omega_l^s$. After analyzing all the basic blocks inside l (using the technique described in Section 5), we may get a new bus context at some node $i \in \pi_l^{in}$ (recall that π_l^{in} are the set of execution graph nodes that may propagate bus context across loop iterations). As a byproduct of this process, we also get the WCET of all basic blocks inside l when the body of l is entered with bus context s_0. Let us assume that for any $s \in \Omega_l \backslash \Omega_l^s$ and $i \in \pi_l^{in}$, $s(i)$ represents the bus context O_i^{in}. Suppose we get a new bus context $s_1 \in \Omega_l$ after analyzing the body of l once. Therefore, we add an edge from s_0 to s_1 in G_l^s. We continue expanding G_l^s until $s_n(i) \subseteq s_k(i)$ for all $i \in \pi_l^{in}$ and for some $1 \leq k \leq n-1$ (where $s_n \in \Omega_l$ represents the bus context at the entry of l after it is analyzed n times). In this case, we finish the construction of G_l^s by adding a backedge from s_{n-1} to s_k. We also stop expanding G_l^s if we have expanded as many times as the relative loop bound of l. Note that G_l^s contains at least two nodes, as the bus context at first loop iteration is always distinguished from the bus contexts in any other loop iteration.

It is worth mentioning that the construction of G_l^s is *much less computationally intensive* than a full unrolling of l. The bus context at the entry of l quickly reaches a fixed-point and we can stop expanding G_l^s. In our experiments, we found that the number of nodes in G_l^s never exceeds ten. For very small loop bounds (typically less than 5), the construction of G_l^s continues till the loop bound. For larger loop bounds, most of the time, the construction of G_l^s reaches the diverged bus context $[0, \ldots, S_l\mathcal{C} - 1]$ quickly (in less than ten iterations). As a result, through a small node count in G_l^s, we are able to avoid the computationally intensive unrolling of every loop.

Generating separate ILP constraints. Using each flow graph G_l^s for loop l, we generate ILP constraints to distinguish different bus contexts under which a basic block can be executed. In an abuse of notation, we shall use $w.i$ to denote that the basic block i is reached with bus context $w.i$ when the immediately enclosing loop of i is reached with bus context w in any iteration. The following ILP constraints are generated to bound the value of M_l^w:

$$\forall w \in \Omega_l : \sum_{x \in \Omega_l} M_l^{x \rightarrow w} = M_l^w;\ M_l^w - 1 \leq \sum_{x \in \Omega_l} M_l^{w \rightarrow x} \leq M_l^w \tag{10}$$

$$\sum_{w \in \Omega_l} M_l^w = N_{l.h} \tag{11}$$

where $N_{l.h}$ denotes the number of times the header of loop l is executed. Equation 10 generates standard flow constraints from each graph G_l^s, constructed for loop l. Special constraints need to be added for the bus contexts with which the loop is entered at the first iteration and at the last iteration.

If w is a bus context with which loop l is entered at the last iteration, M_l^w is more than the execution count of outgoing flows (*i.e.* $M_l^{w \rightarrow x}$). Equation 10 takes this special case into consideration. On the other hand, Equation 11 bounds the aggregate execution count of all possible contexts $w \in \Omega_l$ with the total execution count of the loop header. Note that $N_{l.h}$ will further be involved in defining the CFG structural constraints, which relate the execution count of a basic block with the execution count of its incoming and outgoing edges [Theiling et al. 2000]. Equations 10-11 do not ensure that whenever loop l is invoked, the loop must be executed at least once with some bus context in Ω_l^s. We add the following ILP constraints to ensure this:

$$\forall w \in \Omega_l^s : \; M_l^w \geq N_{l.h}^{w.h} \tag{12}$$

Here $N_{l.h}^{w.h}$ denotes the number of times the header of loop l is executed with bus context w. The value of $N_{l.h}^{w.h}$ is further bounded by the CFG structural constraints.

The constraints generated by Equations 10-12 are sufficient to derive the WCET of a basic block in the presence of non-nested loops. In the presence of nested loops, however, we need additional ILP constraints to relate the bus contexts at different loop nests. Assume that the loop l is enclosed by an outer loop l'. For each $w' \in \Omega_{l'}$, we may get a different element $s_0 \in \Omega_l^s$ and consequently, a different $G_l^s = (V_l^s, E_l^s)$ for loop l. Therefore, we have the following ILP constraints for each flow graph G_l^s:

$$\forall G_l^s = (V_l^s, E_l^s) : \sum_{w \in V_l^s} M_l^w \leq bound_l * (\sum_{w' \in parent(G_l^s)} M_{l'}^{w'}) \tag{13}$$

where $bound_l$ represents the *relative loop bound* of l and $parent(G_l^s)$ denotes the set of bus contexts in $\Omega_{l'}$ for which the flow graph G_l^s is constructed at loop l. The left-hand side of Equation 13 accumulates the execution count of all bus contexts in the flow graph G_l^s. The total execution count of all bus contexts in V_l^s is bounded by $bound_l$, for each construction of G_l^s (as $bound_l$ is the relative loop bound of l). Since G_l^s is constructed $\sum_{w' \in parent(G_l^s)} M_{l'}^{w'}$ times, the total execution count of all bus contexts in V_l^s is bounded by the right hand side of Equation 13.

Finally, we need to bound the execution count of any basic block i (immediately enclosed by loop l), with different bus contexts. We generate the following two constraints to bound this value:

$$\sum_{w \in \Omega_l} N_i^{w.i} = N_i \tag{14}$$

$$\forall w \in \Omega_l : \; N_i^{w.i} \leq M_l^w \tag{15}$$

where N_i represents the total execution count of basic block i and $N_i^{w.i}$ represents the execution count of basic block i with bus context $w.i$. Equation 15 tells the fact that basic block i can execute with bus context $w.i$ at some iteration of l only if l is reached with bus context w at the same iteration (by definition). N_i will be further constrained through the structure of program's CFG, which we exclude in our discussion.

Computing bus contexts at loop exit. To derive the WCET of the whole program, we need to estimate the bus context exiting a loop l (say O_l^{exit}). A recently proposed work ([Kelter et al. 2011]) has shown the computation of O_l^{exit} without a full loop unrolling. In this paper, we use a similar technique as in [Kelter et al. 2011] with one important difference: In [Kelter et al. 2011], a single offset graph G_{off} is maintained, which tracks the outgoing bus context from each loop iteration. Once G_{off} got stabilized, a separate ILP formulation on G_{off} derives the value of O_l^{exit}. In the presence of pipelined architectures, O_i^{out} for any $i \in \pi_l^{out}$ could be responsible for propagating bus context outside of l (refer to Figure 4). Therefore, a separate offset graph is maintained for each $i \in \pi_l^{out}$ (say G_{off}^i) and an ILP formulation for each G_{off}^i can derive an estimation of the bus context exiting the loop (say O_i^{exit}). In [Kelter et al. 2011], it has been proved that the computation of O_l^{exit} is always an *over-approximation* (*i.e. sound*). Given that the value of each O_i^{out} is *sound*,

it is now straightforward to see that the computation of each O_i^{exit} is also *sound*. For details of this analysis, readers are further referred to [Kelter et al. 2011].

7. EFFECT OF BRANCH PREDICTION

Presence of branch prediction introduces additional complexity in WCET computation. If a conditional branch is mispredicted, the timing of the mispredicted instructions need to be computed. Mispredicted instructions introduce additional conflicts in L1 and L2 cache which need to be modeled for a sound WCET computation. Similarly, branch misprediction will also affect the bus delay suffered by the subsequent instructions. In the following, we shall describe how our framework models the interaction of branch predictor with caches and bus. It is important to note that the speculation also impacts the timing of different pipeline stages. Our proposed multi-core WCET analysis framework is based on the work proposed in [Li et al. 2006], which considers the timing effects into pipeline due to speculation by augmenting the execution graph along wrong path execution (*i.e.* along the mispredicted branch path). As the central theme of this paper is to discuss the WCET analysis in multi-core, we shall only discuss the impact of speculation on (shared) caches and bus. For timing interactions that are not specific to multi-core (*e.g.* timing interactions between speculation and pipeline), we refer to our previous work [Li et al. 2006].

Effect on cache for speculative execution. We assume that there could be at most one unresolved branch at a time. Therefore, the number of mispredicted instructions is bounded by the number of instructions till the next branch as well as the total size of instruction fetch queue and reorder buffer.

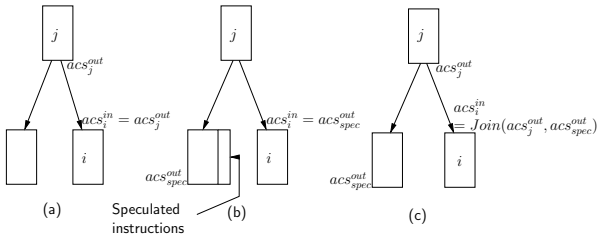


Fig. 5. Computation of acs_i^{in} when the edge (a) $j \rightarrow i$ is correctly predicted, and (b) when the edge $j \rightarrow i$ is mispredicted, (c) A *safe* approximation of acs_i^{in} by considering both correct and incorrect prediction of edge $j \rightarrow i$.

Abstract-interpretation-based cache analysis produces a fixed point on abstract cache content at the entry (denoted as ACS_i^{in}) and at the exit (denoted as ACS_i^{out}) of each basic block i. If a basic block i has multiple predecessors, output cache states of the predecessors are *joined* to produce the input cache state of basic block i. Consider an edge $j \rightarrow i$ in the program's CFG. If $j \rightarrow i$ is an unconditional edge, computation of ACS_i^{in} does not require any change. However, if $j \rightarrow i$ is a conditional edge, the condition could be *correctly* or *incorrectly* predicted during the execution. For a correct prediction, the cache state ACS_i^{in} is still *sound*. On the other hand, for incorrect prediction, ACS_i^{in} must be updated with the memory blocks accessed at the mispredicted path. We assume that there could be at most one unresolved branch at a time. Therefore, the number of mispredicted instructions is bounded by the number of instructions till the next branch as well as the total size of instruction fetch queue and reorder buffer. To maintain a *safe* cache state at the entry of each basic block i, we join the two cache states arising due to the *correct* and *incorrect* predictions of conditional edge $j \rightarrow i$. We demonstrate the entire scenario through an example in Figure 5. In Figure 5, we demonstrate the procedure for computing the abstract cache state at the entry of a basic block i. Basic block i is conditionally reached from basic block j. To compute a *safe* cache content at the entry of basic block i, we combine two different possibilities —- one when the respective branch is *correctly* predicted (Figure 5(a)) and the other when the respective branch is *incorrectly* predicted (Figure 5(b)). The combination is performed through an abstract *join* operation, which depends on the type of analysis (*must* or *may*) being computed. A stabilization on the abstract cache contents at the entry and exit of each basic block is achieved through conventional fixed point analysis.

It is worthwhile to mention that our analysis *conservatively* estimates the impact of speculation on the cache content. The cache hit/miss categorization of a memory reference is determined via the

over-approximation (during the *may* analysis) and the under-approximation (during the *must* analysis) of cache contents at each program point. The *join* operation, as shown in Figure 5(c), ensures that the over-approximation (under-approximation) of cache contents is preserved during the *may* (*must*) analysis in the presence of zero or more branch mispredictions. It is important to note that such an over-approximation and under-approximation consider all possible branch misprediction scenarios (including zero branch mispredictions). Therefore, our analysis preserves the *soundness* in the presence of timing anomaly. However, our modeling is also conservative in the sense that we do not take into account the exact number of branch mispredictions into account for cache analysis.

Effect on bus for speculative execution. Due to branch misprediction, some additional instructions might be fetched from the mispredicted path. As described in Section 6, an execution graph for each basic block B contains a *prologue* (instructions before B which directly affect the execution time of B). If the last instruction of the prologue is a conditional branch, the respective execution graph is augmented with the instructions along the mispredicted path ([Li et al. 2006]). Since the propagation of *bus context* is entirely performed on the execution graph (as shown in Section 5), our shared bus analysis remains unchanged, except the fact that it works on an augmented execution graph (which contains instructions from the mispredicted path) in the presence of speculation.

Computing the number of mispredicted branches. In the presence of a branch predictor, each conditional edge $j \rightarrow i$ in the program CFG can be correctly or incorrectly predicted. Let us assume $E_{j\rightarrow i}$ denotes the total number of times control flow edge $j \rightarrow i$ is executed and $E^c_{j\rightarrow i}$ ($E^m_{j\rightarrow i}$) denotes the number of times the control flow edge $j \rightarrow i$ is executed due to correct (incorrect) branch prediction. Clearly, $E_{j\rightarrow i} = E^c_{j\rightarrow i} + E^m_{j\rightarrow i}$. Value of $E_{j\rightarrow i}$ is further bounded by CFG structural constraints. On the other hand, values of $E^c_{j\rightarrow i}$ and $E^m_{j\rightarrow i}$ depend on the type of branch predictor. We use our prior work ([Li et al. 2005]), where we have shown how to bound the values of $E^c_{j\rightarrow i}$ and $E^m_{j\rightarrow i}$ for history based branch predictors. The constraints generated on $E^c_{j\rightarrow i}$ and $E^m_{j\rightarrow i}$ are as well captured in the global ILP formulation to compute the whole program WCET. We exclude the details of branch predictor modeling in this paper — interested readers are referred to [Li et al. 2005].

8. WCET COMPUTATION OF AN ENTIRE PROGRAM

We compute the WCET of the entire program with N basic blocks by using the following objective function:

$$Maximize\, T = \sum_{i=1}^{N} \sum_{j\rightarrow i} \sum_{w\in\Omega_i} t^{c,w}_{j\rightarrow i} * E^{c,w}_{j\rightarrow i} + t^{m,w}_{j\rightarrow i} * E^{m,w}_{j\rightarrow i} \tag{16}$$

Ω_i denotes the set of all bus contexts under which basic block i can execute. Basic block i can be executed with different bus contexts. However, the number of elements in Ω_i is always bounded by the number of bus contexts entering the loop immediately enclosing i (refer to Section 6). $t^{c,w}_{j\rightarrow i}$ denotes the WCET of basic block i when the basic block i is reached from basic block j, the control flow edge $j \rightarrow i$ is correctly predicted and i is reached with bus context $w \in \Omega_i$. Similarly, $t^{m,w}_{j\rightarrow i}$ denotes the WCET of basic block i under the same bus context but when the control flow edge $j \rightarrow i$ was mispredicted. Note that both $t^{c,w}_{j\rightarrow i}$ and $t^{m,w}_{j\rightarrow i}$ are computed during the iterative pipeline modeling (with the modifications proposed in Section 5). $E^{c,w}_{j\rightarrow i}$ ($E^{m,w}_{j\rightarrow i}$) denotes the number of times basic block i is reached from basic block j with bus context w and when the control flow edge $j \rightarrow i$ is correctly (incorrectly) predicted. Therefore, we have the following two constraints:

$$E^c_{j\rightarrow i} = \sum_{w\in\Omega_i} E^{c,w}_{j\rightarrow i}, \;\; E^m_{j\rightarrow i} = \sum_{w\in\Omega_i} E^{m,w}_{j\rightarrow i} \tag{17}$$

Constraints on $E^c_{j\rightarrow i}$ and $E^m_{j\rightarrow i}$ are proposed by the ILP-based formulation in [Li et al. 2005]. On the other hand, $E^{c,w}_{j\rightarrow i}$ and $E^{m,w}_{j\rightarrow i}$ are bounded by the CFG structural constraints ([Theiling et al. 2000]) and the constraints proposed by Equations 10-15 in Section 6. Note that

in Equations 10-15, we only discuss the ILP constraints related to the bus contexts. Other ILP constraints, such as CFG structural constraints and user constraints, are used in our framework for an IPET implementation.

Finally, the WCET of the program maximizes the objective function in Equation 16. Any ILP solver (*e.g.* CPLEX) can be used for the same purpose.

9. SOUNDNESS OF ANALYSIS

In this section, we shall provide the basic ideas for the proof of the *soundness* of our analysis framework. Due to space constraints, details of the proofs are included in the technical report [Chattopadhyay et al. 2011].

The heart of *soundness* guarantee follows from the fact that we represent the timing of each pipeline stage as an *interval*. Recall that the active timing interval of each pipeline stage is captured by $INTV_i = [earliest[t_i^{ready}], latest[t_i^{finish}]]$. Therefore, as long as we can guarantee that $INTV_i$ always *over-approximates* the actual timing interval of the corresponding pipeline stage in any concrete execution, we can also guarantee the *soundness* of our analysis. To ensure that the interval $INTV_i$ is always an over-approximation, we have to consider all possible latencies suffered by any pipeline stage. The latency of a pipeline stage, on the other hand, may be influenced by the following factors:

Cache miss penalty. Only NC categorized memory references may have variable latencies. Our analysis represents this variable latency as an interval $[lo, hi]$ (Equation 1) where lo (hi) represents the latency of a cache hit (miss).

Functional unit latency. Some functional units may have variable latencies depending on the operands (*e.g.* multiplier unit). For such functional units, we consider the EX pipeline stage latency as an interval $[lo, hi]$ where lo (hi) represents the minimum (maximum) possible latency of the corresponding functional unit.

Contention to access functional units. A pair of instructions may delay each other by contending for the same functional unit. Since only EX stage may suffer from contention, two different instructions may contend for the same functional unit only if the timing intervals of the respective EX stages *overlap*. For any pipeline stage i, an upper bound on contention (say $CONT_i^{max}$) is computed by accounting the cumulative effect of contentions created by all the *overlapping* pipeline stages (which access the same functional unit as i). We do not compute a lower bound on contention and conservatively assume a safe lower bound of 0. Finally, we add $[0, CONT_i^{max}]$ with the timing interval of pipeline stage i. Clearly, $[0, CONT_i^{max}]$ covers all possible latencies suffered by pipeline stage i due to contention.

Bus access delay. Bus access delay of a pipeline stage depends on the incoming bus contexts (O_i^{in}). Computation of O_i^{in} is always an over-approximation as evidenced by Equation 7 and Equation 9. Therefore, we can always compute the interval spanning from *minimum* to *maximum* bus delay using O_i^{in} (Equation 4 and Equation 5).

To conclude, we argue that the longest acyclic path search in the execution graph always results in a *sound* estimation of basic block WCET. Finally, the IPET approach searches for the *longest feasible program path* to ensure a *sound* estimation of whole program's WCET.

10. EXPERIMENTAL EVALUATION

Experimental setup

We have chosen moderate to large size benchmarks from [Gustafsson et al. 2010], which are generally used for timing analysis. Individual benchmarks are compiled into simplescalar PISA (Portable Instruction Set Architecture) [Austin et al. 2002] — a MIPS like instruction set architecture. We use the simplescalar gcc cross compiler with optimization level `-O2` to generate the PISA compliant binary of each benchmark. The control flow graph (CFG) of each benchmark is extracted

Table I. Default micro-architectural setting for experiments

Component	*Default settings*	*Perfect settings*
Number of cores	2	NA
pipeline	1-way, inorder 4-entry IFQ, 8-entry ROB	NA
L1 instruction cache	2-way associative, 1 KB miss penalty = 6 cycles	All accesses are L1 *hit*
L2 instruction cache	4-way associative, 4 KB miss penalty = 30 cycles	NA
Shared bus	slot length = 50 cycles	Zero bus delay
Branch predictor	2 level predictor, L1 size=1 L2 size=4, history size=2	Branch prediction is *always correct*

from its PISA compliant binary and is used as an input to our analysis framework. In our current implementation, the analysis frontend (CFG extractor) and the modeling of pipeline do not appropriately handle `recursions`, `switch` cases and unstructured `goto`, `break` statements inside loops. Such programs from [Gustafsson et al. 2010] are therefore not included in our evaluation.

To validate our analysis framework, the simplescalar toolset [Austin et al. 2002] was extended to support the simulation of shared cache and shared bus. The simulation infrastructure is used to compare the estimated WCET with the observed WCET. Observed WCET is measured by simulating the program for a few program inputs. Nevertheless, we would like to point out that the presence of a shared cache and a shared bus makes the realization of the worst-case scenario extremely challenging. In the presence of a shared cache and a shared bus, the worst-case scenario depends on the interleavings of threads, which are running on different cores. Consequently, the observed WCET result in our experiments may sometimes highly under-approximate the *actual WCET*.

For all of our experiments, we present the WCET overestimation ratio, which is measured as $\frac{Estimated\ WCET}{Observed\ WCET}$. For each reported overestimation ratio, the system configuration during the analysis (which computes $Estimated\ WCET$) and the measurement (which computes $Observed\,WCET$) are kept *identical*. Unless otherwise stated, our analysis uses the default system configuration in Table I (as shown by the column "Default settings"). Since the data cache modeling is not yet included in our current implementation, all data accesses are assumed to be *L1 cache hits* (for analysis and measurement both).

To check the dependency of WCET overestimation on the type of conflicting task (being run in parallel on a different core), we use two different tasks to generate the inter-core conflicts — 1) `jfdctint`, which is a single path program and 2) `statemate`, which has a huge number of paths. In our experiments (Figures 6-8), we use `jfdctint` to generate inter-core conflicts to the first half of the tasks (*i.e.* `matmult` to `lcdnum`). On the other hand, we use `statemate` to generate inter-core conflicts to the second half of the tasks (*i.e.* `minver` to `st`). Due to the absence of any infeasible program path, inter-core conflicts generated by a single path program (*e.g.* `jfdctint`) can be more accurately modeled compared to a multi-path program (*e.g.* `statemate`). Therefore, in the presence of a shared cache, we expect a better WCET overestimation ratio for the first half of the benchmarks (*i.e.* `matmult` to `lcdnum`) compared to the second half (*i.e.* `minver` to `st`).

To measure the WCET overestimation due to *cache sharing*, we compare the WCET result with two different design choices, where the level 2 cache is partitioned. For a two-core system, two different partitioning choices are explored: first, each partition has the same number of cache sets but has half the number of ways compared to the original shared cache (called *vertical* partitioning). Secondly, each partition has half the number of cache sets but has the same number of ways compared to the original shared cache (called *horizontal* partitioning). In our default configuration, therefore, each core is assigned a 2-way associative, 2 KB L2 cache in the *vertical partitioning*, whereas each core is assigned a 4-way associative, 2 KB L2 cache in the *horizontal partitioning*.

Finally, to pinpoint the source of WCET overestimation, we can selectively turn off the analysis of different micro-architectural components. We say that a micro-architectural component has *perfect setting* if the analysis of the same is turned off (refer to the column "Perfect settings" in Table I).

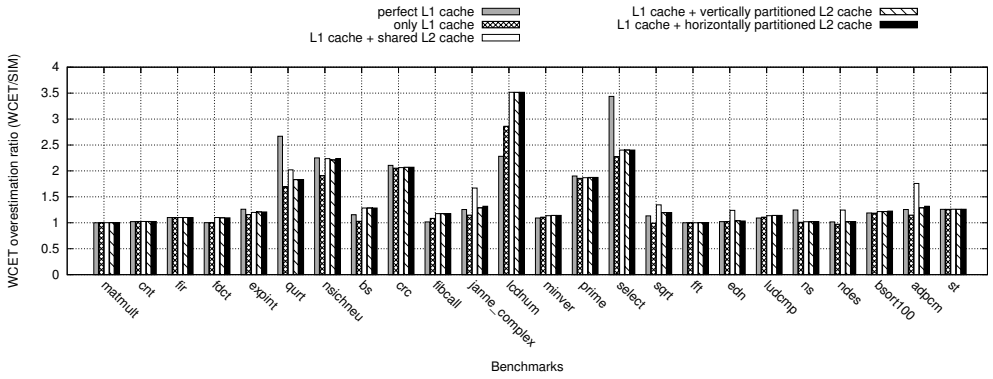


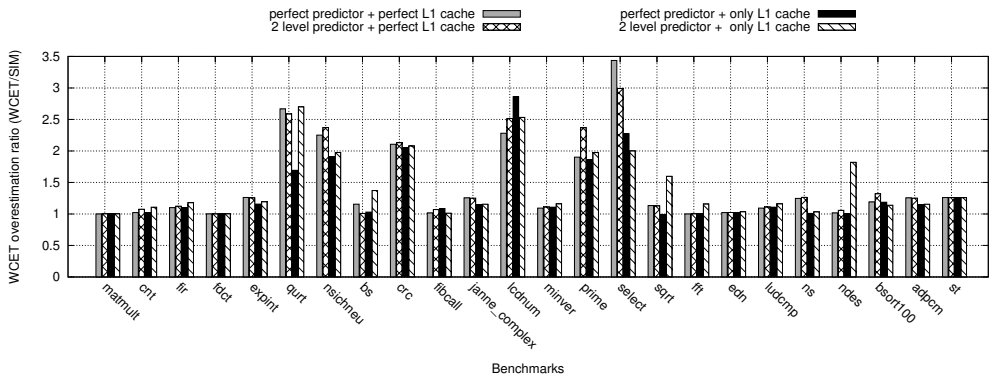
Fig. 6. Effect of shared and partitioned L2 cache on WCET overestimation

Basic analysis result

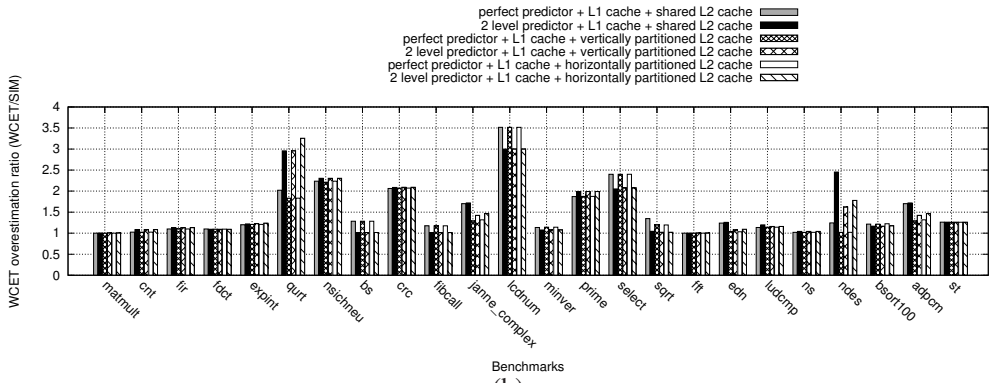
Effect of caches. Figure 6 shows the WCET overestimation ratio with respect to different L1 and L2 cache settings in the presence of a *perfect* branch predictor and a *perfect* shared bus. Results show that we can reasonably bound the WCET overestimation ratio except for a few benchmarks (*e.g.* `qurt`, `nsichneu`, `lcdnum`, `select`). The major source of this overestimation is the presence of many *infeasible* paths in such programs, which may lead to infeasible micro-architectural states and WCET overestimations. These infeasible paths can be eliminated by providing additional user constraints into our framework and hence improving the ILP-based WCET calculation. We also observe that the partitioned L2 caches may lead to a better WCET overestimation compared to the shared L2 caches, with the *vertical* L2 cache partitioning almost always working as the best choice. The positive effect of the vertical cache partitioning is visible in programs such as `adpcm`, `ndes` and `edn`, where the overestimations in the presence of shared L2 caches are higher than the same using partitioned L2 caches. This is due to the difficulty in modeling the inter-core cache conflicts from programs being run in parallel (*i.e.* `jfdctint` and `statemate`).

Effect of speculative execution. As we explained in Section 7, the presence of a branch predictor and speculative execution may introduce additional computation cycles for executing a mispredicted path. Moreover, speculative execution may introduce additional cache conflicts from a mispredicted path. The results in Figure 7(a) and Figure 7(b) show the effect of speculation in L1 and L2 cache, respectively. `qurt` and `ndes` show reasonable increases in the WCET overestimations in the presence of speculation (Figure 7(a) and Figure 7(b)). A similar increase in the WCET overestimation is also observed with `bs` and `sqrt` in the presence of L1 caches and speculation (Figure 7(a)). Such an increase in the overestimation ratio can be explained from the overestimation arising in the modeling of the effect of speculation on caches (refer to Section 7). Due to the abstract *join* operation to combine the cache states in correct and mispredicted path, we may introduce some *spurious* cache conflicts. Nevertheless, our approach for modeling the speculation effect in cache is scalable and produces tight WCET estimates for most of the benchmarks.

Effect of shared bus. Figure 8 shows the WCET overestimation in the presence of a shared cache and a shared bus. We observe that our shared bus analysis can reasonably control the overestimation due to the shared bus. Except for a few benchmarks (*e.g.* `edn`, `nsichneu`, `ndes`, `qurt`), the overestimation in the presence of a shared cache and a shared bus is mostly equal to the overestimation when the shared bus analysis is turned off (*i.e.* using a *perfect* shared bus). Recall that each overestimation ratio is computed by performing the analysis and the measurement on *identical system configuration*. Therefore, the analysis and the measurement both includes the shared bus delay only when the shared bus is *enabled*. For a *perfect shared bus setting*, both the analysis and the measurement consider a *zero latency* for all the bus accesses. As a result, we also observe that our shared



(a)



(b)

Fig. 7. (a) Effect of speculation on L1 cache, (b) effect of speculation on partitioned and shared L2 caches

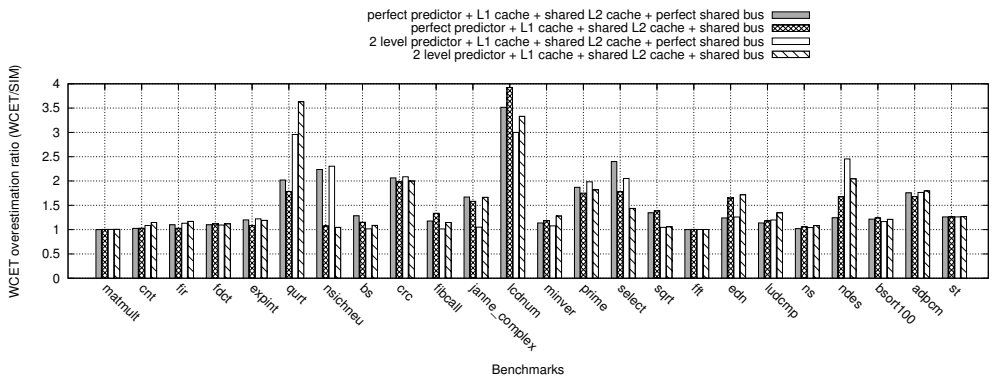


Fig. 8. Effect of shared bus on WCET overestimation

bus analysis might be more accurate than the analysis of other micro-architectural components (*e.g.* in case of `nsichneu`, `expint` and `fir`, where the WCET overestimation ratio in the presence of a shared bus might be less than the same with a *perfect* shared bus). In particular, `nsichneu` shows a drastic fall in the WCET overestimation ratio when the shared bus analysis is *enabled*. For `nsichneu`, we found that the execution time is dominated by shared bus delay, which is most accurately computed by our analysis for this benchmark. On the other hand, we observed in Figure 6

that the main source of WCET overestimation in `nsichneu` is *path analysis*, due to the presence of many infeasible paths. Consequently, when shared bus analysis is turned off, the overestimation arising from path analysis dominates and we obtain a high WCET overestimation ratio. Average WCET overestimation in the presence of both a shared cache and a shared bus is around 50%.

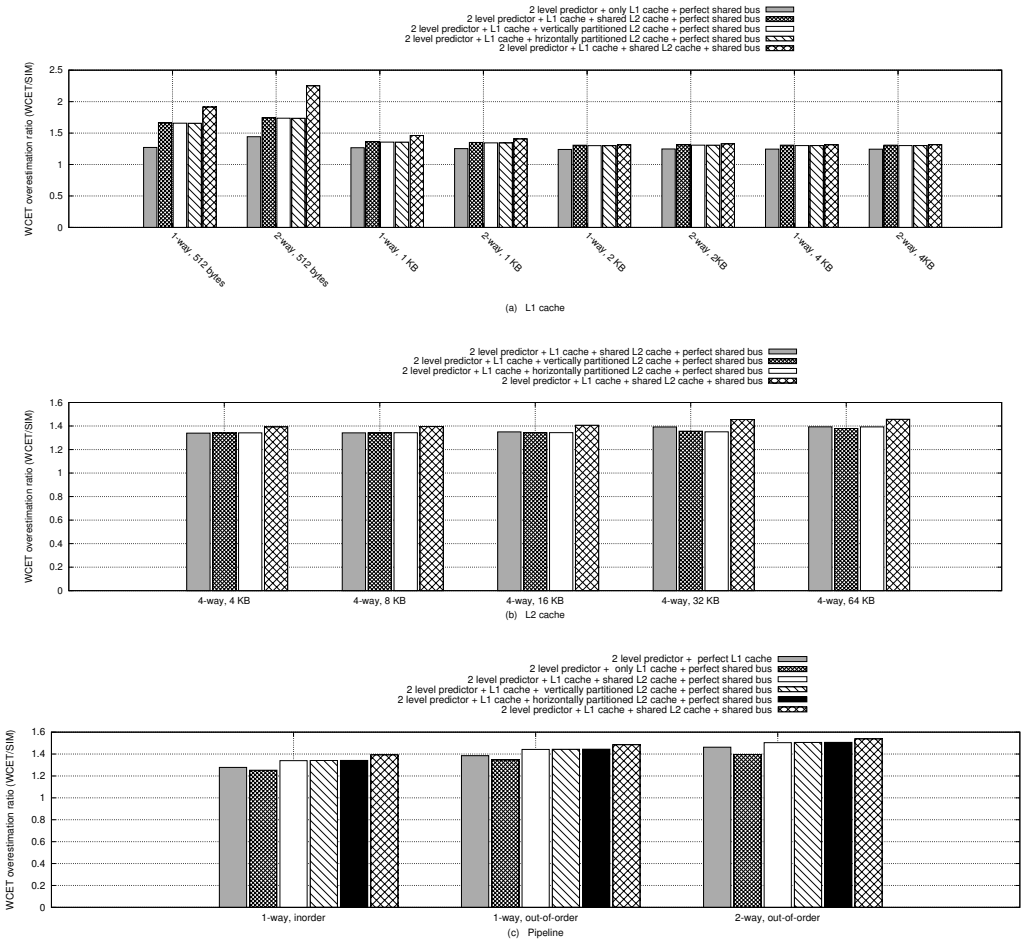


Fig. 9. WCET overestimation sensitivity w.r.t. (a) L1 cache sizes and configurations; (b) L2 cache sizes and configurations; (c) pipeline configurations

WCET analysis sensitivity w.r.t. micro-architectural parameters

In this section, we report the WCET overestimation sensitivity with respect to different micro-architectural parameters. For all the experiments (Figures 9-10), the reported WCET overestimation denotes the *geometric mean* of the term $\frac{Estimated\ WCET}{Observed\ WCET}$ over all the different benchmarks.

We evaluate our framework for different L1 and L2 cache sizes and configurations (Figure 9(a) and Figure 9(b), respectively). We observe that the average WCET overestimation is around 40% (50%) with respect to different L1 (L2) cache configurations. Figure 9(c) presents the WCET overestimation for different pipeline configurations. Superscalar pipelines increase the instruction level parallelism and so as the performance of entire program. However, it also becomes difficult to model the inherent instruction level parallelism in the presence of superscalar pipelines. Therefore, Figure

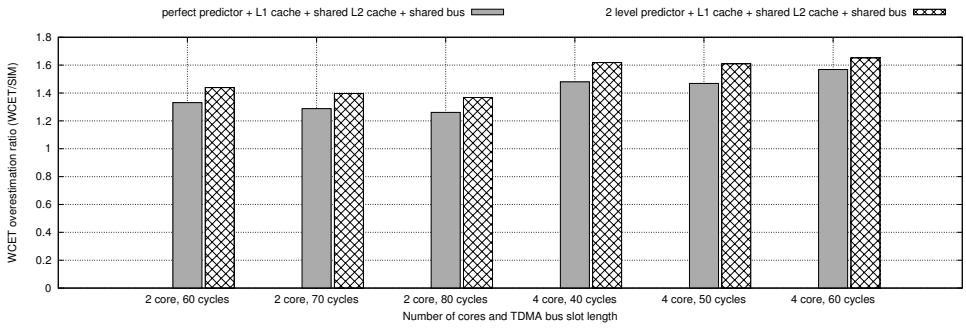


Fig. 10. WCET overestimation sensitivity w.r.t. number of cores and different bus slot lengths

9(c) shows an increase in the WCET overestimation with superscalar pipelines. Finally, Figure 10 shows the WCET overestimation sensitivity with respect to the number of cores and different bus slot lengths. For four core experiments, we take a group of four programs (from left to right as shown in Figure 6) to run in four different cores. Figure 10 reports the geometric mean of WCET overestimation over all the benchmarks. With very high length of TDMA round (*i.e.* number of cores multiplied by TDMA bus slot length), WCET overestimation normally increases (as shown in Figure 10). This is due to the fact that with higher TDMA round lengths, the search space for possible bus contexts (or set of TDMA offsets) increases. As a result, it is less probable to expose the worst-case scenario in simulation with higher bus slot lengths.

Analysis time

We have performed all the experiments on an 8 core, 2.83 GHz Intel Xeon machine having 4 GB of RAM and running Fedora Core 4 operating system. Table II reports the *maximum* analysis time when the shared bus analysis is *disabled* and Table III reports the *maximum* analysis time when all the analyses are enabled (*i.e.* cache, shared bus and pipeline). Recall from Section 4 that our WCET analysis framework is broadly composed of two different parts, namely, micro-architectural modeling and implicit path enumeration (IPET) through integer linear programming (ILP). The column labeled "μ arch" captures the time required for micro-architectural modeling. On the other hand, the column labeled "ILP" captures the time required for path analysis through IPET.

In the presence of speculative execution, number of mispredicted branches is modeled by integer linear programming [Li et al. 2005]. Such an ILP-based branch predictor modeling, therefore, increases the number of constraints which need to be considered by the ILP solver. As a result, the ILP solving time increases in the presence of speculative execution (as evidenced by the second rows of both Table II and Table III).

Shared bus analysis increases the micro-architectural modeling time (as evidenced by Table III) and the analysis time usually increases with the bus slot length. The time for the shared bus analysis generally appears from tracking the bus context at different pipeline stages. A higher bus slot length usually leads to a higher number of bus contexts to analyze, thereby increasing the analysis time.

In Table II and Table III, we have only presented the analysis time for the longest running benchmark (`nsichneu`) from our test-suite. For any other program used in our experiments, the entire analysis (micro-architectural modeling and ILP solving time) takes around 20-30 seconds on average to finish.

The results reported in Table II show that the ILP-based modeling of branch predictor usually increases the analysis time. Therefore, for a more efficient but less precise analysis of branch predictors, one can explore different techniques to model branch predictors, such as abstract interpretation. Shared bus analysis time can be reduced by using different offset abstractions, such as *interval* instead of an *offset set*. Nevertheless, the appropriate choice of analysis method and abstraction depends on the precision-scalability tradeoff required by the user.

Table II. Analysis time [of `nsichneu`] in *seconds*. The first row represents the analysis time when speculative execution was *disabled*. The second row represents the analysis time when speculation was *enabled*

Shared L2 cache										Pipeline					
4 KB		8 KB		16 KB		32 KB		64 KB		1-way inorder		1-way out-of-order		2-way superscalar	
μ arch	ILP	μ arch	ILP	μ arch	ILP	μ arch	ILP	μ arch	ILP	μ arch	ILP	μ arch	ILP	μ arch	ILP
1.2	1.3	1.4	1.3	1.7	1.3	2.3	1.3	4.8	1.2	1.3	1.3	1.2	1.3	1.3	1.4
2.6	240	2.9	240	3.5	238	4.6	238	7	239	2.6	238	2.4	239	2.8	254

Table III. Analysis time [of `nsichneu`] in *seconds*. The first row shows the analysis time when speculation was *disabled*. The second row shows the analysis time when speculation was *enabled*

Number of cores, TDMA bus slot length											
2 core, 60 cycles		2 core, 70 cycles		2 core, 80 cycles		4 core, 40 cycles		4 core, 50 cycles		4 core, 60 cycles	
μ arch	ILP	μ arch	ILP	μ arch	ILP	μ arch	ILP	μ arch	ILP	μ arch	ILP
128	4	160	4.2	198	5.1	199	7.1	228	9.3	257	12.5
205	158	261	181	363	148	373	148	441	165	521	154

11. EXTENSION OF SHARED CACHE ANALYSIS

Our discussion on cache analysis has so far concentrated on the *least-recently-used* (LRU) cache replacement policies. However, a widely used cache replacement policy is *first-in-first-out* (FIFO). FIFO cache replacement policy has been used in embedded processors such as ARM9 and ARM11 [Reineke et al. 2007]. Recently, abstract interpretation based analysis of FIFO replacement policy has been proposed in [Grund and Reineke 2009; Grund and Reineke 2010a] for single level caches and for multi-level caches in [Hardy and Puaut 2011]. In this section, we shall discuss the extension of our shared cache analysis for FIFO cache replacement policy. We shall also show that such an extension will not change the modeling of timing interactions among shared cache and other basic micro-architectural components (*e.g.* pipeline and branch predictor).

11.1. Review of cache analysis for FIFO replacement

We use the *must cache analysis* for FIFO replacement as proposed in [Grund and Reineke 2009]. In FIFO replacement, when a cache set is full and still the processor requests fresh memory blocks (which map to the same cache set), the *first* cache line entering the respective cache set (*i.e.* first-in) is replaced. Therefore, the set of *tags* in a k-way FIFO abstract cache set (say $\mathcal{A}_s$) can be arranged from last-in to first-out order (leftmost capturing the *last-in* position) as follows:

$$\mathcal{A}_s = [T_1, T_2, \ldots, T_k] \tag{18}$$

where each $T_i \subseteq \mathcal{T}$ and $\mathcal{T}$ is the set of all cache tags. Unlike LRU, cache state never changes upon a *cache hit* with FIFO replacement policy. Therefore, the cache state update on a memory reference depends on the *hit-miss* categorization of the same memory reference. Assume that a memory reference belongs to cache tag tag_i. The FIFO abstract cache set $\mathcal{A}_s = [T_1, T_2, \ldots, T_k]$ is updated on the access of tag_i as follows:

$$\tau([T_1, T_2, \ldots, T_k], tag_i) = \begin{cases} [T_1, T_2, \ldots, T_k], & \text{if } tag_i \in \bigcup_i T_i; \\ [\{tag_i\}, T_1, \ldots, T_{k-1}], & \text{if } tag_i \notin \bigcup_i T_i \wedge |\bigcup_i T_i| = k; \\ [\phi, T_1, \ldots, T_{k-1} \cup \{tag_i\}], & \text{otherwise.} \end{cases} \tag{19}$$

The first scenario captures a *cache hit* and the second scenario captures a *cache miss*. Third scenario appears when the static analysis cannot accurately determine the *hit-miss* categorization of the memory reference.

The *abstract join* function for the FIFO must cache analysis is exactly same as the LRU must cache analysis. The join function between two abstract FIFO cache sets computes the intersection of the abstract cache sets. If a cache tag is available in both the abstract cache sets, the *right most* relative position of the cache tag is captured after the join operation.

11.2. Analysis of shared cache with FIFO replacement

We implement the *must cache analysis* for FIFO replacement as described in the preceding. To distinguish the cold cache misses at the first iterations of loops and different procedure calling contexts, our cache analysis employs the *virtual-inline-virtual-unrolling* (VIVU) approach (as described in [Theiling et al. 2000]). After analyzing the L1 cache memory references are categorized as *all-hit* (AH), *all-miss* (AM) or *unclassified* (NC). AM and NC categorized memory references may access the L2 cache and therefore, the L2 cache state is updated for the memory references which are categorized AM or NC in the L1 cache (as in [Hardy and Puaut 2011]).

To analyze the shared cache, we used our previous work on shared cache [Li et al. 2009] for LRU cache replacement policy. [Li et al. 2009] employs a separate shared cache conflict analysis phase. For FIFO replacement policy too, we can use the exactly same idea to analyze the set of inter-core cache conflicts. Shared cache conflict analysis may change the categorization of a memory reference from all-hit (AH) to unclassified (NC). For the sake of illustration, assume a memory reference which accesses the memory block m. This analysis phase first computes the number of unique conflicting shared cache accesses from different cores. Then it is checked whether the number of conflicts from different cores can potentially replace m from shared cache. More precisely, for an N-way set associative cache, hit/miss categorization (CHMC) of corresponding memory reference is changed from all-hit (AH) to unclassified (NC) if and only if the following condition holds:

$$N - AGE_{fifo}(m) < |\mathcal{M}_c(m)| \tag{20}$$

where $|\mathcal{M}_c(m)|$ represents the number of conflicting memory blocks from different cores which may potentially access the same L2 cache set as m. $AGE_{fifo}(m)$ represents the relative position of memory block m in the FIFO abstract cache set and in the absence of inter-core cache conflicts. Recall that the memory blocks (or the tags) are arranged according to the *last-in* to *first-out* order in the FIFO abstract cache set. Therefore, the term $N - AGE_{fifo}(m)$ captures the maximum number of fresh memory blocks which can enter the FIFO cache before m being evicted out.

11.3. Interaction of FIFO cache with pipeline and branch predictor

As described in the preceding, after the FIFO shared cache analysis, memory references are categorized as *all-hit* (AH), *all-miss* (AM) or *unclassified* (NC). In the presence of pipeline, such a categorization of instruction memory references add computation cycle with the instruction fetch (IF) stage. Therefore, we use Equation 1 to compute the latency suffered by cache hit/miss and propagate the latency through different pipeline stages.

Recall from Section 7 that speculative execution may introduce additional cache conflicts. In Section 7, we proposed to modify the abstract interpretation based cache analysis to handle the effect of speculative execution on cache. From Figure 5, we observe that our solution is *independent* of the cache replacement policies concerned. Our proposed modification performs an *abstract join* operation on the cache states along the *correct* and *mispredicted* path (as shown in Figure 5). Therefore, for FIFO replacement polices, the abstract join operation is performed according to the FIFO replacement analysis (instead of LRU join operation we performed in case of LRU caches).

11.4. Experimental result

Figure 11 demonstrates our WCET analysis experience with FIFO replacement policy. We have used the exactly same experimental setup as mentioned in Section 10. On average, our analysis framework can reasonably bound the WCET overestimation for FIFO cache replacement, except for `fdct`, `lcdnum`, `select` and `qurt`. However, such an overestimation is largely due to the presence of a FIFO cache and not due to the presence of cache sharing, as clearly evidenced by Figure 11(a). However, as mentioned in [Berg 2006], the observed worst-case for FIFO replacement may highly under-approximate the *true worst case* due to the *domino effect*.

Figure 11(b) shows that our modeling of the interaction between FIFO cache and the branch predictor (which is configured as in Table I) does not much affect the WCET overestimation (except

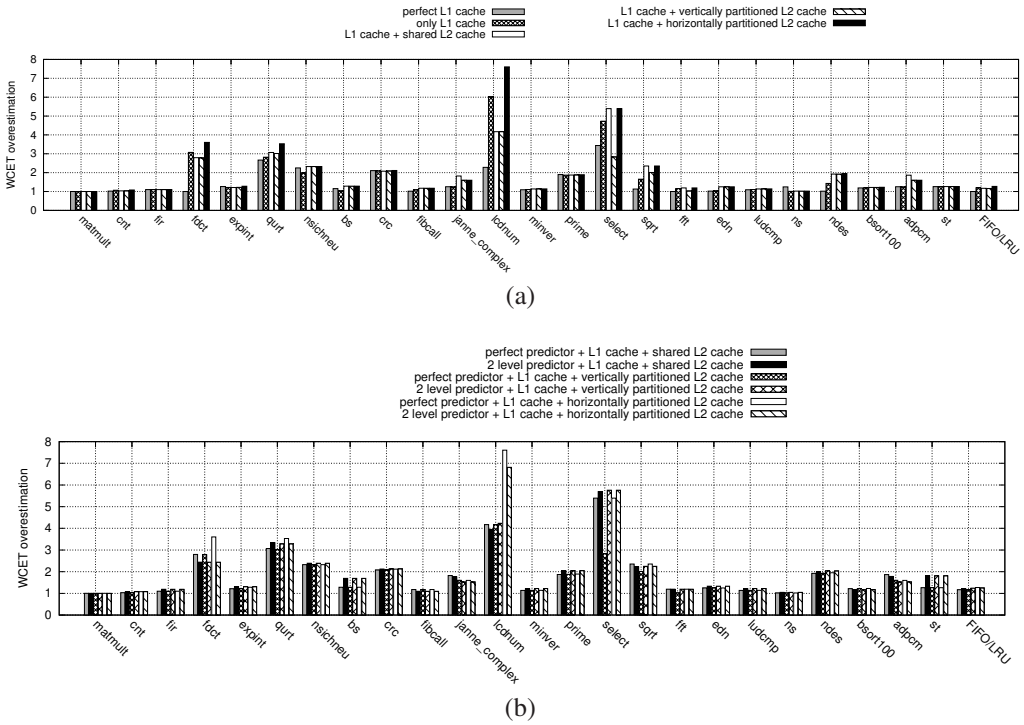


Fig. 11. Analysis of cache in the presence of FIFO replacement policy. In both the figures, the last bars labelled "FIFO/LRU" capture the geometric mean of WCET overestimation over all the benchmarks in the presence of FIFO replacement, considering LRU replacement as a baseline. The WCET overestimation of individual benchmarks use the simulation as a baseline. (a) WCET overestimation w.r.t. different L2 cache architectures, (b) WCET overestimation in the presence of FIFO cache and speculation

for `select` in the presence of vertically partitioned L2 caches). As evidenced by Figure 11(b), the average increase in the WCET overestimation is *minimal* due to the speculation.

We also report the average WCET overestimation of FIFO replacement compared to LRU replacement policy. In Figure 11, the results labelled "FIFO/LRU" capture the geometric mean of WCET overestimation in the presence of FIFO replacement, considering WCET overestimation with LRU replacement as a baseline. Our results show that FIFO replacement does not lead to more than 25% worse WCET estimate on average, when compared to the WCET estimate with LRU replacement. Therefore, we believe that FIFO is a reasonably good alternative of LRU replacement even in the context of shared caches.

11.5. Other cache organizations

In the preceding, we have discussed the extension of our WCET analysis framework with FIFO replacement policy. We have shown that as long as the cache tags in an abstract cache set can be arranged according to the order of their replacement, our shared cache conflict analysis can be integrated. Moreover, our modeling for the timing interaction among (shared) cache, pipeline and branch predictor is independent of the underlying cache replacement policy. Nevertheless, for some cache replacement policies, arranging the cache tags according to the order of their replacement poses a challenge (*e.g.* PLRU [Grund and Reineke 2010b]). Cache analysis based on *relative competitiveness* [Reineke et al. 2007] tries to analyze a cache replacement policy with respect to an equivalent LRU cache, but with different parameters (*e.g.* associativity). Any cache replacement analysis based on relative competitiveness can directly be integrated with our WCET analysis frame-

work. Nevertheless, more precise analysis than the ones based on relative competitiveness can be designed, as shown in [Grund and Reineke 2010b] for PLRU policy. However, we believe that designing more precise cache analysis is outside the scope of this paper. The purpose of our work is to propose a unified WCET analysis framework and any precision gain in the existing cache analysis technique will directly benefit our framework by improving the precision of WCET prediction.

In this paper, we have focused on the *non-inclusive* cache hierarchy. In multi-core architectures, inclusive cache hierarchy may limit performance when the size of the largest cache is not significantly larger than the sum of the smaller caches. Therefore, processor architects sometimes resort to non-inclusive cache hierarchies [Zahran et al. 2007]. On the other hand, inclusive cache hierarchies greatly simplify the cache coherence protocol. The analysis of inclusive cache hierarchy requires to take account of the *invalidations* of certain cache lines to maintain the *inclusion* property (as shown in [Hardy and Puaut 2011] for multi-level private cache hierarchies). Such invalidations may change an *all-hit* (AH) categorized memory reference to *unclassified* (NC) [Hardy and Puaut 2011]. Our shared cache conflict analysis phase can be applied on this reduced set of AH categorized memory references for inclusive caches, keeping the rest of our WCET analysis framework entirely unchanged. Therefore, we believe that the inclusive cache hierarchies do not pose any additional challenge in the context of shared caches and the analysis of such cache hierarchies can easily be integrated, keeping the rest of our WCET analysis framework unchanged.

12. DISCUSSION AND FUTURE WORK

In this paper, we have proposed a unified WCET analysis framework which considers the timing effects of shared caches and shared buses with several micro-architectural components (*e.g.* pipeline and branch predictor). The key contribution of our work is to build a coherent strategy to evaluate the impact of different micro-architectural components on WCET, both in single-core and multi-core architectures. We have evaluated our framework which models pipeline, (shared) instruction caches, dynamic branch predictions, shared buses and the non-trivial timing interactions among the different components. We model a five-stage pipeline, which is similar to the five-stage pipeline in ARM-9 embedded processors. Dynamic branch predictors are also used now in modern embedded processors, such as ARM-CortexA8, ARM-11 and PowerPC-750 [Maiza and Rochange 2011]. Shared caches are widely used in the embedded multi-core processors, such as in ARM-CortexA8. For shared buses, we have currently modeled TDMA arbitration schemes. Existing literatures [Wilhelm et al. 2009; Paolieri et al. 2009b] have discussed that TDMA arbitration is acceptable when guaranteed performance is required (*e.g.* for hard real-time systems), instead of average high performance. Real implementation of such TDMA arbitration includes AEthereal network-on-chip (NOC) [Goossens and Hansson 2010]. In our modeling, the concept of bus context and its ILP-based formulation is generic with respect to different bus arbitration policies. However, for TDMA arbitration policy, a bus context can easily be defined from TDMA offset set (cf. Definition 5.2) which in turn leads to a compositional and tractable shared bus analysis technique. For other dynamic arbitration policies (*e.g.* priority based), the number of bus contexts may easily have an exponential complexity (since the interleaving of different threads on different cores need to be considered), making the whole WCET analysis process either *intractable* or highly *imprecise* in practice.

However, it is worthwhile to point that our current implementation does not include the modeling of a few micro-architectural components. For an accurate WCET estimation, the modeling of such micro-architectural components is essential and is a subject of our future implementation. Examples of such micro-architectural components include data caches and branch target buffers (BTB).

The modeling of data caches is usually more complicated than instruction caches. The key to such difficulties arises due to the fact that different instances of the same instruction may access different data memory blocks (*e.g.* array accesses inside a loop, pointer aliasing). Therefore, the modeling of data caches usually involves an address analysis phase (*e.g.* similar to the analysis proposed in [Balakrishnan and Reps 2004]). The output of address analysis is an over-approximation of the set of addresses accessed by each load/store instruction. Using the results of address analysis, the modeling of data caches has been proposed in [Sen and Srikant 2007].

The data cache modeling proposed in [Sen and Srikant 2007] is a *must analysis*. Therefore, each load/store instruction is classified as *all-hit* (AH) or *unclassified* (NC). The extension of the basic data cache modeling for multi-level data caches (as well as for unified caches) has been discussed in [Chattopadhyay and Roychoudhury 2009]. Since the basic technique applied for such data cache modeling is abstract interpretation, the modeling of data caches can easily be integrated into our framework (*e.g.* refer to Equation 1 for integration with pipeline and Figure 5 for integration with branch prediction). Therefore, the integration of such data cache modeling into our framework does not pose any additional challenge. However, a recent approach ([Huynh et al. 2011]) has shown that a data cache modeling based on address analysis (*e.g.* using [Balakrishnan and Reps 2004]) may highly overestimate the WCET. To overcome the imprecision caused due to address analysis, [Huynh et al. 2011] computes the set of loop iterations in which a particular data memory block could be accessed. Such a computation strategy is useful for data accesses, as the data memory blocks accessed in disjoint loop iterations can never conflict with each other in the data cache. In future, we plan to extend our framework with such precise data cache modeling, such as handling shared data caches and cache coherence in the presence of shared data accesses.

Modern embedded processors also employ a branch target buffer (BTB) to cache the target address of a branch. Our current implementation does not include the modeling of BTBs. As shown in [Grund et al. 2011], the modeling of BTBs can be accomplished via abstract interpretation. The BTB analysis proposed in [Grund et al. 2011] is a combined *must* and *may* analysis. Given any branch instruction address, the analysis proposed in [Grund et al. 2011] classifies a branch instruction as t (*i.e.* the branch instruction must be in the BTB), f (*i.e.* the branch instruction must not be in the BTB) or $\top$ (*i.e.* static analysis cannot determine the inclusion of the branch instruction in the BTB). Such a classification is analogous to the classification in the instruction cache analysis. Therefore, given an upper bound on BTB miss penalty (say BTB_{miss}), such a classification can be integrated into our framework using the technique similar to Equation 1. Moreover, the static analysis of BTB content (as proposed in [Grund et al. 2011]) can be used in our framework to determine the speculative instructions and their effects on caches (cf. Figure 5).

We believe that our framework can be useful to evaluate the precision and scalability of different analysis methodologies. In our current framework, ILP-based branch prediction modeling and the shared bus modeling fill up most of the analysis time. Therefore, exploring different techniques for the modeling of branch predictors and shared buses, along with a notion of scalability will be an interesting direction in future.

Acknowledgement

This work was partially funded by A*STAR public sector funding project 1121202007 (R252-000-476-305) from Singapore, by the ArtistDesign Network of Excellence (the European Community's 7th Framework Program FP7/2007-2013 under grant agreement no 216008) and by Deutsche Forschungsgemeinschaft (DFG) under grant FA 1017/1-1.

REFERENCES

T. Austin, E. Larson, and D. Ernst. 2002. SimpleScalar: An Infrastructure for Computer System Modeling. *Computer* 35, 2 (2002).

G. Balakrishnan and T. W. Reps. 2004. Analyzing Memory Accesses in x86 Executables. In *CC*. 5–23.

C. Berg. 2006. PLRU Cache Domino Effects. In *International Workshop on Worst-Case Execution Time (WCET) Analysis*.

S. Chattopadhyay, L. K. Chong, A. Roychoudhury, T. Kelter, P. Marwedel, and H. Falk. 2011. Chronos for multi-cores: a WCET analysis tool for multi-cores. (2011). http://www.comp.nus.edu.sg/~rpembed/chronos/publication/chronos-multi-core.pdf.

S. Chattopadhyay and A. Roychoudhury. 2009. Unified Cache Modeling for WCET Analysis and Layout Optimizations. In *IEEE Real-Time Systems Symposium*.

S. Chattopadhyay and A. Roychoudhury. 2011. Scalable and Precise Refinement of Cache Timing Analysis via Model Checking. In *IEEE Real-Time Systems Symposium*.

S. Chattopadhyay, A. Roychoudhury, and T. Mitra. 2010. Modeling shared cache and bus in multi core platforms for timing analysis. In *International Workshop on Software & Compilers for Embedded Systems*.

K. Goossens and A. Hansson. 2010. The aethereal network on chip after ten years: goals, evolution, lessons, and future. In *Proceedings of the 47th Design Automation Conference*.

D. Grund and J. Reineke. 2009. Abstract Interpretation of FIFO Replacement. In *Static Analysis Symposium*.

D. Grund and J. Reineke. 2010a. Precise and Efficient FIFO-Replacement Analysis Based on Static Phase Detection. In *Euromicro Conference on Real-Time Systems*.

D. Grund and J. Reineke. 2010b. Toward Precise PLRU Cache Analysis. In *International Workshop on Worst-Case Execution Time (WCET) Analysis*.

D. Grund, J. Reineke, and G. Gebhard. 2011. Branch target buffers: WCET analysis framework and timing predictability. *Journal of Systems Architecture - Embedded Systems Design* 57, 6 (2011).

J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper. 2010. The Mälardalen WCET Benchmarks – Past, Present and Future. In *International Workshop on Worst-Case Execution Time (WCET) Analysis*. http://www.mrtc.mdh.se/projects/wcet/benchmarks.html

D. Hardy, T. Piquet, and I. Puaut. 2009. Using Bypass to Tighten WCET Estimates for Multi-Core Processors with Shared Instruction Caches. In *IEEE Real-Time Systems Symposium*.

D. Hardy and I. Puaut. 2011. WCET analysis of instruction cache hierarchies. *Journal of Systems Architecture - Embedded Systems Design* 57, 7 (2011).

B. K. Huynh, L. Ju, and A. Roychoudhury. 2011. Scope-Aware Data Cache Analysis for WCET Estimation. In *IEEE Real-Time and Embedded Technology and Applications Symposium*.

T. Kelter, H. Falk, P. Marwedel, S. Chattopadhyay, and A. Roychoudhury. 2011. Bus aware multicore WCET analysis through TDMA offset bounds. In *Euromicro Conference on Real-Time Systems*.

X. Li, Y. Liang, T. Mitra, and A. Roychoudhury. 2007. Chronos: A Timing Analyzer for Embedded Software. *Science of Computer Programming* (2007). http://www.comp.nus.edu.sg/~rpembed/chronos.

X. Li, T. Mitra, and A. Roychoudhury. 2005. Modeling Control Speculation for Timing Analysis. *Real-Time Systems* 29, 1 (2005).

X. Li, A. Roychoudhury, and T. Mitra. 2006. Modeling out-of-order processors for WCET analysis. *Real-Time Systems* 34, 3 (2006).

Y. Li, V. Suhendra, Y. Liang, T. Mitra, and A. Roychoudhury. 2009. Timing Analysis of Concurrent Programs Running on Shared Cache Multi-Cores. In *IEEE Real-Time Systems Symposium*.

Y-T. S. Li, S. Malik, and A. Wolfe. 1999. Performance estimation of embedded software with instruction cache modeling. *ACM Trans. Des. Autom. Electron. Syst.* 4, 3 (1999).

T. Lundqvist and P. Stenström. 1999. Timing Anomalies in Dynamically Scheduled Microprocessors. In *IEEE Real-Time Systems Symposium*.

M. Lv, G. Nan, W. Yi, and G. Yu. 2010. Combining Abstract Interpretation with Model Checking for Timing Analysis of Multicore Software. In *IEEE Real-Time Systems Symposium*.

C. Maiza and C. Rochange. 2011. A framework for the timing analysis of dynamic branch predictors. In *RTNS*.

M. Paolieri, E. Quiñones, F. J. Cazorla, G. Bernat, and M. Valero. 2009a. Hardware support for WCET analysis of hard real-time multicore systems. In *International Symposium on Computer Architecture*.

M. Paolieri, E. Quiñones, F. J. Cazorla, G. Bernat, and M. Valero. 2009b. Hardware support for WCET analysis of hard real-time multicore systems. In *ISCA*.

R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley. 2011. A Predictable Execution Model for COTS-Based Embedded Systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium*.

J. Reineke, D. Grund, C. Berg, and R. Wilhelm. 2007. Timing predictability of cache replacement policies. *Real-Time Systems* 37, 2 (2007).

C. Rochange and P. Sainrat. 2009. A Context-Parameterized Model for Static Analysis of Execution Times. *T. HiPEAC* 2 (2009).

J. Rosen, A. Andrei, P. Eles, and Z. Peng. 2007. Bus Access Optimization for Predictable Implementation of Real-Time Applications on Multiprocessor Systems-on-Chip. In *IEEE Real-Time Systems Symposium*.

R. Sen and Y. N. Srikant. 2007. WCET estimation for executables in the presence of data caches. In *EMSOFT*.

H. Theiling, C. Ferdinand, and R. Wilhelm. 2000. Fast and Precise WCET Prediction by Separated Cache and Path Analyses. *Real-Time Systems* 18, 2/3 (2000).

R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand. 2009. Memory Hierarchies, Pipelines, and Buses for Future Architectures in Time-Critical Embedded Systems. *IEEE Trans. on CAD of Integrated Circuits and Systems* 28, 7 (2009).

J. Yan and W. Zhang. 2008. WCET Analysis for Multi-Core Processors with Shared L2 Instruction Caches. In *IEEE Real-Time and Embedded Technology and Applications Symposium*.

M. M. Zahran, K. Albayraktaroglu, and M. Franklin. 2007. Non-Inclusion Property in Multi-Level Caches Revisited. *I. J. Comput. Appl.* 14, 2 (2007).