A Unified WCET Analysis Framework for Multi-core Platforms

Sudipta Chattopadhyay, Chong Lee Kee, Abhik Roychoudhury
National University of Singapore
{sudiptac,cleekee,abhik}@comp.nus.edu.sg

Timon Kelter, Peter Marwedel
Technical University of Dortmund
{timon.kelter,peter.marwedel}@tu-dortmund.de

Heiko Falk
Ulm University
Heiko.Falk@uni-ulm.de

Abstract—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.

I. 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 cache and shared bus. 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* [1]. 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 ([2], [3], [4], [5], [6]) 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* architecture 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 [7], a freely available, open-source, single-

core WCET analysis tool. To evaluate our approach, we have also extended a *cycle accurate simulator* [8] with both shared cache and shared bus support. Our experiments with moderate to large size benchmarks from [9] show that we can obtain tight WCET estimates for most of the benchmarks in a wide range of micro-architectural configurations.

II. RELATED WORK

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 [10]. However, the work proposed in [10] faces scalability problems due to the explosion in number of generated ILP constraints. In [11], 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 [12] for modeling advanced micro-architectural features such as out-of-order and superscalar pipelines. A different paper by the same set of authors [13] has proposed an ILP-based modeling of branch predictors. Our baseline framework is built upon the technique proposed in [12], [13].

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 [2], [3], [14]. All of these approaches extend the abstract interpretation based cache analysis proposed in [11]. 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 with different other micro-architectural features (*e.g.* pipeline and branch prediction). On the other hand, separated shared bus analysis has been proposed in [15], [5], [4]. None of these works model the interactions with pipeline and branch prediction. Additionally, [15] and [4] both assume a *timing-anomaly-free* architecture.

A recent approach [6] has combined abstract interpretation and model checking for analyzing private cache and shared bus, respectively. However, it is unclear whether such a combination would remain scalable in the presence of a shared cache and other micro-architectural features.

To eliminate the problem of pessimism in multi-core WCET analysis, researchers have proposed predictable multi-core architectures [16] and predictable execution models by code transformations [17]. However, we argue that these approaches are orthogonal to the idea of this paper and our idea in this paper can be used to pinpoint the source of overestimation in multi-core WCET analysis.

In summary, there has been little progress on multi-core WCET analysis by modeling individual micro-architectural components (*e.g.* shared cache, shared bus). Our work differs
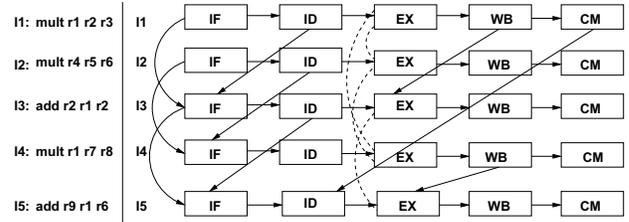

Figure 1. Execution graph for the example program in a 2-way superscalar processor with 2-entry instruction fetch queue and 4-entry reorder buffer. Solid edges show the dependency between pipeline stages, whereas the dotted edges show the contention relation.

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.

III. BACKGROUND

Pipeline modeling through execution graphs: The central idea of pipeline modeling revolves around the concept of the *execution graph* [12]. 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.

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
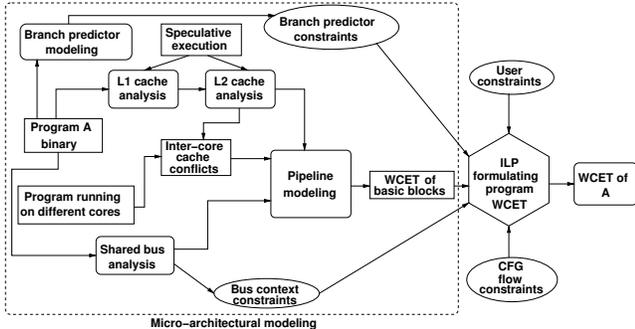
Figure 2. Overview of our analysis framework

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.

IV. 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 [11] 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* [3]. 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 VII for details). Similarly, the timing effects generated by the mispredicted instructions are also taken into account during the iterative pipeline modeling (refer to [12] 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 V for details). Pipeline modeling 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 control flow graph (CFG), micro-architectural modeling (branch predictor and shared bus) and additional user given constraints (*e.g.* loop bound). The modeling of the branch predictor generates constraints to bound the execution count of mispredicted branches (for details refer to [13]). 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 VI). *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 an 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.

V. 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 ready time of node i.
- $earliest[t_i^{start}]$: Earliest start time of node i.
- $earliest[t_i^{finish}]$: Earliest finish time of node i.
- $latest[t_i^{ready}]$: Latest ready time of node i.
- $latest[t_i^{start}]$: Latest start time of node i.
- $latest[t_i^{finish}]$: Latest finish time of node i.

Therefore, the *active* time span of node i can be represented by the interval $[earliest[t_i^{ready}], latest[t_i^{finish}]]$. In the following sections, we shall discuss how the presence of a

shared cache and a shared bus affects the timing information of different pipeline stages.

A. 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:

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

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.

B. 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 last scheduled *round*. Therefore, at time T, the TDMA offset can be precisely defined as $T \bmod S_l\mathcal{C}$.

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}).

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 [4] or [5]):

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

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:

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

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 $j1$. By capturing this property, we can compute O_i^{in} as follows:

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

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.

VI. WCET COMPUTATION OF A BASIC BLOCK

A. *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 [12]. 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 V). 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: $O_i^{in} = \bigcup_{p,e} O_i^{in}(p, e)$ 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 iterations of the loop. The bus contexts at different iterations depend on the set of instructions which can propagate TDMA offsets across loop
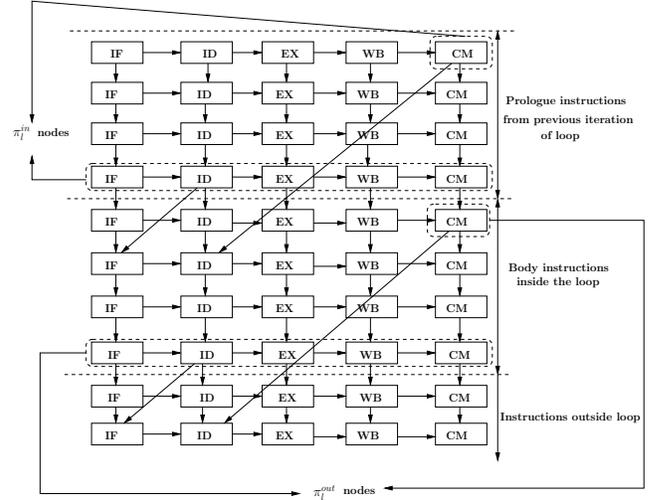

Figure 3. π_l^{in} and π_l^{out} nodes shown with the example of a sample execution graph. π_l^{in} nodes propagate bus contexts across iterations, whereas, π_l^{out} nodes propagate bus contexts outside of loop.

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 3 demonstrates the π_l^{out} and π_l^{in} nodes for a sample execution graph. 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 V). 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.

B. *WCET computation under multiple bus contexts*

Foundation: As discussed in the preceding, a basic block inside some loop may execute under different bus contexts. In this section, we shall show how the execution count of different bus contexts can be bounded by generating additional ILP constraints. These additional ILP constraints are eventually fed to the global ILP formulation. We begin with the following notations to discuss our technique:

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.

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 $f_{w_1 \to w_2} = (w_1, w_2) \in F_l^s$ 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.

$M_l^{w_1 \to w_2}$: 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 $f_{w_1 \to w_2} \notin F_l^s$ for any flow graph G_l^s, $M_l^{w_1 \to w_2} = 0$.

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 V), 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 $s \in \Omega_l \setminus \Omega_l^s$ 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 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.

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:

$$\forall w \in \Omega_l : \sum_{x \in \Omega_l} M_l^{x \to w} = M_l^w \tag{10}$$

$$\forall w \in \Omega_l : M_l^w - 1 \leq \sum_{x \in \Omega_l} M_l^{w \to x} \leq M_l^w \tag{11}$$

$$\sum_{w \in \Omega_l} M_l^w = N_{l.h} \tag{12}$$

where $N_{l.h}$ denotes the number of times the header of loop l is executed. Equations 10-11 generate 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.* $M_l^{w \to x}$). Equation 11 takes this special case into consideration. On the other hand, Equation 12 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 [11]. Equations 10-12 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:

$$\forall w \in \Omega_l^s : M_l^w \geq N_{l.h}^{w.h} \tag{13}$$

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 10-13 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:

$$\forall G_l^s = (V_l^s, E_l^s) : \sum_{w \in V_l^s} M_l^w \leq bound_l * \left(\sum_{w' \in parent(G_l^s)} M_{l'}^{w'} \right) \tag{14}$$

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 14 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 14.

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:

$$\sum_{w \in \Omega_l} N_i^{w.i} = N_i \tag{15}$$

$$\forall w \in \Omega_l : N_i^{w.i} \leq M_l^w \tag{16}$$

6

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 16 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 whole program, we need to estimate the bus context exiting a loop l (say O_l^{exit}). A recently proposed work ([5]) has shown the computation of O_l^{exit} without a full loop unrolling. In this paper, we use a similar technique as in [5] with one important difference: In [5], 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 3). 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 [5], 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 [5].

VII. Effect of branch prediction on Cache

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 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 4. In Figure 4, we demonstrate the procedure for computing the abstract cache state at the entry of a basic block i. Basic block i is
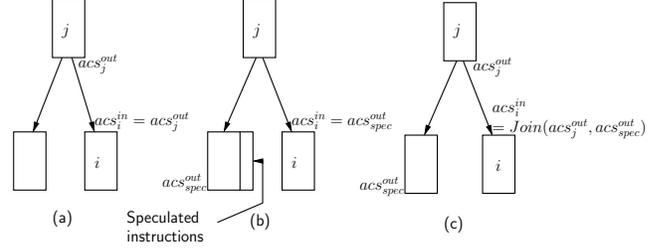

Figure 4. (a) Computation of acs_i^{in} when the edge $j \rightarrow i$ is correctly predicted, (b) Computation of acs_i^{in} when the edge $j \rightarrow i$ is mispredicted, (c) A *safe* approximation of acs_i^{in} by considering both correct and incorrect prediction of edge $j \rightarrow i$.

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 4(a)) and the other when the respective branch is *incorrectly* predicted (Figure 4(b)). The combination is performed through abstract *join* operation, which on the other hand 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.

VIII. WCET computation of an entire program

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

$$Maximize \; T = \sum_{i=1}^{N} \sum_{j \rightarrow i} \sum_{w \in \Omega_i} t_{j \rightarrow i}^{c,w} * E_{j \rightarrow i}^{c,w} + t_{j \rightarrow i}^{m,w} * E_{j \rightarrow i}^{m,w}$$

(17)

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 VI). $t_{j \rightarrow i}^{c,w}$ 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, $t_{j \rightarrow i}^{m,w}$ 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 $t_{j \rightarrow i}^{c,w}$ and $t_{j \rightarrow i}^{m,w}$ are computed during the iterative pipeline modeling (with the modifications proposed in Section V). $E_{j \rightarrow i}^{c,w}$ ($E_{j \rightarrow i}^{m,w}$) 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:

$$E_{j \rightarrow i}^c = \sum_{w \in \Omega_i} E_{j \rightarrow i}^{c,w}, \quad E_{j \rightarrow i}^m = \sum_{w \in \Omega_i} E_{j \rightarrow i}^{m,w} \quad (18)$$

Constraints on $E_{j \rightarrow i}^c$ and $E_{j \rightarrow i}^m$ are proposed by the ILP based formulation in [13]. On the other hand, $E_{j \rightarrow i}^{c,w}$ and $E_{j \rightarrow i}^{m,w}$ are bounded by the CFG structural constraints and the constraints proposed by Equations 10-16 in Section VI.

Finally, the WCET of the program maximizes the objective function in Equation 17. Any ILP solver (*e.g.* CPLEX) can be used for the same purpose.

IX. 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 [18].

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.

X. EXPERIMENTAL EVALUATION

We have chosen moderate to large size benchmarks from [9], which are generally used for timing analysis. Individual benchmarks are compiled into simplescalar PISA (Portable Instruction Set Architecture) [8] — a MIPS like instruction set architecture. The control flow graph (CFG) of each benchmark is extracted from its PISA compliant binary and is used as an input to our analysis framework.

To validate our analysis framework, the simplescalar toolset [8] 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 underapproximate the *actual WCET*.

For all of our experiments, we present the WCET overestimation ratio, which is measured as $\frac{Estimated\ WCET}{Observed\ WCET}$. Our analysis uses the default system configuration in Table I. Since the data cache modeling is not yet included in our current implementation, all data accesses are assumed to be *L1 cache hits*.

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*

Table I
DEFAULT MICRO-ARCHITECTURAL SETTING FOR EXPERIMENTS

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 (Figure 5(a)-(d)), we use jfdctint to generate inter-core conflicts to the first half of the tasks (*i.e.* matmult to nsichneu). On the other hand, we use statemate to generate inter-core conflicts to the second half of the tasks (*i.e.* edn 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
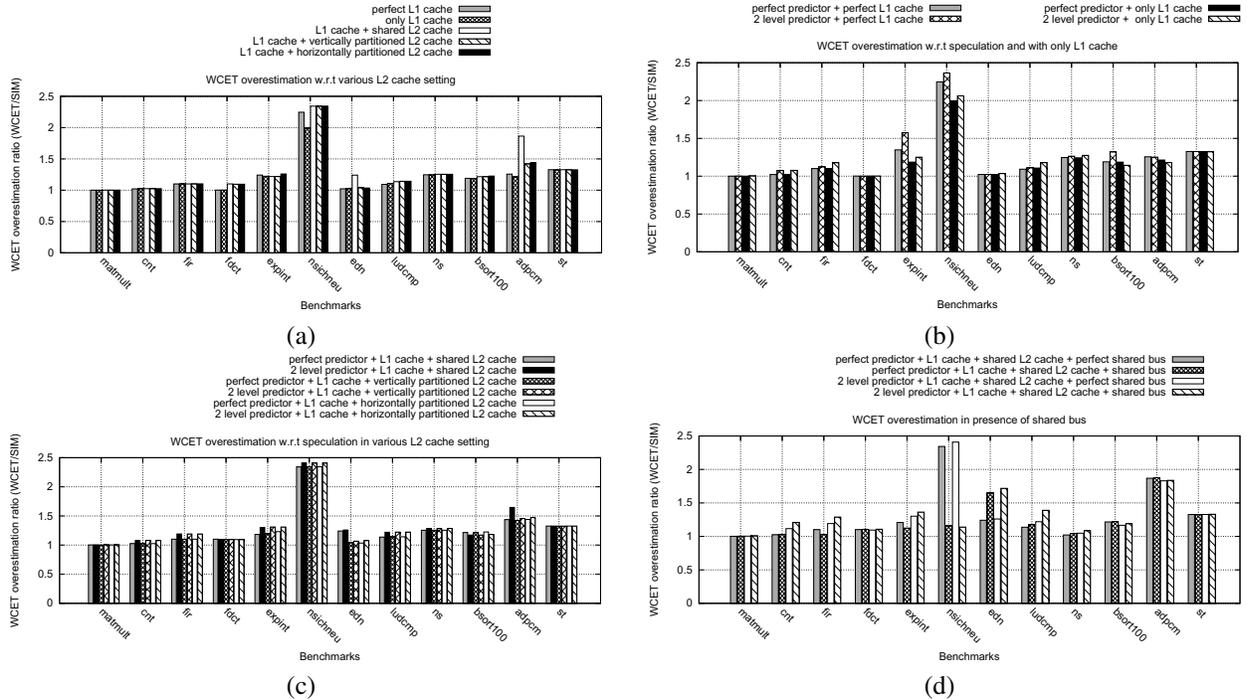
Figure 5. Effect of different micro-architectural parameters in WCET overestimation

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 `nsichneu`) compared to the second half (*i.e.* `edn` 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 number of ways compared to the original shared cache (called *vertical* partitioning). Secondly, each partition has half 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 column "Perfect settings" in Table I).

Effect of caches: Figure 5(a) 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 `nsichneu`. The main source of WCET overestimation in `nsichneu` comes from the *path analysis* and *not* due to the micro-architectural modeling. This is expected, as `nsichneu` contains more than two hundred branch instructions and many *infeasible paths*. These infeasible paths can be eliminated by providing additional user constraints into our framework and hence improving the result. 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 `edn` and `adpcm`, where the overestimation in the presence of a shared cache rises. This is due to the difficulty in modeling the inter-core cache conflicts from `statemate` (a *many-path* program being run in parallel).

Effect of speculative execution: As we explained in Section VII, 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 5(b) and Figure 5(c) show the effect of speculation in L1 and L2 cache, respectively. Mostly, we do not observe any sudden *spikes* in the WCET overestimation just due to speculation. `adpcm` shows some reasonable increase in WCET overestimation with L2 caches and in the presence of speculation (Figure 5(c)). This increase in the overestimation ratio can be explained from the overestimation arising in the modeling of the effect of speculation in cache (refer to Section VII).

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 5(d) 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 `edn` and `nsichneu`, the overestimation in the presence of a shared cache and a shared bus is mostly equal to the overestimation when shared bus analysis is turned off (*i.e.* a *perfect* shared bus). Experiments with `nsichneu` shows some interesting result. We observe that the WCET overestimation ratio decreases by a large factor when shared bus analysis is *enabled*. As we inspect the cause, we found that the execution time of `nsichneu` 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 5(a) 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%.

WCET overestimation sensitivity: We have also conducted detailed experiments to check the WCET overestimation sensitivity with respect to L1 cache size, L2 cache size, bus slot length and different pipelines (inorder, out-of-order and superscalar). On average, our framework results around 40% overestimation and a maximum of 90% overestimation when very small L1 cache is used (512 bytes). For very small L1 cache size, our analysis results in L1 *cache thrashing* which eventually results in an overestimation of shared cache and shared bus traffic. Due to space constraints, details of these results are included in [18].

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 systems. In most of the cases, our analysis finishes within a few seconds. ILP solver time dominates when branch prediction is enabled. When all the micro-architectural features are analyzed (pipeline, L1 and shared L2 cache, shared bus and branch prediction), our analysis takes maximum time (around 300 seconds) for the program `nsichneu`, with an average of 20-30 seconds over all other programs.

XI. CONCLUSION

In this paper, we have proposed a *sound* WCET analysis framework by modeling different micro-architectural components and their interactions in a multi-core processor. Our analysis framework is also *sound* in the presence of *timing anomalies*. Our experiments suggest that we can obtain tight WCET estimates for the majority of benchmarks in a variety of micro-architectural configurations. Apart from design space exploration, we believe that our framework can be used to figure out the major sources of overestimation in multi-core WCET analysis. As a result, our framework can help in designing predictable hardware for real time applications and it can also help writing real time applications for the predictable execution in multi-cores.

REFERENCES

[1] T. Lundqvist and P. Stenström. Timing anomalies in dynamically scheduled microprocessors. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, RTSS, 1999.

[2] J. Yan and W. Zhang. WCET analysis for multi-core processors with shared L2 instruction caches. In *RTAS*, 2008.

[3] Y. Li et. al. Timing analysis of concurrent programs running on shared cache multi-cores. In *RTSS*, 2009.

[4] S. Chattopadhyay, A. Roychoudhury, and T. Mitra. Modeling shared cache and bus in multi core platforms for timing analysis. In *SCOPES*, 2010.

[5] T. Kelter et. al. Bus aware multicore WCET analysis through TDMA offset bounds. In *ECRTS*, 2011.

[6] M. Lv et. al. Combining abstract interpretation with model checking for timing analysis of multicore software. In *RTSS*, 2010.

[7] X. Li et. al. Chronos: A timing analyzer for embedded software. *Science of Computer Programming*, 2007. http://www.comp.nus.edu.sg/~rpembed/chronos.

[8] T. Austin, E. Larson, and D. Ernst. Simplescalar: An infrastructure for computer system modeling. *Computer*, 35(2), 2002.

[9] WCET benchmarks. http://www.mrtc.mdh.se/projects/wcet/benchmarks.html.

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

[11] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise WCET prediction by separated cache and path analyses. *Real-Time Systems*, 18(2/3), 2000.

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

[13] X. Li, T. Mitra, and A. Roychoudhury. Modeling control speculation for timing analysis. *Real-Time Systems*, 29(1), 2005.

[14] D. Hardy, T. Piquet, and I. Puaut. Using bypass to tighten WCET estimates for multi-core processors with shared instruction caches. In *RTSS*, 2009.

[15] J. Rosen et. al. Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip. In *RTSS*, 2007.

[16] M. Paolieri et. al. Hardware support for WCET analysis of hard real-time multicore systems. In *ISCA*, 2009.

[17] R. Pellizzoni et. al. A predictable execution model for COTS-based embedded systems. In *RTAS*, 2011.

[18] S. Chattopadhyay et. al. A unified WCET analysis framework for multi-core platforms. http://www.comp.nus.edu.sg/~rpembed/chronos/publication/chronos-multi-core.pdf.