JCSE Journal of Computing Science and Engineering

Regular Paper

Journal of Computing Science and Engineering,
Vol. 6, No. 4, December 2012, pp. 267-278

Static Timing Analysis of Shared Caches for Multicore Processors

Wei Zhang*
Department of Electrical and Computer Engineering, Virginia Commonwealth University Richmond, VA, USA
wzhang4@vcu.edu

Jun Yan
MathWorks Inc., Boston, MA, USA
Jun.Yan@mathworks.com

Abstract
The state-of-the-art techniques in multicore timing analysis are limited to analyze multicores with shared instruction caches only. This paper proposes a uniform framework to analyze the worst-case performance for both shared instruction caches and data caches in a multicore platform. Our approach is based on a new concept called address flow graph, which can be used to model both instruction and data accesses for timing analysis. Our experiments, as a proof-of-concept study, indicate that the proposed approach can accurately compute the worst-case performance for real-time threads running on a dual-core processor with a shared L2 cache (either to store instructions or data).

Category: Embedded computing

Keywords: Performance; Reliability; WCET analysis; Multicore processors; Unified caches

I. INTRODUCTION

Multicore chips can offer many important advantages such as higher throughput, energy efficiency and density. However, to safely exploit multicore chips for real-time systems, especially hard real-time systems, it is crucial to accurately obtain the worst-case execution time (WCET). This, however, is a very challenging task, due to the enormous complexity caused by inter-thread interferences in accessing shared resources in multicores, such as shared caches.

In the last two decades, WCET analysis has been actively studied, and a good review regarding the state-of-the-art can be found at [1]. Most of the prior research efforts, however, focus on WCET analysis for uniprocessors [2-6], which cannot be used to estimate the WCET for multicore processors due to their inability to estimate the worst-case inter-thread interferences in shared resources such as caches.

There are also some studies on WCET analysis for multi-tasking uniprocessors [7-10], which, however, still cannot be applied to estimate inter-thread cache conflicts on multicore chips. This is because in a multi-tasking uniprocessor system, threads (i.e., tasks) are typically assigned by different priorities and are preemptive. When preemption occurs, the cache memories are taken over by a higher prioritized thread solely and later given back to the lower prioritized thread when the higher prioritized thread finishes. Therefore, the preemption of a thread implicitly causes the preemption of the cache memories simultaneously, making it possible to find such preemption points, at which the higher prioritized threads will produce the

Open Access http://dx.doi.org/10.5626/JCSE.2012.6.4.267 http://jcse.kiise.org

Received 19 April 2012, **Revised** 30 October 2012, **Accepted** 22 November 2012
*Corresponding Author

 pISSN: 1976-4677 eISSN: 2093-8020

worst cache related preemptive delay (CRPD). In a multicore model, however, threads are running in parallel and cannot preempt each other across different cores. Thus, the interferences among threads are solely determined by the timing order information of all the cache accesses from all the threads. So, the worst-case cache related delay for a given thread is determined by a set of interference points caused by the accessing sequences of all the cache accesses and from all the threads. This essential difference makes existing analysis techniques for multitasking uniprocessors [7-10] not applicable to the multicore WCET analysis.

Recently, there have been an increasing number of research efforts on real-time scheduling for multicore platforms [11, 12]. However, all these studies basically assume that the worst-case performance of real-time threads is known a priori. Therefore, it is critical to develop new timing analysis techniques to reasonably estimate the WCET of real-time threads running on multicore processors.

To the best of our knowledge, the state-of-the-art timing analysis techniques for multicore processors studied in recent work [13-15] are unable to estimate the worst-case performance of shared instruction caches of multicores. While these studies [13-15] have made initial contributions towards the timing analysis of multicores, their inability to analyze shared data caches fundamentally limits the applicability of these methods, considering the widespread use of data caches in multicore processors and their significant impact on the execution time, including the WCET. To address this problem, this paper proposes to use an address flow graph (AFG) to conduct a timing analysis, which can be generally applied to not only instruction caches, but also data caches as well as unified caches. Also, to the best of our knowledge, this paper is the first work to extend the implicit path enumeration technique (IPET) [4, 5] from uniprocessors to multicores, and the beauty of our approach is that both the path analysis (of single core) and shared cache analysis are based on IPET, which can implicitly compute the WCET of real-time tasks running on a multicore chip safely and accurately by considering all possible paths of each and all concurrent threads, as well as all possible interactions among them.

II. ASSUMED MULTICORE ARCHITECTURE

In a multicore processor, each core typically has private L1 instruction and data caches. The L2 (and/or L3) caches can be shared or private. While private L2 caches are more time-predictable in the sense that there are no inter-thread L2 cache conflicts, they suffer from other deficiencies. First, each core with a private L2 cache can only exploit separated and limited cache space. Second, separated L2 caches will increase the cache synchronization and coherency cost. Moreover, a shared L2 cache

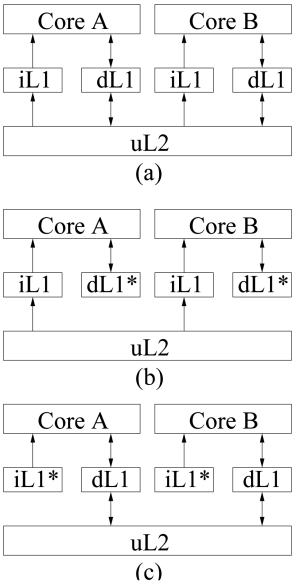


Fig. 1. A dual-core with (a) a shared unified L2 cache, (b) a shared L2 instruction cache (with a perfect L1 data cache), and (c) a shared L2 data cache (with a perfect L1 cache).

architecture makes it easier for multiple cooperative threads to share instructions and data, which become more expensive in separated L2 caches. Therefore, in this paper, we will study the WCET analysis of multicore processors with shared L2 caches (by contrast, the WCET analysis for multicore chips with private L2 caches is a less challenging problem).

In this paper, we focus on examining the WCET analysis for a dual-core processor with a shared L2 cache, although our approach can also be generally applied to multicore processors with a greater number of cores. Fig. 1 shows a processor, where each core has private L1 instruction and data caches, and shares a unified L2 cache. In this work, we focus on analyzing the inter-thread interferences caused by both instruction and data streams. Specifically, we assume the L1 data (instruction) cache in each core is perfect when analyzing the instruction (data) cache, as depicted in Fig. 1b and c, respectively. Due to the common analysis framework of both instruction and data interferences, our approach can also be easily applied to a unified shared cache with both data and instruction accesses, which will be examined in our future work.

III. COMPUTING THE WORST-CASE DELAY OF INTER-THREAD CACHE INTERFERENCES

Cache related delay (CRD) is the delay due to the presence of cache memories, which makes the latency of accessing an instruction or data fluctuate, depending on whether or not the instruction or data can be found from a particular level of cache. Generally speaking, the CRD

problem can be tackled in at least two different ways. One approach is to statically classify each instruction, thus the upper bound latency can be derived for each instruction, which is then integrated with pipeline analysis and path analysis to derive the WCET [2]. Another approach is to use the IPET [4, 5], which is built upon constraint programming to bind the CRD and then leverage integer linear programming (ILP) to compute the CRD and the WCET. This paper adopts IPET to compute WCET. However, the original IPET [4, 5] was proposed for timing analysis of uniprocessors, which cannot analyze multicore processors or caches shared by multiple concurrent threads. In this paper, we propose to use address flow analysis for both instruction and data caches shared in a multicore, which can be easily integrated with IPET to generate constraints for computing the CRD and the WCET for multicores.

A. Constraint Equations for Computing the CRD

Based on IPET [4], WCET analysis for a program can be mathematically defined by using ILP equations and inequalities. Specifically, the WCET can be computed as the maximal value of the following objective function (1).

$$\begin{aligned} WCET &= \sum c_i \times x_i \\ &+ \sum (c_i^{l1\_hit} \times l_i^{l1\_hit} + c_i^{l1\_miss} \times l_i^{l1\_miss}) \\ &+ \sum (c_j^{l2\_hit} \times l_j^{l2\_hit} + c_j^{l2\_miss} \times l_j^{l2\_miss}) \end{aligned} \quad (1)$$

The equation above defines that the WCET is the maximum sum of the cost for each basic block and the cost of the CRD. The cost of each basic block is the scheduled latency (i.e., c_i) multiplied by the number of execution counts (i.e., x_i) of that basic block; and the cost of the CRD is the sum of the hit and miss latency of each cache line block (i.e., $c_i^{l1\_hit}$ and $c_i^{l1\_miss}$, respectively for an L1 cache and $c_j^{l2\_hit}$ and $c_j^{l2\_miss}$, respectively for an L2 cache) multiplied by the number of cache line block hits (i.e., $l_i^{l1\_hit}$ for an L1 cache and $l_j^{l2\_hit}$ for an L2 cache) and misses (i.e., $l_i^{l1\_miss}$ for an L1 cache and $l_j^{l2\_miss}$ for an L2 cache), respectively. The symbols used in this paper are explained in Table 1. More detailed information on IPET can be found at [4].

Equation (2) models the structural constraints of this program, which can be derived from the control flow graph (CFG) of the program.

$$\sum d_{in} = \sum d_{out} = x_i \quad (2)$$

Equation (3) bounds the execution count of a loop header block. Basically, the execution count of a pre-header block timed by the weight of that loop should provide the upper bound for the execution count of this loop header block.

$$x_i^{header} \le loop\_weight \times x_j^{pre\_header} \quad (3)$$

Equation (4) describes the fact that the execution counts of a basic block should be equal to the execution counts of the L1 cache line block holding this basic block.

$$x_i = l_j \quad (4)$$

Equation (5) states that the sum of L1 cache hits and misses for a cache line block j should be equal to the execution counts of this cache line block. This equation also gives the first (but not tight) upper bound of the L1 cache misses of a cache line block, which must be no greater than the execution counts of the cache line block.

$$l_j = l_j^{l1\_hit} + l_j^{l1\_miss} \quad (5)$$

Equation (6) gives a tighter bound of the number of L1 cache misses for a cache line block j. $\sum t^{miss}$ will be detailed in Section III-C.

$$l_j^{l1\_miss} \le \sum t^{miss} \quad (6)$$

Equation (7) is based on the fact that the number of execution counts of a lower level cache line should be equal to the number of execution counts of cache misses from its corresponding upper cache line.

$$\sum l_i^{l1\_miss} = l_p \quad (7)$$

Equation (8) breaks the execution counts of a lower level cache line (i.e., an L2 cache line) into two parts, including the number of cache hits and the number of

Table 1. Symbols used in this paper and their description

Symbol	Description
B_i	Basic block i
L_i	Line block i
c_i	Cost of basic block or line block
x_i	Execution counts of basic block i
l_i	Execution counts of line block i
t_i	State of cache line block i
l1_hit	Level 1 cache hit
l1_miss	Level 1 cache miss
l2_hit	Level 2 cache hit
l2_miss	Level 2 cache miss
l2_intra_miss	Level 2 cache intrinsic cache miss
l2_inter_miss	Level 2 cache extrinsic cache miss
d_{in}	Edge flow in to a basic block
d_{out}	Edge flow out to a basic block

cache misses. Similarly to Equation (5), this equation also gives the first upper bound for the number of misses from L2 cache lines.

$$l_p = l_p^{l2\_hit} + l_p^{l2\_miss} \quad (8)$$

Equation (9) gives another bound regarding the number of misses for a cache line block, in which $\sum t^{miss}$ will be further bounded in our address flow analysis. It should be noted that it is possible for users to provide more data flow analysis constraints to help reduce the number of infeasible paths, which may make the ILP solver derive even tighter WCET analysis results.

$$l_p^{l2\_miss} \leq \sum t^{miss} \quad (9)$$

B. Address Flow Analysis

A program's WCET is determined by the execution time of the code and the CRD caused by loading the instructions and its data to the processor. In our model, thanks to the help of a compiler and a statically-scheduled architecture based on very long instruction word (VLIW) [16], the execution time of each instruction can be statically derived from its static scheduling. However, the CRD is still subject to the dynamic program behavior in terms of the instruction and data memory access patterns and the history of accesses, as well as the actual cache architecture (i.e., cache set associativity, replacement policy, etc.). In a multicore processor with a shared cache, the CRD is also dependent on the memory access behaviors of other concurrent threads that may access the same cache. Motivated by this observation, we start with the address flow analysis and introduce the AFG to unify instruction and data cache timing analysis and to help transform the CRD into an ILP problem that can be automatically solved in a reasonable amount of time.

An AFG, $G = (V, E)$, is defined as a graph that consists of vertices V and edges E. A vertex $v \in V$ is an address related state. Specifically, this state represents the content of the current cache set. This state must satisfy the following requirements: 1) Each vertex is a unique state, which represents the changes caused by the in-flow edges, 2) The out-flow edges of the current state can be used to derive the next state from the current state, and 3) Each vertex is a stable state, which means all the in-flow edges will always produce the current state.

An edge $e \in E$ is a directed line connecting two vertices and the direction indicates the flow direction from one state to another. The change from the source vertex to the destination vertex is caused by the address in the destination vertex. It should be noted that here the address may be an individual memory address of each instruction or data, or an index of a cache line block. A complete AFG should contain all the possible vertices, which are connected by a set of edges.

From the AFG's point of view, the execution of a program can be viewed as running a sequence of addresses loaded from the memory to the processor. This sequence is subject to the control flow of a program to ensure correct semantics. Therefore, the control flow graph can be directly used to represent the AFG for instruction accesses. However, we will analyze the limitation of simply using the CFG to represent the AFG when considering data caches in Section III-B-4, which necessitates the concept of AFG for the timing analysis of unified caches that are common for last-level caches.

In the rest of this section, we will begin with the AFG based analysis for a simple direct-mapped instruction cache and then extend to more complicated cache structures.

1) AFG for Direct-Mapped Instruction Caches

We assume a virtually-addressed cache. After compilation and linking, the virtual address of each instruction is known. Given a cache, each instruction can be mapped to a particular cache line for a direct-mapped cache or to a particular cache set for a set-associative cache in general. Assume that we have a CFG as shown in Fig. 2a, which has 4 instructions, *a*, *b*, *c*, *d*, and all these instructions are mapped to the same cache line *l*. Thus, the AFG of this cache line *l* is shown in Fig. 2b. As can be seen from this graph, starting at the entry point of this program, the first state of this cache line is the access from instruction *a*, so a vertex containing *a* is constructed. Then following the *out_edges* of instruction *a*, the next state of cache line *l* can be either *b* or *c*; therefore, two vertices are added, i.e., *b* and *c*. There are two edges going out from *a* to *b* and *c*, respectively; then either *b* or *c* leads to the access of instruction *d*, which is another vertex added into the AFG.

From this example, it can be seen that in a direct-mapped cache, the state is solely determined by the instructions accessed. Thus, each address (or instruction) forms a vertex in the AFG.

2) AFG for Set-Associative Instruction Caches

Now let us consider a more complicated and generic scenario, i.e., a set-associative cache. The contents of a set-associative cache consist of the history of cache

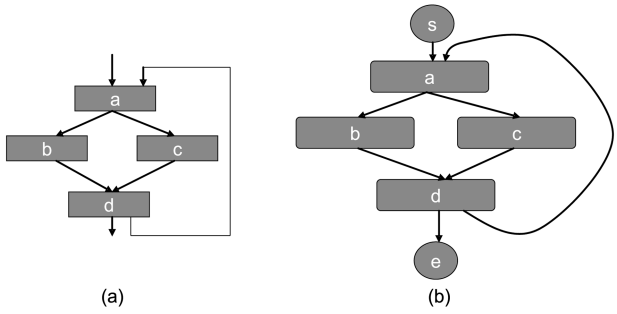


Fig. 2. An example of (a) control flow graph (CFG) and (b) address flow graph (AFG) for a direct-mapped instruction cache.

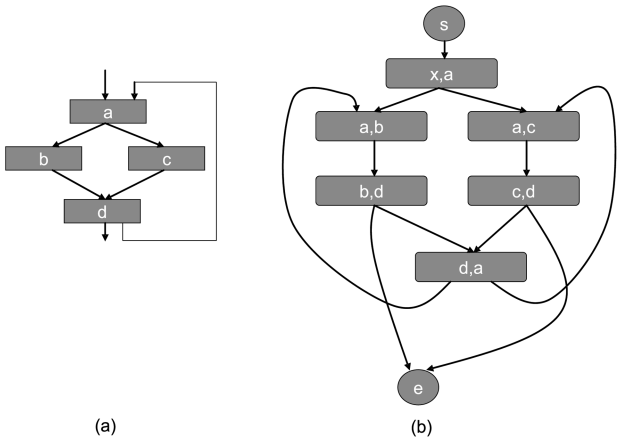


Fig. 3. An example of (a) control flow graph (CFG) and (b) address flow graph (AFG) for a 2-way associative instruction cache.

accesses and the current access. Therefore, for an AFG to correctly represent the address flow of a set-associative cache, the vertex in the AFG must be able to:

- Memorize the history cache accesses;
- Represent the change caused by the current cache access;
- Derive the next possible cache access without violating program semantics.

This can be achieved by using a tuple, called a cache line state (or a cache set state), $c(x_1, x_2,\ldots, x_n)$, where x_i represents the address stored in cache way i, and n is the number of ways for the set-associative cache. Each update can be done based on the cache replacement scheme (e.g., least recently used [LRU]). The most recently used entry can be used to derive the next cache state. We assume that x_1 is the least recently used access.

In Fig. 3a, the same assumption is made as the case of a direct-mapped cache, except that the cache is now a 2-way set-associative. Assume that the initial state of the cache set l is unknown x. Thus, the first cache set state is (x, a). Based on the control flow graph of this program, from the most recent access a, two new states can be constructed, i.e., (a, b) and (a, c). The next access can be either from b to d, or from c to d, thus two new states, i.e., (b, d) and (c, d) are added. The whole AFG for this 2-way set-associative instruction cache is shown in Fig. 3b.

3) AFG for Instruction Caches Shared by Multiple Threads

A more complicated scenario involves two or more threads running simultaneously on a multicore access to a shared cache such as an L2 cache. Then, the abovementioned cache state $c(x_1, x_2,\ldots, x_n)$ is incapable of constructing the address flow graph without losing semantics. The reason is that x_i, *where* $1 \le i \le n$ can store only the cache state of a single thread, while other threads can also access this shared cache. For example, assume that we have two threads T_1 and T_2 and the current cache state c_{curr} contains only the cache access information from T_1, then starting from this point, we lose the capability to derive the correct next state c_{next} since c_{curr} has no information to guide us for deriving the next possible access from T_2.

Motivated by this, assuming we have m threads running concurrently on m cores with a shared cache, we append another tuple $r(s_1, s_2, \ldots, s_m)$ to each vertex of an AFG for a shared cache, which records the most recent access from each thread T_i, where $1 \le i \le m$. Combining these two tuples, i.e., r and c, an AFG can always be constructed for m threads with a *n-way* associative cache by using a set of tuples with $m + n$ entries.

4) AFG for Data Caches

Now, let us take data caches into consideration. In contrast to instructions in a program, data cache WCET analysis typically has difficulty in acquiring the address information for each load/store instruction. Furthermore, what makes it even worse is that even if the data address of each load/store is known, a 1-to-n mapping of this relationship makes a CFG alone useless for data cache analysis. This is because if a load/store instruction is in a loop, although there is only one load/store instruction, it may actually load different data with various memory addresses at different loop iterations, which unfortunately cannot be represented by CFGs. Thus, to tackle the data cache WCET analysis, two issues must be solved: 1) to determine the data address of each instance of a load/store operation, and 2) to construct an AFG to represent all the possible data addresses used by load/store operations.

To determine the data address for general-purpose applications such as programs with pointers and dynamic memory allocation is a very challenging task. Fortunately, hard real-time applications typically restrict the use of those features such as dynamic memory allocation; therefore, it is not uncommon to assume that the data access addresses are statically known for timing analysis in the domain of hard real-time systems, which is also the assumption used in this paper (it should be noted that the focus of this paper is not to tackle how to statically generate addresses for data cache timing analysis, which is orthogonal to our research. Instead, a contribution of this paper is to use AFGs to represent the data and/or instruction addresses, based on which our method can derive the WCET for multicore processors). In addition, we assume for programs with loops, the maximum number of loop iterations is known, which can be either obtained by static analysis or annotation.

Based on the assumption that the data addresses for each load/store and the correct access sequences can be known statically, we propose to expand load/store operations to create the AFG. The idea is to unroll the loop so that the data address accessed by each instance of a load/store operation in each loop iteration is represented as an individual vertex in an AFG. For nested loops, our approach starts from the innermost loop to conduct load/store

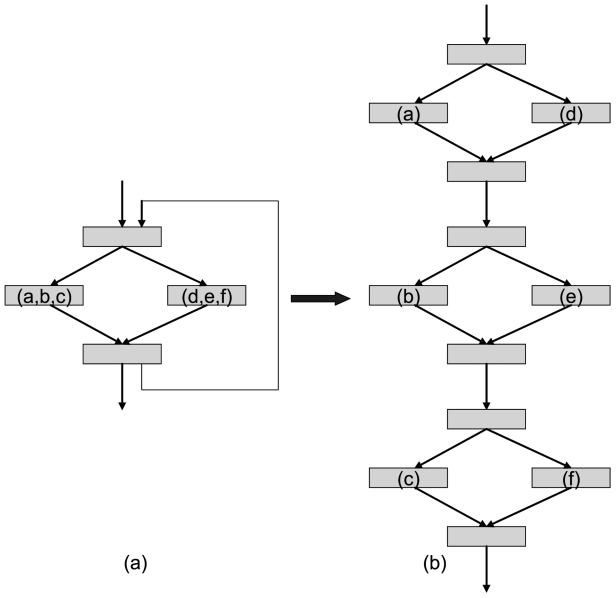


Fig. 4. An example of load/store expansion (a) control flow graph (CFG) and (b) address flow graph (AFG) after load/store expansion.

expansion. Each expansion of the loop needs to remove the back-edge of the expanded loop, while maintaining all the other edges to keep the control flow information. After load/store expansion, each data access can be uniquely represented by an address; therefore, the AFG for a data cache can be constructed in the same way as that for an instruction cache. It should be noted that currently we focus on loop index based accesses; however, the approach can also be extended to any accesses sequence as long as the correct address sequence can be statically obtained. Moreover, the AFG for set-associative data caches and shared data caches can be similarly constructed by using the same principles for instruction caches as discussed in Section III-B-2 and Section III-B-3.

An example is shown in Fig. 4 to illustrate our approach. For simplicity, suppose that we have a loop with three iterations and two load instructions. One load instruction accesses addresses *a*, *b*, *c*, and the other one accesses *d*, *e*, *f*. At the first iteration of the loop, either *a* or *d* is accessed; and at the second iteration, either *b* or *e* is accessed; and at the last iteration, either *c* or *f* is accessed. Then the expansion of the CFG to the AFG is shown in Fig. 4b.

It is worthy to note that the instruction accesses and data accesses are treated in the same way in an AFG, for which only the addresses of instruction or data accesses matter. Therefore, our discussion on the AFG for set-associative caches in Section III-B-2 and AFG for shared instruction caches in Section III-B-3 can be directly applied to the data accesses. However, a major difference between a data access and an instruction access is that inside a loop, the same instruction and thus the same address is accessed for different loop iterations; while for data accesses, the same instruction at different loop iterations may access different memory addresses. This problem can be solved by using the load/store expansion as mentioned above. Therefore, after building the AFT after using load/store expansion, the analysis for the data cache or shared data cache is not different from the instruction cache or the shared instruction cache as previously discussed.

C. Implementation of the AFG

Section III-B discusses the definition of an AFG and its application in different cache structures. In this section we propose to use a combined cache conflict graph (CCCG) to realize the AFG for our assumed architecture, which can handle instruction as well as data caches, direct-mapped as well as set-associative caches, inter-thread shared cache and multi-level cache hierarchies.

The CCCG can be built upon the cache conflict graph (CCG) [4, 5] of each concurrent thread. The CCG was first proposed by Li and Malik [4] and Li et al. [5] to bound worst-case instruction and/or data cache misses for single-core processors. A CCG is basically a projection of a control flow graph for a given thread on a cache set, which contains a set of nodes and edges. In a CCG, each node corresponds to a cache set, and each edge represents a legal path in the CFG between two nodes. A CCG is constructed for every cache set containing two or more conflicting memory object accesses (i.e., instructions or data addresses mapped to the same cache set). A CCG contains a start node 's', an end note 'e', and a node '$B_{k,l}$' for every memory object '$B_{k,l}$' mapped to the same cache set. The start node represents the start of the program, and the end node represents the end of the program. For every node '$B_{k,l}$', a directed edge is drawn from node '$B_{k,l}$' to node '$B_{m,n}$' if there exists a path in the CFG from basic block B_k to basic block B_m without passing through the basic blocks of any other memory object of the same cache set. If there is a path from the start of the CFG to basic block B_k without going through the basic blocks of any other memory object of the same cache line, then a directed edge is drawn from the start node to node '$B_{k,l}$'. The edges between nodes and the end node are constructed analogously. More details about the CCG can be found in [4, 5].

In a CCCG, each vertex is denoted by a tuple $t(r:c)$, which is a combination of tuple r and c. The left side of this tuple records the last access from each thread; and the right side of the tuple represents the current cache line state.

1) Build the CCCG

For illustration purposes, we assume that we have 2 threads (i.e., T_1 and T_2) running on a dual-core with a shared 2-way set-associative cache. Thus, the tuple t in our CCCG approach can be initialized as (S_1, S_2 : x, x),

Algorithm 1 The construction of a CCCG

```
1: BEGIN
2: INPUT t, ccg, cfg
3: OUTPUT cccg
4: for all t.s_i in t do
5:    get an out_edge from t.s_i
6:    for all e in out_edge do
7:       destination address dest ← e.dest
8:       inherit cache state from t, c(x_1, x_2) ← (t.x_1, t.x_2)
9:       if dest is a miss in a higher-level cache then
10:         update the new cache state c(x_1, x_2)
11:      end if
12:      inherit next state t_next from t
13:      update the last access of the i^th thread, t_next.s_i ← dest
14:      update the cache state of t_next to c(x_1, x_2)
15:      add an edge from current t to t_next
16:      append the t_next to the state list
17:      recursively call the algorithm with t_next as the new input.
18:   end for
19: end for
20: terminate if current last access has no out_edge
21: END
```

where S_1 represents the starting node in the CCG of the first thread T_1, S_2 denotes the starting node in the CCG of the second thread T_2, and x means the state for the current cache set is unknown. The next transition t_{next} can be derived by examining the left side of the tuple of the current vertex t_{curr}. If the current access from either T_1 or T_2 has any *out_edge* in its CCG leading to the access of the next cache line block, then our approach updates t_{next} to represent the correct next cache set state. It should be mentioned that for an L1 cache, the right side records the most recent access to the most recent cache line. However, for an L2 cache, the update of the t_{next} must check its upper-level cache state. If the next address transition leads to an L1 hit, then the current L2 t_{next} will simply inherit the same state from t_{curr}.

Algorithm 1 takes the initialized state t and starts walking from the thread s_1. If the last access of thread s_1 has an *out_edge* leading to the next address access, then the next cache set state is generated. The new cache set state initially inherits from t. To update the new cache set state, the algorithm first checks whether the upper-level cache leads to a cache miss or not. If a cache miss occurs at the upper-level cache, then a new cache state is updated based on the current level cache configuration. If a cache hit occurs at the upper-level cache, then the new cache state remains intact as the cache state in t. The algorithm then uses t_{next} to derive a new state. The termination condition of the algorithm is that the final access from all threads has no *out_edge* to go, which indicates that the last access has reached the end of the CCG in each cache line.

2) Bounding Cache Hits

The sum of all the entry edges should equal to 1, as shown in Equation (10). An entry edge is an edge in a CCCG that starts the entry block in a CCG. Specifically, this edge has a source state containing the start entry and its destination state has an entry other than the start entry.

$$\sum e_{entry} = 1 \tag{10}$$

Each CCCG is subject to its own structural constraints as shown in Equation (11). The sum of in-flow edges equals the sum of out-flow edges, which equals the number of execution counts for the current state. However, in a CCCG, each cache line block may sit in a different t due to different paths leading to this cache line block. Equation (12) describes this relationship, where t_i represents the possible state of a line block in a CCCG, and the sum of all the possible states equals the execution counts l_k of the line block. Another scenario is that when a cache line block is in a loop, the state t in the CCCG is also in a loop. In this case there must be a constraint to bound the e_{in} and the execution counts of state t. Inequality 13 shows that the execution count of t must be less than the product of the loop execution counts and the sum of in-flow edges e_{in}, where e_{in} are those none back-edges of the current loop. Since the number of loop execution counts is known, $t_i^{loop\_weight}$ in Inequality (13) is a constant.

$$\sum d_{in} = \sum d_{out} = t_i \tag{11}$$

$$\sum t_i = l_k \tag{12}$$

$$t_i \le t_i^{loop\_weight} \times \sum e_{in} \tag{13}$$

The cache hit bound is calculated by Inequality (14), where e_k is an edge leading to a cache hit.

$$t_i^{hit} \ge \sum e_k \tag{14}$$

Equation (15) links the bounded cache hits and misses to the total number of execution counts of the state t. Now all the variables in our constraint equations are bounded.

$$t_i = t_i^{hit} + t_i^{miss} \tag{15}$$

D. Put Them Together

The constraints from Equations and Inequalities (2)-(15), in conjunction with the objective function (1) can be solved by using an lLP solver. The result of the objective function will be the WCET

IV. AN EXAMPLE OF USING CCCGS

Fig. 5 gives an example to illustrate how to apply the CCCG to WCET analysis. For simplicity, assume there are two threads, a real-time thread (RT) and a non-real-

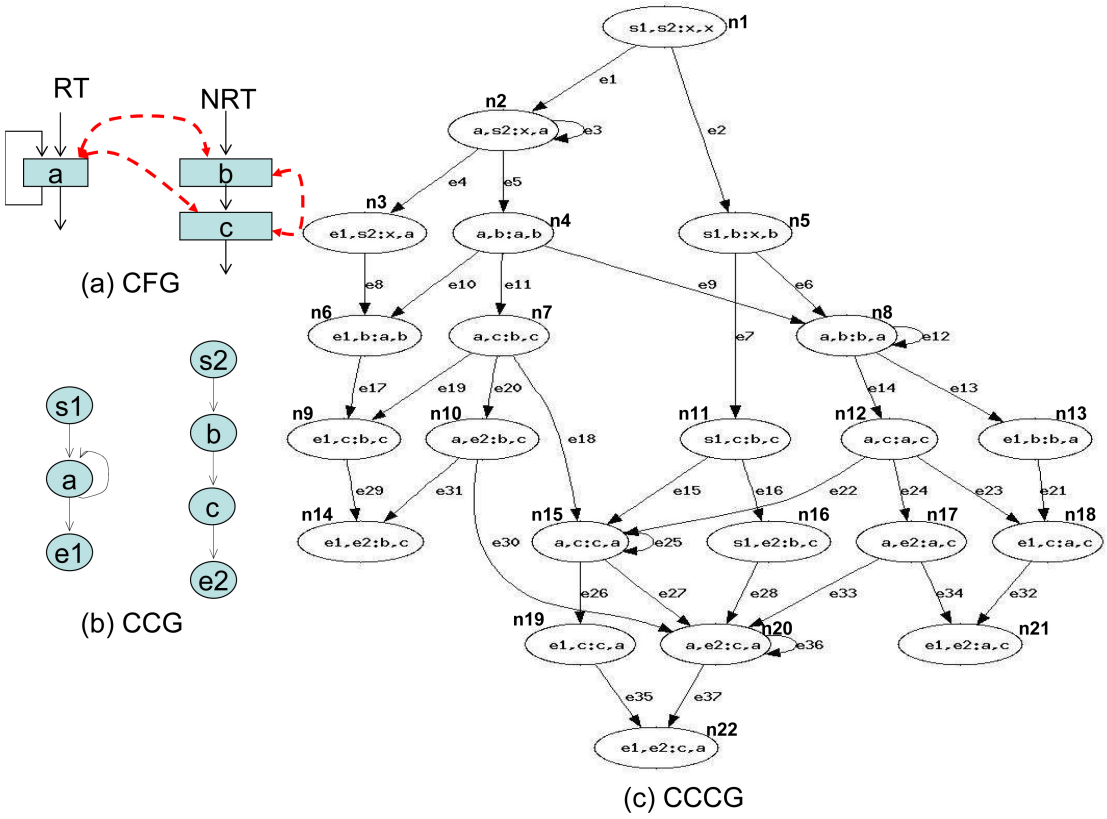


Fig. 5. An example of applying combined cache conflict graphs (CCCGs) to estimate the worst-case execution time. CFG: control flow graph, CCG: cache conflict graph, RT: real-time thread, NRT: non-real-time thread.

time thread (NRT) (It should be noted that the CCCG can be applied to multiple RTs as well). While the RT has only one instruction *a*, the NRT has two instructions, *b* and *c*. Assume the cache is a 2-way set-associative with one set, thus all *a*, *b*, and *c* will be mapped to the same cache line. Also, we assume the cache hit latency is 1 cycle and the cache miss latency is 100 cycles. Fig. 5a shows the CFGs for the RT and NRT; Fig. 5b depicts the CCGs for the RT and NRT; and Fig. 5c draws the CCCG, which is constructed automatically by applying Algorithm 1 to the CFGs and CCGs of both threads.

A. Objective Function

The following objective function below is derived from Equation (1).

$$WCET = 100 \times a^{miss} + a^{hit} \tag{16}$$

B. Structural Constraints

Structural constraints are derived from Equation (2). First, we construct the structural constraints for the RT. In the following equations, *d* represents an edge, and *ds_a* means that the edge starts from the beginning point (i.e., *s*) to an instruction *a*. Similarly, *da_e* means that the edge starts from the instruction *a* to the end of the program (i.e., *e*).

$$a - ds\_a - da\_a = 0 \tag{17}$$

$$a - da\_a - da\_e = 0 \tag{18}$$

Also, we can construct structural constraints for the NRT as the follows.

$$b - ds\_b = 0 \tag{19}$$

$$b - db\_c = 0 \tag{20}$$

$$c - db\_c = 0 \tag{21}$$

$$c - dc\_e = 0 \tag{22}$$

C. Functionality Constraints

For each thread, we assume that each of them executes only once.

$$ds\_a = 1 \tag{23}$$

$$ds\_b = 1 \tag{24}$$

We also assume that the loop in the RT thread executes no more than 10 iterations, which is based on Equation (3).

$$a - 10ds\_a <= 0 \quad (25)$$

D. Cache Constraints

Cache constraints are obtained from Fig. 5c.

1) Connection Constraints

The following equation describes the constraints, i.e., the sum of instruction *a*'s different states equals the execution counts of instruction *a*. This is derived from Equation (5).

$$a - a^{miss} - a^{hit} = 0 \quad (26)$$

2) CCCG Entry Edge Constraints

The RT thread executes only once; thus we have the following equation, which is based on Equation (10).

$$e1 + e6 + e15 + e28 = 1 \quad (27)$$

Similarly, the NRT thread also executes only once, leading to the following equation.

$$e2 + e5 + e8 = 1 \quad (28)$$

3) CCCG Node Constraints

The sum of all the nodes that execute instruction *a* must be equal to the execution counts of instruction *a*, thus we have the following equation, which is derived from Equation (12).

$$n2 + n8 + n15 + n20 - a = 0 \quad (29)$$

Similarly, the sum of all the nodes that execute instruction *b* also must be equal to the execution counts of instruction *b*; hence we can derive the following equation.

$$n4 + n5 + n6 - b = 0 \quad (30)$$

Also, the sum of all the nodes that execute instruction *c* must be equal to the execution counts of instruction *c*, so we have the following equation.

$$n7 + n9 + n11 + n12 + n18 - c = 0 \quad (31)$$

4) Hit Edge Constraints

The following equations are derived from Equation (13).

$$n2 - 10e1 <= 0 \quad (32)$$

$$n8 - 10e9 - 10e6 <= 0 \quad (33)$$

$$n15 - 10e18 - 10e15 - 10e22 <= 0 \quad (34)$$

$$n20 - 10e30 - 10e27 - 10e28 - 10e33 <= 0 \quad (35)$$

5) CCCG Hit Bound

The total cache hits of instruction *a* should equal the sum of all the edges that lead to a possible cache hit for instruction *a*. Thus, we can derive the following equation based on Equation (14) to estimate the number of cache hits.

$$a^{hit} - e3 - e9 - e12 - e22 - e25 - e27 - e33 - e36 >= 0 \quad (36)$$

6) Put Them All Together

Fig. 6 shows the WCET path for the example. The final result from ILP (i.e., the WCET) is 208, which can be derived from Equation (37).

$$100 \times a^{miss} + a^{hit} = 100 \times 2 + 8 = 208 \quad (37)$$

V. EVALUATION METHODOLOGY

It should be noted that in this paper, we assume a time-predictable bus and memory system [17], so that we can focus on studying the timing analysis of the shared caches for multicore processors. In our experiments, we use a cycle-accurate simulator based on SuperESCalar (SESC) simulator to simulate a dual-core processor with either a shared L2 instruction or data cache. The memory hierarchy of the dual-core processor is specified in Table 2. The LP analyzer is implemented by incorporating a commercial ILP solver, CPLEX [18] to handle the linear programming analysis, which generates the worst-case number of misses for both the L1 instruction and data caches and the L2 cache, as well as the WCET.

In the experiments, we chose 10 real-time benchmarks from Mälardalen WCET benchmarks [19], and 2 mediabench applications from Mediabench [20]. Then, each benchmark runs concurrently on the dual-core processor with a benchmark called `crc`, which is randomly selected from Mälardalen WCET benchmark suite [19].

VI. EXPERIMENTAL RESULTS

A. Instruction Cache Timing Analysis Results

To evaluate the effectiveness of the proposed approach,

Table 2. Cache configuration of the base dual-core chip

	Size	Bsize	Assoc	Latency
L1-i(d)-cache	512	16	1	1
L1-d(i)-cache	Perfect			
L2-cache	2 k	32	2	10
Memory	Unlimited			100

we first study the timing analysis for a dual-core with a shared instruction cache. Table 3 compares the estimated and observed (through simulation) number of L1 misses, number of L2 misses, and execution cycles. The last column gives the ratio of the estimated WCET by our approach to the observed WCET through simulation. As can be seen in Table 3, for a number of benchmarks such as `adpcm` and `jfdctint`, the proposed approach can obtain a very tight upper bound of execution cycles, which are within 1% of the observed WCET. On average, the estimated WCET of our approach is 10.1% more than the observed WCET through simulation, which is very accurate considering all possible inter-core cache interferences. Also, we observe that our approach can accurately estimate the worst-case number of L1 and L2 cache misses, which are not too far away from the observed worst-case results.

However, it should be noted that although our approach can achieve a tight bound for most of the benchmarks, the worst-case performance of some benchmarks is still overestimated. In particular, we notice that the estimated WCETs of `fft`, `qsort-exam` are 1.412, 1.266 times more than the simulated results, respectively. One of the major reasons for this overestimation is the intensive `if-then-else` used in the loops in these programs. In both programs, `if-then-else` in loops typically make the programs execute one of the branches (e.g., a then branch) consecutively and then switch to the other branch (e.g., a `else` branch) stay that way for later iterations. This will certainly improve the cache hit ratio since the instructions in either branch are consecutively placed in the cache memory, which can be reused and thus lead to more cache hits. However, in the ILP's semantics, the execution of an `if-then-else` in a loop is most likely interpreted alternatively (e.g., executing one path and then the other path without repeating each path more than once). This is because such an execution on alternative paths can result in a theoretical worst case, which however may not happen at runtime. It is worthy to note that providing further information such as data flow or infeasible paths can help to reduce the overestimation if there is any.

B. Data Cache Timing Analysis Results

As a proof-of-concept study to verify the effectiveness of the proposed load/store expansion approach, we choose six benchmarks whose data access addresses can be statically computed from Mälardalen WCET benchmarks [19]. Table 4 compares the estimated and simulated number of L1 misses, number of L2 misses, and execution cycles for the shared data cache of the simulated dual-core. The last column provides the ratio of the estimated WCET to the observed WCET through simulation. As can be seen in Table 4, the proposed approach can precisely obtain the theoretical WCET on four benchmarks *matmul*, *lms*, *sqrt*, and *wave*. The reason is that all of these four benchmarks actually only have a single path. For the other two benchmarks, *ifthen* and *fir*, which have data accesses from different paths, the overestimation is 7.8% and 5.3%, respectively, indicating that our method for data cache timing analysis by considering inter-core interferences is also very tight.

VII. CONCLUSIONS

This paper presents a novel and unified approach to bounding the worst-case performance of shared instruc-

Table 3. Comparing estimated and simlated worst-case L1, L2 misses and execution cycles for the shared instruction cache

	Estimated results			Observed results (simulation)			
RT	L1 miss	L2 miss	Cycle	L1 miss	L2 miss	Cycle	WCET/Simu ratio
adpcm	657068	515910	65812382	656747	512968	65480672	1.005
bs	21	21	20227	19	19	20009	1.011
crc	1655	765	149284	1643	692	137432	1.086
fft	830	469	59591	626	326	42201	1.412
jfdctint	1431	676	95417	1431	671	94917	1.005
ludcmp	219	153	595400	218	153	573533	1.038
minver	410	218	38172	394	213	36732	1.039
ndes	10791	5690	737603	9872	5167	674225	1.094
qsort-exam	1359	555	80543	1016	453	63602	1.266
qurt	97	72	11007	95	69	10666	1.032
rawcaudio	3772	1687	9950552	3758	1635	8767239	1.135
cordic	1870075	1532070	176648017	1735026	1400280	161822816	1.092
Average							1.101

RT: real-time thread, WCET: worst-case execution time.

Table 4. Comparing estimated and simlated worst-case L1, L2 misses and execution cycles for the shared data cache

	Estimated results			Observed results (simulation)			
RT	L1 miss	L2 miss	Cycle	L1 miss	L2 miss	Cycle	WCET/Simu ratio
matmul	188	188	28214	188	188	28214	1
lms	229	219	30454	229	219	30454	1
sqrt	459	417	73087	459	417	73087	1
wave	483	471	71586	483	471	71586	1
ifthen	166	150	42746	140	137	39656	1.078
fir	726	467	84932	537	456	80648	1.053
Average							1.022

RT: real-time thread, WCET: worst-case execution time.

tion and data caches for multicore processors. While traditional control flow graph based analysis is useful for instruction cache analysis, it is not effective for analyzing data caches, which are important components of multicore chips. To address this problem, we propose to use address flow graphs to model all the possible inter-thread cache conflicts, including both instruction and data accesses, based on which our method can accurately calculate the worst-case inter-thread cache interferences and derive the WCET. We also describe a general method to use combined cache conflict graphs to automatically implement AFGs for set-associative caches shared by multiple threads.

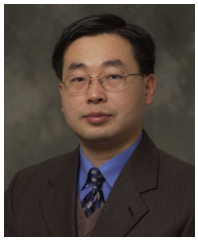
Our experiments indicate that the proposed approach can accurately compute the worst-case performance for real-time threads running on a dual-core processor with a shared L2 cache (either to store instructions or data). Compared to the observed WCET, the estimated WCET is 10.1% larger on average, and can be as low as 1% or less for a number of benchmarks for shared instruction cache analysis. For shared data cache analysis, the estimated WCET is within 2.2% on average compared to the observed WCET by simulation.

In our future work, we would like to conduct unified cache timing analysis on large benchmarks and more cores. Also, we are working on reducing the number of states modeled by CCCGs without affecting the safety and accuracy of analysis. For example, numerous impossible states can be removed by firstly eliminating the infeasible paths on all the concurrent tasks.

REFERENCES

1. J. M. Calandrino, J. H. Anderson, and D. P. Baumberger, "A hybrid real-time scheduling approach for large-scale multicore platforms," *Proceedings of the 19th Euromicro Conference on Real-Time Systems*, Pisa, Italy, 2007, pp. 247-258.
2. C. A. Healy, D. B. Whalley, and M. G. Harmon, "Integrating the timing analysis of pipelining and instruction caching," *Proceedings of the 16th IEEE Real-Time Systems Symposium*, Pisa, Italy, 1995, pp. 288-297.
3. F. Stappert, A. Ermedahl, and J. Engblom, "Efficient longest executable path search for programs with complex flows and pipeline effects," *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, Atlanta, GA, 2001, pp. 132-140.
4. Y. T. S. Li and S. Malik, "Performance analysis of embedded software using implicit path enumeration," *Proceedings of the 32nd Conference on Design Automation*, San Francisco, CA, 1995, pp. 456-461.
5. Y. T. S. Li, S. Malik, and A. Wolfe, "Cache modeling for real-time software: beyond direct mapped instruction caches," *Proceedings of the 17th IEEE Real-Time Systems Symposium*, Washington, DC, 1996, p. 254.
6. G. Ottosson and M. Sjodin, "Worst case execution time analysis for modern hardware architectures," *Proceedings of the ACM SIGPLAN Workshop on Language, Compilers, and Tools for Real-Time Systems*, Las Vegas, NV, 1997.
7. R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, and D. Whalley, et al., "The worst-case execution-time problem: overview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems*, vol. 7, no. 3, article no. 36, 2008.
8. J. H. Anderson, J. M. Calandrino, and U. C. Devi, "Real-time scheduling on multicore platforms," *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, San Jose, CA, 2006, pp. 179-190.
9. V. Kathail, M. S. Schlansker, and B. R. Rau, "HPL-PD architecture specification: version 1.1," Compiler and Architecture Research, HP Laboratories, Palo Alto, CA, Technical Report HPL-93-80(R.1), 2000.
10. IBM ILOG CPLEX optimizer, http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/.
11. J. Yan and W. Zhang, "WCET analysis for multi-core processors with shared L2 instruction caches," *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium*, St. Louis, MO, 2008, pp. 80-89.
12. W. Zhang and J. Yan, "Accurately estimating worst-case execution time for multi-core processors with shared direct-mapped instruction caches," *Proceedings of the 15th IEEE International Conference on Embedded and Real-Time Com-*

puting Systems and Applications, Beijing, China, 2009, pp. 455-463.
13. Y. Li, V. Suhendra, Y. Liang, T. Mitra, and A. Roychoudhury, "Timing analysis of concurrent programs running on shared cache multi-cores," *Proceedings of the 30th IEEE Real-Time Systems Symposium*, Washington, DC, 2009, pp. 57-67.
14. The Malardalen Real-Time Research Centre, WCET benchmark, http://www.mrtc.mdh.se/projects/wcet/benchmarks.html.
15. C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: a tool for evaluating and synthesizing multimedia and communications systems," *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture*, Research Triangle Park, NC, 1997, pp. 330-335.
16. C. G. Lee, H. Hahn, Y. M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim, "Analysis of cache-related preemption delay in fixed-priority preemptive scheduling," *IEEE Transactions on Computers*, vol. 47, no. 6, pp. 700-713, 1998.
17. C. G. Lee, K. Lee, J. Hahn, Y. M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim, "Bounding cache-related preemption delay for real-time systems," *IEEE Transactions on Software Engineering*, vol. 27, no. 9, pp. 805-826, 2001.
18. H. S. Negi, T. Mitra, and A. Roychoudhury, "Accurate estimation of cache-related preemption delay," *Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, Newport Beach, CA, 2003, pp. 201-206.
19. J. V. Busquets-Mataix, J. J. Serrano, R. Ors, P. Gil, and A. Wellings, "Adding instruction cache effect to schedulability analysis of preemptive real-time systems," *Proceedings of the IEEE Real-Time Technology and Applications Symposium,* Brookline, MA, 1996, pp. 204-212.
20. D. Hardy and I. Puaut, "WCET analysis of multi-level non-inclusive set-associative instruction caches," *Proceedings of Real-Time Systems Symposium*, Barcelona, Spain, 2008, pp. 456-466.



Wei Zhang

Wei Zhang is an associate professor in Electrical and Computer Engineering of Virginia Commonwealth University. He received his Ph.D. from the Pennsylvania State University in 2003. From August 2003 to July 2010, he worked as an assistant professor and then as an associate professor at Southern Illinois University Carbondale. His research interests are in embedded and real-time computing systems, computer architecture, compiler, and low-power systems. He has received the 2009 SIUC Excellence through Commitment Outstanding Scholar Award for the College of Engineering, and 2007 IBM Real-time Innovation Award. His research has been supported by NSF, IBM, Intel, Motorola and Altera. He is a senior member of the IEEE. He has served as a member of the organizing or program committees for several IEEE/ACM international conferences and workshops.

Jun Yan

Jun Yan is currently a research scientist at MathWorks. He received his Ph.D. degree in Electrical and Computer Engineering from Southern Illinois University Carbondale (SIUC) in 2009. Before he came to SIUC, he worked in R&D at Lucent Technologies from 2004 to 2005 and at Huawei Technologies from 2002 to 2004. He received his M.S. from Tianjin University, China, in 2002, and B.S. from Shenyang Architecture and Civil Engineering Institute, China, in 1998, respectively.