JCSE Journal of Computing Science and Engineering

Regular Paper
Journal of Computing Science and Engineering,
Vol. 9, No. 4, December 2015, pp. 177-189

Exploiting Static Non-Uniform Cache Architectures for Hard Real-Time Computing

Yiqiang Ding and Wei Zhang*
Department of Electrical and Computer Engineering, Virginia Commonwealth University, Richmond, VA, USA
dingy4@vcu.edu, wzhang4@vcu.edu

Abstract
High-performance processors using Non-Uniform Cache Architecture (NUCA) are increasingly used to deal with the growing wire delays in multicore/manycore processors. Due to the convergence of high-performance computing with embedded computing, NUCA caches are expected to benefit high-end embedded systems as well. However, for real-time systems that use multicore processors with NUCA caches, it is crucial to bound worst-case execution time (WCET) accurately and safely. In this paper, we developed a WCET analysis approach by considering the effect of static NUCA caches on WCET. We compared the WCET in real-time applications with different topologies of static NUCA caches. Our experimental results demonstrated that the static NUCA cache could improve the worst-case performance of real-time applications using multicore processor compared to the cache with uniform access time.

Category: Embedded computing

Keywords: Non-Uniform Cache Architecture; Worst-case execution time; Real-time systems; Multicore processors

I. INTRODUCTION

Following the trend in servers and desktops, multicore processors such as ARM Cortex and Intel Atom have been increasingly used in embedded computing systems. Although multicore processors can potentially benefit real-time systems with better energy efficiency, the performance of real-time systems could be affected by increased global wire delays as more cores are put onto the same die in modern multicore processors. Specifically, large on-chip caches with uniform access time are undesirable because the last-level cache will be shared among all cores in multicore processors due to increased global wire delays. These traditional caches are physically partitioned into sub-banks. The uniform access time is determined by the access time to the furthest sub-bank. Therefore, the uniform access time of data throughout the cache can become longer when the global wire delays are increased. However, the access time of large on-chip caches varies with the physical locations of cache blocks. More specifically, the data residing in the part close to the core can be accessed much faster than the data residing physically further from the core because the access time involves not only the bank access time, but also the time to route to and from the bank.

Non-Uniform Cache Access (NUCA) architecture [1] is proved to be efficient in dealing with the non-uniform access latencies due to the growing wire delay. The NUCA cache is partitioned into multiple banks connected by a communication infrastructure. It is characterized by

Open Access http://dx.doi.org/10.5626/JCSE.2015.9.4.177 http://jcse.kiise.org

This is an Open Access article distributed under the terms of the Creative Commons Attribution Non-Commercial License (http://creativecommons.org/licenses/by-nc/3.0/) which permits unrestricted non-commercial use, distribution, and reproduction in any medium, provided the original work is properly cited.

Received 14 September 2015; **Revised** 30 October 2015; **Accepted** 19 November 2015
*Corresponding Author

This is an extended paper based on the prior conference paper entitled "WCET Analysis of Static NUCA Cache" that was published by the 33rd IEEE International Performance Computing and Communications Conference (IPCCC), December 2014.

Copyright © 2015. The Korean Institute of Information Scientists and Engineers pISSN: 1976-4677 eISSN: 2093-8020

non-uniform access time. The NUCA cache has the following two important characteristics: 1) the mapping policy determines the number of addressable banks contained in the cache and how cache blocks should be mapped into those banks and 2) the movement policy. If a cache block always stays in the same bank, it is called static NUCA. However, if cache block migration is allowed, it is called dynamic NUCA. Since cache block migration can significantly affect the time predictability that is bad for real-time systems, dynamic NUCA was not considered in this paper.

In a multicore processor, NUCA cache is typically used as the last-level cache shared among all cores (e.g., the shared L2 or L3 cache). As the cache is usually shaped in a 2-dimensional die, all cores can be physically distributed along any side of the cache. A cache bank that is physically close to one core cannot be physically close to the others. Even if the same application runs on different cores, its total cache access latency may be different. Therefore, the system topology of a multicore processor can also affect the performance and WCET of the applications if NUCA cache is used.

In the past few years, WCET analysis of multicore processors has received much attention by the real-time system community. A number of techniques have been proposed to conduct timing analysis of multicore processors by considering interferences from shared components such as caches and buses. For example, Yan et al. [2] have proposed a control flow based analysis approach to calculate the WCET of multicore processors with shared instruction caches. Li et al. [3] have estimated the potential conflicts and WCET in shared caches of multicore processors by considering lifetime estimates for the co-running tasks. Lv et al. [4] have proposed to combine abstract interpretation with model checking to estimate the timing bounds of programs running on multicore processors. Recent studies on multicore timing analysis have also considered the impact of bus contention [5] and unified data analysis, instruction, and caches [6]. However, all these prior research efforts only aimed at caches with uniform cache architecture (UCA), which could not be applied safely to NUCA caches with non-uniform cache access latencies. This paper is an extended work based on NUCA cache timing analysis presented in [7]. However, the work in [7] was primarily focused on dual-cores. This paper studied the NUCA cache timing analysis for quad-cores with different NUCA cache topologies. Our results indicated that the best topology for achieving the lowest WCET was different when different benchmarks were used, indicating that the topology for hard real-time applications could customized.

In this work, we performed WCET analysis for real-time applications running on multicore processors with static NUCA shared caches. The major contributions of this work include the following. First, we extended the CCCG-based approach [8, 9] to support the WCET analysis on shared static NUCA caches in multi-core processors. The extended approach calculated the access latency to each bank in the shared cache according to the physical distance between the bank and the core. The latencies were represented in the objective function to safely and accurately derive the WCET. Second, we compared the effect of different system topologies on WCET on multicore processors.

II. ARCHITECTURE AND ASSUMPTIONS

Without losing generality, we studied memory hierarchy consisting of three layers: L1 caches (including the L1 instruction and data caches), L2 cache, and main memory as shown in Fig. 1. The L1 cache was a UCA cache private to each processor. The L2 cache was a static NUCA to solve the problem of increasing global wire delays across the chip. This NUCA L2 cache consisted of multiple banks. All banks were connected within a 2D mesh network to remove most wires resulting from per-bank channels. The mesh network was comprised of the address bus, the data bus, and the switch. The latency to fetch the data from a bank included routing delay of the data and address on the buses, time to access the bank, and contention delay on the buses and the bank. The L2 cache was unified for both data and instructions. It was shared by all cores in a multicore processor.

In order to focus on the effect of static NUCA L2 cache on WCET, we made the following assumptions:

- The access latency to the L1 cache was constant.
- The effect of the L2 cache contention on the buses and the banks was not considered because routing delays in the NUCA cache were significant. In addition, the contention is less of a problem at small tech-

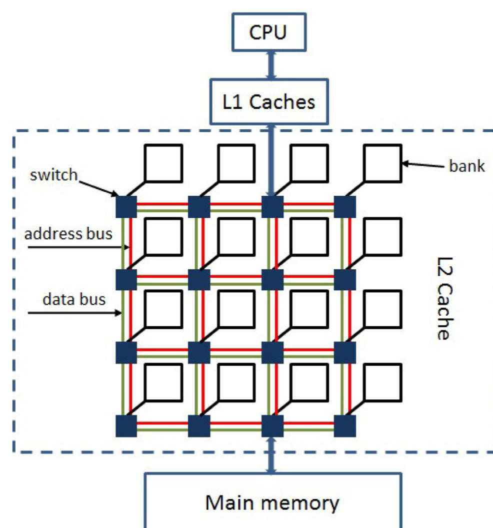


Fig. 1. Memory hierarchy and structure of NUCA-based L2 cache.

nologies [1]. Therefore the access latency to the L2 cache was proportional to the distance between the cache controller and the bank where the data resided.
- The access latency to the main memory was constant.
- The effect of bus contention between L2 cache and the main memory or the effect of contention on the main memory was not considered.

It should be noted that the cost to implement static NUCAs is not unacceptable. According to a study by Kim et al. [1], there are two different ways to implement the static NUCA caches, i.e., private channels and switched channels. In private channel based static NUCAs, each bank has private channels. Therefore, each bank can be accessed independently to achieve maximum speed. However, in switched channels, numerous per-bank channels are added to the array, which can constrain the number of banks. Nevertheless, the overhead of larger and slower banks is less than the delays that would be caused by the extra wires required for more numerous smaller banks. The switched channel based static NUCAs can remove most of those large numbers of wires resulting from per-bank channels. Experiments performed in [1] have revealed that channel overhead is only 5.9% of the total area of the banks.

III. CCCG-BASED APPROACH

The CCCG-based approach [8, 9] is based on implicit path enumeration technique (IPET) proposed by Li and Malik [10] and Li et al. [11]. It extends the cache conflict graph (CCG) in the IPET into a combined cache conflict graph (CCCG) in order to bound the worst-case cache interferences onto shared caches in multicore processors. A CCG is basically a projection of a control flow graph of a given thread on a cache set. It contains a set of nodes and edges. In a CCG, each node corresponds to a cache set while each edge represents a legal path in the CFG between two nodes. More details about CCG can be found in [10, 12].

In a CCCG, each vertex is denoted by a tuple $t(r : c)$, which is a combination of tuples r and c. The left side of this tuple records the last access from each thread while the right side of the tuple represents the current cache line state. The CCCG makes it possible to model concurrent cache accesses from threads running simultaneously on different cores, thus supporting WCET analysis of shared caches in multicore processors [8, 9]. However, the original CCCG-based approach is not aware of non-uniform cache access latencies. Thus, it cannot be directly applied to accurately estimate WCET for NUCA caches.

A. IPET

The IPET calculates the WCET by solving ILP problem. In the IPET, the WCET of a thread is represented by an objective function subjected to structural constraints, functionality constraints, and micro architectural constraints. The structural constraints show the constraints from the control flow of the program. The functionality constraints specify the loop bounds and other path information. The micro-architectural constraints represent the timing impact of micro-architectural components such as cache memories. All constraints are represented as equations or inequalities to bound the objective function in the ILP problem.

B. Constraints for Computing the WCET in Multicore Processors with Shared L2 Caches

The WCET of a thread running on a multicore processor with a shared L2 cache can be described as the objective function as shown in Eq. (1). The WCET is defined as the sum of the timing cost of each basic block and the timing cost of the access to each cache line block. The timing cost of a basic block equals to its execution latency timed by its execution count. The timing cost of the access to a cache line block equals to the cost of cache with miss and hit timed by the number of access to this cache line block. All symbols used in this paper are explained in Table 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 structural constraints are described in Eq. (2). It can derived from the control flow graph of a program.

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

Eq. (3) represents the functionality constraint. The upper bound of the execution count of a loop header block equals to the execution count of the pre-header block of this loop timed by the weight of this loop.

$$x_i^{header} \leq loop_weight \times x_j^{pre_header} \quad (3)$$

Eq. (4) describes the relationship between basic block B_i and L1 cache line block L_i contained in B_i.

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

The cache constraints are demonstrated in Eqs. (5)–(9), where l_j and l_p are execution counts of cache line blocks on L1 cache and L2 cache, respectively. Eq. (5) states the fact that the number of misses and hits of an L1 cache line block should be equal to the total execution count of this line block. Eqs. (6) and (9) are for the upper bound of the number of misses for a cache line block. It can be derived from the CCCG explained in Section III-C.

Table 1. Symbols used in this paper

Symbol	Description
B_i	Basic block i
L_i	Line block i
C_i	Timing cost of a basic block or a line block
X_i	Execution count of basic block i
L_i	Execution count of line block i
t_i	Execution count of tuple i in CCCG
$l1_hit$	L1 cache hit
$l1_miss$	L1 cache miss
$l2_hit$	L2 cache hit
$l2_miss$	L2 cache miss
d_{in}	Execution count of an edge going into a basic block
d_{out}	Execution count of an edge coming out of a basic block
e_{in}	Execution count of an edge going into a tuple in CCCG
e_{out}	Execution count of an edge coming out of a tuple in CCCG
e_{entry}	Execution count of an edge going into the starting tuple in CCCG
e_{hit}	Execution count of an edge leading to a cache hit for a tuple in CCCG
T_c	Computation time (by assuming a perfect memory)
T_{l1}	L1 cache access latency
T_{mem}	Main memory access latency

$$l_j = l_j^{l1_hit} + l_j^{l1_miss} \tag{5}$$

$$l_j^{l1_miss} \le \sum t_j^{miss} \tag{6}$$

$$\sum l_j^{l1_miss} = l_p \tag{7}$$

$$l_p = l_p^{l2_hit} + l_p^{l2_miss} \tag{8}$$

$$l_p^{l2_miss} \le \sum t_p^{miss} \tag{9}$$

C. Combined Cache Conflict Graph

The CCG is created for each cache set in IPET. Vertices in a CCG are line blocks mapped into the same cache set. Edges can be used to describe the valid transitions between the line blocks complying with the control flow of the thread analyzed. However, the CCG can only represent the cache states and related transitions caused by line blocks from one thread. It must be enhanced in the following three aspects so that it can represent the behavior of shared cache caused by line blocks from all co-running threads in a multicore processor. First, it should be able to memorize historical cache accestses from all threads. Second, it should represent change in cache state caused by a cache access from any thread. Third, it should be able to derive cache states without violating the control flow or program semantics of any thread.

In a CCCG, each vertex represents a comprehensive cache state. It is denoted by a tuple $t(r : c)$ consisting of two sub tuples r and c. The sub tuple $r(l_1, l_2, ..., l_m)$ records the most recent cache access from each thread, where l_i is the most recent cache access from the thread T_i. The sub tuple $c(x_1, x_2, .., x_n)$ represents a cache state, where x_i represents the line blocks located in cache. Both i and n are associativity of the cache being modeled. To build the CCCG for a cache set, we first construct the CCG of this cache set for each of the concurrent threads. The first state in the CCCG is initialized as $t(s_1, s_2, s_m : \phi_1, \phi_2, ..., \phi_n)$, where s_i is the starting node in the CCG of the thread T_i while ϕ_i in the ith way of the cache set is blank. Other cache states are derived by traversing all paths from the start vertex to the end vertex of all CCGs. For a cache state t_{curr} already existing in the CCCG, if the current cache access $l_{i,j}$ of a thread T_i has an outgoing edge d_i that leads to another cache access $l_{i,k}$ in its CCG, a new cache state t_{next} is created by updating both sub tuples r and c in t_{curr} with $l_{i,k}$. For example, if t_{curr} is represented as $t(l_1, ...l_{i,j}, ...l_m : x_1, ..., x_{i,j}, ... x_n)$, t_{next} can be represented as

$t(l_1, \ldots l_{i,k}, \ldots l_m : x_1, \ldots, x_{i,k}, \ldots x_n)$. The update of sub tuple c can be implemented with different cache replacement policies such as FIFO and LRU. In addition, an edge is added from t_{curr} to t_{next} to indicate the transition. It should be mentioned that the upper-level cache state in case of an L2 cache should be checked before the creation of t_{next}. If the transition leads to an L1 cache hit, t_{next} can be just simply inherited from t_{curr}.

Since the CCCG describes the cache behavior on a shared cache caused by cache accesses from all co-running threads, it can produce constraints to bound the worst-case cache access latency for a given thread considering that the inter-thread cache can interfere with other threads. The sum of all entry edges of the CCCG should equal to 1 (Eq. (10)). The structural constraint and the functionality constraint of the CCCG are shown in Eq. (11) and Inequality (13), respectively. The constraints of the relationship between the tuple in the CCCG and the line blocks involved in this tuple are described in Eqs. (12) and (6) as well as Eq. (9) mentioned in Section III-B. In addition, the constraints that bound the cache hit of the tuple in the CCCG are represented in Eqs. (14) and (15). More details of the CCCG can be found in [8, 9].

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

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

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

$$t_i \le loop_weight \times \sum e_{in} \tag{13}$$

$$t_i^{hit} \le \sum e_{hit} \tag{14}$$

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

IV. AN EXAMPLE OF USING CCCGs

An example to illustrate how to apply the CCCG to WCET analysis [9] is illustrated in Fig. 2. For simplicity, it was assumed that there were two threads: an RT and an 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. Assuming that the cache is 2-way set-associative with one set, a, b, and c should be mapped to the same cache line. In addition, we assumed that the cache hit latency was 1 cycle and the cache miss latency was 100 cycles. The CFGs for the RT and NRT are shown in Fig. 2(a). CCGs for the RT and NRT are depicted in Fig. 2(b). CCCG constructed automatically by applying Algorithm 1 to the CFGs and CCGs of both threads is shown in Fig. 2(c).

A. Objective Function

The following objective function is derived from Eq. (1).

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

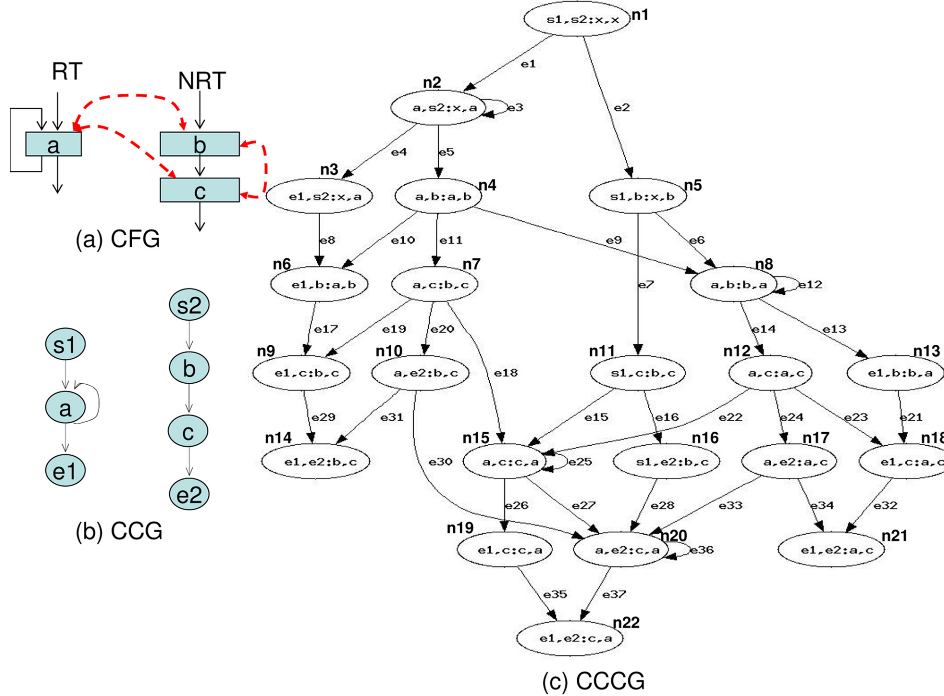


Fig. 2. An example of applying CCCGs to estimate the WCET [10].

B. Structural Constraints

Structural constraints were derived from Eq. (2). First, we constructed 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).

$$a - ds_a - da_a = 0 \quad (17)$$

$$a - da_a - da_e = 0 \quad (18)$$

In addition, we could construct structural constraints for NRT using the following equations:

$$b - ds_b = 0 \quad (19)$$

$$b - db_c = 0 \quad (20)$$

$$c - db_c = 0 \quad (21)$$

$$c - dc_e = 0 \quad (22)$$

C. Functionality Constraints

For each thread, we assumed that each of them could execute only once.

$$ds_a = 1 \quad (23)$$

$$ds_b = 1 \quad (24)$$

We also assumed that the loop in the RT thread could execute no more than 10 iterations based on Eq. (3).

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

D. Cache Constraints

Cache constraints were obtained from Fig. 2(c).

1) Connection Constraints: The following equation describes the constraints, i.e., the sum of instruction a's different states equals to the execution counts of instruction a. This is derived from Eq. (15).

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

2) CCCG Entry Edge Constraints: The RT thread was executed only once. Therefore, we generated the following equation based on Eq. (10).

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

Similarly, the NRT thread was executed only once, leading to the following equation:

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

3) CCCG Node Constraints: The sum of all nodes that execute instruction a must be equal to the execution counts of instruction a. Therefore, we have the following equation derived from Eq. (12).

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

Similarly, the sum of all nodes that executed instruction b also should be equal to the execution counts of instruction b. Hence, we could derive the following equation:

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

The sum of all the nodes that execute instruction c must also be equal to the execution counts of instruction c. Therefore, we have the following equation:

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

4) Hit Edge Constraints: The following equations are derived from Eq. (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 be equal to 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 Eq. (14) for the bound number of cache hits:

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

6) Put Them All Together: The WCET path for a given example is shown in Fig. 3. The final result from ILP (i.e., the WCET) was 208 which could be derived from Eq. (37).

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

V. TIMING ANALYSIS OF NUCA CACHES

To enable the multicore WCET analysis of static NUCA cache, the CCCG-based approach was extended in two aspects. First, the extended approach needed to calculate

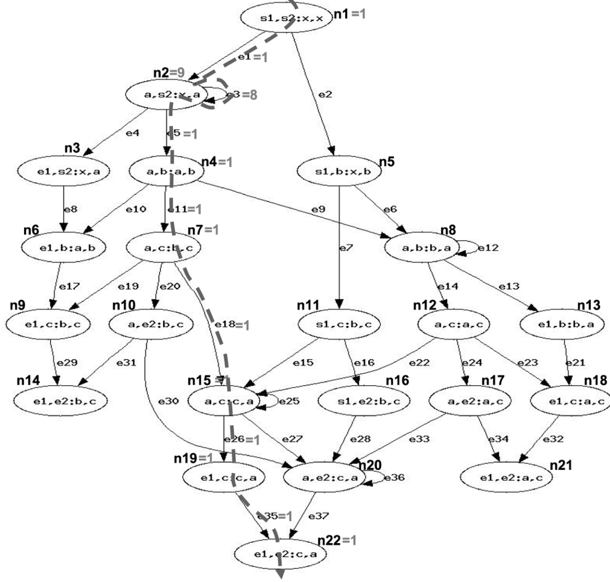


Fig. 3. The worst-case path for a given example [10].

the access latency to each bank of the cache from each core according to the physical distance between them. Second, the objective function to compute the WCET needed to be modified by replacing the uniform access latency to all L2 cache blocks with a specific access latency for each cache block according to the bank where the cache block was mapped into.

As mentioned in Section II, the NUCA-based L2 cache consisted of multiple banks connected to a 2D mesh network. Each bank B_i could be viewed as a node in a matrix with *m* rows and *n* columns that could be represented by a tuple $< row_i, col_i >$, where row_i was the row id and col_i was the column id of bank B_i. The delay between two adjacent nodes could be defined as a constant *d*. As the cache controller between the L1 cache and the L2 cache was connected to a switch in the mesh network, the cache controller could be represented by a tuple $< row_c, col_c >$. If the transfer between two adjacent nodes was defined as a hop, the distance between a bank and a cache controller could be represented by the number of hops on the route between them. The access latency L_i to a bank B_i in the L2 cache from a core could be described in Eq. (38), where $abs(row_i - row_c) + abs(col_i - col_c)$ was the number of hops of the route from the cache controller to the bank B_i. The latency of transferring through the cache controller was assumed to be a constant *d*.

$$L_i = (abs(row_i - row_c) + abs(col_i - col_c) + 1) \times d \quad (38)$$

To calculate the access latency of each L2 cache block, it is necessary to map the id of the cache block to the id of the bank where the data reside. The mapping relation is described in Eq. (39), where $Bank_i$ represents the bank id while CB_j represents the cache block id. Thus, the access

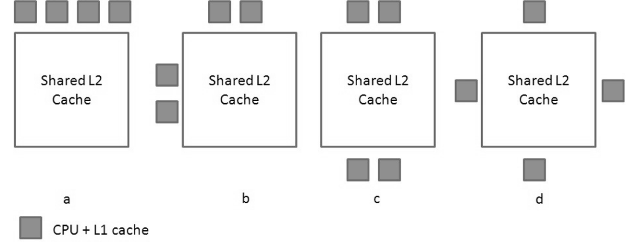


Fig. 4. Different topologies in a 4-core processor with a shared L2 cache.

latency to the data residing in the cache block CB_j can be defined by Eq. (40), which is equal to the access latency to the bank that the cache block is assigned to.

$$Bank_i = CB_j \% (blocks\ per\ bank) \quad (39)$$

$$D_j = L_i \quad (40)$$

The objective function representing the WCET in the CCCG-based approach could be modeled by Eq. (41), where A_CB_j was the number of accesses to the cache block CB_j and D_j was the access latency to this cache block as described in Eq. (40).

$$WCET = T_c + \sum T_{l1} + \sum A_CB_j \times D_j + \sum T_{mem} \quad (41)$$

VI. SYSTEM TOPOLOGY

As shown in Fig. 1, the shared L2 cache is usually in a shape of a square. The access latency to a bank in the L2 cache from a core varies if the core is connected to different sides of the L2 cache physically because the physical distance between the core and a bank changes with different layouts of multicore processor.

Typically, there are four types of system topology in a multicore processor with shared L2 caches: (a) all cores are connected to the same side of the L2 cache; (b) all cores are connected to two neighboring sides of the L2 cache symmetrically; (c) all cores are connected to two opposite sides of the L2 cache symmetrically; (d) all processors are connected to four sides of the L2 cache symmetrically. An example of four different topologies in case of a 4-core processor with a shared L2 cache is shown in Fig. 4.

VII. EVALUATION METHODOLOGY

A. Experimental Framework and Benchmarks

The experimental framework of our multicore WCET analysis approach consisted of three main components: a front end, a back end, and an ILP solver. The front end

was based on a Trimaran compiler that could compile the benchmarks into COFF format binary code. The global control flow graph (CFG) and related information of the instructions were then constructed by disassembling the binary code generated by the compiler. The back end could perform static cache analysis [13] on intermediate representation (IR) of each benchmark, supporting the extended CCCG-based approach. A commercial ILP solver-CPLEX [14] was used to solve the ILP problem to compute the WCET. In addition, we extended the simulator of the Trimaran infrastructure to simulate multicore processors and report the execution cycle results.

In the simulated multicore processor, the L2 cache was shared. Each core had its own L1 instruction cache and L1 data cache. The L1 instruction cache was directed and mapped. Its total size and cache block size were 512 bytes and 8 bytes, respectively (Note that it is not uncommon to use small caches for WCET analysis research. This is because the size of the WCET benchmarks is mostly very small. Thus a regular cache, for example a 16 kB cache, can easily hold all the instructions of a benchmark, making it impossible to study cache misses that typically occur in real-life applications). The L1 data cache was assumed to be perfect as our work focused on the timing analysis of instruction caches. The impact of data cache accesses will be studied in our future work. The shared L2 cache was a 2-way set-associative cache with a total size of 2048 bytes. Its block size was 16 bytes.

The timing characteristics of each level of the memory hierarchy in our study are defined in Table 2. The L1 cache was a UCA cache and the uniform access latency was defined as 1 cycle. In order to analyze the effect of NUCA cache on WCET, the L2 cache was implemented with NUCA or UCA cache. The NUCA-based L2 cache was partitioned into 16 banks, with the unit delay between two adjacent switches in the communication network defined as 1 cycle. Comparatively, the uniform access latency of the UCA-based L2 cache was defined as 7 cycles, which was the delay of the longest route between two switches in the communication network of NUCA-based L2 cache (refer to Fig. 1). The uniform access latency to the main memory was defined as 100 cycles.

Table 2. Timing characteristics of the memory hierarchy

Level	Access latency
L1 cache	1 cycle
L2 cache	
NUCA	16 bank, 1 cycle per hop
UCA	7 cycles
Main memory	100 cycles

Malardalen WCET benchmarks [15] were used. The description and characteristics of each benchmark are listed in Table 3, including the number of instructions, the number of L2 cache accesses, and simulated WCET on a single-core processor. Some benchmarks were selected for each experiment. Each benchmark was used as an independent thread running on one core of the multicore processor.

B. Topologies of NUCA Caches Evaluated

To define the system topology of the multicore processor with a shared L2 cache, the four sides of the shared L2 cache were labeled as *Up*, *Right*, *Down*, or *Left*, starting from the top side in a clockwise order.

We executed the experiments on both 2-core processor and 4-core processor. For the 2-core processor, one core was fixed on the *Up* side of the shared L2 cache while the other core was positioned to one of the four sides of the shared L2 cache for different topologies.

For the 4-core processor, experiments were run on NUCA caches with four different topologies as shown in the following:

- Up&Up&Up&Up (UUUU): all cores were connected to the *Up* side of the shared L2 cache.
- Up&Up&Right&Right (UURR): two cores were connected to the *Up* side of the shared L2 cache while the other two cores were positioned to the *Right* side.
- Up&Up&Down&Down (UUDD): two cores were

Table 3. Benchmark description and its characteristics

Benchmark	Description	No. of insts	L1 accesses	L2 accesses	Simulated WCET
Cover	Multiple path testing	64	798	16	2086
Expint	Computing an exponential integer function	130	11391	21	28985
Insertsort	Insertion sort on a reverse array	62	200	16	1510
Jfdctint	Discrete-cosine transformation	279	459	72	5488
Ludcmp	Read ten values, output half to LCD	259	819	85	5846
Minver	Inversion of floating point matrix	341	666	100	5940
Qsort	Non-recursive quick sort algorithm	148	590	57	3787
Select	Select the Nth largest number	124	1109	31	4633

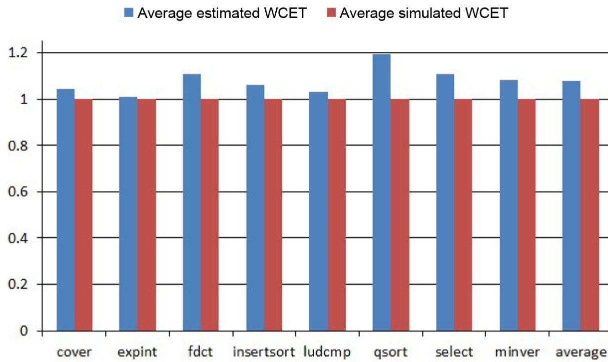


Fig. 5. Average estimated WCET compared to average simulated WCET for a 2-core processor with a static L2 NUCA cache. The y-axis represents WCET normalized to average simulated WCET.

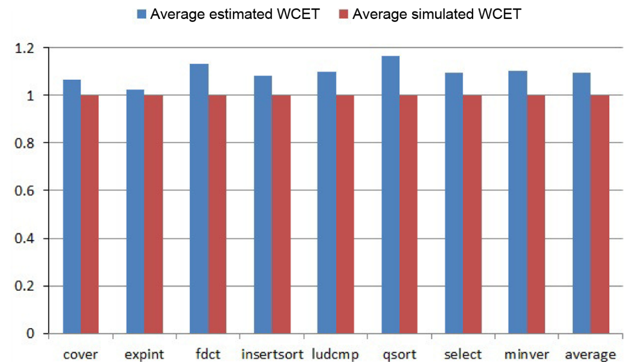


Fig. 6. Average estimated WCET compared to average simulated WCET for a 4-core processor with a static L2 NUCA cache. The y-axis represents WCET normalized to average simulated WCET.

connected to the *Up* side of the shared L2 cache while the other two cores were positioned to the *Down* side.

- Up&Right&Down&Left (URDL): each core was positioned to each side of the shared L2 cache.

VIII. EXPERIMENTAL RESULTS

A. Safety and Tightness of Proposed WCET Analysis for NUCA Caches

Safety and tightness are two desirable goals for any WCET analysis technique. While safety guarantees that the estimated WCET is the upper bound of the execution time, tightness ensures that the estimated WCET is as close as possible to the actual WCET. Otherwise, the WCET may be overestimated, thus impacting the efficiency of real-time scheduling. In our experiments, we first compared the average estimated WCET of each benchmark with the average observed WCET through simulation to assess the safety and tightness of the proposed WCET analysis for NUCA caches. The average estimated/simulated WCET of a benchmark was the average value of the estimated/simulated WCETs obtained from different experiments based on four different topologies. The average estimated and simulated WCETs for a 2-core processor with a static L2 NUCA cache are shown in Fig. 5. The average estimated WCET was found to be slightly larger than the average simulated WCET for every benchmark, indicating that our WCET analysis could report the upper bound of the execution time. The gap between the average estimated WCET and the average simulated WCET ranged from 1.82% to 19.03%, with an average of approximately 7%.

The average estimated WCET compared to the average simulated WCET of each benchmark for a 4-core processor with a static L2 NUCA cache is shown in Fig. 6. The estimated WCET was found to be always larger than the simulated WCET, indicating the safeness of our WCET analysis. The gap between the average estimated WCET and the average simulated Fig. 6. The WCET ranged from 3.16% to 18.74% for all benchmarks, with an average of approximately 9%. Considering the conservative nature of cache timing analysis, the variation of NUCA cache access latencies and topologies, we believe that our WCET analysis of the NUCA caches is reasonably accurate.

B. NUCA vs. UCA Caches

We also compared the estimated and observed WCETs of NUCA and UCA caches to understand the pros and cons of using NUCA caches for real-time systems. The estimated WCET of all the benchmarks in different topologies of the NUCA cache compared to estimated WCET of the UCA cache for a 2-core processor is shown in Table 4. In general, the estimated WCETs of all the benchmarks of the NUCA cache are smaller than those of the UCA cache. For example, the estimated WCETs of the benchmark *Jfdctint* of the NUCA cache were 2.66%, 5.21%, 3.03%, and 2.18% lower than those of the UCA cache when the second core was on the Up, Right, Down, and Left sides of the NUCA cache, respectively. On average, the estimated WCETs for all benchmarks of the NUCA cache were 2.67%, 2.17%, 2.01%, and 2.83% lower than those of the UCA cache when the second core was on the Up, Right, Down, and Left sides, respectively.

The simulated (or observed) WCET of all the benchmarks in different topologies of the NUCA cache compared to the estimated WCET of the UCA cache for a 2-core processor is shown in Table 5. The average simulated WCETs of the NUCA cache for all benchmarks were found to be 2.89%, 2.27%, 2.14%, and 3.04% lower than those of the UCA cache when the second core was on the Up, Right, Down, and Left sides, respectively. These results indicate that NUCA caches can reduce the WCET compared to UCA caches. For an NUCA cache, different benchmarks were found to have their optimal estimated/simulated WCETs with different topologies

Table 4. Comparison of the estimated WCET of the NUCA cache with 4 different topologies and the UCA cache in a 2-core processor

Benchmark	Estimated WCET					Normalization			
	UCA	Up	Right	Down	Left	Up	Right	Down	Left
Cover	2486	2422	2464	2464	2422	0.9743	0.9912	0.9912	0.9743
Expint	29585	29487	29550	29550	29515	0.9967	0.9988	0.9988	0.9976
Insertsort	1810	1746	1794	1794	1746	0.9646	0.9912	0.9912	0.9646
Jfdctint	6609	6433	6265	6409	6465	0.9734	0.9479	0.9697	0.9782
Ludcmp	7260	7035	7038	7090	6837	0.9690	0.9694	0.9766	0.9417
Minver	7075	6897	6793	6669	6865	0.9748	0.9601	0.9426	0.9703
Qsort	5901	5626	5797	5797	5720	0.9534	0.9824	0.9824	0.9693
Select	5333	5228	5257	5263	5212	0.9803	0.9857	0.9869	0.9773

Table 5. Comparison of the simulated WCET of the NUCA cache with 4 different topologies and the UCA cache in a 2-core processor

Benchmark	Simulated WCET					Normalization			
	UCA	Up	Right	Down	Left	Up	Right	Down	Left
Cover	2386	2322	2364	2364	2322	0.9732	0.9908	0.9908	0.9732
Expint	29285	29187	29250	29250	29215	0.9967	0.9988	0.9988	0.9976
Insertsort	1710	1646	1694	1694	1646	0.9626	0.9906	0.9906	0.9626
Jfdctint	5988	5819	5655	5798	5849	0.9718	0.9444	0.9683	0.9768
Ludcmp	7046	6829	6845	6881	6629	0.9692	0.9715	0.9766	0.9408
Minver	6540	6376	6274	6152	6341	0.9749	0.9593	0.9407	0.9696
Qsort	4587	4321	4490	4487	4411	0.9420	0.9789	0.9782	0.9616
Select	4833	4728	4757	4763	4712	0.9783	0.9843	0.9855	0.9750

because the locations of their cache references were dependent on benchmark behavior. For example, benchmark *Jfdctint* had the lowest estimated/simulated WCET if it was executed on the core connected to the *Right* side of the L2 NUCA cache, while benchmark *Qsort* had the minimum estimated/simulated WCET when the second core was connected to the *Up* side of the L2 NUCA cache.

C. Quad-Core Results

We also compared the estimated and simulated WCETs between the NUCA and the UCA caches on a 4-core processor. The estimated WCET of all the benchmarks in different topologies of the NUCA cache and the estimated WCET of the UCA cache of a 4-core processor are listed in Table 6. In these experiments, we formed a group of benchmarks where each group consisted of 4 benchmarks running concurrently on different cores. The group number is shown in the first column (i.e., No) of this table. In general, the estimated WCET of each benchmark in any topology of the NUCA cache was smaller than that of the UCA cache. For example, the estimated WCET of the benchmark *Ud* with the topologies of *UUUU*, *UURR*, *UUDD*, and *URLD* of the NUCA cache was 2.29%, 3.63%, 5.22%, and 5.22% lower than that of the UCA cache, respectively. The average WCETs of the NUCA cache with the *UUUU*, *UURR*, *UUDD*, and *URLD* topologies was 3.33%, 3.02%, 3.60%, and 3.88% lower than those of the UCA cache, respectively.

The simulated WCETs of all benchmarks of the NUCA cache with 4 different topologies are less than those of the UCA cache (Table 7). In this table, the group number is shown in the first column (i.e., No). These results confirmed that the NUCA cache could achieve lower WCET than the UCA cache with the same size. We also found that the differences between the estimated WCET and the simulated WCET were mostly caused by overestimated cache misses due to the conservative nature and safety required for WCET analysis.

However, no topology of the NUCA cache could always achieve the lowest overall WCET than other topologies in the experiments. For example, the overall estimated WCET of *URLD* topology was found to be the lowest in Experiment 4, while the lowest overall estimated WCET was achieved in *UURR* topology in Experiment 5. It is because a benchmark has different cache access schemes

Table 6. Comparison of the estimated WCET of the NUCA cache with 4 different topologies and the UCA cache in a 4-core processor

No	Benchmark	Estimated WCET					Normalization			
		UCA	UUUU	UURR	UUDD	URDL	UUUU	UURR	UUDD	URDL
1	Cover	2586	2522	2522	2522	2522	0.9753	0.9753	0.9753	0.9753
	Expint	29585	29487	29487	29487	29515	0.9967	0.9967	0.9967	0.9976
	Jfdctint	6309	6133	5965	6109	6109	0.9721	0.9455	0.9683	0.9683
	Insertsort	1810	1746	1794	1794	1746	0.9646	0.9912	0.9912	0.9646
2	Expint	29485	29387	29387	29387	29387	0.9967	0.9967	0.9967	0.9967
	Jfdctint	6909	6733	6733	6733	6565	0.9745	0.9745	0.9745	0.9502
	Qsort	4901	4626	4797	4797	4797	0.9439	0.9788	0.9788	0.9788
	Select	5333	5228	5257	5263	5212	0.9803	0.9857	0.9869	0.9773
3	Ludcmp	8160	7935	7935	7935	7935	0.9724	0.9724	0.9724	0.9724
	Minver	7675	7497	7497	7497	7393	0.9768	0.9768	0.9768	0.9633
	select	5333	5228	5257	5263	5263	0.9803	0.9857	0.9869	0.9869
	Ud	7300	6991	7151	7177	7139	0.9577	0.9796	0.9832	0.9779
4	Cover	2586	2322	2322	2322	2322	0.8979	0.8979	0.8979	0.8979
	Ludcmp	7560	7335	7335	7335	7358	0.9702	0.9702	0.9702	0.9733
	Minver	7775	7597	7493	7369	7369	0.9771	0.9637	0.9478	0.9478
	Qsort	5501	5226	5397	5397	5320	0.9500	0.9811	0.9811	0.9671
5	Ud	7100	6791	6791	6791	6791	0.9565	0.9565	0.9565	0.9565
	select	5433	5328	5328	5328	5357	0.9807	0.9807	0.9807	0.9860
	insertsort	1910	1846	1894	1894	1894	0.9665	0.9916	0.9916	0.9916
	Jfdctint	7409	7233	7065	7209	7265	0.9762	0.9536	0.9730	0.9806
6	Qsort	5401	5126	5126	5126	5126	0.9491	0.9491	0.9491	0.9491
	insertsort	1910	1846	1846	1846	1894	0.9665	0.9665	0.9665	0.9916
	Ud	6900	6591	6751	6777	6777	0.9552	0.9784	0.9822	0.9822
	Ludcmp	7460	7235	7258	7290	7037	0.9698	0.9729	0.9772	0.9433
7	Minver	7575	7397	7397	7397	7397	0.9765	0.9765	0.9765	0.9765
	Ud	7100	6791	6791	6791	6951	0.9565	0.9565	0.9565	0.9790
	Expint	29685	29587	29650	29650	29650	0.9967	0.9988	0.9988	0.9988
	Cover	2586	2522	2564	2564	2522	0.9753	0.9915	0.9915	0.9753

when it runs on the cores connected to different sides of the NUCA cache. In addition, different benchmarks have different cache access schemes if they run on the core connected to the same side of the NUCA cache.

IX. CONCLUSION

Growing wire delay has made the traditional caches with uniform access latencies inefficient for multicore processors, especially when the number of cores keeps increasing. As multicore processors are widely used in real-time systems, it becomes crucial to develop a WCET analysis method for multicore processors with NUCA caches.

In this work, we extended the CCCG-based approach to support the WCET analysis on shared L2 cache implemented with the static NUCA in multicore processors. Our experimental results showed that the static NUCA cache could improve the worst-case performance of the real-time applications in the 2-core and 4-core processors compared to UCA caches. We also compared the effect of different topologies of static NUCA cache on WCET for a 2-core processor. Our evaluation indicated that no topology could achieve the optimal WCET for all benchmarks. The best topology to minimize WCET is depen-

Table 7. Comparison of the simulated WCET of the NUCA cache with 4 different topologies and the UCA cache in a 4-core processor

No	Benchmark	Simulated WCET					Normalization			
		UCA	UUUU	UURR	UUDD	URDL	UUUU	UURR	UUDD	URDL
1	Cover	2486	2422	2422	2422	2422	0.9743	0.9743	0.9743	0.9743
	Expint	29485	29387	29387	29387	29450	0.9988	0.9967	0.9967	0.9988
	Jfdctint	5888	5719	5555	5698	5698	0.9677	0.9434	0.9677	0.9677
	Insertsort	1810	1746	1794	1794	1746	0.9646	0.9912	0.9912	0.9646
2	Expint	29285	29187	29187	29187	29187	0.9967	0.9967	0.9967	0.9967
	Jfdctint	6288	6119	6119	6119	5955	0.9731	0.9731	0.9731	0.9470
	Qsort	4387	4121	4290	4287	4287	0.9394	0.9779	0.9772	0.9772
	Select	5133	5028	5057	5063	5057	0.9795	0.9852	0.9864	0.9852
3	Ludcmp	7346	7129	7129	7129	7129	0.9705	0.9705	0.9705	0.9705
	Minver	7140	6976	6976	6976	6874	0.9770	0.9770	0.9770	0.9627
	Select	5233	5128	5157	5163	5163	0.9799	0.9855	0.9866	0.9866
	Ud	6672	6379	6533	6566	6524	0.9561	0.9792	0.9841	0.9778
4	Cover	2486	2422	2422	2422	2422	0.9743	0.9743	0.9743	0.9743
	Ludcmp	6846	6629	6629	6629	6645	0.9683	0.9683	0.9683	0.9706
	Minver	7140	6976	6874	6752	6752	0.9770	0.9627	0.9457	0.9457
	Qsort	4887	4621	4790	4787	4711	0.9456	0.9802	0.9795	0.9640
5	Ud	6272	5979	5979	5979	5979	0.9533	0.9533	0.9533	0.9533
	Select	5233	5128	5128	5128	5157	0.9799	0.9799 0	.9799	0.9855
	Insertsort	1910	1846	1894	1894	1894	0.9665	0.9916	0.9916	0.9916
	Jfdctint	6488	6319	6155	6298	6349	0.9740	0.9487	0.9707	0.9786
6	Qsort	4687	4421	4421	4421	4421	0.9432	0.9432	0.9432	0.9432
	Insertsort	1910	1846	1846	1846	1894	0.9665	0.9665	0.9665	0.9916
	Ud	6172	5879	6133	6056	6056	0.9525	0.9937	0.9812	0.9812
	Ludcmp	6746	6529	6545	6581	6329	0.9678	0.9702	0.9755	0.9382
7	Minver	6840	6676	6676	6676	6676	0.9760	0.9760	0.9760	0.9760
	Ud	6472	6179	6179	6179	6333	0.9547	0.9547	0.9547	0.9785
	Expint	29485	29387	29450	29450	29450	0.9967	0.9988	0.9988	0.9988
	Cover	2486	2422	2464	2464	2422	0.9743	0.9912	0.9912	0.9743

dent on memory access patterns of different applications.

In our future work, we would like to perform WCET analysis for NUCA caches with more number of cores. As embedded systems are increasingly using heterogeneous multicore as System-on-Chip (SoC), we would also like to extend our method for timed analysis of embedded Graphics Processing Units (GPU) caches.

ACKNOWLEDGMENTS

This work was funded in part by NSF grants CCF 1063645 and CNS 1421577.

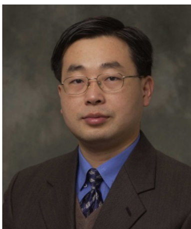
REFERENCES

1. C. Kim, D. Burger, and S. W. Keckler, "An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, San Jose, CA, 2002, pp. 211-222.
2. J. Yan and W. Zhang, "WCET analysis for multi-core processors with shared L2 instruction caches," in *Proceedings of 14th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'08)*, St. Louis, MO, 2008, pp. 80-89.
3. Y. Li, V. Suhendra, Y. Liang, T. Mitra, and A. Roy-

choudhury, “Timing analysis of concurrent programs running on shared cache multi-cores,” in *Proceedings of 30th IEEE Real-time System Symposium (RTSS)*, Washington, DC, 2009, pp. 57-67.
4. M. Lv, W. Yi, N. Guan, and G. Yu, “Combining abstract interpretation with model checking for timing analysis of multicore software,” in *Proceedings of 31st IEEE International Real-Time System Symposium (RTSS)*, San Diego, CA, 2010, pp. 339-349.
5. T. Kelter, H. Falk, P. Marwedel, S. Chattopadhyay, and A. Roychoudhury, “Bus-aware multicore WCET analysis through TDMA offset bounds,” in *Proceedings of the 23rd Euromicro Conference on Real-Time Systems (ECRTS),* Porto, Portugal, 2011, pp. 3-12.
6. S. Chattopadhyay, L. K. Chong, A. Roychoudhury, T. Kelter, P. Marwedel, and H. Falk, “A unified WCET analysis framework for multi-core platforms,” in *Proceedings of IEEE 18th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Beijing, China, 2012, pp. 99-108.
7. Y. Ding and W. Zhang, “WCET analysis of static NUCA caches,” in *Proceedings of the 33rd IEEE International Performance Computing and Communications Conference (IPCCC)*, Austin, TX, 2014, pp. 1-6.
8. W. Zhang and J. Yan, “A unified timing analysis approach for shared caches of multicores,” in *Proceedings of the Work-in-Progress (WIP) session of 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Chicago, IL, 2011.
9. W. Zhang and J. Yan, “Static timing analysis of shared caches for multicore processors,” *Journal of Computing Science and Engineering*, vol. 6, no. 4, pp. 267-278, 2012.
10. Y. S. Li and S. Malik, “Performance analysis of embedded software using implicit path enumeration,” in *Proceedings of the ACM SIGPLAN 1995 Workshop on Languages, Compilers, & Tools for Real-Time Systems (LCT-RTS 1995)*, La Jolla, CA, 1995, pp. 88-98.
11. Y. S. Li, S. Malik, and A. Wolfe, “Performance estimation of embedded software with instruction cache modeling,” in *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, San Jose, CA, 1995, pp. 380-387.
12. Y. S. Li, S. Malik, and A. Wolfe, “Cache modeling for real-time software: beyond direct mapped instruction caches,” in *Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS'96)*, Washington, DC, 1993, pp. 254-263.
13. C. A. Healy, R. D. Arnold, F. Mueller, D. Whalley, and M. G. Harmon, “Bounding pipeline and instruction cache performance,” *IEEE Transactions on Computers*, vol. 48, no. 1, pp. 53-70, 1999.
14. Homepage of CPLEX, http://www.ilog.com/products/cplex/.
15. Malardalen WCET Research Group, Malardalen WCET benchmark suite, http://www.mrtc.mdh.se/projects/wcet.

Yiqiang Ding

Yiqiang Ding is currently a Ph.D. student in Electrical and Computer Engineering at Virginia Commonwealth University. He received B.S. degree in computer science in 2002 and M.S. degree in computer engineering in 2005 from Beijing University of Posts and Telecommunications, China. He worked in Motorola China Design Center as a system engineer from 2005 to 2007. His research interests included embedded and real-time computing systems, computer architecture, and compiler.



Wei Zhang

Wei Zhang is a professor in the Department of Electrical and Computer Engineering at Virginia Commonwealth University. He received his Ph.D. from Pennsylvania State University in 2003. From August 2003 to July 2010, he worked as an assistant professor and then tenured associate professor at Southern Illinois University Carbondale. His research interests include embedded computing systems, real-time computing systems, computer architecture, compiler, and low-power systems. He has received the 2009 Excellence through Commitment Outstanding Scholar Award from College of Engineering at Southern Illinois University Carbondale and 2007 IBM Real-time Innovation Award. He has received 5 research grants from National Science Foundation. His research and educational efforts have been supported by leading IT companies such as IBM, Intel, Motorola, and Altera. He has published more than 120 papers in refereed journals and conference proceedings. He is a senior member of the IEEE and an associate editor for the Journal of Computing Science and Engineering. He has served as a member of the organization or program committees for several IEEE/ACM international conferences and workshops.