Counter-Based Approaches for Efficient WCET Analysis of Multicore Processors with Shared Caches

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

Abstract
To enable hard real-time systems to take advantage of multicore processors, it is crucial to obtain the worst-case execution time (WCET) for programs running on multicore processors. However, this is challenging and complicated due to the inter-thread interferences from the shared resources in a multicore processor. Recent research used the combined cache conflict graph (CCCG) to model and compute the worst-case inter-thread interferences on a shared L2 cache in a multicore processor, which is called the CCCG-based approach in this paper. Although it can compute the WCET safely and accurately, its computational complexity is exponential and prohibitive for a large number of cores. In this paper, we propose three counter-based approaches to significantly reduce the complexity of the multicore WCET analysis, while achieving absolute safety with tightness close to the CCCG-based approach. The basic counter-based approach simply counts the worst-case number of cache line blocks mapped to a cache set of a shared L2 cache from all the concurrent threads, and compares it with the associativity of the cache set to compute the worst-case cache behavior. The enhanced counter-based approach uses techniques to enhance the accuracy of calculating the counters. The hybrid counter-based approach combines the enhanced counter-based approach and the CCCG-based approach to further improve the tightness of analysis without significantly increasing the complexity. Our experiments on a 4-core processor indicate that the enhanced counter-based approach overestimates the WCET by 14% on average compared to the CCCG-based approach, while its averaged running time is less than 1/380 that of the CCCG-based approach. The hybrid approach reduces the overestimation to only 2.65%, while its running time is less than 1/150 that of the CCCG-based approach on average.

Category: Embedded computing

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

I. INTRODUCTION
In hard real-time systems, the success of the schedulability analysis depends on accurately obtaining the worst-case execution time (WCET) of the tasks. While a multicore processor can potentially benefit real-time systems for achieving better performance and energy efficiency, it is very challenging to obtain the WCET accurately on a multicore processor due to complex inter-thread interferences in shared resources on a multicore chip, such as a shared L2 cache.

Multicore WCET analysis has been actively investigated recently. Yan and Zhang [1, 2] proposed a coarse-grain approach to calculate the WCET of multicore processors with shared instruction caches. The worst-case inter-thread cache interferences and the WCET were cal-
The rest of this paper is organized as follows. First, Section II briefly describes the CCCG-based approach and identifies its computational complexity as exponential. Then, Section III introduces the abstract interpretation-based techniques to reduce the computational complexity of the CCCG by predicting the cache behavior and excluding the infeasible paths. We present three counter-based approaches to bound the WCET of multicore processors in Section IV. Section V describes the evaluation methodology, and Section VI gives the experimental results. Finally, we draw conclusions in Section VII.

II. THE CCCG-BASED APPROACH

The CCCG-based approach [6, 7] is based on the implicit path enumeration technique (IPET) proposed by Li et al. [11], and it extends the cache conflict graph (CCG) of the IPET into the CCCG in order to bound the worst-case cache interferences in a shared cache of a multicore processor.

A. IPET

The IPET calculates the WCET by solving an ILP
The WCET of a thread running on a multicore processor with a shared L2 cache can be computed by maximizing the objective function shown in Eq. (1). More specifically, the WCET is defined as the sum of the timing cost of each basic block and the timing cost of accesses to each cache line block from both L1 and L2 caches. The timing cost of a basic block equals its execution latency multiplied by its execution count (i.e., the maximal number of times this basic block is executed). The timing cost of a cache line block is the sum of the latencies of a cache miss or hit at the corresponding level multiplied by the corresponding number of accesses to this cache line block. The symbols used in this paper are explained in Table 1.

\[
WCET = \sum c_i x_i + \sum (c_{i, \text{hit}}^{1, \text{hit}} x_{i, \text{hit}}^{1, \text{hit}} + c_{i, \text{miss}}^{1, \text{miss}} x_{i, \text{miss}}^{1, \text{miss}}) + \sum (c_{j, \text{hit}}^{2, \text{hit}} x_{j, \text{hit}}^{2, \text{hit}} + c_{j, \text{miss}}^{2, \text{miss}} x_{j, \text{miss}}^{2, \text{miss}})
\]

(1)

The structural constraints are described following Eq. (2). Inequality (3) is the functionality constraint, which states that the upper bound of the execution count of a loop header block is equal to the product of the execution counts of the pre-header block and the weight of this loop. Eq. (4) describes the relationship between basic block \(B_i\) and L1 cache line block \(L_j\) contained in \(B_i\). The cache constraints are demonstrated in Eqs. (5) to (9), where \(l_j\) and \(l_p\) are the execution counts of cache line blocks on the L1 cache and the L2 cache, respectively. For example, Eq. (5) indicates that the misses of an L1 cache line block are bounded by the execution count of this cache line block. Note that Eqs. (6) to (9) basically give the upper bound of the number of misses for a cache line block. They are derived from the CCG, which will be explained in Section II-C.

\[
\sum d_{\text{in}} = \sum d_{\text{out}} = x_i
\]

(2)

\[
x_i \leq \text{loop_weight} \times x_{\text{pre_header}}
\]

(3)

\[
x_i = l_j
\]

(4)

\[
l_j = l_j^{1, \text{hit}} + l_j^{1, \text{miss}}
\]

(5)

\[
l_j^{1, \text{miss}} \leq \sum l_j^{\text{miss}}
\]

(6)

\[
\sum l_j^{1, \text{miss}} = l_p
\]

(7)

\[
l_p = l_p^{2, \text{hit}} + l_p^{2, \text{miss}}
\]

(8)

\[
l_p^{2, \text{miss}} \leq \sum l_p^{\text{miss}}
\]

(9)

### C. Combined Cache Conflict Graph

A CCG is created for each cache set in IPET. The vertices in the CCG are the cache line blocks mapped into the same cache set, and the edges describe the valid transitions between the cache line blocks according to the control flow of the thread being analyzed. However, the CCG can only represent the cache states and related transitions caused by cache line blocks from one thread. It
needs to be enhanced in the following three regards to be able to model the behavior of a shared cache in a multicore processor. First, it should be capable of memorizing the historical cache accesses from all the threads. Second, it must represent the change of the cache state caused by a cache access from any thread. Last, it should be able to derive the cache states without violating the control flow and program semantics of any thread to obtain the WCET safely.

A CCCG significantly extends CCGs to satisfy the three requirements mentioned above for supporting timing analysis of a shared cache in a multicore processor. In a CCCG, each vertex represents a comprehensive cache state and is denoted by a tuple \( r(x,c) \) consisting of two sub-tuples \( r \) and \( c \). The sub-tuple \( r(l_1,l_2,\ldots,l_m) \) records the most recent access from each thread, where \( l_i \) is the most recent cache access from thread \( T_i \), and \( m \) is the number of threads. The sub-tuple \( c(x_1,x_2,\ldots,x_n) \) represents a cache state, where \( x_i \) represents the line block in cache way \( i \) and \( n \) is the associativity of the cache being modeled. The construction of the CCCG for a cache set begins with building the CCGs of this cache set for all the threads. The first state in the CCCG is initialized as \((s_1, s_2,\ldots,s_n; \phi_1, \phi_2,\ldots,\phi_n)\), where \( s_i \) is the starting node in the CCG of the cache way \( i \). If the \( i \)-th way of the cache set is empty. Other cache states are derived by traversing all the paths from the starting vertices to the end vertices of all the CCGs. For a cache state \( t_{curr} \) already existing in the CCCG, if the current cache access \( l_{ij} \) of a thread \( T_i \) has an outgoing edge \( d_i \) leading to another cache access \( l_{jk} \) in its CCG, a new cache state \( t_{next} \) is created by updating both sub-tuples \( r \) and \( c \) in \( t_{curr} \) with \( l_{ij} \) and \( l_{jk} \). For example, if \( t_{curr} \) is represented as \( r(l_1, l_2,\ldots,l_m;x_1,\ldots,x_n) \), \( t_{next} \) can be represented as \( r(l_1,\ldots,l_m;x_1,\ldots,x_{i'},\ldots,x_{i''},\ldots,x_n) \). The update of the sub-tuple \( c \) can be implemented with different cache replacement policies, such as FIFO or LRU. Also, an edge is added from \( t_{curr} \) to \( t_{next} \) to indicate this transition. It should be mentioned that the creation of \( t_{next} \) must check its upper-level cache state in the case of an L2 cache. If the transition leads to an L1 cache hit, \( t_{next} \) is simply inherited from \( t_{curr} \).

Since the CCCG describes the cache behavior on a shared cache caused by the cache accesses from all the co-running threads, it can produce the constraints to bound the worst-case cache access latency for a given thread by considering all the possible inter-thread cache interferences from other threads. The constraints generated from the CCCG are as follows.

Eq. (10) indicates that the sum of the execution counts of all the entry edges of the CCCG should be equal to 1. The structural constraint and the functionality constraint of the CCCG are shown in Eq. (11) and Inequality (13), respectively. Eqs. (12), (6), and (9) in Section II-B describe the constraints of the relationship between the tuple in the CCCG and the line blocks involved in this tuple. Additionally, the constraints to bound the number of cache hits of the tuple in the CCCG are described in Eqs. (14) and (15).

\[
\sum e_{entry} = 1 \tag{10}
\]
\[
\sum e_{in} = \sum e_{out} = t_i \tag{11}
\]
\[
\sum t_j = l_i \tag{12}
\]
\[
t_i \leq \text{loop_weight} \times \sum e_{in} \tag{13}
\]
\[
l_i^\text{hit} \leq \sum e_{hit} \tag{14}
\]
\[
t_i = l_i^\text{hit} + l_i^\text{miss} \tag{15}
\]

### D. An Algorithm to Build the CCCG Automatically

For simplicity, we assume two threads \( T_1 \) and \( T_2 \) are running on a dual-core processor with a shared 2-way set-associative cache. Thus, the tuple \( t \) in our CCCG approach can be initialized as \((S_1, S_2 : x, x)\), 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. For an L1 cache, the right side records the most recent access to the most recent cache line.
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 \textit{out_edge} leading to the next address access, then the next cache set state is generated. The new cache set state is initially inherited from \( t \). To update the new cache set state, the algorithm first checks whether or not the upper-level cache leads to a cache miss. If a cache miss happens 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 \( t_{\text{next}} \) to derive a new state. The termination condition of the algorithm is that the last accesses from all threads have no \textit{out_edge} to go to, which indicates that the last access has reached the end of the CCG in each cache line.

### E. An Example of Using CCCGs to Compute WCET

Fig. 1 gives an example to illustrate how to apply the CCCG for the WCET analysis of parallel programs running on a multicore processor with a shared cache. For simplicity, we assume that there are two threads, a real-time thread (RT) and a non-real-time thread (NRT)—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 2-way set-associative with one set. Thus, \( a \), \( b \), and \( c \) will all be mapped to the same cache line. Also, we assume the cache hit latency is 1 cycle and that the cache miss latency is 100 cycles. Fig. 1(a) shows the control flow graphs (CFGs) for the RT and NRT, while Fig. 1(b) depicts the CCGs for the RT and NRT, and Fig. 1(c) shows the CCCG, which is constructed automatically by applying Algorithm 1 to the CFGs and CCGs of both threads.

#### 1) Objective Function

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

\[
WCET = 100 \times a^{\text{miss}} + a^{\text{hit}}
\]  

#### 2) Structural Constraints

Structural constraints are derived from Eq. (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

---

**Fig. 1.** An example of applying combined cache conflict graphs (CCCGs) to estimate the worst-case execution time. (a) Control flow graph, (b) cache conflict graph, and (c) CCCG. RT: real-time thread, NRT: non-real-time thread.
edge starts from the instruction $a$ to the end of the program (i.e., $e$).
\[
a - ds_a - da_a = 0 \quad (17)
\]
\[
a - da_a - da_e = 0 \quad (18)
\]

Also, we can construct structural constraints for the NRT as follows.
\[
b - ds_b = 0 \quad (19)
\]
\[
b - db_c = 0 \quad (20)
\]
\[
c - db_c = 0 \quad (21)
\]
\[
c - de_e = 0 \quad (22)
\]

3) Functionality Constraints
For each thread, we assume that each of them executes only once.
\[
ds_a = 1 \quad (23)
\]
\[
ds_b = 1 \quad (24)
\]

We also assume that the loop in the RT thread executes no more than 10 iterations, which is based on Eq. (3).
\[
a - 10ds_a <= 0 \quad (25)
\]

4) Connection Constraints
Cache constraints are obtained from Fig. 1(c).

The following equation describes the constraints, i.e., the sum of instruction $a$’s different states is equal to the execution counts of instruction $a$. This is derived from Eq. (5).
\[
a - a^{hit} - a^{miss} = 0 \quad (26)
\]

5) CCCG Flow Constraints
The following equations specify the flow constraint that the sum of in-flow and out-flow must be equal to the execution counts of each node, which are derived from Eq. (10).
\[
n1 - e1 - e2 = 0 \quad (27)
\]
\[
n2 - e1 - e3 = 0 \quad (28)
\]
\[
n2 - e3 - e4 - e5 = 0 \quad (29)
\]
\[
n3 - e4 = 0 \quad (30)
\]
\[
n3 - e8 = 0 \quad (31)
\]
\[
n4 - e5 = 0 \quad (32)
\]
\[
n4 - e10 - e11 - e9 = 0 \quad (33)
\]
\[
n5 - e2 = 0 \quad (34)
\]
\[
n5 - e6 - e7 = 0 \quad (35)
\]
\[
n6 - e8 - e10 = 0 \quad (36)
\]
\[
n6 - e17 = 0 \quad (37)
\]
\[
n7 - e11 = 0 \quad (38)
\]
\[
n7 - e19 - e20 - e18 = 0 \quad (39)
\]
\[
n8 - e9 - e6 - e12 = 0 \quad (40)
\]
\[
n8 - e13 - e14 - e12 = 0 \quad (41)
\]

6) CCCG Entry Edge Constraints
The RT thread executes only once, which leads to the following equation based on Eq. (10).
\[
e1 + e6 + e15 + e28 = 1 \quad (67)
\]

Similarly, the NRT also executes only once, leading to the following equation.
\[
e2 + e5 + e8 = 1 \quad (68)
\]

7) CCCG Node Constraints
The sum of all the nodes that execute instruction $a$ must be equal to the execution counts of instruction $a$, which leads to the following equation derived from Eq. (12).
\[
n2 + n8 + n15 + n20 - a = 0 \quad (69)
\]

Similarly, the sum of all the nodes that execute instruction $b$ also must be equal to the execution counts of instruction $b$, which leads to the following equation.
\[
n4 + n5 + n6 - b = 0 \quad (70)
\]

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.
\[
n7 + n9 + n11 + n12 + n18 - c = 0 \quad (71)
\]
8) Hit Edge Constraints

The following equations are derived from Eq. (13).

\[ n_2 - 10e1 \leq 0 \]  
\[ n_8 - 10e9 - 10e6 \leq 0 \]  
\[ n_{15} - 10e18 - 10e15 - 10e22 \leq 0 \]  
\[ n_{20} - 10e30 - 10e27 - 10e28 - 10e33 \leq 0 \]

9) 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) to bound the number of cache hits.

\[ a_{hit}^a - e_3 - e_9 - e_{12} - e_{22} - e_{25} - e_{27} - e_{33} - e_{36} \geq 0 \]

10) Put Them All Together

Fig. 2 shows the WCET path for the example. The final result from ILP (i.e., the WCET) is 208, which can be derived from Eq. (77).

\[ 100 \times a^{miss} + a^{hit} = 100 \times 2 + 8 = 208 \]  

F. The Computational Complexity of the CCCG-Based Approach

As can be seen from the example shown in Fig. 1, the CCCG is very complicated, even for very simple code. The complexity of the CCCG is determined by the number of tuples in it. The increase of the number of tuples not only increases the time to build the CCCG, but also increases the timing complexity of deriving the constraints. As mentioned in Section II-C, the tuple records the most recent cache accesses from the CCGs of all the threads, and stores the cache states caused by these cache accesses. Since there is no fixed timing order between the cache accesses from different threads, and different orders of accesses to the same set of line blocks may result in different states of a cache set, a tuple can be considered as a permutation of the line blocks from the CCGs of all the threads.

Without losing generality, we assume that there are \( N \) threads running on a multicore processor with a shared cache. The size of the shared cache is \( C \); the cache block size is \( B \); and the associativity is \( S \). There are \( M \) connected line blocks in the CCG of each thread for a given cache set—\( M \) is the minimal number of the cache line blocks of any thread, if different threads have different numbers of cache line blocks in a cache set. In order to create a tuple for the CCCG of this cache set, one line block is selected from each CCG first, resulting in \( M^N \) possibilities. Then, the selected line blocks are fully permuted, leading to \( N! \) permutations. As a result, the worst-case number of tuples in the CCCG is \( M^N \times N! \). Hence the computational complexity of building a CCCG
is $O(M^t \times N^t)$, and the worst-case timing complexity of generating the constraints is also $O(M^t \times N^t)$ if all the tuples are traversed. As the cardinality of the cache set is equal to $C/(B \times S)$ and a CCCG is created for each cache set, the computational complexity of the CCCG-based approach is $O(C/(B \times S) \times (M^t \times N^t))$. It is clear that the computational complexity increases exponentially with increase in the number of the threads, making it prohibitive to compute the WCET of a thread running on a multicore processor with a large number of cores.

**III. REDUCING THE COMPLEXITY OF CCCG**

To decrease the computational complexity of the CCCG-based approach, we propose excluding the cache line blocks that will not impact the worst-case cache analysis from a CCCG. Specifically, there are two types of line blocks that can be excluded from the CCCG:

1. The accesses to the cache line blocks that always result in misses even without the inter-thread cache interferences, because they cannot be worsened by inter-thread cache interferences.
2. The cache line blocks on infeasible paths of a thread.

We use the cache analysis method based on abstract interpretation proposed by Theiling and Ferdinand [8] to detect the first type of cache line blocks. It applies the abstract interpretation framework to describe the cache semantics, including the control flow representation, abstract cache state, and abstract cache update function. Three types of analyses are used to categorize each memory reference. The **may analysis** is used to detect cache line blocks that are guaranteed to be absent in the cache, to which the accesses always result in cache misses.

Gustafsson et al. [9] proposed a method of abstract execution to automatically detect infeasible paths. Abstract execution is also based on abstract interpretation. It executes the program in the abstract domain, with abstract values for the program variables and the abstract versions of the operators. The method used to detect infeasible paths is a variation of abstract execution. It includes specific recorders, collectors, update points, update operators, and abstract state transition functions. This method finds sequences of nodes that are never executed during the same iteration of an abstract domain. The line blocks that are detected to be on the infeasible paths are excluded from the building of the CCCG.

These approaches are expected to reduce the computational complexity of the CCCG. Table 2 in Section VI-D demonstrates that they do in fact reduce the running time of all the experiments by 38.1% on average compared to the original CCCG-based approach. However, the theoretical computational complexity of generating a CCCG is still exponential because the mechanism of building a CCCG is not changed. The CCCG-based approach is thus practically infeasible for obtaining the WCET of large programs running on multicores with a large number of cores, even with the exclusion of unnecessary cache line blocks from the CCCG. Therefore, it is necessary to explore a new approach, which not only has much less complexity than the CCCG-based approach, but also computes the WCET safely with an accuracy that is ideally the same as or very close to that of the CCCG-based approach.

**IV. THE COUNTER-BASED APPROACHES**

**A. Motivation**

To find an efficient, scalable WCET analysis approach for shared caches of multicore processors, we have extensively studied the cache access and interference behavior of concurrent threads. We observe that with an increased number of cores (and threads), it becomes very likely that the conflicting cache accesses from different threads exceed the number of set associativity, which is referred to as interference saturation in this paper. In this case, we also observe that it is very likely for the concurrent threads to interleave their accesses in a way to result in cache misses, i.e., the worst-case scenario. This motivates us to develop a simple yet effective approach by counting the number of cache line blocks mapped to each cache set across all the co-running threads, based on which we can derive a safe upper bound of the worst-case shared cache performance.

To study the cache behavior based on the relation

---

**Table 2.** Comparison of the running time among the original CCCG-based approach, the CCCG (after removing always misses and accesses from infeasible paths), BC, EC, and HC schemes on a 4-core processor

<table>
<thead>
<tr>
<th>No.</th>
<th>Original CCCG</th>
<th>CCCG</th>
<th>BC</th>
<th>EC</th>
<th>HC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>206657</td>
<td>119063</td>
<td>453</td>
<td>468</td>
<td>1156</td>
</tr>
<tr>
<td>2</td>
<td>374512</td>
<td>226410</td>
<td>832</td>
<td>1045</td>
<td>2108</td>
</tr>
<tr>
<td>3</td>
<td>552862</td>
<td>379184</td>
<td>993</td>
<td>1042</td>
<td>2343</td>
</tr>
<tr>
<td>4</td>
<td>639038</td>
<td>340913</td>
<td>570</td>
<td>560</td>
<td>1983</td>
</tr>
<tr>
<td>5</td>
<td>417417</td>
<td>239705</td>
<td>860</td>
<td>876</td>
<td>1578</td>
</tr>
<tr>
<td>6</td>
<td>450913</td>
<td>304902</td>
<td>515</td>
<td>525</td>
<td>1672</td>
</tr>
<tr>
<td>7</td>
<td>429111</td>
<td>322551</td>
<td>562</td>
<td>577</td>
<td>2436</td>
</tr>
<tr>
<td>Average</td>
<td>438630</td>
<td>276104</td>
<td>684</td>
<td>726</td>
<td>1897</td>
</tr>
</tbody>
</table>

between the associativity of a cache set and the number of line blocks mapped into it from all the co-running threads, we conduct some experiments in a simulated multicore environment by extending SimpleScalar [12]. The memory hierarchy of the simulated environment and the benchmarks used in the experiments are the same as the experimental framework described in Section V-A.

The experiments examine the interference-saturation-miss probability, which is defined in Definition 1. The experiments are conducted on a 2-core processor, a 4-core processor, and an 8-core processor. Fig. 3 demonstrates the averaged interference-saturation-miss probability in all experiments of 2 cores, 4 cores, and 8 cores. We observe that the probabilities of 4 cores and 8 cores are much higher than that of 2 cores. For example, the probability for the benchmark select is only 35.25% in the case of 2 cores, but it rises to 83.5% for 4 cores, and 90.54% for 8 cores. The last three columns of Fig. 3 show the averaged results for all the benchmarks. The averaged interference-saturation-miss probability of all the benchmarks is almost 90% for 8 cores, and more than 80% for 4 cores.

**Definition 1.** Interference-Saturation-Miss Probability: The probability of the accesses to a cache line block of a shared L2 cache resulting in cache misses if the set associativity is fewer than the total number of cache line blocks assessed by all the concurrent threads.

It is important to note that the results in Fig. 3 are just simulated results based on particular inputs, since it is generally very difficult to exhaust all inputs to derive the actual worst-case cache behavior, if not possible. However, it should be noted that the worst-case interference-saturation-miss probability must be no better than what we have observed in Fig. 3. On the other hand, if the number of cache line blocks mapped to a cache set across all the threads is fewer than the set associativity, then the references to these cache line blocks can only result in cold misses in the worst case, which can be easily identified statically. These observations motivate us to develop a more efficient counter-based approach to estimate the worst-case inter-thread L2 cache misses safely and accurately.

### B. A Basic Counter-Based Approach

We propose a counter-based approach to efficiently estimate the worst-case cache behavior of a multicore processor with a shared cache by exploiting the fact that the interference-saturation-miss probability is very high, especially when the number of cores is large. The basic idea of the counter-based approach is to compare the associativity of a cache set with the number of line blocks mapped into it across all the concurrent threads. We associate a counter with each cache set to count the number of cache line blocks mapped to it across all the concurrent threads. The worst-case cache behavior can then be computed according to the following rules.

1. **Rule I:** If the number in the counter is larger than the associativity of a cache set, the references to any cache line block in this cache set are classified as first miss [13], which indicates that only the first reference to a cache line block is treated as a cache miss (i.e., cold miss), and all remaining references will result in cache hits.

   \[ \sum l_j^{1\text{-miss}} = l_p^{\text{miss}} \]  
   \[ \sum l_j^{1\text{-miss}} - 1 = l_p^{\text{hit}} \]  

2. **Rule II:** If the number in the counter is smaller than or equal to the associativity of a cache set, the references to the line blocks in this cache set are classified as first miss [13], which indicates that only the first reference to a cache line block is treated as a cache miss (i.e., cold miss), and all remaining references will result in cache hits.

This approach is referred to as the basic counter-based approach. Clearly, this approach can safely estimate an upper bound of the shared cache performance, because the interference-saturation-miss probability cannot be more than 100%, which has been conservatively assumed in Rule I. In Rule II, based on the assumption, the number of cache accesses that may lead to inter-core interferences can actually fit into the cache set; therefore, they can only suffer from cold misses, which is already the worst-case scenario.

The main procedure to implement the counter-based approach is almost the same as the IPET, except that the CCG and related cache constraints are only built for the higher-level separated cache, i.e., the private L1 cache. The cache constraints on the shared cache are described in two equations by modifying Eq. (7). Specifically, Eq. (78) represents the cache constraint of the line blocks classified as always miss, and Eq. (79) describes the cache constraint of the line blocks classified as first miss.

\[ \sum l_j^{1\text{-miss}} = l_p^{\text{miss}} \]  
\[ \sum l_j^{1\text{-miss}} - 1 = l_p^{\text{hit}} \]
Complexity Analysis: We estimate the computational complexity of the basic counter-based approach by following the assumptions made in Section II-F. As the counter for a cache set is calculated by checking all the cache line blocks in the CCGs for this cache set from all the threads, the complexity of calculating one counter is equal to \(O(N \times M)\). Hence, the complexity of calculating all the counters is \(O(C(B \times S) \times (N \times M))\), because each cache set has a counter. Furthermore, the complexity of the classification of the worst-case cache behavior based on the counter is \(O(C(B \times S) \times (N \times M))\). As a result, the computational complexity of the basic counter-based approach is \(O(C(B \times S) \times (N \times M))\), which has linear relationship with the number of cores or the number of cache accesses.

C. An Enhanced Counter-Based Approach

While the basic counter-based approach can greatly reduce the complexity of analysis, it may lead to overestimation. There are at least two factors that may lead to an overestimated WCET. First, the cache line blocks on the infeasible paths of each thread should not be taken into account. Second, the line blocks on the mutually exclusive paths in a thread should not be calculated in the same counter, because they cannot happen at the same time. We thus propose an enhanced approach to solve these problems. To address the first problem, we use the abstract execution proposed by Gustafsson et al. [9] to detect and eliminate the infeasible paths of all the threads before the calculation of the counter starts. To solve the second problem, we use data flow information extracted by the compiler to detect mutually exclusive paths, and the cache line blocks accessed by instructions from those paths are only counted once to derive the worst case. The enhanced counter-based approach consists of four major steps:

1. Construct a CCG for each thread on a given cache set.
2. Detect and eliminate the back-edges of the loops in each CCG to transfer the CCG into a direct acyclic graph.
3. Find the longest path from the starting vertex to the ending vertex in each CCG by using the dynamic programming algorithm, and then calculate the number of cache line blocks on the longest path as the counter of the corresponding thread for the given cache set.
4. Add up the counters of all the threads, and the result is the final counter for the given cache set.

Fig. 4 illustrates an example of enhancing the calculation of the counter. Fig. 4a is a CCG of a thread on a given cache set. It contains three line blocks and a loop involving the line block \(ln2\). In the basic counter-based approach, the counter should be equal to 3. Following step 2 in the enhanced counter-based approach, the back-edge of the loop from \(ln2\) to \(ln2\) is detected and eliminated. The resulting CCG is shown in Fig. 4b.

It is clear that the longest path from the start vertex to the end vertex is \(start \ln2 \ln3 \text{end}\), as marked by the dotted line, and there are only two line blocks on this path. Therefore, the counter of the thread on the cache set is 2, not 3.

The computational complexity of the enhanced approach counter-based approach is equal to the computational time complexity of the basic counter-based approach plus the computational complexity of finding a longest path in each CCG, which has a linear relationship with the number of vertices of the CCG.

D. A Hybrid Approach

Although the shared cache is accessed by all the threads running in a multicore processor, not all the cache sets are accessed by all the threads. It is possible that a given cache set is accessed by only one or two threads. The computational complexity of building the CCCG in this case is small and acceptable. Moreover, if a cache set is accessed by only one or two threads, the interference-saturation-miss probability is likely to become smaller, so applying the counter-based approach in this case may lead to the overestimation of cache misses.

We propose a hybrid approach by combining the enhanced counter-based approach and the CCCG-based approach. Specifically, the cache sets in a shared L2 cache are first classified into two categories: one consisting of the cache sets accessed by more than 2 threads, and the other including the cache sets accessed by up to 2 threads. For the first type of cache sets, the counter-based approach is still used to estimate the worst-case cache behavior. However, for the second type of cache sets, the CCCG-based approach is used to bound the worst-case cache misses more accurately, because the CCCG-based approach can still run efficiently with only two threads, and the computational complexity is increased exponentially and hard to afford with a large number of threads.

![Fig. 4. The examples of enhancing the calculation of the counter. (a) Original cache conflict graph (CCG) and (b) CCG without the loop.](http://dx.doi.org/10.5626/JCSE.2013.7.4.285)
As the computational complexity of the hybrid counter-based approach is dominated by the cache sets applied to the CCCG-based approach, the computational complexity of the hybrid counter-based approach is $O(C/B \times S) \times M^2$ with $N = 2$, following the assumptions and notations used in Section II-F.

V. EVALUATION METHODOLOGY

A. Experimental Framework

The experimental framework of our multicore WCET analysis approach consists of three main components: the front end, the back end, and the ILP solver. The front end compiles the benchmarks to be analyzed into COFF format binary code by the GCC compiler, which is translated into MIPS ISA. Then, it obtains the global CFG and related information of the instructions by disassembling the binary code generated by the compiler. The back end performs static cache analysis [14] by using the three counter-based approaches. A commercial ILP solver—CPLEX [15]—is used to solve the ILP problem to compute the WCET.

Our experiments are mainly focused on a 4-core processor with a shared L2 cache simulated by an extended SimpleScalar simulator. In this quad-core processor, each core has its own L1 instruction cache and L1 data cache. The L1 instruction cache is directed-mapped, its total size and cache block size are 512 bytes and 8 bytes, respectively, and its miss penalty is 4 cycles. The L1 data cache is assumed to be perfect, since our evaluation focuses on the analysis of the cache behavior of the instructions. The shared L2 cache is 2-way set-associative, and its size is 2048 bytes. The block size of the shared L2 cache is 16 bytes, and its miss penalty is 100 cycles.

The benchmarks are from Mälardalen WCET benchmarks [16], whose salient characteristics are described in Table 3. Some benchmarks are selected for each experiment. Each benchmark is regarded as a thread running on a core of the multicore processor.

B. Scheme Studied

In our experiments, we studied four different schemes based on different WCET analysis approaches as follows:

- The CCCG scheme: The CCCG-based approach [6, 7] is used to calculate the WCET in this scheme. Two techniques based on abstract interpretation mentioned in Section III are used to reduce the computational complexity of the original CCCG-based approach.
- The basic counter (BC) scheme: The basic counter-based approach described in Section IV-B is used to calculate WCET in this scheme.
- The enhanced counter (EC) scheme: The enhanced counter-based approach described in Section IV-C is used to calculate WCET in this scheme.
- The hybrid counter (HC) scheme: The hybrid counter-based approach described in Section IV-D is used to calculate WCET in this scheme.

VI. EXPERIMENTAL RESULTS

We have conducted seven experiments to compare the WCET and the worst-case L2 cache misses among all the four schemes on the 4-core processor. Each benchmark of a group is selected from the list of benchmarks shown in Table 3. Table 4 compares the worst-case L2 cache misses and the WCET of the benchmarks of the BC scheme, the EC scheme, and the HC scheme with the CCCG scheme. The last three columns give the ratios of the WCETs of three counter-based schemes to the WCET of the CCCG scheme.

A. Comparing BC with CCCG

As shown in Table 4, the WCETs of the BC scheme are much larger than that of the CCCG scheme. For example,
the estimated WCETs of the benchmark \texttt{qsort} of the BC scheme are larger than those of the CCCG scheme by 59.1%, 41.2%, and 40.1% in Experiments 2, 4, and 6, respectively. On average, the estimated WCET of the BC scheme is 29.7% larger than that of the CCCG scheme.

The difference between the BC scheme and the CCCG scheme is caused by the overestimation of the worst-case L2 cache misses from the basic counter-based approach. First, some cache line blocks on the infeasible paths or mutually-exclusive paths are counted unnecessarily. If the value of the overestimated counter is larger than the associativity of this cache set, some cache accesses to this cache set are estimated as cache misses by mistake. Second, if the line blocks mapped into a cache set are from only one or two threads, the probability of an inter-thread L2 cache interference is very low, if not zero, because a thread is more likely to reuse its cache content (i.e., hits) if there are few interferences from another thread, or if there are none at all. So, applying the basic counter-based approach will possibly lead to the overestimation of L2 cache misses than the CCCG-based approach.

B. Comparing EC with CCCG

As demonstrated in Table 4, the enhanced counter-based approach always returns a tighter number of esti-

<table>
<thead>
<tr>
<th>No.</th>
<th>Benchmark</th>
<th>CCCG L2 miss WCET</th>
<th>BC L2 miss WCET</th>
<th>EC L2 miss WCET</th>
<th>HC L2 miss WCET</th>
<th>BC WCET ratio</th>
<th>EC WCET ratio</th>
<th>HC WCET ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Insertsort</td>
<td>11 1,762</td>
<td>16 2,262</td>
<td>14 2,062</td>
<td>11 1,762</td>
<td>1.28</td>
<td>1.17</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Cover</td>
<td>13 2,545</td>
<td>16 2,845</td>
<td>15 2,745</td>
<td>13 2,545</td>
<td>1.12</td>
<td>1.08</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Jfdctint</td>
<td>40 6,093</td>
<td>56 8,093</td>
<td>52 7,293</td>
<td>44 6,493</td>
<td>1.33</td>
<td>1.20</td>
<td>1.07</td>
</tr>
<tr>
<td></td>
<td>Expint</td>
<td>17 29,518</td>
<td>21 29,922</td>
<td>19 29,721</td>
<td>17 29,518</td>
<td>1.02</td>
<td>1.01</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Jfdctint</td>
<td>46 6,693</td>
<td>57 7,793</td>
<td>54 7,293</td>
<td>48 6,893</td>
<td>1.16</td>
<td>1.12</td>
<td>1.03</td>
</tr>
<tr>
<td></td>
<td>Expint</td>
<td>16 29,418</td>
<td>21 29,922</td>
<td>18 29,619</td>
<td>16 21,419</td>
<td>1.02</td>
<td>1.01</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Qsort</td>
<td>29 4,735</td>
<td>57 7,535</td>
<td>44 6,235</td>
<td>33 5,135</td>
<td>1.59</td>
<td>1.32</td>
<td>1.09</td>
</tr>
<tr>
<td></td>
<td>Select</td>
<td>24 5,286</td>
<td>31 5,990</td>
<td>29 5,790</td>
<td>25 5,386</td>
<td>1.13</td>
<td>1.10</td>
<td>1.02</td>
</tr>
<tr>
<td>3</td>
<td>Select</td>
<td>24 5,286</td>
<td>31 5,990</td>
<td>28 5,690</td>
<td>24 5,286</td>
<td>1.13</td>
<td>1.08</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Ludcmp</td>
<td>53 7,905</td>
<td>83 10,905</td>
<td>69 9,505</td>
<td>55 8,105</td>
<td>1.38</td>
<td>1.20</td>
<td>1.03</td>
</tr>
<tr>
<td></td>
<td>Ud</td>
<td>44 7,109</td>
<td>62 8,914</td>
<td>53 8,014</td>
<td>44 7,109</td>
<td>1.25</td>
<td>1.13</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Minver</td>
<td>64 8,382</td>
<td>102 12,196</td>
<td>86 10,596</td>
<td>59 7,892</td>
<td>1.46</td>
<td>1.26</td>
<td>1.05</td>
</tr>
<tr>
<td>4</td>
<td>Cover</td>
<td>13 2,545</td>
<td>16 2,845</td>
<td>16 2,845</td>
<td>13 2,545</td>
<td>1.12</td>
<td>1.12</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>qsort</td>
<td>35 5,335</td>
<td>57 7,535</td>
<td>45 6,335</td>
<td>35 5,535</td>
<td>1.41</td>
<td>1.19</td>
<td>1.04</td>
</tr>
<tr>
<td></td>
<td>ludcmp</td>
<td>47 7,305</td>
<td>85 11,105</td>
<td>68 9,405</td>
<td>50 7,605</td>
<td>1.52</td>
<td>1.29</td>
<td>1.04</td>
</tr>
<tr>
<td></td>
<td>minver</td>
<td>55 7,492</td>
<td>102 12,196</td>
<td>83 10,294</td>
<td>59 7,892</td>
<td>1.63</td>
<td>1.31</td>
<td>1.05</td>
</tr>
<tr>
<td>5</td>
<td>insertsort</td>
<td>12 1,862</td>
<td>15 2,162</td>
<td>12 1,862</td>
<td>12 1,862</td>
<td>1.16</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>ud</td>
<td>42 6,914</td>
<td>62 8,914</td>
<td>50 7,714</td>
<td>45 7,214</td>
<td>1.29</td>
<td>1.12</td>
<td>1.04</td>
</tr>
<tr>
<td></td>
<td>select</td>
<td>25 5,390</td>
<td>31 5,990</td>
<td>27 5,590</td>
<td>25 5,390</td>
<td>1.11</td>
<td>1.04</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>jfdctint</td>
<td>51 7,193</td>
<td>66 8,693</td>
<td>60 8,093</td>
<td>54 7,493</td>
<td>1.21</td>
<td>1.13</td>
<td>1.04</td>
</tr>
<tr>
<td>6</td>
<td>insertsort</td>
<td>12 1,862</td>
<td>15 2,162</td>
<td>12 1,862</td>
<td>12 1,862</td>
<td>1.16</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>qsort</td>
<td>34 5,235</td>
<td>55 7,335</td>
<td>40 5,835</td>
<td>34 5,235</td>
<td>1.40</td>
<td>1.11</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>ludcmp</td>
<td>46 7,205</td>
<td>83 10,905</td>
<td>71 9,705</td>
<td>52 7,804</td>
<td>1.51</td>
<td>1.35</td>
<td>1.08</td>
</tr>
<tr>
<td></td>
<td>ud</td>
<td>40 6,714</td>
<td>62 8,914</td>
<td>45 7,214</td>
<td>42 6,914</td>
<td>1.33</td>
<td>1.07</td>
<td>1.03</td>
</tr>
<tr>
<td>7</td>
<td>cover</td>
<td>13 2,545</td>
<td>16 2,845</td>
<td>15 2,745</td>
<td>13 2,545</td>
<td>1.12</td>
<td>1.08</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>expint</td>
<td>18 29,622</td>
<td>21 29,922</td>
<td>20 29,821</td>
<td>18 29,622</td>
<td>1.02</td>
<td>1.01</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>minver</td>
<td>64 8,392</td>
<td>101 12,195</td>
<td>14 10,796</td>
<td>70 8,992</td>
<td>1.45</td>
<td>1.29</td>
<td>1.07</td>
</tr>
<tr>
<td></td>
<td>ud</td>
<td>42 6,914</td>
<td>62 8,914</td>
<td>54 8,114</td>
<td>45 7,214</td>
<td>1.29</td>
<td>1.17</td>
<td>1.04</td>
</tr>
</tbody>
</table>

Counter-Based Approaches for Efficient WCET Analysis of Multicore Processors with Shared Caches

Yiqiang Ding and Wei Zhang

mated L2 cache misses than the basic counter-based approach, leading to more accurate WCET. For example, the estimated WCET of `qsort` of the EC scheme in Experiment 2 is reduced by almost 30% compared with that of the BC scheme. On average, the estimated WCET of the EC scheme is only 14% larger than that of the CCCG scheme. The main reason is that although the enhanced counter-based approach tightens the estimated WCET by calculating the counter of the cache set more accurately, it still leads to overestimation of the L2 cache misses if a cache set only contains the line blocks from one or two threads, in which case, the \textit{interference-saturation-miss probability} is low.

\section*{C. Comparing HC with CCCG}

Table 4 also compares the worst-case L2 cache misses and the WCET between the HC scheme and the CCCG scheme. It is clear that the difference between the HC scheme and the CCCG scheme is even smaller, and the estimated WCETs of some benchmarks are actually the same in both schemes, such as \textit{insertsort}, \textit{cover}, and \textit{expint} in Experiment 1. The estimated WCET of the HC scheme is only 2.65% larger than that of the CCCG scheme on average, indicating that the HC scheme can predict the worst-case shared cache performance very tightly. However, the HC scheme is still not as accurate as the CCCG-based approach, because the worst-case \textit{interference-saturation-miss probability} may not be 100%, even if there are more than 2 threads.

\section*{D. The Running Time}

Table 2 compares the running time of all the experiments of the BC, EC, HC, and CCCG schemes, along with the original CCCG-based approach, which does not use any of the complexity-reduction techniques described in Section III. The averaged running time in the BC and EC schemes are 684 and 726 ms, respectively, while the averaged running time of the CCCG scheme is more than 2 \(\times 10^5\) ms. This demonstrates that the counter-based approaches significantly reduce the computational complexity of the multicore WCET analysis. As the hybrid counter-based approach applies the CCCG-based approach to a part of the cache sets, its averaged running time increases to 1897 ms. However, it is still much less than that of the CCCG-based approach. On average, the EC and HC schemes reduce the running time of the CCCG scheme by more than 380 and 150 times, respectively.

\section*{E. Associativity Analysis}

In order to study the impact of the set associativity of a shared L2 cache on the timing analysis, we varied the associativity of the shared L2 cache from 1 to 2 and 4. As shown in Fig. 5, the associativity results of 1 and 4 are similar to the associativity results of 2. Also, the difference between the EC and HC schemes becomes smaller when the associativity is increased from 1, 2 to 4. The differences are 13.27\%, 11.35\% to 10.59\%, respectively. The main reason is that the probability of a cache set containing line blocks from only one or two threads decreases with the increase of the set associativity.

\section*{VII. CONCLUSION}

Following the trend of adopting multicore for desktop and server computing, the performance improvement of future real-time embedded systems is likely to depend on the increase of the number of cores. Therefore, it becomes increasingly important to design an efficient and scalable timing analysis method to safely and accurately estimate the WCET for hard real-time tasks running on multicore processors with shared resources. While there have been several research efforts on multicore timing analysis \cite{1-4, 6}, they have either been limited to instruction caches only, or have significant computation cost that is not scalable.

This paper proposed three counter-based approaches to significantly reduce the complexity of WCET analysis for shared caches in multicore, while achieving accuracy close to the CCCG-based approach with guaranteed safety. All three of these approaches are based on our empirical observation that in a multicore processor, especially in multicore with more cores, it is very likely for the concurrent threads to interleave their accesses to a shared cache in a way that results in inter-core cache misses, \textit{i.e.}, the worst-case scenario. The basic counter-based approach uses a counter associated with each cache set to compare the worst-case number of line blocks mapped into a cache set with its associativity. It predicts the worst-case cache behavior of the line blocks in the cache set according to the comparison result. The enhanced
counter-based approach uses some techniques to enhance the accuracy of calculating the counters. Furthermore, a hybrid approach combines the enhanced counter-based approach and the CCCG-based approach to further improve the accuracy of the estimation.

Our experiments demonstrate that the enhanced counter-based approach overestimates the WCET by 14% on average compared to the CCCG-based approach on a 4-core processor, while its running time is less than 1/380 that of the CCCG-based approach on average. The hybrid counter-based approach reduces the overestimation to only 2.65%, while its averaged running time is less than 1/150 that of the CCCG-based approach.

In future work, we will extend the counter-based approaches to data cache analysis as well. Also, we plan to quantitatively evaluate the scalability of different counter-based approaches with more cores.

ACKNOWLEDGMENTS

This work was funded in part by NSF grant CCF 0914543.

REFERENCES

Wei Zhang

Dr. Wei Zhang is an associate professor (tenured) in Electrical and Computer Engineering of Virginia Commonwealth University. Dr. Wei Zhang received his Ph.D. from the Pennsylvania State University in 2003. From August 2003 to July 2010, Dr. Zhang worked as an assistant professor and then as an associate professor (tenured) at Southern Illinois University Carbondale. His research interests are in embedded and real-time computing systems, computer architecture, compiler, and low-power systems. Dr. Zhang has received the 2009 SIUC Excellence through Commitment Outstanding Scholar Award for the College of Engineering, and 2007 IBM Real-time Innovation Award. Dr. Zhang has received 5 research grants from the National Science Foundation. In addition, his research and educational efforts have been supported by industry including leading IT companies such as IBM, Intel, Motorola, and Altera. Dr. Zhang has published more than 100 papers in refereed journals and conference proceedings. He is a senior member of the IEEE, and an associate editor of the Journal of Computing Science and Engineering. He has served as a member of the organizing or program committees for several IEEE/ACM international conferences and workshops.

Yiqiang Ding

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