Multicore-Aware Code Co-Positioning to Reduce WCET on Dual-Core Processors with Shared Instruction 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
For real-time systems it is important to obtain the accurate worst-case execution time (WCET). Furthermore, how to improve the WCET of applications that run on multicore processors is both significant and challenging as the WCET can be largely affected by the possible inter-core interferences in shared resources such as the shared L2 cache. In order to solve this problem, we propose an innovative approach that adopts a code positioning method to reduce the inter-core L2 cache interferences between the different real-time threads that adaptively run in a multi-core processor by using different strategies. The worst-case-oriented strategy is designed to decrease the worst-case WCET among these threads to as low as possible. The other two strategies aim at reducing the WCET of each thread to almost equal percentage or amount. Our experiments indicate that the proposed multicore-aware code positioning approaches, not only improve the worst-case performance of the real-time threads but also make good tradeoffs between efficiency and fairness for threads that run on multicore platforms.

Category: Embedded computing

C3 [SPECIAL-PURPOSE AND APPLICATION-BASED SYSTEMS]: Real-time and embedded systems; J7 [COMPUTERS IN OTHER SYSTEMS]: Real time

Keywords: Worst-case execution time analysis; Multicore processors; Shared caches; Hard real-time

I. INTRODUCTION
Multicore processors have become the mainstream computing platform due to their advantages of high parallelism and low power consumption. Following the adoption of multicore processors in desktop and server domains, multicore processors have also been used in embedded systems. In particular, with an increase in the demand for high performance by high-end real-time applications such as HDTV and real-time multimedia processing applications, multi-core processors are expected to be increasingly used in real-time systems. In the near future, researchers have envisioned a possible deployment of real-time systems on large-scale multi-core processors that are composed of tens or even hundreds of cores on a single chip [1].

Real-time systems can be classified into hard real-time systems and soft real-time systems by the characteristics of the timing constraints. In hard real-time systems, it is critical to accurately obtain the worst-case execution time (WCET) for each task and this provides the basis of task scheduling. As an inaccurate WCET will lead to an inap-
propriate task scheduling paradigm of a real-time system, this can degrade the quality of service or even result in the failure of a system. In addition to estimating the WCET accurately, it is also desirable to optimize real-time code to reduce WCET. Better WCET of a task offers the real-time scheduler more flexibility to schedule this task for meeting its deadline. Moreover, among the other benefits, reducing WCET of a computing task can help to conserve the power that is used by the system [2]. The basic idea is that with WCET, information is available and if a task still has slacks then the clock rate can be lowered in order to reduce the power dissipation while at the same time meeting the deadlines.

In order to reduce the WCET of real-time tasks (i.e., to obtain “better” WCET), code positioning approaches have been proposed [3, 4]. However, current WCET-oriented code positioning approaches center on the enhancement of the WCET of single-threaded application on unprocessors. This cannot be effectively applied to multicore processors with shared caches. This is due to these code positioning algorithms [3, 4] which only reduce the intra-thread cache conflicts but cannot detect the inter-thread cache conflicts or avoid them. Furthermore, these approaches may reduce the intra-thread L1 cache misses at the cost of more inter-thread shared L2 cache misses, whose penalty is usually much more than that of an L1 cache miss. Thus it may hurt the overall performance. Therefore, it is crucial to develop multicore-aware code positioning techniques for real-time applications that run on multicore platforms.

In this paper, we assume that two real-time threads are running concurrently on a dual-core processor with a shared L2 cache. Our goal is to reduce the WCET of these threads (in some applications with mixed real-time and non-real-time tasks, a real-time thread may run concurrently with a non-real-time thread. However, it should be noted that code position for this scenario is actually less challenging, as the performance of the non-real-time thread can be sacrificed for the enhancement of the WCET of the real-time thread [5]. Whereas in this paper, the WCETs of both the real-time threads are to be considered.). We have studied three approaches: a worst-case-oriented (WCO) approach, and two fairness-oriented approaches, including the percentage-fairness-oriented (PFO) and amount-fairness-oriented (AFO) schemes. All these approaches are built upon multicore cache WCET analysis but with different optimization goals (preliminary work of this project was published in a conference paper [6]. In this paper, we have explored the effectiveness of the code co-positioning strategies with different cache sizes and memory latencies and they are likely to become important for future multicore/manycore processors). Our experiments show that all these three proposed techniques can effectively reduce the WCET for co-running real-time threads in order to achieve their respective optimization goals.

The rest of the paper is organized as follows. Section II reviews related work. Section III describes the proposed multicore-aware code positioning approaches. The evaluation methodology is explained in Section IV and the experimental results are presented in Section V. Finally, the conclusions are made in Section VI.

II. RELATED WORK

Code positioning is a software optimization that is done by the compiler to reorder or position the structural units (e.g., a basic block or a procedure) within a program. As some processors induce a pipeline delay that is related to the transfer of control, these delays are more common in embedded systems. They do not have branch prediction and target buffers to reduce the complexity of the processor. The compiler attempts to position the basic blocks to minimize the number of both the unconditional jumps and taken branches to reduce the pipeline delays. Furthermore, for the processors with instruction caches, code positioning can help to improve the instruction cache locality and reduce cache miss penalties. Traditional code positioning algorithms mostly aim at the enhancement of the average-case execution time (ACET) by reordering the basic blocks to make the most frequently traversed edges that are contiguous in memory [7-11]. However, as the most frequently traversed edges may not be a part of the worst-case paths, the WCET cannot be guaranteed to be reduced by these approaches. Even if the WCET path is taken into account by the code positioning algorithm, a change in the positioning may result in a different path which becomes the worst-case path.

In order to improve the worst-case performance in a processor with instruction caches, a code positioning approach is proposed. It is done to focus on positioning the basic blocks on the worst-case path in the program and to reduce the pipeline delay that is caused by the transfer of controls [3]. The main idea of this basic block positioning algorithm is to select the edges between the basic blocks on the worst-case path to be contiguous and this will minimize the WCET. Recently, another WCET-oriented approach is proposed to reduce the number of cache conflict misses by means of placing procedures that contribute to the WCET, so that they are mapped contiguously in memory layout to avoid the overlapping of cache lines belonging to a caller and a callee procedure [4]. However, both of these approaches have not considered the inter-thread cache conflicts in multicore computing platforms.

Cache partitioning is another useful method to isolate tasks in a multitasking real-time system to allow individual analysis of cache behavior. Thus, it enhances the time predictability of each task. There are mainly two types of cache partitioning approaches, i.e., hardware-based [12, 13] and software-based [14, 15]. In hardware-based cache
partitioning, address mapping hardware is inserted into the processor along with a cache to restrict cache accesses to a single contiguous cache segment at any one time. Therefore each task has the right to access a private cache segment for one or more partitions. In contrast, the software-based approach can provide more flexibility and possibly better performance over the hardware-based cache partitioning. It creates a private cache partitioning for each task by assigning it a separate address space in the cache with the use of a compiler and a linker. Thus, it can provide a large number of independent cache partitions at no additional cost with good flexibility. Our multicore-aware code positioning techniques proposed in this paper are complementary to the cache partitioning approaches. Multicore-aware code positioning enables different tasks to share caches still for the achievement of benefits such as efficient cache space usage, low-cost cache coherency and easy sharing [16] while minimizing the inter-core cache conflicts. Moreover, as a pure software-based technique, multicore-aware code positioning does not need to modify the hardware while achieving “better” WCET for real-time tasks that run on multicore processors.

III. OUR APPROACHES

A. Overview

In a multi-core processor, each core typically has private L1 instruction and data caches. The L2 (and/or L3) caches can be either shared or separated. While private L2 caches are more time-predictable that is there are no inter-core L2 cache conflicts, they suffer from other deficiencies. First, each core with a private L2 cache can only exploit separated and limited cache space. Due to the great impact of the L2 cache hit rate on the performance of multi-core processors, private L2 caches may have worse performance than a shared L2 cache with the same total size as each core with shared L2 cache may make use of the aggregate L2 cache space more effectively. Besides, separated L2 caches increase the cache synchronization and coherency cost [16]. Moreover, a shared L2 cache architecture makes it easier for multiple cooperative threads to share instructions and data. This becomes more expensive in separated L2 caches [16]. Therefore, in this paper, we will study the WCET analysis of multicore processors with shared L2 caches.

For simplicity, in this study, we assume that two real-time threads run concurrently on different cores of a dual-core processor with private L1 caches and a shared L2 cache but our techniques can be applied or adapted for multiple threads that run on multicore chips with multi-level memory hierarchies. We have proposed three strategies in order to optimize the WCET of both the threads for making different tradeoffs. These strategies include a WCO strategy and two fairness-oriented strategies that include both AFO and PFO. The WCO aims at the improvement of the performance of the real-time thread with the longest WCET, as this type of thread mostly impacts the performance of the whole system. By comparison, AFO and PFO attempt to treat all the real-time threads fairly, and their goals are to reduce the worst-case performance of each concurrent real-time thread by approximately the same amount or percentage, respectively.

Fig. 1 depicts the main working flow of the WCET-oriented co-optimization architecture and it mainly consists of two sub-flows. The sub-flows of both the threads are initialized with code analysis that includes control flow analysis and static cache analysis. The inter-thread cache conflict analysis algorithm calculates the worst-case inter-thread L2 cache conflict set. According to the inter-thread L2 cache conflict set, the programs of both the threads are positioned following the assigned strategy to reduce the inter-thread L2 cache conflicts for attaining different optimization goals. After the code positioning phase, the WCET analysis is conducted to find out the new WCETs for both threads. They are compared with their original WCETs for guiding the co-optimization. It is worthy to note that the flow of both threads from code analysis to code positioning may be repeated for several times for the achievement of optimal results.

B. Worst-Case-Oriented Code Positioning

The objective of WCO is to minimize the longest WCET of both the real-time threads (i.e., reducing the worst-case WCET of co-running threads), whose algorithm is described in Algorithm 1. The inputs of the algorithm are the two programs that are to be optimized. In line 2, the termination variable that finishes the repetitive
optimization process is initialized. In the next three lines, fundamental data that are required by the algorithm are calculated for both the programs. This includes the original WCETs for both the programs and the L2 cache conflict instruction list. After the completion of the initialization phase, the original WCETs of both the programs are compared with each other in the core of the algorithm from line 6 to 26. Then, the program with a smaller original WCET will be positioned for the optimization of the WCET of the other program maximally. As shown from line 7 to line 15, in case that the original WCET of P1 is larger than that of P2, program P2 will be positioned at line 8, in which the conflict instructions from P2 that lead to the largest inter-thread cache conflicts will be allocated at new memory addresses. These memory addresses map to L2 cache blocks with the minimal conflicts and the corresponding instructions from P1. It should be noted that in a virtual memory system, this memory address remapping can be easily implemented by the updation of the virtual-to-physical address mapping information in the page table and TLB. The WCETs of both the programs will be analyzed again when the positioning of P2 finishes at lines 9-10. If the WCET of P1 is still larger than that of P2, then the termination variable will be assigned true, else the function of WC_Oriented_Code_Positioning will be executed recursively for the reduction of the WCET of P2. This now becomes the thread with the longest WCET. In the other case (line 16 to 25), the positioned program turns to be P1 as the original WCET of P2 is larger than that of P1 and other steps are almost the same as the first case. Finally, the algorithm will not be terminated until the value of the termination variable equals true.

C. Fairness-Oriented Code Positioning

While WCO focuses on the optimization of a single thread that has the worst WCET among the co-running threads, FO code positioning needs to optimize all the co-running threads to ensure fairness. As the WCETs of both threads may vary significantly, the “fairness” has different meanings and implications. This depends on the optimization objectives. In this work, the FO strategy is divided into two different schemes according to the “fairness” goals and it includes 1) reducing approximately the same amount of WCET, or 2) reducing approximately the same percentage of WCET. Accordingly, these two schemes are named AFO code positioning and PFO code positioning, respectively.

1) Amount-Fairness-Oriented Scheme

AFO code positioning algorithm aims at the reduction of the WCETs of both the co-running threads by approximately equal amount. When the WCO code positioning approach is applied, only the instructions of the thread with shorter (i.e. “better”) WCET are positioned to reduce the WCET of the other thread maximally. In this case, the amount of WCET is reduced by avoiding the inter-thread L2 cache conflicts which is the same to both threads. However, the difference in the amount of WCET reduction can be caused by different intra-thread L1 and L2 cache misses due to the WCO code positioning. Therefore, AFO can leverage WCO to decrease the inter-thread cache misses, while it tries to recover some of the positioned instructions in WCO. This is done by a process named code de-positioning in order to ensure that the intra-thread cache miss penalties of both threads are reduced by approximately the same amount.

The algorithm of AFO is demonstrated in Algorithm 2. The inputs and the initialization phase are the same as those of WCO. In line 6, the WCO algorithm is invoked to reduce the inter-thread L2 cache misses. In this algorithm, P2 is assumed to be the thread with a larger WCET. Therefore, only the instructions from P1 are positioned by WCO. In the core of this algorithm, some positioned instructions of P1 are recovered to their original positions by de-positioning at line 8. The corresponding instructions from P2 are positioned instead to avoid the inter-thread L2 cache conflicts at line 9. Then, the present WCETs of both programs are computed at lines 10 and 11.

!![Algorithm 1 WC_Oriented_Code_Positioning]

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>begin</td>
</tr>
<tr>
<td>2</td>
<td>boolean terminate = false;</td>
</tr>
<tr>
<td>3</td>
<td>P1_wcet = WCETAnalysis(P1);</td>
</tr>
<tr>
<td>4</td>
<td>P2_wcet = WCETAnalysis(P2);</td>
</tr>
<tr>
<td>5</td>
<td>Conflict_Op_List = Build_Conflict_Op_List(P1, P2);</td>
</tr>
<tr>
<td>6</td>
<td>repeat</td>
</tr>
<tr>
<td>7</td>
<td>if P1_wcet &gt; P2_wcet then</td>
</tr>
<tr>
<td>8</td>
<td>Positioning(P2, Conflict_Op_List);</td>
</tr>
<tr>
<td>9</td>
<td>P1_wcet = WCETAnalysis(P1);</td>
</tr>
<tr>
<td>10</td>
<td>P2_wcet = WCETAnalysis(P2);</td>
</tr>
<tr>
<td>11</td>
<td>if P1_wcet &gt; P2_wcet then</td>
</tr>
<tr>
<td>12</td>
<td>terminate = true;</td>
</tr>
<tr>
<td>13</td>
<td>else</td>
</tr>
<tr>
<td>14</td>
<td>WC_Oriented_Code_Positioning(P1, P2);</td>
</tr>
<tr>
<td>15</td>
<td>end if</td>
</tr>
<tr>
<td>16</td>
<td>else</td>
</tr>
<tr>
<td>17</td>
<td>Positioning(P1, Conflict_Op_List);</td>
</tr>
<tr>
<td>18</td>
<td>P1_wcet = WCETAnalysis(P1);</td>
</tr>
<tr>
<td>19</td>
<td>P2_wcet = WCETAnalysis(P2);</td>
</tr>
<tr>
<td>20</td>
<td>if P1_wcet &lt; P2_wcet then</td>
</tr>
<tr>
<td>21</td>
<td>terminate = true;</td>
</tr>
<tr>
<td>22</td>
<td>else</td>
</tr>
<tr>
<td>23</td>
<td>WC_Oriented_Code_Positioning(P1, P2);</td>
</tr>
<tr>
<td>24</td>
<td>end if</td>
</tr>
<tr>
<td>25</td>
<td>until terminate == true;</td>
</tr>
<tr>
<td>26</td>
<td>end</td>
</tr>
</tbody>
</table>
Algorithm 2 AF_Oriented_Code_Positioning

1: begin
2: boolean terminate = false;
3: Original_P1_wcet = WCET Analysis(P1);
4: Original_P2_wcet = WCET Analysis(P2);
5: Conflict_Op_List = Build_Conflict_Op_List(P1, P2);
6: WC_Original_Code_Positioning(P1, P2);
7: repeat
8:   De-positioning(P1, Conflict_Op_List);
9:   Positioning(P1, Conflict_Op_List);
10:  P1_wcet = WCET Analysis(P1);
11:  P2_wcet = WCET Analysis(P2);
12:  \( \Delta W = \text{Calculate Amount Variation}(\);
13:  if \( \Delta W \geq \Delta \text{Original } W \) then
14:     terminate = true;
15:  else
16:     Original_W = \( \Delta W \)
17: end if
18: until terminate == true;
19: end

through WCET analysis. It is based on which the difference of WCET reduction of both programs (i.e., \( \Delta W \)) is calculated at line 12. If this difference is larger than the difference of last iteration (i.e., \( \Delta \text{Original } W \)), then the termination variable is set true, else the smaller difference is assigned to \( \Delta \text{Original } W \) to further minimize the difference in terms of the amount of WCET reduction for both threads. This algorithm is repeated until the value of termination variable becomes true.

2) Percentage-Fairness-Oriented Scheme

While AFO targets approximately the same amount of WCET reduction, PFO aims at about a similar percentage of WCET reduction. The principle of PFO code positioning is described as follows. In the multicore processor with shared L2 cache, the WCET of one thread can be broken into the computation time by assuming perfect caches namely, the L1 cache miss penalty and the L2 cache miss penalty. The L2 cache miss penalty consists of two parts: 1) the intra-thread L2 cache miss penalty caused by L2 cache conflicts between instructions from the same thread and 2) the inter-thread L2 cache miss penalty due to the cache conflicts between the instructions from different threads that share the L2 cache. The WCET of one thread can be calculated by Equation 1, where \( E \) stands for the computation time without considering the cache misses, \( L_1 \) is L1 cache miss penalty and \( L_2 \) and \( Out_L_2 \) represent the intra-thread and inter-thread L2 cache miss penalty, respectively.

\[
\text{WCET} = E + L_1 + (L_2 + Out_L_2)
\]  (1)

After code positioning, the inter-thread cache conflicts will be decreased. However, the intra-thread cache conflicts may increase both on L1 and L2 caches. As the computation time \( E \) is the same before or after code positioning, the improvement of the WCET after code optimization can be illustrated as Equation 2.

\[
\Delta \text{WCET} = \Delta \text{Out}_L_2 + \Delta L_1 + \Delta \text{In}_L_2
\]  (2)

As the goal of PFO code positioning is to maximally reduce the WCET of each real-time thread by an approximately equal percentage. Assume that there are two threads, i.e., Thread A and Thread B. Equation 3 can be used to characterize this scheme. In this equation, \( WCET_A \) and \( WCET_B \) are the original WCETs of Thread A and Thread B, respectively. \( \Delta WCET_A \) and \( \Delta WCET_B \) are derived from Equation 2 and they denote the change of the WCET for each thread.

\[
\frac{\Delta WCET_A}{WCET_A} = \frac{\Delta WCET_B}{WCET_B}
\]  (3)

As the execution time \( E \) may vary substantially for different real-time threads, it becomes very hard. If not impossible, to guarantee the same percentage of WCET reduction if \( E \) is considered. Moreover, as the execution time \( E \) is insensitive to cache-based optimizations, the PFO scheme focuses on the reduction of the same percentage of L1 and L2 cache misses for both threads through cooperative code positioning. We also find out that while the reduction of inter-thread cache conflict is mutual, the L1 cache misses and L2 intra-thread misses are heavily dependent on how many instructions which are positioned for that particular thread. Specifically, the more instructions are positioned for that thread more is the possibility for the occurrence of intra-thread L1 and L2 cache conflicts. This leads to an increased L1 cache miss penalty and intra-thread L2 cache miss penalty. Therefore, in order to reduce the WCETs of both threads by approximately equal percentage, the number of instructions to be positioned for each thread should be inversely proportional to its original WCET as depicted in Equation 4.

\[
\frac{\text{Instr}_\text{NumB}}{\text{WCET}_B} = \frac{\text{Instr}_\text{NumA}}{\text{WCET}_A}
\]  (4)

Algorithm 3 illustrates the algorithm of PFO code positioning approach. The inputs of the algorithm are the two programs that are to be optimized. In line 2, the termination variable that completes the optimization is initialized. In the next three lines, the original WCETs of both programs are calculated and the L2 cache conflict instruction list is also determined. In the core of the algorithm from line 6 to line 19, firstly the instructions that are required to be positioned for both programs are identified according to the designing principle of PFO that was aforementioned. Then both programs are positioned at line 9 and line 10. From line 11 to line 12, the WCETs of both programs are analyzed again after code positioning.
ing. Based on the original WCETs and new WCETs of both programs, the WCET percentage variance between these two programs is calculated. This is done to determine whether or not the WCET percentage variance after code positioning process is smaller than the original WCET percentage variance at lines 13 and 14. If this is true, then the original WCET percentage variance $\Delta P$ is set to be the most recently calculated WCET percentage variance $\Delta P$ at line 17, else the termination variable is set to true at line 15. This algorithm is repeated until the value of the termination variable becomes true.

**Algorithm 3** 

\begin{algorithm}
begin
begin
\begin{enumerate}
\item \begin{enumerate}
\item boolean terminate = false;
\item Original_P1_wcet = WCET Analysis(P1);
\item Original_P2_wcet = WCETAnalysis(P2);
\item Conflict.Op.List = Build_Conflict.Op.List(P1, P2);
\end{enumerate}
\item repeat
\begin{enumerate}
\item Positioning(P1, Pos.Op.List);
\item Positioning(P2, Pos.Op.List);
\item P1_wcet = WCETAnalysis(P1);
\item P2_wcet = WCETAnalysis(P2);
\item $\Delta P = \text{Calculate}_\text{Percentage}_\text{Variation}()$;
\item if $\Delta P >= \Delta \text{Original}_P$ then
\end{enumerate}
\item terminate = true;
\item \textbf{end if}
\item $\Delta \text{Original}_P = \Delta P$
\item \textbf{until} terminate == true;
\end{enumerate}
end
end
end
end
end
end

**D. Inter-thread L2 Cache Conflict Analysis**

In the co-optimization architecture depicted in Fig. 1, inter-thread L2 cache conflict analysis is an important step to identify the worst-case inter-core L2 cache conflicts and the associated instructions from different cores. This provides the inter-thread L2 cache conflict set to code positioning algorithms. We propose to leverage Yan et al.’s recent work in [17] to analyze the worst-case inter-thread L2 cache conflicts. The main steps of this algorithm are described in Algorithm 4.

The inputs of this algorithm are the programs from both the co-running threads. Initially, the L2 cache status sets for only one thread alone (i.e., without considering inter-thread conflicts) and they are calculated for both threads. This identifies the groups of instructions within the same thread that share the same cache lines. In order to find out the worst-case inter-thread instruction interferences from two different threads, we distinguish instructions in loops from those that are not in loops. Each instruction from each thread is examined, whose L1 cache access behavior can be easily obtained by using static analysis techniques for instruction caches (line 5-6) [17].

If there exists an L1 miss, then it is checked where this miss happens (line 7), i.e., in or out of loops. If this miss occurs in a loop, then it is necessary to determine whether or not the cache line used by this instruction will be occupied by the instructions from the other thread, and whether those instructions are also in a loop or not. The cache line used by this instruction from Thread 1 can be found by the function Find_Cache_Line at line 7. Function Find_Conflict.Op at line 8 helps to check if there is any instruction from Thread 2 by using the same L2 cache line. If there is an instruction from Thread 2 which also uses the same L2 cache line, then this instruction will be named as conflict_op and then it be checked in a

**Algorithm 4** 

\begin{algorithm}
begin
begin
\begin{enumerate}
\item T1_Cache_Pos = Initialize_Cache_Pos(T1);
\item T2_Cache_Pos = Initialize_Cache_Pos(T2);
\item for op in T1 do
\begin{enumerate}
\item if Is_L1_Miss(op) then
\end{enumerate}
\item \textbf{if} Is_L1_Loop(op) \textbf{then}
\item \textbf{if} conflict_op = null \textbf{then}
\item \textbf{end if}
\item \textbf{if} Is_In_Loop(conflict_op) \textbf{then}
\item weight = Min_Weight(op, conflict_op);
\item Add_Conflict_Matrix(op, conflict_op.weight);
\item \textbf{else}
\item Add_Conflict_Matrix(op, conflict_op, 1);
\item \textbf{end if}
\item \textbf{end if}
\item \textbf{else}
\item cache_line = Find_Cache_Line(op, T1_Cache_Pos);
\item conflict_op = Find_Cache_Line (T2_Cache_Pos);
\item if conflict_op! = null then
\item \textbf{end if}
\item \textbf{end if}
\item \textbf{end if}
\item \end{enumerate}
\end{enumerate}
end
end
end
end
end
end
end
end
end
loop or not at line 10. If the conflicting instruction happens to be in a loop as well, then the worst-case number of conflicts of these conflicting instructions is equal to the smaller one of the worst-case number of access times from these two threads (line 11). This can be obtained from control flow analysis.

The inter-thread L2 cache conflict set is constructed in the format of a matrix. In this type of matrix, a row index represents the instruction number from Thread I and a column index denotes the instruction number from Thread II. The element of this matrix is a cache conflict reference object. It contains the L2 cache line number and the frequency of conflicts. After obtaining the worst-case conflict frequency, a cache conflict reference object is generated and it is added to the matrix at the place that is determined by the index number of the conflicting instructions (line 12). If the conflicting instruction from Thread II is not in a loop, then the inter-thread L2 cache conflict can occur only once in the worst case. Therefore, the frequency attribute of the cache conflict reference object is 1 and it is added into the inter-thread L2 cache conflict set by function Add_Conflict_Matrix at line 14. Moreover, if the instruction from Thread I is outside a loop, then the worst-case conflict frequency is only 1 as well (line 18-22). More details about this inter-thread L2 cache instruction interference analysis can be found at [17].

E. WCET Calculation

The WCET of a real-time task is computed by using the implicit path enumeration technique (IPET) proposed by Li and Malik [18, 19]. In IPET, the WCET of each task is calculated by maximizing the objective function in Equation 5 where, $c_i$ is the execution cost of the basic block $i$, including cache miss penalty and $b_i$ represents the number of times the basic block $i$ is executed. To legally maximize the objective function, program structural constraints should be taken into account. They are derived from the program’s control flow information for each basic block $i$, as described in Equation 6. In this equation, in_edge $i$ is the sum of the edges that enter the basic block $i$ and out_edge $i$ is the sum of the edges that exit the basic block. This should be equal to each other.

$$\text{Total execution time} = \sum_{i=1}^{n} c_i \times b_i \quad (5)$$

$$\sum_{\text{in.edge} i} = \sum_{\text{out.edge} i} = b_i \quad (6)$$

When an instruction runs in a multi-core processor with a shared L2 cache, on a multi-core processor with a shared L2 cache can be derived by using static cache analysis. Moreover, the state of L2 instruction cache accesses for each basic block includes the potential inter-thread L2 cache conflicts. It can be computed by using the inter-thread L2 cache conflict analysis algorithm that is depicted in Section III-D. Therefore, the total number of cache misses can be calculated in Equation 7. Where, $b_i$ denotes the number of times basic block $i$ is executed, $m_1$ is the number of L1 cache misses of the basic block $i$ and $m_2$ and $m_2'$ account for the numbers of intra-thread L2 cache misses and inter-thread L2 cache misses of basic block $i$, respectively.

$$\text{Cache misses} = \sum_{i=1}^{n} m_1 i + (m_2 + m_2') i \times b_i \quad (7)$$

Equation 8 integrates the penalty of cache misses into the objective function to accurately compute the WCET of the whole program. In this equation, $e_i$ represents the basic execution latency of basic block $i$ by assuming a perfect cache, $l_{\text{1 penalty}}$ stands for the L1 cache miss penalty and $l_{\text{2 penalty}}$ denotes the L2 cache miss penalty.

$$c_i = e_i + m_1 i \times l_{\text{1 penalty}} + (m_2 + m_2') i \times l_{\text{2 penalty}} \quad (8)$$

As a result, the WCET of the real-time thread can be calculated by using an integer linear programming (ILP) solver in order to maximize the objective function in Equation 5.

IV. EVALUATION METHODOLOGY

We evaluate the proposed multicore-aware code positioning schemes on a heterogeneous dual-core processor with a shared L2 cache. In order to achieve better performance, energy efficiency and low cost, embedded applications have increasingly used heterogeneous systems. This includes multiple programmable processor cores, specialized memories and other components on a single chip [20]. For instance, most of the hand-held devices now adopt a heterogeneous dual-core architecture that is composed of a digital signal processing (DSP) core and an advanced RISC machines (ARM) core. In this paper, we focus on the evaluation of the multi-thread code positioning on a heterogeneous dual-core processor that consists of a very long instruction word (VLIW)-based DSP core and a general-purpose core. The VLIW core is based on the Hewlett-Packard Lab PlayDoh (HPL-PD) 1.1 architecture [21] and the general-purpose core is similar to the Alpha 21264 processor [22]. More specifically, the simulation tools of Trimaran [23] and Chronos [24] (including SimpleScalar [25]) are extended to simulate this framework. The front end of Chronos compiles the other thread benchmark into common object file format (COFF) format binary code by gcc compiler. This is targeted to SimpleScalar. By disassembling the binary code,
the global control-flow graph (CFG) and the related information of instructions are acquired by Chronos front end. Hence, it helps in the static cache analysis. A commercial ILP solver-CPLEX [26] is used to solve the ILP problem to obtain the estimated WCET.

Without losing generality, we assume a dual-core processor with two-level cache memories. Each core has its own L1 instruction cache and L1 data cache. Moreover, both cores share the same L2 cache in order to utilize the aggregate L2 cache space. Note that multicore processors can also use separated L2 caches to achieve better time predictability. However, a shared L2 cache has a few important advantages such as fast data sharing, reduced cache-coherency complexity and false sharing and possibly superior cache performance [16]. In order to limit the scope of this study and to focus on instruction cache analysis, the L1 data cache of each core is assumed to be perfect. In order to compare the worst case performance with an average case performance in the heterogeneous dual-core processor, the memory hierarchy of SimpleScalar simulator is integrated into a Trimaran's memory hierarchy and the core of SimpleScalar is linked to Trimaran’s simulator by means of multi-thread programming. Therefore, an environment where two threads can run at the same time on different cores with a shared L2 cache has been simulated. The basic configuration of a simulated hybrid dual-core processor is shown in Table 1.

In our experiments, we choose sixteen benchmarks from Malardalen WCET benchmark suite [27] and based on them we form eight benchmark pairs by selecting a thread from each group as shown in Table 2. The performance results of WCO code positioning, PFO code positioning and AFO code positioning are compared with the obtained baseline performance results. In this no code positioning approach is applied.

V. EXPERIMENTAL RESULTS

A. Performance Results of WCO

Fig. 2a compares the WCETs of eight benchmark pairs between the WCO scheme and the baseline scheme. They are normalized with respect to the results of the baseline scheme. We can clearly see that the WCO scheme can decrease the WCET for all the threads as reducing the inter-thread L2 cache misses benefits of both threads. The percentages of WCET reduction for those eight benchmark pairs ranges from 1.14% to 15.85% and they depend on how much percentage the inter-thread L2 cache miss

| Table 1. Basic configuration of a simulated heterogeneous dual-core processor |
|----------------|----------------|----------------|
| Parameter       | VLIW core       | General-purpose core |
| Core            | VLIW core       | General-purpose core |
| Datapath        | 4 IFUs, 2 FPUs, 2 Ld/Sts, 1 BrU 64 registers | 4 issue, 64 registers, 80-RUU, 40-LSQ |
| L1 I-cache      | 128 bytes, direct-map, 8 byte block, 1 cycle latency | 128 bytes, direct-map, 8 bytes block, 1 cycle latency |
| L1 D-cache      | Perfect          | Perfect          |
| L2 cache        | 2,048 bytes, direct-map, 16 bytes block, 4 cycle latency | Unlimited, 100 cycle latency |
| Memory          | Unlimited       | Unlimited       |

VLIW: very long instruction word, IFU: instruction fetch unit, FPU: floating point unit, RUU: register update unit, LSQ: load-store queue.

| Table 2. Estimated and simulated worst-case performance results of the baseline scheme |
|----------------|----------------|----------------|
| Thread I       | Thread II      | Sim. WCET | Sim. L2 miss rate (%) | Est. WCET | Est. L2 miss rate (%) |
| Est. WCET | Est. L2 miss rate (%) | Sim. WCET | Sim. L2 miss rate (%) | Est. WCET | Est. L2 miss rate (%) |
| BS    | 3040 | 58.70 | 2401 | 55.18 | Fft1 | 10677 | 30.69 | 8988 | 26.97 |
| Cover | 29987 | 18.97 | 24918 | 16.30 | Ndes | 367695 | 2.61 | 332330 | 2.36 |
| Expint | 10488 | 61.64 | 8570 | 51.52 | Qsort | 26793 | 18.81 | 20208 | 16.35 |
| Fdct | 16496 | 8.56 | 13982 | 7.24 | Startup | 11710 | 34.91 | 9246 | 30.14 |
| Insertsort | 5627 | 60.29 | 4116 | 55.18 | Fibcall | 3426 | 59.18 | 2214 | 46.51 |
| Qurt | 10375 | 31.06 | 8127 | 25.23 | Crc | 105904 | 3.50 | 85327 | 3.22 |
| Sqrt | 9042 | 60.64 | 7030 | 54.39 | Minver | 20798 | 30.46 | 16843 | 27.64 |
| Ud | 24175 | 18.75 | 19615 | 16.71 | Biquad | 7943 | 47.79 | 6128 | 46.06 |

WCET: worst-case execution time.
The variation of L2 cache miss rate of these benchmarks can be seen in Fig. 2b. For the WCO scheme, it is likely that the thread with the longest WCET is not positioned, as the WCET of this thread is much larger than that of the other thread. In this case, the L2 cache miss rate of this thread with the longest WCET can be reduced more by the WCO scheme than both the PFO or AFO schemes as no additional intra-thread L1 cache misses and intra-thread L2 cache misses will occur in this thread with the WCO scheme. For instance, in benchmark pair 3 both the WCET and L2 cache miss rate of benchmark Qsort in the WCO scheme are lower than those of the PFO and AFO schemes. On the other hand, its counterpart benchmark Expint has higher L2 cache miss rate and larger WCET in the WCO scheme than those of the AFO scheme (which has better results than PFO). We also notice that for the benchmark pair 5, L2 miss rates and WCETs of both threads are adequately reduced by all the three schemes. This is due to the difference between the original WCETs of both benchmark pairs is relatively small and the L2 cache miss penalty takes a large fraction of their respective WCET.

B. Performance Results of PFO

The performance results of the PFO code positioning approach are also demonstrated in Fig. 2. It indicates that this approach can reduce the WCETs of both the threads within a benchmark pair by approximately equal percentages. For example, the difference of WCET reduction percentage for benchmark pair 2 that consists of Cover and Ndes is only 0.03%. Even for the worst case, the difference between WCET optimization percentage for benchmarks Insertsort and Fibcall is just 0.64%. On average, the variation of WCET optimization percentage for these eight benchmark pairs is only 0.29%.

However, we also find out that the percentage of WCET reduction by PFO varies much across the different benchmark pairs. For example, the WCET of the first benchmark pair is reduced by more than 5%, while the percentages of WCET reduction for benchmark pairs 2 and 6 are just about 1%. This is due to the effect of the PFO approach on WCET reduction is mainly determined by two factors. First, to ensure fairness of WCET optimization, a wide discrepancy between the original WCETs of both threads limits the degree of WCET improvement for both benchmarks, for instance the benchmarks in pair 2 and pair 6. Second, the percentage of inter-thread L2 cache miss penalty in the original WCET is another important factor for the determination of the WCET enhancement through code positioning. Generally, if this percentage is higher, then there is more room for potential WCET enhancement. The first factor also leads to another conclusion that the PFO approach is generally worse than the other two approaches in terms of the reduction in the worst-case execution time (i.e., achieving “better” WCET). This can be observed in Fig. 2 in case of both WCET and L2 cache miss rate. In other words, while the PFO approach can achieve fairness in terms of the percentage of WCET optimization for co-running threads, it indeed compromises the efficiency of WCET optimization as compared to WCO and AFO.

C. Performance Results of AFO

Fig. 2 also illustrates the normalized WCET and L2 cache miss rate in case of the AFO scheme with respect to the Baseline Scheme. The AFO scheme can not only reduce the WCET and L2 cache miss rate for both threads in each benchmark pair, but it can also achieve the fairness in terms of the amount of WCET reduction. Moreover, the differences in the reduced WCETs between both benchmarks only range from 4 cycles to 120 cycles across all pairs. On an average, the difference of WCET reduction is only about 80 cycles. This is less than the latency that incurred from one L2 cache miss indicating that the fairness in terms of the amount of WCET reduc-

http://dx.doi.org/10.5626/JCSE.2012.6.1.12

Yiqiang Ding and Wei Zhang
tion between co-running threads is achieved.

Interestingly, when we compare AFO with WCO (note that PFO in general is inferior to both AFO and WCO as aforementioned), we find out that for all benchmark pairs, while WCO can decrease the WCET for one thread more than AFO, AFO can often reduce the WCET of the other thread (in the same pair) more than WCO. The reason is that the AFO approach does not position some of the instructions of the thread that would be aggressively positioned by the WCO approach for reducing the inter-core L2 cache misses. This often leads to an increase of intra-core L2 cache miss on one thread while it leads to a decrease of it on the other thread. As an example, in the first benchmark pair, the benchmark FFT gets 2.14% improvement on WCET in the WCO scheme than in the AFO scheme. However, the WCET of BS in the AFO scheme is about 3.29% and it is better compared to the WCO scheme. Therefore, we believe AFO is comparable to WCO in terms of WCET optimization, while achieving fairness in terms of the amount of WCET reduction by considering both the co-running threads.

D. Sensitivity Analysis

We have also made experiments to study the effects of the code positioning approach with the different L2 cache sizes. Table 3 shows the performance optimization percentage in the PFO scheme as the L2 cache size increases from 1,024 bytes to 2,048 bytes and 4,096 bytes, while the L1 cache size of both cores is kept to 128 bytes. As one can expect, PFO strategy makes both threads gain the approximated optimization percentages in all these three cache models. With 1,024 bytes L2 cache configuration, the average variation of the optimization percentages is nearly 0.5% and the maximum value of the optimization percentage difference is only 1.3%. The average value is almost 0.43% in case of 4,096 bytes L2 cache. In principle, the performance optimization percentage will increase with an increase of the cache size, as there are possibly more free memory spaces where more conflicting instructions can be positioned. This is proved by the results of several benchmark pairs in the cache size sensitive experiments. For example, the average percentage of WCET improvement of benchmark pair 5 increases from 4.75% to 8.2%, and 10.53%, when the size of the L2 cache increases from 1,024 to 2,048, and 4,096 bytes, respectively. However, the average percentage in the results of some benchmark pairs does not vary much with the variation of the L2 cache size. However, for benchmark pair 2 the average optimization percentage only ranges from 0.455% to 0.64% and it does not increase strictly proportional to the size of L2 cache. This exceptional result may be caused by the fact that the benchmark is not sensitive to this scope of the cache size.

For the WCO scheme, there are three cases in the performance optimization variation of the worst-case WCET with an increase of the L2 cache size as shown in Fig. 3. For some benchmark pairs, the optimization percentage of the worst-case WCET by WCO code positioning increases with a larger size of L2 cache. For example, the worst-case WCET of benchmark pair 3 is improved by 8.05% when the cache size is 1,024 bytes. When the L2 cache size is increased to 2,048 bytes and 4,096 bytes, its WCET is reduced by 10.5% and 11.25%, respectively. This is due to the inter-thread L2 conflicts that are avoided by code positioning take a larger portion of the worst-case WCET with larger L2 caches for these benchmark pairs. However, the optimization percentage decreases when the L2 cache size increases from 1,024 to 2,048, and 4,096 bytes in some benchmark pairs, such as benchmark pair 1 and 8, as the proportion of the inter-thread L2 misses contained by the worst-case WCET is reduced when the L2 cache size increases. Another case is not quite straightforward and the optimization percentage of

<table>
<thead>
<tr>
<th>No.</th>
<th>1,024 L2 cache</th>
<th>2,048 L2 cache</th>
<th>4,096 L2 cache</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Thread I</td>
<td>Thread II</td>
<td>Thread I</td>
</tr>
<tr>
<td>1</td>
<td>5.19</td>
<td>3.79</td>
<td>5.12</td>
</tr>
<tr>
<td>2</td>
<td>0.48</td>
<td>0.53</td>
<td>0.65</td>
</tr>
<tr>
<td>3</td>
<td>4.50</td>
<td>4.13</td>
<td>5.18</td>
</tr>
<tr>
<td>4</td>
<td>4.49</td>
<td>4.03</td>
<td>7.4</td>
</tr>
<tr>
<td>5</td>
<td>4.72</td>
<td>4.78</td>
<td>12.57</td>
</tr>
<tr>
<td>6</td>
<td>0.88</td>
<td>0.74</td>
<td>1.47</td>
</tr>
<tr>
<td>7</td>
<td>4.12</td>
<td>3.61</td>
<td>6.11</td>
</tr>
<tr>
<td>8</td>
<td>1.30</td>
<td>1.43</td>
<td>4.47</td>
</tr>
</tbody>
</table>

Values are presented as number (%).

Fig. 3. Maximum optimization percentage of worst-case execution time (WCET) in worst-case-oriented scheme with L2 cache size ranging from 1,024 bytes to 4,096 bytes and they are normalized with respect to that of a 2,048 bytes L2 cache.
the worst-case WCET increases when the L2 cache size increases from 1,024 bytes to 2,048 bytes. However, it decreases with the L2 cache which increases from 2,048 bytes to 4,096 bytes. The result of benchmark pair 4 is an instance of this case. In this case, there is more memory space to position the conflicting instructions in order to avoid more inter-thread L2 cache misses when the L2 cache size increases from 1,024 bytes to 2,048 bytes. Thus, the portion of the worst-case WCET taken by the inter-thread cache misses are eliminated by code positioning will increase. On the other hand, when the L2 cache size increases from 2,048 bytes to 4,096 bytes, the number of inter-thread L2 cache misses is reduced and thus it takes a smaller fraction of the WCET. Even if all the inter-thread cache misses can be avoided by code positioning, the optimization percentage induced by them is still less than that in the case of 2,048 byte L2 cache. This is the reason for the decrease in the optimization percentage of the worst-case WCET while the L2 cache size increases from 2,048 bytes to 4,096 bytes.

As illustrated in Table 4, the average difference of the reduced WCET of all the eight benchmark pairs in AFO scheme ranges from 70.5 to 42.5 to 55.75 and the L2 cache size increases from 1,024 bytes to 2,048 bytes to 4,096 bytes. The result of each benchmark pair does not very greatly between the difference changing and the L2 cache size. Therefore, the difference of the reduced WCET is not sensitive to the L2 cache size when it achieves fairness on the amount of reduced WCET by the AFO code positioning.

We have studied sensitivity with the memory latency that is increased from 100 cycles to 200 cycles. Under this configuration, the L2 cache miss penalty for a benchmark obviously takes a larger part of its WCET. By further deduction, it can be derived that the inter-thread L2 cache conflicts avoided by our co-optimization approach are responsible for a larger portion of the WCET. Therefore, the effect of optimization in this case is better than that of the basic configuration. This derivation has to be proved by the experiment result that is shown in Fig. 4. For WCO scheme, the average optimization percentage of the WCET of the worst-case thread reaches 9.94% while this data in the basic configuration with 100 cycles of memory latency is only 8.41% as depicted in Fig. 3. And, the WCET optimization percentages of all the benchmark pairs in PFO Scheme with 200 cycles of memory latency are higher when compared to that

![Table 4](image)

**Table 4.** Reduced worst-case execution time of amount-fairness-oriented scheme with respect to baseline scheme with L2 cache size ranging from 1,024 bytes to 2,048 bytes and 4,096 bytes.

<table>
<thead>
<tr>
<th>No.</th>
<th>1,024 L2 cache</th>
<th>2,048 L2 cache</th>
<th>4,096 L2 cache</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Thread I</td>
<td>Thread II</td>
<td>Thread I</td>
</tr>
<tr>
<td>1</td>
<td>1472</td>
<td>1368</td>
<td>1180</td>
</tr>
<tr>
<td>2</td>
<td>5020</td>
<td>4932</td>
<td>8020</td>
</tr>
<tr>
<td>3</td>
<td>1968</td>
<td>2000</td>
<td>1900</td>
</tr>
<tr>
<td>4</td>
<td>2772</td>
<td>2856</td>
<td>2340</td>
</tr>
<tr>
<td>5</td>
<td>768</td>
<td>772</td>
<td>868</td>
</tr>
<tr>
<td>6</td>
<td>2872</td>
<td>2788</td>
<td>2500</td>
</tr>
<tr>
<td>7</td>
<td>1136</td>
<td>1044</td>
<td>948</td>
</tr>
<tr>
<td>8</td>
<td>2436</td>
<td>2512</td>
<td>2436</td>
</tr>
</tbody>
</table>

![Fig. 4](image)

**Fig. 4.** Worst-case execution time (WCET) optimization percentage of worst-case-oriented (WCO) scheme, percentage-fairness-oriented (PFO) scheme and amount-fairness-oriented (AFO) scheme with the 200 cycles memory latency, which are normalized with respect to that of the 100 cycle memory latency.
obtained in the basic configuration as shown in Table 3. As an example, WCET optimization percentages of the benchmarks in benchmark pair 4 are 9.01% and 8.02%, respectively, and they are larger than 7.40% and 6.94% that was obtained in the basic configuration. Moreover, in case of an AFO Scheme, the average optimized percentage of WCET with 200 cycles of memory latency is about 18% higher when normalized with respect to the results with 100 cycle memory latency for Thread I, and it is about 30% higher for Thread II as shown in Fig. 4c.

E. Comparison of Code Positioning Schemes with Separated L2 Caches

In order to compare the performance of the proposed code positioning schemes with the technique of cache partitioning, in our experiments, the 2,048 bytes L2 cache is separated by half. Moreover, only one thread can access one of them to simulate a simple hardware-based cache partitioning (i.e. a separated L2 cache architecture). In this paper, it is also called as the SC scheme. As shown in Fig. 5a, for all benchmark pairs the WCETs of both or at least one of the WCO and AFO schemes are better than that of the SC scheme. Even in some of the benchmarks, the performance of the SC scheme is worse than that of the baseline scheme, for instance Cover and Qsort. This is due to the cache partitioning which helps to reduce cache interferences between the different threads and it may bring much more intra-thread cache conflicts as the actual cache mapping space is reduced by half. This is especially problematic if the code size of the working set exceeds the cache size, which is very likely in embedded processors due to the resource constraints. Therefore, for the benchmarks evaluated in this paper, we believe that the code positioning approaches that were studied in this paper are more effective than simply separating the L2 cache by half in improving the worst-case execution time for real-time tasks.

VI. CONCLUSIONS

Current studies on real-time software on multicore processors are mainly focused on methods to accurately estimate the WCET. With the available WCET information, it is also desirable to optimize (or to reduce) WCET on multicore platforms by exploiting compiler transformations. This paper studies the novel compilation time code co-positioning approaches on multicore platforms in order to co-optimize the worst-case performance for real-time threads that run concurrently on a multicore processor with a shared L2 cache. We have studied three different multicore-aware code position schemes to either maximally reduce the longest WCET or to ensure fairness of WCET enhancement among all the co-running threads that can be automatically performed during the compilation time.

Our experiments indicate that the WCO scheme can efficiently reduce the worst-case execution time for a single thread with the worst WCET. It also indicates that the AFO and PFO schemes can reduce the WCETs of co-running threads by approximately the same amount or percentage, respectively. The evaluation also shows that the multicore-aware code positioning approaches are generally more effective than simply separating the L2 cache by half in order to reduce the worst-case execution time. Moreover, our study indicates that the proposed multicore-aware code co-positioning approach is even more effective for processors with longer memory access latency. This is likely to be true for future multicore/manycore processors due to the memory wall problem. However, the effectiveness of the three optimization strategies on different L2 cache sizes is largely dependent on the behavior of the concurrent real-time threads.

In our future work, we plan to investigate the interactions between inter-thread code positioning which is studied in this paper and intra-thread code positioning such as the work in [3] in terms of improving the worst-case per-

![Fig. 5. Worst-case execution time (WCET) and L2 cache miss rate of the worst-case-oriented (WCO) scheme, the amount-fairness-oriented (AFO) scheme (which is better than percentage-fairness-oriented (PFO)) and the separated L2 cache (SC) scheme, which are normalized with respect to the baseline scheme.](image-url)
formance and to possibly combine them in an efficient manner to achieve even better worst-case performance for real-time application. Moreover, this work is also built upon the inter-thread L2 instruction cache analysis technique in [17] but the multicore-aware code positioning approach is independent of a specific timing analysis technique. So we plan to examine the use of other multicore timing analysis techniques in order to further improve the effectiveness of WCET-oriented optimizations. Moreover, it will also be interesting to study the multicore-aware data layout optimizations, in addition to code positioning that was studied in this paper, for the attainment of better WCET for real-time tasks that run on multicore processors.

REFERENCES

Multicore-Aware Code Co-Positioning to Reduce WCET on Dual-Core Processors with Shared Instruction Caches

Yiqiang Ding

Yiqiang Ding is currently a Ph.D student in Electrical and Computer Engineering at Southern Illinois University Carbondale. 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.

Wei Zhang

Dr. Wei Zhang is an associate professor 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 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. His research has been supported by NSF, IBM, Intel, Motorola and Altera. He is a senior member of the IEEE. He has served as a member of the organizing or program committees for several IEEE/ACM international conferences and workshops.