WCET Analysis for Multi-Core Processors with Shared L2 Instruction Caches

Jun Yan and Wei Zhang
Department of Electrical and Computer Engineering
Southern Illinois University Carbondale
Carbondale, IL 62901
{jun,zhang}@engr.siu.edu

Abstract

Multi-core chips have been increasingly adopted by microprocessor industry. For real-time systems to safely harness the potential of multi-core computing, designers must be able to accurately obtain the worst-case execution time (WCET) of applications running on multi-core platforms, which is very challenging due to the possible runtime inter-core interferences in using shared resources such as the shared L2 caches.

As the first step toward time-predictable multi-core computing, this paper presents a novel approach to bounding the worst-case performance for threads running on multi-core processors with shared L2 instruction caches. The idea of our approach is to compute the worst-case instruction access interferences between different threads based on the program control flow information of each thread, which can be statically analyzed. Our experiments indicate that the proposed approach can reasonably estimate the worst-case shared L2 instruction cache misses by considering inter-thread instruction conflicts. Also, the WCET of applications running on multi-core processors estimated by our approach is much better than the estimation by simply assuming all L2 instruction accesses are misses.

1. Introduction

With the scaling of technology and the diminishing return of complex uniprocessors, computer industry is rapidly moving towards single-chip multi-core processors or chip multiprocessors (CMP). Multi-core processors have been widely used in servers, desktops, and embedded systems. In particular, with the growing demand of high performance by high-end real-time applications such as HDTV and video encoding/decoding standards, it is expected that multi-core processors will be increasingly used in real-time systems for achieving higher performance/throughput cost-effectively. Actually, it is projected that real-time applications will be likely deployed on large-scale multi-core platforms with tens or even hundreds of cores per chip fairly soon [21].

For real-time systems, especially hard real-time systems, it is crucial to obtain the worst-case execution time (WCET) of each real-time task, which will provide the basis for schedulability analysis. Missing deadlines in those systems may lead to serious consequences. While the WCET of a single task can be measured for a given input, it is generally infeasible to exhaust all the possible program paths through measurement. Another approach to obtaining WCET is to use static WCET analysis (or simply called WCET analysis). WCET analysis typically consists of three phases: program flow analysis, low-level analysis, and WCET calculation. While the program flow analysis analyzes the control flow of the assembly programs that are machine-independent, the low-level analysis analyzes the timing behavior of the microarchitectural components. Based on the information obtained from the program flow analysis and the low-level analysis, the WCET calculation phase computes the estimated worst-case execution cycles by using methods such as path-based approach [5, 6] or IPET (Implicit Path Enumeration Technique) [7, 8, 9].

While there have been many research efforts on WCET analysis for single-core processors [1, 5, 6,

1080-1812/08 $25.00 © 2008 IEEE
DOI 10.1109/RTAS.2008.6

7, 8, 9, 10], to the best of our knowledge, no prior work has studied the WCET analysis of multi-core processors. A major reason is probably the significant complexity involved with the WCET analysis for multi-core processors. Even for today's single-core processors, many architectural features such as cache memories, pipelines, out-of-order execution, speculation and branch prediction have made "accurate timing analysis very hard to obtain" [15]. Multi-core computing platforms can further aggravate the complexity of WCET analysis due to the possible inter-thread interferences in shared resources such as L2 caches, which are very difficult to analyze statically. While recently there have been some research efforts on real-time scheduling for multi-core platforms [21, 22, 23], all these studies basically assume that the worst-case performance of real-time threads are known. Therefore, it is a necessity to reasonably bound the WCET of real-time threads running on multi-core processors before multi-core platforms can be safely employed by real-time systems.

As the first step towards WCET analysis of multi-core processors, this paper examines the timing analysis of shared L2 instruction caches for multi-core processors. In this paper, we assume data caches are perfect, thus data references from different threads will not interfere with each other in the shared L2 cache[1]. We propose to exploit program control flow information (i.e., loops) of each thread to safely and efficiently estimate the worst-case L2 instruction cache conflicts. Built upon the static cache analysis results, we integrate them with the pipeline analysis and path analysis to obtain the WCET for multi-core processors.

The rest of the paper is organized as follows. First, we discuss the difficulty of WCET analysis for multi-core chips with shared caches due to the timing anomalies in Section 2. Then we describe our approach to computing the worst-case shared L2 instruction cache performance and the WCET for multi-core processors in Section 3. The evaluation methodology is given in Section 4 and the experimental results are presented in Section 5. Finally, we make concluding remarks in Section 6.

[1]It should be noted that this paper does not solve the full problem of WCET analysis for multi-core chips. However, we believe we have made an important step by reasonably bounding the worst-case shared multi-core cache performance due to instruction accesses.

2. Difficulties in WCET Analysis for Multi-Core Chips with Shared L2 Caches

In a multi-core processor, each core typically has private L1 instruction and data caches. The L2 (and/or L3) caches can be either private or shared. While private L2 caches are more time-predictable in the sense that there are no inter-core L2 cache conflicts, each core can only exploit limited cache space. Due to the great impact of the L2 cache hit rate on the performance of multi-core processors [16, 17], private L2 caches may have worse performance than shared L2 caches with the same total size, because each core with shared L2 cache can make use of the aggregate L2 cache space more efficiently. Moreover, shared L2 cache architecture makes it easier for multiple cooperative threads to share instructions, data and the precious memory bandwidth to maximize performance. Therefore, in this paper, we focus on studying WCET analysis of multi-core processors with shared L2 caches (by contrast, the WCET analysis for multi-core chips with private L2 caches is a less challenging problem).

2.1 A Dual-Core Processor with a Shared L2 cache

Without losing generality, we assume a dual-core processor with two levels of cache memories, and the proposed static analysis approach can be easily extended to multi-core processors with multi-level memory hierarchy. As can be seen from Figure 1, in this dual-core processor, each core has its own L1 instruction cache and L1 data cache, and both cores share the same L2 cache to best utilize the aggregate L2 cache space. As aforementioned, we assume the L1 data cache of each core is perfect. Also, we assume two threads consisting of a real-time thread and a non-real-time thread are running on these two cores simultaneously, and our task is to safely and accurately estimate the WCET of the real-time thread (assuming non-preemptive execution)[2] by taking into account the

[2]It should be noted that this paper focuses on analyzing the WCET of a single real-time task running on a dual-core processor, thus the study of the effects of context switching within a single

possible L2 cache interferences from the non-real-time thread.

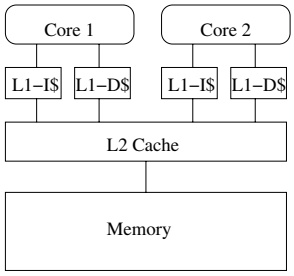

Figure 1. A dual-core processor with a shared L2 cache.

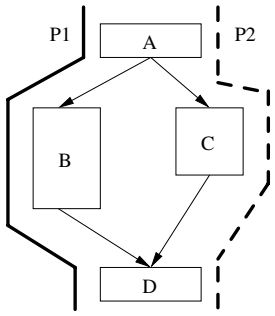

Figure 2. An example of a timing anomaly in a multi-core processor.

2.2 Timing Anomalies in Multi-Core Computing

The inter-thread cache conflicts in multi-core processors with shared cache memories can lead to timing anomalies. Timing anomalies were first discovered in out-of-order superscalar processors by Lundqvist and Stenstrom [24], where the worst-case execution time does not necessarily relate to the worst-case behavior. For instance, Lundqvist and Stenstrom [24] found that a cache miss in a dynamically-scheduled processor may result in a shorter execution time than a cache hit, which is counterintuitive. Similarly we find that in a multi-core processor with a shared L2 cache, the worst-case behavior of a single thread does not necessarily lead to the worst-case execution time of that thread, because of the inter-thread cache conflicts.

For example, Figure 2 shows the control flow graph of a code segment, which contains two paths: *P1 (A-B-D)* and *P2 (A-C-D)*. Suppose *P1* is the worst-case path of this code segment without considering the impact of other threads. After we take into account the inter-thread cache conflicts; however, *P1* may not be the worst-case path. For instance, if another thread running on another core evicts several instructions of the block C, while none or fewer instructions of block B are replaced by other threads in the shared L2 cache, then path *P2 (A-C-D)* may become the worst-case path and thus lead to the worst-case execution time for this thread. The reason is that the penalty of the inter-thread L2 cache misses occurring during the execution of the block B can be larger than the difference between path lengths of *P1* and *P2*, which is also the necessary and sufficient condition for the aforementioned time anomaly to happen.

Because of the timing anomalies in multi-core processors, the WCET analysis of each thread running on each core cannot be performed independently, which can significantly increase the complexity of the timing analysis. In particular, although current timing analysis techniques [1] can reasonably bound the performance of a single-core processor, they cannot be easily extended to compute the worst-case performance of each thread running on a multi-core processor. For instance, in Figure 2, while we can use existing single-core WCET analysis techniques [1] to obtain the worst-case path, i.e., *P1 (A-B-D)*, we must update this calculation by integrating the inter-thread cache conflicts information. Therefore, the critical problem of WCET analysis for multi-core processors with shared instruction caches is to safely and accurately identify the worst-case inter-core cache conflicts.

3. Our Approach

We propose a WCET analysis approach for multi-core processors with shared L2 caches with three major steps, including cache analysis, pipeline analysis and path analysis, which are built upon the extension of a single-core timing analysis tool called Chronos [2]. In this section, we first introduce the static cache analysis to bound the worst-case L2 instruction misses

core and/or across multiple cores falls out of scope of this paper.

by considering the inter-core instruction interferences in subsection 3.1. Then we explain pipeline analysis and path analysis in subsection 3.2 and subsection 3.3 respectively.

3.1. Static Analysis of Inter-Core Instruction Interferences in the Shared L2 Cache

The most difficult problem of the multi-core WCET analysis is to reasonably bound the worst-case inter-core interferences in the shared L2 caches. The inter-core L2 instruction interferences depend on several factors, including (1) the instruction addresses of the L2 accesses of each thread, (2) which cache block these instructions may be mapped to, and (3) *when* these instructions are accessed. While (1) and (2) can be statically analyzed, (3) is challenging. In this paper, we propose to efficiently identify the worst-case inter-core instruction interferences by distinguishing instructions that are in loops from those instructions not in loops (i.e., used at most once).

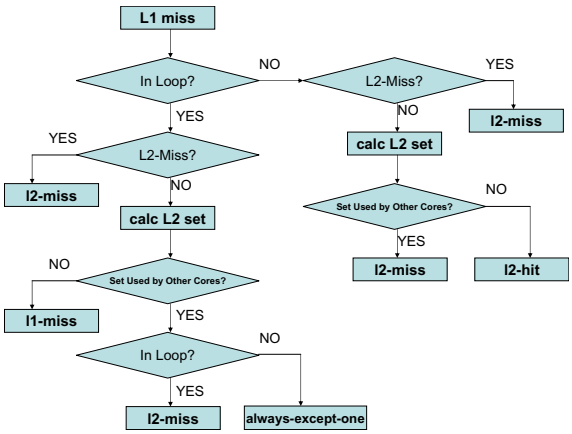

Figure 3. The flowchart of the multi-core static instruction cache analysis algorithm.

Our approach to estimating the worst-case inter-core instruction interference is shown in Figure 3, which works on each basic block level. As can be seen, when a L1 cache miss is determined (by using cache static analysis proposed by C. Ferdinand and R. Wilhelm [3]), this information is used as the input to determine the worst-case number of L2 misses. Specifically, when there is a L1 cache miss, we first check whether or not this miss happens in a loop. If this miss is not in the loop, then we determine whether or not it is a L2 miss. If it is not a L2 miss but a L2 hit, then we calculate its cache set number and the conflict set due to L2 accesses from other core(s). If another core may use this set during its execution time, then this L2 hit becomes a L2 miss (in the worst-case). Otherwise, it is still a L2 hit (i.e., "always hit" [3] in the shared L2 cache).

Another situation is when a L1 miss occurs in a loop, as can be seen from Figure 3. If this L1 miss hits in L2, we need to determine whether or not this set is used by others cores and whether or not it is used in a loop. If this cache set is used by other cores and at the same time used in a loop, then this L2 hit is classified as a L2 miss. However, if this cache set is used by another core but is accessed by instructions not in loops, then this L2 hit becomes "always-except-one hits". Otherwise, it is identified as a L2 miss.

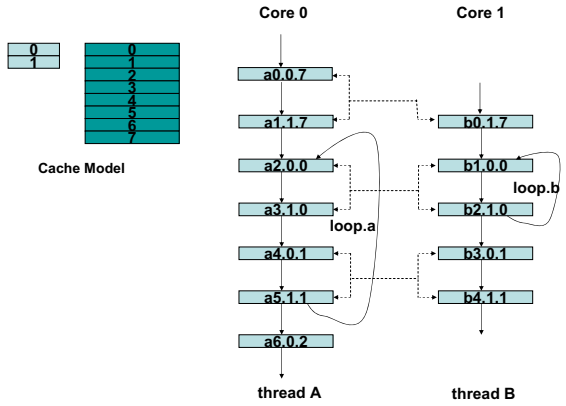

Figure 4. An example of WCET analysis of a mult-core chip.

For instance, Figure 4 shows an example to illustrate our approach to bounding the worst-case shared L2 instruction cache performance by considering the inter-core interferences. As can be seen in Figure 4, without loss of generality, we assume that two threads, A and B, are running in a dual-core processor. In this processor, core 0 is time-sensitive and is running

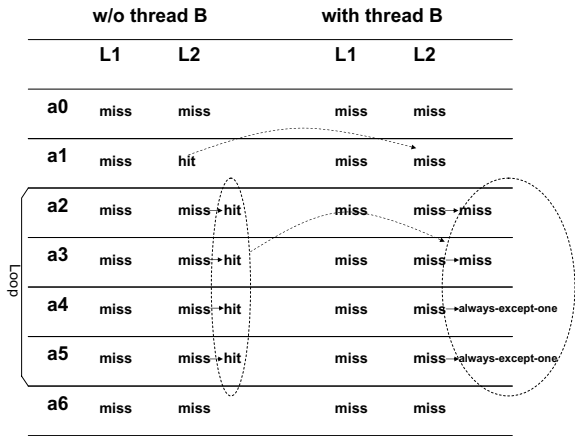

Figure 5. Status of each instruction in core 0 with and w/o considering interferences from core 1.

thread A; core 1 is not time-sensitive and is running thread B. The control flow graph of two threads are also given in Figure 4. The cache model we use is shown in Figure 4, in which L1 cache has 2 sets and each set can hold 1 instruction, and L2 cache has 8 sets and each set can hold 2 instructions. Each instruction in Figure 4 is labeled as the follows. The starting letter is the affiliated thread number. The number immediately following this letter is the number of this instruction. Then, the next number is the set number of the L1 cache. The last number indicates the set number of the L2 cache. For instance, b2.1.0 means that this is the 2nd instruction in thread B, which refers to the set 1 of the L1 cache and the set 0 of the L2 cache.

The status of each instruction with and without considering inter-core interferences is shown in Figure 5. For instance, without considering thread B, instruction a0.0.7 is a cold miss of L1 in set 0 and a cold miss of L2 in set 7. After that instruction, a1.1.7 should only suffer L1 miss, since it uses the set 1 in the L1 cache and the set 7 in the L2 cache. However, if we consider thread B that is concurrently running in core 1, instruction b0.1.7 may use the set 7 of L2 cache too. Thus it may happen that when core 0 finishes instruction a0 but before the execution of instruction a1, core 1 starts to run instruction b0. In this case, contents in the set 7 of L2 cache will be evicted by core 1. Thus, the status of instruction a.1.1.7 is changed to L2 cache miss in the worst case.

Figure 5 also illustrates how to exploit loop information to categorize the status of each instruction. For example, loop.a in thread A contains 4 instructions, i.e., a2.0.0, a3.1.0, a4.0.1 and a5.1.1. As can be seen, a2 and a4 conflict with each other in the L1 instruction cache, and so do a3 and a5. However their references to L2 have no conflict. During each iteration of loop.a, core 0 needs to fetch these 4 instructions from the L2 cache. Therefore, without considering thread B, the number of L2 cache misses of thread A is 2 at the first iteration and becomes 0 for the subsequent iterations. If we take thread B into consideration, however, as can be seen in Figure 4, in the worst-case, alternative running of instructions (a2, a3) and (b1, b2) will lead to two extra L2 cache misses when accessing a2 and a3 from the L2 cache. In contrast, since instruction a4 and a5 only interference with instruction b3 and b4, which are not in any loop of thread B, their status become "always-except-one hits".

For each core in a multi-core processor, the number of L1 instruction cache misses can be easily obtained by using static analysis techniques for instruction caches [3]. By using the algorithm depicted in Figure 3, we can statically categorize the L2 instruction accesses for each basic block by considering the possible inter-core interferences in the shared L2 cache, which are then used to compute the worst-case number of L2 instruction misses for the program (i.e., the real-time thread) by using the ILP (Integer Linear Programming) equation as shown in Equation 1.

In Equation 1, m_i is the number of L2 cache misses (i.e., "always misses" [3]) of basic block i, b_i is the number of times the basic block i is executed, and $b\_always\_except\_one_i$ is the number of misses caused by "always-except-one hits", which is only determined by the execution of basic block i. More specifically, if the basic block i is executed, the number of misses is the sum of "always-except-one hits" in this basic block; if basic block i is not executed, then the number of miss is zero. Therefore, b_i^* is 1 if basic block i is executed or 0, otherwise.

$$Cache\_Misses = \sum m_i \times b_i + \sum b\_always\_except\_one_i \times b_i^* \quad (1)$$

3.2. Pipeline Analysis

The static analysis of both L1 and L2 caches provides basis for the pipeline analysis to determine the worst-case latency of each instruction at different pipeline stages, as depicted in Figure 6. For any L2 instruction access (which obviously must be a L1 miss), the pipeline latency will be updated based on its categorization by considering possible conflicts from other co-running threads. As can be seen from Figure 6, function *conflict_in_loop* returns true if the *set* is used by other cores and at the same time the references are from loops, which will convert "always hits" to "always misses". Similarly, function *conflict_in_program* returns true if the *set* is used in other cores no matter where it is used, which will convert "always hits" to "always-except-one hits". For each L2 reference to an "always miss" instruction, the L2 miss penalty will be added into the pipeline latency, in addition to the L1 miss penalty. For L2 instruction that is categorized as "always-except-one hits", the L2 miss latency is only added into the pipeline latency for the first time this loop instruction is accessed. Also, for L2 accesses that are statically categorized as "misses", the L2 miss penalty will be added into the pipeline latency.

Based on pipeline analysis, the cost of each basic block can be determined, which is used in the objective function given in (2). In this function, c_i is the cost of basic block i, and b_i is the number of execution of basic block i. The objective of this function is to maximize its value to obtain the worst-case execution cycles.

$$\sum c_i \times b_i \quad (2)$$

3.3. Path Analysis

The path analysis determines possible paths of a program based on the control flow constraints. As shown in equation (3), in_flow_i is the sum of the edges coming into basic block i and out_flow_i is the sum of edges coming out of basic block i. Both of them should be equal to the number of execution times of basic block i.

$$\sum in\_flow_i = \sum out\_flow_i = b_i \quad (3)$$

Finally, by put together equations (1), (2) and (3), the WCET of the real-time thread can be calculated by using a ILP solver.

4 Evaluation Methodology

The WCET analysis for multi-core processors is based on extending Chronos timing analysis tool [2]. Chronos is originally a single core WCET analysis tool, which targets SimpleScalar architecture. We have extended it to implement the proposed inter-core cache static analysis and pipeline analysis for multi-core processors, as shown in Figure 7. We use gcc to compile two threads (i.e. a real-time thread and a non-real-time thread) into ELF format targeting MIPS R3000 architecture, which can be run on SESC simulator [18] to obtain simulated performance. Since Chronos [2] originally targets SimpleScalar binary code, which is based on COFF format, the front end of Chronos has been retargeted to support SESC binary code based on ELF format, which has been implemented in the dissemble stage in Figure 7. After disassembling the binary code, the CFG (Control Flow Graph) for each individual procedure is constructed. Then Chronos [2] translates the CFG into *transformed* CFG, which is constructed by traversing the call graph of the program and combining individual CFGs into a global CFG. We extend the cache miss analysis in [3] to support the shared L2 cache analysis for multi-core chips. We use a commercial ILP solver – CPLEX [20] to solve the ILP problem to obtain the estimated WCET.

To compare the worst-case performance with the average-case performance (i.e., the simulated performance based on typical inputs), we use SESC simulator [18] to simulate a dual-core processor as depicted in Figure 1, in which each core is a 4-issue superscalar processor with 5 pipeline stages. The important parameters of the dual-core memory hierarchy are given in Table 1. The benchmarks are selected from SNU real-time benchmarks [19].

	simu			wcet			
Benchmarks	L1 miss	L2 miss	Cycle	L1 miss	L2 miss	Cycle	WCET/Simu cycle ratio
bs	19	15	1738	43	27	3110	1.789
fibcall	13	11	1386	28	17	2247	1.621
insertsort	28	25	3407	58	33	4720	1.385
matmul	43	38	6287	77	48	9439	1.501
qurt	152	95	11511	287	175	20079	1.744

Table 2. Comparing the simulated L1 and L2 misses and execution cycles results with the analyzed WCET results.

	size	bsize	assoc	latency
L1-i-cache	512	16	1	10
L1-d-cache	perfect			
L2-u-cache	2k	32	1	100

Table 1. Configurations of the dual-core chip memory hierarchy.

5 Experimental Results

Table 2 compares the execution cycles, the number of L1 misses and the number of L2 misses between the observed results through simulation and the estimated results through WCET analysis. In this experiments, we choose five real-time benchmarks (i.e., `bs`, `fibcall`, `insertsort`, `matmul` and `qurt`) from SNU benchmark suite [19], and another benchmark `adpcm-test` is used as a non-real-time benchmark, which is executed simultaneously with each real-time benchmark on the dual-core processor. Since our concern is to obtain the worst-case performance for the real-time benchmarks, Table 2 only shows the simulated and analyzed results for those five real-time benchmarks, by taking into account the L2 cache interferences from `adpcm-test`.

As can be seen in the last column of Table 2, the estimated WCET is not too far from the observed WCET for most benchmarks. The overestimation in our WCET analysis mainly come from three sources. Firstly, the worst-case execution counts of basic blocks estimated through ILP calculation are often larger than the actual execution counts during simulation. Secondly, cache static analysis approach [3] used for the L1 instruction cache analysis is very conservative. As can be seen in Table 2, the estimated number of L1 misses is much larger than the simulated number of L1 misses, which will not only directly increase the estimated WCET, but also lead to overestimation of L2 misses. Thirdly, our static L2 instruction miss analysis does not consider the timing of interferences from other threads (i.e., *when* other threads may interference). In the actual simulation, although there may be some L2 instruction interferences between two threads, as long as the possible "sharing" of L2 cache blocks occurs at separated time intervals, the actual L2 cache performance of the real-time thread will not be impacted. Finally, because the miss latency of L2 is much larger than that of L1, even slight overestimation of L2 misses could have large impact on the estimated WCET.

Due to the difficulty of analyzing the inter-thread cache interferences and bounding the worst-case performance of the shared L2 caches in a multi-core chip, an obvious solution is to simply disable the shared L2 cache (i.e., assuming that every access to the L2 cache is a miss), which provides the reference values we compare the results of our analysis to. Table 3 compares the estimated WCET by assuming all L2 accesses are misses with the WCET estimated by our approach. As we can see, by statically bounding the L2 cache instruction interferences, the estimated WCET is much smaller than the results by assuming all the L2 accesses are misses, indicating the enhanced tightness of WCET analysis. This improvement is because our approach can reasonably estimate the upper bound of the L2 instruction misses by considering inter-core interferences, which can be seen from Table 2 by comparing the simulated number of L2 misses (i.e., column three) with the estimated number of L2 misses (i.e., column six).

```
Procedure Latency Calculation
INPUT: inst
OUTPUT: lat
BEGIN
IF inst IS l1_miss THEN
  IF inst IN loop THEN
    IF inst IS l2_hit THEN
      set = calc_set(inst);
      conflict = conflict_in_loop(set, other_cores);
      IF conflict IS true THEN
        lat = l1_miss_lat + l2_miss_lat + 1;
       ELSE
        conflict = conflict_in_program(set, other_cores);
        IF conflict IS true THEN
          lat = l1_miss_lat + l2_miss_lat + 1;
          mark(inst,always_except_one);
        ELSE
          lat = l1_miss_lat + 1;
        END
       END
    ELSE  /* l2 miss already*/
      lat = l1_miss_lat + l2_miss_lat + 1;
    END
  ELSE   /* l1 miss not in the loop*/
    IF inst IS l2_hit THEN
      set = calc_set(inst);
      conflict = conflict_in_program(set, other_cores);
      IF conflict IS true THEN
        lat = l1_miss_lat + l2_miss_lat + 1;
      ELSE
        lat = l1_miss_lat + 1;
      END
    ELSE   /*l2 miss*/
      lat = l1_miss_lat + l2_miss_lat + 1;
    END
  END
END
END
```

Figure 6. Algorithm to calculate worst-case instruction latency in a dual-core processor with a shared L2 cache.

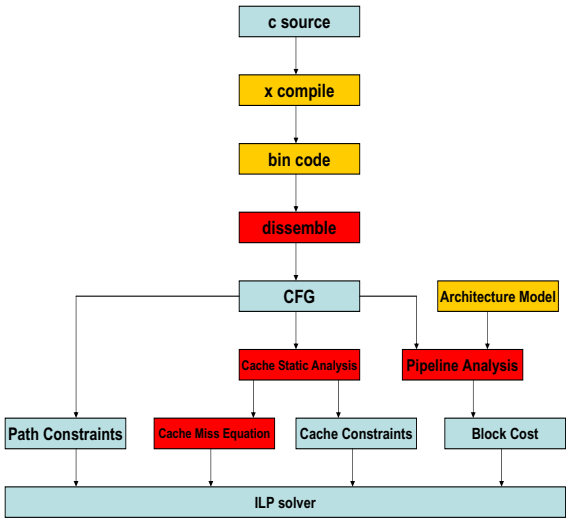

Figure 7. Extension of Chronos to support the WCET analysis of multi-core processors. Note that the extended functions are shown in dark color.

6 Concluding Remarks

This paper presents a novel and effective approach to bounding the worst-case performance of multi-core chips with shared L2 instruction caches. To accurately estimate the runtime inter-core instruction interferences between different threads, we propose to categorize L2 accesses by exploiting program control flow information (i.e., instructions in loops vs. instructions not in loops). The cache analysis results (including the shared L2 cache) are then integrated with the pipeline analysis and path analysis through ILP equations to obtain the worst-case execution cycles. Our experiments indicate that the proposed approach can reasonably bound the worst-case performance of threads running on multi-core processors by considering the inter-thread interferences due to instruction accesses to the shared L2 cache by different co-running threads. Also, compared with the approach by simply disabling L2 caches to avoid interferences, our approach can provide much better worst-case performance for real-time benchmarks.

In our future work, we plan to further enhance the tightness of the static analysis for shared L2 caches. Specifically, we would like to take into account the time ranges of interferences to minimize the overestimation of worst-case instruction interferences between threads. Also, we plan to investigate shared data cache analysis for multi-core chips based on prior work on data cache timing analysis for single-core processors [11, 12, 13, 14], which can then be integrated with this work to fully analyze the worst-case performance of shared caches for multi-core processors.

Benchmarks	All_misses	Our_approach	ratio
bs	4910	3110	0.633
fibcall	3347	2247	0.671
insertsort	7320	4720	0.645
matmul	12539	9439	0.753
qurt	32279	20079	0.622

Table 3. Comparing the WCET results by assuming all L2 accesses are misses and by using our static analysis approach. Column 4 (i.e., ratio) is the ratio of the results by our approach to the results by assuming all L2 accesses are misses.

Acknowledgment

This work was funded in part by the NSF grant CNS 0720502 and IBM 2007 Real-time Innovation Award. We would like to thank the anonymous referees for the detailed comments that helped us improve the paper.

References

[1] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckman, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, P. Stenstrom. The Worst-case execution time problem - overview of methods and survey of tools. In ACM Transactions on Embedded Computing Systems, January 2007.

[2] X. Li, Y. Liang, T. Mitra and A. Roychoudhury. Chronos: a timing analyzer for embedded software. http://www.comp.nus.edu.sg/rpembed/chronos, October 2007.

[3] C. Ferdinand and R. Wilhelm. Fast and efficient cache behavior prediction for real-time systems. In Real-Time Systems, 17((2/3), 1999.

[4] X. Li, A. Roychoudhury, and T. Mitra. Modeling out-of-order processors for software timing analysis. In Proc. of the 25th IEEE International Real-Time Systems Symposium (RTSS'04), 2004.

[5] C. A. Healy, D. B. Whalley, and M. G. Harmon. Integrating the timing analysis of pipelining and instruction caching. In Proc. of the IEEE Real-Time Systems Symposium, December 1995.

[6] F. Stappert, A. Ermedahl and J. Engblom. Efficient longest execution path search for programs with complex flows and pipeline effects. In Proc. of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES), 2001.

[7] Y. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In Proc. of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Real-Time Systems, 1995.

[8] Y. T. S. Li, S. Malik and A. Wolfe. Cache modeling and path analysis for real-time software. In Proc. of the 17th Real-Time Systems Symposium, 1996.

[9] G. Ottosson and M. Sjodin. Worst-case execution time analysis for modern hardware architectures. In Proc. of ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems, June 1997.

[10] P. Puschner and A. Burns. A review of worst-case execution-time analysis. Journal of Real-Time Systems, 18(2/3):115-128, May 2000.

[11] R. White, F. Muller, C. Healy, D. Whalley, and M. Harmon. Timing analysis for data caches and set-associative caches. In Proc. of the IEEE Real-Time Technology and Applications Symposium, June 1997.

[12] H. Ramaprasad and F. Mueller. Bounding worst-case data cache behavior by analytically deriving cache reference patterns. In Proc. of the IEEE Real-Time and Embedded Technology and Applications Symposium, 2005.

[13] T. Lundqvist and P. Stenstrom. A method to improve the estimated worst-case performance of data caching. In Proc. of the 6th International Conference on Real-Time Computing Systems and Applications (RTCSA'99), Dec. 1999.

[14] J. Staschulat and R. Ernst. Worst case timing analysis of input dependent data cache behavior.

In Proc. of the 18th Euromicro Conference on Real-Time Systems (ECRTS06), 2006.

[15] C. Berg, J. Engblom and R. Wilhelm. Requirements for and design of a processor with predictable timing. WDSPB, 2004.

[16] C. Liu, A. Sivasubramaniam and M. T. Kandemir. Organizing the last line of defense before hitting the memory wall for CMP. In Proc. of HPCA, 2004.

[17] J. Chang and G. Sohi. Cooperative caching for chip multiprocessors. In Proc. of the 33rd Annual International Symposium on Computer Architecture, 2006.

[18] J. Renau et al. SESC simulator. http://sesc.sourceforge.net, Jan. 2005.

[19] Homepage of SNU real-time benchmark suite. http://archi.snu.ac.kr/realtime/benchmark/, Oct 2007.

[20] Homepage of CPLEX. http://www.ilog.com/products/cplex/, Oct 2007.

[21] J. Calandrino, J. Anderson, and D. Baumberger. A hybrid real-time scheduling approach for large-scale multi-core platforms. In Proc. of the 19th Euromicro Conference on Real-Time Systems, July 2007.

[22] J. Calandrino, D. Baumberger, T. Li, S. Hahn, and J. Anderson. Soft real-time scheduling on performance asymmetric multi-core platforms. In Proc. of the 13th IEEE Real-Time and Embedded Technology and Applications Symposium, April 2007.

[23] J. H. Anderson, J. M. Calandrino, and U. Devi. Real-time scheduling on multi-core platforms. In Proc. of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium, April 2006.

[24] T. Lundqvist and P. Stenstrom. Timing anomalies in dynamically scheduled microprocessors. In Proc. of Real-Time System Symposium, 1999.