Fangyuan CHEN, Dongsong ZHANG, Zhiying WANG

School of Computer Science and Technology, National University of Defense Technology , China

Characterizing the Inter-Thread Interference of Multi-Core Architectures for Accurate WCET Estimations of Real-Time Applications

Abstract. *Worst-Case Execution Time (WCET) estimation of an application in a real-time system becomes more difficult due to inter-thread interference in shared resources multi-core architectures. This paper proposes an iterative approach to analyze cache interference based on circular dependencies between inter-thread interference and instruction fetch time. Our experiments indicate that the proposed approach can reasonably estimate inter-thread interference in shared caches and improve the tightness of WCET estimation by an average of 17.5%.*

Streszczenie. *W artykule zaproponowano iteracyjną metodę do analizy ukrytych interferencji bazującą na cyrkularnych zależnościach między wątkiem interferencji i czasie dostarczania instrukcji. (**Charakterystyka interferencji przy architekturze wielordzeniowej do dokładnego określania parametru WCET w czasie rzeczywistym**)*

Keywords: Inter-Thread Interference, Multi-Core Architectures, Shared Cache, WCET.
Słowa kluczowe: WCET – worst case execution time (czas realizacji w najgorszym przypadku)

Introduction

With the growing demand of high performance by high-end real-time applications, it is expected that multi-core processors will be increasingly used in real-time systems to achieve high performance/throughput cost-effectiveness. For real-time systems, it is crucial for schedulability analysis and performance checking to obtain the WCET. However, multi-core platforms aggravate the complexity of WCET analysis due to the possible inter-thread interference in shared resources. Therefore, to provide safe and tight WCET estimations it is important to model and analyze shared resources in multi-core architectures.

To the best of our knowledge, recent research efforts into the analysis of inter-thread interference in shared cache without reckon on instruction fetch time [1, 2, 3, 4]. Recent research efforts make a simple common assumption: the instructions of co-running threads that map to the same shared cache line must interfere with each other. However, in practice, interference may not always occur because of other factors, such as execution order and execution timing. Therefore, the above mentioned assumption is too pessimistic to obtain tight WCET estimation.

We manage to reasonably estimate inter-thread interferences by introducing timing relation analysis into an address mapping method. Based on our previous work5,6, we propose an approach to tighten WCET estimation using an iterative method. The proposed approach uses a shared cache status of the last iteration to analyze the next iteration and obtains a tighter inter-thread interference estimate. Our experimental results indicate that the proposed approach can improve the tightness of WCET estimation.

This paper is organized as follows. Section Motivation analyzes the circular dependencies between instruction fetch time and inter-thread interference in shared cache. Section Overview of Our Approach describes an overview of the proposed iterative approach. Section Analysis Components gives a detailed analysis of the components of our analysis framework. In Section Evaluation Methodology, we give the experimental results and discuss them. Finally, in Section Conclusions, we conclude the paper.

Motivation

In our previous work, we studied inter-thread interference in shared cache based on instruction fetch timing [5, 6]. During runtime execution, instruction fetch time determines the behavior of cache access, by which inter-thread interferences in shared cache can be determined. However, the exact instruction fetch time can only be obtained dynamically at runtime, not with static analysis. Therefore, based on research [7] on pipeline analysis, our approach bounds the time interval of an instruction fetch (IF) operation through static analysis to compute the operation's earliest and latest instruction fetching time. Specifically, given instruction *I*, *I*'s earliest fetch time is denoted as $earliest(t_{IF(I)}^{start})$ and *I*'s latest fetch time is denoted as $latest(t_{IF(I)}^{start})$. Instruction Fetch Time (IFT) is our process that uses the statically computed time interval of each instruction and the time order relationship between instructions to determine inter-thread interference in shared cache. Using IFT, we deduce cache access behaviors, which imply inter-thread interference in shared cache. In our previous work [5, 6], we found the interference status affects thread execution; however, in turn, thread execution time may change inter-thread interference.

We assume a dual-core architecture with two levels of cache. Each processor core has a private L1 cache and share the same L2 cache. To simplify the model, the cache line size of L1 equals that of an instruction; as a result, all instructions miss in the L1 cache and need to access shared L2 cache. The size of shared L2 cache line equals that of two instructions. L1 and L2 are directly mapped. L1 miss latency is set to be 0. L2 hit and miss latencies are assumed to be 10 and 20 cycles respectively.

Fig.1 illustrates the circular dependency between instruction fetch time and inter-thread interference in shared cache. In Fig. 1(a), instruction *a0* performs IF operation at time 0 and completes at time 20. Only at time 20 can instruction *a1* start to perform its IF operation, that is, $earliest(t_{IF(a1)}^{start}) = latest(t_{IF(a1)}^{start}) = 20$. Similar analysis can be performed on instructions *a2*, *a3*. That is,

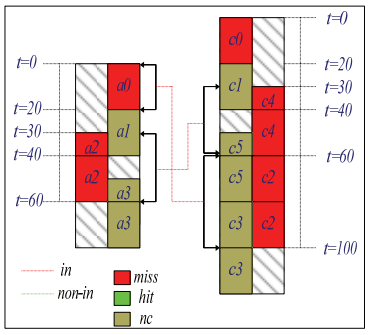
$$earliest(t_{IF(a2)}^{start}) = 30\ latest(t_{IF(a2)}^{start}) = 40\ ,$$

$$earliest(t_{IF(a3)}^{start}) = 50\ latest(t_{IF(a3)}^{start}) = 60\ .$$

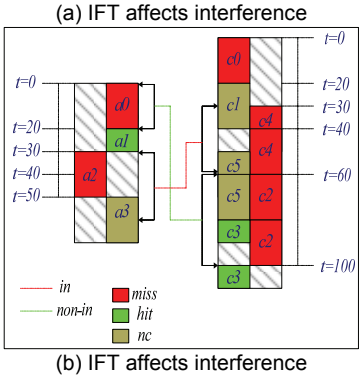
Similar analysis is performed on instructions from thread *c*. We get IFTs of *c2* and *c3*:

$$earliest(t_{IF(c2)}^{start}) = 60\ latest(\ t_{IF(c2)}^{start}) = 80\ ,$$

$$earliest(t_{IF(c3)}^{start}) = 80 latest(\ t_{IF(c3)}^{start}) = 100\ .$$



(a) IFT affects interference



(b) IFT affects interference

Fig. 1 The circular dependency between inter-thread interference and IFT.

From the IFTs we find that *c2* and *c3* must access shared cache after *a0* and *a1*. Thereby they do not interfere with each other. The deduced inter-thread interference is shown in Fig. 1(b) with *c3* colored green to indicate that it hits in the L2 cache.

In turn, we find inter-thread interference also affects IFTs. Inter-thread interference determines the shared cache hit/miss categorization, which affects the execution time of cache accesses. Furthermore, the execution time of current cache accesses affects that of the next cache access as well as IFTs. That is, inter-thread interference affects IFTs. For example, under the initial pessimistic assumption (all instructions are interfered) in Fig. 1(a), we obtain the maximum possible execution time range through which the inter-thread interference is deduced as shown in Fig. 1(b). So the shared cache accesses of *a1* and *c1* are reset to be hit, as shown in Fig. 1(b). For the next instruction *a2*, because *a1* is a shared cache hit, *a2* starts IF operation at time 30, excluding the possible IFT 40 in Fig. 1(a). Consequently the IFT of *a3* is 50, excluding the possible IFT 60 in Fig. 1(a). Hence it is concluded that inter-thread interference may affect IFTs.

Overview of Our Approach

In Section 2 we discussed the circular dependency between inter-thread interference and IFT, which requires us to develop an iterative solution. We perform shared cache analysis without consideration of inter-thread interference. Based on this shared cache hit/miss classification we proceed to our iterative process.

The first step of the iterative process is to analyze shared cache hit/miss classification based on current inter-thread interferences. For current shared cache hit/miss classification, we refine it based on current inter-thread interferences obtained from previous iteration. Then we can get a new shared cache hit/miss classification which is the base of the next step. The status "*in*" means that current instruction is interfered by co-running threads and becomes shared cache nc from hit, "*non-in*" the current instruction is not interfered with and remains as a shared cache hit. In the first iteration, we initialize inter-thread interference to perform future analysis. We do a safe and pessimistic initial assumption on inter-thread interferences: all L2 hits are interfered by co-running threads and become L2 nc to obtain maximum possible time range to guarantee safe WCET estimation.

Then we proceed to static timing analysis to compute instruction fetch times. According to the new shared hit/miss classification the latency of IF operation is computed. Then earliest and latest times of instructions can be obtained. This work has been done in our previous work.

Analysis of inter-thread interferences is performed based on instruction fetch time (Details in 4). At last we check the inter-thread interferences pattern. If the interferences pattern is changed (we try to eliminate some interferences through our proposed Theorem 1), the shared cache analysis has to be repeated. Otherwise, we come into final step to compute WCET estimation based on the inter-thread interferences which is obtained by the last iteration. The iterative analysis refines inter-thread interferences to obtain a tight WCET estimation.

Analysis Components

- Analysis of Inter-Thread Interferences

We try to reasonably bound inter-thread interferences in shared cache by IFT. We proposed a sufficient but not necessary condition of noninterference.

Theorem 1: For any instruction *aj* which is private cache miss, it is a shared cache hit caused by locality of the shared cache access of instruction *ai*. Instruction *bm* from co-running thread is mapped to same shared cache line as *aj*. There is a sufficient, but not necessary condition of inter-thread noninterference in shared cache:

$$earliest(t_{IF(bm)}^{start}) > latest(t_{IF(aj)}^{start}) \parallel latest(t_{IF(bm)}^{start}) < earliest(t_{IF(ai)}^{start}) \quad (1)$$

Proof: The proof is omitted due to limited space.

In shared cache multi-core architectures, we proposed a novel approach to analyze the inter-thread interferences based on Theorem 1.

```
Algorithm 1. Interference_Analysis (inst)
1       if inst is private cache miss then
2          if (inst is shared cache hit) then
3          set =L2_set (inst); Ref = Conflict(set);
4          if (Ref =NULL) then
5             interfering_status=non-in;
6          else
7             IFT_inst= IFT (inst);
8             flag=0;
9             for any ref belongs to Ref do
10            IFT_ref = IFT (ref);
11            non-in=IFT_Analysis(IFT_inst, IFT_ref);
12            if (! non-interfere) then
13               interfering_status= in;
14               flag=1; break;
15            end for
16            if(flag=0)
17            interfering_status=non-in;
```

If a private L1 cache miss occurs during threads execution, then processor cores will access shared cache and inter-thread interferences in shared cache may happen. Based on the L1 cache miss, we begin to analyze shared cache access. When the access to shared cache (for example shared L2 cache) misses due to the thread itself, the instruction in shared cache is a L2 miss and we do not need to analyze runtime inter-thread interference. When the access to shared cache hits, we need to check whether the hit status is changed by co-running threads. Inter-thread interference analysis is described in Algorithm 1.

interfering_status keeps track of the interfering status of current instruction *inst*. Interfering instruction set *Ref* records all the instructions which are mapped to the same shared cache line as *inst*. It is computed according to the set of *inst* and address analysis (line 3). Function *IFT* ( ) is used to obtain instruction fetch time (Details in Section 4). We perform static analysis based on IFT to determine the interfering status: If there is an instruction from the interfering set *Ref* that does not satisfy Theorem 1(function *IFT_Analysis* ( ) analyzes the interfering status based on Theorem 1), then *inst* is interfered and we set the interfering status of *inst* to be "*in*" (line 9 to 15); (2) Otherwise, all instructions will not interfere with *inst* and the interfering status is "*non-in*" (line 16 to 17).

- Iteration

In this paper we try to refine inter-thread interferences by iteration, as shown in Algorithm 2. For any instruction inst, we obtain its interfering statuses based on current IFT through inter-thread interference analysis (line 4). The obtained interfering status is compared with current status to check changed or not (line 5 to 6). If there is an interfering status which is changed, we need to repeat the static analysis. Otherwise, we need not to do the analysis again (line 9). Now we proceed to prove that the iterative analysis shown in Fig. 2 terminates.

```
Algorithm2 Iterative_Analysis
1       changed =0;
2       do
3         for each instruction inst do
4         temp_status =Interference_Analysis(inst)
5         if ( temp_status!= interfering_status) then
6           changed =1;
7           interfering_status = temp_status ;
8         end for
9       while (changed ==1);
```

Theorem 2: For all inter-thread interferences, the interfering status of "*non-in*" monotonically increases and "*in*" monotonically decreases across different iterations of our analysis framework.

Proof: We prove by induction.

Base Case: In the Initialization, all the inter-thread interferences are initialized to be "*in*" and there are no interfering statuses of "*non-in*". So in the first iteration, the "*non-in*" status cannot be smaller and the "*in*" status cannot be larger. Therefore for base case the interfering status of "*non-in*" monotonically increases.

For IF operation, under the interfering status of "*in*", all the lowest latency is set to be shared cache hit and the highest latency is set to be shared cache miss. Thus the lowest latency cannot become smaller and the earliest start time of IF operation will not be changed [8]. Meanwhile for the latest start time, because the monotonic increase of "*non-in*" and monotonic decrease of "*in*", the highest latency cannot become larger and the latest start time of IF operation can only become smaller.

Induction Step:

1) We need to show that the interfering status of "*non-in*" remains "*non-in*" from iteration *k* to iteration *k+1*. Due to limited space we omit the proof.

2) We proceed to show that the interfering status of "*in*" may be changed to "*non-in*" from iteration *k* to iteration *k+1*. Due to limited space we omit the proof.

We conclude that the number of interfering status of "*non-in*" maybe increase from iteration *k* to iteration *k+1* and the interfering status of "*in*" monotonically decreases from iteration *k* to iteration *k+1*.

As interfering status of "*in*" decreases monotonically across iterations, the analysis must terminate.

Evaluation Methodology

The WCET analysis for multi-core processors is based on extended Chronos [9] timing analysis tool. Chronos is originally a single core WCET analysis tool. We have extended it by introducing shared cache analysis to implement WCET analysis for multi-core processors.

We have performed experiment for a dual-core platform where each core is a 2-issue superscalar processor with 5 pipeline stages. The L1 and L2 miss latencies are set to be 10 cycles and 100 cycles, respectively. The benchmarks are selected from SNU [10] real-time benchmarks.

We chose *matmul* as the co-running thread. We compare our method, labeled as WCET-Timing, with 1) our previous work without iteration[5], labeled as WCET-w/o-iteration, and 2) research [8] which uses iterative method to analyze inter-thread interferences in terms of lifetime overlap, labeled as WCET-lifetime.

Table 1 and Table 2 compare the WCET-w/o-iteration which does not adopt the iterative method, with the WCET estimated by our approach, WCET-Timing, which considers the circular dependency between inter-thread interferences and IFT. Inter-thread interference may change IFT, and in turn IFT may affect the former. For the method without iteration (WCET-w/o-iteration), improvement of tightness can be obtained based on IFT, however this result has an effect on the next IFT. Our iterative approach (WCET-Timing) furthers utilize the effect to obtain tight WCET estimation. Our approach refines inter-thread interference across iterations, which affords an opportunity to obtain a tighter WCET estimation, as shown in Table 1.

Table 1. Compare with the Method without Iteration

bench	L2 hit	noninterference		Ratio (%)
		WCET-w/o-iteration	WCET-Timing	
bs	3	0	1	0.33
insertsort	2	0	2	1.0
matmul	4	0	3	0.75
fibcall	1	0	1	1.0
jdcint	2	0	2	1.0
Average				0.81

For example, for *matmul*, there are 4 misses caused by the interferences of *matmul-unrolling* without considering iterative method while only 1 misses when adopting iterative method to analyze the circular dependency between inter-thread interferences and IFT. Average 81% of inter-thread interferences are improved by our proposed iterative approach.

Table 2 shows the tightness improvement of WCET estimations. Experimental results indicate that our proposed approach can improve the tightness of WCET estimations by 7% on average, as compared with the method without iteration.

Table 2. Compare with the Method without Iteration

bench	WCET-w/o-iteration	WCET-Timing	Ratio (%)
bs	2092	1992	95.2
insertsort	2767	2567	92.7
matmul	6221	5921	92.1
fibcall	1490	1390	93
jdcint	4505	4305	91.8
Average			93.0

In order to evaluate the validation and tightness of our proposed approach, we compare the results from research [8], which considers threads with overlapped lifetimes always interfere with each other, and results obtained by our approach, as shown in Table 3.

Table 3. Comparation Results

bench	WCET-lifetime	WCET-Timing	Ratio (%)
bs	2292	1992	86.9
insertsort	3567	2567	72.0
matmul	6721	5921	88.1
fibcall	1790	1390	77.7
jdcint	4905	4305	87.8
Average			82.5

As can be seen in the last column of Table 3, the estimated WCET obtained by our approach is not too far from the observed WCET for most benchmarks. However, for the method only considering lifetime, it is farther than our approach. Compared with WCET-lifetime, our proposed approach can improve the tightness of W CET estimation by 17.5% on average.

Conclusions

This paper proposes a novel approach to analyze WCET in multi-core systems. Our iterative approach incorporates both instruction fetch time and inter-thread shared instruction cache interference. The iterative method tightens the WCET estimation by refining the inter-thread shared cache interference.

Acknowledgments

We are grateful to Peter F. Sweeney for his valuable constructive comments and helpful advice. This work was supported in part by the Excellent Graduate Innovation of NUDT under Grant No.B100601 and Hunan Provincial Innovation Foundation for Postgraduate under Grant No.CX2010B026.

REFERENCES

[1] J. Anderson, J. Calandrino, and U. Devi. Real-time scheduling on multicore platforms. RTAS, 2006.

[2] B. K. Bershad, B. J. Chen, D. Lee, and T. H. Romer. Avoiding conflict misses dynamically in large direct mapped caches. In ASPLOS, 1994.

[3] Nan Guan, Martin Stigge, Wang Yi and Ge Yu. Cache-Aware Scheduling and Analysis for Multicores. In EMSOFT, 2009.

[4] J. Yan and W. Zhang. WCET analysis for multi-core processors with shared L2 instruction caches. In Proc. of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium, April 2008.

[5] F.Y. Chen, D.S. Zhang and Z.Y. Wang. Static Analysis of Run-Time Inter-Core Interferences for Consecutive or Inconsecutive Concurrent Programs in Shared Cache Multicore Architectures. In Proc. of the 3rd International Conference on Computer Design and Applications (ICCDA 2011), May 2011.

[6] F.Y. Chen, D.S. Zhang and Z.Y. Wang. Static Analysis of Run-Time Inter-Thread Interferences in Shared Cache Multi-Core Architectures based on Instruction Fetching Timing. In Proc. of the IEEE International Conference on Computer Science and Automation Engineering (CSAE 2011), June 2011.

[7] Li Xianfeng, Microarchitecture Modeling for Timing Analysis of Embedded Software. Dissertation for the Degree of Doctor of Philosophy in Computer science, National University of Singapore, 2005.

[8] S. Chattopadhyay, A. Roychoudhury, and T. Mitra. Modeling shared cache and bus in multi-cores for timing analysis. In SCOPES, 2010.

[9] 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.

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

Authors

Fangyuan Chen: received her B.S. degree from Shangdong University in 2005. She received M.S. degree from National University of Defense Technology in 2008. Both in Computer Science. She is currently working toward the Ph.D. degree in the College of Computer Science, National University of Defense Technology, Changsha, China. Her current research interests include real-time systems and computer architectures. She is a student. Email:fychen@nudt.edu.cn.