Optimizing Instruction Prefetching to Improve Worst-Case Performance for Real-Time Applications

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

Received 3 November 2008; Accepted 11 February 2009

While the average-case performance is important for general-purpose applications, worst-case performance is crucial for real-time systems to ensure schedulability and reliability. Recent work has shown that simple prefetching techniques such as the Next-N-Line prefetching can benefit both average-case and worst-case performance; however, the improvement on the worst-case execution time (WCET) is rather limited and inefficient. This paper presents two instruction prefetching approaches that are specially designed to enhance the worst-case performance, including the loop-based prefetching and WCET-oriented prefetching. Our experiments indicate that both instruction prefetching techniques can achieve better worst-case execution cycles than the Next-N-Line prefetching while having various impacts on the average-case performance.

Categories and Subject Descriptors: C3 [**SPECIAL-PURPOSE AND APPLICATION-BASED SYSTEMS**]
Real-time and embedded systems; J7 [**COMPUTERS IN OTHER SYSTEMS**]: Real time
General Terms: Performance, Reliability
Additional Key Words and Phrases: Instruction prefetching, worst-case execution time, real-time systems

1. INTRODUCTION

Many embedded systems ranging from heart pacemakers to automobile and aircraft controllers require real-time computing. Missing deadlines in those systems may endanger human lives or lead to other severe consequences. Worst-case execution time (WCET) is critical for the schedulability and reliability of real-time systems. The WCET can be typically obtained by using measurement or static analysis. The measurement-based approach, however, is generally unsafe, since one cannot exhaust the measurements on all the possible program paths. By comparison, static WCET

analysis is a promising approach to safely and accurately estimate the WCET for real-time applications [Wilhelm et al. 2008].

Cache memories have been widely used in microprocessors to mitigate the speed gap between the slow memory and the fast processor. While cache memories are detrimental to the time predictability of execution, prior work [Arnold et al. 1994; Healy et al. 1995] shows that the worst-case performance of processors with instruction caches can be reasonably bounded, which significantly outperforms the worst-case performance of a microprocessor without using any instruction cache.

To further improve the instruction cache performance, various instruction prefetching techniques [Smith 1978; Chow et al. 2004; Smith 1982; Smith and Hsu 1992; Pierce and Mudge 1996; Joseph and Grunwald 1997; Luk and Mowry 1998; Xia and Torrellas 1996; Reinman et al. 1999; Srinivasan et al. 2001] can be used. These instruction prefetching techniques generally prefetch instructions that are expected to miss before they are accessed, thereby potentially reducing the number of instruction cache misses or the associated miss penalties. In particular, instruction prefetching is effective at reducing the cold misses, which is important for real-time computing in a multiprogramming context, where instructions of a real-time thread may be evicted by instructions of other real-time or non-real-time tasks. Recent work [Yan and Zhang 2007] shows that the Next-N-Line prefetching [Smith 1978; 1982] can benefit both the average-case and the worst-case execution time; however, the improvement on the worst-case performance is moderate and inefficient. The reason is that the Next-N-Line prefetcher, designed for enhancing the average-case performance, always prefetches the next N cache lines, regardless of the program control flow. This may lead to excessive conflicts between the prefetched instructions and other useful instructions in the cache. Such a cache pollution effect is especially problematic for static timing analysis, since the WCET analyzer does not have the exact runtime information and typically has to conservatively estimate the worst-case cache pollution by considering all the possible instructions that might be affected by the prefetched instructions, which are harmful to the tightness of WCET analysis.

While all the previous instruction prefetching techniques [Smith 1978; 1982; Smith and Hsu 1992; Pierce and Mudge 1996; Joseph and Grunwald 1997; Luk and Mowry 1998] were designed for improving the average-case performance, this paper proposes two instruction prefetching techniques specifically devised for enhancing the worst-case performance, which are particularly useful for real-time applications. More specifically, we study a loop-based approach and a WCET-oriented approach to prefetching instructions efficiently for reducing the worst-case execution time of real-time applications, which can also overcome the deficiencies of the Next-N-Line prefetching [Smith 1978; 1982]. Our experiments indicate that both proposed instruction prefetching techniques can achieve much better worst-case execution cycles than the Next-N-Line prefetching. Also, the loopbased approach outperforms the Next-N-Line prefetching in the average-case performance, while the WCET-oriented approach is not always superior in terms of the average-case performance.

The rest of this paper is organized as the follows. Section 2 introduces the loop-based instruction prefetching strategy. Section 3 presents the WCET-oriented instruction prefetching technique. The evaluation methodology is explained in Section 4, and the

experimental results are given in Section 5. Related work is discussed in Section 6. Finally, we draw conclusions in Section 7.

2. LOOP-BASED INSTRUCTION PREFETCHING

The Next-N-Line prefetcher, proposed by Smith et al. [Smith 1978; 1982], always prefetches the next N cache lines. While this prefetching policy is effective during the sequential execution phase of the program, it becomes less useful or even harmful when the control flow of runtime execution changes. In particular, the Next-N-Line prefetcher [Smith 1978; 1982] is problematic at the boundaries of loops, which are common in real-time applications. Since loops are likely to execute by many times (especially in the worst case), the instructions prefetched after the loop branch (e.g., the branch that leads to the backward edge in the control flow graph [Muchnic 1997]) are useless, if not detrimental, except for the last loop iteration. Moreover, this adverse effect of wrong instruction prefetching will be repeated, until the loop is finished. To address this problem, we propose the loop-based prefetching, which exploits the loop information to enhance the efficiency and performance of the Next-N-Line prefetcher.

The loop-based instruction prefetching is built upon the Next-N-Line prefetcher [Smith 1978; 1982]. In the loop-based prefetching, normally instructions are still prefetched sequentially just like the Next-N-Line prefetcher; however, when a loop branch is encountered, the loop-based prefetcher will prefetch *N lines of instructions from the beginning of the loop* (i.e., the target address of the loop branch), not the next N lines after the loop. The reason is that in the worst case, the loop branch is usually taken except the last iteration.

Consequently, by prefetching instructions from the beginning of the loop rather than the subsequent instructions after the loop body (i.e., on the fall-through path), the direction of the instruction prefetching is kept consistent with the runtime instruction fetching flow (except for the last loop iteration), potentially resulting in better performance.

The architectural support for the loop-directed instruction prefetching is depicted in Figure 1. We extend the traditional Next-N-Line prefetcher by simply adding a few components, including a *loop branch address* register, a control signal *LoopBranch-Enable*, a hardware table, and a multiplexer. The hardware table is used to store the address of each loop branch and the associated loop header (i.e., the first instruction of each loop). Since both loop branches and loop headers can be identified statically at the compilation time [Muchnic 1997], we propose to store those addresses into the hardware table before we run the program. Because typically there are only a small number of static loops in a real-time program (although they may dominate the execution time), the loop address hardware table can be kept small. In our experiments, we use a hardware table with 8 entries, which has insignificant hardware overhead.

To direct the hardware prefetcher at runtime, we propose to use the compiler to detect and annotate the loop branches at the compilation time. This can be simply achieved by using either special opcodes for loop branches or exploiting unused fields in the branch instructions for annotation. At runtime, when a loop branch instruction is being executed, its address is then passed to the *loop branch address* register, and

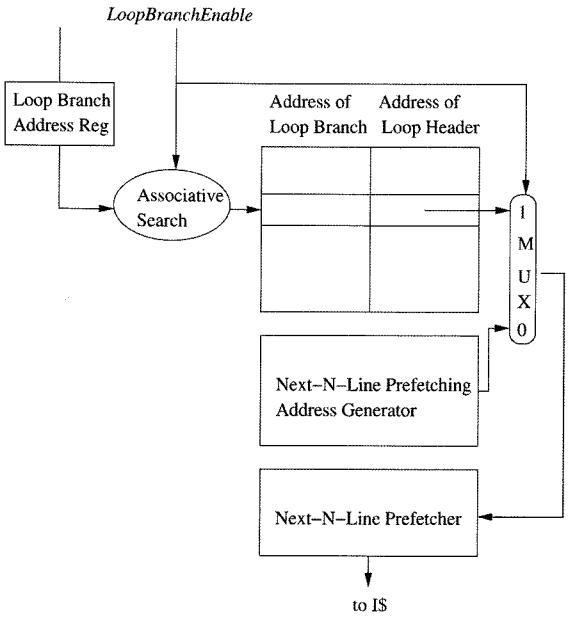


Figure 1. The architectural support for the loop-directed instruction prefetching.

the *LoopBranchEnable* signal is enabled to 1 (normally, *LoopBranchEnable* is 0 for non-loop-branch instructions). As we can see from Figure 1, the *LoopBranchEnable* signal activates the associative search circuit to find the corresponding loop header address (i.e., the target address of the loop branch) in the hardware table. This loop header address is then passed to the multiplexer controlled by the *LoopBranchEnable* signal. Since the *LoopBranchEnable* signal is enabled, the hardware prefetcher will prefetch instructions from the loop header instead of the next instruction (i.e., *PC*+4) after the loop branch. When a non-loop-branch instruction is executed, however, the *LoopBranchEnable* signal will be disabled. In that case, the loop-directed prefetcher will prefetch sequential instructions according to the Next-N-Line prefetching address generator. It should be noted that for a processor that employs branch prediction, the hardware overhead of the loop-directed prefetching can be further reduced, because the hardware table shown in Figure 1 actually functions like a branch target address (BTB) table (but only for loop branches), which may reuse the information in the branch prediction hardware.

3. WCET-ORIENTED INSTRUCTION PREFETCHING

In addition to the loop-based prefetching, which is essentially a hybrid approach by combining the hardware-based Next-N-Line prefetcher and the compiler-directed loop information, we also study a pure software-basedWCET-oriented instruction prefetching whose goal is to specifically optimize the worst-case performance, even probably at the cost of the average-case performance. The idea of this approach is that the static timing analyzer, for instance, the static cache simulation [Arnold et al. 1994; Healy et al. 1995], can statically identify instructions (called *statically missed instructions* in

the rest of this paper) that will lead to instruction cache misses in the worst-case scenario, which can be exploited to efficiently guide the instruction prefetching. Specifically, the static cache simulation classifies the caching behavior of instructions into four different categories based on their conditions. These four categories are summarized below (note more details can be seen in [Arnold et al. 1994; Healy et al. 1995]).

(1) *Always hit:* A reference to an instruction is *always hit* if this instruction is guaranteed to be always in the cache when it is accessed.
(2) *Always miss:* A reference to an instruction is *always miss* if this instruction is guaranteed to be not in the cache when it is accessed.
(3) *First hit:* A reference to an instruction in a loop is *first hit* if the first access to this instruction is a hit, while all remaining references to this instruction are guaranteed to be misses.
(4) *First miss:* A reference to an instruction in a loop is *first miss* if the first access to this instruction is a miss, while all remaining references to this instruction are guaranteed to be hits.

Therefore, based on the static cache analysis, the compiler can insert prefetching instructions in appropriate points in the program to reduce the penalties for those *statically missed instructions*. If these *statically missed instructions* can be prefetched in a sufficient amount of time (i.e., equal to or larger than the instruction cache miss latency), then they are successfully converted into "*always hit*" instructions, thus reducing the number of worst-case cache misses. Alternatively, even if a *statically missed instruction* cannot be prefetched ahead enough to become a hit, prefetching can still reduce the penalty of this miss in some degree, which also benefits the WCET.

3.1 ISA Support

Today's microprocessors often provide non-blocking software prefetching instructions, which can force a cache fill at a specified address in anticipation of an upcoming cache miss [Panda et al. 1997]. In this work, we assume that we can extend the instruction set to specify the address of an instruction that needs to be prefetched, which is called the *instruction prefetching address field* in this paper. This *instruction prefetching address field* is used to specify the relative distance between the address of the prefetched instruction and the current instruction, which can be calculated statically. By default, all the bits of the instruction prefetching address field are zero, indicating no prefetching is needed. Since in this work we only allow the compiler to insert prefetching instruction within the same basic block (see below), the prefetching distance is typically short. Therefore, only a few bits (4 bits are assumed in this paper) are needed to specify the prefetching distance. Note that in some instruction formats such as MIPS ISA, certain fields (e.g., *shamt*) are rarely used by most instructions, which may be exploited to encode the prefetching information without increasing the width of the instruction set.

3.2 Compiler Support

With the support of instruction prefetching in the ISA, the compiler's job is to first call the static cache simulation [Arnold et al. 1994; Healy et al. 1995] to obtain the instruction categorization information. Then for each "*always miss*" instruction, a prefetching instruction (by specifying the *instruction prefetching address field*) is inserted in the appropriate place to boost worst-case performance. It should be noted that it is always safe to insert the prefetching instructions into the program, since they will not change the state of the machine, but to either increase or decrease performance. As aforementioned, in this work, the compiler simply inserts the prefetching instructions within the same basic block to guarantee that they will be definitely executed to improve the worst-case performance, regardless of the runtime control flow[1]. Therefore, the prefetching distance of the WCEToriented approach can be calculated by using Equation 1, in which MP represents the L1 instruction miss penalty, I_p denotes an instruction that needs to be prefetched (i.e., an "*always miss*", "*first miss*" or "*first hit*" instruction), and I_f stands for the first instruction in this basic block whose *instruction prefetching address field* is available[2].

As we can see in Equation 1, the prefetching distance of the WCET-oriented prefetching is the minimum of the L1 instruction cache miss penalty and the schedule time difference between the prefetched instruction and the earliest instruction in this basic block that can store the prefetching distance information. The reason is that prefetching too early (i.e., beyond the L1 instruction cache miss penalty) is unnecessary. And at the same time, the compiler attempts to insert the prefetching instruction as early as possible to maximally reduce the miss latency for enhancing

Table I. Configuration parameters and their values of the base VLIW processor.

Configuration Parameters	Values
Processor	
Functional Units	2 integer FUs
	2 floating-point FUs
	1 load/store unit
	1 branch unit
Register File	16 global registers
Cache and Memory Hierarchy	
L1 Instruction Cache	512 bytes, direct-mapped
	8 byte blocks, 1 cycle latency
L1 Data Cache	perfect
Memory	8 cycle, unlimited size

[1]Note that while inserting the prefetching instructions across basic block boundaries may lead to better WCET, but it may also result in either excessive prefetched instructions inserted in all possible paths or the possibility that the WCET is not improved at all if the prefetched instruction is not inserted on the worst-case path.

[2]It should be noted that in a large basic block, multiple instructions may need to be prefetched; however, each static instruction in the basic block can only carry the prefetching information for one *statically missed instruction.*

the WCET.

$$PD = min(MP, sched(I_p) - sched(I_f)) \quad (1)$$

For instructions that are categorized as "*first miss*" and "*first hit*", we propose to use loop peeling [Muchnic 1997] to peel the first loop iteration. More specifically, for "*first miss*" instructions, prefetching instructions are only inserted for the first loop iteration after loop peeling. This is because for the rest of the loop, this "*first miss*" instruction is guaranteed to be always hits. In contrast, for "*first hit*" instructions, the compiler will insert prefetching instructions for the remaining of the loop iterations, except the first one.

4. EVALUATION METHODOLOGY

We study both the worst-case and average-case performance of different instruction prefetching techniques on a HPL-PD [Kathail et al. 2000] based VLIW processor by using Trimaran compiler/simulator infrastructure. We have modified the backend compiler Elcor and the VLIW simulator to support the Next-N-Line prefetching, the loop-directed prefetching and the WCET-oriented prefetching. The WCET analyses of these instruction prefetching techniques are based on the prior work in [Yan and Zhang 2007], which are implemented as independent modules in Elcor to report the worst-case performance. The average-case performance is obtained by using the VLIWsimulator in the Trimaran framework. The important parameters of the baseline VLIWprocessor are given in Table I. Note that to limit the scope of this study, we assume a perfect data cache (although our work is independent of the data cache) which is also presumed in [Yan and Zhang 2007].

We select six benchmarks from the SNU real-time benchmark suite for the evaluation. All the benchmarks are compiled by using the Trimaran compiler. The salient characteristics of the benchmarks are given in Table II.

Table II. The salient characteristics of the selected Malardalen real-time benchmarks.

Benchmark	Description	Static Instrs	access	misses	Miss Rate
Bmm	multiplies two matrices	552	101157	293	0.29%
Fib mem	Computes a Fibonacci number using a linear recurrence	93	237	41	17.30%
Nested	Sum up the elements in a two-dimensional array	120	2860	76	2.66%
Fibcall	Fibonacci series function	43	208	25	12.02%
Ludcmp	LU decomposition algorithm	265	3799	360	9.48%
Matmul	Matrix multiplication	186	2838	58	2.04%

5. EXPERIMENTAL RESULTS

5.1 Worst-Case Performance Results

Figure 2 compares the worst-case performance of the Next-N-Line prefetching, the loopdirected prefetching and the WCET-oriented prefetching with the prefetching

distance varied from 2 to 4, 8 and 16, which are normalized with the WCET of the base scheme that does not use any instruction prefetching. We use NLP-i (or LP-i) to represent the Next-N-Line prefetching (or loop-directed prefetching) with a prefetching distance i. As shown in Figure 2, the NLP, LP and WCET-oriented prefetching schemes improve the worstcase performance in most cases, except when the prefetching distance is too large (e.g., NLP-16 and LP-16 for `Fibcall`), which turns out to be harmful for WCET. In particular, both the NLP and LP schemes are successful for benchmarks having more instruction caches misses, for instance `Fib mem` and `Fibcall`, whose I-cache miss rates are 17.3% and 12.02% respectively as given in Table II. Also, we observe that the WCET-oriented prefetching is very effective at improving the worst-case performance. Actually, except `Nested` and `Fibcall`, the WCET-oriented prefetching outperforms both the Next-NLine prefetching and the loop-based prefetching for all the different prefetching distances. On average, the WCET-oriented prefetching is better than both the Next-N-Line prefetching and the loop-based prefetching with different prefetching distances, except the LP-8 scheme, which can achieve the best WCET with the prefetching distance equal to the L1 miss penalty.

For the NLP and LP schemes, we observe that when the prefetching distance increases from 0 (i.e., base) to 2 and 4, the number of worst-case execution cycles is reduced. However, we also observe that when the prefetching distance increases beyond 4 (or 8), the Next-N-Line prefetching (or the loop-directed prefetching) results in worse WCET, because of the aggravated instruction cache pollution. By comparing the NLP scheme with the LP scheme with the same prefetching distance, we observe that the loop-directed prefetching is always superior to the Next-N-Line prefetching. The reason is that the LP scheme can reduce cache pollution caused by excessively prefetched instructions outside loops through prefetching the right instructions to support loop execution. Specifically, the best loop-directed prefetching scheme (i.e., LP-8) can reduce the base WCET by 21.7% on average, which is 3.4% more than that

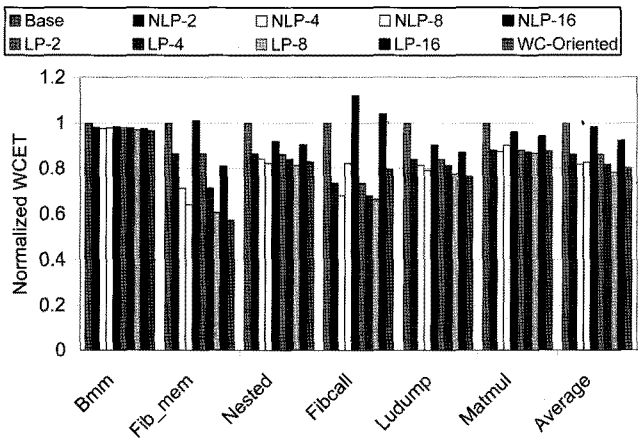


Figure 2. Normalized worst-case execution cycles for different prefetching schemes with the prefetching distance varied from 2 to 4, 8 and 16, which are normalized with the worst-case execution cycles of the Base (without instruction prefetching).

of the best Next-N-Line prefetching scheme (i.e., NLP-4). In contrast to both NLP and LP schemes, whose performance is heavily dependent on tuning the prefetching distance, the WCET-oriented prefetching is independent of the prefetching distance, which can still attain the best WCET for most benchmarks.

5.2 Average-Case Performance Results

Besides the worst-case performance, we also comparatively study the average-case performance of different prefetching schemes by using the simulator. Figure 3 compares the simulated execution cycles of the NLP and LP schemes of different prefetching distances, as well as the WCET-oriented prefetching, which are normalized with the base execution cycles without instruction prefetching. Generally, we observe that for both the NLP and LP schemes, the best average-case performance is achieved when the prefetching distance is 2 (note `Fibcall` and `Ludump` are the two exceptions, whose best performance results are achieved when the prefetching distance is 4). These optimal average-case performance results are in contrast to the best WCET that can only be attained with a larger prefetching distance (i.e., 4 for NLP and 8 for LP), as demonstrated in Figure 2. The reason is that the WCET analyzer normally has to conservatively estimate the benefits of prefetched instructions, while the simulator can accurately measure the effects of cache pollution.

In addition, we find that generally the WCET-oriented prefetching is not very effective at improving the average-case performance, as compared to the best NLP and LP schemes with the optimal prefetching distance. The reason is that both NLP and LP will prefetch multiple instructions (i.e., the next N lines) according to the runtime execution path, while the WCET-oriented prefetching only statically inserts prefetching instructions based on the worst-case performance estimation, which may be different from an average-case program behavior. Moreover, in this work, the WCET-oriented prefetching is constrained to intra-block prefetching, which often limits the scope of prefetching and according the performance improvement.

6. RELATED WORK

Much work on instruction prefetching has focused on improving the average-case performance [Smith 1978; Chow et al. 2004; Smith 1982; Smith and Hsu 1992; Pierce and Mudge 1996; Joseph and Grunwald 1997; Luk and Mowry 1998; Xia and Torrellas 1996; Reinman et al. 1999; Srinivasan et al. 2001]. By comparison, only a few studies have examined the impact of prefetching on the worst-case performance, which is critical for real-time applications. Specifically, Lee et al. conducted the worst-case timing analysis for a buffered prefetch scheme based on timing schema [Lee et al. 1994], which basically replaces the instruction cache and thus is different from our work. Batcher and Walker [Batcher and Walker 2006] proposed the interrupt triggered software prefetching to reduce the number of cache misses due to inter-task interferences. In contrast, this paper studies a loop-directed hardware prefetching and a WCET-oriented software prefetching to enhance the worst-case performance for a single task. Chen et al. [Chen et al. 2001] described an example of cache analysis with instruction prefetching as part of a retargetable timing analysis tool; however, no

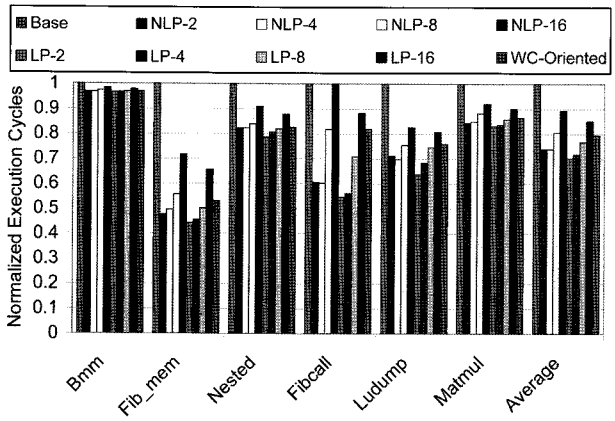


Figure 3. Normalized execution cycles for different prefetching schemes with the prefetching distance varied from 2 to 4, 8 and 16, which are normalized with the base execution cycles without instruction prefetching.

quantitative worst-case performance results are provided in [Chen et al. 2001].

As aforementioned, the closely related work to this paper is [Yan and Zhang 2007], which extensively investigates the impact of the Next-N-Line instruction prefetching on the worst-case execution time. Built upon the prior work in [Yan and Zhang 2007], this paper proposes the loop-directed prefetching and WCET-oriented prefetching, which can achieve better worst-case performance than the Next-N-Line prefetcher, and thus are preferable for real-time applications.

There are also many related works in worst-case execution time analysis for cache memories. Healy et al. studied static cache simulation for analyzing timing behavior of instruction caches accurately [Arnold et al. 1994], which was then integrated with pipeline analysis to derive WCET [Healy et al. 1995]. Mueller studied the use of the static cache analysis to set-associative caches [Mueller 1997]. Li and Malik proposed an integer linear programming based approach to computing WCET bounds of programs on processors with pipelines and caches [Li and Malik 1995; Li et al. 1995]. Ferdinand andWilhelm proposed persistence analysis to estimate the WCET of data caches [Ferdinand and Wilhelm 1998]. Ramaprasad and Mueller exploited the cache miss equations (CME) framework to accurately predict the data cache timing behavior for scalar and non-scalar references whose reference patterns are known at the compilation time [Ramaprasad and Mueller 2005]. Staschulat et al. studied input dependent WCET analysis for data caches [Staschulat and Ernst 2006]. Hardy and Puaut examined worst-case performance analysis for multi-level non-inclusive set-associative instruction caches. Currently, there are also some commercial WCET analysis tools available, such as aiT. A good summary of contributions in the area of WCET analysis can be found in [Wilhelm et al. 2008].

7. CONCLUDING REMARKS

Built upon prior work on [Yan and Zhang 2007], this paper studies both loop-based instruction prefetching and WCET-oriented instruction prefetching that specifically

aims at improving the worst-case performance for real-time applications. Compared with the Next-NLine prefetching [Smith 1978; 1982] that is designed for enhancing the average-case performance, the loop-directed approach can not only relieve cache pollution by not prefetching instructions after the loop branches, but also boosts performance by prefetching the right instructions during the loop execution. On the other hand, the WCET-oriented prefetching is a pure software-based approach that is fundamentally different from the hardware-based Next-N-Line prefetching [Smith 1978; 1982]. The idea of the WCET-oriented prefetching is to exploit the worst-case instruction cache categorization information to efficiently prefetch instructions for reducing WCET.

Our experiments indicate that the WCET-oriented prefetching can attain better WCET than both the Next-N-Line prefetching and most of the loop-based prefetching schemes with various prefetching distances. However, the WCET-oriented prefetching is less effective at improving the average-case performance, as compared to the worst-case performance, because it is specifically designed for prefetching the missed instructions in the worst-case, which is usually different from the average-case behavior. Moreover, we find that the loop-directed prefetching outperforms the Next-N-Line prefetching in both the worst-case and the average-case performance. We believe both the loop-based and WCET-oriented prefetching techniques provide interesting design options for real-time applications to boost the worst-case performance.

In our future work, we would like to further enhance the WCET-oriented instruction prefetching by analyzing the worst-case path to support instruction prefetching across basic blocks without significantly increasing the number of prefetched instructions. Also, it is possible to combine the loop-based and WCET-oriented prefetching approaches to obtain better WCET and energy efficiency for real-time applications.

ACKNOWLEDGMENTS

This work was funded in part by NSF grant CNS 0720502.

REFERENCES

Homepage of snu real-time benchmarks. *http://archi.snu.ac.kr/realtime/benchmark/.*

Hompage of ait worst-case execution time analyzers. *http://www.absint.com/ait/.*

Trimaran homepage. *http://www.trimaran.org.*

ARNOLD, R., MULLER, F., WHALLEY, D., AND HARMON, M. 1994. Bounding worst-case instruction cache performance. *Proc. of the Real-Time Systems Symposium.*

BATCHER, K. AND WALKER, R. 2006. Interrupt triggered software prefetching for embedded cpu instruction cache. *Proc. of RTAS.*

CHEN, K., MALIK, S., AND AUGUST, D. 2001. Retargetable static timing analysis for embedded software. *Proc. of ISSS.*

CHOW, P., HAMMARLUND, P., AAMODT, T., MARCUELLO, P., AND WANG, H. 2004. Hardware support for prescient instruction prefetch. *Proc. of International Symposium on High Performance Computer architecture.*

FERDINAND, C. AND WILHELM, R. 1998. On predicting data cache behavior for real-time systems. *Proc. of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded System.*

HEALY, C., WHALLEY, D., AND HARMON, M. 1995. Integrating the timing analysis of pipelining and instruction caching. *Proc. of the Real-Time Systems Symposium.*

JOSEPH, D. AND GRUNWALD, D. 1997. Prefetching using markov predictors. *Proc. of ISCA.*

KATHAIL, V., SCHLANSKER, M., AND RAU, B. 2000. Hpl-pd architecture specification: version 1.1. *HPL Technical Report.*

LEE, M., MIN, S., AND KIM, C. 1994. A worst case timing analysis technique for instruction prefetch buffers. *Microprocessing and Microprogramming.*

LI, Y. AND MALIK, S. 1995. Performance analysis of embedded software using implicit path enumeration. *Proc. of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Real-Time Systems.*

LI, Y., MALIK, S., AND WOLFE, A. 1995. Efficient microarchitecture modeling and path analysis for real-time software. *Proc. of the 16th Real-Time Systems Symposium.*

LUK, C. AND MOWRY, T. 1998. Cooperative prefetching: compiler and hardware support for effective instruction prefetching in modern processors. *Proc. of MICRO.*

MUCHNIC, S. 1997. Advanced compiler design and implementation. *Morgan Kaufmann Publishers.*

MUELLER, F. 1997. Generalizing timing predictions to set-associative caches. *Proc. of Euromicro Workshop on Real-Time Systems.*

PANDA, P., DUTT, N., AND NICOLAU, A. 1997. Memory data organization for improved cache performance in embedded processor applications. *ACM Transactions on Design Automation and Electronics Systems* 4.

PIERCE, J. AND MUDGE, T. 1996. Prefetching in supercomputer instruction caches. *Proc. of MICRO.*

RAMAPRASAD, H. AND MUELLER, F. 2005. Bounding worst-case data cache behavior by analytically deriving cache reference paerns. *Proc. of the 11th IEEE Real-Time and Embedded Technology and Applications Symposium.*

REINMAN, G., CALDER, B., AND AUSTIN, T. 1999. Fetch directed instruction prefetching. *Proc. of the 32nd International Symposium on Microarchitecture.*

SMITH, A. 1978. Sequential program prefetching in memory hiearchies. *IEEE Computer* 2:7–21.

SMITH, A. 1982. Cache memories. *Computing surveys* 3:473–530.

SMITH, J. AND HSU, W. 1992. Prefetching in supercomputer instruction caches. *Supercomputing.*

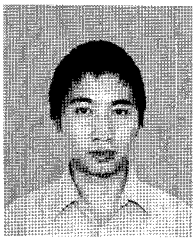
SRINIVASAN, V., DAVIDSON, E., TYSON, G., CHARNEY, M., AND PUZAK, T. 2001. Branch history guided instruction prefetching. *Proc. of the 7th International Conference on High Performance Computer Architecture (HPCA).*

STASCHULAT, J. AND ERNST, R. 2006. Worst case timing analysis of input dependent data cache behavior. *Proc. of the 18th Euromicro Conference on Real-Time Systems (ECRTS06).*

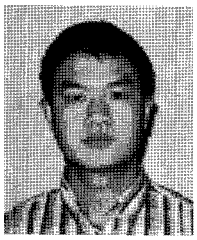
WILHELM, R., ENGBLOM, J., ERMEDAHL, A., HOLSTI, N., THESING, S., WHALLEY, D., BERNAT, G., FERDINAND, C., HECKMAN, R., MITRA, T., MUELLER, F., PUAUT, I., PUSCHNER, P., STASCHULAT, J., AND STENSTROM., P. 2008. The worst case execution time problem - overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems* 3:1–53.

XIA, C. AND TORRELLAS, J. 1996. nstruction prefetching of systems codes with layout optimized for reduced cache misses. *Proc. of the International Symposium on Computer Architecture.*

YAN, J. AND ZHANG, W. 2007. Wcet analysis of instruction caches with prefetching. *Proc. of ACM SIGPLAN/SIGBED 2007 Conference on Languages, Compilers, and Tools for Embedded Systems.*



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.



Jun Yan is currently a Ph.D student at Southern Illinois University Carbondale (SIUC). Before he came to SIUC, he worked in R&D at Lucent Technologies from 2004 to 2005 and at Huawei Technologies from 2002 to 2004. He received his MS from Tianjin University, China, in 2002, and BS from Shenyang Architecture and Civil Engineering Institute, China, in 1998, respectively.

Wei Zhang received the B.S. degree in computer science from the Peking University in China in 1997, the M.S from the Institute of Software, Chinese Academy of Sciences in 2000, and the Ph.D. degree in computer science and engineering from the Pennsylvania State University in 2003. He joined the Electrical and Computer Engineering Department at Southern Illinois University Carbondale as an assistant professor in August 2003. He has become an associate professor since July 1, 2007. His current research interests are in embedded computing systems, low-power design, computer architecture and compiler. His research has been supported by NSF, Altera and SIUC. He is a member of the IEEE and ACM. He has served as a member of the technical program committees for several IEEE/ACM conferences and workshops.