A Hybrid Hardware/Software Generated Prefetching Thread Mechanism on Chip Multiprocessors

Hou Rui, Longbing Zhang, and Weiwu Hu

Key Laboratory of Computer System and Architecture,
Institute of Computing Technology, Chinese Academy of Sciences.
100080 Beijing, China
{hourui, lbzhang, hww}@ict.ac.cn

Abstract. This paper proposes a hybrid hardware/software generated prefetching thread mechanism on Chip Multiprocessors(CMP). Two kinds of prefetching threads appear in our hybrid mechanism. Most threads belong to Dynamic Prefetching Thread, which are automatically generated, triggered, spawn and managed by hardware; The others are of Static Prefetching Thread, targeting at the *critical delinquent loads* which can not be accurately or timely predicted by Dynamic Prefetching Thread. Static Prefetching Threads are statically generated by binary-level optimization tool with the guide of profiling information. Also, some aggressive thread construction policies are proposed. Furthermore, the necessary hardware infrastructure for CMP supporting this hybrid mechanism are described. For a set of memory limited benchmarks with complicated access patterns, an average speedup of 3.1% is achieved on dual-core CMP when constructing basic hardware-generated prefetching thread, and this gain grows to 31% when adopting our hybrid mechanism.

1 Introduction

Advances in integrated circuit technology afford great opportunities for Chip Multiprocessors(CMP). It is really a challenge to utilize multi-cores in CMP to accelerate sequential programs. Thread-based prefetching technique is a promising approach to achieve this purpose. It typically uses additional execution pipelines or idle thread contexts in a multithreaded processor(CMP or SMT) to execute helper threads that perform dynamic prefetching for the main thread. Pure hardware-generated prefetching thread mechanisms[1,3,5,7,8,12,16] are transparent to compiler. However, such mechanisms might be inaccurate or suffer from higher memory bandwidth because it is difficult for hardware to observe and analyze the large range runtime execution. Traditional software-generated prefetching thread techniques[2,4,10,11] are typically accurate due to the better understandability on program semantics and data structures, but might incur additional instruction overhead and can not observe runtime behaviors.

It is necessary to adopt the advantages of both traditional hardware and software methods. To the best of our knowledge, this paper firstly proposes a novel hybrid hardware/software generated prefetching thread mechanism on Chip Multiprocessors.

The main contributions of this work are: (1) A hybrid hardware/software generated prefetching thread mechanism on Chip Multiprocessors is proposed; (2) Two aggressive thread construction policies, known as “Self-Loop” and “Fork-on-Recursive-Call”, are

W.E. Nagel et al. (Eds.): Euro-Par 2006, LNCS 4128, pp. 506–516, 2006.

presented for Dynamic Prefetching Thread; (3) "Thread Merging" policy is proposed for Static Prefetching Thread, which also adopts "Multi-Chain" policy; (4) The necessary hardware infrastructure for CMP supporting this hybrid mechanism is designed.

The rest of this paper is organized as follows: Section 2 introduces Dynamic Prefetching Thread. Section 3 describes the challenges to Dynamic Prefetching Thread. A hybrid hardware/software generated prefetching thread mechanism is proposed in Section 4. And Section 5 is performance evaluation. Section 6 is conclusion.

2 Dynamic Prefetching Thread

Many researchers found that a small number of static loads, known as *delinquent loads*, are responsible for the vast majority of memory stall cycles. Furthermore, not all the instructions contribute to the address computation of the future delinquent load[2,3,7]. Motivated by these observations, we try to extract these sequence of instructions as prefetching thread from the executed instruction trace by means of hardware, and utilize idle cores to execute such threads that perform dynamic prefetching for the main thread. Such threads are called *Dynamic Prefetching Thread(DPT)*, which are automatically generated, triggered, spawned and managed by hardware. It should exit when meeting exceptions or interrupts. The operating system should make no response to these exceptions and interrupts except for TLB exception.

2.1 The Hardware Infrastructure Supporting Dynamic Prefetching Thread

Figure 1(a) illustrates the typical CMP architecture with DPT support. The black blocks are the necessary hardware infrastructure supporting DPT. The "DPT Generator" is in charge of extracting DPT, located off the pipeline critical path. It has no effects on the pipeline frequency due to its back-end work mode. The "shadow register" is used for quickly initializing the context of the new spawned thread.

The organization of DPT Generator is shown in Figure 1(b). The committed load instructions in original thread and their corresponding execution information(such as L2 hit/miss flag) are sent to the back-end DPT Generator. These load instructions will first probe the trigger pointer selector, "Spawn Table". Once a trigger pointer is identified,

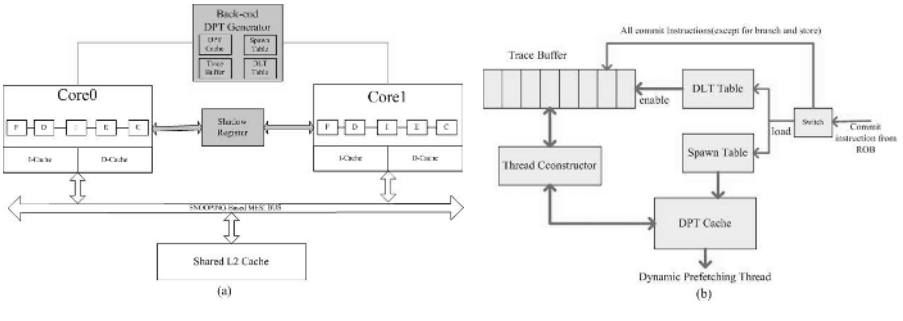

Fig. 1. The architecture of CMP with Dynamic Prefetching Thread support

the corresponding prefetching thread stored in DPT Cache is dispatched on idle core and run in parallel with original thread to perform dynamic prefetching for the targeted delinquent loads; otherwise it will query and update the Delinquent Load Table(DLT Table), which is in charge of identifying the delinquent load.

When any delinquent load is identified, DPT Generator begins to collect the committed instructions from the main core running original program. This collection does not stop untill the same delinquent load comes again or the Trace Buffer is full(If Trace Buffer is full, this mechanism is abort). After this collection, Thread Constructor performs a reverse walk of the trace to extract relevant instructions which contribute to the address computation of the targeted delinquent load. Then it produces a sequence containing these instructions in program order, oldest (lead) to youngest (candidate load). For simplicity, we only focus on the register dependence but ignore both memory and control-flow dependence during this reverse analysis. This policy is similar to Slice Processor[7], and we adopt it as our *basic policy*. Meanwhile, the trigger point is chosen for each Dynamic Prefetching Thread. These maps are recorded in DPT Cache.

The current CMP memory hierarchy is utilized to store prefetching results. No modifications are needed for memory hierarchy in this work.

Identify the delinquent load

The delinquent loads are identified at runtime via DLT Table. It is a PC-indexed table with 128 entries and each has 5-bit counters. One out-chip cache load miss(L2 Miss in our simulation) increases the corresponding counter by 4, otherwise decreases it by 1. A delinquent load is selected once the counter value exceeds 31. Predictor entry is allocated only when an L2 load miss occurs.

"Shadow Register" mechanism

The main core running original thread is to initialize the registers of the idle core when a DPT is dispatched. "Shadow Register" is for such quick initialization mechanism. It keeps the same data content with the main core. Some modifications are needed in pipeline to support this mechanism. The value and logical index of the destination register are attached with each issued instruction and reserved in ROB entries. Thus this information can be sent to the "Shadow Register" at commit time. The main core has the write privilege whereas the other cores running prefetching threads are only be allowed to read it. During the thread extraction phase, the live-in registers should be analyzed and used for marking some flags in renaming table of new core so as to differentiate the "Shadow Register" and local registers. Only the first access about the live-in registers on prefetching cores should access the "Shadow Register".

Trigger point and Spawn time

The delinquent load itself is selected as the trigger point. And the commit time is selected as spawn time because it is suitable for the loosely-coupled feature of CMP. Although choosing decode time as spawn time can spawn the thread earlier, it has problems in transporting register context among multi-cores. The reason is that the value of the instruction's destination register is still unavailable at decode time. Therefore the commit time is selected as the spawn time. It just needs to copy corresponding registers to initialize the new thread context at spawn time.

2.2 Aggressive Thread Construction Policies

(1) "Self-Loop" Policy

In basic policy, one Dynamic Prefetching Thread only prefetches one future instance of the static delinquent load. In "Self-Loop" policy, *the future N instances of the same delinquent load instruction are prefetched in the same Dynamic Prefetching Thread at one trigger point*(N=10 in our simulation). We accomplish this purpose via adding loop structure on basic-policy constructed thread code. The framework of new added loop structure is so stable that hardware implementation has high feasibility. This policy enlarges the prefetching range and helps the thread speculatively prefetch farther delinquent loads that are not visible in current pipeline. And it can also decrease the cost of thread initialization by merging multi-threads into one. Furthermore, "Self-Loop" policy need not copy register values between consecutive prefetching threads, since such threads are run on one core in our policy. This policy needs less prefetching cores(usually 1-4 cores are enough), thus releasing the access contention for "Shadow Register".

(2) "Fork-on-Recursive-Call" Policy

Most nodes in tree or graph structures connect two or more sub-nodes. This inherent memory parallelism can be exploited for prefetching. When the main program accesses one sub-tree or sub-graph, other idle cores can be utilized to speculatively access the other sub-tree or sub-graph. What's more, the recursive function is one of the primary methods used to access such structures. When any recursive call instruction is executed, a new prefetching thread is dispatched on one idle core starting from the next instruction address. Then the idle core begins to speculatively execute the following instructions. By means of this approach, idle cores are utilized to speculatively access the other sub-tree or sub-graph for prefetching. This is the "Fork-on-Recursive-Call" policy.

A hardware stack and Recursive Call Table are used for identifying the recursive call and recording the recursive entries for each recursive call. They work in back-end and are placed in DPT Generator. Any function call instruction(e.g, jal, jalr in MIPS ISA) at the top of ROB will trigger the following step:

(a) Looking up the Recursive Call Table to find whether this call is recursive. If some entry is found, then goto (b), else goto (c).

(b) The following PC of the current call instruction is sent to idle core to be speculatively executed. And exits here.

(c) The instruction's PC enters the hardware stack. It will look up the previous stack entries before entering the stack. If some entry matches, a recursive call is identified, and the PC is recorded in Recursive Call Table. Otherwise, it is just stored in the stack. The stack should be emptied if it is full.

Any return instruction(e.g, jr in MIPS ISA) should update the stack at commit time. If the stack is empty, nothing is done; otherwise the top stack entry is popped.

The store instructions are considered as nop operation since the speculatively executed thread is only used for prefetching and should not modify the architecture state. A counter is used to control the execution distance of prefetching thread. The prefetching thread also looks up the Recursive Call Table when any call instruction is executed. If one recursive call is identified, the counter begins to work and increase one for each

instruction. In this work, the prefetching thread will not stop untill the counter exceeds 200 or some exception occurs.

3 The Challenges to Dynamic Prefetching Thread

The following three cases are great challenges for Dynamic Prefetching Thread.

(1) The loops with two or more delinquent loads. When there are two or more delinquent loads in the same loop structure, usually some of them are not timely prefetched by Dynamic Prefetching Threads. The reason is that each such threads usually targets at only one static delinquent load. If the number of processor core is small, several Dynamic Prefetching Threads separately targeting at different loads compete for the scarce idle cores. Thus some of prefetching threads have no chance to be dispatched.

(2) The loops with two or more levels. Larger prefetching range can be expected at the outer-level loop. Yet it is hard for the hardware to identify and collect the whole execution trace of the outer loop iterations. Therefore the prefetching timeliness and range are limited.

(3) The hot regions with complicated control flow. The instruction traces are unstable in this case. It is hard for hardware to analyze and conclude all the conditions at runtime. The prefetching accuracy might be quite low.

4 The Hybrid Hardware/Software Prefetching Thread Mechanism

Although software-generated prefetching thread might incur additional instruction overhead and can not observe runtime behaviors, it can overcome the challenges to Dynamic Prefetching Thread. We proposes a hybrid hardware/software generated prefetching thread mechanism on Chip Multiprocessors. Two kinds of prefetching threads appear in our hybrid mechanism. Most threads belong to Dynamic Prefetching Thread, which are automatically generated, triggered, spawn and managed by hardware; The others are of *Static Prefetching Thread(SPT)*, targeting at the *critical delinquent loads* identified by profiling information. SPT is statically generated by binary-level optimization tool. The software tool can understand the program semantics better, thus higher prefetching accuracy and larger prefetching range are anticipated for SPT.Furthermore, benefiting from the concentration on critical delinquent loads, SPT incurs little additional instruction overhead.

This hybrid mechanism is effectively composed of DPT and SPT where DPT is predominant. These two kinds of threads are efficiently combined by the identification of critical delinquent loads. An enhanced compilation flow and the corresponding profiling mechanism are proposed to support the identification of critical delinquent loads and the SPT construction. By the way, SPT has higher execution priority than DPT. All such threads are transparent to operating system.

4.1 Compilation in Hybrid Mechanism

The enhanced compilation supporting the hybrid mechanism is illustrated as the following steps:

(1) The program is compiled by general source-code compiler(e.g, gcc);

(2) The binary is run directly on CMP ***without*** Dynamic Prefetching Thread support. The instruction addresses of the TOP N most frequent load misses are collected via performance counter. Regarding these instructions, we call the set, which is composed of *(instruction address, the number of cache misses)* pairs, as ***Miss_Set0***;

(3) The binary is run directly on CMP ***with*** Dynamic Prefetching Thread support. The instruction addresses of the TOP N most frequent load misses are collected via performance counter. Regarding these instructions, we call the set, which is composed of *(instruction address, the number of cache misses)* pairs, as ***Miss_Set1***;

(4) Then the set of *critical delinquent loads*, which can not be accurately or timely prefetched by Dynamic Prefetching Thread, are identified according to the following formula:

$$\begin{aligned} Critical\_Set = \{x \mid & \exists x,\ \exists y0,\ \exists y1, \\ & (x, y0) \in Miss\_Set0, \\ & (x, y1) \in Miss\_Set1, \\ & and\ (y0 - y1)/y0 < \delta\} \end{aligned}$$

In this formula, x is instruction address, y0 and y1 are the numbers of cache misses, and the δ is the assumed threshold for identifying critical instructions.

(5) Targeting at these critical delinquent loads, the binary-level SPT tool can extract more effective prefetching threads from original binary, attach them in a special program text segment , and regenerate the final version SPT-enhanced binary.

4.2 The Binary-Level SPT Tool

Firstly, the binary is loaded and disassembled. Guided by the relocation information in binary head section(e.g, ELF head), all basic blocks and their relationships(functions and branches) are identified. Then the control flow graph(CFG) is constructed. Secondly, the loop structures or functions containing *critical delinquent loads* are located, and the tool makes analysis on such zones based on several specific thread construction policies. All the instructions, which contribute to the address computation of the critical delinquent loads, are extracted. Such extracted instructions are Static Prefetching Thread, placed in a special program text segment at the bottom of original binary. During these analysis, the register live-ins of Static Prefetching Thread are also attained, which is helpful to choose a spawn point and insert a spawn instruction in original binary. Finally, some adjustments are necessary since original binary is modified, and then we get the SPT-enhanced binary by the SPT tool.

4.3 Thread Construction Policies for SPT

(1) "Thread Merging" Policy

"Thread Merging" policy is proposed to overcome the case where there are several delinquent loads in the same loop. In this policy, all the static delinquent loads in the same loop are prefetched by one prefetching thread.

According to profiling, SPT tool can observe that more than one critical delinquent loads appear in the same loop structure. Through analyzing the register and control dependence from the loop header to bottom(still ignoring memory dependence), all instructions contributing to the computation of these delinquent loads' addresses are extracted. The loop header is selected as the spawn point before which spawn instruction is inserted. Of course, "Self-Loop" can also be merged with "Thread Merging" policy.

(2) Multi-Chain Policy

Multi-Chain policy is described in [9]. We apply it to deal with the case where there are delinquent loads in loop structure with two or more levels. Such case is common in pointer-chasing applications, which tend to traverse composed data structures consisting of multiple independent pointer chains. Multi-Chain policy exploits this inter-chain memory parallelism. When the original thread accesses one pointer chain, Static Prefetching Threads simultaneously perform their speculative traversal of other possible future chains on idle cores. Consequently, the serialized memory latency can be tolerated by overlapping cache misses across independent pointer-chain traversals.

When SPT tool observes that there are little overlap work between the sequent critical delinquent loads in the loop with two or more levels, multi-chain policy is adopted to construct SPT. First, it analyze the inner loop, and extract all instructions contributing to the address computation of these delinquent load, including the loop induction variables and corresponding instructions. These extracted instructions are called as sub-thread. Meanwhile, the register live-ins of sub-thread are attained. Then the outer loop is analyzed, all instructions related are also extracted. These instructions are located before the sub-threads. Then the whole SPT is constructed. The spawn instruction is inserted directly before the entry of the inner loop.

4.4 Hardware Support for Hybrid Mechanism

(1) Extensions to the Instructions Set Architecture

Two additional instructions are needed. One is the *spawn* instruction. Its format is "spawn start-of-SPT". This instruction explicitly indicates one SPT dispatch and register context initialization. The other is the *stop* instruction, indicating that the prefetching thread is to be finished. It has no operator and is also used for DPT.

(2) Profiling mechanism for the TOP N out-chip load instructions

The critical delinquent loads identification needs to collect the top N most frequent out-chip loads for the execution of whole program. A hardware/software cooperative profiling mechanism is designed for such purposed.

A new Performance Counter(PC) is provided to record recent the top N most frequent out-chip load instruction, which is similar to Cache Miss Lookaside Buffer[13] aiming at releasing the access pressure on L2 cache. The new Performance Counter consists of*(process ID, instruction address, counter)* tuples with process ID and instruction address as index. It is implemented as content-indexed array(CAM). The Process ID is used for distinguishing the instructions from original or prefetching thread, and only the former is concerned. To improve the accuracy, the tuples are broken into two segments: HOT and LRU region. Once an out-chip load commits, a lookup in the PC(both the LRU and HOT segment) is performed. If it doesn't match, the least-used entry in LRU segment is replaced by the instruction and the counter is initialized as one; Otherwise,

the corresponding counter is increased. Furthermore, if it matches the LRU segment and the counter is larger than the minimum in HOT segment, these two entries are exchanged. The size of LRU and HOT segments are important for the profiling accuracy. We find 32 is suitable in our simulation.

However, the Performance Counter can only record recent out-chip load instruction. In order to record the top N most frequent out-chip loads for the execution of whole program, software are needed to record and accumulate the performance counter at intervals. Such function is implemented in the timer interrupt entry of operating system. Since PC only works for profiling, this mechanism does not decrease the performance and has no additional power dissipation.

Table 1. Simulated CMP Processor Parameters

Processor core		**Memory Hierarchy**	
Number of cores / Frequency	2core/2GHz	Cache sizes	32KB IL1, 32KB DL1, 512KB L2
Fetch / Issue / Commit Width	4 / 4 / 4	Cache associativity	4-way L1, 8-way L2
I-window / ROB / LSQ size	64 / 128 / 64	Cache Hit/Miss latencies	L1:2/3 cycles, L2: 9/11 cycles
Int/FP registers	184	Cache line sizes/ports	L1:32B,2ports, L2:32B,4ports
LdSt/Int/FP units	2 /4 / 2	L1-L2, L2 cache Store policy	write-back
Execution latencies	similar to MIPS R10000	MSHRs	L1:64 , L2:128
Branch predictor	16K-entry gshare hybrid	Memory Bus	split transaction, 2words/cycle
RAS entries	16	Main memory latency	minimum 200 cycles

Hardware Supporting Hybrid Prefetching Thread	
Trace Buffer Size	256 entries
DPT Cache size / associativity	32kB / 2 way
Thread construction time	200 cycles
Thread initiation time	6 cycles
Shadow Register size / port	64*32B / 4w4r ports
δ for Critical_Set	0.5

5 Experiments

5.1 Simulation Methodology

The evaluation is performed by a detailed CMP architecture simulator based on SESC [17] implementing MIPS ISA, which is a cycle-accurate execution-driven simulator. The CMP cores are out-of-order superscalar processors. Table 1 lists the parameters in details. To demonstrate the performance potential of our architecture, we just use the dual core configuration for simplicity.

The memory limited benchmarks are selected from the Olden pointer intensive programs[6], and SPEC CPU2000. A large number of cache misses in these benchmarks are due to relatively irregular access patterns involving pointers, hash tables, tree/graph, indirect or complicated array references, or a mix of them, which are typically difficult for prefetching. The train sets are used for SPEC benchmarks to achieve reasonable simulation times. In addition, all benchmarks are compiled with gcc -O3 and simulated for one billion committed instructions after fast-forwarding the initialization with cache warmup.

The full mechanisms of Dynamic Prefetching Thread are simulated in details. Yet Static Prefetching Threads are constructed manually. SPT tool now can read and modify

the MIPS binaries, while the implementation of SPT thread construction policies is still in development. These hand-generated Static Prefetching Threads demonstrate the performance potential of our hybrid mechanism.

5.2 Performance Evaluation

The performance speedup of our hybrid mechanism is illustrated in Figure 2. Since DPT is predominant in our hybrid mechanism, the speedup of DPT adopting basic and aggressive polices is also presented to make comparisons, and the speedup of pure SPT mechanism is ignored in Figure 2. With regards to the DPT mechanism, it can be observed that significant improvements are achieved with aggressive policies. The aggressive policies achieve 21.5% speedup on average while the basic policy only achieves 3.1% speedup. Furthermore, the performance can be further improved by the hybrid mechanism. SPT can overcome the challenges to DPT(especially for swim, mgrid, equake, em3d and mst). The performance speedup is increased to 31% on average when adopting the hybrid mechanism.

To understand the performance speedup, the prefetching coverage and timeliness information is provided to have a deep insight at the prefetching activity in Figure 3. Each bar is broken into eight segments according to the fractions of the miss latency hidden by prefetching, e.g, less than 10 cycles, between 10 and 50 cycles and so on.

For swim, mgrid,art, equake and mcf, most of the speedup benefits from the larger coverage and better timeliness achieved by DPT with "Self-Loop" policy. Through enlarging the prefetching range and number per prefetching thread, "Self-Loop" policy makes the thread generate more timely and more farther prefetching requests illustrated in Figure 3. For instance, the coverage of swim is about 2% in basic policy, and it increases to 21% in aggressive policies. And the performance speedup for swim also increases from 0 to 32% with the improvements of prefetching coverage and timeliness.

The "Fork-on-Recursive-Call" policy stimulates the performance improvements for treeadd, perimeter and tsp, since these benchmarks access tree-like structures via

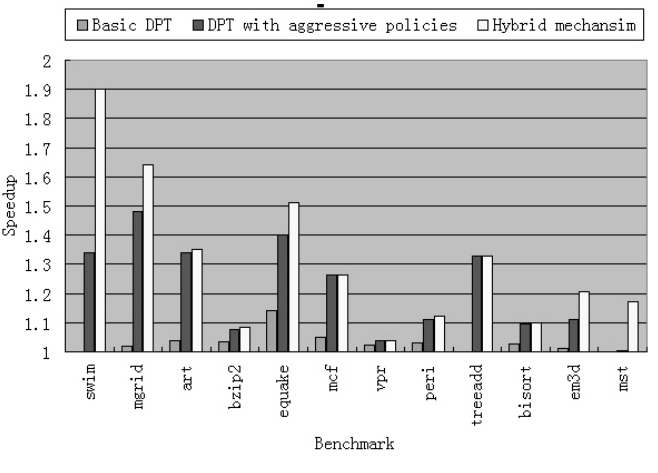

Fig. 2. The performance speedup of several prefetching thread mechanisms

recursive calls. This policy effectively exploits the memory parallelism indicated by the the kernel data structures and then improves the prefetching coverage and timeliness, especially for treeadd(31% performance improvement).

Although most benchmarks have been accelerated significantly by Dynamic Prefetching Thread, there are still considerable performance potential to be exploited by our hybrid mechanism. For swim, mgrid and equake, the "hot" loops always have several delinquent loads. Static Prefetching Threads constructed by "Thread Merging" policy prefetches these delinquent loads using one thread, leading to the significantly improved prefetching coverage demonstrated in Figure 3. For mcf and vpr, there are usually one delinquent loads in hot loop or thread contentions are scarce, Static Prefetching Threads almost have no effects. For mst and em3d, most pointer accesses have little overlap work, so Dynamic Prefetching Threads almost have no effects on them. Fortunately, it is observed that the kernel data structures are accessed by loops with two or more levels, "Multi-Chain" policy can effectively accelerate these benchmarks via higher-level prefetching(the performance improvement for em3d is 20%, mst is 18%). These phenomenons are demonstrated in Figure 3.

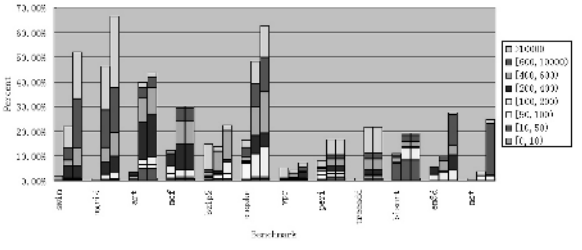

Fig. 3. The prefetching coverage and timeliness analysis. (For each group, the left bar: DPT with basic policy, the middle: DPT with aggressive policies, the right: hybrid mechanism.)

6 Conclusion

This paper firstly proposes a hybrid hardware/software generated prefetching thread mechanism on Chip Multiprocessors.This hybrid mechanism is effectively composed of Dynamic Prefetching Thread and Static Prefetching Thread. The former is predominant and dynamically generated by hardware, and the latter is complementary and statically generated by software. These two kinds of threads are efficiently combined by an enhanced compilation flow and the corresponding profiling mechanism.

For a set of memory limited benchmarks, an average speedup of 3.1% is achieved on dual-core CMP when constructing DPT with basic policy, and this gain grows to 21.5% when adopting aggressive policies. Although significant improvements can be achieved by DPT, the performance can still be further improved by the hybrid mechanism. SPT is an effective complement to DPT. The performance speedup is increased to 31% on average when adopting the hybrid mechanism.

Acknowledgements

We would appreciate the anonymous reviewers for their advices. This work is supported by the National Science Foundation for Distinguished Youth Scholar(60325205), the Basic Research Foundation of the ICT, CAS under Grant No.20056020; the 863 Hi-Tech Research and Development Program of China (2005AA1100102005AA119020); the National Grand Fundamental Research 973 Program of China, National Basic Research Program of China under No 2005CB321600.

References

1. A. Roth , G. Sohi. Speculative data-driven multithreading. In *7th HPCA*, pages 37-48, 2001.
2. J. Collins, H. Wang, etc. Speculative precomputation: Long-range prefetching of delinquent loads. In *the 28th ISCA*, pages 14-25, July 2001.
3. J. D. Collins, D. M. Tullsen, H. Wang, etc. Dynamic speculative precomputation. In *the 34th annual ACM/IEEE International Symposium on Microarchitecture*, pages 306-317, 2001.
4. S. Liao, P. Wang, etc. Post-Pass Binary Adaptation for Software-Based Speculative Precomputation. In *ACM Programming Language Design and Implementation*, June 2002.
5. Jeffery A. Brown, Hong Wang et al., Speculative Precomputation on Chip Multiprocessors. In *the 6th MTEAC*, November, 2002.
6. M. Carlisle. Olden: Parallelizing programs with dynamic data structures on distributed-memory machines. *PhD Thesis, Princeton University Department of Computer Science*, 1996.
7. A. Moshovos, D. Pnevmatikatos, and A. Baniasadi. Slice processors: An implementation of operation-based prediction. In *the 15th International Conference on Supercomputing*, pages 321-334, June 2001.
8. H. Zhou. Dual-core execution: building a highly scalable single-thread instruction window. In *the 14th PACT*, 2005.
9. N. Kohout, S. Choi and D. Yeung. Multi-chain prefetching: Exploiting memory parallelism in pointer-chasing codes. In *ISCA Workshop on Solving the Memory Wall Problem*, 2000.
10. T. Mowry and A. Gupta. Tolerating latency through software controlled prefetching in shared-memory multiprocessors. In *Journal of Parallel and Distributed Computing*, pages 87-106, June 1991.
11. C. Luk. Tolerating memory latency through softwarecontrolled pre-execution in simultaneous multithreading processors. In *the 28th ISCA*, pages 40-51, July 2001.
12. Ilya Ganusov and Martin Burtscher. Future Execution: A Hardware Prefetching Technique for Chip Multiprocessors. In *PACT 2005*, pages 350–360, 2005.
13. Brian N. Bershad, Dennis Lee et al. Avoiding Conflict Misses Dynamically in Large Direct-Mapped Caches. In *the 6th ASPLOS*. Pages: 158–170. 1994.
14. J. Huh, D. Burger, S. Keckler. Exploring the design space of future CMPs. In *the 10th PACT*, pages 199–210, September 2001.
15. Doug Burger, James R. Goodman. Billion-transistor architectures: there and back again. *Computer*, Page(s):22-28. Mar 2004.
16. O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead execution: an alternative to very large instruction windows for out-of-order processors. In *the 9th HPCA*, 2003.
17. Jose Renau, Basilio Fraguela, James Tuck et al., *http://sesc.sourceforge.net*. January, 2005.