Available online at www.sciencedirect.com
SCIENCE DIRECT®
J. Parallel Distrib. Comput. 66 (2006) 1076–1089
Journal of Parallel and Distributed Computing
www.elsevier.com/locate/jpdc

Speculative pre-execution assisted by compiler (SPEAR)☆

Won W. Ro[a], Jean-Luc Gaudiot[b,*]

[a]Department of Electrical and Computer Engineering, California State University, Northridge, CA, USA
[b]Department of Electrical Engineering and Computer Science, University of California, Irvine, CA, USA

Received 16 July 2004; received in revised form 20 December 2005; accepted 22 March 2006
Available online 22 May 2006

Abstract

Speculative pre-execution achieves efficient data prefetching by running additional prefetching threads on spare hardware contexts. Various implementations for speculative pre-execution have been proposed, including compiler-based static approaches and hardware-based dynamic approaches. A static approach defines the p-thread at compile time and executes it as a stand-alone running thread. Therefore, it cannot efficiently take dynamic events into account and requires a higher fetch bandwidth. Conversely, a hardware approach is, by essence, able to dynamically make use of run-time information. However, it requires more complex hardware and also lacks global information on data and control flow. This paper proposes Speculative Pre-Execution Assisted by compileR (SPEAR), a pre-execution model which is a hybrid of the two approaches. It relies on a post-compiler to extract the p-thread code from program binaries and uses custom-designed hardware to execute the p-thread.

Keywords: Data prefetching; Speculative pre-execution; Simultaneous Multi-Threading; Instruction-Level Parallelism

1. Introduction

Long memory access latencies notoriously constrain the performance of modern microprocessors. Consequently, cache misses cause significant and unexpected pipeline stalling and waste a considerable amount of CPU cycles. This problem will continue growing as memory latency increases. For this reason, more than ever, latency hiding is an important design requirement, and various forms of data prefetching have been developed to counter this problem. However, traditional prefetching methods strongly rely on the predictability of memory access patterns and often fail when they are faced with irregular patterns. Indeed, the access regularity is diminishing in many modern applications such as multimedia processing, databases, scientific applications, etc. [11].

Speculative pre-execution has been developed to respond to the perceived need for a new prefetching method [1,7,8,12,14–17,19,20,22]; instead of predicting future accesses, this promising technique speculatively executes future cache miss instructions. To this end, the instructions which frequently cause misses (often called *delinquent loads* or *d-loads*) and the backward slices (the set of instructions upon which the miss-causing instructions have a data dependency) are decoupled from the original code and executed separately in an additional prefetching thread [6]. The prefetching thread (often referred to as *p-thread*) must be sufficiently lightweight in order to run faster than the main program flow. Therefore, as long as the p-thread is executed early enough, timely prefetching can be achieved. Parallel execution of the p-thread and the main program is feasible in single-chip multithreaded architectures such as *Simultaneous Multi-Threading* [9] or *Chip Multi-Processor* [13].

As for the implementation of speculative pre-execution, prior work; can be categorized into two distinct groups: *compiler-based static approaches* and *hardware-based dynamic approaches*. The first group strongly depends on a static analysis by the compiler to extract the p-threads, either from the high-level language [12,15] or at the binary level [14]. In this

☆ This paper is an extended version of the paper "SPEAR: A Hybrid Model for Speculative Pre-Execution" which appeared in the Proceedings of the 18th International Parallel and Distributed Processing Symposium, April 2004.

* Corresponding author. Henry Samueli School of Engineering, University of California, ET549, Irvine, CA 92697-2625, USA. Fax: +1 949 824 3779.
E-mail addresses: wro@csun.edu (W.W. Ro), gaudiot@uci.edu (J.-L. Gaudiot).

0743-7315/$ - see front matter © 2006 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2006.03.010

approach, the global program structure can be used to efficiently construct the p-threads. However, it cannot cope with dynamic events. Moreover, since the p-thread is fetched as a stand-alone running thread, forking a new thread requires some software intervention. In addition, a higher fetch bandwidth is needed to fetch multiple threads.

At the other end of the spectrum, in a dynamic approach, the p-threads are constructed at run-time with the aid of additional hardware [1,7,16]. The execution (often called *triggering*) of the p-thread is also dynamically launched from the instruction queue or the reorder buffer. This approach is faster and can efficiently handle dynamic events. However, it inevitably results in additional hardware complexity.

As a compromise between the two approaches, we propose here Speculative Pre-Execution Assisted by compileR (SPEAR), a new prefetching method. In our approach, the identification of the p-threads is handled by software in a static way, while triggering the p-thread is dynamically controlled by the hardware. The main contributions of our paper therefore lie in the following:

- Design of an effective interaction between the compiler-based p-thread construction and the hardware-based p-thread triggering: for this purpose, we have developed an automated tool for p-thread identification and proposed a feasible hardware design.
- Design of a front-end which is capable of efficiently delivering the p-thread: our architecture model does not require additional instruction fetches for the p-thread; the p-thread instructions are dynamically spawned from the instruction fetch queue (IFQ).

Before we embark on the description of our new scheme, we will state the problem in better detail in Section 2. Our scheme is then presented in detail (hardware design in Section 3 and compiler support in Section 4). Section 5 presents the experimental results and analysis.

2. Background and design motivation

In this section, the general execution mechanism of speculative pre-execution is illustrated as an example. In addition, the various implementations for pre-execution are explained in the second part. Finally, problem definition and design motivations are included.

2.1. General working mechanism of pre-execution

Fig. 1 shows a general example of speculative pre-execution with the innermost loop of Lawrence Livermore Loop 4 (lll4). This example is given only for a better understanding of the speculative pre-execution as background knowledge. Note that it is not meant to describe our SPEAR model.

The high-level source code written in C is shown in Fig. 1(a). Fig. 1(b) shows a dynamic instruction stream of two consecutive iterations for the innermost loop. From the access profiling step, the load instruction, which is marked ①, has been identified as the cause of a large number of cache misses. Therefore, the instruction is marked as a delinquent load, which means that it becomes a choice candidate for prefetching. The corresponding operation of the delinquent load (①) consists in loading a value matching the term $y[j]$ in the high-level language code (Fig. 1(a)). For each delinquent load, the backward slice should be chased, based on the data dependencies. The backward slice consists of all the previous instructions upon which the delinquent load has a data dependency. In the example of Fig. 1(b), the shaded instructions correspond to the backward slice of the delinquent load. Indeed, the backward slice computes the access address for load $y[j]$. At this point, the p-thread can finally be constructed; it consists of the delinquent load and its backward slice (Fig.1(c)).

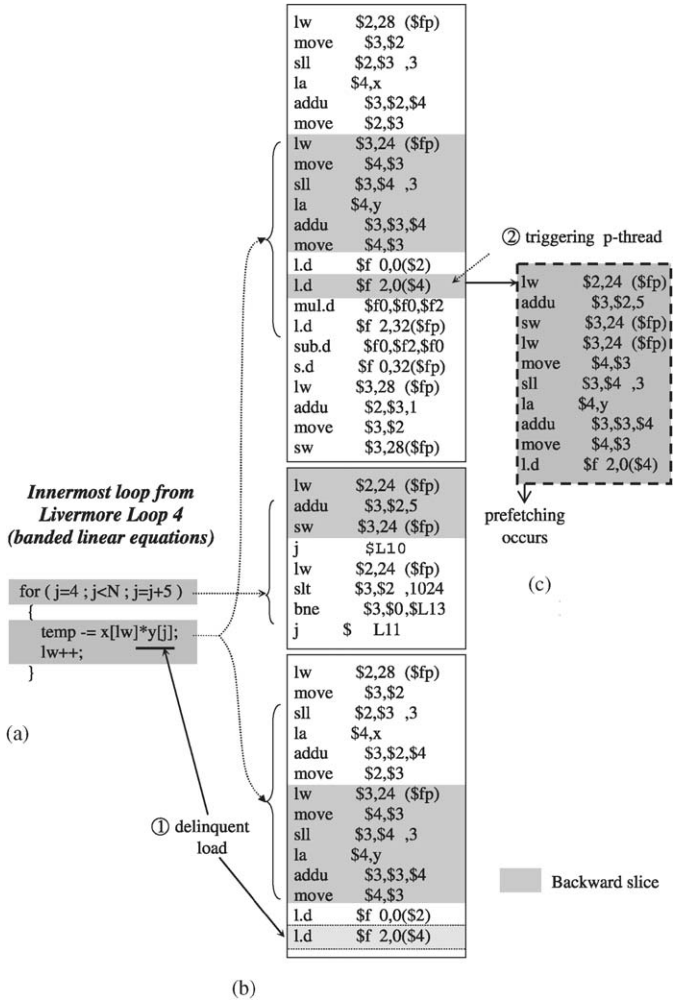

Fig. 1. Speculative pre-execution example: (a) high-level language code, (b) dynamic instruction stream and (c) p-thread execution.

The p-thread runs on an additional hardware context and only updates the data cache without changing the semantic state of the main program. The p-thread is lightweight and thus able to run faster than the main program flow. Indeed, the size of the p-thread and the starting point of the p-thread (*triggering point*) are very important for the effectiveness of prefetching. However, the dynamic behavior of a superscalar architecture is very

hard to predict. For that reason, all previous related research projects have heuristically defined the triggering point. A more quantitative analysis of the trigger point might improve the performance of the speculative prefetching [18]. In our example in Fig. 1, the instruction labelled ② is assumed to be a triggering point (note that this example is provided as an illustration of the working mechanism). Finally, the triggered p-thread runs on another hardware context so that timely prefetching can be achieved.

2.2. Various implementations of pre-execution

In the previous section, the execution of the p-thread has been shown from the point of view of the assembly code. However, the actual implementation of pre-execution could be achieved at any level of the compiling procedure or even at run-time by hardware. Indeed, various forms of the pre-execution model have been proposed in prior work; Fig. 2 describes three categories of implementation of the pre-execution model. Fig. 2(a) shows the pre-execution model which works directly from the high-level language. The source-to-source compiler identifies the p-thread and produces another high-level code (either attached to or annotated in the original source code [15] or as a stand-alone piece of code [12]). In this approach, a delinquent load is identified by the specific line number in which it appears and the variable which is the subject of the load. Therefore, the backward slicing is achieved by following the data dependencies between the variables. Indeed, the analysis on the high-level code is quite beneficial since the program information can be used. However, this method usually results in a large amount of p-thread code compared to the other two approaches. Also, various operational latencies at execution time cannot be determined in the high-level source code.

Fig. 2(b) shows the assembly or binary level implementation [8,14,19,22]; the delinquent loads are defined at the instruction level, and the backward slice is chased according to the register dependencies. In this approach, the execution time of the p-threads can be predicted by examining the assembly code. Therefore, the triggering point of a p-thread can be decided according to the predicted execution time of the p-thread. In addition, global program structures can be used to define triggering points, the range of the p-threads, and the parallelization of the p-threads. In this approach, the p-thread generated is also assembly or binary code and works as a separate procedure or stream of instructions. Liao et al. [14] verified the feasibility of this approach by designing an automated binary tool.

The diagram in Fig. 2(c) shows the dynamic approach which inherently depends on additional logic to identify and extract the p-threads at run-time [1,7,16]. The hardware logic derives the dependency graph from the instructions, based on the register dependencies. This can be achieved either at the level of the instruction fetch queue [1] or after the commit stage [7,16]. The triggering of the p-thread is also supported by special hardware. Obviously, this hardware-based p-thread approach imposes additional hardware overhead.

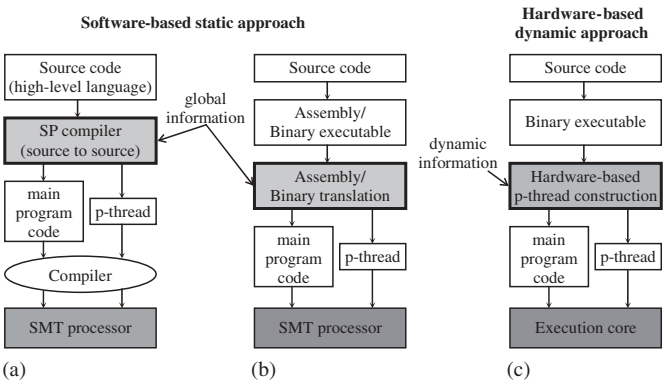

Fig. 2. Various implementations for pre-execution.

2.3. Problem definition and design motivation

Let us now turn our attention to the potential problems which can be caused by using a strictly static or a strictly dynamic approach. Indeed, it is these problems which have motivated us to develop a hybrid model for pre-execution.

The limits of a static approach: The two approaches illustrated in Fig. 2(a) and (b) construct the p-thread based on a static analysis. Of course, such a software approach cannot use run-time information which in turn causes several significant problems. First of all, a purely static method for program slicing yields a large amount of p-thread code [14]. In fact, this is a serious weakness since the p-thread should be lightweight so as to run faster than the main program. Also, the p-thread constructed in a static way cannot distinguish between the multiple dynamic instances of a static instruction. This may cause performance degradation because not every instance of a p-thread instruction produces cache misses. Third, triggering the execution of a p-thread requires additional overhead. The triggering operation is achieved with the help of multithreaded hardware and spawning procedures. This means that some software intervention is required to find a free context in SMT architecture, to assign the context to the p-thread, and to copy the live-in values from the main thread to the p-thread. Finally, it requires additional fetch bandwidth to fetch the p-thread instructions.

The limits of a dynamic approach: Essentially, a dynamic approach performs three functions with hardware structures: (1) identification and marking of p-thread instructions within the instruction stream of the application, (2) extraction of the marked p-thread instructions, and (3) triggering of pre-execution using the p-thread instructions. Indeed, the first disadvantage of the hardware approach is the complexity caused by the additional hardware. Also, since the prefetching thread is constructed by using an instruction window of a fixed size, the prefetching range is limited. Finally, the control flow of the p-thread is decided by the main program flow. Therefore, when the d-load is located in a basic block which is not correctly predicted, the d-load cannot be executed earlier and prefetching cannot be achieved.

Design considerations: Compared to existing dynamic approaches, our SPEAR model is a technique which retains the

instruction queue architecture for the purpose of extracting p-thread instructions as well as triggering the execution of the instructions. However, it performs the p-thread instruction identification by software, at compile time. Our design is motivated by the desire to take advantage of the benefits of each approach by developing

- A hybrid model of speculative pre-execution: the p-thread is identified at compile time (static p-thread construction), but it is marked, extracted, and executed in hardware.
- Static p-thread identification with dynamic information: although the p-thread is constructed by static analysis, the run-time information from profiling is used.

Our design is better in that it is less complex than the previous hardware approaches since we eliminate the p-thread construction; the task is now performed at compile time. On the other hand, the SPEAR architecture can utilize the run-time information which cannot be used in the previous static approaches.

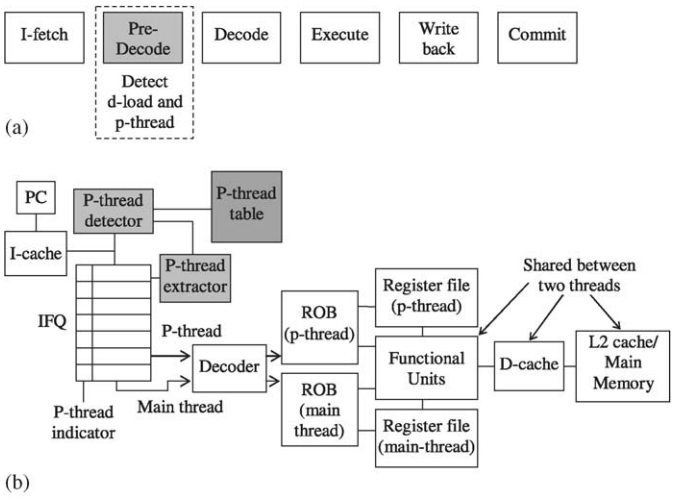

Fig. 3. Hardware description of the SPEAR: (a) pipeline stages and (b) hardware implementation for SPEAR.

3. Architectural support for SPEAR

SPEAR is a hybrid model for pre-execution which means that both hardware design and compiler support are crucial to achieve efficient p-thread execution. Detailed hardware description of the SPEAR is presented in this section and the compiler support will be explained immediately after.

A distinctive feature of our architecture is that the code which corresponds to the pre-execution is a strict subset of the main program code and is not stored in duplicate memory locations. Instead, those instructions that also belong to the p-threads are simply marked with appropriate "p-thread indicators" during the predecoding stage.

When a p-thread is triggered, the instructions marked as belonging to the p-thread are extracted from the IFQ. Note that, in this paper, "extraction" is merely the reading operation on the marked p-thread instructions. Since the p-thread is spawned from the IFQ, the control flow of the p-thread is decided by the branch prediction of the main program. Although this approach contains some disadvantages and limits as mentioned in the hardware approach section, it is still an acceptable strategy given the effective branch predictions of current processor architectures. Note that a similar observation was made in [1].

3.1. Hardware description

The structure of the SPEAR hardware is depicted in Fig. 3. The baseline architecture is an SMT model since this will enable the support of the simultaneous execution of the main thread and the p-thread. To facilitate the detection of the p-threads, their extraction, and their execution, several hardware structures need to be added on top of the basic SMT model. Since the p-thread instructions are extracted from the main thread instructions, they do not need to be fetched from the instruction cache. Also, it should be noted that no additional units (multiple PCs, multiple return address stacks, multiple branch prediction units, and fetch policy deciding logic) are needed since the main thread and the p-thread actually share the same code.

Fig. 3(a) shows the six pipeline stages of our SPEAR-enhanced architecture. Note how a second stage has been added in order to permit the detection of the p-thread instructions and the d-load instructions. The p-thread information which has been identified by the SPEAR compiler can be delivered to the architecture by either of two methods. The first one consists in adding new *op codes* to the instruction set and use them for p-threads and d-loads. The second method uses a special buffer called the p-thread table (PT) to store the p-thread instructions. We selected the latter approach since the former inherently requires a modification of the instruction set architecture.

A distinct feature of our PT is that it does not contain actual instructions. Instead, it only contains the PCs of the p-thread instructions. The portable instruction set architecture (PISA) instruction is 64 bit wide; storing the p-thread instruction itself instead of its address requires the same storage space. However, storing p-thread instructions instead of their address is not as flexible as our approach. It always executes the same code regardless of the dynamic behavior such as branch prediction.

Indeed, the PT is designed to find out whether the decoded instruction is a p-thread instruction or not. For this purpose, a special unit called the p-thread detector (PD) compares the PC of each instruction to the contents of the PT. A content addressable memory (CAM) can be used to implement the PT. In our simulation, the number of instructions in the p-threads is less than 1000 in most of the benchmarks. Therefore, we simulated the PT as a simple 8 KB storage unit. We modelled the access time of CAM to be approximately one cycle. We believe it is possible since the structure is much simpler and smaller than an instruction buffer (instruction window) or Level 1 cache of today's superscalar. When the time comes for actual implementation, additional study of the size needed for the PT should be undertaken. If the instruction in the predecoding

stage is identified as a p-thread instruction, the PD sets the p-thread indicator of the instruction to "*on*". Each slot of the IFQ includes a field for p-thread indicators.

The PT also provides information for the d-loads (recall that the d-loads are the target loads to pre-execute). If an instruction in the predecoding stage is detected as a d-load, the PD changes the machine state to the *pre-execution mode*. This will be explained in the next subsection. The PT also provides the register names for the live-in values which should be copied from the main thread context over to the p-thread context.

After the predecoding stage, the instructions are sent to the IFQ. Recall that the IFQ was originally designed to store instructions which have already been fetched, and to send them to the decoder in a FIFO order. In our SPEAR model, its further purpose is to mark the p-thread instructions with the appropriate p-thread indicators. As stated above, each entry of the IFQ has an associated p-thread indicator field which identifies p-thread instructions.

The other important functions of the IFQ are to extract the p-thread instructions and to send them to the decoder. The extracting and sending operations are controlled by the p-thread extractor (PE). The PE is activated when the execution of the p-thread has been triggered. This operation is described in the following subsection.

3.2. Triggering of the prefetching thread

The execution of a p-thread is initiated when a *d-load* has been detected by the PD at the predecoding stage. However, in order to guarantee a sufficient prefetching distance, the number of instructions inside the IFQ should be more than a predetermined number. In our experiments, we empirically used half of the IFQ size for this number; more research on this number, along with the size of the IFQ, should be conducted in the future. When the above conditions are satisfied, the predecoding logic enables *pre-execution mode*. This means that the PE becomes active and begins extracting instructions for pre-execution and delivering them to the decoder.

Although an instruction has been detected as a p-thread instruction during the predecoding stage, it is dormant in the IFQ until the pre-execution mode is entered. At this point, the triggering logic is activated and waits until all instructions which are already decoded have been committed. This guarantees that a deterministic state has been reached before the live-in values can be copied. After that, the live-in values are copied from the main thread (remember that the register names for the live-in values are provided by the PT). In our simulations, we assumed that each copy operation would take one clock cycle. Finally, the execution of the p-thread can be started.

Our IFQ is implemented as a circular FIFO and operates as a conventional FIFO buffer when it is in the normal state. During the pre-execution mode, the PE becomes active and scans each entry starting with the head of the IFQ in order to extract the p-thread instructions. Whenever the PE meets an instruction whose p-thread indicator is "*on*", it extracts the instruction and sends it to the decoder. Our scheme requires random accesses but without any decoding operation. It only checks the p-thread flags sequentially. It does not require any crossbar/gather hardware since it simply fetches instruction from the fetch queue to reorder buffer. It has a simpler and smaller structure than an instruction buffer.

In fact, it only copies the instruction to the input field of the decoder and leaves the instruction in the IFQ for the main thread to execute. This is because, although the instruction has been selected and delivered to the decoding logic as a p-thread instruction, it also needs to be executed as part of the main thread as well (recall that our p-thread is a strict subset of the main thread and that p-thread instructions should be executed as part of both threads). Therefore, each p-thread instruction needs to remain in the IFQ after being sent to the decoder as part of the p-thread. However, to prevent multiple executions of the p-thread instructions, the PE must set the p-thread indicator of the instruction to "*off*" immediately after reading it.

The number of p-thread instructions which can be extracted in a single cycle can be as high as half of the issue bandwidth. This strategy is designed so as not to overly penalize the main thread instructions. In our simulation, the issue bandwidth is assumed to be eight instructions; therefore, the PE can extract as many as four p-thread instructions in a single cycle. The remaining bandwidth is used to decode the instructions for the main thread. To point to the instruction from which the PE starts to scan at the next cycle, the IFQ needs one more pointer, the "p-thread head". It points to the next instruction of the p-thread instruction which was just extracted. At the next cycle, the PE starts to scan each entry of the IFQ starting from the p-thread head.

When the d-load which initiated the pre-execution mode has been extracted and sent to the decoder, the extraction operation of the PE terminates. For this purpose, the PE remembers the IFQ entry of the d-load which initiated the pre-execution mode. When the IFQ entry for an extracted instruction is matched to the d-load, the PE terminates the operations and becomes inactive. SPEAR triggers only one p-thread at a time.

3.3. Support for multithreaded execution

The p-thread is executed as a thread running concurrently with the main program thread. To support this, the processor should operate under a multithreaded configuration in the pre-execution mode. Every operation of an instruction is tagged with a dedicated *thread id*. In our model, 0 is assigned to the main program thread as a thread id, while 1 is assigned to the p-thread.

After the predecoding stage, the decoder performs the instruction decoding, detects the register dependencies, and renames the registers. After that, it assigns the instructions to the corresponding reorder buffer based on the *thread id*. The reorder buffer of our simulation is based on the register update unit (RUU) [4] which actually functions as scheduling logic for instruction execution. It also performs as the physical registers and the reorder buffer.

Basically, any ready instruction can be issued if the appropriate functional unit is available. However, since the p-thread needs to run faster than the main thread, the p-thread instructions are given scheduling priority. This means that the instructions from the p-thread are selected for execution first. If the number of ready instructions for the p-thread is less than the issue bandwidth, the remaining bandwidth can be given to the main thread.

During the execution stage, the functional units are shared between the threads. Actually, most other operations in the remaining pipeline stages are quite similar to existing SMT architectures [9]. Since the goal of the p-thread execution is to pre-execute the d-load, after the d-load is retired from the reorder buffer at the commit stage, the pre-execution mode is finished and the processor returns to the normal mode.

4. Description of the SPEAR compiler

An automated software tool (our SPEAR compiler) has been developed to produce the SPEAR binary. It has been designed to efficiently drive our proposed SPEAR hardware. SPEAR uses the global program graph along with the dynamic information to identify the p-thread. As we will see in this section, the SPEAR compiler is meant to directly work on the binary code.

4.1. Overview of the SPEAR compiler

Our SPEAR compiler has two distinct properties; it uses the dynamic information from the profiling and applies a region-based p-thread selection. Before we can bring a detailed description of each property, the overall compiling procedure is described in Fig. 4.

The input to the p-thread construction tool is the SimpleScalar binary named PISA [4]; the binary is produced by SimpleScalar targeting gcc-2.6.3. The output produced after all compilation steps is the SPEAR executable binary. Inside the tool, four individual modules are implemented. At the beginning, the binary is sent to the *control-flow graph* (*CFG*) *drawing tool* (①) and the *profiling tool* (②). The first one is developed to create the CFG and identify the loop region. The second one is designed to collect run-time information through profiling; we intentionally used different input data sets for profiling and benchmark simulation.

After the two modules, the program slicing module (③) collects the information obtained in the previous two modules and constructs p-threads using a hybrid slicing method [10]. Indeed, the slicing module performs the core operation, which constructs p-threads (details will be given in the next subsection). The last module is the attaching tool (④) which attaches the p-thread information to the SPEAR binary. The p-thread information will be loaded into the PT at program execution time. The four modules are developed from the *SimpleScalar-3.0* tool set [4].

4.2. Static p-thread construction with dynamic information

The program slicing tool (③) obtains the *program structure information* and *dynamic information* from the previous two modules (① and ②, respectively). Primarily, our slicing method is applied to the static program structure with the CFG which is drawn by the CFG drawing tool. In the slicing tool, each static instruction has its own data structure. In addition, the data structures for basic blocks are defined and pointed to by the corresponding instructions. The control flow is defined by identifying the target address of each conditional/unconditional jump instruction. The procedures are also defined by identifying jump instructions to the function calls. On the other hand, the profiling tool derives the graph of data flow among instructions. The dynamic instances of instructions are analyzed and dependencies are examined by the source/destination-register names. Furthermore, the access addresses of store and load instructions are analyzed to find the memory address dependencies. The other important task of the profiling tool is to identify delinquent loads. For this purpose, the profiling tool counts the number of cache misses for each static load instruction. When the number of cache misses is higher than some predetermined value, the p-thread construction phase is initiated. It is achieved by the backward chasing through the data dependency graph.

Control-flow detection for the p-thread: Our backward slice chasing is performed along the dynamic dependencies among instructions; the profiling tool delivers the data dependencies at the moment when the frequent cache misses happen. Therefore, the backward chasing only follows through the control flow which truly affects the cache miss instructions. For example, the static backward slicing for the delinquent load in

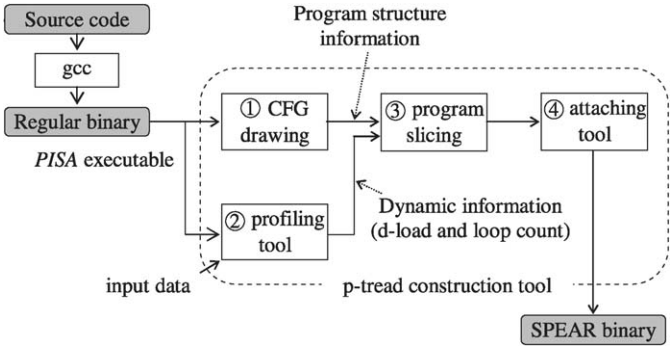

Fig. 4. The operations of the SPEAR compiler.

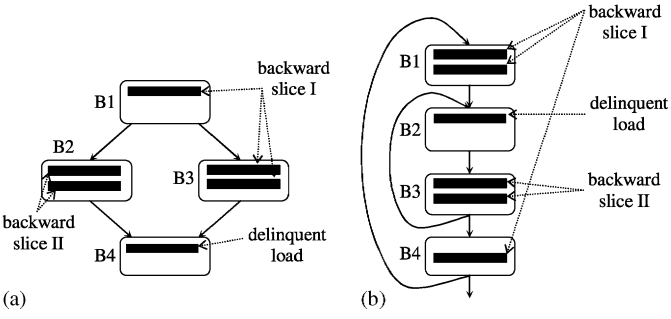

Fig. 5. Using run-time information for the p-thread identification.

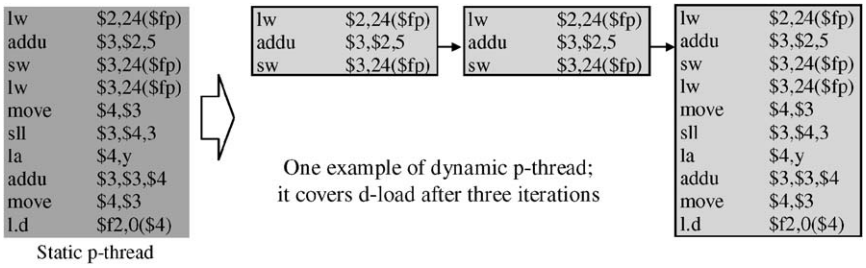

Fig. 6. Dynamic execution of the p-thread.

block $B4$ (in Fig. 5(a)) results in the backward slices I (in $B1$ and $B3$) and II (in $B2$). However, the result from the profiling tool indicates that the majority of cache misses happens when the program runs via $B3$. This suggests that the p-thread does not need to include backward slice II. The same observation is made in Fig. 5(b). The outer loop execution causes more cache misses at the load in $B2$ than during the execution of the inner loop. Therefore, the backward slice I (in $B1$ and $B4$) is a better candidate for inclusion in the p-thread than the backward slice II (in $B3$).

Region-based prefetching range: In addition to using dynamic information, our p-thread construction applies a region-based approach. The base region for a p-thread is an innermost loop where a delinquent load is located. Each loop has an expected delay (the *d-cycle*), which is obtained from the profiling tool; the average cycle time for one loop iteration is calculated. When an outer loop is added, the *d-cycle* of the loop is added to that of the delinquent load. The slicing tool defines the prefetching range based on the accumulated *d-cycles*. At this point, we use 120 (empirically chosen) as the criterion for the prefetching range. Also, regions across function calls are not considered for the prefetching range.

Dynamic p-thread scheduling: If a pure static method is used to determine the p-thread instructions, it would not be able to distinguish between the dynamic instances of the p-thread instructions. However, when the pre-execution targets a delinquent load after a certain number of loop iterations, every instance of the p-thread instructions is not required to be executed. For example, when the p-thread constructed in Fig. 1(c) intends to cover the cache miss (which would be caused by the delinquent load) after three iterations, only the instructions which contain loop-carried dependencies need to be executed for the first two iterations. An example of optimal dynamic execution is described in Fig. 6. This method is called *selective execution*; the previous loop iterations before the iteration for the d-load execution only execute the instructions which contain loop-carried dependencies. The selective execution of the p-thread instructions is not possible with the previous approaches; instead some previous approaches proposed chaining trigger [8], which actually spawns multiple loop-iterations simultaneously. In fact, chaining triggering consumes large amount of resources. At this point of the project, the selective execution has not been implemented in our tool. However, the hybrid characteristic of our SPEAR model renders selective execution feasible. A more detailed study for the selective execution will be performed as future work.

4.3. An example: the update stressmark

In this subsection, the *update* stressmark, which is one of the 15 benchmarks of our simulation, is shown as an example for the p-thread construction by our SPEAR compiler. Although our compilation procedures are actually performed directly on the binary code, the explanation will now be given using the high-level code for readability.

The high-level source code for the *while-loop* of the *update* stressmark is shown in Fig. 7. It is a pointer chasing benchmark in which the median value of a given size of window is followed. One iteration of the *while-loop* is defined as an one-hop operation, while the outer *for-loop* (with an increment of *ll*) searches for the median of the corresponding hop. Upon detecting that the current index has the median of the current hop, the control flow exits the *for-loop*. The median value determines the starting point of the next hop. Fig. 7(b) shows the basic block diagram with the necessary control-flow information. The program body designed to find the median value (named C in Fig. 7) is combined into a single extended block for readability. In fact, several basic blocks and complex control flows would exist inside of C.

Our profiling tool detects that the load instruction matching $field[index + ll]$ in B causes a large number of cache misses; since the index variable is calculated by the median of the random numbers, the starting load (which is $x = field[index + ll]$) of each hop is prone to cache miss. After identifying the delinquent load, the backward slice is chased by following the dynamic control flow at each cache miss. According to our profiling tool, a majority of cache misses are caused when the program comes from block D (which means that the outer *for-loop* is more prone to cache misses than the inner *for-loop*). Therefore, the statement for the variable *index* (which is $index = (partition + hops)\%(f - w);$) in D is detected as the first backward slice. Also, the statement for the variable *partition* in the block C (which is $partition = x;$) is included as the backward slice. Finally, the statement for the variable x in the basic block C is also included (which is $x = field[index + ll];$). After defining the backward slice, a p-thread is constructed together with the delinquent load and the backward slice (Fig. 7(c)).

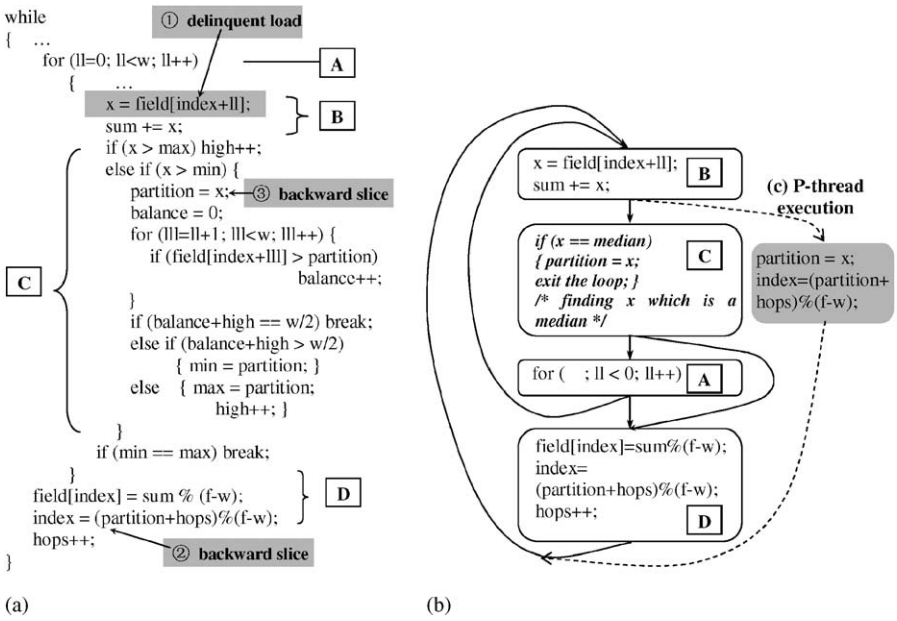

Fig. 7. P-thread construction with the update stressmark: (a) source code for the update stressmark and (b) corresponding control-flow graph.

5. Experimental results and analysis

To validate our architecture model, we have designed a complete cycle-by-cycle simulator and have performed intensive simulation experiments with 15 benchmarks.

5.1. Simulation environment

Benchmark selection: The target applications of our simulation are mainly memory-intensive applications. We chose six applications from the Atlantic Aerospace Stressmark suite [3], three applications from the Atlantic Aerospace Data-Intensive Systems Benchmarks suite [2], and six applications from the SPEC2000 suite. The benchmarks are compiled at the peak optimization level of the SimpleScalar target gcc compiler (version 2.6.3) and the benchmarks from SPEC2000 are simulated using the reference input set.

It should be noted that the complete set of data intensive systems (DIS) benchmarks includes five benchmarks which are more realistic and larger than stressmarks. Stressmark includes seven smaller benchmarks, which extract and show the kernel operations of data-intensive programs (Table 1). One of the stressmark and two of the DIS benchmarks could not be tested in our simulation; we encountered compilation problems with those three benchmarks which appeared to be incompatible with the SimpleScalar targeting gcc on our Linux machine.

In addition to those specific DIS applications, we also examined the performance of SPEAR with a subset of SPEC2000 (six memory-intensive applications picked out of the SPEC2000 suite). Only those were simulated as we felt that complete SPEC benchmarking would be outside the scope of the work since our SPEAR specifically targets memory-intensive applications.

Simulation parameters: The architectural simulator which models our SPEAR is based on the *sim-outorder.c* module [4]. The configurations we have tested are the baseline superscalar architecture with a 128 entry reorder buffer and the SPEAR models with two sizes of IFQ (128 and 256) (since the IFQ size is believed to affect the prefetching capability of the p-thread, we simulated two different IFQ sizes). We first suspected that the large IFQ might degrade the performance of the baseline superscalar. We tested the benchmarks with a smaller IFQ for the baseline superscalar and found that larger IFQ produces better performance even in the baseline superscalar architecture. Table 2 shows the detailed simulation parameters.

5.2. Benchmark results and analysis

Fig. 8 shows the performance results of our SPEAR architecture. Note that the left bars correspond to the performance of the baseline superscalar architecture. The middle bars and the right bars show the performance results of the SPEAR architecture with two different IFQ sizes (128 and 256, respectively). For demonstration purposes, the diagram shows the normalized performance based on the baseline superscalar architecture. The performance is measured in terms of IPC of the main program thread.

The SPEAR design improves the performance of 11 out of 15 applications. The best result reaches an 87.6% performance improvement, which is achieved with *mcf*. On the average, a 12.7% speedup is achieved with 128 IFQ and a 20.1% speedup is achieved with 256 IFQ.

However, four applications (*tr*, *field*, *fft*, and *gzip*) experience a slight performance degradation between 1% and 6.2%. *Tr* and *gzip* do not successfully work with our IFQ-based pre-execution because of the low branch hit ratio. Also, *gzip* contains too

Table 1
The simulated benchmarks summary

Suite	Name (abbreviation)	Skipped instructions	Simulated instructions
Stressmark	Pointer	Full running	85.9M
	Update	Full running	53.2M
	Neighborhood (nbh)	Full running	763.1M
	Transitive closure (tr)	Full running	929.7M
	Matrix	300M	500M
	Field	Full running	552.9M
DIS benchmarks	Data management (dm)	Full running	507.5M
	Ray tracing (ray)	300M	1B
	Fast Fourier transform (fft)	1B	500M
SPEC CINT2000	164.gzip	1B	500M
	181.mcf	1B	500M
	175.vpr	1B	500M
	256.bzip2	1B	500M
SPEC CFP2000	183.equake	1B	1B
	179.art	1B	500M

Table 2
Simulation parameters

Issue width	8
Commit width	8
Instruction fetch queue size	128 and 256
Reorder buffer size	128 instructions
Integer functional units	ALU (×4), MUL/DIV
Floating-point functional units	ALU (×4), MUL/DIV
Number of memory ports	2
Data L1 cache configuration	256 sets, 32 blocks, 4-way set associative, LRU
Data L1 cache latency	1 CPU clock cycle
Unified L2 cache configuration	1024 sets, 64 blocks, 4-way set associative, LRU
Unified L2 cache latency	12 CPU clock cycles
Memory access latency	120 CPU clock cycles

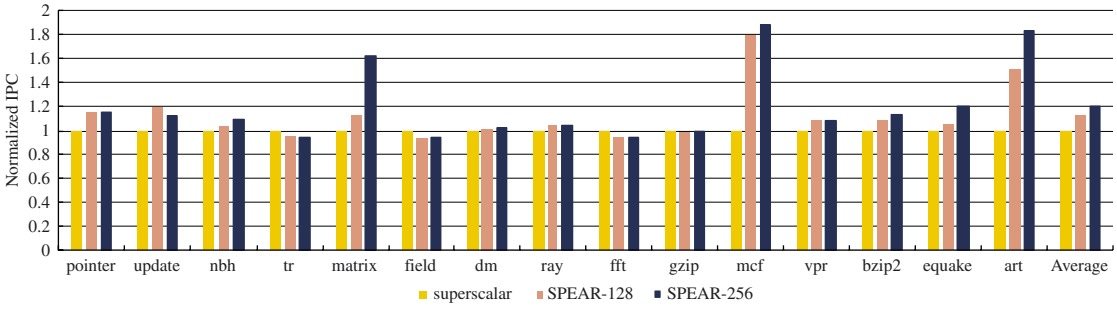

Fig. 8. Performance of the SPEAR models (normalized to the baseline superscalar).

many d-loads (49.2M) which causes an excessive amount of triggering operations. This may corrupt the efficient execution of the p-thread. As for *fft*, the p-threads contain a large number of instructions (1129) which may slow the execution of the p-thread. In the case of *field*, the cache miss rate is too low to benefit from prefetching.

The two applications from SPEC CFP2000 which target floating-point operations show the advantage of the SPEAR model; this is due to the fact that these applications contain long latency floating-point operations which mask the long memory latency operations. In fact, decoupled memory accesses are particularly beneficial with long latency floating-point operations.

Table 3
Performance enhancement with a longer IFQ

	SPEAR-256/SPEAR-128	Branch hit ratio	IPB
Pointer	1	0.9788	7.08
Update	0.94	0.8865	8.72
nbh	1.06	0.9958	15.21
tr	0.99	0.8865	22.55
Matrix	1.45	0.9942	11.75
Field	1	0.987	39.3
dm	1.01	0.8907	4.92
Ray	1	0.956	7.21
fft	1	0.9893	10.32
gzip	1	0.8986	6.08
mcf	1.05	0.9098	3.45
vpr	1	0.9005	5.92
bzip2	1.04	0.9425	6.24
equake	1.15	0.9018	6.18
art	1.21	0.9504	6.43

Effect of a longer IFQ: Table 3 has been prepared in order to show the effectiveness of a longer IFQ. The second column shows the performance enhancement ratio of SPEAR-256 over SPEAR-128. The remaining two columns correspond to the branch hit ratio and average instructions per branch (IPB). Two benchmarks (*update* and *tr*) experience a slight performance degradation with the longer IFQ (6% and 1%, respectively) due to the fact that the p-thread execution of those applications suffers from branch mispredictions. In fact, the hit-ratio of branch prediction in the two applications is comparatively low. In contrast, the hit ratio of *matrix* is quite high (99.4%). *Matrix* achieves the highest performance enhancement when using the longer IFQ. Indeed, the effectiveness of the long IFQ strongly depends on the branch prediction of the main thread [1].

Effect of dedicated resources: Since the p-thread is simply a subset slice of the main program, the execution behavior of both threads would be very similar. This means that two threads may seek simultaneous access to the same functional units, since the instruction stream between the two threads is similar. This may cause some resource conflicts. In order to measure the impact of resource constraint due to the simultaneous execution of the main thread and the p-thread, we designed and tested two more SPEAR models with separate functional units for the p-thread execution.

Fig. 9 depicts the performance results of the additional two architecture models along with the original two designs. *SPEAR.sf-128* and *SPEAR.sf-256* correspond to the dedicated resource models with separate functional units which are very similar to the *Chip Multi-Processor* architecture model [13]. It is very similar to the previous SPEAR architecture except each thread has dedicated resources. Therefore, each of the functional units we have is duplicated for p-thread running. The communications between the main thread and the p-thread only can be achieved through L1 data cache. Note that there is no additional communication method required. Therefore, the model does not require any tight coupling between the dedicated functional units of each thread.

Once again, all results are normalized to the baseline superscalar architecture. As the results indicate, having dedicated functional units improves the p-thread performance. More specifically, *tr* achieves a 33.2% improvement in performance with *SPEAR.sf-128* over *SPEAR-128* and a 39.3% improvement with *SPEAR.sf-256* over *SPEAR-256*. On the average, an 18.9% speedup is achieved with 128 IFQ and *sf* mode (*SPEAR-128.sf*) and a 26.3% speedup is achieved with 256 IFQ and *sf* mode (*SPEAR-256.sf*).

Indeed, the longer queue improves performance by a factor of 7.4% for both SPEAR and SPEAR.sf models, while the dedicated resource models improve performance by a factor of 6.2% with either 128- or 256-entry IFQs.

Cache miss reduction: The number of cache misses (on L1 data cache) is measured to show the effectiveness of the prefetching capability. Fig. 10 shows a reduction in the total number of cache misses for the *SPEAR-128* and *SPEAR-256*. As the results indicate, the number of cache misses is considerably reduced by the speculative pre-execution of the SPEAR. The best result is achieved with *art*, with a reduction of 38.8% in the number of cache misses. On the average, 19.7% of all cache misses are eliminated by the SPEAR-256 architecture. However, this reduction in the number of the cache misses does not directly influence the overall performance; the ratio of the number of load instructions over the total number of instructions also needs to be considered along with the reduction in the number of cache misses.

Long latency tolerance: To demonstrate how well the SPEAR model would tolerate long memory latencies, the benchmarks have also been simulated under varying memory latencies. The resulting performance with six benchmarks is depicted in Fig. 11. The longest latency configuration is identified as *memory access latency* $= 200$ and $L2$ *cache access latency* $=$ 20. The shortest case is designated as *memory access latency* $=$ 40 and $L2$ *cache access latency* $= 4$. Three more test cases were used in between.

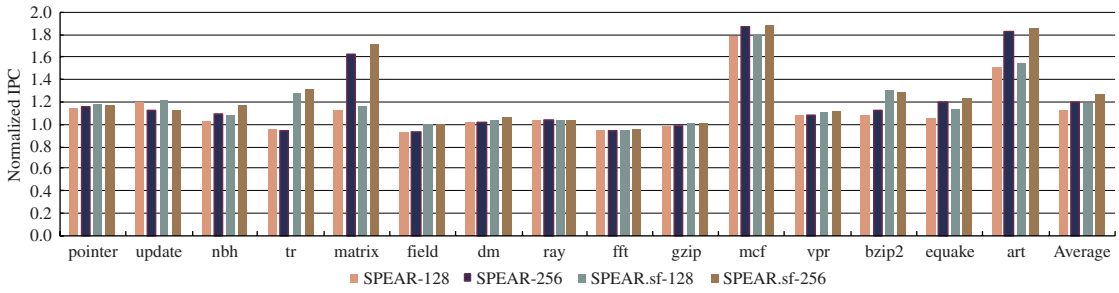

Fig. 9. Effect of dedicated resources (normalized to the baseline superscalar).

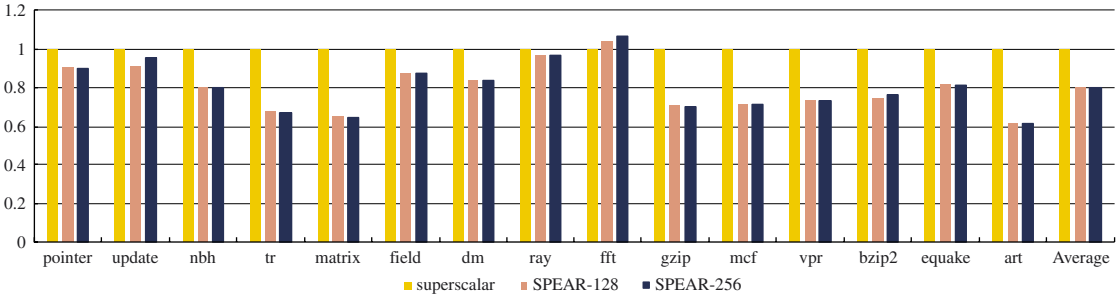

Fig. 10. Reduction of cache misses compared to the baseline superscalar.

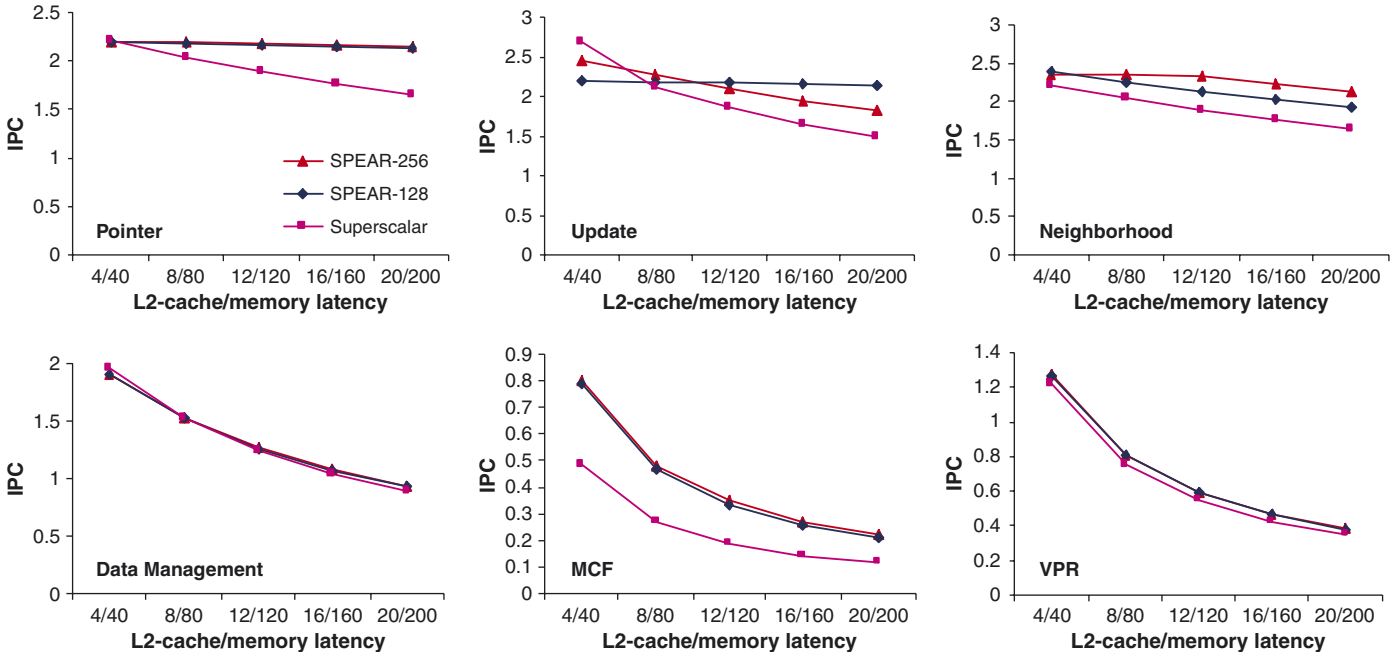

Fig. 11. Latency tolerance for various memory latencies.

The results demonstrate a robust performance for the long access latency configurations in three applications (*pointer*, *update*, and *neighborhood*). Indeed, the prefetching capability of the pre-execution reduces the impact of the cache misses and produces a stable performance for the long latencies. Although the remaining three benchmarks (*dm*, *mcf*, and *vpr*) experience some amount of performance degradation with longer latencies, they still perform better than an equivalent superscalar architecture. The performance of superscalar architecture drops severely with all six benchmarks.

The best performance is achieved with the pointer stressmark which experiences a 2% performance drop for the longest latency configuration compared to the shortest latency configuration (with SPEAR-256). At the same time, when simulating a superscalar architecture, the longest latency configuration exhibits a drop of 25.3% in performance. When averaging all the six benchmarks, the performance degradations of *SPEAR-128* and *SPEAR-256* at the longest latency are respectively 39.7% and 38.4% (when compared to the performance of the shortest latency configuration). The baseline superscalar architecture loses a whole 48.5% at the longest latency.

In the case of *pointer*, *update*, *neighborhood*, and *dm*, the degradation ratio obtained with longer latencies is relatively smaller than that for the equivalent superscalar architecture. On the contrary, *mcf* and *vpr* exhibit a decrease in performance, the shape of which parallels exactly what is observed in a superscalar architecture. This is due to the fact that the prefetching range (how far the p-thread runs compared to the main thread) of those two applications is not large enough to cover the long access latencies. Although the p-thread execution provides an efficient prefetching method, the prefetching range might be constrained by several factors such as application structures, program behavior, and branch prediction.

6. Related work

Roth and Sohi proposed their speculative data-driven multithreading (DDMT) which introduced a pre-execution model to mask performance-degrading instructions [19]. In their approach, the data-driven threads run on an additional context of an SMT processor and pre-execute the performance-degrading instructions. Indeed, the data-driven threads include the cache miss instructions and branch misprediction instructions (which are named *critical instructions* [19]) with their backward slices. The concept of performance-degrading slices was originally introduced by Zilles and Sohi [21]. They also proposed speculative slices to pre-execute those performance-degrading slices [22]. The execution model of speculative slices is close to our SPEAR. However, it does not include an automatic compiler.

Collins et al. [8] introduced speculative precomputation based on the SMT features of the Itanium processor. They also defined a small number of static loads as *delinquent loads* and include the backward slice as precomputation slices (p-slices) for data prefetching thread. It should be noted that their work introduced a new concept (the *chaining trigger mechanism*) which allows a speculative thread to trigger other speculative threads.

A hardware approach for the same concept was also proposed [7]. They designed and implemented additional hardware resources to construct and trigger p-slices at run-time. The delinquent load identification table (DLIT) is designed to identify delinquent loads at run-time. The retired instruction buffer (RIB) is a hardware structure to construct a p-thread after a delinquent load has been identified. The RIB operations are initiated whenever a delinquent load lacking a p-slice is committed. Once the RIB finishes constructing a p-slice, those instructions in the p-slice should be stored in the slice cache (SC). After that, the slice information table (SIT) is used to initiate the p-slice in the later execution. The effect of out-of-order execution with pre-execution was also analyzed [20].

Two more hardware-based dynamic approaches have been introduced by Annavaram et al. [1] and Moshovos et al. [16]. A dependence graph precomputation (DGP) scheme dynamically uncovers the prefetching slice for cache miss instructions [1]. When the *Predecode stage* detects the load/store instruction which is marked for prefetching (equivalent to the delinquent load), it automatically draws the dependence graph for the instructions inside IFQ. The instructions are chased backward based on register dependencies. In the DGP scheme, the speculative prefetching slice runs on a specially designed hardware called the *Precomputation Engine*. The other hardware approach is introduced by the *Slice-Processor* [16], which uses an additional hardware structure called *Slicer* to construct prefetching slice in the commit stage. The *Slicer* stores the scout thread (which is equivalent to the p-thread) in the slice cache, and the scout thread is initiated upon detecting *lead* instructions in the main program flow.

P-thread construction in a static way using a compiler was proposed earlier. Luk [15] introduced compiler algorithms to extract speculative pre-execution code; an analysis on a given high-level code finds and annotates the prefetching slice (*p-thread*). The actual execution of the p-thread is supported by the multithreading features of the SMT architecture. The triggering operation is handled totally in software. Another approach at the high-level language level can be found in Kim and Yeung's work [12] which is closely related to Luk's work, but it develops automated compiler algorithms. The last approach [14] is different from the previous two in the sense that the analysis is done at the binary level; they also proposed a region-based slicing method with global program information such as data-flow and control-flow analysis. Those analyses are not possible with hardware based p-thread construction. It is the closest to our SPEAR architecture model. However, the triggering operation is not hardware-oriented and imposes significant software overhead.

7. Conclusions

Traditional data prefetching methods strongly depend on future-event predictions and often fail when they are faced with irregular memory access patterns. Indeed, the need for new data prefetching methods has grown in modern processor design. In this paper, we have proposed our SPEAR model (Speculative Pre-Execution Assisted by compileR) which is a hybrid of two previous approaches (*compiler-based static* and *hardware-based dynamic*) for speculative pre-execution.

The SPEAR model strongly relies on compiler analysis to construct the p-threads. For this purpose, we developed a software tool which directly operates on the binary. In addition to that, the hardware model for the fast triggering of a p-thread has been defined and tested. The performance of the proposed model has been evaluated with 15 memory-intensive applications. Our software tool and the associated hardware model operate.

On the average, the experimental results show that our SMT-based SPEAR model achieves a 12.7% improvement with the shorter IFQ and a 20.1% with the longer IFQ. Also, the separate functional unit version results in an 18.9% and 26.3% speedup with the two IFQ sizes. We also showed the performance at the long memory latencies by varying the access latencies. On average, our SPEAR loses a 39.7% (with the shorter IFQ) and a 38.4% (with the longer IFQ) of the performance at the longest latency compared to the shortest latency configuration. The performance of the baseline superscalar architecture drops by as much as a 48.5% at the longest latency.

Further research on the prefetching range needs to be conducted. More algorithms on the region selection can improve the p-thread performance. Also, the actual effectiveness of the p-thread execution will be investigated. In addition, the overhead and complexity of the proposed hardware will be analyzed by considering VLSI implementation issues [5].

Acknowledgments

This paper is based upon work supported in part by NSF Grant No. CCF-0541403. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] M. Annavaram, J.M. Patel, E.S. Davidson, Data prefetching by dependence graph precomputation, in: Proceedings of the 28th Annual International Symposium on Computer Architecture, 2001.
[2] Atlantic Aerospace Electronics, Data-Intensive Systems Benchmarks Suite Analysis and Specification, 2000, URL: ⟨http://www.aaec.com/projectweb/dis/⟩.
[3] Atlantic Aerospace Electronics, DIS Stressmark Suite, 2000, URL: ⟨http://www.aaec.com/projectweb/dis/⟩.
[4] D. Burger, T. Austin, The simplescalar tool set, Technical Report CS-TR-97-1342, University of Wisconsin, Madison, 1996.
[5] J. Burns, J.-L. Gaudiot, SMT layout overhead and scalability, IEEE Trans. Parallel Distributed Systems 13 (2) (2002) 922–933.
[6] R. Chappell, J. Stark, S. Kim, S. Reinhardt, Y. Patt, Simultaneous subordinate microthreading (SSMT), in: Proceedings of the 26th Annual International Symposium on Computer Architecture, 1999.
[7] J.D. Collins, D.M. Tullsen, H. Wang, J.P. Shen, Dynamic speculative precomputation, in: Proceedings of the 34th Annual International Symposium on Microarchitecture, 2001.
[8] J.D. Collins, H. Wang, D.M. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, J.P. Shen, Speculative precomputation: long-range prefetching of delinquent loads, in: Proceedings of the 28th Annual International Symposium on Computer Architecture, 2001.
[9] S. Eggers, J. Emer, H. Levy, J. Lo, R. Stamm, D. Tullsen, Simultaneous multithreading: a platform for next-generation processors, IEEE Micro (1997) 12–18.
[10] R. Gupta, M.L. Soffa, Hybrid slicing: an approach for refining static slices using dynamic information, in: Proceedings of the Third Symposium on Foundations of Software Engineering, 1995.
[11] S. Hong, S. McKee, M. Salinas, R. Klenke, J. Aylor, W. Wulf, Access order and effective bandwidth for streams on a direct rambus memory, in: Proceedings of the Fifth International Symposium on High Performance Computer Architecture, 1999.
[12] D. Kim, D. Yeung, Access order and effective bandwidth for streams on a direct rambus memory, in: Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems, 2002.
[13] V. Krishnan, J. Torrellas, A chip-multiprocessor architecture with speculative multithreading, IEEE Trans. Comput. 48(9) (1999).
[14] S.S.W. Liao, P.H. Wang, H. Wang, G. Hoflehner, D. Lavery, J.P. Shen, Post-pass binary adaptation for software-based speculative precomputation, in: Proceedings of the Programming Language Design and Implementation, 2002.
[15] C.-K. Luk, Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processor, in: Proceedings of the 28th Annual International Symposium on Computer Architecture, 2001.
[16] A. Moshovos, D.N. Pnevmatikatos, A. Baniasadi, Slice-processors: an implementation of operation-based prediction, in: Proceedings of the International Conference on Supercomputing, 2001.
[17] W.W. Ro, J.-L. Gaudiot, Compiler support for dynamic speculative pre-execution, in: Proceedings of the Seventh Workshop on Interaction between Compilers and Computer Architectures, 2003.
[18] A. Roth, G.S. Sohi, A quantitative framework for automated pre-execution thread selection, in: Proceedings of the 35th Annual International Symposium on Microarchitecture, 2001.
[19] A. Roth, G.S. Sohi, Speculative data-driven multithreading, in: Proceedings of the Seventh International Symposium on High Performance Computer Architecture, 2001.
[20] P.H. Wang, H. Wang, J.D. Collins, E. Grochowski, R.-M. Kling, J.P. Shen, Memory latency-tolerance approaches for itanium processors: out-of-order execution vs. speculative precomputation, in: Proceedings of the Eighth International Symposium on High Performance Computer Architecture, 2002.
[21] C.B. Zilles, G.S. Sohi, Understanding the backward slices of performance degrading instructions, in: Proceedings of the 27th Annual International Symposium on Computer Architecture, 2000.
[22] C.B. Zilles, G.S. Sohi, Execution-based prediction using speculative slices, in: Proceedings of the 28th Annual International Symposium on Computer Architecture, 2001.

Won W. Ro received the B.S. degree in Electrical Engineering from Yonsei University, Seoul, Korea, in 1996. He received the M.S. and Ph.D. degrees in Electrical Engineering from the University of Southern California in 1999 and 2004, respectively. He also worked as a research scientist in Electrical Engineering and Computer Science Department in University of California, Irvine. Dr. Ro is currently an Assistant Professor in the Department of Electrical and Computer Engineering of the California State University, Northridge. His current research interest includes high-performance microprocessor design, compiler optimization, and embedded system design.

Jean-Luc Gaudiot (S'76-M'82-SM'91-Fellow '99) was born in Nancy, France, in 1954. He received the Diplôme d'Ingénieur from the École Supérieure d'Ingénieurs en Electrotechnique et Electronique, Paris, France, and the M.Sc. and Ph.D. degrees in Computer Science from the University of California, Los Angeles. Professor Gaudiot is currently a Professor and the chair of the Electrical Engineering and Computer Science Department at the University of California, Irvine. Prior to joining UCI in January 2002, he was a Professor of Electrical Engineering at the University of Southern California since 1982, where he served as Director of the Computer Engineering Division for three years. He has also done microprocessor systems design at Teledyne Controls, Santa Monica, California (1979–1980) and research in innovative architectures at the TRW Technology Research Center, El Segundo, California (1980–1982). He consults for a number of companies involved in the design of high-performance computer architectures. His research interests include multithreaded architectures, fault-tolerant multiprocessors, and implementation of reconfigurable architectures. He has published nearly 150 journal and conference papers. His research has been sponsored by NSF, DoE, and DARPA, as well as a number of industrial organizations. He was the Editor-in-Chief of the IEEE Transactions on Computers from 1999 to 2003. In June 2001, he was elected (and re-elected in 2003) chair of the IEEE Technical Committee on Computer Architecture. He is the founding editor (with Yale Patt and Kevin Skadron) of the Computer Architecture Letters, the refereed Newsletter of the IEEE Computer Society Technical Committee on Computer Architecture, and a Co-Guest editor of a Special Issue of the International Journal of Computer Applications in Technology, Applications for High Performance Systems, Fourth Quarter 2004. Dr. Gaudiot is a member of the ACM, the ACM SIGARCH and the IEEE. He has

also chaired the IFIP Working Group 10.3 (Concurrent Systems). He was Co-General Chairman of the 1992 International Symposium on Computer Architecture, Program Committee Chairman of the 1993 IFIP Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism, the 1993 IEEE Symposium on Parallel and Distributed Processing (Systems Track), the 1995 Parallel Architectures and Compilation Techniques Conference (PACT 95), the High Performance Computer Architecture Conference in 1999 (HPCA-5), the First Workshop on Embedded Parallel Architectures held in conjunction with the 10th International Symposium on High Performance Computer Architecture (2004), the Conference on Computing Frontiers (CF '04), the 2004 Simpósio Brasileiro de Arquitetura de Computadores e Processamento de Alto Desempenho (SBAC-PAD), and the 19th International Symposium on Parallel and Distributed Processing Systems (IPDPS 2005), and Topic Program Chair of the "Topic 8: Parallel Computer Architecture and Instruction-Level Parallelism", held at the Eighth International Euro-Par Conference (Euro-Par '02). From 1994 to 1998, he was a Distinguished Visitor of the IEEE Computer Society. He is now a Fellow of the IEEE, based upon his work on the programmability and reliability of multiprocessor systems.