Compiler Support for Dynamic Speculative Pre-Execution

Won W. Ro and Jean-Luc Gaudiot[†]

Department of Electrical Engineering
University of Southern California
wro@usc.edu

[†]Department of Electrical Engineering
and Computer Science
University of California, Irvine
gaudiot@uci.edu

Abstract

Speculative pre-execution is a promising prefetching technique which uses an auxiliary assisting thread in addition to the main program flow. A prefetching thread (p-thread), which contains the future probable cache miss instructions and backward slice, can run on the spare hardware context for data prefetching. Recently, various forms of speculative pre-execution have been developed, including hardware-based and software-based approaches. The hardware-based approach has the advantage to use runtime information dynamically. However, it requires a complex implementation and also lacks global information such as data and control flow. On the other hand, the software-oriented approach cannot cope with dynamic events and imposes additional software overhead. As a compromise, this paper introduces a hybrid model enhanced with novel compiler support for the dynamic pre-execution of a p-thread.

1. Introduction

Today's processor performance is strongly limited by the data access latencies upon cache misses. Truly, the speed-gap between processor and main memory continues to grow and increases the impact of miss penalties. Chances to execute many hundreds of instructions can easily disappear due to unexpected pipeline stalling. As a result, latency hiding becomes an important technique to avoid performance degradation upon cache misses. Basically, hiding the access latency has been pursued by using various forms of data prefetching. However, previous approaches have strongly depended on the predictability of future memory accesses and have often failed upon encountering irregular memory accesses such as pointer chasing. Those characteristics are common features of today's popular memory-bound or data-intensive applications.

Recently, other data prefetching approaches, which do not depend on predictability but rather on executing future access instructions, have been proposed [1][7][8][12][15][17][18][21][25]. Those approaches extract the future probable cache miss slice from the original code and execute it as an additional helping thread in a multithreading hardware. The cache miss thread (often called *p-thread* or *p-slice*) is lightweight and can run faster than the main program thread. Therefore, as long as the cache miss thread is executed early enough, timely prefetching can be achieved. This thread-based prefetching is often called *speculative pre-execution* or *speculative precomputation*. The parallel execution of the p-thread and the main program thread is made possible by the multithreading features of SMT (Simultaneous Multi-Threading) [10] or CMP (Chip Multi-Processor) [13] architectures.

Previous research on speculative pre-execution falls into two distinct categories. In the first group, all the procedures are handled by additional hardware implementations [1][8][18]. Since both p-thread construction and execution (often called *triggering*) are handled at runtime by hardwired circuit, the legacy binary code can still be used in this approach. However, this requires additional hardware circuitry. On the other hand, the second group strongly depends on a static analysis by the compiler to extract the p-thread either from the high level language [12][17] or at the binary level [15]. The static analysis can provide global program flow information. However, some software interaction to trigger p-thread is required and an additional overhead is unavoidable.

This paper proposes Compiler Assisted Speculative Pre-execution (CASP) as a compromise technique. Indeed, the CASP is a hybrid model of the two above approaches. Our design principle aims at bringing out the respective merits from the two camps. For that

This paper is based upon work supported in part by DARPA grant F30602-98-2-0180 and by NSF grants CSA-0073527 and INT-9815742. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA or NSF.

purpose, we divide the speculative pre-execution into two distinct operations: *p-thread construction* and *p-thread triggering*. In our approach, the first step, p-thread construction, is handled by the compiler in a static way and the second step, p-thread triggering, is controlled dynamically by the hardware. Therefore, the global program information can be considered for the effective construction of the p-thread, and the fast triggering of the p-thread is made possible by hardware-controlled spawning. In other words, compile time information is delivered to accelerate hardware triggering, lowering the hardware complexity. Indeed, Compiler Assisted Speculative Pre-execution (CASP) provides effective data prefetching along with a novel hardware and software design.

This paper is organized as follows: Section 2 reviews the background research. Section 3 describes the detailed hardware and software design characteristics of the proposed architecture. Section 4 includes the experimental results and thorough analysis. Finally, conclusions and future work are included towards the end.

2. Background Research

Our architecture model is strongly motivated by previous research on speculative pre-execution. In this section, the general concepts and the execution mechanism of the speculative pre-execution are explained in detail. In addition to that, a thorough survey for the recent related work is given in the second half of the section.

2.1. Overview of Speculative Pre-execution

In the speculative pre-execution model, timely data prefetching is achieved by the pre-execution of the p-thread. To better understand the concept, an example with Lawrence Livermore Loop 1 (lll1) is shown in Figure 1. Figure 1-(a) shows the source code written in a high-level language and Figure 1-(b) shows the dynamic instruction stream at iteration ***i*** and iteration ***i+1*** for the innermost loop. In the analysis, we assumed that frequent cache misses occur at the last instruction, which is labeled ①. Then, the instruction is defined as a *delinquent load*. Indeed, delinquent loads are candidate instructions for speculative prefetching. The corresponding operation of the delinquent load is loading the *y[k]* value in the high-level language code (Figure 1-(a)).

After uncovering the delinquent load, the backward slice is constructed based on the data dependencies. The backward slice includes all the previous instructions on which the delinquent load has a data dependency. Usually, the address calculation instructions are included in the backward slice. In the example code in Figure 1-(b), the instructions shaded in grey show the backward slice of the delinquent load. Indeed, the corresponding operation of the backward slice is the address calculation of *y[k]*. After figuring out the backward slice, the p-thread can be constructed together with the delinquent load and the backward slice. All this procedure for p-thread construction can be handled by hardware at runtime or software at compile time.

To execute the prepared p-thread, a trigger instruction needs to be defined inside the main program flow. When the main program detects the trigger instruction, the p-thread is spawned on any available hardware context. Once initiated, the p-thread can run faster than the main program thread since it is lightweight. In Figure 1, the instruction labeled ② indicates the trigger instruction of our example code. The proper choice for the trigger instruction is very important since the triggering point decides the size of the p-thread as well as the prefetching distance. Since the dynamic behavior of the current superscalar architecture is very hard to predict, all previous research strongly depends on heuristic methods to determine the trigger instruction. A more quantitative analysis on the trigger instruction might improve the performance of speculative prefetching.

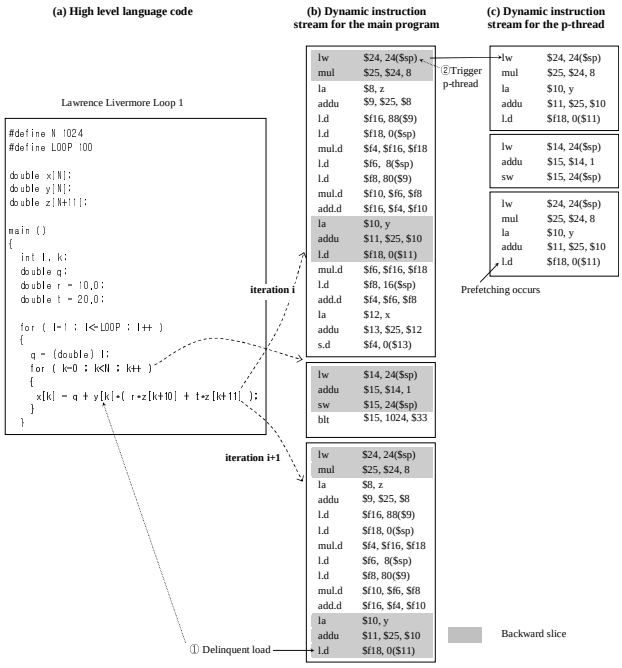

Figure 1: Speculative pre-execution example

The p-thread runs on another hardware context and only updates the cache status without store operations. Therefore, it does not affect the program state. For the multithreaded execution of the p-thread, previous research projects use either SMT architecture [7][8][12][15][17][21] or the additional pipeline for p-thread [1]. Indeed, the second method can be considered an extension of the CMP architecture. In the SMT

model, the resources can be shared between the main program thread and the p-thread. Therefore, high utilization can be achieved. On the contrary, in the dedicated prefetching pipeline model, more dedicated resources are given to prefetching thread.

2.2. Related Work

Various versions of data prefetching using speculative pre-execution have been proposed recently. Depending on the hardware/software implementation, there have been several approaches. Roth and Sohi proposed Speculative Data-Driven Multithreading (DDMT) [20][21]. In their approach, the miss stream called *performance degrading slice* or *data-driven thread* (DDT) includes the future cache miss instructions (*critical instructions* in [21]) and their backward slice. The miss stream is executed in a multithreaded manner using another spare context of Simultaneous Multithreading. The concept of *performance degrading slice* was originally introduced by Zilles and Sohi in [24].

Collins et al. developed Speculative Precomputation using the SMT features of the Itanium processor in [7]. They also define a small number of static loads as *delinquent loads* and include the backward slice as Precomputation Slices (*p-slices*) for data prefetching thread. Notably, their work introduced a new concept named the *chaining trigger* mechanism which allows the speculative thread to trigger another speculative thread. A hardware approach of Speculative Precomputation can also be found in [8]. They designed and implemented additional hardware resources to construct and trigger p-slices at run time.

Another approach using hardware for prefetching thread was introduced by Annavaram in [1]. A Dependence Graph Precomputation scheme (DGP) dynamically uncovers the prefetching slice for cache miss instructions. Whenever the Pre-decode stage detects the load/store instruction which is marked for prefetching (equivalent to delinquent load), it automatically derives the Dependence Graph. The instructions waiting in the Instruction Fetch Queue are chased based on the register dependencies. In the DGP scheme, the speculative prefetching slice runs on an additional piece of hardware called the Precomputation Engine and only updates the cache status. Another hardware approach is introduced as the Slice processor [18], which uses an additional hardware structure called *Slicer* to construct a p-thread in the commit stage instead of the fetch queue. The Slice processor stores the p-thread in the Slice-cache, and the p-thread is initiated upon detecting a trigger instruction in the main program flow.

Software controlled construction of the p-thread in a static way is also proposed in [12][15][17]. Luk proposed a high level language-based approach for speculative pre-execution [17]. A manual analysis on the given C code defines and annotates the prefetching slice (*p-thread*). The actual execution of the p-thread is supported by the SMT features of the architecture. However, the trigger operation is handled totally in software. Another approach at the high-level language 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 [15] 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.

3. Compiler Assisted Speculative Pre-Execution

Finely tuned speculative pre-execution with intelligent compiler support can provide an effective data prefetching method. In the proposed architecture model, we aim to emphasize the relative merit of the *software approach* vs. the *hardware approach*. Indeed, the Compiler Assisted Speculative Pre-execution (CASP) is a hybrid model. In this section, the design motivation and architecture characteristics are explained in detail. Also, the detailed software algorithms and hardware design are presented.

3.1. Overall Design Concepts

The actual design procedure of the speculative pre-execution is composed of three major steps. The first step is defining the delinquent loads. It is usually driven by the cache access-profiling. The second step is the construction of the p-thread, which includes the delinquent loads and the backward slice. The final step is the runtime execution of the p-thread. The three square boxes in Figure 2 show the three-step procedure of the speculative pre-execution. The two core operations of the speculative pre-execution are the construction of the p-thread and the triggering of the p-thread. Indeed, our design motivation is located in the proper interaction between these two steps. The lower box in the Figure 2 shows the design characteristics of the proposed CASP architecture model.

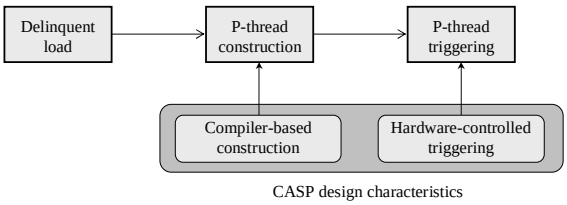

Figure 2: Procedure of speculative pre-execution

To achieve effective speculative prefetching, the first important design choice is in constructing the p-thread. Previous research falls into two distinct categories, depending on how to construct the p-thread: *Compiler-based construction* and *Hardware-based construction.* The compiler-based approach extracts the p-thread at either source code level (for example, C to C compilation) [12][17] or at the binary level [15]. The main advantage of the compiler oriented p-thread construction is the use of global program information. Also, the complex hardware logic for the p-thread construction can be eliminated: although the hardware-based p-thread construction [1][8][18] is fast, it imposes additional hardware logic.

Another design selection should be made regarding the triggering method of the p-thread. First, *Software-controlled triggering* uses the software code for the spawning of the p-thread. It is achieved by the multithreaded hardware and the spawning procedures (such as searching available context and propagating the live-in values). Usually, the p-thread constructed by the compiler is also initiated by the software-controlled triggering. The second method, *Hardware-controlled triggering*, utilizes additional hardware to rapidly spawn the p-thread. Generally, the p-thread constructed by the hardware also depends on the hardware for triggering. The trade-offs between the two approaches are between software-overhead and hardware-complexity.

Our architecture model is motivated by the fact that the *Compiler-based construction* can also potentially benefit the *Hardware-controlled triggering*. In other words, our architecture depends on the compiler analysis to construct the p-thread and utilizes the hardware to trigger the p-thread at runtime. It is achieved by delivering the p-thread information down to the architecture level by annotating each instruction. Since the two steps of p-thread construction and p-thread triggering can be separated and are not required to bond to the same methodologies, our architecture selects the most beneficial design technique at each step. Therefore, we can find a compromise design with synergy effects between both designs. A more detailed description for each step is given in the following sections.

3.2. P-thread Construction with Global Program Flow

In this section, the compiler operation for the p-thread construction is discussed in detail. Our p-thread construction tool works at the binary level. At first, the basic blocks are identified and the loop region for the delinquent load is defined. Indeed, the region based p-thread construction can be achieved with the necessary program flow information such as data flow and control flow. For detailed analysis, the *Pointer Stressmark*, which is one of the seven benchmarks in Atlantic Aerospace DIS Stressmark Suite [27], is considered as an example.

The source code of the inner loop of the *Pointer Stressmark* is shown in Figure 3. It is a pointer chasing benchmark following the median value of a given size of window. Each pointer chasing is expressed as a *hop*. In the source code, one iteration of the while loop corresponds to one hop operation and the outer *for-loop* (with an increment of *ll*) searches for the median of the corresponding hop. The median value decides the starting point of the next hop. Upon detecting that the current index has the median of the current hop, the control flow exits the for-loop.

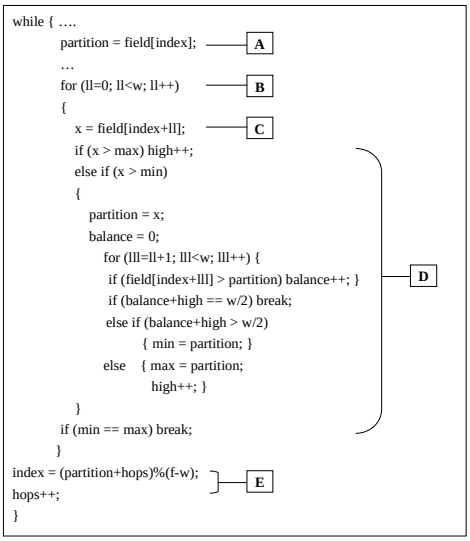

Figure 3: Loop operation of the Pointer Stessmark

The program flow with the basic blocks for the above source code is depicted in Figure 4. The solid lines with an arrow indicate the control flow. Each basic block is named (such as A, B, C, and E) as matching to the corresponding part of the source code in Figure 3. For readability, the part to find the median value (named D in Figure 3) is just combined as one extended block (which is the square block named D in Figure 4). Although block D is not a basic block, it is easier to understand to bind it as a block. In fact, there exist several basic blocks inside of D.

Upon finding the median value inside block D, the control flow exits the "C-D-B" loop and moves to the basic block E (The "C-D-B" loop corresponds to the outer *for-loop* with an increment of *ll*). The index value for the next hop is calculated at the basic block E, and a new iteration for the next hop starts. Indeed, the backward edge from the basic block E to the basic block A identifies the control flow to the next hop. Since the value of the *index* variable is decided by the median of

the random numbers at the previous iteration, the starting load (which is *partition = field [index]* in the basic block A) of each hop is prone to cache misses. In our cache access-profile, the load instruction for the *partition* in the basic block A is found to cause a considerable amount of cache misses. Therefore, the instruction is defined as a *delinquent load* for the analysis. Based on the data dependencies and the control flow, we can find the backward slice of the delinquent load, and finally, the p-thread can be constructed.

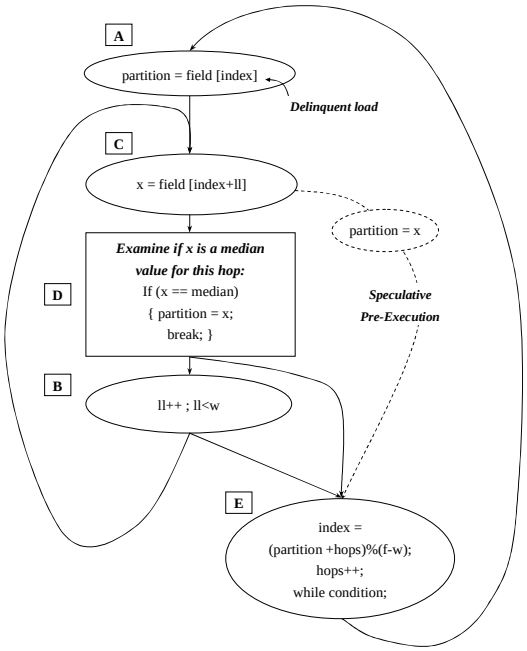

Figure 4: Program flow with the basic blocks

In the speculative pre-execution model, the delinquent loads and the backward slice are extracted from the original program and executed in a multithreaded manner for the data prefetching. In our example, the load instruction in the basic block A is defined as a delinquent load. The first statement for the backward slice is the statement for the variable *index* (which is *index = (partition + hops) % (f-w);*) in the basic block E. Then, the statement for the variable *partition* in block D (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.

The p-thread is composed of the delinquent load and minimal instructions required to compute the input value of the delinquent load. It is executed in parallel with the main program thread and should be lightweight in order to run faster than the main program flow. The actual beauty of speculative pre-execution of this example lies in skipping some operations in block D and jumping to the basic block E speculatively. The minimum necessary instruction is *partition = x*. The dotted line from the basic block C to E in Figure 4 shows the speculative pre-execution path. The final code for the p-thread is shown in Figure 5-(b).

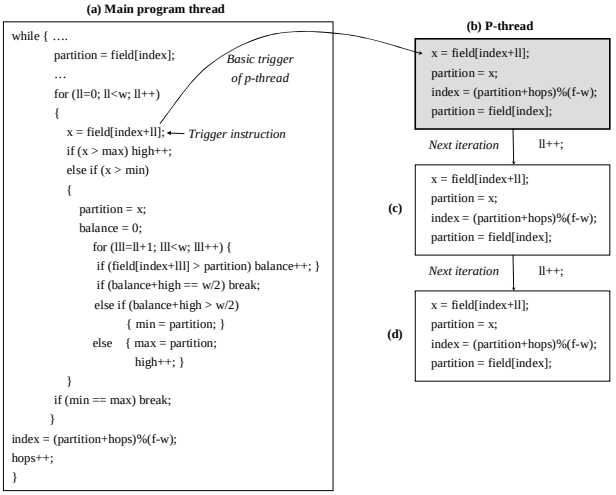

Figure 5: Triggering of the p-thread

The code in Figure 5-(a) runs as the main program flow and triggers the p-thread at runtime. The arrow from Figure 5-(a) to Figure 5-(b) shows the triggering operation. For the speculation of the multiple iterations, manipulation to increment variable *ll* also can be added to the p-thread. Figure 5-(c) and Figure 5-(d) shows the speculative pre-execution across the multiple loop iterations. It is a similar operation to the chaining trigger [7][8].

3.3. Hardware Description for P-thread Triggering

In the CASP architecture model, several hardware implementations are designed to facilitate triggering of the p-thread. Figure 6 shows the detailed hardware description of the CASP architecture model. It is based on the Simultaneous Multi-Threading architecture [10]. The resources, except for the register files and the reorder buffers, are shared between the main program thread and the p-thread. Since the p-thread instructions are copied from the instruction fetch queue, the additional fetch units for the p-thread are not required. Therefore, the p-thread control flow strongly depends on the branch prediction of the main program flow. It is a reasonable assumption with the effective branch prediction strategies of the current processor architecture. A similar observation is made in [1].

The most important structure of our architecture is a long-range instruction fetch queue (IFQ). The IFQ

identifies the p-thread instructions and supplies those identified instructions to the p-thread reorder buffer (p-ROB) via the decoding logic. Since the p-thread information is annotated with each instruction, a simple pre-decoding logic at the IFQ can identify the p-thread. It is done at the "detect p-thread" stage among the six pipeline stages. The IFQ is a FIFO queue with one additional bit named the *p-thread indicator* for each entry. The p-thread indicator specifies whether the instruction in the entry is included as the p-thread or not.

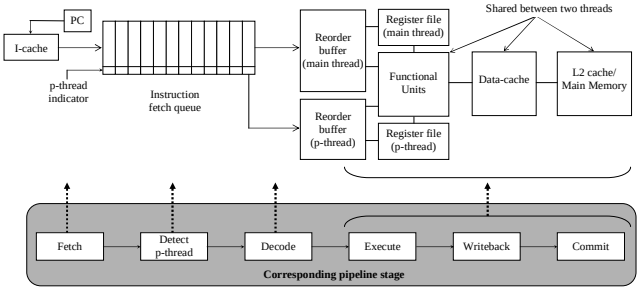

Figure 6: Hardware description of the CASP architecture

Although an instruction is detected as a p-thread instruction, it is dormant in the instruction fetch queue until any trigger instruction is found at the decoding stage. Upon detecting a trigger instruction, the decode logic initiates the *triggering state*. Then, the triggering logic is activated and waits until all instructions ahead of the trigger instruction are committed. It guarantees the deterministic state before copying the live-in values. After the trigger instruction becomes the oldest instruction in the reorder buffer of the main thread, a spawning operation is initiated. The spawning operation copies all the live-in register values of the main thread to the p-thread register file. It also copies the p-thread instructions which reside in the main thread reorder buffer, to the p-thread reorder buffer.

After copying the live-in register values and the necessary instructions, the p-thread is executed as an independent running thread. Since the processor is now running as a multithreaded processor, every operation should be accompanied with the thread ID. We assign "0" to the main program thread as a thread ID, and "1" to the p-thread. The IFQ supplies the p-thread instructions to the decoding logic based on the p-thread indicator. Although the p-thread instruction is sent to the decoding logic as an operation of p-thread execution, it should also be executed as the main thread. Indeed, the instructions which have a p-thread indicator "on" should be included in both threads. Therefore, every p-thread instruction should remain in the IFQ once it is decoded as a part of the p-thread. For that purpose, we can check it as "executed as a part of a p-thread" by resetting the p-thread indicator as "off". Therefore, only the instructions of which the p-thread indicator is "off", are decoded as the main thread under the triggering state. They are entitled to exit the IFQ after decoding.

The decoding logic renames the registers and sends instructions to the corresponding reorder buffer based on the thread ID. Other operations in the remaining pipeline stages are very close to the existing SMT architecture model. The functional units, the cache and the memory are shared between two threads. After the delinquent load retires the commit stage, the triggering operation is finished and the processor returns to the normal state.

4. Experimental Results

The performance of the proposed architecture is accurately evaluated with a number of data intensive benchmark programs. We performed a deterministic simulation on the CASP architecture and analyzed it based on the performance results.

4.1. Benchmark Descriptions

Table 1: Benchmarks Description

Benchmark	Problem	Characteristic
Data Management	Traditional DBMS processing	Index algorithms and ad hoc query processing
SAR Ray Tracing	SAR image simulation	Utilizes Image-domain approach
Pointer	Pointer following	Small blocks at unpredictable locations. Can be parallelized
Update	Pointer following with memory update	Small blocks at unpredictable locations
Field	Collect statistics on large field of words	Regular, with little re-use
Neighborhood	Calculate image texture measures by finding sum and difference histograms	Regular access to pairs of words at arbitrary distances
Transitive Closure	Find all-pairs-shortest-path solution for a directed graph	Dependent on matrix representation, but requires reads and writes to different matrices concurrently

The target applications of our architecture are memory-bound applications. Applications causing large amounts of data traffic are also called as data-intensive applications. We chose two applications from the Atlantic Aerospace Data-Intensive Systems Benchmarks Suite [26] and five applications from the Atlantic Aerospace Stressmark Suite [27]. Table 1 shows the characteristics of the seven benchmarks. The detailed descriptions have been obtained from [26][27].

Indeed, today's popular applications such as database management and image processing often experience non-contiguous memory access patterns. Therefore, processor stalling is easily caused by data starvation. These applications are more stream-based and result in more cache misses due to the lack of locality. Moreover, the increasing use of the Object-Oriented Programming model correspondingly increases the underlying use of the pointers. Due to the serial nature of the pointer processing, memory accesses become a severe performance bottleneck in existing computer systems.

4.2. Simulation Environments

Our simulator is based on the SimpleScalar 3.0 tool set [2]. The p-thread construction is implemented by modifying the *sim-safe.c* module. The architectural simulator, which models the CASP architecture in Section 3.3, is implemented based on the *sim-outorder.c* module. It models the detailed pipeline operation as well as architectural delays. The parameters are summarized in Table 2.

Table 2: Simulation parameters

Branch predict mode	Bimodal
Branch table size	2048
Issue width	8
Commit width	8
Instruction fetch queue size	various (128, 256, 512, and 1024)
Reorder buffer size	64 instructions
Integer functional units	ALU(x 4), MUL/DIV
Floating point functional units	ALU(x 4), MUL/DIV
Number of memory ports	2
Data L1 cache configuration	256 sets, 32 block, 4 -way set associative , LRU
Data L1 cache latency	1 CPU clock cycle
Unified L2 cache configuration	1024 sets, 64 block, 4 – way set associative, LRU
Unified L2 cache latency	12 CPU clock cycles
Memory access latency	120 CPU clock cycles

The configurations we have tested are the baseline superscalar architecture with a 64 entry reorder buffer and the CASP architectures with various sizes of IFQ. Since the IFQ size is believed to affect the prefetching capability of the p-thread, we simulated various IFQ sizes: 128, 256, 512 and 1024. The results are presented in the next section.

4.3. Results and Analysis

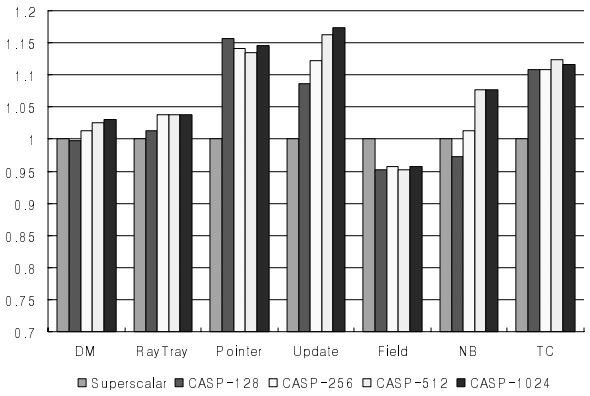

Figure 7: Performance comparison to the baseline superscalar architecture

Figure 7 shows the performance results of the CASP architecture. Five architecture models including the baseline superscalar architecture have been simulated with each benchmark. For each benchmark, the far left bar corresponds to the normalized performance of the baseline superscalar architecture. The remaining four bars show the CASP architecture with four different IFQ sizes (128, 256, 512, and 1024 respectively). The original performance is measured by instruction per cycle (IPC). For demonstration purposes, the diagram shows the normalized performance based on the baseline superscalar architecture.

The best result reaches a 17.3% performance improvement, which is achieved with the *Update Stressmark* under the 1024-entry IFQ configuration. Indeed, the six benchmarks show better performance of the CASP over the baseline superscalar architecture. Only the *Field Stressmark* experienced performance degradation with all four CASP models. It is because the *Field Stressmark* contains a relatively small number of cache misses and therefore cannot benefit much from the speculative pre-execution feature of the CASP.

Regarding the IFQ size, three benchmarks (*Data Management*, *Update*, and *Neighborhood*) show performance enhancement with an increase of the IFQ. However, other benchmarks do not show any benefit from the long range IFQ. It is due to the fact that the long range IFQ also suffers from mispredicted branches.

The same observation is made in [1]. The average performance enhancement over the baseline superscalar architectures for each IFQ configuration is shown in Figure 8. The average performance enhancement is calculated with all seven benchmarks.

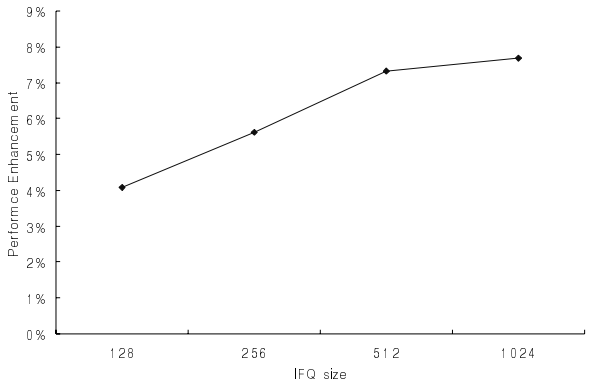

Figure 8: Average performance improvement with various IFQ sizes

To show the effectiveness of the prefetching of the CASP architecture, the number of level 1 cache misses has also been calculated. Figure 9 shows the reduction of the total cache misses of the CASP (with 512 IFQ), as compared to the baseline superscalar architecture. As the results indicate, the number of cache misses is considerably reduced by the speculative pre-execution of the CASP architecture. The best result is achieved by the *Transitive Closure Stressmark*, which reduces 22% of the cache misses. On the average, cache misses are reduced by 14.2% in the CASP (with 512 IFQ) architecture.

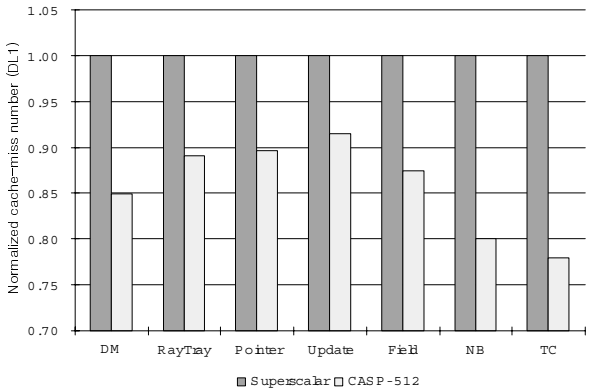

Figure 9: Reduction of cache miss rate compared to the baseline architecture

Although the *Data Management*, *Ray Tracing*, and *Field Stressmark* reduce the cache misses considerably, the performance improvement (which is demonstrated in Figure 7) of the three benchmarks is not extensive. It is due to the fact that the total number of cache misses is relatively small in these three benchmarks. Therefore, the reduction of the cache misses has less impact on the performance than the other four benchmarks.

To demonstrate how well the CASP would tolerate a long memory latency, the benchmarks are simulated under various memory latencies. The performance results with four benchmarks (*Data management*, *Pointer*, *Update*, and *Neighborhood*) are depicted in Figure 10. The longest latency configuration we considered is: *memory access latency = 160* and *L2 cache access latency = 16*. The shortest case is: *memory access latency = 40* and *L2 cache access latency = 4*. Two more cases are designed between the above two cases.

The prefetching capability of the CASP architecture provides robust performance at the long latency configurations compared to the baseline superscalar architecture. As the results indicate, the performance of the CMAS is fairly stable at the long latencies. Only the *Data Management*, which showed relatively small performance enhancement in Figure 7, experienced performance degradation at the long latencies. On the contrary, the performance of the superscalar architecture drops severely when all four benchmarks are faced with long memory latencies.

5. Conclusions

It has been a truism that the memory access regularity is diminishing in today's popular applications. Hence, the traditional data prefetching methods, which strongly depend on the memory address predictions, often fail. As a result, there is a strong need for new data prefetching methods in the modern processor architecture field. As one possible solution, we have presented the new data prefetching method named CASP (Compiler Assisted Speculative Pre-execution).

The CASP architecture is a hybrid model of the two previous approaches (*software-based* and *hardware-based*) for the speculative pre-execution method. We demonstrated the performance results of the proposed architecture with seven data intensive benchmarks. The compiler assisted p-thread construction and the hardware supported triggering of the p-thread coordinate quite well and yield good performance results.

Future work could investigate more on the effect of the IFQ size for the effective speculative prefetching. The IFQ size also can affect the prefetching distance as well as the definition of the triggering instructions. Also, the relationship between the various branch prediction schemes and the speculative prefetching capability will be further investigated. In addition to that, more compiler algorithms for the automated p-thread construction will be followed.

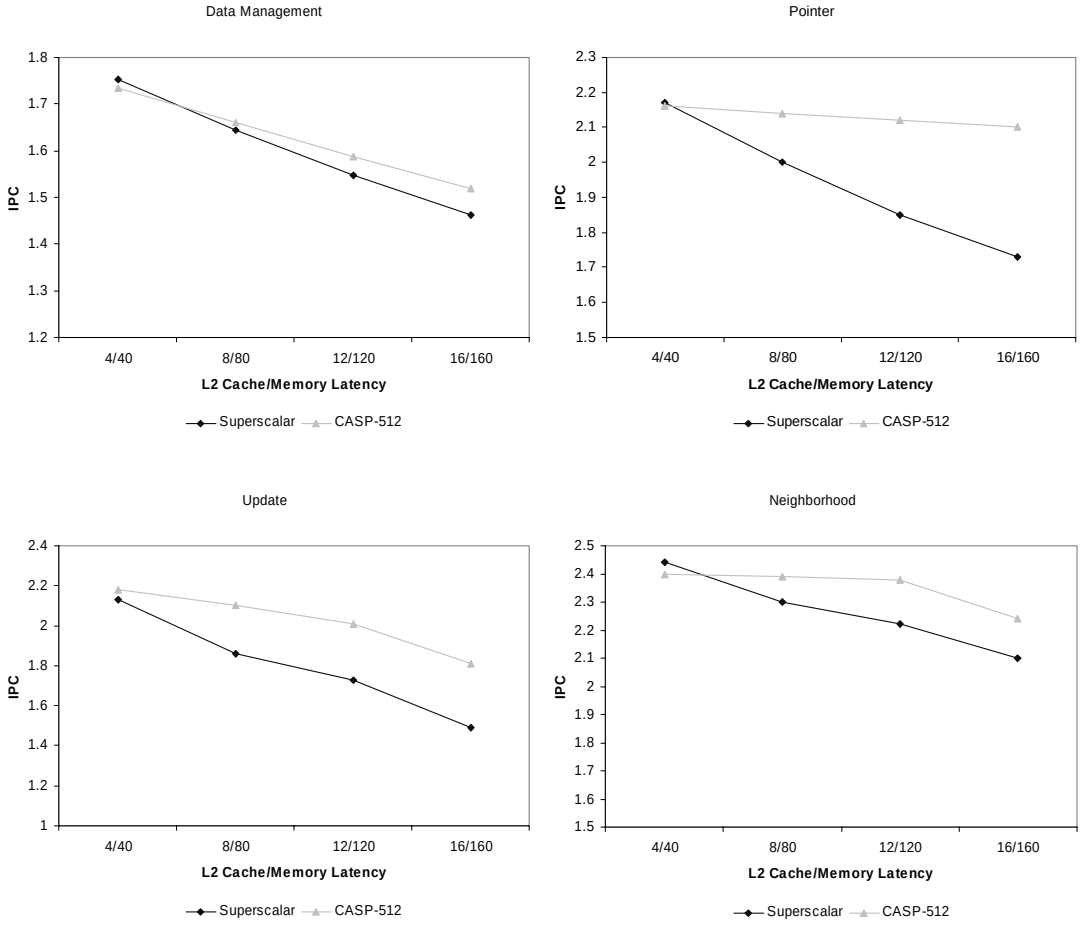

Figure 10: Latency tolerance for various memory latencies

References

[1] M. Annavaram, J. M. Patel, and E. S. Davidson, Data Prefetching by Dependence Graph Precoumputation. In *Proceedings of the 28th International Symposium on Computer Architecture*, June 2001.

[2] D. Burger and T. Austin. The SimpleScalar Tool Set. Version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin-Madison, June 1997.

[3] J. Burns and J.-L. Gaudiot. SMT Layout Overhead and Scalability. *IEEE Transactions on Parallel and Distributed Processing Systems*, Volume 13, Number 2, Feb. 2002.

[4] R. Chappell, J. Stark, S. Kim, S. Reinhardt, and Y. Patt, Simultaneous Subordinate Microthreading (SSMT). In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, May 1999.

[5] T.-F. Chen and J.-L. Baer. Effective Hardware-Based Data Prefetching for High-Performance Processors. *IEEE Transactions on Computers*, 44(5):609--623, May 1995.

[6] S. Crago, A. Despain, J.-L. Gaudiot, M. Makhija, W. Ro, and A. Srivastava. A High-Performance, Hierarchical Decoupled Architecture, In *Proceedings of MEDEA Workshop*, Oct. 2000.

[7] J. D. Collins, H. Wang, D. M. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. P. Shen. Speculative Precomputation: Long-range Prefetching of Delinquent Loads, In *Proceedings of the 28th International Symposium on Computer Architecture*, June 2001.

[8] J. D. Collins, D. M. Tullsen, H. Wang, and J. P. Shen. Dynamic speculative precomputation. In *Proceedings of the 34th Annual International Symposium on Microarchitecture*, Dec. 2001.

[9] M. Dubois and Y. Song. Assisted execution. Technical Report CENG #98-25, Department of EE-Systems, University of Southern California, Oct. 1998.

[10] S. Eggers, J. Emer, H. Levy, J. Lo, R. Stamm, and D. Tullsen. Simultaneous Multithreading: A Platform for Next-generation Processors, *IEEE Micro*, Sep./Oct. 1997.

[11] S.I. Hong, S.A. McKee, M.H. Salinas, R.H. Klenke, J.H. Aylor, and W.A. Wulf, Access Order and Effective Bandwidth for Streams on a Direct Rambus Memory. In *Proceedings of the 5th International Symposium on High-Performance Computer Architecture*, Jan. 1999.

[12] Dongkeun Kim and Donald Yeung. Design and Evaluation of Compiler Algorithms for Pre-Execution., In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems.* San Jose, CA, October 2002.

[13] V. Krishnan and J. Torrellas. A Chip-Multiprocessor Architecture with Speculative Multithreading. *IEEE Transactions on Computers*, Vol. 48, No. 9, Sep. 1999.

[14] L. Kurian, P. T. Hulina, and L. D. Coraor, Memory Latency Effects in Decoupled Architectures. *IEEE Transactions on Computers*, vol. 43, no. 10, Oct. 1994.

[15] Steve S.W. Liao, Perry H. Wang, Hong Wang, Gerolf Hoflehner, Daniel Lavery, and John P. Shen. Post-Pass Binary Adaptation for Software-Based Speculative Precomputation, In *Proceeding of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, Jun, 2002.

[16] C.-K. Luk and T. C. Mowry. Compiler Based Prefetching for Recursive Data Structures. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1996.

[17] C.-K. Luk. Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processor, In *Proceedings of the 28th International Symposium on Computer Architecture*, June 2001.

[18] Andreas Moshovos, Dionisios N. Pnevmatikatos, and Amirali Baniasadi, Slice-Processors: An Implementation of Operation-Based Prediction, In *Proceedings of the 15th international conference on Supercomputing*, June 2001.

[19] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. A Case for Intelligent DRAM: IRAM. *IEEE Micro*, April, 1997.

[20] A. Roth, C. B. Zilles, and G. S. Sohi, Speculative Miss/Execute Decoupling. In *Proceedings of MEDEA Workshop*, Oct. 2000.

[21] A. Roth and G. S. Sohi, Speculative Data-Driven Multithreading, In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, Jan. 2001.

[22] J. Smith. Decoupled Access/Execute Computer Architecture. In *Proceedings of the 9th International Symposium on Computer Architecture*, Jul. 1982.

[23] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, June 1995.

[24] C. B. Zilles and G. S. Sohi, Understanding the Backward Slices of Performance Degrading Instructions. In *Proceedings of the 27h International Symposium on Computer Architecture*, June 2000.

[25] C. Zilles and G. Sohi. Execution-based prediction using speculative slices. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, June 2001.

[26] Data-Intensive Systems Benchmarks Suite Analysis and Specification , http://www.aaec.com/projectweb/dis/

[27] DIS Stressmark Suite, http://www.aaec.com/projectweb/dis/DIS_Stressmarks_v1_0.pdf