Design and Effectiveness of Small-Sized Decoupled Dispatch Queues [*]

Won W. Ro[1] and Jean-Luc Gaudiot[2]

[1] Department of Electrical and Computer Engineering
California State University, Northridge
wro@csun.edu
[2] Department of Electrical Engineering and Computer Science
University of California, Irvine
gaudiot@uci.edu

Abstract. Continuing demands for high degrees of Instruction Level Parallelism (ILP) require large dispatch queues in modern superscalar microprocessors. However, such large queues are inevitably accompanied by high circuit complexity which correspondingly limits the pipeline clock rates. This is due to the fact that most of today's designs are based upon a centralized dispatch queue which depends on globally broadcasting operations to wake up and select the ready instructions. As an alternative to this conventional design, we propose the design of hierarchically distributed dispatch queues, based on the access/execute decoupled architecture model. Simulation results based on 14 data intensive benchmarks show that our DDQ (Decoupled Dispatch Queues) design achieves performance comparable to a superscalar machine with a large dispatch queue. We also show that our DDQ can be designed with small-sized, distributed dispatch queues which consequently can be implemented with low hardware complexity and high clock rates.

1 Introduction

Reaching high degrees of Instruction Level Parallelism (ILP) through multiple-instruction issue and out-of-order execution has been an essential part of modern microprocessor design. During the last decade, superscalar architectures have dominated the commercial market by adopting a hardwired scheduling logic that enables dynamic instruction scheduling. However, conventional dynamic scheduling possesses an inherent scaling problem as far as the size of the *dispatch queue* is concerned since the wake up and select logic requires a one-cycle operation and cannot be pipelined [1].

Another important issue is how to solve the dramatically growing speed gap between processor and main memory. This performance gap causes long access latencies at cache misses and forces the cache miss instructions to be stalled. It

[*] This paper is based upon work supported in part by NSF grant CCF-0541403. Any opinions, findings, and conclusions or recommendations are those of the authors and do not necessarily reflect the views of NSF.

consequently means all the instructions that depend on the cache miss instructions should stay inside the dispatch queue. In fact, those instructions would occupy the slots for considerable amounts of time, which would result in a reduction of the number of available entries in the dispatch queue. Therefore, the long memory latency also implies the need for a large dispatch queue. However, as described earlier, a large queue will eventually cause a scaling problem.

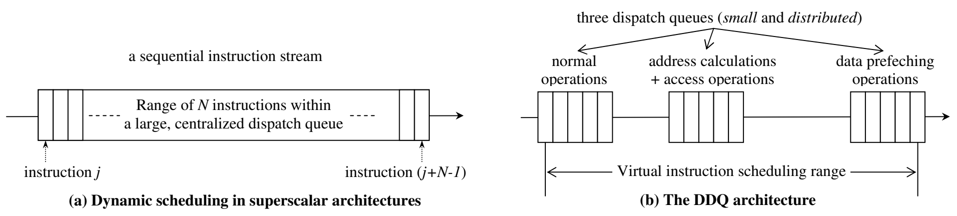
a sequential instruction stream
Range of N instructions within a large, centralized dispatch queue
instruction j
instruction (j+N-1)
(a) Dynamic scheduling in superscalar architectures
three dispatch queues (small and distributed)
normal operations
address calculations + access operations
data prefeching operations
Virtual instruction scheduling range
(b) The DDQ architecture

Fig. 1. Distributed instruction scheduling on the decoupled dispatch queues

As an alternative to a large dispatch queue, we propose Decoupled Dispatch Queues (DDQ) which can be implemented with a three small-sized dispatch queues. It aims at reducing the critical path delay of a large queue. The basic motivation is to mask the long memory access latencies without increasing the size of a single dispatch queue. The DDQ enables asynchronous scheduling of three instruction groups which are separated according to the memory access role of the instructions (computation instructions, memory access instructions, and prefetching instructions); this means there is a dedicated dispatch queue for each of the three instruction groups. Three dispatch queues are, at any given moment, asynchronously dealing with different points of a sequential instruction stream. However, it is virturally operating as if we had a large queue (Fig. 1).

Performance evaluation is based on a cycle-time simulator which is developed from SimpleScalar 3.0 [2]. Compared to a superscalar architecture with a 256-entry dispatch queue, our DDQ achieves a similar performance (*98.5%*) with three 128-entry dispatch queues. When the dispatch queue is reduced by as much as one fourth (64 entries), the DDQ still performs at *91.3%* of the baseline performance. With 32-entry dispatch queues, the performance still remains as high as *86.7%*. Moreover, reduction in the queue size will eventually contribute to the higher clock rate.

The rest of the paper is organized as follows. In Section 2, we describe background research and previous work related to the complexity-effective dispatch queue design. Section 3 presents the detailed description of the proposed DDQ architecture. Section 4 includes experimental results and performance analysis. Conclusions and future work are included in Section 5.

2 Background Research

Access/execute decoupled architecture concepts are not new and we now describe them in some detail, while several related research projects are surveyed.

2.1 Access/execute decoupled architectures

Access/execute decoupled architectures have been developed to tolerate long memory access latencies [3–10]. Latency tolerance is achieved by separating the original, single instruction stream into two streams: the *access stream* and the *execute stream*. By definition, the access stream includes memory access operations: load/store instructions and address calculation instructions. Other remaining instructions (commonly referred to as computation instructions) are included in the execute stream. Timely data prefetching can be achieved by running the access stream ahead of the execute stream; any processor stalling due to data delivery can be eliminated by the early execution of the access stream. The time difference between the access instruction produces a data element and the execute instruction needs the data is called the *slip distance*. The two independent instruction streams processed by each processing unit exploit instruction-level parallelism while providing memory latency tolerance. In general, the communications between the two streams are achieved via a set of FIFO queues.

2.2 Related work

There have been several research projects which have sought to solve the complexity problem of a large dispatch queue by splitting it into multiple queues. Palacharla *et al.* have performed an initial analysis of the potential complexity of large window superscalar architectures [1]. They have proposed a dependence-based instruction queue design, in which the instructions are sent to separate FIFO queues based on the data dependencies. At the issue stage level, only the head instructions of each FIFO queue are considered for issuing. Their initial analysis demonstrates the advantage of a small-sized queue and has motivated further research on the clustered microprocessor design. The clustering is essentially related to the partitioning of a dispatch queue and functional units [1, 11–13]. Also, clustered architectures separate the instructions based on register dependencies. Furthermore, the *speculative multithreading* technique has been developed [14, 15] with the idea in mind of focusing on software separation (thread selection and scheduling) and speculative thread spawning on each separated processing unit.

Although the distributed queue design has been proposed in many prior research projects, none of them separate the instruction stream based on the memory access functionality as originally proposed in the early decoupled architectures. Actually, the access/execute decoupled architecture model can even be considered one type of clustered architectures. However, the difference lies in the separation of the instruction streams; the task separation is done according to the memory operations in the decoupled architectures and our DDQ.

Several previous projects in decoupled architectures also attempted to solve the complexity problem of superscalars [3, 10]. However, none of them addressed the problem of the cache misses on the access processor. To the best of our knowledge, DDQ is the first work which proposes an implementation of data prefetching on the access/execute decoupled architectures (except our previous work in [16]).

3 DDQ: hierarchically decoupled dispatch queues

This section first describes the problems of traditional decoupled architectures and presents the idea behind our development of the proposed DDQ architecture. It also includes the hardware and software descriptions of the design.

3.1 Problems of the access/execute decoupling

Our initial motivation is to solve the complexity problem of a monolithic dispatch queue in superscalar machines by using access/execute decoupled architectural concepts. As described earlier, the advantage of decoupled architectures can be exploited only if the slip distance is larger than the memory access latency. However, several factors in the current access/execute decoupled architecture designs prevent the access stream from running far ahead of the execute stream.

First of all, frequent synchronization between the two streams prohibits early execution of the access stream. In fact, the access stream also requires data from the execute stream; some control operations as well as data operations need data from the computation results of the execute stream. Therefore, synchronization between the two streams can happen at a certain point of the execution. We call this phenomenon a *loss of decoupling* event [17].

Secondly, frequent cache misses in the access stream prevent early execution of the access stream running on the access processor (AP). If a cache miss on the AP has a sufficient time until any instruction in the EP requires the data, the latency can be tolerated. However, frequent cache misses may cause the access processor to lag further behind. For example, two or more consecutive cache misses on the AP will slow down the execution of the access stream. From the above observations, we find that the cache misses in the access processor should be reduced.

3.2 Description of the DDQ architecture

The DDQ architecture includes one additional processing unit to achieve data prefetching on the access processor. Consequently, our architecture requires one more stream separation in addition to the access steam and the execute stream. An additional stream named the *data prefetching stream* is intended to run ahead of the access stream, achieving another hierarchy of the prefetching from the memory to the L1 data cache. Fig. 2 shows the proposed DDQ architecture. It has a single fetch unit and separates three streams at the pre-decoding stage. The stream separation information (which indicates the stream to which the instruction belongs) is already annotated with each instruction at compile time (it is described in the following subsection).

Three dedicated processing units for each of the execute stream, the access stream, and the data-prefetching stream are loosely combined; they are respectively the EPU (Execute Processing Unit), the APU (Access Processing Unit), and the DPPU (Data-Prefetching Processing Unit). The operations of the EPU and the APU are very similar to that in conventional access/execute decoupled architectures. The load data queue (LDQ) and the store data queue (SDQ) facilitate communications between the EPU and the APU. To guarantee the correctness of the communication order between the two processors, we use the indexed data queue concept which is first introduced in the DS (Decoupled Superscalar) architecture [10]. The indexed data queues are implemented to declare the FIFO order which is assigned at decoding time. However, the queue entries can be accessed out-of-order.

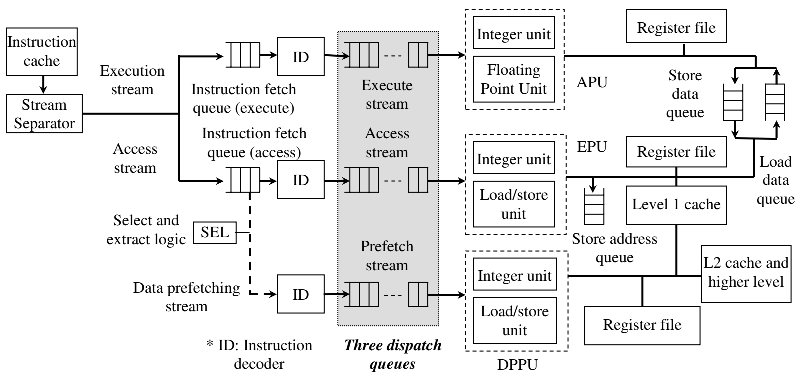

Fig. 2. The DDQ architecture

The basic idea behind the DPPU is similar to the speculative pre-execution concept [18, 19], which extracts the future probable cache miss slices from the original code and executes them as an additional prefetching thread. The data-prefetching stream of the DDQ is equivalent to the p-thread in speculative pre-execution [18]. It contains the future probable cache miss instructions (target loads) and their backward slice (backward slice includes every instruction upon which the target loads have data dependencies). Access profiling is used to detect probable cache miss instructions at compiler time. The DPPU operation is very loosely coupled with the processor above it since data communications occur only through the L1 data cache.

The execution of the DPPU is triggered at runtime. When the stream separator detects the target load instructions, it triggers the execution of the DPPU. For that purpose, Select and Extract Logic (SEL) is implemented. When the triggering is initiated by the stream separator, the SEL is enabled and looks into the instruction fetch queue to select the instructions which are tagged as data-prefetching stream (those instructions have been pre-detected and tagged by the stream separator beforehand). After that, SEL extracts and sends those instructions to the instruction decoder which is dedicated to the data-prefetching dispatch queue. The extraction operation is a copy operation of the instruction bits, since the AP still needs to hold and execute those instructions. The separation information for the three streams is defined and embedded on each instruction by the DDQ binary translator which is described in the next subsection.

The main target of this design is to reduce the size of a single dispatch queue so that we can reduce the wire delay for the wake up and select logic. Although the total number of queue entries in the entire processor should be multiplied by three, the clock rate is only affected by the size of a single, largest dispatch queue. There are no data bypassing networks or wake up and select logic connected between two different queues.

3.3 Software support for the stream separation

The DDQ binary code is produced by the DDQ binary translator which directly works on the SimpleScalar binary code. The tool analyzes the SimpleScalar binary code and separates it into three streams based on the instruction functionality. After that, the annotation field of each instruction of SimpleScalar binary is used to convey the each stream information (including information on the target load instructions) down to the hardware.

The separation of the access stream and the execute stream is very similar to that in conventional decoupled architectures. At the beginning, each load/store instruction is defined as the access stream. After that, the backward slice of the load/store instruction is included in the access stream. The remaining instructions of the code are separated as part of the execute stream. In our design, additional separation for the data-prefetching stream must be identified for the DPPU operations. Basically, the data-prefetching stream, which includes the probable cache miss instructions and their backward slice, is a subset of the access stream. If an instruction has been detected as a frequently miss-causing instruction by the access profiling, it is identified as a target load instruction for prefetching operation and defined as a part of the data prefetching stream. Finally, its backward slice is chased and included as the data prefetching stream. More detailed description can be found in our previous work in [16].

4 Experimental results and analysis

This section presents the experimental results and the performance analysis of the DDQ architecture.

4.1 Simulation environment

The DDQ simulator has been designed based on the sim-outorder simulator of the SimpleScalar 3.0 tool set [4]. The baseline superscalar architecture for performance comparison has a 256-entry dispatch queue with 8-way issue and commit. In the DDQ model, each dispatch queue size is tested from 32, 64, to 128. The issue and commit width is also reduced to 4. The EPU is implemented with all the functional units except for the load/store units. The APU and DPPU only have integer units and load/store units. In addition, we assume 12 CPU cycles for L2 cache access latency and 120 cycles for memory access latency.

The set of benchmarks we have selected include 14 applications: six applications chosen from the Atlantic Aerospace Stressmark suite (pointer, update, field, neighborhood, transitive closure, and matrix), three benchmarks from the Atlantic Aerospace Data-Intensive Systems Benchmarks suite (data management, ray tracing, and fast Fourier transform), and five selected from the SPEC2000 suite (gzip, vortex, bzip2, art, and equake). The SPEC benchmarks have been compiled at peak optimization level and tested with the reference input set.

We have performed simulations with the above 14 benchmarks for the three different machine models: superscalar (*sus*), access/execute decoupled architecture (*aed*), and our model (*ddq*). For all simulation results, the performance is measured in terms of IPC (instructions per cycle) and normalized to the baseline superscalar models (*sus.256.8*). Note that the first term specifies the architecture model while the second and the third numbers correspond to the dispatch queue size and issue width. For example, *ddq.32.4* indicates a processor model for a DDQ configuration with 32-entry dispatch queues and 4-way issue width.

4.2 Performance results and analysis

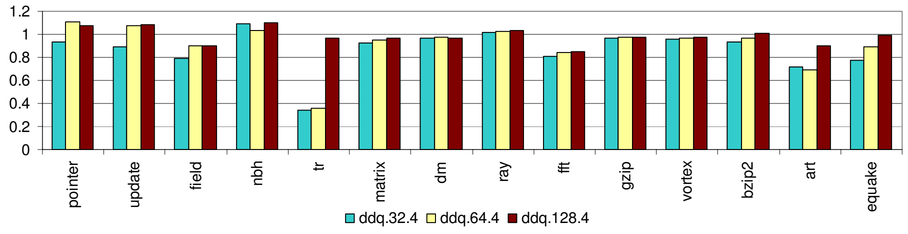

Fig. 3. Performance results for three DDQ models (normalized IPC to sus.256.8)

We have simulated three different configurations of the DDQ by using three dispatch queue sizes: 32, 64, and 128 entries. They are respectively called *ddq.32.4*, *ddq.64.4*, and *ddq.128.4*. The performance results for the three configurations are shown in Fig. 3. The performance of each model is measured in terms of IPC and normalized to that of *sus.256.8*. Although we cannot quantify the expected clock rates of our design at this point, we know that the smaller dispatch queues in our design would ultimately contribute to higher clock rates. Indeed, as previous research indicates [1], the critical path delay shows a quadratic dependency on the dispatch queue size and issue width.

As the results indicate, the *ddq.128.4* configuration yields a performance comparable to the baseline superscalar model in most benchmarks. However, four benchmarks (*field*, *tr*, *fft*, and *art*) show a weak performance compared to the other benchmarks. *Field* does not encounter many cache misses with the superscalar model and did not benefit from the data prefetching. Also, *tr* suffers from a low branch hit-ratio which prevents a successful speculative prefetching. As for *fft*, the DPPU has too many instructions in the prefetching stream. It causes cache pollution that correspondingly degrades the performance. In addition, *art* does not provide good performance since it works too well with the baseline model which has a 256-entry instruction window. The wide range scheduling is very beneficial to *art* and diminishes the advantages of the DDQ approach. The other 10 benchmarks show very close or even better performance compared to the baseline architecture.

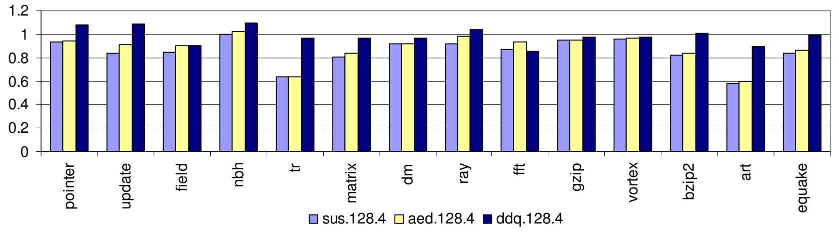

Fig. 4. Performance results with 128-entry queues (normalized IPC to sus.256.8)

For a further detailed analysis, Fig. 4 shows how *ddq.128.4* improves the performance over a traditional access/decoupled architecture (*aed.128.4*) and a superscalar (*sus.128.4*). The results demonstrate that *ddq.128.4* performs even better than the baseline model (*sus.256.8*) in 6 benchmarks in spite of having half-sized dispatch queues and half-sized issue width. However, *sus.128.4* and *aed.128.4* do not show good performance results in most benchmarks. In particular, *tr* and *art* shows noticeably low performance in the *sus* and *aed* configurations. We also performed benchmark simulations with 64-entry dispatch queues; the results show very similar tendency and characteristic. To avoid including too many redundant figures, only the average performance for the 64-entry configurations is presented later in this section.

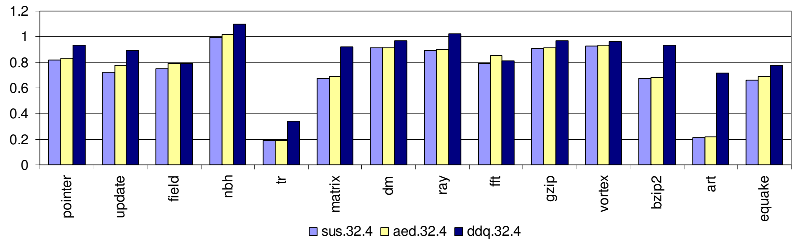

Fig. 5. Performance results with 32-entry queues (normalized IPC to sus.256.8)

Fig. 5 illustrates the performance of three architecture models with 32-entry dispatch queues and 4-way issue width; again, all results are normalized to *sus.256.8*. In this result, the dispatch queues in DDQ are as small as one eighths of the baseline model. However, the DDQ still reaches better than 80% of the baseline performance in 11 benchmarks. In contrast, more than half of the benchmarks (8 out of 14) cannot achieve 80% of the performance with *sus* and *aed*. More specifically, it should be noted that *tr* and *art* lose about 80% of the performance in those two configurations. Both models are affected much by the restriction of the queue size since neither configuration is assisted by prefetching.

The average performance over the 14 benchmarks is shown in Fig. 6. On average, *ddq.128.4* reaches up to 98.5% of the baseline performance with half-sized dispatch queues and half-sized issue width. However, the *aed.128.4* model experiences a 12.2% performance degradation. These results clearly demonstrate the advantage of the data prefetching operations of the DDQ. With *ddq.64.4*, the average performance still remains above 91%. More over, the *ddq* configuration remains in the range of over 86.7% of the baseline performance even for the smaller configurations such as 32-entry queues. However, *sus* and *aed* experience severe performance degradation when the dispatch queue is small.

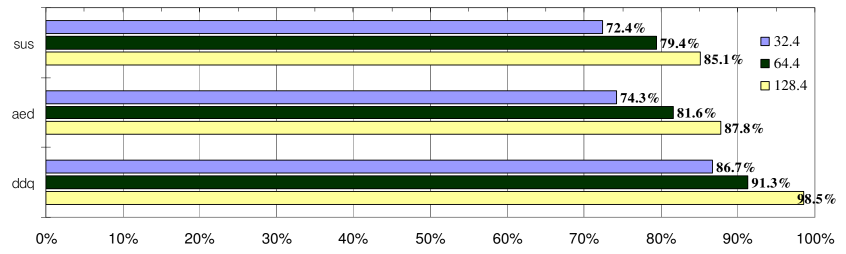

Fig. 6. Average performance with the small dispatch queue models

5 Conclusions and Future Work

The DDQ is based on the simple observation that the partitioning of a dispatch queue can reduce the complexity of a centralized design as well as the size of each component. Each processing unit is decoupled and works fairly independently of the others. The performance results show that the proposed architecture achieves performance comparable to that of the baseline superscalar architecture which has a large dispatch queue. In addition, our DDQ can be implemented with a faster clock since each processing unit has a smaller dispatch queue.

The main feature of the DDQ is having small dispatch queues, so that we can reduce the complexity and wire delay of the instruction scheduling logic. This eventually contributes to achieving higher clock rates. Even though the total number of queue entries over the DDQ grows with the each queue size times three, the clock rate is only affected by the size of a single dispatch queue. The three distributed dispatch queues do not require any data bypassing from the different functional units, nor share any instruction scheduling logic. Considering the clock rate improvement afforded by the size of a dispatch queue, these performance results are encouraging.

References

1. Palacharla, S., Jouppi, N.P., Smith, J.E.: Complexity-effective superscalar processors. In: Proceedings of the 24th Annual International Symposium on Computer Architecture. (1997)
2. Burger, D., Austin, T.: The simplescalar tool set. Technical Report CS-TR-97-1342, University of Wisconsin-Madison (1996)
3. Farrens, M., Nico, P., Ng, P.: A comparison of superscalar and decoupled access/execute architectures. In: Proceedings of the 26th Annual International Symposium on Microarchitecture. (1993)
4. Goodman, J.R., Hsieh, J.T., Liou, K., Pleszkun, A.R., Schechter, P.B., Young, H.C.: PIPE: A vlsi decoupled architecture. In: Proceedings of the 12th Annual International Symposium on Computer Architecture. (1985)
5. Jones, G.P., Topham, N.P.: A comparison of data prefetching on an access decoupled and superscalar machine. In: Proceedings of the 30th Annual International Symposium on Microarchitecture. (1997)
6. Kurian, L., Hulina, P.T., Coraor, L.D.: Memory latency effects in decoupled architectures. IEEE Transactions on Computers **43** (1994)
7. Smith, J.: Decoupled access/execute computer architecture. In: Proceedings of the 9th Annual International Symposium on Computer Architecture. (1982)
8. Tyson, G., Farrens, M., Pleszkun, A.: MISC: A multiple instruction stream computer. In: Proceedings of the 25th Annual International Symposium on Microarchitecture. (1992)
9. Wulf, W.A.: Evaluation of the WM architecture. In: Proceedings of the 19th Annual International Symposium on Computer Architecture. (1992)
10. Zhang, Y., Adams III, G.B.: Performance modeling and code partitioning for the DS architecture. In: Proceedings of the 25th Annual International Symposium on Computer Architecture. (1998)
11. Farkas, K.I., Chow, P., Jouppi, N.P., Vranesic, Z.: The multicluster architecture: Reducing cycle time through partitioning. In: Proceedings of the 30th Annual International Symposium on Microarchitecture. (1997)
12. Canal, R., Parcerisa, J.M., González, A.: Speculative data-driven multithreading. In: Proceedings of the 6th International Symposium on High Performance Computer Architecture. (2000)
13. Kemp, G.A., Franklin, M.: PEWs: A decentralized dynamic scheduler for ILP processing. In: Proceedings of the ICPP. (1996)
14. Krishnan, V., Torrellas, J.: A chip-multiprocessor architecture with speculative multithreading. IEEE Transactions on Computers **48** (1999)
15. Marcuello, P., González, A.: Clustered speculative multithreaded processors. In: Proceedings of the 13th International Conference on Supercomputing. (1999)
16. Ro, W.W., Gaudiot, J.L., Crago, S.P., Despain, A.M.: HiDISC: A decoupled architecture for data-intensive applications. In: Proceedings of the 17th IPDPS. (2003)
17. Bird, P., Rawsthorne, A., Topham, N.: The effectiveness of decoupling. (In: Proceedings of the 7th International Conference on Supercomputing)
18. Collins, J.D., Wang, H., Tullsen, D.M., Hughes, C., Lee, Y.F., Lavery, D., Shen, J.P.: Speculative precomputation: Long-range prefetching of delinquent loads. In: Proceedings of the 28th Annual International Symposium on Computer Architecture. (2001)
19. Roth, A., Sohi, G.S.: Speculative data-driven multithreading. In: Proceedings of the 7th International Symposium on High Performance Computer Architecture. (2001)