Speculative Execution of a Non-Blocking Multithreaded Architecture

Joseph M. Arul, Tsozen Yeh and HsuanYu Chen
Fu Jen Catholic University, Hsin Chuang 242, Taipei, Taiwan, R.O.C.
{arul,yeh,allen94 @csie.fju.edu.tw}

Abstract

In the modern architectures, if we want to increase the processor performance, we need to increase the ILP. TLP has been complemented to ILP in multithreaded architectures. In this paper, we present an evaluation of modern processor that decouples memory accesses to alleviate the gap, uses a non-blocking multithreaded together with the dataflow paradigm. We provide both clock cycles per instruction (CPI) and instructions per clock cycle (IPC) evaluation of a multithreaded architecture by using speculative execution. The existing architecture has been evaluated previously and shown that it has outperformed MIPS like architectures. In this particular study, we try to implement speculative execution of multithread on this unique architecture. Some of the benchmarks we used include I-structure that is unique to dataflow architecture and other benchmarks are without I-structure. All the benchmarks have shown speedup of about 1.3. In a speculative execution, it divides the thread aggressively and the mutual exclusion and dependence are guaranteed to be parallel. Thus it can increase the performance of any program with high probability. We have used different architectural simulators to prove the existing performance improvement of speculative execution.

1. Introduction

In the past decade, microprocessors have been improving their performance at a rate of 50-60% per year. It was mainly due to increase in clock rates and improvement in compiler technology and improvement in instruction throughput (IPC). Achieving this performance improvement in the future would be a tremendous challenge, since it is already facing technology-driven limitations. The researchers may not be able to sustain clock speed improvements with all the existing technologies on a single core. Due to this fact, there has been increasing interest in architecture concurrent processing by dual-core, multi-core with a support of hyper-threading technology.

Switching to multi-core or Chip-multiprocessors (CMP) systems need to be complemented to achieve higher levels of performance techniques, such as out-of-order execution, branch and value prediction and speculative instruction execution. Some computer architects have advocated a paradigm shift from high performance from high throughput using distributed components or dual-core. With this paradigm comes a renewed interest in multithreaded architecture. In the future, computer architect would focus on multithreaded architecture as opposed to single threaded architecture. In our research, we would like to present the speculative execution method and the results of the usage of speculative execution on a non-blocking multithreaded architecture. This architecture is unique, because it combines the advantages of control flow and dataflow system. This multithreaded scheduled dataflow (SDF) has been researched for a long time [1-4].

In the past, many have discontinued their research in dataflow architecture due to the complex hardware for communicating operands among instructions. In our architecture we still keep the instruction as dataflow model and the synchronization at the thread level using control-flow semantics. The detailed architecture will be presented in Section 4.

The speculative execution is an important method of Instruction Level Parallelism (ILP). In this research, we try to use the speculative execution in SDF architecture and explain how we implemented the speculative execution. We not only present the speculative execution of the SDF architecture but also the experiment results using few benchmarks on SDF.

Multithreading paradigm has been included in modern CPUs with the emergence of hyper-threading. Multithreaded architecture allows performance design tradeoffs that may not be available in single threaded machine. Most of the computers used are von Neumann architecture or control flow architecture. The speculative execution is also researched in control flow architecture. Hence, we try to use speculative execution in a non-blocking multithreaded architecture and observe the performance improvement over non-speculative execution. The main difference is that the control flow programs are partitioned into many procedures and executed based on the program counter, but the dataflow program has to be executed based on data driven and divided into many non-blocking threads in our SDF architecture. We extend the original thread, which is a non-blocking thread to support speculative thread level parallelism. How to run the loop iteration by non-blocking thread is a main problem since in the iterative processing, the execution must encounter a series of branch prediction condition. By using speculative execution we try to create many blocking threads and compare with the non-speculative execution. We will see how the data flows in many speculative threads of SDF architecture. However, in these architectures we need to extend few new instructions to support the speculative execution. No matter control flow or dataflow scheme they all need additional instructions and registers for speculative execution.

Here, we have used two different simulators with the same base engine for non-speculative and speculative SDF architecture in order to observe the number of instructions and clock cycles. Then, we evaluate the CPI (Clock cycles Per Instruction) and IPC (Instructions per Clock cycle). After running few benchmarks, each benchmarks show significant improvement of speculative execution over non-speculative execution.

The reminder of this paper is organized as follows. Section 2 presents the background about instruction level parallelism and related research on speculative execution for many fields. Section 3 introduces the concept and purpose of speculation in detail. Section 4 proposes the speculative execution in SDF architecture and compare with the non-speculative execution in SDF architecture. The experimental results of these two different schemes of execution in SDF architecture and its comparison using different applications will be presented in Section 5. Section 6 draws the conclusions by observing these two executed schemes.

2. Background and Related Research

In this Section, we will present an overview of all the related researches for the SDF architecture and speculative execution.

2.1 Background

Today, parallel systems apply to accelerate processor execution performance. In order to generate more efficient parallel language, excellent compilers need to produce optimizing scheduled assembly code. However, with a sequential application, the compiler is difficult to extract parallelism due to some of the limitations as follow:

- sequential program is hard to divide
- many dependencies will occur in pipeline stage
- need good algorithm for branch prediction
- must detect which instructions must be rearranged

In this situation, the speculation is an efficient approach that reorders instructions, moving a load instruction across a store instruction or an instruction across a conditional jump (i.e. branch). Sometimes speculative parallelism also called thread-level speculation (TLS) that assumes system can execute multi-loop level parallelism optimistically. It must be divided by converting thread-level parallelism (TLP) into instruction-level parallelism. Simultaneous multithreading (SMT) is that modern multiple issue processors often have more functional unit parallelism available than a single thread that can effectively use. In the SMT case, the thread-level parallelism and instruction-level parallelism are exploited simultaneously with multiple threads using the issue slots in a single clock cycle.

2.2 Related Research

Speculative parallelization is introduced in [6]. There are two topics that are introduced: compiler-based automatic parallelization and speculative parallelization. Most compilers focus on loop level parallelism and how to execute on different loop iterations simultaneously. Hence, to develop a parallel algorithm for the compiler is the major concept. The main advantage of speculative parallelization is that it can automatically parallelize loop of sequential program and does not need to know the dependence at compile time.

The compiler that can support for dynamic speculative pre-execution has been proposed in [7]. Various forms of speculative pre-execution have been developed, including hardware-based and software-based approaches. The hardware-based requires a complex implementation and lacks global information such as data flow and control flow. The other approach cannot deal with dynamic events and imposes additional software overhead. In this research, they enhance novel compiler to support for the dynamic pre-execution of a pre-fetching thread which contains the future probable cache miss instructions that can run on the spare hardware context for data pre-fetching.

The data speculative multithreaded architecture is mentioned in [8]. Their research presents a novel processor for micro-architectures, called Data Speculative Multithreaded Architecture (DaSM) that relieves three of the most important bottlenecks (data dependencies, a relatively small instruction window, and a limited fetch bandwidth respectively) of superscalar processor. DaSM does not modify the ordinary programs compiled for a superscalar processor implementation. Their processor implements an effective large instruction window that is made up of several non-adjacent small windows.

Each small window is built using the conventional control speculation approach whereas the creation of a new window is based on speculating on highly predictable branches. Their research is the combination of the data speculation and multiple threads of control in a promising alternative to relieve the most critical bottlenecks of current superscalar microprocessors.

Some researchers also have exploited the effect of speculative execution on cache performance [9]. The ideal method for examining the cache performance of a speculative processor is to generate memory reference traces with a full execution simulator and use them as input to a cache simulator. The main result of the study is that deep speculation causes a significant increase by usually less than 15%. In fact, by calculating the traffic ratio they found that cache efficiency actually increases as speculation increase.

By using speculative computation and parallelizing techniques to improve scheduling of control-based design is presented in [10]. They improve the already proposed ILP scheduling approaches by extending the case of speculative computation. This means that the standard techniques for high-level synthesis can be considered obsolete in certain number of new designs. To deal with this problem, the effectiveness of speculative code is transformed into mixed control and data flow design to reduce the length of the result schedules.

In the next chapter, we will present in detail about speculative execution in computer architecture of this particular research. How to apply speculative execution, pipeline execution and instruction set architecture on Scheduled Dataflow architecture will be presented in section 4.

3. Speculative Execution

In any program execution, there are part of the program that which should be executed, other part of the program which should not be executed and finally there are statements that cannot be proven to be in either of the two above mentioned. Speculative execution is that which part of the program cannot be proven either certainly to be run or not to be run.

The speculative execution is that part of the code to be run concurrently until it is proven that they are not needed. Speculative execution means performance optimization for the program by running certain part of the program concurrently or in parallel in a multithreaded architecture. It is useful by running early, which consumes less time and space. In this chapter we will discuss about all the speculative concepts and how it is used in the modern microprocessors.

3.1 The Speculative Execution and the Modern Microprocessors

In modern pipelined microprocessors, speculative execution is used to reduce the cost of conditional branch instructions. Branch condition may not be known until the branch is evaluated. Hence, branch prediction technique is used which is to guess the most likely to branch direction. If it is proven wrong, the executed part of the code is discarded. Those discarded instructions consume CPU cycles and power consumption in an embedded systems or laptop computers. Definitely for the miss-predicted branches there will be penalty. Modern microprocessors have conditional move instructions. These instructions move data if the condition is met. In this situation it eliminates branching.

The speculative execution means early execution and is often cheaper because the value needed for the computation or execution is brought in before. When a program starts executing an array of 10,000 we can think of bringing the data early knowing that the program would need all the data in an array. It can also be said as prediction. In the case of prediction, we try to predict what should be done and knowing what should be done, we apply the early known direction. We need to also design a strong prediction algorithm. In the case of speculative execution no prediction is made before hand. Eager evaluation is also a form of speculative evaluation. What is an eager evaluation? Figure 3-1 shows an example of eager evaluation.

```
x = (6 + 8) * (1+2^3)

printf("%d",x);

printf("%d",x+3);
```

Figure 3-1 Eager Evaluation

In the above situation, x can be evaluated early and stored as 126. Thus, we can reduce the storage and also print statement can be evaluated once instead of twice. This form of speculative evaluation reduces storage and number of calculations to one.

The speculative execution approach in a modern processor would exploit the increasing abundance of spare processing cycles to execute ahead of time certain instructions for applications that would otherwise stall on disk I/O. When an application needs some data and if that data may not be in memory, it needs to be fetched from disk stalling the CPU. Speculative execution approach uses these cycles to fetch the data needed in the future rather than stalling the CPU later on when similar situation arises.

Modern microprocessors use speculative execution to decrease the clock cycle cost of conditional jump instructions and more advanced processors add the branch prediction for control speculative execution.

```
If (i = = j)          ; condition

then A = B + C        ; statement 1

else D = E + F        ; statement 2
```

Figure 3-2 A Code for Control Condition

Figure 3-2 shows an example of conditional execution. In the above situation, ***if***, ***then***, and ***else*** are the branch instructions. The condition ***i = = j*** will decide which statement must be executed. An advanced processor speculatively executes both the statement1 and statement2 at the same time. Then, discard one of the statements that does not need. The other method is to 'guess' by using branch prediction that only executes one statement that likely to result. Sometimes, the processor must undo one statement if the processor finds that executed the wrong statement. The branch prediction schemes are usually implemented by hardware.

The VLIW (Very Long Instruction Word) and superscalar architecture are both static and dynamic technique for speculation. In a VLIW, the compiler wants to target a wide-issue processor so that the compiler would be necessary to develop a region scheduling technique by speculation such as trace scheduling. A superscalar processor executes one or more instructions that can issue in a single pipeline stage by using speculative execution to pre-fetching multiple instructions simultaneously. This processor still requires that the compiler to schedule instruction. Hence, the speculative execution is an efficient way of instruction scheduling for VLIW and superscalar architecture. No matter in VLIW or superscalar, the purpose of execution is to decrease the CPI (Clock cycles Per Instruction) and increase the IPC

(Instructions Per clock Cycle) by using speculative execution.

3.2 Speculative Execution on Multithreaded Processors

A dual core or multi-core microprocessor implements multiprocessing in a single physical package. The multithreading paradigm complements the multi-core to exploit instruction level parallelism. The goal of multithreading hardware support is to allow quick switching between a blocked thread and another thread ready to run. Speculation at this level in a multithreaded multi-core processor can overcome the limitations in dividing a single program into multiple threads. Thus, it can enhance performance through parallelization. A speculatively multithreaded multi-core processor can perform parallel execution of a conventional sequential program. If we use a non-speculative multithreaded program, it conservatively divides the program and its mutual independence and execution can be only guaranteed. With a speculative execution there would be a high probability of execution.

According to Sohi and Roth, where it is control driven or data driven, speculation can aggressively divide the program into multiple threads that can guarantee high probability of execution in a multithreaded processor [12]. The speculative multithreading model considers each program region into a separate thread that guarantees high degree of parallelism, whether it is control driven or data driven.

3.2.1 Non-speculative Control-driven VS. Speculative Control-driven

The programs are divided into control-driven threads via control flow boundaries. In control-driven multithreading, the thread executes the contiguous segments and reconstructs the sequential execution by dividing the dynamic instructions. Based on this situation, we have to find the division point in order to minimize inter-thread data dependencies.

In non-speculative control-driven multithreading, it must guarantee two special scenarios. There are execution-certainty and data-integrity of threads. In fact, the thread cannot be undone. In order to achieve execution-certainty and maximize concurrency, the non-speculative control-driven thread can only be forked for execution. The data-integrity must be noticed when the data are shared among threads. It means the thread accessing to memory location with other threads must be synchronized. These situations and dividing a program into non-speculative control-driven threads are relied on the programmer and compiler. The programmer must understand clearly the parallel algorithm for data-sharing and minimizing synchronization among different control-driven threads.

In the above problems, it can be solved by using speculation. In speculative control-driven multithreading, the threads reconstruct the correct total order of memory operations. In general, these works can be detected and recovered from inter-thread memory ordering violations for hardware support in speculative control-driven threads. With such support, the hardware can buffer or undo an entire control-driven thread, and change its architected state. The threads cannot be guaranteed at their final usefulness when the threads are spawned. Hence, the usefulness likelihood is high and the parallelism characteristics are more important.

3.2.2 Non-speculative Data-driven VS. Speculative Data-driven

The other model is dividing programs into data-driven multithreads via dataflow boundaries. The creation or execution of the data-driven thread is relied on loading data. In other words, the data-driven threads only need the inputs to trigger their execution. Sometimes, a data-driven thread is triggered by receiving previous data-driven thread and then sends the results to a next data-driven thread. In general, converting the imperative code to data-driven code can only be constructed for code written in data-driven language. The non-speculative data-driven threads have two major problems. First, we must ensure the programs can be divided into data-driven multithreads. Second, the programmer usually creates the sequential semantics so the resulting presentation will break for imperative programs.

The speculation is also to solve these problems. The data-driven threads usually need the additional assisted thread when run ahead or pre-execute. Hence, the purpose of speculation is to reduce the additional assisted thread in speculative data-driven threads. The data-driven multithread will be constructed via dataflow information, and spawned to pre-execute at some instructions that might cause problem in the future. It has the option of using the result directly or repeating the execution.

3.3 Speculative Execution on a Non-blocking Multithreaded Architecture

The gap between processing speeds and disk access time has been increasing every year. Memory sizes have been increasing rapidly, so too the application data requirements. We use a non-blocking multithreaded architecture and a decoupled architecture where a synchronization processor, which fetches the data from memory, and the execution processor execute the operands that are stored in the registers, are used to build the gap.

Instead of stalling Execution Processor (EP) for slow memory while fetching, Synchronization (SP) is used to fetch the data from memory. We do speculative execution by exploiting the knowledge and information that is available. In this approach we pre-fetch for virtual memory accesses as well as explicit I/O calls, enabling it to provide the benefit regardless of the I/O access methods used in any applications. Here we use the mechanism to estimate the impact of memory use by speculative execution on SDF system performance, and thereby controlling speculative execution when memory resources are not abundant.

In our research, we create the speculative thread differently from the normal thread. We also extend the instructions for speculative execution on SDF architecture. In general, in the loop iteration that usually uses the value from previous loop iteration. Due to this reason, the speculative execution on SDF architecture executes the speculative thread to ‘guess’ the result that will be used for the next iteration. The benefit of this is, it does not need additional threads.

The other factor that affects the execution time is the I-structure mechanism on speculative execution of SDF architecture. In modern architectures, arrays are used to store data. The feature of I-structure mechanism is that write once, read many times. Hence, here we use I-structure instead of array. We offer a new instruction and algorithm for pre-fetching from I-structure memory. In other words, we need to calculate the distance for pre-fetch in order to pre-execute on speculative execution. For example, if the input value is ‘N’ for iteration times, the distance is N/2. When input value is 10, the distance is 5 for pre-fetch. While the program is fetching data from position 2 of I-structure, it will pre-fetch data from position 7 of I-structure. In section 4, we will discuss in more detail about this extension of instruction set, speculative thread structure and iteration in speculative execution of SDF architecture.

In section 5, we present the algorithm to calculate the distance for pre-fetching I-structure memory and various other benchmarks results.

4. Non-blocking Multithreaded Architecture

In this section we will present the non-blocking multithreaded architecture that we have used for our implementation and experimentation. We will also explain how we implemented a speculative and non-speculative execution.

4.1 Scheduled Data Flow (SDF) Architecture Overview

Most of the computers used are von Neumann model where the program executes using control flow architecture. In the control flow scheme, a program will be partitioned into many procedures (or functions, modules, methods, etc.) Programs are executed based on the program counter. But in SDF architecture, a program has to be divided into many non-blocking threads. Each thread has several code blocks and is divided into three portions, which are pre-load, execute and post-store sequentially.

4.1.1 Non-blocking Thread Structure

The SDF architecture uses two processors, which are Synchronization Processor (SP) and Execution Processor (EP) to follow the precedent decoupled architecture's AP and EP [14]. Pre-load and Post-store portions of a thread can be executed by Synchronization Processor (SP). Execute portion of a thread can be executed by Execution Processor (EP). Figure 4-1 shows this structure. In other words, SDF architecture has three main components: Synchronization Processor (SP), Execution Processor (EP) and thread schedule unit. Each thread is represented by four continuations: FP, IP, RS, and SC. FP is the Frame Pointer that is responsible for inputting and storing values by a thread. IP means the Instruction Pointer, which points to the thread code. RS represents a dynamically allocated register context that is a Register Set. SC is Synchronization Count that is the number of inputs needed for a thread before it can be scheduled for execution.

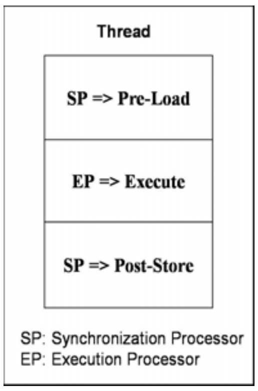

Figure 4-1 Thread Structure for SDF Architecture

When a thread receives its values, the synchronization count will be decremented until the count value becomes zero and the thread is scheduled on SP. The frame memory is allocated simultaneously while a thread is being created. The data needed for the thread is stored into the related frame memory. SP will pre-load data from frame memory into the thread's register context. This is the role of SP to execute thread's pre-load portion. SP is also responsible for a thread's post-store portion. When a thread terminates its job, if the result is needed for other threads, they will store the result into the related frame memory for other threads to load its data. The I/O instructions such as INPUT (read the value from the device) or OUTPUT (write the result to the device) is also done by SP in pre-loading portion (or post-store portion). The other processor is EP that is responsible for threads execution portion. The EP not only includes the arithmetic instructions (e.g. ADD, MUL) but also is responsible for frame allocation by using instruction such as FALLOC (allocates and initializes a frame memory for a thread). The SDF code uses RR instructions to store a pair of consecutive registers. For instance, ADD RR2 means that does ADD operation using registers R2 and R3. The compiler for SDF should divide any high level program into many threads. Since the compiler is not ready at the moment, we use assembly language to evaluate speculative and non-speculative threads.

4.1.2 Execution and Synchronization Pipeline

First, we describe the execution pipeline that has four units: instruction fetch, decode, execute, and write back (see Figure 4-2). Instruction fetch unit is similar to a normal fetch unit that behaves like program counter that points to the next instruction. Decode and register fetch units obtain a pair of registers that contain two source operands for the instruction. Execute unit executes the instruction and sends the results to write-back unit along with the destination register numbers. Write back unit writes two values to the register file.

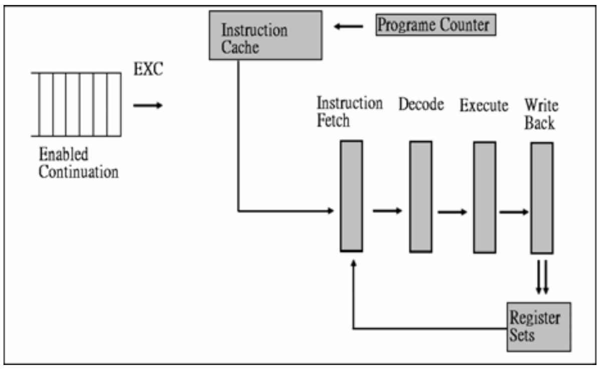

Figure 4-2 Execution Pipeline (EP)

Second, we continue to describe the other pipeline that is synchronization pipeline. The synchronization pipeline has five stages: instruction fetch, decode, memory access, effective address, and write-back (see Figure 4-3). As mentioned earlier, the synchronization pipeline handles pre-load and post-store instructions.

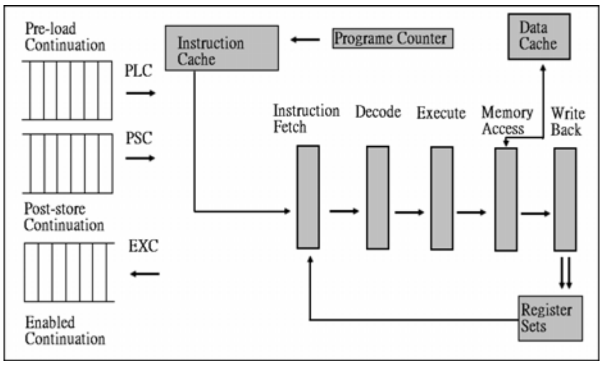

Figure 4-3 Synchronization Pipeline (SP)

The instruction fetch unit retrieves an instruction belonging to the current thread using program counter. The decode unit decodes the instruction and fetches registers. The effective address unit computes addresses for LOAD and STORE instructions. LOAD and STORE instructions only reference the frame memories of threads by using a Frame Pointer (FP) and an offset into the frames; both of which are contained in registers. The memory access unit completes LOAD and STORE instructions. Pursuant to a post-store, the synchronization count of a thread is decremented. The write-back unit completes LOAD (pre-load) and IFETCH instructions by storing the values in appropriate registers.

4.2 Non-speculative Execution in SDF Architecture

In this section, we will describe the original execution of SDF architecture and how to run the loop iteration by non-blocking threads. Figure 4-4 presents the original method for loop iteration execution. In the original method that is a non-speculative execution that needs two threads to procure single loop iteration. One thread’s function does the operation of loop iteration; the other thread is responsible to check loop condition.

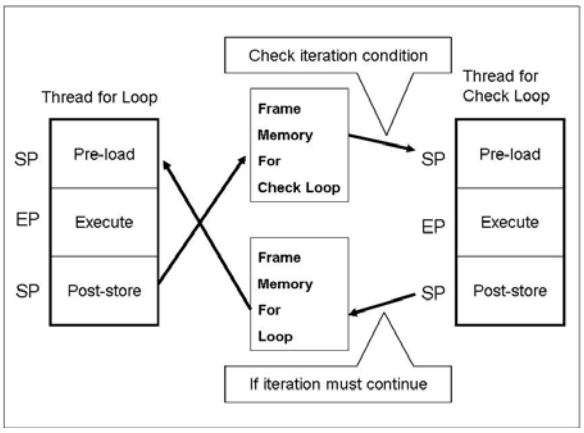

Figure 4-4 A Non-speculative Iteration Execution

We use the "FALLOC" instruction to allocate the frame memory for the specified thread checking condition. The thread uses the frame pointer and the offset to load or store data from frame memory. When the thread sends the data to the other thread, we use the "STORE" instruction to pass the data via frame memory. If the thread, which is responsible for checking the loop, decides whether the condition is valid, it will free the frame memory for loop. So we always use the "FFREE" instruction to release the thread and the frame memory. This is how the non-speculative thread execution is performed.

4.3 Speculative Execution in SDF Architecture

In this section, we will describe the speculative execution using SDF architecture and how to run the loop iteration by speculative threads.

4.3.1 Extension of Instruction Set and the Thread Structure

In order to speculate a program by using SDF code, we must add few new instructions. Speculative thread also must be modified differently from the non-speculative thread. Figure 4-5 shows the new thread structure for speculative mechanism, which we can compare with Figure 4-1.

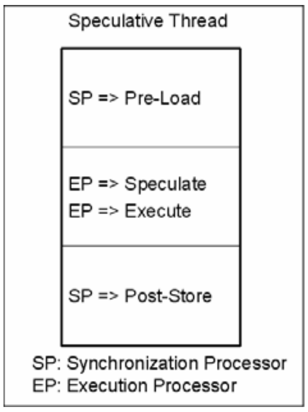

Figure 4-5 Speculative Thread

Speculative thread also consists of three parts that are 'pre-load', 'execute' and 'post-store'. In the SP phase, the speculative thread's function is same as the previous thread and it not only can fetch data from I-structure memory but also use 'SREAD' (by calculating the distance for speculative read) instruction to pre-fetch data from I-structure memory. In the 'execution' part, it can speculate an execution. By allocating a frame memory for speculative thread, it uses 'SFALLOC' (associate a speculative frame to a code-block) instruction to allocate speculative frame memory. We can use PUT (put value into any register) instruction instead of PUTR1 (put immediate data into register R1) instruction in order to solve the dependence.

4.3.2 Iteration in Speculative Execution

In the iterative processing, the execution must encounter a series of branch prediction condition. If we use branch instruction in EP phase, it will predict the result and use 'FORKEP' (Schedule the execution of code on Execution Processor) instruction executing EP again. The speculative thread does not wait to execute and choose in which SP must be executed. The benefit of this mechanism is that we don't need the other threads to check for the branching condition.

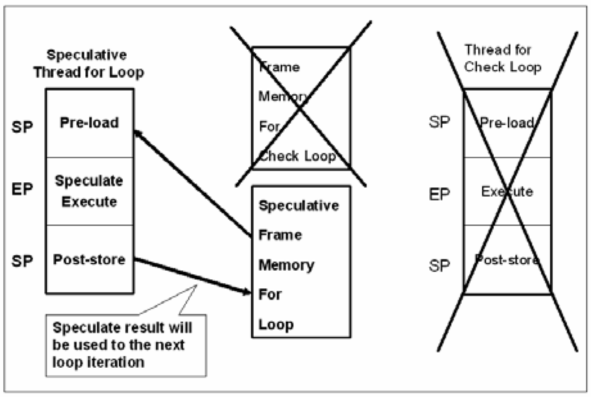
Speculative Thread for Loop
SP
Pre-load
EP
Speculate Execute
SP
Post-store
Speculate result will be used to the next loop iteration
Frame Memory For Check Loop
Speculative Frame Memory For Loop
Thread for Check Loop
SP
Pre-load
EP
Execute
SP
Post-store

Figure 4-6 A Speculative Iteration Execution

Figure 4-6 presents this method where we can compare with Figure 4-4. In the original method that needs the other thread that is responsible to check the loop condition. But in the loop iteration that usually uses the result from previous loop iteration. According to this attribute, the speculative thread 'guess' the result that will be used for the next iteration. The speculative thread allocates the result into the speculative frame memory for the next speculative thread pre-load. In the loop iteration, the thread usually will be executed early so that pre-execute can reduce the execution time.

Two flow paths are decided while speculative frame memory stores the data from speculative thread. One path is executed by 'predict correct' situation, then speculative thread is continued to execute. The other path is executed by 'predict incorrect' situation, which means the other loop condition that occurs must exit the loop iteration, then sends the data to other threads and free the current speculative thread. The branch prediction result will be known in EP phase of speculative thread. The benefit of this is that it does not need additional frame memory and the threads to check it.

In the next chapter, we will present seven programs that are used to implement the comparisons of these two schemes. According to the experimental results, we can note the better performance improvement by using the speculative execution in SDF.

5. Speculative VS. Non-speculative Benchmark Evaluation

5.1 Benchmarks used to Experiment

In this chapter we will present the experiment results. Some benchmarks are without using I-structure memory and other benchmarks are using I-structure memory. Simple benchmark such as running a loop where the program is entered 'n' and it adds 1 to n. Then the other benchmarks such as factorial program, Fibonacci program and prime number program are also presented. Using I-structure programs are Loop_IFETCH program which fetches data from the I-structure memory; matrix multiplication and linear search (add the data to be searched are stored in the I-structure). All the programs are written and tested by using speculative and non-speculative scheme. For these experiments we used two different simulators to estimate program's execution clock cycles and instructions in order to calculate CPI (Clock cycles per Instruction) and IPC (Instructions per Clock). One was an SDF simulator that uses a speculative execution. The other simulator uses a non-speculative execution.

The environment variables are set for the frame allocation policy, maximum number of frames, frame size and maximum number of register sets respectively as follows. (see Figure 5-1). The computer system and architecture lab of Fu Jen Catholic University department of Computer Science & Information Engineering maintains these simulators.

Operations	Description
-fpolicy cq | st | sh	Set frame allocation policy (cq=circular queue, st=stack, sh=stackh)
-frames NO_OF_FRAMES	Set maximum number of frames
-fsize WORDS	Set frame size
-regsets NO_OF_REGSETS	Set maximum number of register sets
Operations Setting	
FRAME SIZE = 256 FRAMES PER PROCESSOR = 256 REGISTER SETS PER PROCESSOR = 64 FRAME ALLOCATION POLICY = 0/1	

Figure 5-1 Environment Variables Setting

In the following paragraph we will present the method of execution for various benchmarks and results are presented. We will also present the IPC and the CPI for these benchmarks.

5.2 Programs without I-structure Usage

In this and the next section, we will present the experimental results that use seven different typical programs. These program comparison include summation, factorial calculation, Fibonacci and prime number. Table 5-1 presents the results of a simple summation using a single loop. The program uses different data size such as 1000 to 5000. The first column shows the data sizes from 1000 to 5000. Second column presents clock cycles for various data sizes. In the third column we present the total number of instructions used for various data size of the program. The fourth and the fifth column present the CPI and the IPC for non-speculative execution.

Table 5-1 Summation Program

Non-speculative				
Data size	**Clock Cycles**	**Instructions**	**CPI**	**IPC**
1000	156857	75023	2.091	0.478
2000	313641	150023	2.091	0.478
3000	470443	225023	2.091	0.478
4000	627227	300023	2.091	0.478
5000	784011	375023	2.091	0.478
Speculative				
Data size	**Clock Cycles**	**Instructions**	**CPI**	**IPC**
1000	36094	27037	1.335	0.749
2000	72094	54037	1.334	0.750
3000	108094	81037	1.334	0.750
4000	144094	108037	1.334	0.750
5000	180094	135037	1.334	0.750

The CPI for the non-speculative execution is consistent for various data size, which is 2.091. The IPC is also consistent for various data size, which is 0.478. Similarly, the same program was written for speculative execution. The result decreases the CPI from 2.091 to 1.334 and increases the IPC from 0.478 to 0.749. The speculative and non-speculative execution for the sake of simplicity, we keep the cycle count as 1 cycle for all the instructions irrespective of various opcodes. In a real architecture environment the cycles may very depending on the type of opcodes. In some environment it could take 50 to 100 cycles for opcodes such as multiply instructions.

Table 5-2 shows the results of factorial program. The factorial program uses a single loop and also invokes computational data sizes from 5 to 25 for speculative execution and non-speculative execution. We use a non-recursive version of the factorial program calculation.

Table 5-2 Factorial Calculation Program

Non-speculative				
Data size	**Clock Cycles**	**Instructions**	**CPI**	**IPC**
5	840	398	2.111	0.474
10	1625	773	2.102	0.476
15	2410	1148	2.099	0.476
20	3195	1523	2.098	0.477
25	3980	1898	2.097	0.477
Speculative				
Data size	**Clock Cycles**	**Instructions**	**CPI**	**IPC**
5	273	171	1.596	0.626
10	453	306	1.480	0.675
15	633	441	1.435	0.697
20	813	576	1.411	0.708
25	993	711	1.397	0.716

Table 5-2 presents non-speculative execution, where we can find the CPI is about 2.1 and the IPC is about 0.477. It shows not only very significant improvement for various data sizes but also great improvement in clock cycles. In speculative execution, as the previous program we transfer speculative instructions to execute in this program. We can observe that the CPI and IPC here too. The CPI is about 1.4 and the IPC is about 0.7. It shows more significant improvement for non-speculative execution.

Table 5-3 Fibonacci Program

Non-speculative				
Data size	**Clock Cycles**	**Instructions**	**CPI**	**IPC**
5	520	274	1.898	0.527
15	1660	894	1.857	0.539
25	2800	1514	1.849	0.541
35	3940	2134	1.846	0.542
45	5080	2754	1.845	0.542
Speculative				
Data size	**Clock Cycles**	**Instructions**	**CPI**	**IPC**
5	222	138	1.609	0.622
15	622	448	1.388	0.720
25	1022	758	1.348	0.742
35	1422	1068	1.331	0.751
45	1822	1378	1.322	0.756

In our experimental Fibonacci program, we write Fibonacci assembly program by using non-recursive method because recursive method will use additional registers to cause not enough register situation, but also frame memory. In dataflow architecture the recursive programs may not be suitable. In a recursive program one thread spawn another thread and so on. Thus, there will not be many parallelism to exploit implicit parallelism by having multithreaded architecture. Table 5-3 presents the results of this program. The CPI is about 1.85 and the IPC is roughly about 0.54 in non-speculative execution. But in speculative execution, the CPI is reduced from 1.85 to 1.34 and the IPC increased from 0.54 to 0.75. We can see the similar result for the factorial program too.

Table 5-4 presents the results for prime number program, which shows IPC over 1. This program is a simple prime number search program, where the user enters n, and the program finds all the prime number between 1 and n. The CPI is about 1.31 and the IPC is about 0.76 in non-speculative execution. The values are stable even for different data sizes. In speculative execution, the CPI is 0.807 and the IPC is 1.239 for different data sizes. Table 5-4 shows similarly for various data sizes, the IPC values we can see the improvement.

Table 5-4 Prime Number Program

Non-speculative				
Data size	**Clock Cycles**	**Instructions**	**CPI**	**IPC**
100	81361	61594	1.321	0.757
200	303627	231027	1.314	0.761
300	588279	446995	1.316	0.760
400	976091	741663	1.316	0.760
500	1498501	1138104	1.317	0.759
Speculative				
Data size	**Clock Cycles**	**Instructions**	**CPI**	**IPC**
100	28087	34518	0.814	1.229
200	100871	125051	0.807	1.240
300	193535	239707	0.807	1.239
400	318973	395287	0.807	1.239
500	487223	603804	0.807	1.239

5.3 Programs with I-Structure Usage

The I-structure is the feature of data flow architecture to fetch/store data and it has the characteristic that write once, read many times. I-structure affects the execution time where the data is read from memory and stores the data for the first time in memory. Any memory read can be done many times, but write could lead to further data hazards. Since I-structure usually causes data hazard, it can be a bottleneck in the dataflow architectures. The possible data hazards are RAW (read after write), WAW (write after write) and WAR (write after read). Note that RAR (read after read) case is not a hazard. In the following, three programs use I-structure for these comparisons are Loop_IFETCH, matrix multiplication and sequential search algorithm respectively. In a common architecture we use arrays as a memory to store data. Here we use I-structure instead of array. Since this architecture uses a dataflow like instruction execution we use I-structure instead of an array.

First is the Loop_IFETCH program. This program uses a single loop and I-structure to store the data. Actually, the Loop_IFETCH is designed program that's function produces a series of fetching notion at the same memory location. It will cause consecutive RAW and WAR. In other words, the Loop_IFETCH program is similar to summation program. Initially we write all the data in I-structure then read and operate the data from I-structure memory. Following this model we also use pre-fetching mechanism by using SREAD instruction. Hence, SREAD is introduced as a speculative read operation. By using this method we need to calculate the distance for pre-fetch. In this case, the distance is calculated by dividing iteration number of times. For example, if we input "N" for iteration times, the distance is N/2. When input value is 10, the distance value is 5 for pre-fetch. While the program is fetching data from position 1 of I-structure, it will pre-fetch data from position 6 of I-structure. Figure 5-2 presents this concept by using high-level language.

```
Add 1 to N by using speculative
input n and only run n/2 rounds
   for(i=1;i<n/2;i++)
   {
      a=hash[f[i]];
      a=a + hash[f[i+n/2]];
      hash[0]+=a;
   }
```

Figure 5-2 Loop_IFETCH Program Described Using C language

Table 5-5 Summation Program Using I-Fetch

Non-speculative				
Data size	**Clock Cycles**	**Instructions**	**CPI**	**IPC**
200	16094	12026	1.338	0.747
400	32094	24026	1.336	0.749
600	48094	36026	1.335	0.749
800	64094	48026	1.335	0.749
1000	80094	60026	1.334	0.749
Speculative				
Data size	**Clock Cycles**	**Instructions**	**CPI**	**IPC**
200	3764	3607	1.044	0.958
400	7464	7207	1.036	0.966
600	11164	10807	1.033	0.968
800	14864	14407	1.032	0.969
1000	18564	18007	1.031	0.970

From table 5-5 we can observe that the CPI is about 1.33 and the IPC is about 0.479 for non-speculative execution. Whereas the speculative execution, the CPI is about 1.03 and the IPC is about 0.96. No matter how much the data size is increased, the execution is very stable in speculative or non-speculative scheme. In spite of the high number of fetches, the CPI is very low in speculative execution. It again verifies that the speculative execution mechanism presents an excellent paradigm to obtain a better execution performance as compared to non-speculative execution mechanism.

The second program is matrix multiplication. Table 5-6 presents the results of running matrix multiplication using different sizes of data. This program uses three nested loops and also uses I-structure applying speculative and non-speculative mechanisms. Matrix multiplication program performs several calculation in the inner most loop.

A[ ] and B[ ] are stored in the I-structure memory. Matrix multiplication needs an awful lot of calculation for addition and multiplication and I-structure simultaneously. In this program, we use two SREAD instructions for pre-fetching. The distance is divided in the third loop. Figure 5-3 shows this method in the high level language as well as in a simple pseudo code to apply speculative execution.

Non-speculative Execution Program	**Speculative Execution Program**
for(i=0;i<N;i++)	for(i=0;i<N;i++)
for(j=0;j<N;j++)	for(j=0;j<N;j++)
for(k=0;k<N;k++)	for(k=0;k<N/2;k++)
{	{
C[i][j] += A[i][k] * B[k][j];	a = A[i][k] * B[k][j];
}	b= A[i][k+D] * B[k+D][j];
	C[i][j] += a+b;
	}
Example: 4 x 4	**Example: 4 x 4**
C[1][1] + = A[1][0] x B[0][1]	C[1][1] + = A[1][0] x B[0][1]
C[1][1] + = A[1][1] x B[1][1]	+ A[1][2] x B[2][1]
C[1][1] + = A[1][2] x B[2][1]	C[1][1] + = A[1][1] x B[1][1]
C[1][1] + = A[1][3] x B[3][1]	+ A[1][3] x B[3][1]

Figure 5-3 Matrix Multiplication Described Using C language

The matrix multiplication program experimental results are similar to the first program, summation program, where the SDF program has to access I-structure memory. Table 5-6 presents the CPI, which is about 1.6, and the IPC is about 1.2 for non-speculative execution. This shows the situation where the value is very stable for matrix multiplication program. In speculative execution, the CPI is about 1.2 and the IPC is about 0.8, these values are also similar for matrix multiplication program.

After running for various data sizes from 2 * 2 to 32 * 32, the performance improvement is about 1.2 in all cases.

Table 5-6 Matrix Multiplication

Non-speculative				
Data size	**Clock Cycles**	**Instructions**	**CPI**	**IPC**
2x2	2283	1383	1.651	0.606
4x4	8587	5237	1.640	0.610
8x8	32673	20409	1.601	0.625
16x16	127423	80609	1.581	0.633
32x32	518859	320433	1.619	0.618
Speculative				
Data size	**Clock Cycles**	**Instructions**	**CPI**	**IPC**
2x2	984	755	1.303	0.767
4x4	3324	2643	1.258	0.795
8x8	12372	9947	1.244	0.804
16x16	47940	38667	1.240	0.807
32x32	191376	152555	1.254	0.797

Final program is a linear search. Linear search program also uses a single loop and I-structure to store the data. The process of searching data must be considered for this program, like EQ or NE opcodes, in order to execute correctly for the given program. Because the speculative execution applies branch prediction and dynamic scheduling method, process of pipeline execution will reorder some instructions.

Table 5-7 Linear Search

Non-speculative				
Data size	**Clock Cycles**	**Instructions**	**CPI**	**IPC**
500	60998	32524	1.875	0.533
1000	121926	65024	1.875	0.533
1500	182854	97524	1.875	0.533
2000	243782	130024	1.875	0.533
2500	304710	162524	1.875	0.533
Speculative				
Data size	**Clock Cycles**	**Instructions**	**CPI**	**IPC**
500	16061	11513	1.395	0.717
1000	32061	23013	1.393	0.718
1500	48061	34513	1.393	0.718
2000	64061	46013	1.392	0.718
2500	80061	57513	1.392	0.718

Table 5-7 presents the results of linear search. The CPI for the non-speculative execution is consistent for various data sizes. The IPC is also consistent for various data sizes, which are about 0.533. Similarly, the same program was written for speculative execution. The CPI is 1.39 and the IPC is 0.718.

5.4 Analysis of the CPI Improvement

In this section, we extract the CPI from all the experiment of program in order to understand the execution performance clearly. Since the CPI is a common measurement unit, many computer architectures always use this method for presenting the improvement between various architectures.

Table 5-8 Average CPI without I-structure

Non-speculative				
Sum	**Factorial**	**Fibonacci**	**Prime**	**Average**
2.091	2.111	1.898	1.321	1.855
2.091	2.102	1.857	1.314	1.841
2.091	2.099	1.849	1.316	1.839
2.091	2.098	1.846	1.316	1.838
2.091	2.097	1.845	1.317	1.837
Speculative				
Sum	**Factorial**	**Fibonacci**	**Prime**	**Average**
1.335	1.596	1.609	0.814	1.338
1.334	1.480	1.388	0.807	1.252
1.334	1.435	1.348	0.807	1.231
1.334	1.411	1.331	0.807	1.221
1.334	1.397	1.322	0.807	1.215

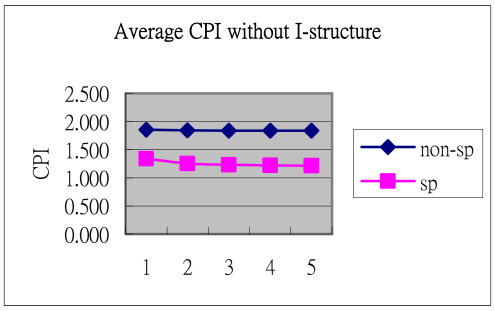

Figure 5-4 IPC Graph Using Table 5-8

Table 5-8 presents the average CPI without I-structure usage. First column presents the CPI for summation program, second column presents the factorial program, third column presents Fibonacci program, fourth column presents prime number program, and final column shows the average for non-speculative execution. The average CPI is about 1.8. For speculative execution part, the average CPI is about 1.2 better than non-speculative execution.

Table 5-9 also presents the average CPI with I-structure usage. First column presents summation program, second column presents matrix multiplication program, third column presents linear search program, and the final column is average CPI using non-speculative execution. The average CPI is about 1.6. Using speculative execution part, the average CPI is about 1.2 better than non-speculative execution.

Table 5-9 Average CPI with I-structure

Non-speculative			
IFETCH	**Matrix**	**Search**	**Average**
1.338	1.651	1.875	1.621
1.336	1.640	1.875	1.617
1.335	1.601	1.875	1.604
1.335	1.581	1.875	1.597
1.334	1.619	1.875	1.609

Speculative			
IFETCH	**Matrix**	**Search**	**Average**
1.044	1.303	1.395	1.247
1.036	1.258	1.393	1.229
1.033	1.244	1.393	1.223
1.032	1.240	1.392	1.221
1.031	1.254	1.392	1.226

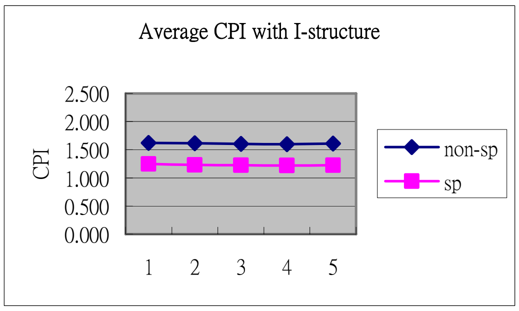

Figure 5-5. IPC Graph Using Table 5-9

According to our experimental results no matter whether we use or not use I-structure mechanism, the speculative execution has better performance than the non-speculative execution. Speculative execution is a form of prediction where it encounters a loop and the fetching of data, it can assume that the data needed can be fetched ahead of time and save many cycles. As a result, the CPI as well as IPC is improved.

6. Conclusions

The main goal of this research is to provide a qualitative evaluation of a unique non-blocking multithreaded architecture that clearly decouples all memory access from the execution pipeline. Even though previously reported that this architecture achieves scalable performance that is comparable to simple scalar architecture in this research, we show the speculative execution vs. non-speculative execution of multithreaded architecture.

We can exploit ILP by using SP and EP concurrently and TLP by using multithreading technique simultaneously. The hardware used in SDF is much simpler. It is also possible to build several processors on a single chip to further speedup the execution. There are several projects for implementing multiple processors on the same chip as an effective use of extra chip area.

By using speculative execution of SDF architecture it has better performance than non-speculative execution of SDF architecture. We can show that the speculation is a method that has branch prediction in dynamic scheduling. The speculative execution is a ‘guess’ behavior that wants to reduce data hazard and control hazard stalls at the same time. In other words, the purpose of a speculative execution is that the branch could be executed earlier for solving dependences and hazards.

In the original SDF implementation, the non-speculative execution needs two threads to procure single loop iteration. One thread does the operation of loop iteration and the other thread is responsible for checking loop condition. In the speculative execution, the speculative thread guesses the result to execute the next iteration.According to our experimental results that are presented show that the speculative execution has better performance than the non-speculative execution no matter we use or do not I-structure memory mechanism.

We have used few benchmarks to show that the speculative execution is an efficient way of instruction-level parallelism in the multithreaded scheduled dataflow architecture. Not only using multithread improves performance of a program but enhance more than the average program performance by using speculative execution.

Reference

[1] K.M. Kavi, H.S. Kim, and A.R. Hurson, “Scheduled Dataflow Architecture: A Synchronous Execution Paradigm for Dataflow,” IASTED J. and Applications.Vol.21, no. 3, pp.114-124, Oct. 1999.

[2] K.M. Kavi, J.M. Arul and R.M. Giorgi, “Execution and Cache Performance of the Scheduled Dataflow Architecture,” J.Universal Computer Science, special issue on multithreaded and chip multiprocessors, vol. 6, no.10, pp.948-967, Oct. 2000.

[3] K.M. Kavi, R.M. Giorgi, J.M. Arul, “Scheduled Dataflow: Execution Paradigm, Architecture, and Performance Evaluation,” IEEE Trans. On Computers, Vol. 50, No. 8, pp.834-846, August 2001.

[4] J.M. Arul, Tsozeh Yeh, Chiacheng Hsu, Janjr Li, “ An Efficient Way of Passing of Data in a Multithreaded Scheduled Dataflow Architecture,” Proc. 8th International Conference on High-Performance Computing in Asia-Pacific Region, pp.487-492, Dec. 2005.

[5] Huiyang Zhou, Chao-Ying Fu, Eric Rotenberg, Thomas M. Conte,“A Study of Value Speculative Execution and Misspeculation Recovery in Superscalar Microprocessors,” Department of Electric & Computer Engineering, North Carolina State University, pp.--23.

[6] Arturo González-Escribano, Diego R. Llanos. “Speculative Parallelization,” ISSN 0018-9162, IEEE Press On Computer, vol. 39, no. 12, pp. 126-128, December 2006.

[7] Won W. Ro, Jean-Luc Gaudiot, “Compiler Support for Dynamic Speculative Pre-Execution,” Proceedings

of the Seventh Workshop on Interaction between Compilers and Computer Architectures, p.14, February 08-08, 2003.

[8] Marcuello, Pedro Antonio Gonzalez, "Data Speculative Multithreaded Architecture," 24th Euromicro Conference Proceedings, IEEE 1998, vol. 1, pp. 321-324.

[9] Jim Pierce, Trevor N. Mudge,"The Effect of Speculative Execution on Cache Performance," Proceedings of the 8th International Symposium on Parallel Processing, p.172-179, April 01, 1994.

[10] Roberto Cordone, Fabrizio Ferrandi, Gianluca Palermo, Marco Domenico Santambrogio, Donatella Sciuto, "Using Speculative Computation and Parallelizing Techniques to Improve Scheduling of Control based Designs," The 11th Asia and South Pacific Design Automation Conference Technical Program, IEEE 2006, pp 898-904.

[11] F. Chang and G. A. Gibson, "Automatic I/O Hint Generation Through Speculative Execution," Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, LA, February 1999, pp. 1-14.

[12] G. Sohi and A. Roth, " Speculative Multithreaded Processors," IEEE Computer 34, 4 (2001), 66-73.

[13] V. Krishnan and J. Torrellas, "Chip-Multiprocessor Architecture with Speculative Multithreading," IEEE Trans. Computers, vol. 48, no.9, pp.886-880, Sept.1999.

[14] J.E SMITH, "Decoupled access/execute computer architectures," Proceedings of the 9th annual symposium on Computer Architecture, p.112- 119, April, 1982.

[15] Gonzales, J., and Gonzalez, A., "Speculative Execution via Address Prediction and Data Prefetching," Proc. of International Conference on Supercomputing 1997, ACM, NY, pp. 196-203 (1997).

[16] Pedro Marcuello , Jordi Tubella , Antonio González, "Value prediction for speculative multithreaded architectures," Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture, p.230-236, November 16-18, 1999, Haifa, Israel.