Confidence Estimation for Register Value Communication in Speculative Multithreaded Architectures

Mohamed M. Zahran, Manoj Franklin, and Renju Thomas
Department of Electrical and Computer Engineering
University of Maryland, College Park, MD, 20742
{mzahran, manoj, renju}@eng.umd.edu

Abstract

There is a growing interest in the use of speculative multithreading to speed up the execution of programs. In this execution model, threads are extracted from a sequential program and are speculatively executed in parallel, without violating sequential program semantics. Speculative multithreading is appealing because it provides the power of parallel processing to speed up ordinary applications, which are typically written as sequential programs. The most common microarchitecture design of a speculative multithreaded processor consists of a group of processing elements connected as a unidirectional ring. In this microarchitecture, the data values have to be communicated from each processing element to its successor. Memory values are communicated using shared memory. However, register values have to be communicated through the unidirectional ring; when to communicate the register values and what values to communicate has to be decided. Communicating the register values every time they are modified requires a huge amount of bandwidth, but not all of the values communicated carry useful information. On the other hand, by communicating the values at the physical commit time of each thread, successor threads may be executing using old values. Hence, we need a smart mechanism that forwards in a timely manner only those values that are likely to be correct.

In this paper we propose a confidence estimator for each register value. Whenever any register value's confidence goes beyond a threshold, it is sent to the successor processing element. As this is a hardware solution, no compiler support is required, alleviating the need for re-compilation. Detailed simulation results show an improvement of 11.4% in instruction per cycle as compared to sending values only at thread physical commit. The proposed scheme has an average slowdown of 2% in instruction per cycle as compared to the other extreme scheme of sending all the values every cycle.

1 Introduction

There has been a growing interest in the use of speculative multithreading (SpMT) to speed up the execution of a single program [1] [5] [7] [9] [10] [11] [12]. The compiler or the hardware extracts threads from a sequential program, and the hardware executes multiple threads in parallel, most likely with the help of multiple processing elements. Whereas a single-threaded processor can only extract parallelism from a group of adjacent instructions that fit in a dynamic scheduler, a speculative multithreaded processor can extract parallelism from multiple, non-adjacent, regions of a program's execution.

Decentralization of critical resources is needed in order to avoid sophisticated circuits, leading to longer cycle time. Multiscalar is a main example of decentralized speculative multithreaded architecture [5]. Processing elements (PEs) in multiscalar are organized as a unidirectional ring. Program is partitioned into threads, that are assigned to PEs for parallel execution. Threads can be built using the compiler [4] or the hardware [10]. Data values flow from the head thread down to the tail. Data values are divided as memory and register values. Memory values are communicated through the shared memory. Register values are communicated through the unidirectional ring.

In order to get the best performance from these decentralized execution models, register values must be communicated in an efficient way. Inefficient communication may lead to execution delay or a lot of re-executions, which may affect performance severely. If a thread communicates its register values only at its physical commit time, successor threads will most often start execution based on wrong values. On the other hand, if we send all register values every cycle, a huge bandwidth will be required. In the current state-of-the-art chips, bandwidth dominates energy consumption. Furthermore, if we try to send only modified registers in each PE in every cycle, then we need a fairly complicated interconnection and register comparison logic in each PE, which will increase energy consumption and may affect cycle time. Moreover, a lot of re-execution may take place due to sending a correct value, followed by a wrong value then again the same correct value for the same register. Hence a compromise needs to be found between frequency of sending values and technological aspects.

In this paper, an efficient way, that depends on hardware, for communicating register values is presented. Confidence values will be assigned to register values in each PE. Whenever a register value's confidence estimation goes beyond a specific threshold, the value is sent to the successor PE. Our experiments show that less than half of the registers need to be sent during the life time of a thread. Detailed simulation results show an increase of 11.4% in the instruction per cycle (IPC) as compared to sending the values only during physical commit.

This paper is organized as follows. Section 2 presents some background information about the SpMT execution model and microarchitecture. Section 3 presents the proposed scheme for adding confidence estimation to register values and discusses the implementation issues. Section 4 presents the experimental evaluation of the proposed scheme. Section 5 presents a brief discussion of some related work. Finally, section 6 presents the conclusions.

2 The Speculative Multithreaded Execution Model

In this section the SpMT execution model is presented. First the thread model is shown, followed by the microarchitecture. The thread model specifies the sequencing of threads, the name spaces (such as registers and memory addresses) used for inter-thread communication, and the ordering semantics among distinct threads.

2.1 Multithreading Thread Model

The idea behind multithreading is to have multiple flows of control within a process, allowing parts of the process to be executed in parallel. In the *parallel threads* model, threads that execute in parallel are control-independent, Under this model, compilers and programmers have had little success in parallelizing most of the non-numeric applications. For such applications, a different thread control flow model called **sequential threads** model has been proposed. In that model, threads are extracted from sequential code and run in parallel, without violating the sequential program semantics. Inter-thread communication between two threads will be strictly in one direction, as dictated by the sequential thread ordering. No explicit synchronization operations are necessary. This relaxation makes it possible to "parallelize" non-numeric applications into threads, even if there is a potential inter-thread data dependence.

Examples of prior proposals using sequential threads are the multiscalar model [5][11], the superthreading model [3], the trace processing model [10], and the dynamic multithreading model [1]. In the sequential threads model, threads can be *non-speculative* or *speculative* from the control point of view. If a model supports speculative threads, then it is called speculative multithreading (SpMT). This model is particularly useful to deal with the complex control flow present in typical non-numeric programs. In fact, many of the prior proposals using sequential threads implement SpMT [3][5][8][9][10][11].

In this paper we are targeting SpMT models, because register values communication is a major issue for its performance.

2.2 Speculative Multithreaded Processors Microarchitecture

Most the speculative multithreading architectures proposed so far use a single level of multithreading. The program is partitioned into threads and multiple threads are run in parallel using multiple PEs. The PEs are usually organized as a circular queue in order to maintain sequential thread ordering, as indicated in Figure 1. These execution models use multiple register files in order to keep several versions of each register [2][13].

2.3 Register File in Multithreaded Architectures

When threads do not share a common register space, it is straightforward to implement the register file (RF) at the microarchitectural level —each

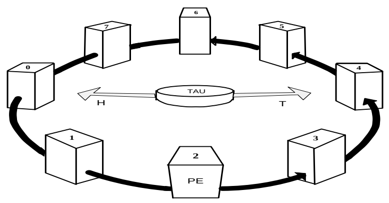

Figure 1: Circular Queue Arrangement of PEs in a Multiscalar Processor

PE can have its own register file, hence providing fast register access. Even when threads share a common register space at the ISA level, it is useful that at the microarchitectural level we still provide a separate register file in each PE to support fast register access, as a centralized register cannot provide a 1-cycle multi-port access time with today's high clock rates. There are two ways to achieve this decentralization, both of which provide faster register access times due to physical proximity and fewer access ports per physical register file.

- *RF Partitioning:* In this approach, each physical register file implements an independent set of ISA-visible registers. Notice that when a PE needs a register value stored in a non-local register file, the value is fetched through an interconnection network that interconnects the PEs.

- *RF Replication:* With the replication scheme, a physical copy of the register file is kept in each PE, so that each PE has a local copy of the shared set register space. These register file replica maintain different *versions* of the register space; i.e., the multiple copies of the register file store register values that correspond to the processor state at different points in a sequential execution of the program. In general, replication avoids unnecessary communication. However, if not done carefully, it might increase communication by replicating data that is not used in the future. A multithreaded processor that uses the replication scheme is the multiscalar processor [5].

The microarchitectures studied in this paper are using RF replication as it is the most common used among SpMT. Thus there are several versions of a register value, each corresponding to the register value at a different point in the sequential execution.

3 Confidence Estimation for Register Value Communication

Previous work [13] has presented some strategies for communicating register values for multiscalar processors, which depend on the compiler. In this section we present a method that depends totally on the hardware.

The main reason that makes register value communication a main factor for performance, is that non-head threads are executed in parallel with the head thread, and they need register values to begin executing. If there is no data value prediction, then these threads will not find useful values to work with, unless we have an efficient mechanism to forward information from the predecessor threads to the successor threads. The main issues in coming up with an efficient mechanism are:

- When are register values sent from a thread to its successor?

- How can the recipient thread be confident of the values received?

- What values should be sent from a thread to its successor?

The first issue deals with the timing and frequency at which we send register values. When a thread is assigned to a PE, it begins to use the current values of the registers of the predecessor. But these values are likely to change during the execution of the predecessor thread. If we re-send the values at the commit time of the predecessor, then we risk making the thread work with obsolete values, and a lot of re-execution is likely to happen, which may severely affect the overall performance. Also, if we send the values every time there is a change, then we suffer from two things: (i) Huge bandwidth requirements; (ii) A register value is likely to change several times during the lifetime of a thread, so if we send the value every time there is a change then a lot of useless re-executions are likely to be encountered. In this paper we are depending solely on hardware; the hardware cannot tell when the last value of a register will be produced in the predecessor thread. Consequently it is better to use a heuristic here.

The second issue is related to whether the recipient thread can confidently use the value received. We need to be sure that the values sent to PE have high probability of being correct.

The last issue is concerned with the bandwidth required to send values. We want to limit the number of register values sent as well as the number of times the values are sent during the lifetime of a thread.

We propose adding a confidence estimator in each PE to generate confidence values. Each register value received will be accompanied by a confidence value. If the confidence value is below some threshold then the value is not sent to the successor PE. In this way we are limiting the number of reexecutions that may happen due to a register value changes several times per thread lifetime. The confidence of the values coming from the predecessor depends on the following factors:

- *Distance from the header*: the confidence decreases with the distance of the predecessor from the head PE. This is because the head PE is executing the only non-speculative thread. As we go farther from the head PE, the amount of speculative data as well the probability of a register value to change increase, hence the confidence decreases.

- *Timing*: the confidence increases with the number of cycles that have elapsed since the assignment of the thread to the PE. As we reach the end of the lifetime of a thread, the probability of a register value to change decreases, hence the confidence increases.

- *Update mask*: we keep a register mask for each previously executed thread. The update mask is a string of bits, each of which corresponds to a register. When a bit is set, it means the register has been modified the last time the thread has been executed. We are assuming that if a thread modifies a register, it is likely to do so in its next execution, hence the confidence of this register value during the next execution of the thread decreases. The mask is associated with each static thread (each static thread is characterized by its starting PC). However, the mask is modified each time its corresponding thread is executed.

The above three factors are combined to generate a single value for each register value. The confidence value is generated by subtracting a weighted sum of the above three factors from the maximum confidence value. The weights have been obtained through many experiments and adjustments, and are shown in section 4.

Finally, it is to be noted that when a supertask is committing, it is sending *all* its altered register values for the last time and the confidence is maximum.

3.1 Implementation Issues

When a non-head thread is assigned to a PE, it receives all register values from the predecessor thread, to be able to start executing. After that and through the lifetime of the thread, the confidence estimator generates confidence values for each register in each PE except the head. These values are stored as extra fields in the register file within each PE.

The part of the confidence that is related to the distance from the head, as well as the part related to the update mask are calculated only once. Then, whenever a register is modified, the timing factor is calculated and the confidence estimation is generated for the modified register. It is to be noted that very few registers get modified per cycle, hence few hardware is needed to calculate the confidence estimation for them in one cycle. Whenever the confidence value goes beyond a specified threshold, the value is sent in the next cycle to the successor thread. It is to be noted that the confidence estimator does not calculate any confidence for a register unless two conditions are satisfied: (i)The successor PE must not be idle, and (ii)the register must have been modified. Hence, as soon as a register is modified and the successor PE is not idle, a confidence is calculated, the update mask bit is set, and the comparison is done. Two special registers are used to hold the update mask. One of them holds the old update mask, that has been generated from a previous execution of the thread, and this is the mask used to calculate the confidence estimation. The other contains the update mask of the current execution, it replaces the old one in case of physical commit of the thread and is discarded in case of thread squash. One important aspect of the update mask is that it is speculative in nature. For instance, when an instruction modifies a register and then got squashed, then the conservative approach is to keep the bit set in the mask because another instruction may have modified it. The aggressive approach is to reset the bit. We can keep track of which instructions modified which registers and add a register count for each bit, but this will be too much complication and hardware with not enough reward. Hence we opt for the conservative approach.

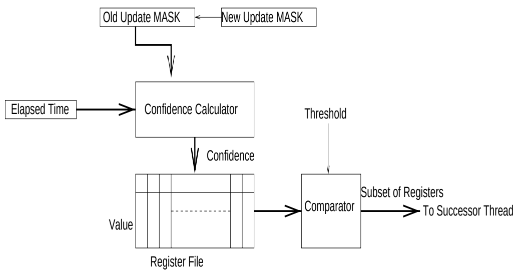

Figure 2: Confidence Estimator in a PE

PE Parameter	Value
Max task size	32 instructions
PE issue width	2 instructions/cycle
Thread predictor	2-level predictor 1K entry, pattern size 6
L1 - Icache	16KB, 4-way set assoc., 1 cycle access latency
L1 - Dcache	128KB, 4-way set assoc., 2 cycle access latency
Functional unit latencies	Integer/Branch :- 1 cycle Mult/Divide :- 10 cycles
Number of PEs	4

Table 1: Default Parameters for the Experimental Evaluation

Figure 2 shows the hardware parts that need to included in each PE. The sequencer shown in Figure 1 triggers the confidence estimator in a PE whenever a task is assigned to its successor. The *elapsed time register* keeps tracks on the number of cycles elapsed since the thread assignment to the PE. If the data bus is not wide enough to carry the register values needed from a PE to its successor, the values are sent in several cycles. Although we have not faced this scenario.

4 Experimental Evaluation

In this section we present the experimental evaluation of our scheme. The experimental methodology is presented followed by the results.

4.1 Experimental Methodology and Setup

Our experimental setup consists of a detailed cycle-accurate execution-driven simulator based on the MIPS-I ISA. The simulator accepts executable images of programs, and does cycle-by-cycle simulation. The simulator faithfully models all aspects of the architecture. Some of the hardware parameters are fixed at the default values given in Table 1.

For benchmarks, we use a collection of 8 programs from SPECint 95/2000. The programs are compiled for a MIPS-Ultrix platform with a MIPS C (Version 3.0) compiler using the optimization flags distributed in the SPEC benchmark makefiles. We ran each simulation for 100 million instructions.

The confidence estimation has a maximum value of MAX. The threshold is taken to be 0.5MAX. Penalty of update mask is 0.25MAX. Timing penalty is (0.25MAX) divided by the elapsed time. Finally, the distance penalty is scale*distance, where scale is 0.25MAX divided by total number of PEs. The above three penalties are added together and subtracted from MAX to generate the confidence value.

4.2 Experimental Results

Figure 3 shows the speedup obtained using our proposed scheme as compared to passing the register values at the assignment of a thread and at its physical commit time. As expected, we have obtained speedup in all benchmarks with an average speedup of 11.6%. The least speedup comes from `compress95` (3.9%) because it has the highest percentage of squashed threads due to mispredictions.

Beside speedup, the confidence estimation for register values decreases useless usage of resources. This is indicated in Table 2. In the table, we see the average active PEs per cycle. We can see that although the IPC of the new scheme is higher, it has lower active PEs per cycle. This is due to the fact that confidence estimation avoids sending useless values. Hence some instructions may be waiting for a value to come. This is better than using old values and then having to squash or re-execute.

It is useful to see how much we loose by using our scheme as opposed to the unrealistic scheme of sending all the values every cycle. Four of the eight benchmarks used have less than 1% slowdown. For the other four, `li` and `m88ksim` have 1% slowdown, and `vpr` has 3% slowdown. The highest slowdown of 11% is obtained from `ijpeg`. The average slowdown is about 2%.

5 Related Work

Register passing mechanisms have been studied in [13], where several strategies that varies in aggressiveness have been presented. They depend mainly

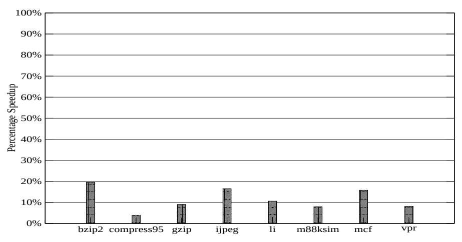

Figure 3: IPC Speedup

Benchmark	Average Active PEs per Cycle	
	No Conf.	Conf.
bzip2	3.75	3.73
compress95	3.45	3.42
gzip	3.76	3.77
ijpeg	3.63	3.32
li	3.22	3.13
m88ksim	3.67	3.51
mcf	3.72	3.62
vpr	3.67	3.55

Table 2: Average Active PEs per Cycle With and Without Confidence Estimations

on the compiler to analyze the threads generated.

Confidence estimation has been tried in many areas. For example, in [6], several confidence estimators for speculative control have been analyzed.

6 Conclusions

Register passing mechanism is a major issue for the performance of speculative multithreaded processors. Although memory values are communicated through shared memory, register values are communicated through the wire in the uni-directional ring that connects the PEs. In order to avoid useless execution as well as taking into account technological factors, we proposed using confidence estimation for register values. Whenever a register value is higher than a specific threshold, the value is sent to the successor PE. The proposed mechanism is not expensive in terms of hardware and does not need high bandwidth. Speedup of 11.4% over the the conservative strategy of sending the values at thread assignment and commit is achieved, and only 2% of slowdown is obtained as compared to the other extreme scheme of sending all register values every cycle.

The proposed method depends solely on hardware and does not rely on any compiler support. Future work involves incorporating the hardware solution with the compiler support to make use of compiler analysis in building the masks as well as suggesting priority for register values to be forwarded in case the available bus does not allow all values to be forwarded at once.

References

[1] H. Akkary and M. A. Driscoll. A dynamic multithreading processor. In *Proc. 31st Int'l Symposium on Microarchitecture*, 1998.

[2] S. E. Breach, T. N. Vijaykumar, and G. S. Sohi. The anatomy of the register file in a multiscalar processor. In *Proc. 27th Annual International Symposium on Microarchitecture (MICRO-27)*, pages 181–190, 1994.

[3] J-Y. Tsai et.al. Integrating parallelizing compilation technology and processor architecture for cost-effective concurrent multithreading. *Journal of Information Science and Engineering*, 14, March 1998.

[4] M. Franklin. *The Multiscalar Architecture.* Ph.d. thesis, technical report 1196, Computer Science Department, University of Wisconsin-Madison, 1993.

[5] M. Franklin. *Multiscalar Processors.* Kluwer Academic Publishers, 2002.

[6] Dirk Grunwald, Artur Klauser, Srilatha Manne, and Andrew R. Pleszkun. Confidence estimation for speculation control. In *Proceedings of the 25th annual international symposium on Computer architecture (ISCA)*, pages 122–131, 1998.

[7] L. Hammond, B. Nayfeh, and K. Olukotun. A single-chip multiprocessor. *IEEE Computer*, 1997.

[8] V. Krishnan and J. Torrellas. Executing sequential binaries on a clustered multithreaded architecture with speculation support. In *Proc. Int'l Symposium on High Performance Computer Architecture (HPCA)*, 1998.

[9] P. Marcuello and A. Gonzalez. Clustered speculative multithreaded processors. In *Proc. Int'l Conference on Supercomputing*, pages 20–25, 1999.

[10] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. E. Smith. Trace processors. In *Proc. 30th Annual Symposium on Microarchitecture (Micro-30)*, pages 24–34, 1997.

[11] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proc. 22nd Int'l Symposium on Computer Architecture (ISCA22)*, pages 414–425, 1995.

[12] J.-Y. Tsai, B. Zheng, and P.-C. Yew. Improving instruction throughput and memory latency using two-dimensional superthreading. Technical report, University of Minnesota, 1998.

[13] T. N. Vijaykumar, S. E. Breach, and G. Sohi. Register communication for the multiscalar architecture. Technical Report 1333, University of Wisconsin-Madison, 1997.