Return-Address Prediction in Speculative Multithreaded Environments

Mohamed Zahran[1] and Manoj Franklin[2]

[1] ECE Department, University of Maryland, College Park, MD 20742
mzahran@eng.umd.edu
[2] ECE Department and UMIACS, University of Maryland, College Park, MD 20742
manoj@eng.umd.edu

Abstract. There is a growing interest in the use of speculative multithreading to speed up the execution of sequential programs. In this execution model, threads are extracted from sequential code and are speculatively executed in parallel. This makes it possible to use parallel processing to speed up ordinary applications, which are typically written as sequential programs. This paper has two objectives. The first is to highlight the problems involved in performing accurate return address predictions in speculative multithreaded processors, where many of the subroutine call instructions and return instructions are fetched out of program order. A straightforward application of a return address stack popular scheme for predicting return addresses in single-threaded environments does not work well in such a situation. With out-of-order fetching of call instructions as well as return instructions, pushing and popping of return addresses onto and from the return address stack happen in a somewhat random fashion. This phenomena corrupts the return address stack, resulting in poor prediction accuracy for return addresses. The second objective of the paper is to propose a fixup technique for using the return address stack in speculative multithreaded processors. Our technique involves the use of a distributed return address stack, with facilities for repair when out-of-order pushes and pops happen. Detailed simulation results of the proposed schemes show significant improvements in the predictability of return addresses in a speculative multithreaded environment.

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] [4] [5] [9] [10] [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 (PEs). 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 dynamic program. For many non-numeric programs, SpMT may be the only option for exploiting parallelism [13].

Non-numeric programs typically tend to have a noticeable percentage of subroutine calls and returns. In order to obtain good performance for such programs, it is important to perform return address prediction [7]. That is, when a fetch unit encounters a return instruction, it must predict the target of that return, and perform speculative execution along the predicted path, instead of waiting for the return instruction to be executed. It is important to obtain high return address prediction accuracies in speculative multithreaded processors. Otherwise, many of the speculative execution happening in the processor will be along incorrect paths, and the advantages gained by speculative multithreading are lost. The traditional scheme for predicting return addresses is a *return address stack (RAS)* [7][11]. The RAS works based on the last-in first-out (LIFO) nature of subroutine calls and returns. When the fetch unit encounters a subroutine call, it pushes the return address to the top of the RAS; when the fetch unit encounters a return instruction, it pops the topmost entry from the RAS, and uses it as the predicted target of the return instruction. The fundamental assumption in the working of the RAS is that *instructions in the dynamic instruction stream are encountered by the fetch engine in program order.*

The above fundamental premise is violated in the case of speculative multithreaded processors, where multiple threads from a single sequential program are executed in parallel, causing call instructions (as well as return instructions) to be fetched in an order different from the dynamic program order. If all of the active threads share a common RAS, then pushes and pops to the RAS may happen out-of-order, thereby affecting the accuracy of return address predictions. The RAS mechanism, which works very well when accesses are done in correct order (especially with adequate fixup to handle branch mispredictions [6][11]), performs poorly when accesses are done in an incorrect order[1].

On the other hand, if each active thread has a private RAS that maintains only the return addresses of the calls encountered in that thread, then also the prediction accuracy is likely to be poor. This is because, in an SpMT environment, it is quite possible for a subroutine's call and return instructions to be present in different threads. This is likely to result in lower prediction accuracy, unless there is communication between the private RASes and proper repair mechanisms are included.

It is worthwhile to note that some repair mechanisms are beneficial even for using RAS in a single-threaded environment [6][11]. These repair mechanisms are useful for dynamically scheduled single-threaded processors. When several threads simultaneously update the RAS in a random order, these repair techniques will not yield good performance, as we show in this paper.

This paper investigates return address prediction in SpMT processors. It analyzes different scenarios that lead to incorrect predictions, and investigates a technique to perform accurate return address prediction in SpMT processors.

[1] Notice that this problem can be easily solved for multi-program multithreaded environments such as SMT [15] and Tera [2], by providing a separate RAS for each active thread. The same approach can be used in traditional multiprocessing environments where the parallelly executed threads do not have a sequential order.

The rest of this paper is organized as follows. Section 2 illustrates how speculative multithreading impacts a return address stack, and renders it unsuitable for accurate prediction of return addresses. Section 3 presents solutions to handle this problem, both for environments using constrained threads and for environments using unconstrained threads. Section 4 presents an experimental evaluation of the proposed schemes, and Section 5 presents the conclusions.

2 Inadequacy of a Shared Return Address Stack

The conventional RAS [7] was designed with a single-threaded processor in mind. We present two frequently occurring scenarios that corrupt the RAS in an SpMT processor in which all threads access a single RAS. These scenarios illustrate the need to use a distributed RAS, where each thread has its own RAS.

2.1 Constrained Threads

In speculative multithreaded architectures with a single RAS, the PEs can update the RAS in an incorrect order. For simplicity, let us first consider an SpMT environment in which each thread can have at most one call or return instruction. We call such threads *constrained threads*. SpMT processors that use constrained threads of this nature are the multiscalar [12], trace processor [10] and superthreaded processor [14]. Figure 1 shows two constrained dynamic threads[2] T0 and T1 that have been assigned to PE0 and PE1 respectively. Each PE (processing element) is nothing more than a small scale superscalar processor. PEs are connected together in a uni-directional ring [4]. Both threads have one call instruction each. If these two threads are executed in parallel, there is a possibility that the call instruction in T1 is fetched prior to fetching the call instruction in T0. If this happens, the return address of the call in T1 (i.e., B) is pushed to the RAS prior to pushing the return address of the call in T0 (i.e., A). Figure 1 shows this scenario where T0 is executed in PE0 and T1 in PE1.

2.2 Unconstrained Threads

If a single thread is permitted to have multiple call and/or return instructions as in [1][3][16][8], then the above problem gets exacerbated, as indicated in Figure 2. The figure shows two unconstrained dynamic threads, T0 and T1, that have been assigned to PE0 and PE1 respectively. PE 0 encounters its first call instruction and pushes return address `A` onto the RAS. PE 1 subsequently encounters its call instruction, and pushes address `C` onto the RAS. Finally, PE 0 encounters the second call instruction in its dynamic thread, and pushes address `B`. Now the RAS is incorrect, because the two addresses pushed by PE 0 must have been

[2] By "dynamic thread" we mean the path taken through the thread at run-time. A dynamic thread's instructions may not necessarily be in contiguous locations in the static program. In this paper, by "threads" we mean "dynamic threads" unless otherwise stated

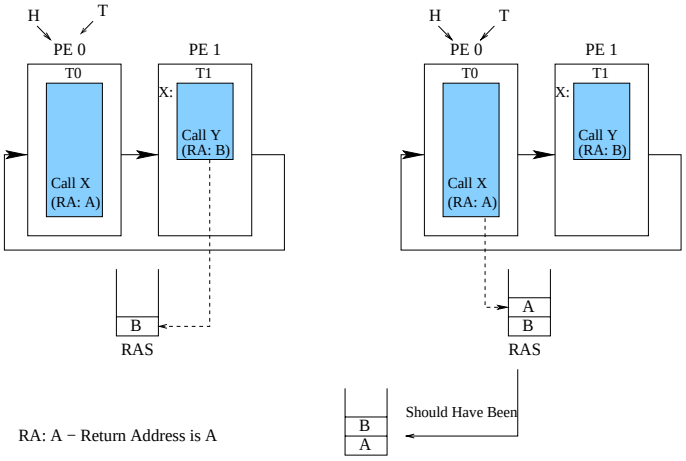

Fig. 1. Problem with Single RAS when using Constrained Threads

pushed *before* PE 1 pushed address `C`. The main problem here is that each thread is accessing the RAS, assuming that it is the only running thread. This is the main consequence of the RAS being designed with the single-threaded model in mind. In an SpMT processor, many threads will be concurrently accessing the RAS, frequently causing out-of-order updates to the RAS. If the manner in which the RAS is updated is not controlled, it gets corrupted very quickly, leading to poor performance.

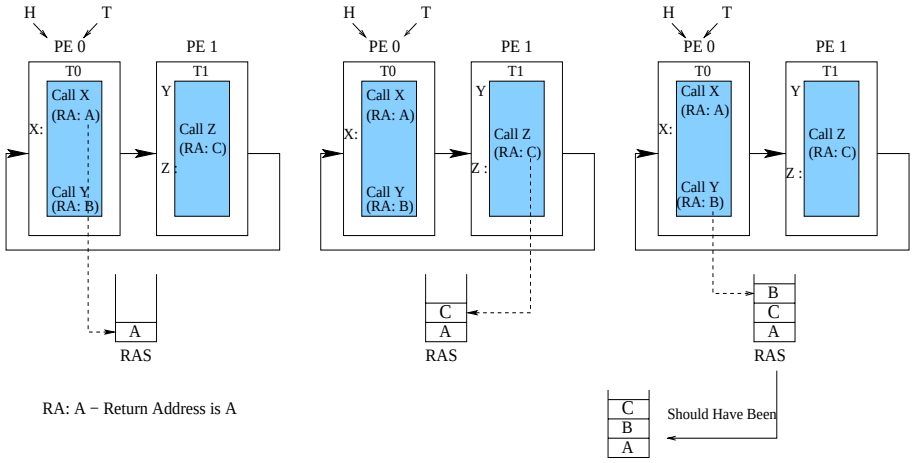

Fig. 2. Problems of Single RAS with Unconstrained Threads

3 Design Issues and Proposed Schemes

In this section we present schemes for dealing with the problematic scenarios depicted in Section 2. We propose separate solutions for SpMT processors using constrained threads and those using unconstrained threads. We shall begin with the constrained threads case. For this environment, we propose fixup mechanisms for a single shared return address RAS. We then move on to SpMT environments with unconstrained threads, and propose a solution for such environments.

3.1 Shared Return Address Stack with Fixup

If threads are constrained to have at most one call instruction or one return instruction, then the primary reason for out-of-order updates to the RAS is that some threads may have their call/return instruction fetched earlier than that of their predecessors. This can happen either because the thread is much shorter than its predecessors or because its call/return instruction is embedded early on in one of the control flow paths in the thread.

The fixup we propose to handle this situation is as follows: if a thread contains a subroutine call, the return address of the call (which is a constant) is noted along with other information pertaining to the thread. This is done at compile time, when threads are formed. Each thread is allowed to have a maximum number of targets. A detailed description on how threads are formed can be found in [4]. At run time, whenever a thread-level prediction is done to determine the successor thread (i.e., picking one of the possible targets of the current thread), a check is done to verify if that prediction amounts to executing a subroutine call or return. If the prediction amounts to executing a call, then the return address associated with that call is pushed onto the RAS, even before the corresponding call instruction is fetched by its PE. If the prediction corresponds to executing a subroutine return, then the return address at the top of the RAS is popped, and used as the predicted target of the return instruction. Thus, popping of a return address from the RAS is performed prior to fetching the return instruction. Because thread-level predictions are done in program order, this fixup guarantees that all updates due to subroutine calls and returns update the RAS in the proper order, as long as there are no thread-level mispredictions.

When thread-level mispredictions occur, the reverse sequence of operations is performed during rollback. Thus, when a misprediction is detected, threads are squashed, starting from the youngest one. When a thread is squashed, a check is made to verify if the thread had earlier performed a push or pop to the RAS. If so, the action is undone. If a pop was done earlier, the return address that was popped off earlier is pushed onto the RAS. Similarly, if a push was done earlier, then a pop is done at rollback time. Notice that this fixup scheme with a shared RAS works only if threads are constrained to have at most a single call/return instruction.

3.2 Distributed Return Address Stack

When a thread is allowed to have multiple call/return instructions, the shared RAS scheme with fixup can perform poorly, as illustrated in Figure 2. To handle this case, we propose a distributed RAS, where each PE has its own individual RAS, and each RAS communicates with its neighbors. The working of the distributed RAS can be summarized as follows:

- When a call instruction is fetched in a PE, its return address is pushed to the corresponding PE's RAS.
- When a return instruction is fetched in a PE, the return address is predicted by popping from the PE's RAS. If its RAS is empty, forwards the request to its predecessor PE.
- When a thread commits, its PE's RAS contents (if any) migrate to the bottom of the successor PE's RAS. It is to be noted that during our experiments, we have never experienced stack overflow. For all the benchmarks a stack of size 40 is more than enough.
- When a thread is squashed, to undo a push, its PE pops the top address from its RAS if it is not empty. If the RAS is empty, the forward the request to the *successor* RAS.
- When a thread is squashed, to undo a pop, a PE pushes onto its *own* RAS the return address that it had previously popped.

To adhere to the concept of the order imposed on the threads assigned to the PEs, the RASes must be connected in the same manner as the PEs. To be able to fulfill the other points, this connection must be bidirectional. In order to undo a pop, each PE must store the address that it had previously popped. In case a PE is executing a thread having multiple return instructions, then the PE stores all of the popped addresses in a *queue* called predicted PC. Figure 3 shows the entire scheme incorporated in an SpMT processor. By distributing the RAS in this manner, we can reduce the number of disruptions to the return address stack.

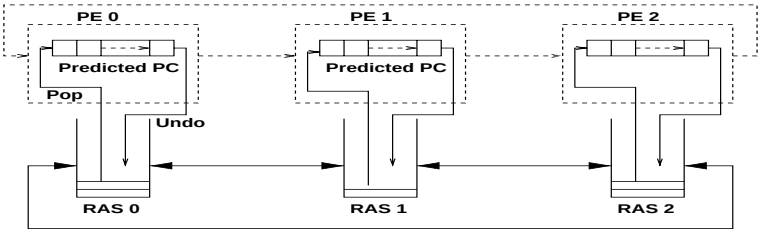

Fig. 3. Distributed Return Address Stack

4 Experimental Evaluation

In this section we present a detailed quantitative evaluation of the presented techniques. Such an evaluation is important to study the performance gain that can be obtained from these techniques and to see how efficient they are with standard benchmarks.

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; it is not trace driven. The simulator faithfully models all aspects of a speculative multithreaded microarchitecture.

Two different thread types were used: constrained and unconstrained. Table 1 shows the default hardware parameters used in the experiments. The benchmarks used are from SpecInt95 and SpecInt2000. Because of detailed cycle-accurate simulation, the experiments take a long time to run. Each data point is obtained by simulating the benchmark for 100 million instructions. To know the effect of return address prediction mechanisms on the performance of processors in general, we counted the number of return instructions in the first 100 million dynamic instructions. We found that `m88ksim`, `li` and `compress95` have the highest count; `bzip2` and `mcf` have the lowest number of return instructions, thus the impact of the return prediction may not be high for these two benchmarks.

Parameter	Value
Number of PEs	4
Max thread size	32
PE issue width	2 instructions/cycle
Thread-level 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	Int/Branch: 1 cycle; Mul/Div: 10 cycles
RAS size	40 entries
RHT size	1K entries

Table 1. Default Values for Simulator Parameters

4.2 Results for Shared RAS with Fixup

The first set of experiments simulate a speculative multithreaded architecture running constrained threads. These experiments use a single RAS. Each PE has only a single predict PC register, since this is the highest number of call/return allowed. Figure 4 shows the percentage of return mispredictions without and with fixup. As seen in the figure, the repairing mechanism leads to substantially fewer mispredictions in all the benchmarks. This highlights the impact of fixup

operations in the RAS of an SpMT processor. Table 2 shows the speedup obtained using repairing mechanism over non-repair. On average, there is a speedup of 10.3%. We note that a small increase in the accuracy of `m88ksim` leads to speedup of 10.5%, and this is due to the fact that `m88ksim` has the largest number of return instructions. The same argument can be said about `compress95` which is on of highest benchmarks in terms of return instructions in the dynamic instructions stream. Furthermore, no speedup is obtained from `mcf` and this is due to the fact that it has the lowest number of return instructions in the 100M dynamic instructions stream among all the benchmarks. It is to be noted that a single return address misprediction can result in a big loss of performance. Because this leads all subsequent PEs to execute threads in a wrong path. This is quite different from branches, because a branch mispredictions may be enclosed in a thread and does not affect other threads.

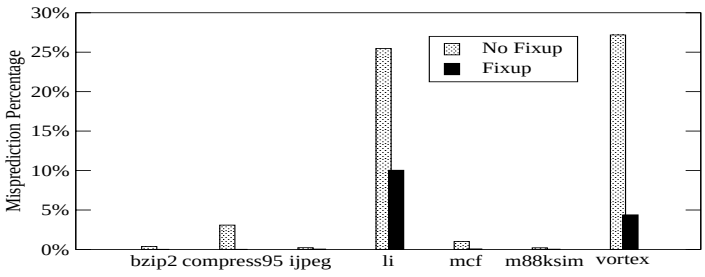

Fig. 4. Percentage of Return Address Mispredictions in Speculative Multithreaded Architecture Using a Shared RAS and Constrained Threads

Benchmark	Speedup
bzip2	3.7%
compress95	24.6%
ijpeg	4.7%
li	11.0%
mcf	0.0%
m88ksim	10.5%
vortex	13.8%

Table 2. Speedup Obtained by RAS Fixup (Over that Without Fixup) in a Speculative Multithreaded Architecture Using a Shared RAS and Constrained Threads

4.3 Results for Distributed RAS

The next set of experiments deal with unconstrained threads and distributed RAS. In these experiments, we allow each thread to have a total of up to 4 calls and returns. It is to be noted that the only constraint on the number of call/return instructions per thread is hardware availability, i.e., the number of entries in the predicted PC queue. Thus, we use a distributed RAS as explained

in Section 3, with each PE having a predicted PC queue of length 4. Figure 5 shows the results we obtained. These results are a mixed bag. The mispredictions are low for `bzip2`, `compress95` and `m88ksim`, and high for `li` and `vortex`. As expected the number of mispredictions is higher than that obtained with constrained threads. This results in part from the complex scenarios that arise due to the larger threads. For example, if a PE has a return instruction and its stack is empty, then the PE will try to pop a return address from the predecessor RAS. If the predecessor has not yet executed all of its returns and calls, then the PE will certainly pop a wrong value, because at that point of time the stack is not yet complete and the correct address must be the top of the stack after all the calls.

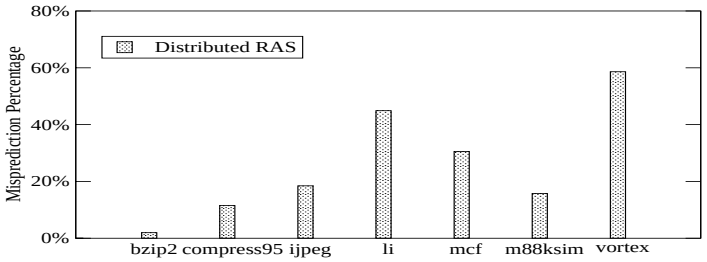

Fig. 5. Percentage of Return Address Mispredictions in a Speculatively Multithreaded Architecture Using a Distributed RAS and Unconstrained Threads

5 Conclusions

Several researchers and a few vendors are working on speculative multithreaded (SpMT) processors, to harness additional levels of parallelism. When multiple threads are fetched and executed in parallel, return instructions belonging to these threads are very likely to be fetched out of order. This impacts the return address history recorded in a return address stack (RAS), and affects its capability to accurately predict return addresses. This paper demonstrated the problems associated with using a shared RAS.

We also investigated a technique for reducing the number of return address mispredictions in SpMT processors. The technique is for SpMT processors that work with simple threads, where each thread can have at most one call/return instruction. This scheme monitors thread-level predictions to determine if execution of a thread involves taking a subroutine call or return, and updates the RAS in correct program order. The second technique we proposed involves the use of a distributed RAS mechanism, and is applicable for environments where a thread can have multiple calls and returns embedded in it. We described how the distributed RASes communicate with each other to collectively provide a RAS with fewer disruptions.

The results of our experiments show that for the constrained threads case, our technique provides very high return address prediction accuracies. For the unconstrained threads case, our distributed RAS scheme works well for some benchmarks. Further work is needed to improve the accuracy for the remaining benchmarks.

References

1. H. Akkary and M. A. Driscoll. A dynamic multithreading processor. In *Proc. 31st Int'l Symposium on Microarchitecture*, 1998.
2. R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, , and J. B. Smith. The tera computer system. In *Proc. International Conference on Supercomputing*, pages 1–6, 1990.
3. A. Bhowmik and M. Franklin. A general compiler framework for speculative multithreading. In *Proc. ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, August 2002.
4. M. Franklin. *Multiscalar Processors*. Kluwer Academic Publishers, 2002.
5. L. Hammond, B. Nayfeh, and K. Olukotun. A single-chip multiprocessor. *IEEE Computer*, 1997.
6. S. Jourdan, J. Stark T-H. Hsing, and Y. N. Patt. The effects of mispredicted-path execution on branch prediction structures. In *Proc. Int'l Conference on Parallel Architectures and Compilation Techniques (PACT)*, October 1996.
7. D. R. Kaeli and P. G. Emma. Branch history table prediction of moving target branches due to subroutine returns. In *Proc. 18th Int'l Symposium on Computer Architecture*, 1991.
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. K. Skadron, P. S. Ahuja, M. Martonosi, and D. W. Clark. Improving prediction for procedure returns with return-address-stack repair mechanisms. In *Proc. 31st Int'l Symposium on Microarchitecture*, pages 259–271, 1998.
12. 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.
13. J. G. Steffan and T. C. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *Proc. Int'l Symposium on High Performance Computer Architecture*, pages 2–13, 1998.
14. J-Y. Tsai, J. Huang, C. Amlo, D. J. Lilja, and P-C. Yew. The superthreaded processor architecture. *IEEE Transactions on Computers*, 48(9):881–902, 1999.
15. D. M. Tullsen, S. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proc. 22th Int'l Symposium on Computer Architecture*, 1995.
16. M. Zahran and M. Franklin. Hierarchical multi-threading for exploiting parallelism at multiple granularities. In *Proc. 5th Workshop on Multithreaded Execution, Architecture and Compilation (MTEAC-5)*, 2001.