RHT: A Context-Based Return Address Predictor

Mohamed Zahran
Department of Electrical Engineering
City College of New York of
City University of New York
New York, NY 10031
mzahran@ccny.cuny.edu
Phone:+1-212-650-7310

Manoj Franklin
Department of Electrical and Computer Engineering
University of Maryland
College Park, MD 20742
manoj@eng.umd.edu

Abstract

With the widespread use of modular programming, subroutine calls/returns become a major programming construct and must be dealt with in the most efficient way during execution. The popular scheme for predicting return addresses in single-threaded processors is the return address stack (RAS). However, it does a poor job for multithreaded execution models such as speculative multithreading (SpMT), in which threads are extracted from sequential code and are speculatively executed in parallel. This is because SpMT processors may fetch subroutine call instructions and return instructions out of program order. With out-of-order fetching of calls and returns, push and pop operations to the return address stack happen in a somewhat random fashion, corrupting the return address stack.

In this paper we propose a scheme for accurately predicting return addresses for the newly introduced schemes. It involves using a return history table (RHT) that stores the frequently encountered return addresses, along with proper context. The RHT works on the principle of locality of return addresses (as opposed to the last-in first-out nature of subroutine calls and returns), and is therefore insensitive to disruptions in the call/return order. Detailed simulation results show that the prediction accuracy of return addresses in an SpMT processor can be improved greatly with the use of a small RHT. This leads to an average IPC (Instruction Per Cycle) improvement of 10% over the IPC obtained with a conventional centralized return address stack.

The proposed RHT scheme can also be used in other multithreading models such as simultaneous multithreading (SMT) and chip multiprocessing (CMP).

Keywords: Return Address Prediction, Speculative Multithreading, Microarchitecture

1 Introduction

Advances in programming languages lead to more prevalent use of modular programming, resulting in more call/return instructions in the programs executed [2]. In order to obtain good performance for such programs, it is important to perform return address prediction [8]. That is, when the 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. and the advantages gained by speculative multithreading are lost.

The traditional scheme for predicting return addresses is a *return address stack (RAS)* [8][12]. 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 corresponding return address to the top of the RAS; upon encountering a return instruction, the fetch unit 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 of the dynamic instruction stream are encountered by the fetch unit in the correct program order.*

For multithreaded architectures using techniques such as speculative multithreading (SpMT) [1] [5] [9] [10] [13], the return address stack is not a feasible solution. In these architectures, 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 is limited to extracting parallelism from a group of adjacent instructions that fit in a dynamic scheduler, an SpMT processor can extract parallelism from multiple, non-adjacent, regions of a sequential program. For many non-numeric programs, SpMT appears to be the only option for exploiting parallelism [14]. 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,

The basic premise of a return address stack — in-order fetching of the subroutine call/return instructions in a dynamic instruction stream — is violated in the case of speculative multithreaded processors, when 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 [7][12]), performs poorly when accesses are done in an incorrect order

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 (and does not communicate with other RASes), 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 incorrect predictions, unless there is communication between the private RASes and proper repair mechanisms are included.

At this point, it is important to note that some RAS repairing techniques have been proposed earlier to handle control mispredictions for single-threaded processors [7][12]. Those techniques are geared for single-threaded out-of-order execution processors, and not for speculative multithreaded processors, which perform out-of-order instruction *fetch* also! When several threads simultaneously update the RAS, the techniques in [7][12] by themselves are insufficient, as they assume that the RAS is accessed by a single thread only.

With the need for both high throughput as well as high performance for a single thread, we can expect the emerge of paradigms which combine both SMT and SpMT in the near future. Therefore, we need to find a common and simple solution for the problem of return address prediction. The solution must be simple, in order to provide fast prediction, especially with the clock frequency skyrocketing. The solution must be applicable without change to both SMT/CMP as well as SpMT.

This paper investigates return address prediction in SpMT processors, and the solution provided can be incorporated in CMP and SMT processors without change. In this paper, we analyze different scenarios that lead to incorrect predictions in SpMT processors, and investigates a technique to perform accurate return address prediction. The proposed technique is based on the idea of *locality of return addresses*. We can categorize locality of return addresses into two groups:

- **Locality of return instructions:** Although there are many return instructions in the dynamic stream, as indicated in the second column of Table 1, there are only a few distinct return instructions. This is indicated in the third column of the table. This is expected, because the same subroutine can be called from different places. Furthermore, these distinct return instructions are not occurring with the same frequency. More often, a small number of return instructions occur more than 70% of the time. For example, both `ijpeg` and `vpr` have a return instruction that occurs 99% of the time. On the other extreme, the return instructions of `vortex` are equally distributed. For `compress95`, two return instructions occur together 72% of the time. The impact of this type of locality is that the amount of information that needs to be processed in order to make accurate return address prediction is small, hence requiring less hardware and storage.

- **Locality of return instruction's targets:** A look at the rightmost 5 columns of Table 1 shows that the majority of the return instructions have only one or two targets. This means that the targets of return instructions exhibit a repetitive behavior. However, some return instructions have many targets and the percentage of these instructions is not negligible. The best scenario for prediction is to have return instructions with a high frequency of occurrence and a small number of targets. Unfortunately, this is not always the case. Sometimes a frequently occurring return instruction has many targets. For example, the return instruction that occurs in `vpr` 99% of the time has 6 targets and these targets are somewhat equally distributed. For such return instructions, the behavior of the targets may not be easily predictable. We need to make use of extra context information that may be available. In this paper we propose the use of a history table that is insensitive to disruptions in the call/return order.

The rest of this paper is organized as follows. Section 2 presents background and related work. Section 3 presents the return history table (RHT) as a new mechanism for return address prediction. Section 4 presents a simulation-based experimental evaluation of the proposed schemes. Section 5 presents the conclusions.

2 Background and Related Work

Program execution involves many subroutine calls and returns. Most often the subroutine calls are nested; that is, subroutine `f1` calls subroutine `f2`, which calls `f3`, and so on. When a return instruction is fetched, the correct return address must be predicted, in order to continue speculative fetching of subsequent dynamic instructions. Because the return address to be predicted is the address of the instruction immediately following the most recently encountered call instruction, the best structure for predicting return addresses is a stack (Last In First Out).

2.1 Return Address Stack (RAS)

The return address stack (RAS) [15][8] is a microarchitectural stack that stores the return addresses of the recently encountered call instructions. Notice that the RAS is different from the regular stack used for implementing the activation frames of dynamically encountered subroutines, and for parameter passing between subroutines. Figure 1 shows an example code having nested subroutine calls. Part (i) shows the static code. Part (ii) shows the dynamic calling sequence through this code. Each edge in this control graph indicates either a subroutine call or a return from a subroutine. When subroutine `X` is called, its return address `A` is pushed onto the RAS. The call to subroutine `Y` causes the return address `B` to be pushed onto the RAS. Finally, after subroutine `Z` is called, the RAS contents will be as shown in part (iii) of Figure 1. Each return instruction encountered thereafter will pop off the return address at the top of the RAS. The above RAS works fine as long as call and return instructions are fetched in program order.

2.2 Fixup to Handle Control Mis-Speculations

Branch mispredictions have made the problem of return address prediction more challenging, as they

Table 1: Number of Return Instructions, and Distribution of Return Instructions Based on The Number of Targets. These Data are for the first 100M Dynamic Instructions of SPECint 95/2000 Benchmarks. These benchmarks are written in C. Programs written in C++ or Java tend to have an even higher count of return instructions.

Benchmark	Total Number of Dynamic Return Instructions	Number of Distinct Return Instructions	Number of Targets per Return Address				
			1	2	3	4	≥ 5
bzip2	873,449	34	79.4%	8.8%	2.9%	2.9%	6.0%
compress95	2,565,679	38	73.7%	7.9%	10.5%	0.0%	7.9%
ijpeg	2,286,735	112	76.8%	12.5%	6.3%	0.9%	3.5%
li	3,259,336	150	62.7%	14.0%	8.0%	3.3%	12.0%
vortex	1,894,679	415	55.2%	13.7%	5.8%	3.6%	21.7%
vpr	4,285,130	30	80.0%	6.7%	6.7%	3.3%	3.3%

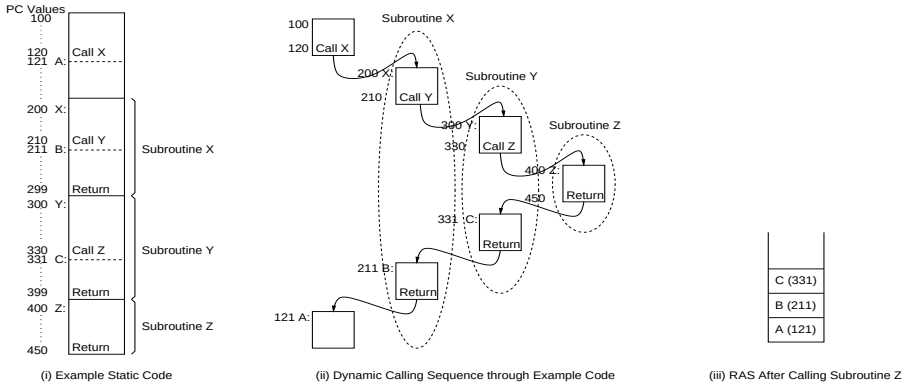

Figure 1: Example Code, and the Use of an RAS to Store and Predict Return Addresses

cause incorrect addresses to be pushed onto the RAS, and improper pops to be done from the RAS. This problem was identified in [7], where a solution was also presented. The solution consists of checkpointing the RAS state information at each branch prediction, so as to reinstate the RAS top, after detecting a control misprediction. The state information of the RAS consists of the top of the stack pointer as well as the pointer pointing to the RAS entry to be written for the next subroutine call. Another repair scheme was proposed in [12] to overcome this problem to a limited extent. This scheme involved checkpointing the *contents* of the RAS top entry, in addition to checkpointing the *location* of the RAS top. By saving the contents of the top stack entry along with the location of the RAS top, a mis-speculated sequence of at least two pops followed by at least one push is necessary to corrupt the RAS. All of the above work assumes that the RAS is accessed by only a single thread. In SpMT processors, as we will see, a return instruction may be encountered even before its corresponding call has been executed! Hence the problem of return address prediction is more challenging when multiple threads simultaneously access the RAS in a random fashion.

2.3 Speculative Multithreading (SpMT) Basics

The central idea behind speculative multithreading is to have multiple flows of control within a process, allowing different parts of the process to be executed in parallel. Threads are extracted from sequential code and are speculatively run in parallel, without violating the sequential program semantics. Inter-thread communication between two threads (if any) will be strictly in one direction, as dictated by the sequential thread ordering. No explicit synchronization operations are necessary.

Examples of SpMT are the multiscalar model [5][13], the superthreading model [4], the trace processing model [10], the clustered speculative multithreaded model [9], and the dynamic multi-threading model [1]. The Processing Elements (PEs) of an SpMT processor are usually organized as a circular queue as indicated in Figure 2, in order to maintain sequential thread ordering. To maintain sequential semantics, new threads are activated at the tail of the queue, and completed threads are retired from the head, by a thread allocation unit (TAU). Whenever an incorrect speculation is detected, the incorrect threads are squashed from the tail side.

There are several strategies for thread spawning. In the clustered speculative multithreaded architecture [9], threads are spawned at loop iterations. In the dynamic multithreading processor [1], threads are spawned at subroutines call/returns as well as post-loop threads. Threads may be generated at run-time by the hardware as done by the trace processors [10]; or at compile time, like the multiscalar processor[13].

While spawning threads at subroutine calls/returns

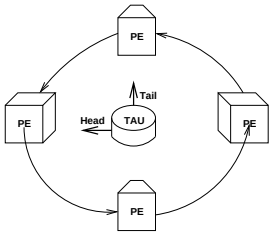

Figure 2: Circular Queue Arrangement of PEs in a Typical SpMT Processor

reduces the problem of return address prediction, some subroutines are too small to be spawned, because they do not expose enough parallelism and they are consuming resources at the same time. Spawning at loop iterations or post-loops is a good strategy because loops expose parallelism. However, a loop iteration can contain subroutine calls/returns, and this will expose the problems of RAS in an SpMT processor.

2.4 Problems of RAS in an SpMT Processor

The conventional RAS was designed with a single-threaded speculative processor in mind. When used in an SpMT processor, it is likely to perform poorly, because multiple threads may access it out of order. This problem was first identified in [16] where a centralized RAS (CRAS) with fixup was proposed for threads having at most one call/return instruction. However, when threads are allowed to have multiple call/return instructions, CRAS is likely to perform poorly even with the fixup techniques proposed in [16]. An intuitive solution is to have several RASes. We can think of two types of RASes.

- **Private RASes:** Each thread has its own RAS and there are no interaction between these RASes. This method cannot be used in our case, because quite often the call instruction and its corresponding return instruction will be in two different threads. Private RASes are good for multi-program environments where the different threads are coming from different programs.
- **Distributed RASes (DRAS):** Each thread has its own RAS, but the RASes are interconnected. DRAS is proposed and evaluated in [16] but the results were not very encouraging for long threads having multiple call/returns.

3 Return History Table (RHT) Predictor

In this paper, we propose a new prediction method that is less sensitive to out-of-order call and return instructions, and is independent of the progress of the predecessor threads. Moreover, this scheme is simple and general enough to be immune to the way threads are generated. The new return address predictor uses a table called Return History Table (RHT). It does not use an RAS, but instead makes use of the return instruction's context to predict the return address. The idea of RHT is presented below, followed by the implementation issues.

3.1 RHT Predictor

The basic idea is to use a table called *Return History Table (RHT)* that keeps a history of the return addresses organized on a context basis, and make a prediction based on the stored history. The RHT predictor works on the principle of *locality of return addresses*, as opposed to the last-in first-out nature of subroutine calls and returns. It is therefore less sensitive to disruptions in call/return order. The RHT predictor's structure is shown in Figure 3. Its main component is the history table. Each entry of this table consists of 3 fields: a `tag`, a `return address`, and a `confidence value`. The predictor works as follows:

When a prediction is required, an appropriate context is submitted to a hash function, which outputs an index into the RHT. Notice that the same return instruction PC may have several entries in the RHT, depending on the context. This is better than indexing the table using only the return address, given that the same subroutine is likely to be called from different places in a program. The output of the hash function is used to access the RHT, and the tag is used for comparison. The tag consists of the least significant bits of the index generated by the hash function. If the tag matches and the confidence value is higher than a specific threshold, the corresponding return address is returned as a prediction. The RHT is updated each time a return instruction physically commits.

The RHT has some similarities with the target cache proposed in [3] for predicting targets of indirect jumps. However, in [3], branch path and pattern information are used to provide an index to the table. Furthermore, the target cache is assuming that a single thread accessing the target cache.

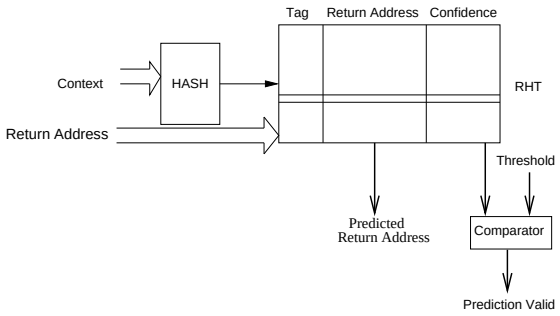

Figure 3: Structure of RHT Predictor

3.2 Implementation Issues of RHT

To get the best performance from RHT, several implementation issues must be considered. The important issues are the context to be used to access the RHT, the hash function, and the derivation of the confidence values.

Context: The return address of a dynamic return instruction depends on the place from which that subroutine was called, and not directly on the control flow within the subroutine[1]. Hence the *context* of the return instruction is pivotal for a correct return address prediction. This context is used to index the RHT table. A poorly chosen context will lead to a lot of aliasing, resulting in poor prediction or no prediction at all. Because the same subroutine can be called from different places in the program, the PC of the return instruction is not enough as a context. Other available pieces of information are: the return instruction's thread ID (current thread ID) and predecessor thread ID[2]. If a subroutine is called from different places, then the predecessor thread ID is likely to be different for each different call. We need different entries in the RHT table for each different call. Hence it may be beneficial to include the predecessor thread ID also in the context. Furthermore, the same subroutine may belong to different threads because static code can be overlapped in dynamic code. So, it is better to include the current thread ID also in the context. Generally speaking, the efficiency of any context depends on the way the threads are formed. For instance, the above context is efficient most of the time, however it can fail in the following two situations:

- If a *static* thread contains multiple calls to the same subroutine, each call has a different return address, but the context happens to be the same.
- If a subroutine has been partitioned into multiple threads, then all dynamic instances of the subroutine's return instruction are likely to have the same context because the context depends only on the return instruction's thread ID and its predecessor thread's ID.

Hash Function: The context information needs to be condensed before using it to index the RHT. The hash function takes the context information as input and forms an index into the RHT. Desirable characteristics of a hash function are computational simplicity and information preservation. The hash function needs to be as simple as possible in order to avoid RHT access delays. Moreover, the index formed by the hash function must result in the least aliasing possible. After some trial and error experiments, we have chosen the exclusive-OR of the context information as our hash function. The most significant bits of the EXOR function's output are discarded, depending on the RHT size. We found that folding the extra bits of the context information instead of discarding them results in more aliasing here.

Confidence Value: A confidence mechanism is crucial to get the best performance from the RHT predictor. This is because a subroutine may be called from different places in a program, but different instances of its return instruction may have the same context (i.e., the same current thread ID and predecessor thread ID), in which case it is more beneficial not to make any prediction. The confidence Value can be determined using a saturating counter that is increment at times of correct prediction and decremented at times of misprediction. Such a confidence estimator has been used before in the context of branch prediction [6].

3.3 Important Characteristics and Advantages of RHT

The RHT is not really a conventional data value predictor. For example, the stride and last value predictors cannot be used because return addresses do not follow a stride pattern. A two-level context based predictor [11] is not a good choice for return address prediction. First, the context of return address prediction does not need a complicated design as the 2-level table configuration, because the context is implicit in the indexing part of RHT. Furthermore, the table size does not need to be large because addresses are more restricted than the more general data values. Return instructions are also fewer than branches and result-producing instructions.

One of the main advantages of RHT is that it can be easily incorporated in any speculative multithreaded architecture. For instance, it can be added to the DMT [1] instead of using multiple RASes, and it needs much less hardware and communication requirements. However, the evaluation of the above suggestions is out of the scope of this paper.

Moreover, RHT can be incorporated without change in CMP and SMT paradigms. RHT can be shared among the independent threads. with the appropriate context information in the hashing, aliasing can be kept at a minimum.

Another advantage is the time taken for recovery. A corruption in an RAS leads to several subsequent mispredictions. This is not the case with an RHT, because each prediction does not depend heavily on the previous operations.

3.4 Hybrid of CRAS and RHT

The hybrid system works as follows. When a return address prediction is to be made, RHT and CRAS are interrogated simultaneously. If RHT makes a prediction, then it is taken and the address popped from the CRAS is saved to be restored in case of thread squashing. If the RHT does not provide any prediction, then the value popped from CRAS is used. Otherwise, no prediction is made. In case of a call instruction, the return address is pushed in the CRAS as usual.

4 Experimental Evaluation

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

[1] It is possible that the control flow within the subroutine may be correlated to the place from where it was called.

[2] Each static thread is given a unique ID. If threads are formed at run-time, then the thread ID can be the PC of the thread starting instruction.

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 architecture (a generalized multiscalar architecture [5][13] with much larger threads).

We use 6 benchmark programs from SpecInt95 and SpecInt2000. The benchmarks are compiled using the accompanied makefiles. We used RAS of 40 entries with fixup.

We use RHT table of 1K entries. For RHT to make a prediction, the confidence must be larger than 1. This means the target of the return instruction must occur twice in a raw in order to make prediction. The confidence estimator is simply a saturating counter. 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, after skipping the startup stage. The rest of the parameters are similar to [16].

Thread Spawning Strategy: In the experiments conducted in this paper, threads and generated and spawned using a combination of software and hardware. Static threads are formed at compile time as follows. Instructions are concatenated together until either 128 instructions are reached, a specific number of calls is reached (in our experiments 4 calls), or a return instruction is encountered. This information is conveyed to the hardware with the program binary. At run-time, as soon as a thread is assigned to a PE, a thread prediction is done to determine the successor thread. The predicted thread is assigned to the successor PE. This process is continued until all PEs are assigned threads.

RHT Context: The RHT context used in all our experiments in this paper, uses the PC of the return instruction, the ID of the predecessor thread, and the ID of the current thread (i.e. the thread containing the return instruction). The fact that the same function can be included in the several threads, makes the current thread ID important component of the context. And, the fact that the same function can be called from several different places makes the predecessor thread ID another must in the context. In summary, the three pieces of information (PC, current thread ID, and predecessor thread ID) are needed together in order to avoid interference. The hash function used to combine the above context is the exclusive-OR function.

4.2 RHT Performance as Compared to RAS With Fixup

The first set of experiments shows the need for the RHT, and the inappropriateness of a conventional RAS. It compares three return address prediction schemes: a conventional centralized RAS, an RHT, and a hybrid system of CRAS and RHT. Figure 4(i) shows the percentage of return mispredictions with a CRAS, an RHT, and an RHT_CRAS hybrid. For each benchmark, there is a bar corresponding to each one of these three schemes. As seen in the figure, The addition of the RHT to the CRAS leads to a substantially higher number of correct predictions in all the benchmarks. The RHT by itself has a higher prediction accuracy than the CRAS for `compress95` and `vpr`. The hybrid system rarely makes a non-prediction. Sometimes, aliasing causes the RHT to make incorrect predictions. For example, `ijpeg` and `vortex` have the lowest RHT return prediction accuracies. This is because `ijpeg` has a large number of dynamic instructions per function. Each function spans several threads and hence the current thread ID, the predecessor thread ID as well as the PC of the return instruction are invariant, although the function may be called from different places. On the other hand, `vortex` has a small number of instructions per function call. This results in two functions to be encompassed in the same thread, thereby having different return addresses for the same context but different return addresses.

Figure 4(ii) shows the speedup obtained by the RHT over the CRAS. On average, there is a speedup of 10%. As expected, `ijpeg` does not exhibit any speedups due to its high misprediction rate. However, `vpr` has the highest speedup (47.87%) because it has the highest number of return instructions and it benefits the most from the RHT.

5 Conclusions

Speculative multithreading (SpMT) is an emerging technique to to harness additional levels of parallelism from a sequential program. In an SpMT processor, call and return instructions belonging to simultaneously executed threads are 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 investigated a new technique for reducing the number of return address mispredictions in SpMT processors. We proposed to augment a centralized RAS with a return history table (RHT) that works on the principle of locality of return addresses instead of the last-in-first-out principle of the stack. This makes the RHT less sensitive to disruptions caused by out-of-order fetch of call/return instructions

The proposed scheme was evaluated using experiments conducted with a cycle-accurate SpMT simulator. The results of our experiments showed that an RHT of 1024 entries leads to an average speedup of 10% over a conventional centralized return address stack. The overall performance can be improved further by using careful thread formation, either by hardware or software, to prevent problematic scenarios occurring in programs such as `ijpeg` and `vortex`.

The RHT predictor proposed in this paper can be applied to other multithreading paradigms such as simultaneous multithreading (SMT) and chip multiprocessing (CMP).

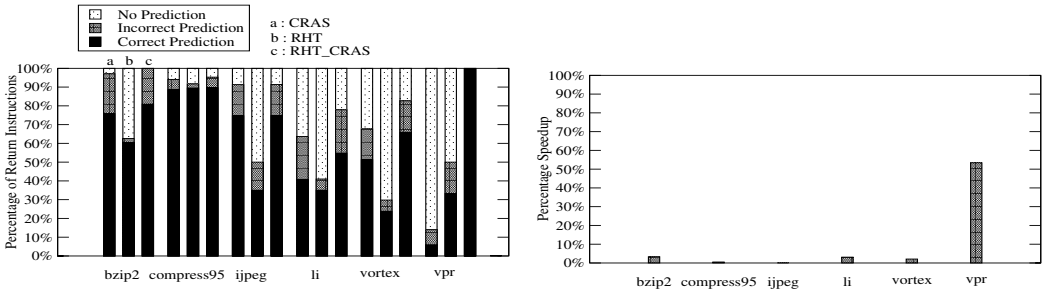

Figure 4: Results obtained for a Multiscalar Architecture with Different Return Address Predictors: (i) Return Address Prediction Accuracy; (ii) Speedup Obtained by the Hybrid System over CRAS

References

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

[2] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures.* Morgan Kaufmann, 2002.

[3] P-Y Chang, E. Hao, and Y. Patt. Target prediction for indirect jumps. In *Proc. 24th Int'l Symposium on Computer Architecture*, 1997.

[4] 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.

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

[6] E. Jacobsen, E. Rotenberg, and J. E. Smith. Assigning confidence to conditional branch predictions. In *International Symposium on Microarchitecture*, pages 142–152, 1996.

[7] 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.

[8] 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.

[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] Y. Sazeides and J. E. Smith. Implementations of context based value predictors. Technical report, University of Wisconsin-Madison, 1997.

[12] 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.

[13] 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.

[14] 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.

[15] C. F. Webb. Subroutine call/return stack. Tech. disc. bulletin, IBM, 1988. Vol. 30, No. 11.

[16] M. Zahran and M. Franklin. Return address prediction in speculative multithreaded environments. In *Proc. Int'l Conference on Hi-Performance Computing (HiPC)*, 2002.