Correct Alignment of a Return-Address-Stack after Call and Return Mispredictions

Veerle Desmet* Yiannakis Sazeides+ Constantinos Kourouyiannis+ Koen De Bosschere*

*Dept. of Electronics and Information Systems
Ghent University, Belgium
veerle.desmet@elis.UGent.be
koen.debosschere@elis.UGent.be

+Department of Computer Science
University of Cyprus, Nicosia
yanos@cs.ucy.ac.cy
cs01kc1@cs.ucy.ac.cy

Abstract

It is accepted that the alignment of a return-address-stack can be preserved by checkpointing on branches and recovering on branch mispredictions the top-of-stack pointer. However, virtually all published literature, is vague or proposes sub-optimal algorithms on how to repair the alignment of a return-address-stack on call and return mispredictions. Possible manifestations of this ambiguity are lower performing implementations of return-address-stack mechanisms in processors and simulators. This paper suggests that on call and return mispredictions the recovered top-of-stack pointer should point to the next and previous stack position respectively relative to the checkpointed top-of-stack pointer.

It is established experimentally that the proposed method can improve the performance, over an otherwise identical processor configuration, on the average by 2% and by as much as 10%. This corresponds to a drop in return-address mispredictions by 40%, from 11.3% downto 6.7%. This work also considers whether the performance of one of the best known return-address-stack methods, proposed several years ago, is satisfactory with increasing pipeline depth. The results show that the average performance of the method is still satisfactory, but for some benchmarks appears insufficient. This may suggest a need for better return-address predictors.

1. Introduction

Deeply-pipelined superscalar processors rely on an aggressive fetch mechanism to provide several useful instructions every cycle. Central to any modern fetch mechanism is a branch predictor that can quickly provide the target address of a branch. Performance can be improved by fetching instructions from a predicted path and entering them speculatively in the pipeline. However, to maintain correctness, predicted instructions need to be verified against their actual target address. This verification is typically performed several cycles after the prediction, when the actual branch target is known.

In most cases predictions are correct and speculative work is useful. Nevertheless, mispredictions occur and speculation inevitably results in executing some instructions along a mispredicted path. These misspeculated instructions affect the processor state, including registers, caches and branch prediction structures, in a variety of ways [7]. Once a misprediction is detected, it is sufficient for correctness to recover the architectural state as if no mis-speculated instruction was executed [6, 14]. It is also useful, for better performance, to recover some microarchitectural state to limit the detrimental effects of wrong path updates. Recovery is facilitated by checkpointing, at the time of a branch prediction, the information that needs to be recovered in the case of a misprediction [6, 7, 14].

Branch prediction structures are probably the most critical microarchitectural structures that need recovery following a misprediction [3, 7, 13, 19]. High performance processors employ a variety of branch predictors each aimed at a different type of branch and/or branch behavior. As a consequence, different checkpoint/recovery strategies may be required depending on the branch type and branch predictor.

This paper is concerned with return-address predictors. Return instructions are executed frequently in programs to steer the control flow at the end of routines and several predictors have been proposed to

predict the return address of a routine [2, 4, 5, 7, 8, 9, 10, 13, 15, 16, 18]. Most of these predictors rely on a return-address-stack (RAS) that is accessed using a top-of-stack pointer (TOS) [8]. Call instructions push addresses and return instructions pop them. Such an algorithm ensures, with an infinite RAS, that any balanced sequence of pushes and pops will lead always to correct return address prediction.[1] Alas, mis-speculation can lead to incorrect updates of the TOS and the RAS content [13]. Incorrect updates of the TOS may result in a RAS that is not-aligned, i.e. the TOS points to a wrong entry in the RAS, whereas incorrect updates of RAS content may lead to a corrupted RAS content. Therefore, most schemes employ a repair mechanism to checkpoint parts of the RAS on each branch instruction, so that when a branch misprediction is detected the RAS alignment and/or content is restored.

Most published work about return-address predictors clearly states the procedure for recovering the alignment from a conditional branch misprediction. However, almost all previous literature, with the exception of one patent [18], is vague or proposes low-performing algorithms for recovering the alignment of a return-address predictor after a call/return misprediction. Possible manifestations of this ambiguity are low-performing implementations of return-address predictors in processors and simulators. Although, we have no evidence for the former, for the latter we have examined several public domain processor simulators and we have found some that implement the RAS alignment recovery sub-optimally.

This paper presents a method for preserving alignment of a RAS after call and return mispredictions. Specifically, it is suggested that on call and return mispredictions the recovered TOS should point to the next and previous stack position respectively relative to the checkpointed TOS pointer. This is the same recovery method as the one proposed by [18]. However, our paper quantifies, as far as we know for the first time, the performance implications with and without this alignment method. The paper also investigates whether one of the best known return-address predictors, proposed several years ago, still provides satisfactory performance with increasing pipeline depth size.

The remainder of this paper is organized as follows. Section 2 further motivates this paper and precisely describes how to correct the alignment of a RAS after a call and return misprediction. After discussing related work in Section 3, Section 4 details on the simulation methodology. Section 5 evaluates the performance impact of correct–alignment and considers its significance in combination with an uncorruption technique. Section 5, also examines the effect of pipeline depth on the RAS misprediction rate. Section 6 concludes the paper.

[1] setjmps and longjumps lead to unbalanced push and pop sequences, and can result in return mispredictions.

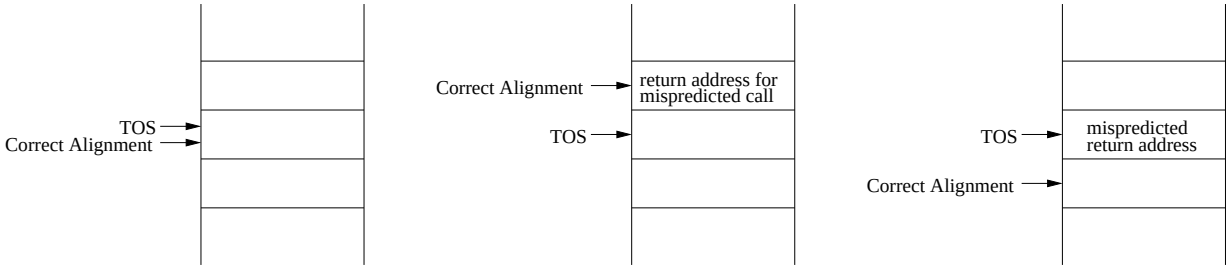
2. Correct–Alignment after Call and Return Mispredictions

Let us assume a RAS-based return-address predictor with a repair mechanism that checkpoints the TOS before executing each branch. Figure 1 illustrates for different types of branches where a checkpointed TOS points when its corresponding branch gets mispredicted.

For a conditional branch, and other branch types that do not update the RAS, the checkpointed TOS is aligned. I.e. it points exactly to the position it was before the mispredicted branch entered the pipeline, thus as if no wrong path TOS updates were performed. For a mispredicted call, however, the checkpointed TOS is not aligned because a call correctly pushes its return address in the RAS even if its target address is mispredicted. Therefore, a mispredicted call should repair the TOS to point to the next position of the checkpointed TOS. I.e. to point to the mispredicted call's return address so that when the return, corresponding to the call, is predicted it can get the right return address. Similarly, if a return is mispredicted—actual return address does not correspond to the predicted address popped from RAS—the restored TOS should point to the position previous to the mispredicted return address and not to the checkpointed TOS. This aims to prevent a subsequent return prediction to get the same address as a previously executed and mispredicted return.

Henceforth, we refer to the above method that keeps the RAS aligned as *correct–alignment* and the not-aligned scheme as *incorrect–alignment*. We note that the not-aligned approach is not incorrect but may be lower performing in the case of call and return mispredictions. It may be useful to also note that the restoration of the correct alignment, after a misprediction, is not sufficient for correct prediction since the stack's content may have been corrupted by mis-speculated calls.

We describe two ways for implementing *correct–alignment*. The first checkpoints the TOS before a branch enters the pipeline, and depending on the type of a mispredicted branch it recovers the TOS as discussed above: (i) to the checkpointed position for branches that do not affect the RAS, (ii) to the position next to the checkpointed TOS for a call mis-



(a) Mispredicted conditional branch

(b) Mispredicted call

(c) Mispredicted return

Figure 1. Correct–alignment. TOS = top-of-stack before the branch enters the pipeline. The recovered top-of-stack should point to: the initial position after a mispredicted conditional branch (left); the next position after a mispredicted call (center); the previous position after a mispredicted return (right).

prediction, (iii) to the position previous to the checkpointed TOS for a return misprediction. Alternatively, we can checkpoint the TOS after a branch has speculatively updated the predictor structures and possibly the RAS, and later simply restore the TOS to its checkpointed position. The first method is the one suggested by [18] and probably it is easier to implement because it does not require knowing the type of a branch prior to checkpointing.

We have examined several public domain processor simulators and we have found the widely used SimpleScalar simulator [1] and many of its derivatives to implement the *incorrect–alignment*. This may indicate that previous work may have drawn inaccurate conclusions by understating or overstating the importance of mechanisms that their significance is influenced by the RAS performance. In Appendix A we describe how to implement the correct–alignment of a RAS in SimpleScalar. We note that HydraScalar [12] used in the seminal work on return-address prediction by Skadron *et al.* [13] model a RAS with correct–alignment.

In the remainder of this paper we attempt to quantify empirically the importance of correct over incorrect–alignment.

3. Related work

In this section we discuss previously proposed return-address prediction techniques. These works are categorized into RAS for performance and RAS for security and reliability. The fundamental difference between these two categories is that the former does not have to guarantee the semantic correctness of the predicted return address whereas the latter does. The discussion is mainly concerned with the branch misprediction recovery strategy proposed by each of the methods.

3.1. Return-Address-Stack for performance

Webb [16] pioneered a call/return stack that prevents stack desynchronization due to false subroutine return addresses by tagging each potential return address with its associated call target.

Kaeli and Emma [8] explored a call/return detection scheme with two stacks to solve target mispredictions of moving target branches due to subroutine returns.

Steely *et al.* [15] use a ring buffer structure to provide return addresses. A ring pointer counter provides a backup for recovering after a misprediction.

Eickemeyer [2] presents a call-return-stack and tracks the number of calls in progress by a prediction counter and an update counter. Whenever a misprediction is detected, the update counter is copied into the prediction counter. The recovery method proposed in this work can preserve alignment after mispredicted conditional and call branches, but is unclear if it can after return mispredictions.

Hoyt *et al.* [4] claim a method with several pointer checkpoints for recovering from branch mispredictions as soon as possible. Although different actions are considered for verifying different branch types, it is not clear that correct–alignment is preserved on call/return mispredictions.

McMahan [10] uses distinct global stack pointers for three pipeline stages in order to repair the TOS af-

ter mispredicted branches. The difference between the handling of conditional and call/return mispredictions is not considered.

Jourdan *et al.* [7] proposed a sophisticated list-like return prediction mechanism, which is capable of eliminating return address corruption while on the misspeculated path. Their scheme as described can suffer from *incorrect–alignment*.

Skadron *et al.* [13] evaluate a repair mechanism for a RAS that restores the TOS pointer and the TOS content. This mechanism eliminates almost all effects of stack corruption. However, this work did not consider the effects of pipeline depth on RAS accuracy.

An Intel patent by Yeh [18] proposes a mechanism that restores the TOS depending on the branch misprediction type. The recovery algorithm implements *correct–alignment*. However, it does not provide a quantification of the importance of *correct–alignment*.

McDonald [9] describes a roll-back mechanism to recover from a corrupted RAS when multiple calls and/or returns have been speculatively executed prior to resolving a mispredicted conditional branch.

Hummel *et al.* [5] introduce a return address stack mechanism where reads/writes are done at a fixed position while adjusting the entire stack content up or down. This alternative stack mechanism also tracks the relative number of call/return movements in order to ensure correct recovery.

We note that almost all of the above proposals did not discuss recovery from call and return mispredictions.

3.2. Return-Address-Stacks for Reliability

Park and Lee [11] describe a technique for RAS overflow protection, and Ye and Kaeli [17] introduce a reliable return address stack for security purposes. The aim of these works is to detect and recover from stack smashing attacks, and therefore they rely on an (additional) uncompromised RAS which is maintained by updating the RAS during commit. Due to their different nature, these defense techniques cannot benefit from *correct–alignment*.

4. Methodology

We use SimpleScalar's cycle accurate simulator *sim-outorder* to implement both correct and incorrect–alignment. The fix required to model *correct–alignment* in SimpleScalar is described in Appendix A. Table 1 summarizes the parameters for our baseline setup.

Parameter description	Setting
RAS	32 entries
Pipeline depth	20 stages
Instruction-window	128
Fetch/Decode/Issue/ Commit width	up to 4 instructions per cycle
Functional Units	4 INT ALU's, 1 INT mult/div, 4 FP ALU's, 1 FP mult/div
Memory ports	2
Branch Predictor	hybrid: 16 KiB meta, 16 KiB bimodal, 32 KiB gshare (16 bits history)
BTB	2048-entry, 2-way
Misprediction penalty	2 cycles on misfetch
L1 data-cache	8 KiB, 4-way, 64 B blocks, LRU, 2-cycle latency
L1 instruction-cache	16KiB, 2-way, 64B blocks, LRU, 2-cycle latency
L2	unified: 2 MiB, 8-way, 256B blocks, LRU, 7-cycle latency

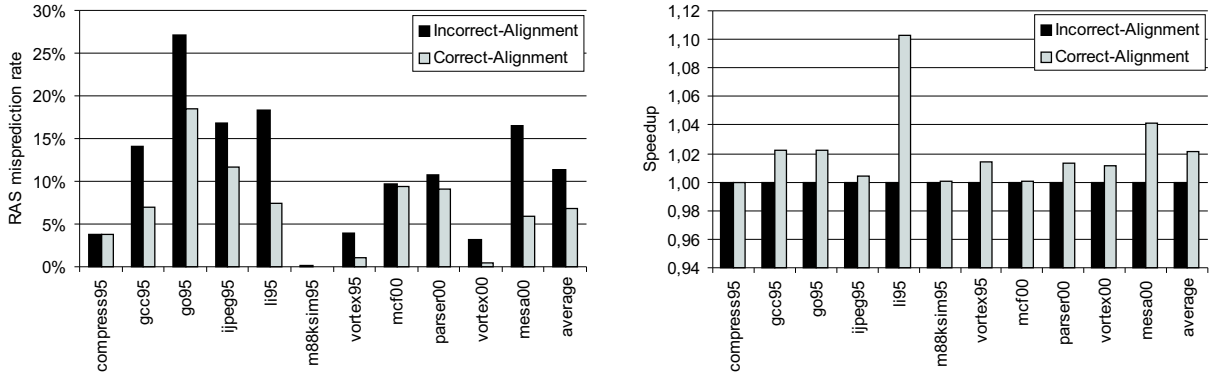
Table 1. Baseline configuration.

For representative return-address behavior we selected the benchmarks from SPEC95 and SPEC2000 with more than one-million committed returns. All benchmarks were compiled using the *gcc* compiler version 2.6.3 with optimizations flag `-O3`. For the SPEC95 programs we consider complete execution with train inputs, while for SPEC2000 we measure for a fixed number of committed instructions after fast-forwarding using reference inputs. Table 2 shows the basic statistics of the benchmarks used in this study.

5. Evaluation

In this section, we first evaluate the effect of *correct–alignment* and then consider the effects of pipeline depth on RAS performance.

The performance metric used to compare different RAS mechanisms is RAS misprediction rate, i.e. the number of incorrectly predicted return address predictions per committed return. Moreover, we provide speedup—measured in IPC (Instructions Per Cycle) improvement over a baseline configuration—to stress the importance of *correct–alignment* on overall performance.



(a) Return address misprediction rate

(b) Speedup over incorrect–alignment

Figure 2. Correct–alignment versus incorrect–alignment.

Benchmark	Fast fwd	Comm Inst	Call	Ret	Call Misp	Ret Misp
compress95	0	443	2.8	2.8	0.0	3.8
gcc95	0	177	1.4	1.4	0.5	6.9
go95	0	133	1.0	1.0	0.0	18.5
ijpeg95	0	553	0.2	0.2	1.4	11.7
li95	0	202	3.0	2.9	2.3	7.4
m88ksim95	0	241	4.0	4.0	0.0	0.1
vortex95	0	101	2.0	2.0	0.0	1.0
mcf00	2,000	100	1.3	1.3	0.0	9.5
parser00	400	100	2.5	2.5	0.0	9.1
vortex00	100	100	2.0	2.0	0.0	0.4
mesa00	350	350	1.2	1.2	0.0	5.8
average			1.9	1.9	0.4	6.7

Table 2. Benchmark summary: number of fast-forwarded and committed instructions in millions, fraction of instructions that are calls and returns, and the call/return misprediction rates in the baseline setup with correct–alignment.

5.1. Correct–Alignment

Figure 2(a) compares the RAS misprediction rate between incorrect–alignment and correct–alignment for the different benchmarks. Obviously, correct–alignment is better over all benchmarks. Recall that this improvement is without any apparent additional hardware cost. On average, the misprediction rate for return-addresses is decreased from 11.3% downto 6.7%, or a reduction by 40%. In terms of speedup (see Figure 2(b)), *correct–alignment* improves overall performance by 2% on average, and up to 10% for *li95*.

Thus far, we assumed a repair mechanism that recovers only the TOS. Skadron *et al.* [13] demonstrate that the additional checkpointing of the TOS content eliminates almost all return address mispredictions because it solves some forms of stack corruption.

Figure 3 illustrates that uncorruption (rightmost bar for each benchmark), as proposed by Skadron *et al.*, is effective for decreasing misprediction rate and increasing speedup. Misprediction rate is reduced from 6.7% downto 1.4% on average. Figure 3 also shows that the potential of the RAS content repair mechanism can be understated significantly, often by more than 10%, when the stack is incorrectly-aligned. Specifically, Skadron's content-recovery reduces by 63% the mispredictions in a processor with *incorrect–alignment*, whereas the reduction is 78% with *correct–alignment*.

5.2. Call-Uncorruption Optimization

An additional optimization that we considered, on top of correct–alignment, is on a call misprediction to re-write the call's return address into the RAS entry pointed by the recovered TOS. This way, we ensure that after a call misprediction the return address is uncorrupted. Importantly, the *call-uncorruption* optimization is available without additional checkpointing hardware overhead since the return address for a call can easily be computed using the call's address.

Simulations show, however, that the call-uncorruption optimization does not affect the misprediction rate (only *gcc95* and *li95* have a slight improvement) since RAS corruptions after a call misprediction are rarely observed in our setup. A possible reason for this, is the relatively small refetch

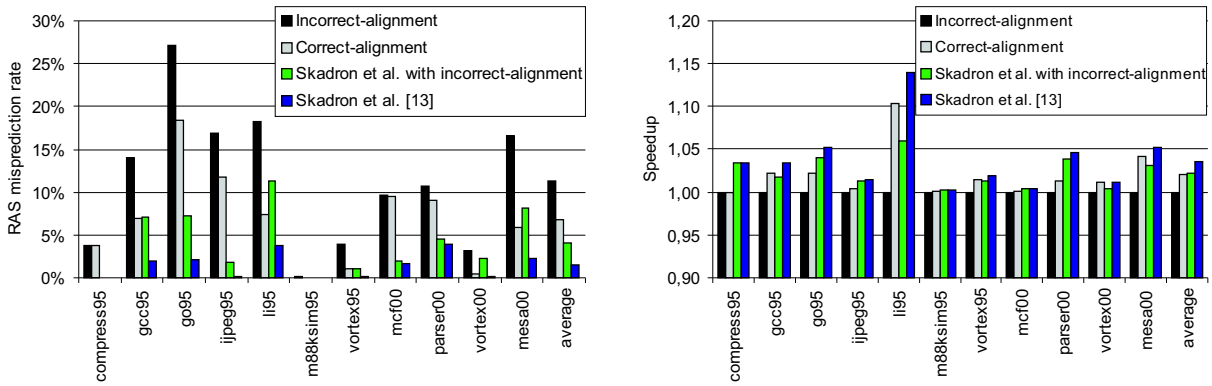


Figure 3. Other repair mechanisms: uncorruption can be understated if the RAS is not correctly-aligned. Left: return address misprediction rate. Right: Speedup over incorrect–alignment.

penalty for direct call mispredictions. In a different setup with larger refetch penalty, the call-uncorruption optimization might be useful for restoring part of the RAS content.

5.3. Effects of Deeper Pipelines

One of the best performing return-address predictors was proposed in [13]. However, the evaluation of that predictor did not consider the effects of increasing pipeline depth which has grown considerably in recent years. As pipelines deepen, the penalty associated with a branch misprediction increases and more (mis-)speculative instructions—including branches—enter the pipeline. This increase in mis-speculated instructions per misprediction may corrupt more severely the content of a RAS. Figure 4 presents the effects of the pipeline depth and RAS size for the repair mechanism in [13].

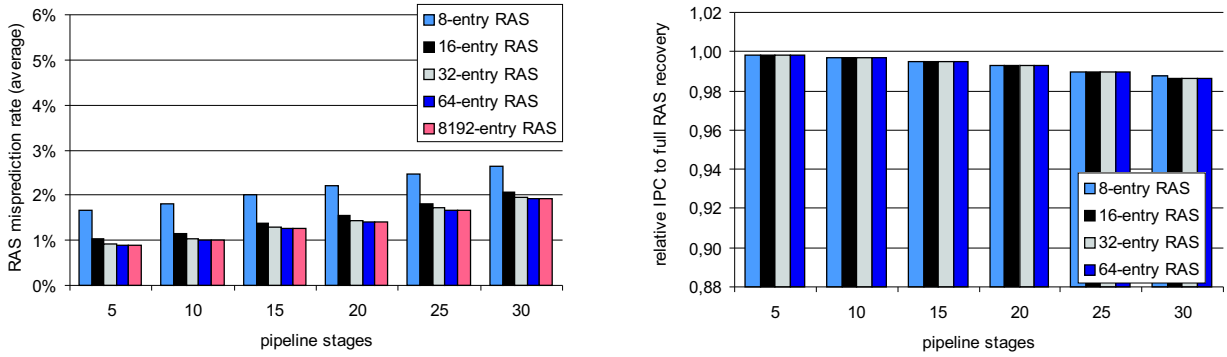
Fig. 4(a) shows the average RAS misprediction rate and the IPC relative to a scheme with full RAS content checkpointing/recovery. The results show that these two metrics get slightly worse with deeper pipelines. This behavior is observed irrespective of the RAS size. This may be perceived as satisfactory, however, a closer examination, shown in Figure 4(b), reveals that for some benchmarks, such as *li95* and *parser00*, the RAS misprediction rate increases significantly with deeper pipelines and the IPC relative to full RAS recovery drops seriously. Note that the results in Figure 4(b) were obtained using a 32-entry RAS.

Overall the results in Figure 4 suggest that for some benchmarks with increasing pipeline depth a RAS suffers from more corruption and that there is a need for more efficient RAS content recovery mechanisms.

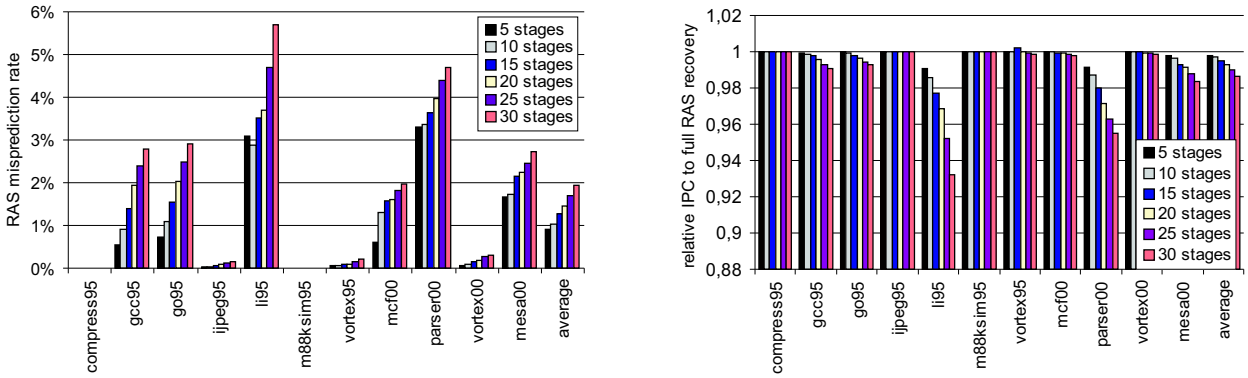
Impact of Pipeline Depth on RAS Corruption

The remainder of this section analyzes the impact of pipeline depth on RAS corruption. More specifically, after each misprediction recovery we determine the distance to the RAS entries, relative to the recovered TOS, that are corrupted. These RAS corruptions are detected by comparing the speculatively updated RAS against an uncorrupted RAS. Figure 5 shows the distribution of the corruption distances for a RAS with 32 entries. A corruption distance of 0, i.e. the TOS data is corrupted, is denoted as *tos*. The largest possible corruption distance with a 32 entry RAS is 31. It is important to note that the RAS entries at distance 1 and 31 are the two neighboring entries of the recovered TOS in the backward and forward direction respectively.

The data in Fig. 5(a) show that the average number of RAS corruptions per kilo instructions increases with deeper pipelines but not the corruption distance. Clearly, corruption is concentrated around the recovered TOS, both in the backward and forward direction. This concentration of corruption around the TOS suggests that the corruption in the backward direction occurs mainly when there are more pops than pushes in the mis-speculated path, whereas the corruption in the forward direction is due to more pushes than pops. An important difference between forward and backward corruption is that the forward corruption is usually not detrimental to performance because correct path execution will overwrite the forward corrupted entries with useful information. In contrast, backward RAS corrup-



(a) Average influence of RAS size



(b) Per benchmark behavior for a 32-entry RAS

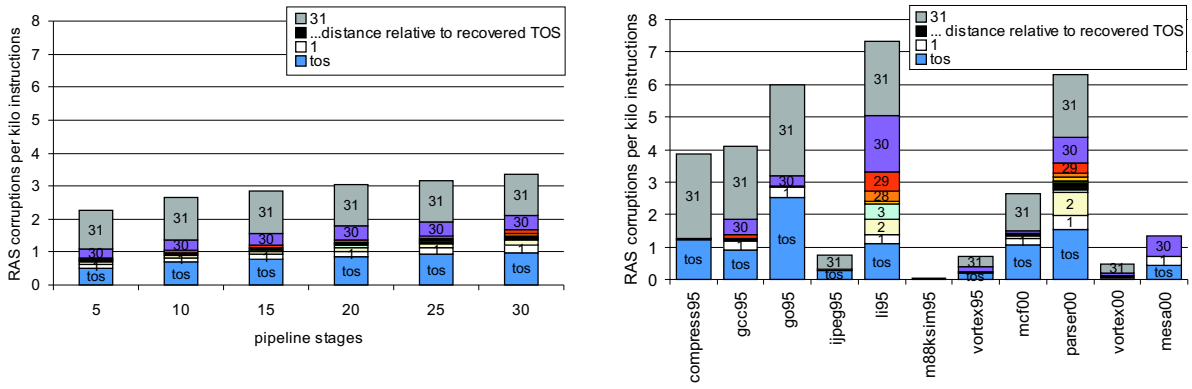
Figure 4. Influence of deeper pipelines.

tion can not be cured unless the corrupted RAS entries are restored. The above indicates that the evaluated RAS predictor [13] can recover most of the backward corruption because it is capable of restoring the TOS content.

Fig. 5(b) presents the breakdown of RAS corruption distance with a 20-stage pipeline for all benchmarks. For most programs the corruption behavior is similar to the average shown in Fig. 5(a). However, *li95* and *parser00* have substantial corruption at larger distances. This illuminates why only recovering the TOS content is not sufficient for these two benchmarks and thus the large performance degradation observed in Fig. 4.

6. Conclusion

This paper presents a method for preserving the alignment of a RAS after different types of branch mispredictions. Specifically it is argued that if the TOS is checkpointed before each branch then the recovered TOS should point to: (i) its checkpointed position after a mispredicted conditional or other branch type that does not update the RAS; (ii) the next position after a mispredicted call; (iii) the previous position after a mispredicted return. We demonstrate that correct–alignment improves average speedup by 2%, and up to 10%. It is also shown that the significance of a previously proposed RAS repair mechanism can be understated significantly, often by more than 10%, when the stack is incorrectly aligned. This, demonstrates that by relying on an incorrectly–aligned RAS, previ-



(a) Average influence of pipeline depth

(b) Per benchmark behavior for a 20-stage pipeline

Figure 5. RAS corruptions per kilo instructions: break down for a 32-entry RAS. Label 'x' marks the contribution of RAS corruptions at distance 'x' relative to the recovered TOS.

ous work may have drawn inaccurate conclusions. We further propose an optimized recovery scheme—call-uncorruption—which is able to eliminate some RAS corruptions without requiring additional hardware. Finally, we evaluate the effect of increasing pipeline depth on one of the best known RAS predictors and conclude that on average its performance is satisfactory, but not for all programs. This may indicate a need for new return-address predictors that can recover more content after misprediction.

Acknowledgments

Veerle Desmet is supported by a grant from the Flemish Institute for the Promotion of the Scientific-Technological Research in the Industry (IWT) and by the Fund for Scientific Research-Flanders (FWO). Yiannakis Sazeides research activity is supported by Intel, University of Cyprus and HiPEAC.

Appendix A

To preserve correct–alignment in SimpleScalar [1] as well as in some derived simulators, a small fix is needed in *bpred.c* in the function *bpred_lookup()* as indicated in Figure 6. We like to note that this code purely models correct–alignment as stated in section 2. It provides neither top-of-stack content repair nor the call-uncorruption optimization discussed in section 5.2.

```
  /* record pre-pop TOS; if this branch is executed speculatively
   * and is squashed, we'll restore the TOS and hope the data
   * wasn't corrupted in the meantime. */
  if (pred->retstack.size)
    *stack_recover_idx = pred->retstack.tos;
  else
    *stack_recover_idx = 0;

  /* if this is a return, pop return-address stack */
  if (is_return && pred->retstack.size)
    {
      md_addr_t target = pred->retstack.stack[pred->retstack.tos].target;
      pred->retstack.tos = (pred->retstack.tos + pred->retstack.size - 1)
                           % pred->retstack.size;
      pred->retstack_pops++;
      dir_update_ptr->dir.ras = TRUE; /* using RAS here */
      *stack_recover_idx = pred->retstack.tos; /* Correct-Alignment */
      return target;
    }

#ifndef RAS_BUG_COMPATIBLE
  /* if function call, push return-address onto return-address stack */
  if (is_call && pred->retstack.size)
    {
      pred->retstack.tos = (pred->retstack.tos + 1) % pred->retstack.size;
      pred->retstack.stack[pred->retstack.tos].target =
         baddr + sizeof(md_inst_t);
      pred->retstack_pushes++;
      *stack_recover_idx = pred->retstack.tos; /* Correct-Alignment */
    }
#endif /* !RAS_BUG_COMPATIBLE */
```

Figure 6. Changes needed in SimpleScalar's bpred_lookup() to preserve correct–alignment.

References

[1] D. Burger, T. M. Austin, and S. Bennett. Evaluating future microprocessors: The SimpleScalar Tool Set. Technical report, Computer Sciences Department, University of Wisconsin-Madison, July 1996.

[2] J. Eickemeyer. International Business Machines Corporation. Computer system branch prediction of subroutine returns. *United State Patent Number 5,313,634*. May 1994.

[3] E. Hao, P.-Y. Chang, and Y. N. Patt. The effect of speculatively updating branch history on branch prediction accuracy, revisited. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 228–232, Nov. 1994.

[4] B. D. Hoyt, G. J. Hinton, D. B. Papworth, A. K. Gupta, M. A. Fetterman, S. Natarajan, S. Shenoy, and R. V. D'Sa. Intel Corporation. Method and apparatus for implementing a four stage branch resolution system in a computer processor. *European Patent Number 661,625*. July 1995.

[5] V. E. Hummel and H. Sharangpani. Intel Corporation. Return register stack target predictor. *United State Patent Number 6,560,696*. May 2003.

[6] W.-M. W. Hwu and Y. N. Patt. Checkpoint repair for high-performance out-of-order execution machines. *IEEE Transactions on Computers*, 36(12):1496–1514, Dec. 1987.

[7] S. Jourdan, J. Stark, T.-H. Hsing, and Y. N. Patt. The effects of mispredicted-path execution on branch prediction structures. In *Proceedings of the 5th International Conference on Parallel Architectures and Compilation Techniques*, pages 58–67, Oct. 1996.

[8] D. R. Kaeli and P. G. Emma. Branch history table prediction of moving target branches due to subroutine returns. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 34–42, May 1991.

[9] T. C. McDonald. IP-First, LLC. Method and apparatus for correcting an internal call/return stack in a microprocessor that speculatively executes call and return instrcutions. *United State Patent Number 6,314,514*. Nov. 2001.

[10] S. C. McMahan. Cyrix Corporation. Branch processing unit with a return stack including repair using pointers from different pipe stages. *United State Patent Number 5,706,491*. Jan. 1998.

[11] Y.-J. Park and G. Lee. Repairing return address stack for buffer overflow protection. In *Proceedings of the first conference on Computing Frontiers*, pages 335–342, Apr. 2004.

[12] K. Skadron and P. S. Ahuja. HydraScalar: A multipath-capable simulator. *Newsletter of the IEEE Technical Committee on Computer Architecture*, Jan. 2001.

[13] K. Skadron, P. S. Ahuja, M. Martonosi, and D. W. Clark. Improving prediction for procedure returns with return-address-stack repair mechanisms. In *Proceedings of the 31st Annual International Symposium on Microarchitecture*, pages 259–271, Nov. 1998.

[14] J. E. Smith and A. R. Pleszkun. Implementing precise interrupts in pipelined processors. *IEEE Transactions on Computers*, 37(5):562–573, May 1988.

[15] S. C. Steely and D. J. Sager. Digital Equipment Corporation. Subroutine return prediction mechanism using ring buffer and comparing predicated address with actual address to validate or flush the pipeline. *Unites States Patent Number 5,179,673*. Jan. 1993.

[16] C. F. Webb. Subroutine call/return stack. *IBM Technical Disclosure Bulletin*, 30(11):221–225, Apr. 1988.

[17] D. Ye and D. Kaeli. A reliable return address stack: microarchitectural features to defeat stack smashing. *ACM SIGARCH Computer Architecture News*, 33(1):73–80, Mar. 2005.

[18] T.-Y. Yeh. Intel Corporation. Return address predictor that uses branch instructions to track a last valid return address. *United State Patent Number 6,253,315*. June 2001.

[19] T.-Y. Yeh and Y. N. Patt. A comprehensive instruction fetch mechanism for a processor supporting speculative execution. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 129–139, Nov. 1992.