Speculative Return Address Stack Management Revisited

HANS VANDIERENDONCK
Ghent University
and
ANDRÉ SEZNEC
IRISA/INRIA

Branch prediction feeds a speculative execution processor core with instructions. Branch mispredictions are inevitable and have negative effects on performance and energy consumption. With the advent of highly accurate conditional branch predictors, nonconditional branch instructions are gaining importance.

In this article, we address the prediction of procedure returns. On modern processors, procedure returns are predicted through a return address stack (RAS). The overwhelming majority of the return mispredictions are due to RAS overflows and/or overwriting the top entries of the RAS on a mispredicted path. These sources of misprediction were addressed by previously proposed speculative return address stacks [Jourdan et al. 1996; Skadron et al. 1998]. However, the remaining misprediction rate of these RAS designs is still significant when compared to state-of-the-art conditional predictors.

We present two low-cost corruption detectors for RAS predictors. They detect RAS overflows and wrong path corruption with 100% coverage. As a consequence, when such a corruption is detected, another source can be used for predicting the return. On processors featuring a branch target buffer (BTB), this BTB can be used as a free backup predictor for predicting returns when corruption is detected.

Our experiments show that our proposal can be used to improve the behavior of all previously proposed speculative RASs. For instance, without any specific management of the speculative states on the RAS, an 8-entry BTB-backed up RAS achieves the same performance level as a state-of-the-art, but complex, 64-entry self-checkpointing RAS [Jourdan et al. 1996]. Therefore, our proposal can be used either to improve the performance of the processor or to reduce its hardware complexity.

Categories and Subject Descriptors: B.1 [**Hardware**]: Control Structures and Microprogramming; C.1.1 [**Processor Architectures**]: Single Data Stream Architectures

General Terms: Algorithms, Experimentation, Performance

Additional Key Words and Phrases: Return address prediction, corruption detection, back-up predictor

Authors' addresses: H. Vandierendonck, Universiteit Gent, Vakgroep ELIS, Sint-Pietersnieuwstraat 41, B-9000 Gent, Belgium; A. Seznec, IRISA, Campus de Beaulieu, 35042 RENNES Cedex, France.

ACM Reference Format:
Vandierendonck, H. and Seznec, A. 2008. Speculative return address stack management revisited. ACM. Trans. Architec. Code Optim. 5, 3, Article 15 (November 2008), 20 pages. DOI = 10.1145/1455650.1455654 http://doi.acm.org/10.1145/1455650.1455654

1. INTRODUCTION

Processors with speculative execution rely on branch prediction to fetch useful instructions long before the correct branch targets are computed. Different types of branches exist—conditional, indirect and returns—and each is predicted using custom techniques. With the advent of highly accurate conditional branch predictors [Seznec 2005; Seznec and Michaud 2006; Jiménez 2005], nonconditional branch instructions are gaining importance.

Return addresses in particular are predicted by means of a return address stack (RAS). A RAS keeps track of pairs of call and return instructions [Kaeli and Emma 1991; Webb 1988]. The rationale is that a procedure may be called from multiple sites in a program. However, each time a return instruction is executed, it matches with one particular call instruction. Thus, a call instruction pushes a return target on a RAS, and the corresponding return instruction pops its predicted target off the stack. The RAS is typically implemented as a circular buffer to store return addresses and a top-of-stack pointer (TOS) pointing to the current top of the stack [Skadron et al. 1998].

In theory, a RAS can predict return targets with a 100% accuracy, were it not for two phenomena. First, the RAS may be too small to hold the entire call stack. When the RAS overflows, the TOS pointer wraps around and overwrites older RAS entries. Although the next few returns are predicted correctly, mispredictions will occur when the TOS wraps back to the overwritten entries. A second source of mispredictions results from speculative execution [Jourdan et al. 1996; Skadron et al. 1998]. The RAS must be updated for speculatively fetched instructions. A speculatively fetched call pushes a return target on the RAS to allow its corresponding return to pick it up. When a misprediction is detected, the call instruction may be squashed, leaving the wrong return target on the RAS. As a consequence, one branch misprediction may induce additional return address mispredictions.

These problems have been studied in the literature, but the proposed solutions increase the RAS design complexity. RAS overflows can only be avoided by increasing the RAS size, but cycle time constraints set an upper bound to a practical RAS size. RAS corruption due to speculative execution can be reduced (e.g., by checkpointing the top RAS entry together with the TOS) [Skadron et al. 1998], but the other RAS entries remain vulnerable to corruption and are responsible for a large number of mispredictions [Desmet et al. 2005]. Alternatively, the self-checkpointing RAS of Jourdan et al. [1996] avoids corruption due to speculative execution but requires a considerably larger RAS to avoid corruption due to overflow.

In this article, we present simple confidence estimators for a RAS, more precisely corruption detectors. Then, we show that these estimators can be

used to select an alternate source (i.e., the branch target buffer (BTB) or the indirect jump predictor which are already present in the microarchitecture) for return prediction. This allows to improve the return address prediction accuracy and/or reduce the hardware complexity of the RAS.

The sources of mispredictions on the RAS are well known. Our main contribution is to propose two low-complexity hardware mechanisms that detect the RAS overflows and the wrong path speculative overwrites of the top entries of the RAS, respectively. These mechanisms have 100% coverage of RAS corruption. When RAS corruption is detected, the prediction can be computed by a backup predictor, in our case the BTB, but also possibly the indirect jump predictor. Thus, the backup predictor is effectively for free and the added design complexity for the RAS is limited to the corruption detectors.

Our BTB-backup RAS allows to increase the performance of all previously proposed RAS designs. Our experiments also show that augmenting any RAS design with a BTB back-up is much more cost-effective than increasing the size of a state-of-the-art conditional branch predictor. Moreover, when a simple 8-entry RAS design is augmented with BTB-backup, it reaches the same performance level as was obtained by a state-of-the-art but complex 64-entry self-checkpointing RAS [Jourdan et al. 1996]. This complexity reduction could be leveraged in SMT processors where a RAS predictor is needed for each thread [Hily and Seznec 1996], and of course in multicores.

In the remainder of this article, we first present our evaluation framework and motivation for the study. Then, we discuss related work (Section 3), present the corruption detectors for the RAS (Section 4) and the backup prediction scheme (Section 5). Then, we evaluate the corruption detectors and the backup predictor (Section 6). Section 7 concludes this article.

2. EVALUATION FRAMEWORK AND MOTIVATION

2.1 Simulation Framework

The RAS corruption detectors are evaluated in a 4-issue processor model (Table I). A 96-entry reorder buffer is used. Long instruction sequences may be speculatively fetched. This may cause significant RAS corruption. The conditional branch predictor is a 64 KB O-GEHL [Seznec 2005] and indirect branches are predicted by a cascaded predictor with room for 256 branch targets [Driesen and Holzle 1998]. The branch misprediction penalty is 20 cycles. This processor model is implemented in the sim-flex simulator (`http://www.ece.cmu.edu/~simflex/`).

2.2 Benchmark Selection and Motivation

Accurate return prediction is a major issue on only a subset of the applications—applications featuring a significant ratio of procedure calls. This class of applications is, however, a very large class.

Therefore, as a benchmark set, we select the SPEC CPU2000 integer benchmarks that execute a significant fraction of return instructions (i.e, 7 out of 12 SPEC CPU2000 integer applications) as well as a selection of the TPC-D

Table I. Baseline Processor Model

Processor Core	
Issue width	4 instructions
ROB, issue queue	96
Load-store queue	64
Dispatch-execute delay	5 cycles
Fetch Unit	
Fetch width	4 instructions, 2 branches/cycle
Instruction fetch queue	16 instructions
Fetch-dispatch delay	9 cycles
Cond. branch predictor	64Kbits O-GEHL
Return address stack	32 entries
Branch target buffer	256 sets, 4 ways
Cascaded branch target predictor	64 sets, 4 ways 8-branch path history
Memory Hierarchy	
L1 I/D caches	64KB, 4-way, 64B blocks
L2 unified cache	256KB, 8-way, 64B blocks
L3 unified cache	4MB, 8-way, 64B blocks
Cache latencies	1 (L1), 6 (L2), 20 (L3)
Memory latency	350 cycles

queries. These queries are executed by the postgres v6.3 database engine on a 100MB B-Tree indexed database. Only a subset of the TPC-D queries are presented as the other queries behave similar to the presented ones.

The benchmarks are compiled for the Alpha ISA using the native cc compiler with optimization flags "-fast" and are statically linked. Representative simulation intervals of 500M instructions are determined using SimPoint [Sherwood et al. 2002].

To point out the importance of accurately predicting returns on these applications, we summarized the types of control transfers and their respective prediction accuracies for the baseline processor model with a 32-entry simple RAS in Table II. On the benchmark set, return instructions constitute 1% to 3% of the executed instructions. When using a 32-entry simple RAS, return mispredictions constitute an important fraction of all branch mispredictions. For the whole benchmark suite, return mispredictions constitute 32% of all branch mispredictions. Note that if the BTB was used as the return predictor, the number of return mispredictions would be huge (7.68 misp/KI). Furthermore, we will see in Section 6 that,when using current state-of-the-art RAS designs [Jourdan et al. 1996; Skadron et al. 1998] the RAS misprediction rate still represents a significant part of the overall branch misprediction rate.

3. RELATED WORK

The earliest reference on return address prediction goes back to Webb [Webb 1988] who explicitly linked return addresses to their corresponding call instructions. Kaeli and Emma improve a branch target predictor with a two-stack return address predictor [Kaeli and Emma 1991].

Table II. Characterization of Benchmarks Executing on the Baseline Processor Model

Bench-mark	IPC	Conditional		Direct	Indirect		Returns		
		Freq.	MPKI	Freq.	Freq.	MPKI	Freq.	MPKI	BTB
crafty	2.25	8.2%	3.42	1.4%	0.2%	0.83	1.0%	1.14	4.56
eon	2.20	4.4%	2.30	2.1%	0.6%	1.30	2.2%	1.46	9.51
gap	1.21	8.8%	0.37	1.3%	1.4%	0.02	1.9%	0.13	12.93
gcc	1.96	11.4%	4.06	1.9%	0.5%	1.29	1.1%	1.03	5.99
parser	1.49	10.9%	4.13	2.5%	0.0%	0.00	1.9%	1.36	6.38
perlbmk	2.59	8.8%	0.66	1.7%	1.1%	1.40	1.8%	0.48	2.26
vortex	2.41	10.1%	0.13	2.9%	0.0%	0.02	1.5%	0.06	8.77
Q1	2.55	9.1%	0.87	3.3%	0.6%	0.12	2.8%	1.36	5.41
Q4	2.22	10.3%	0.42	3.3%	0.5%	0.50	3.0%	1.41	12.71
Q6	2.20	10.3%	0.39	3.2%	0.6%	0.98	3.0%	2.15	10.22
Q16	2.76	14.2%	0.66	3.5%	0.2%	0.00	2.4%	0.32	4.03
Q17	2.15	9.8%	0.80	3.5%	0.4%	0.06	3.2%	1.58	12.21
average	2.05	9.7%	1.56	2.5%	0.5%	0.57	2.1%	1.01	7.68

Instruction frequency is expressed as the percentage of all executed instructions. MPKI is mispredicts per kilo instruction for the instruction type. Column returns/BTB shows the number of mispredicts when return targets are predicted by the BTB.

Wrong path misprediction is especially addressed by the self-checkpointing stack presented by Jourdan et al. [1996]. This RAS design is not sensitive to corruption by wrong-path instructions, but requires a larger RAS as popped entries remain in the RAS. We will see in Section 6 that this solution, eliminating wrong-path corruption but increasing overflow corruption, is very effective but requires a large number of entries (e.g., 128).

Skadron et al. [1998] propose to checkpoint some of the RAS contents along with the TOS pointer. Clearly, checkpointing the full RAS contents is unfeasible, but checkpointing only the top RAS entry eliminates most effects of RAS corruption. However, the noncheckpointed RAS entries remain vulnerable to corruption. These entries are still responsible for a large number of mispredictions, even for large RAS sizes [Desmet et al. 2005].

Annavaram, Diep, and Shen [2002] analyze a commercial OLTP workload running on an Oracle database server. This workload has many RAS mispredictions due to frequent context switching and explicit return target manipulation. As a result, the number of calls and returns differs, and the TOS is misaligned. This behavior also results from the setjmp/longjmp mechanism in C programs. These mispredictions cannot be classified as overflow or wrong path corruption. However, it is important to note that when a misaligned TOS is encountered, the subsequent returns on the TOS will also be mispredicted until the next call.

Accurate RAS prediction is important to obtain high performance in pipelined processor, especially when pipelines are deeper [Desmet et al. 2005].

The importance of RAS prediction varies strongly between programs and may depend on programming practice, programming language, and compiler optimizations. Programmers can choose to partition their programs in more or less functions and compilers may be more or less aggressive when inlining functions. Calder et al. [1994] found that object-oriented programs execute more function calls and returns than procedural languages, although our analysis shows a large spectrum of behaviors for C++ programs, ranging from almost

no function calls to the highest fraction of calls we observed in any benchmark. Still, many programs execute a high fraction of function calls and returns. In this study, we restrict our evaluation to those programs.

In practice, processors implement relatively small RASs, ranging from 8 [McNairy and Soltis 2003; Song 1997] to 32 entries [Gwennap 1996].

Several authors have investigated the use of RAS to protect against buffer overflow attacks. Buffer overflow attacks occur when a program overwrites return addresses in the software call stack, after which arbitrary malicious code may be executed. To avoid this, secure RASs have been proposed to maintain a secure copy of the call stack [McGregor et al. 2003; Xu et al. 2002]. These secure RASs are visible by the instruction set, they are not speculative and do not address performance.

Ye and Kaeli [Ye and Kaeli 2005] have proposed a design which aims at addressing both performance and security without modifying the instruction set. Their reliable return address stack (RRAS) is designed to detect returns even when they are hidden through unstructured programmation. The main objective is to detect all cases of return address mispredictions. This goal is priviledged over performance. RAS overflows are avoided by spilling the RAS to memory on overflows and refilling it on underflows. Also, the RAS is saved and restored on context switches to maintain the correct state. Misalignment of the TOS, which may be due to hacks, compiler tricks, or direct assembly programming, is handled by scanning the RAS in reverse order from top to bottom to determine the correct TOS. Finally, the RRAS *prevents* the occurence of wrong-path corruption of a return address by stalling the fetch of a call instruction as long as any speculative return instruction is in-flight. On a deep wide-issue superscalar pipeline, this last feature alone will significantly impair performance. All these measures increase the accuracy of the RAS to nearly 100% and, therefore, address the security, but at the cost of an overall processor performance decrease.

In contrast, the techniques we investigate in this article address return address prediction accuracy to improve the overall processor performance: on the detection of any misprediction on the RAS, fetch is not stopped, but an alternate source of prediction is used.

4. RAS CORRUPTION DETECTION

RAS corruption is detected at the same time as making the prediction. By adding a small amount of additional state (e.g., another TOS, corruption bits), it is possible to quickly determine that the specific RAS entry used in the current prediction has been corrupted. For brevity, we say that return predictions using corrupted state are corrupted return predictions.

We present two simple hardware mechanisms that respectively detect RAS overflows and corruption by wrong-path instructions. These mechanisms detect these respective situations with 100% coverage.

Note that corruption by overflows or by wrong-path instructions does not always result in a return misprediction. False detections are a natural consequence of program behavior. Firstly, it is consistent with the operation of the

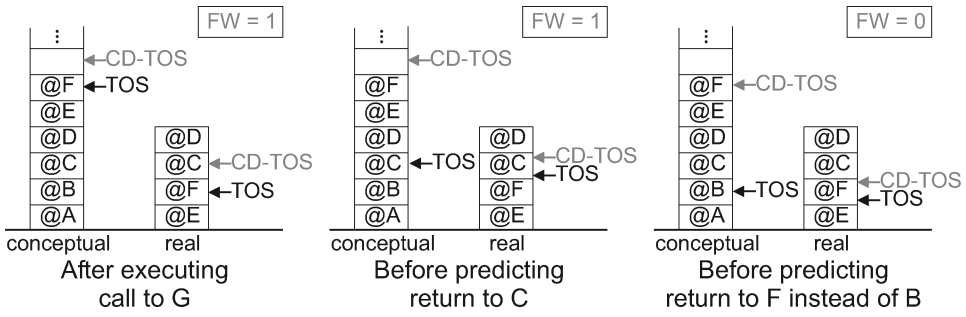

Fig. 1. Detecting overflows on a finite RAS. A chain of procedure calls is executed, with procedure A calling B, B calling C, etc, up to F calling a procedure G. *@A* indicates the address to return to in procedure A, when returning to A. The gray parts are additions to the RAS for the purpose of corruption detection. The FW box shows the first-wrap bit.

RAS that overflows can be correct predictions (e.g., when executing recursive procedures, the same return address is repeatedly pushed on the RAS). Even though overflow occurs, return address prediction remains correct. Secondly, RAS activity on wrong-path instruction sequences can be very complex. RAS contents may be overwritten with wrong-path return addresses that are the same as the correct return addresses. This can happen when a procedure is repeatedly called from a loop and control flow inside the procedure is mistakenly predicted to leave the procedure. The next iteration of the loop executes the same call, thereby overwriting the RAS with the same return address.

4.1 Detecting RAS Overflows on a Conventional RAS

Call instructions push their return address on the RAS. The corresponding return instructions pop the address off the stack. The TOS points to the current top of the stack. Conceptually, the number of return addresses on the stack grows as large as necessary (Figure 1, left).

In practice, the RAS is finite in size. When the TOS exceeds the physical RAS size, it wraps around to the bottom of the RAS. Hereby, call instructions start to overwrite older RAS entries, erasing the return addresses of prior calls. When program control leaves procedure C, procedure C's return instruction is predicted to jump back to procedure F instead of B (Figure 1, left).

We detect RAS overflows by means of a second pointer to the RAS: the CD-TOS (corruption detection TOS). The TOS and CD-TOS together identify the region of the stack that holds the most recently pushed return addresses. Overflow occurs when a call instruction overwrites the entry pointed to by the CD-TOS, then, the CD-TOS is incremented by 1. Underflow occurs when a return address is predicted with the TOS and CD-TOS pointing to the same RAS entry; then, the CD-TOS is decremented by 1.

A return instruction is predicted correctly as long as the TOS differs from the CD-TOS. This is certainly true for the first return after a call instruction. After executing a longer sequence of returns, the TOS wraps around and at some point it meets the CD-TOS (Figure 1, right). From this moment on, predictions are made using addresses that have been overwritten by more recent instructions

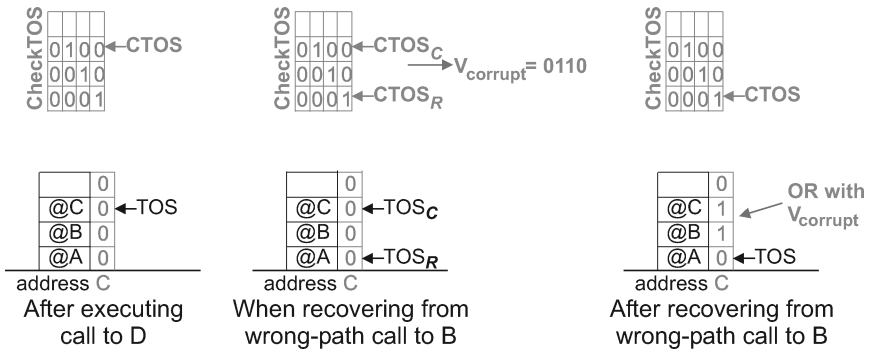

Fig. 2. Detecting wrong-path activity on the RAS. A chain of procedure calls is executed with procedure A calling B, procedure B calling C and procedure C calling D. The call to B is on the correct path, while the others are wrong-path calls. The figure shows what happens when squashing the wrong-path instructions. In this example, the RAS is 4 entries large so the vector length in the CheckTOS table is $N = 4$. The gray parts are additions to the RAS for the purpose of corruption detection.

(@E and @F), erasing the old return addresses (@A and @B). These predictions are made using corrupted state, so they may be incorrect predictions.

Note that the TOS and CD-TOS already meet at the last correctly predicted return address (Figure 1, middle). This condition is detected by checking a first-wrap bit. Call instructions that increase the CD-TOS also set the first-wrap bit. A return address prediction uses corrupted state when the TOS and CD-TOS are equal and the first-wrap bit is not set. When the CD-TOS is decremented, the first-wrap bit is reset.

The CD-TOS and first-wrap bit are computed along the correct execution path. They are checkpointed and recovered together with the TOS to protect them from corruption by speculative execution.

4.2 Detecting Wrong-Path RAS Pushes

During speculative execution, wrong-path call and return instructions may be fetched and act upon the RAS. When a branch misprediction is detected, the RAS state is partially recovered by copying the TOS from checkpointed storage [Jourdan et al. 1996]. Wrong-path call instructions, however, have the side-effect of overwriting RAS entries, which may cause future RAS mispredictions. This occurs, for example, when a wrong-path return and then a wrong-path call instruction are executed. The wrong-path call instruction modifies the RAS entry for the return (Figure 2). After recovery, the TOS is correctly restored, but the RAS contents remain incorrect.

We introduce a simple mechanism to determine which entry has been overwritten by a wrong path return. Every entry in the RAS is augmented with a corruption bit. The underlying idea is that, when the corruption bit is set, it indicates that the RAS entry has been overwritten by a wrong-path call.

When fetching instructions, we implicitly assume that the instruction fetch is performed on the right path. Therefore, on a push by a call, the corruption

bit is set to 0 indicating that the entry is valid. When a misprediction is later detected, one would like to be able to invalidate immediately all the entries of the RAS that have been corrupted on the wrong path.

To track which RAS entries have been modified, we associate an extra local checkpoint mechanism associated with the RAS. For an N-entry RAS, this checkpoint mechanism consists of a table of N-bit vectors. We will refer to this table as the CheckTOS table. On a call, the index of the RAS entry written by the push is checkpointed in decoded format on entry CTOS on the CheckTOS table; that is, if entry X is written on the RAS, a vector of N bits is written on the CheckTOS table with bit X set to 1 and the other bits set to 0. On the call, CTOS is incremented. Thus, CheckTOS holds the RAS entries modified by all inflight call instructions in the region delimited by the CTOS of the recovered instruction, $CTOS_R$ and the current CTOS, $CTOS_C$. Note that CTOS must be checkpointed in the global checkpoint mechanism.

During a misprediction recovery, the checkpoint vectors associated with all the calls on the wrong path (i.e., all entries between $CTOS_R$ and $CTOS_C$) are ORed. The resulting vector $V_{corrupt}$ represents the entries that have been overwritten by the calls on the wrong path (i.e., the 1's in this vector correspond exactly with the entries that have been overwritten by wrong path calls on the RAS).

The vector of corruption bits of the RAS entries is then updated by an OR with the vector $V_{corrupt}$.

Our mechanism detects all RAS corruptions as long as the number of calls encountered on the wrong path is lower than the number of entries in the CheckTOS table.

In our simulations, we encountered no more than 10 calls on the wrong path before detecting the misprediction, except for parser where the number of wrong-path calls peaked at 15. The simulations presented in this article assume 16 entries in the CheckTOS table.

The CheckTOS table is very small: For an 8-entry RAS, the CheckTOS table contains 16 entries of just 8 bits each (i.e., 16 bytes). This is clearly smaller than the size of the RAS, which could be up to 128 bytes for a 64-bit architecture.

4.3 Detecting Corruption on Self-Checkpointing RAS

We refer to the self-checkpointing RAS of Jourdan et al. [1996] as the SC-RAS. The SC-RAS has a particular allocation behavior that calls for specific mechanism to detect corruption.

4.3.1 *Operation of the SC-RAS.* The SC-RAS explicitly links each RAS entry to the logically lower RAS entry using a next-on-stack (NOS) pointer [Jourdan et al. 1996]. The next entry to allocate is indicated by the NEXT pointer. Thus, when executing a call instruction, the NOS pointer and return address fields of the entry pointed to by NEXT are initialized and the TOS is updated to point to this entry. Finally, the NEXT pointer is incremented by one.

When executing a return instruction, the predicted return address is read from the entry pointed to by TOS. The NOS field of this entry is copied to

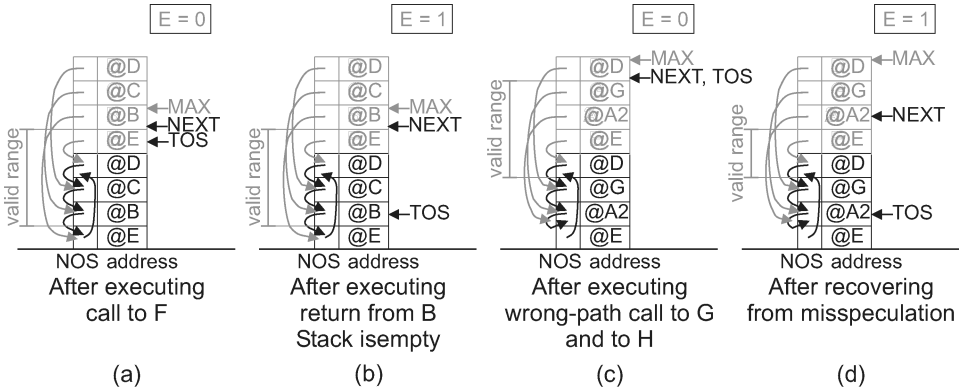

Fig. 3. Detecting SC-RAS corruption. A chain of procedure calls is executed, with procedure A calling B, B calling C, etc, up to E calling a procedure F (a). Correct path returns follow immediately, taking control back to procedure A (b). Then, two wrong-path calls are executed to G and H (c) and recovery is performed (d). Extending the RAS pointers gives the impression of a larger RAS (virtual copy shown in gray). The MAX pointer and empty bit (E box) are additions to the RAS for the purpose of detecting corruption.

the TOS, thereby popping this entry from the stack. The NEXT pointer is *not* modified by return instructions. This allocation behavior implies that all RAS entries are cyclically reallocated by the progressing NEXT pointer.

We distinguish two types of SC-RAS corruption: (1) mispredictions due to reallocation of SC-RAS entries and (2) mispredictions due to wrong-path SC-RAS pushes. Note that, contrary to the simple RAS, wrong-path SC-RAS pushes cause mispredictions mostly at the bottom of the stack, which is much less harmful.

4.3.2 *Detecting Reallocation of SC-RAS Entries.* The region of SC-RAS entries that may hold correct state is delimited by two pointers. On the one hand, the TOS pointer delimits the top of the stack: Entries that are above the TOS pointer have not yet been allocated and they are not part of the correct stack state. On the other hand, the NEXT pointer serves as a re-allocation pointer that invalidates entries at the bottom of the stack. Thus, the region of the SC-RAS that may hold correct state can be deduced by comparing the TOS and NEXT pointer.

Comparing pointers that may wrap around requires us to know which pointer is above the other one. E.g. when NEXT=TOS+1 then an N-entry SC-RAS may either contain $N-1$ addresses or none, depending on how this situation was reached. To track the relation between the TOS and NEXT pointers, we simply make the TOS, NOS and NEXT pointers wider to extend their range (e.g., by 1 bit. This extra bit is dropped when accessing the SC-RAS. Using this modification, the valid RAS entries are found in the range [NEXT-N,NEXT] where N is the SC-RAS size. This is depicted in (Figure 3(a)), where the extended ranges of the RAS pointers are depicted by duplicating the RAS (only the black part actually exists).

The SC-RAS prediction is corrupted when the prediction is based on an SC-RAS entry outside the valid range. To support the corruption detector, we add an empty bit to the SC-RAS and operate it as follows. The empty bit is initially 1. The empty bit is set when executing a return with a NOS field outside the valid range. The empty bit is cleared when executing a call instruction. Return predictions use corrupted state when the empty bit was set to 1 before making the prediction.

Furthermore, we make the following modification to the SC-RAS to improve its efficiency. If the stack was not empty before the return and does not become empty after the return, then NOS is copied to TOS (i.e., the entry is popped). If the stack was empty before the return or becomes empty on the return, then the empty bit is set, the RAS is not popped since there are no more valid entries in the RAS (Figure 3(b)).

Note that when executing a call on an empty stack, then the caller to the current function is unknown on the RAS. The corresponding pushed entry must reflect this situation: its NOS field is written with an out-of-range number (e.g., MAX-N-1) (Figure 3(c)).

4.3.3 *Detecting Wrong-Path SC-RAS Pushes.* Wrong-path call instructions allocate RAS entries that follow sequentially on the entry pointed to by NEXT. The NEXT pointer is recovered from checkpoint storage when detecting a misprediction, so the corruption of these entries can not be deduced from the stack pointers. Thus, we add a MAX pointer that tracks the maximum value of the NEXT pointer. The valid range becomes [MAX-N,NEXT].

When executing wrong-path calls, NEXT is incremented and MAX tracks the maximum value of NEXT (Figure 3(c)). NEXT and MAX differ after recovering from the misspeculation. The entries corrupted by wrong-path instructions have been subtracted from the valid range (Figure 3(d)). Subsequent call instructions will enlarge the valid range.

4.4 Dealing with Misaligned TOS and Context Switches

As pointed out by Annavaram et al. [2002], in some applications, explicit management of return targets or frequent context switching may lead to TOS misalignment. These mispredictions cannot be classified as overflow or wrong path corruption. However, once a TOS misalignment has been encountered, the subsequent returns on the TOS will be also misaligned until a call is encountered.

Although detecting the first TOS misalignment appears to be hard, we can use the respective overflow detectors that have been presented above to classify the subsequent returns as corrupted predictions. When a TOS misalignment is encountered, we force our mechanisms to consider that the overall RAS is empty (i.e., we force CD-TOS=TOS and FW=0 for the overflow detector for conventional RAS and we set the empty bit for the overflow detector for SC-RAS).

To avoid even the first return misprediction due to a TOS misalignment associated with a context switch, the confidence estimators may be reset similarly on a context switch.

This feature is not validated in our experiments since we had no access to applications featuring this type of TOS misalignments and our simulation framework did not accurately model operating system effects.

5. USING A FREE BACKUP PREDICTOR FOR CORRUPTED RETURNS

The mechanisms presented in the previous section detect whether or not the top entry of the RAS has been corrupted since its write by the associated call. Although the content of this entry may still deliver a correct prediction, the accuracy of the prediction delivered by this entry is likely to be quite poor on average. Therefore, using an alternate source for predicting the returns when the top entry is corrupted will deliver better performance even if the accuracy of this alternate source is relatively low.

We chose here to use an already present component, the BTB, as this alternate source. The BTB holds information on all branch instructions, including the branch type (conditional, indirect, return, etc.) and a branch target address. The BTB is known to be a relatively poor return address predictor [Skadron et al. 1998]. Table II shows that the accuracy of the BTB on returns is only around 63% of good predictions on our benchmark selection. However, in this article, we use the BTB prediction only when the RAS prediction is known to come from a corrupted RAS entry. Therefore, the BTB prediction accuracy does not need to be very high to improve overall accuracy.

Some processor designs do not implement BTBs (e.g., Alpha EV6 [Kessler 1999] or the canceled Alpha EV8 [Seznec et al. 2002]). On such processors, the indirect jump predictor could be used as the alternate source for predicting returns when the RAS top entry is corrupted. Experiments presented in Section 6.5 illustrate that the indirect jump predictor is a slightly more accurate alternate source for predicting the low confidence returns than the BTB and leads to performance slightly higher than the BTB.

Note that, in both cases, the alternate source for return predictions is already present in the instruction fetch front end and, therefore, their new functionality as backup return predictor comes for free.

6. EVALUATION

We evaluate our BTB backup predictor on three RAS designs: the simple RAS (i.e., a RAS without any speculative management enhancement); the checkpointed top RAS from Skadron et al. [1998], which we will refer to as the CT-RAS; and the self-checkpointing RAS from Jourdan et al. [1996]. As a first step, we evaluate 32-entry RAS for the three designs, since this medium size is often considered as a good tradeoff. Then, we vary the RAS sizes.

6.1 Evaluation of 32-Entry RAS Designs

6.1.1 *Simple RAS.* Table III illustrates the performance of a 32-entry simple RAS alone and with a BTB backup. On average, the baseline design achieves 2.05 IPC and 1.01 mispredicted return instructions per 1,000 instructions.

Table III. Performance and Analysis of the Backup Predictor on a 32-entry Simple RAS

	Baseline			Backup	
Bench-mark	IPC	Return MPKI	PVN	IPC	Return MPKI
crafty	2.25	1.14	85.1%	2.30	0.52
eon	2.20	1.46	69.2%	2.30	0.33
gap	1.21	0.13	92.8%	1.21	0.00
gcc	1.96	1.03	70.5%	1.97	0.72
parser	1.49	1.36	59.2%	1.52	0.56
perlbmk	2.59	0.48	54.4%	2.62	0.14
vortex	2.41	0.06	95.8%	2.41	0.01
Q1	2.55	1.36	91.7%	2.67	0.10
Q4	2.22	1.41	71.2%	2.32	0.32
Q6	2.20	2.15	76.7%	2.37	0.37
Q16	2.76	0.32	46.8%	2.80	0.05
Q17	2.15	1.58	92.1%	2.26	0.21
average	2.05	1.01	72.3%	2.10	0.28

The overflow and wrong-path corruption detectors classify all the return mispredictions as corrupted on our benchmark suite. We have already pointed out in Section 4 that some return predictions may use corrupted state but are nonetheless correctly predicted. We characterized this phenomenon (column PVN in Table III): 47% to 96% are effectively mispredicted by the RAS.

When using the BTB as a backup predictor for corrupted predictions on the RAS, performance (measured in IPC) improves by 2.5%, on average, on our benchmark set and can improve up to 7.6% on Q6. The average return mispredictions drop from 1.01 MPKI to 0.28 MPKI, a factor of 3.6 difference. Note that all mispredictions are now due to the backup predictor. RAS predictions are used only when they are correct.

6.1.2 *CT-RAS.* On the CT-RAS, the top entry must be checkpointed on every branch [Skadron et al. 1998]. In addition to the top entry of the RAS, our scheme requires to checkpoint and restore the corruption bit of the top RAS entry to make the wrong-path instructions detector work correctly.

For 32 entries, the average CT-RAS accuracy is 0.39 MPKI and the baseline IPC is 2.09 (Table IV). This confirms that, as stated in [Skadron et al. 1998], for a 32-entry RAS, most of the return mispredictions are due to overwrites of the top entry of the RAS by a wrong path instruction.

Enabling the BTB backup predictor allows to further reduce the misprediction rate on returns to a mere 0.09 MPKI. This further translates into a small average increase of performance (2.09 to 2.12 IPC).

6.1.3 *The SC-RAS.* The baseline 32-entry SC-RAS design leads to a disappointing performance with an IPC only slightly higher than the Simple RAS (2.07 against 2.05) and a return misprediction rate only slightly lower than the Simple RAS. We will see in Section 6.2 that SC-RAS needs more entries to be efficient.

Table IV. Performance and Analysis of the Backup Predictor on a 32-entry CT-RAS

	Baseline			Backup	
Bench-mark	IPC	Return MPKI	PVN	IPC	Return MPKI
crafty	2.33	0.10	74.0%	2.34	0.04
eon	2.30	0.35	62.6%	2.33	0.07
gap	1.21	0.07	97.1%	1.21	0.00
gcc	2.00	0.32	75.4%	2.01	0.19
parser	1.51	0.67	68.9%	1.53	0.34
perlbmk	2.60	0.36	52.1%	2.63	0.11
vortex	2.41	0.02	99.5%	2.41	0.00
Q1	2.57	1.13	90.5%	2.67	0.10
Q4	2.29	0.46	61.4%	2.32	0.10
Q6	2.33	0.52	61.8%	2.37	0.00
Q16	2.79	0.05	74.9%	2.80	0.00
Q17	2.20	0.86	94.3%	2.27	0.08
average	2.09	0.39	71.4%	2.12	0.09

Enabling the backup predictor on corrupted return predictions yields an increase of the average IPC from 2.07 to 2.12 and a decrease of the average return rate from 0.74 misp/KI to 0.09 misp/KI.

6.1.4 *Comparing the 32-Entry RAS Designs.* The three 32-entry RAS designs we have analyzed have quite distinct behaviors in their baseline form. For instance, the Simple RAS performs very poorly on the SPEC benchmark in general. On the other hand, the SC-RAS design performs very poorly on TPC-D benchmarks, Q16, and Q17, but shines on SPEC benchmarks.

These differences are due to the types of mispredictions that these designs encounter. The self-checkpointing RAS behaves differently from the simple RAS and CT-RAS with respect to overflows and wrong-path activity. The SC-RAS only suffers from overflow mispredictions. This is reflected in the PVN of the corruption detector which is very high, 81% to 100% with an average of 99.3% (Table V). However, the number of RAS overflows is also very high compared with the other designs—a valid RAS entry associated with a call is overwritten by overflow whenever 32 other calls are fetched before the corresponding return.

On the other hand, there are very small differences in terms of performance and misprediction return rates when the RAS is backed up with the BTB and corruption detectors.

6.2 Varying RAS Size

Table VI illustrates the average performance when varying RAS sizes for the three previously considered RAS designs and for the different combinations of the corruption detectors, the BTB being used as backup predictor.

First, it can be noticed that the SC-RAS design is the most effective design when no backup predictor is used. However a large number of entries is required on this design (around 128).

Table V. Performance and Analysis of the Backup Predictor on a 32-Entry SC-RAS

	Baseline			Backup	
Bench-mark	IPC	Return MPKI	PVN	IPC	Return MPKI
crafty	2.33	0.17	98.8%	2.34	0.06
eon	2.28	0.65	99.9%	2.33	0.05
gap	1.21	0.08	99.9%	1.21	0.01
gcc	2.01	0.16	99.8%	2.02	0.04
parser	1.53	0.22	80.8%	1.53	0.14
perlbmk	2.61	0.26	100.0%	2.63	0.03
vortex	2.37	0.46	100.0%	2.40	0.09
Q1	2.60	0.87	100.0%	2.68	0.12
Q4	2.17	1.72	100.0%	2.30	0.29
Q6	2.26	1.38	100.0%	2.37	0.01
Q16	2.56	1.90	100.0%	2.79	0.13
Q17	2.14	1.51	100.0%	2.26	0.13
average	2.07	0.74	99.3%	2.12	0.09

Table VI. Average IPC for Varying RAS Size, RAS Configuration and Corruption Detector

Simple RAS						
RAS Size	4	8	16	32	64	128
Baseline	1.90	2.02	2.05	2.05	2.05	2.05
Overflow	2.02	2.04	2.05	2.05	2.05	2.05
Wrong-path	1.94	2.07	2.10	2.10	2.10	2.10
Both	2.06	2.10	2.10	2.10	2.10	2.10
CT-RAS						
RAS Size	4	8	16	32	64	128
Baseline	1.93	2.06	2.09	2.09	2.09	2.09
Overflow	2.06	2.08	2.09	2.09	2.09	2.09
Wrong-path	1.95	2.08	2.11	2.12	2.12	2.12
Both	2.07	2.11	2.11	2.12	2.12	2.12
SC-RAS						
RAS Size	4	8	16	32	64	128
Baseline	1.83	1.94	2.03	2.07	2.10	2.11
Overflow	2.04	2.08	2.11	2.12	2.12	2.12

The overflow and wrong-path corruption detectors each target a particular phenomenon that causes RAS mispredictions. However, the importance of these phenomena depends on the RAS design and RAS size.

On simple RAS and CT-RAS predictors, the wrong-path detector improves performance for all sizes. For these two designs, the overflow detector is also useful for small RAS (8 entries or less), but has very limited impact for larger RAS.

The SC-RAS shows a very different behavior. The SC-RAS suffers only very marginally from wrong-path activity—extra misses are encountered only when an overflow occurs on the wrong path and would not have occurred on the right path. But the allocation policy of the SC-RAS creates overflow corruption by construction, therefore the overflow detector improves the SC-RAS performance across all RAS sizes.

6.3 Design Complexity Tradeoffs

6.3.1 *RAS Design Complexity Tradeoff.* Our proposal improves the performance of the three considered designs for the return predictor. The extra performance gain on a quite complex 128-entry SC-RAS can be considered as marginal. However the performance gain on an 8-entry Simple RAS is much more significant—an 8-entry Simple RAS backed up with the BTB reaches the same performance level as a 64-entry SC-RAS.

The SC-RAS has a clearly more complex design (multiple stack pointers, explicit linking of RAS entries) than the simple RAS. Furthermore, it should be significantly larger than the simple RAS with corruption detectors before reaching the same performance.

Therefore, our BTB-backedup RAS design can be used as a way to reduce RAS design complexity.

6.3.2 *Overall Branch Predictors Complexity Tradeoff.* To improve the overall branch misprediction rate, one can use our BTB backup RAS or try to improve the accuracy of the conditional branch predictor through enlargening it for instance.

We run experiments comparing the respective benefits of quadrupling the conditional predictor size from 64Kbits to 256Kbits and the use of a BTB-backup RAS.

Our experiments showed that, quadrupling the OGEHL storage budget only reduces the overall average misprediction rate by 0.20 misp/KI, 0.27 misp/KI and 0.23 misp/KI (with very different behaviors depending on benchmarks) for respectively 32-entry simple RAS, 32-entry CT-RAS and 32-entry SC-RAS. At the same time, using the BTB-backup RAS results in respective reductions of the overall average misprediction rates by 0.73 misp/KI, 0.30 misp/KI, 0.65 misp/KI for respectively 32-entry simple RAS, 32-entry CT-RAS and 32-entry SC-RAS. This better overall misprediction rate also translates in a slightly higher overall performance increase (0.03 IPC in average for simple RAS and SC-RAS).

That is, enhancing any 32-entry RAS design with corruption detectors and BTB-backup is more effective than quadrupling the OGEHL predictor size from 64Kbits to 256Kbits.

6.4 Pipeline Depth Impact

In Section 6.2, the 8-entry simple RAS backed up with the BTB and the 64-entry SC-RAS were found to achieve very similar performance level for our baseline 16-cycle pipeline design. Table VII compares the performance achieved with different RAS designs and assuming different pipeline depths. Note that varying the pipeline depth impacts many aspects of processor performance. To factor out these effects, performance is presented as a fraction of the performance that would be achieved with perfect return prediction. The table also illustrates the return misprediction rates. Two designs are considered, 8-entry simple RAS and 64-entry SC-RAS.

As expected, the performance loss and the return misprediction rate increase with the pipeline depth. The performance loss for the 8-entry simple RAS

Table VII. Varying Pipeline Depth; Fraction of the Perfect-RAS IPC and Return Mispredictions per 1,000 Instructions. 8-Entry Simple RAS and 64-Entry SC-RAS

Simple RAS (8 entries)						
	Percentage Perfect-RAS IPC					
Depth (cycles)	5	11	16	21	26	31
Baseline	98.5%	96.7%	94.9%	93.2%	91.7%	89.8%
BTB backup	99.6%	99.1%	98.8%	98.3%	97.7%	97.1%
JUMP backup	99.8%	99.5%	99.2%	98.9%	98.6%	98.3%
	Return MPKI					
Depth (cycles)	5	11	16	21	26	31
Baseline	0.99	1.31	1.51	1.63	1.72	1.84
BTB backup	0.25	0.36	0.39	0.41	0.46	0.50
JUMP backup	0.11	0.16	0.20	0.22	0.23	0.25
SC-RAS (64 entries)						
	Percentage Perfect-RAS IPC					
Depth (cycles)	5	11	16	21	26	31
Baseline	99.5%	99.2%	98.8%	98.6%	98.3%	98.2%
BTB backup	99.9%	99.9%	99.8%	99.8%	99.7%	99.7%
JUMP backup	99.9%	99.9%	99.9%	99.9%	99.8%	99.8%
	Return MPKI					
Depth (cycles)	5	11	16	21	26	31
Baseline	0.30	0.30	0.30	0.30	0.30	0.30
BTB backup	0.05	0.05	0.05	0.05	0.05	0.05
JUMP backup	0.02	0.02	0.02	0.02	0.02	0.02

without backup becomes even very high; for a pipeline depth of 31 cycles, more than 10% of the performance is wasted by the imperfect return prediction. A deeper pipeline results in more calls and returns on the wrong path. This results in more wrong path corruptions.

In contrast, the 8-entry simple RAS with BTB backup predictor remains quite efficient with deep pipelines. However, the BTB predictor is used more often for return prediction (from 1.09 used per 1,000 instructions for a 5-cycle pipeline to 2.07 for a 31-cycle pipeline); therefore, the overall return misprediction rate is also increased.

When the pipeline is relatively short (less than 20 cycles), the 8-entry simple RAS predictor with BTB-backup achieves better or same performance than the 64-entry SC-RAS.

The simple RAS is affected both by RAS overflows and wrong path corruptions. The more complex SC-RAS is only affected by RAS overflows. Using a deeper pipeline only means a larger number of calls on the wrong path. But this only translates to a very marginal increase of the number of RAS overflows on the large 64-entry SC-RAS—less than 0.01 per 1,000 instructions extra uses of the backup predictor for a 31-cycle pipeline.

6.5 BTB vs. Indirect Jump Predictor

Until now, we have considered using the BTB as the backup predictor for returns. On processors featuring an indirect jump predictor, this jump predictor can be used as the backup predictor for returns. Table VII also illustrates the use of the indirect jump predictor as backup predictor.

It can be noted that when backed up with a jump predictor, the 64-entry SC-RAS achieves nearly perfect return mispredictions; the backup predictor is used for 0.30 returns per 1,000 instructions and only 0.02 misp/KI on returns is encountered, but for a BTB backup, the misprediction rate was already very low (0.05 misp/KI).

The advantage of using the jump predictor as the backup predictor is more pronounced when using a 8-entry simple RAS. When the pipeline depth is increased, the accuracy advantage of the indirect jump predictor is translated in a more pronounced accuracy benefit on return predictions and finally translates in a more pronounced performance advantage.

For a very deep 31-cycle pipeline, a simple RAS backed with the indirect jump predictor is still more efficient than a 64-entry SC-RAS. Therefore, we recommend to use the indirect jump predictor as the back-up predictor, particularly when the pipeline is very deep.

7. CONCLUSION

Highly accurate conditional and branch target predictors exacerbate the importance of correct return prediction. Ideally, a RAS should predict return targets with a 100% accuracy. In practice, a simple medium size RAS has relatively poor behavior. To improve the behavior of the RAS predictor, complex RAS designs were proposed (e.g., self-checkpointing RAS) [Jourdan et al. 1996]. A large (e.g., 128-entry) self-checkpointing RAS achieves nearly perfect return prediction.

In this article, we have shown that the two phenomena (RAS entry corruption through RAS overflow and wrong-path overwrite) that are responsible for RAS mispredictions can be detected through very simple hardware detectors. These detectors always detect the RAS corruption. Therefore, when the RAS top is known to be corrupted, one can rely on an alternate source of prediction. The BTB or the indirect jump predictor can be used as such a free alternate predictor for returns.

Our experiments show that our proposal can be used to improve the behavior of the previously proposed speculative RASs; for example, Jourdan et al. [1996] and Skadron et al. [1998].

In particular, it allows to achieve very high accuracy using a naive simple RAS design as the base component; an 8-entry BTB-backed up simple RAS achieves the same performance level as a state-of-the-art, but complex, 64-entry self-checkpointing RAS [Jourdan et al. 1996]. Therefore, our (BTB or jump predictor)-backed up RAS might be seen as a way to improve the performance or as a way to reduce RAS design complexity. This complexity reduction could be leveraged in SMT processors where a RAS predictor is needed for each thread [Hily and Seznec 1996], and of course in multicores. For instance, an 8-core 4-way multithreaded processor (e.g., the Sun Niagara) should feature 32 RAS.

The RAS corruption detectors can also be used as branch confidence estimators (e.g., to control fetch gating or in SMT fetch policies). Until now, in most studies considering such a usage, the prediction of a return through the return stack was uniformly considered as high confidence.

When the top of the stack is uncorrupted, the accuracy of the prediction is typically very high, nearly 100%, while the accuracy of the prediction provided by the alternate predictor is lower. Therefore, our corruption detectors can be used as a simple means to discriminate between low confidence and high confidence return predictions. This can naturally be used to improve all the hardware mechanisms that use branch confidence estimations (e.g., fetch gating) [Manne et al. 1998; Buyuktosunoglu et al. 2003] or SMT fetch policies [Tullsen et al. 1996; Luo et al. 2001]. An overview of applications of confidence estimation is provided in Jacobsen et al. [1996] and Grunwald et al. [1998].

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers for their constructive comments. Hans Vandierendonck is a postdoctoral research fellow with the Fund for Scientific Research–Flanders. His research is supported in part by the Flemish Institute for the Promotion of Scientific-Technological Research in the Industry (IWT) and Ghent University. André Seznec was partially supported by an Intel research grant. The collaboration was supported by the HiPEAC Network of Excellence.

REFERENCES

Annavaram, M., Diep, T., and Shen, J. 2002. Branch behavior of a commercial OLTP workload on intel IA32 processors. In *Proceedings of the 2002 IEEE International Conference on Computer Design*. 242–249.

Buyuktosunoglu, A., Karkhanis, T., Albonesi, D. H., and Bose, P. 2003. Energy efficient co-adaptive instruction fetch and issue. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*. 147–156.

Calder, B., Grunwald, D., and Zorn, B. 1994. Quantifying behavioral differences between C and C++ programs. *Journal of Programming Languages 2*, 4, 313–351.

Desmet, V., Sazeides, Y., Kourouyiannis, C., and De Bosschere, K. 2005. Correct alignment of a return-address-stack after call and return mispredictions. In *Workshop on Duplicating, Deconstructing and Debunking*. 25–33.

Driesen, K. and Holzle, U. 1998. The cascaded predictor: Economical and adaptive branch target prediction. In *Proceedings of the 30th Symposium on Microarchitecture*.

Grunwald, D., Klauser, A., Manne, S., and Pleszkun, A. 1998. Confidence estimation for speculation control. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*. 122–131.

Gwennap, L. 1996. Digital 21264 sets new standard. *Microprocessor Report 10*, 14 (Oct.), 1–6.

Hily, S. and Seznec, A. 1996. Branch prediction and simultaneous multithreading. In *Proceedings of the 5th International Conference on Parallel Architectures and Compilation Techniques*. 169–173.

Jacobsen, E., Rotenberg, E., and Smith, J. 1996. Assigning confidence to conditional branch predictions. In *Proceedings of the 29th Annual ACM/IEEE International Conference on Microarchitecture*. 142–152.

Jiménez, D. 2005. Piecewise linear branch prediction. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*. 382–393.

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

KAELI, D. R. AND EMMA, P. G. 1991. Branch history table prediction of moving target branches due to subroutine returns. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*. 34–42.

KESSLER, R. 1999. The Alpha 21264 microprocessor. *IEEE Micro 19*, 2, 24–36.

LUO, K., FRANKLIN, M., MUKHERJEE, S. S., AND SEZNEC, A. 2001. Boosting SMT performance by speculation control. In *Proceedings of the 15th International Parallel & Distributed Processing Symposium (IPDPS-01)*.

MANNE, S., KLAUSER, A., AND GRUNWALD, D. 1998. Pipeline gating: speculation control for energy reduction. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*. 132–141.

MCGREGOR, J., KARIG, D., SHI, Z., AND LEE, R. 2003. A processor architecture defense against buffer overflow attacks. In *Proceedings of the International Conference on Information Technology: Research and Education*. 243–250.

MCNAIRY, C. AND SOLTIS, D. 2003. Itanium 2 processor microarchitecture. *IEEE Micro 23*, 2 (Apr.), 44–55.

SEZNEC, A. 2005. Analysis of the O-GEometric History Length branch predictor. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*. 394–405.

SEZNEC, A., FELIX, S., KRISHNAN, V., AND SAZEIDES, Y. 2002. Design trade-offs for the Alpha EV8 conditional branch predictor. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*. 295–306.

SEZNEC, A. AND MICHAUD, P. 2006. A case for (partially) TAgged GEometric history length branch prediction. *Journal of Instruction-Level Parallelism 8*, 1 (Feb.), 1–23.

SHERWOOD, T., PERELMAN, E., HAMERLY, G., AND CALDER, B. 2002. Automatically characterizing large scale program behavior. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*.

SKADRON, K., AHUJA, P. S., MARTONOSI, M., AND CLARK, D. W. 1998. Improving prediction for procedure returns with return-address-stack repair mechanisms. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*. 259–271.

SONG, P. 1997. UltraSPARC-3 aims at MP servers. *Microprocessor Report*.

TULLSEN, D. M., EGGERS, S. J., EMER, J. S., LEVY, H. M., LO, J. L., AND STAMM, R. L. 1996. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*. 191–202.

WEBB, C. F. 1988. Subroutine call/return stack. *IBM Tech. Disc. Bulletin 30*, 11 (Apr.), 221–225.

XU, J., KALBARCZYK, Z., PATEL, S., AND IYER, R. 2002. Architecture support for defending against buffer overflow attacks. In *Proceedings of the 5th Workshop on Evaluating and Architecting System Dependability*.

YE, D. AND KAELI, D. 2005. A reliable return address stack: Microarchitectural features to defeat stack smashing. *Computer Architecture News 33*, 1 (Mar.), 73–80.

Received October 2007; revised March 2008; accepted July 2008