Checkpoint Allocation and Release

AMIT GOLANDER and SHLOMO WEISS
Tel-Aviv University

10

Out-of-order speculative processors need a bookkeeping method to recover from incorrect speculation. In recent years, several microarchitectures that employ checkpoints have been proposed, either extending the reorder buffer or entirely replacing it. This work presents an in-depth study of checkpointing in checkpoint-based microarchitectures, from the desired content of a checkpoint, via implementation trade-offs, and to checkpoint allocation and release policies. A major contribution of the article is a novel adaptive checkpoint allocation policy that outperforms known policies. The adaptive policy controls checkpoint allocation according to dynamic events, such as second-level cache misses and rollback history. It achieves 6.8% and 2.2% speedup for the integer and floating point benchmarks, respectively, and does not require a branch confidence estimator. The results show that the proposed adaptive policy achieves most of the potential of an oracle policy whose performance improvement is 9.8% and 3.9% for the integer and floating point benchmarks, respectively. We exploit known techniques for saving leakage power by adapting and applying them to checkpoint-based microarchitectures. The proposed applications combine to reduce the leakage power of the register file to about one half of its original value.

Categories and Subject Descriptors: C.1 [**Processor Architectures**]: Processors

General Terms: Design, Performance

Additional Key Words and Phrases: Checkpoint, misprediction, out-of-order execution, rollback, early register release, leakage

ACM Reference Format:

Golander, A. and Weiss, S. 2009. Checkpoint allocation and release. *ACM Trans. Architect. Code Optim.* 6, 3, Article 10 (September 2009), 27 pages.
DOI = 10.1145/1582710.1582712 <http://doi.acm.org/10.1145/1582710.1582712>

1. INTRODUCTION

Modern out-of-order (OOO) processor cores concurrently process many instructions in different stages of their pipelines. Such processors require a mechanism to recover the processor state after the occurrence of an incorrect speculation. There are two well-known approaches for recovering the processor state: the reorder buffer (ROB) [Smith and Pleszkun 1988] and checkpointing [Hwu and

Author's address: S. Weiss, Department of Electrical Engineering – Systems, Tel-Aviv University, Tel-Aviv, 69978, Israel; email: weiss@eng.tau.ac.il.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2009 ACM 1544-3566/2009/09-ART10 \$10.00
DOI 10.1145/1582710.1582712 <http://doi.acm.org/10.1145/1582710.1582712>

ACM Transactions on Architecture and Code Optimization, Vol. 6, No. 3, Article 10, Publ. date: September 2009.

Fig. 1. A taxonomy of microarchitectures using checkpoints. The division is based on the reason for using checkpoints.

Patt 1987]. In recent years, several microarchitectures that employ checkpoints have been proposed, either extending the reorder buffer (“hybrid” microarchitectures that use both recovery approaches, ROB and checkpointing) or entirely replacing the ROB with checkpointing (ROB-free).

Figure 1 presents several key microarchitectures that use checkpoints and maps four sets of goals that checkpoints help accomplish. Reliability hybrids use checkpoints to recover from errors. Resource reclamation hybrids speculatively reuse resources, such as registers and load/store queue entries, and use checkpoints to recover from misspeculations. Memory wall hybrids employ checkpoints to handle long latency events, such as L2 cache misses, by either skipping over the load that caused the miss or predicting the load value. Faster recovery hybrids use checkpoints to recover faster from branch mispredictions without waiting until the mispredicted branch reaches the head of the ROB. These hybrid microarchitectures are further described in the related work section (Section 7). This article focuses on ROB-free microarchitectures in which checkpoints are used extensively for recovering from mispredictions.

A recovery is comprised of two phases: rollback, in which the latest safe checkpoint preceding the point of misprediction is recovered, and re-execution in which the code segment between the recovered checkpoint and the mispredicting instruction is executed again. Checkpoint-based microarchitectures apply a checkpoint allocation policy—a heuristic algorithm, indicating when the processor state should be saved. Taking a checkpoint consumes two resources, a checkpoint buffer entry and registers that are part of the checkpoint’s mapping table. A checkpoint allocation policy should be balanced, taking too few checkpoints will require larger code segments to re-execute on recovery and

taking too many checkpoints may consume all available resources bringing the processor to a temporary halt. Easing the pressure on such resources can also be achieved by releasing unneeded checkpoints as soon as they are no longer required (OOO checkpoint release management).

The primary objective of this article is to present a detailed study of checkpoint allocation and release in a checkpoint-based microarchitecture. Specifically, the article makes the following contributions.

- (1) It presents a novel adaptive checkpoint allocation policy in which the policy changes according to dynamic events, such as second-level cache misses, rollback history, and the utilization of the register file and the checkpoint buffer. It also compares the performance of the new policy with two previously proposed checkpoint allocation policies.
- (2) It evaluates the effectiveness of OOO checkpoint release and presents a streamlined scheme that releases OOO only checkpointed registers but does not require the complex OOO checkpoint release logic.
- (3) It evaluates whether the state of predictors, following wrong-path execution, should be restored, reset, or left as is after rollback.
- (4) It introduces two new applications of known techniques for saving leakage power. First, it applies a combination of two techniques—early register release and turning free registers off—to the register management mechanism in checkpoint-based microarchitectures. Second, it applies the technique of drowsy caches to checkpointed registers.

The remainder of the article is organized as follows. Section 2 describes the simulation methodology. Section 3 examines the checkpoint buffer organization and studies the benefits of including the state of different predictors in the checkpointed state. Section 4 shows that previously proposed checkpoint allocation policies are suboptimal and introduces an adaptive policy that achieves better results. Section 5 investigates trade-offs involving in-order and OOO checkpoint release. Section 6 presents a technique that reduces the leakage power of checkpointed and released registers. Finally, related work is surveyed in Section 7, and conclusions are drawn in Section 8.

2. EXPERIMENTAL METHODOLOGY

We used SimpleScalar [Burger and Austin 1997] for performance simulation. We modified its sim-outorder model to implement checkpoints; variable length pipelines; variable size register files; instruction queues (IQ); a state-of-the-art branch predictor (TAGE [Seznec and Michaud 2006]); a mechanism that completely hides the rollback penalty, regardless of the state of the pipeline (CRB [Golander and Weiss 2008]); and a method to reuse the outcome of branches, achieving near-perfect prediction during re-execution (RbckBr [Golander and Weiss 2007]). The simulation parameters of the baseline microarchitecture (refer to Table I) mostly follow the parameters of Power5 [Kalla et al. 2004]. A relatively large load-store queue (LSQ) is needed for checkpoint processors, which commit stores only when the relevant checkpoint is committed [Cristal et al. 2004].

Table I. Baseline Processor Parameters

Branch predictor	Five component, 70Kbit TAGE [Seznec and Michaud 2006] predictor. BTB: 128 sets, 4-way. RAS: 8-entry.
Confidence estimator	16Kbit JRS [Jacobsen et al. 1996], used by LowConfBr alone: 4K-entry table, 4-bit saturating counters, confidence threshold = 15.
Front end	Decode: 2 cycles. Rename and dispatch: 3 cycles.
Resources	INT/FP: 128/128 register file (a single cycle structures) IQ: 96. LSQ: 96. Processor width: 4.
Checkpoints	8-deep in-order checkpoint buffer. $Min_{threshold} = 64$; $Max_{threshold} = 256$; $St_{threshold} = 32$.
Caches and memory	Inst.-L1: 64KB, 256 sets, 2-way, 2 cycles. Data-L1: 32KB, 64 sets, 4-way, 2 cycles. Level one caches use 128B blocks, broken into four 32B sectors. Unified-L2: 1MB, 8-way, 8 cycles, 128B blocks. Memory access time is 200 cycles.

Fig. 2. Organization of an 8-deep in-order checkpoint buffer.

All 26 SPEC CPU2000 benchmarks were used. Results were measured on a 300 million instruction interval, starting after half a billion instructions. The main results were also measured on longer ($2\times$ and $4\times$) intervals, in order to verify that the performance of structures that require a training period, such as the confidence estimator, is stable. Leakage power consumption was estimated using version 4.2 of Cacti [Tarjan et al. 2006], and a 70nm technology, the most advanced technology the tool was verified on.

3. CHECKPOINT BUFFER ORGANIZATION

The baseline microarchitecture does not use a ROB, and speculative execution is supported by the use of checkpoints. A checkpoint saves the state of a program at a certain point during the execution of the program. Checkpoints are taken periodically according to a checkpoint allocation policy and are stored in a checkpoint buffer.

3.1 In-Order Checkpoint Buffer

Figure 2 illustrates an in-order checkpoint buffer. The buffer consists of two parts: a cyclic buffer that stores checkpointed versions of the program counter

and a hardware stack for maintaining the rest of the information included in a checkpoint. The cyclic buffer supports random access to all the checkpointed PCs. This allows any PC to be restored as soon as a rollback is started after detecting a misprediction, so that instructions from the correct path can be fetched without delay. Every checkpoint has an instruction counter (not drawn) that keeps track of the number of outstanding instructions for that checkpoint. The head of the cyclic buffer is incremented only if the counter of the pointed checkpoint is zero, thus all relevant instructions executed.

The second part of the checkpoint buffer is managed as a stack, using two control signals: *Save* that allocates a new checkpoint by performing a stack-push, and *Restore* that flushes a single checkpoint by performing a stack-pop operation. Obviously, a full rollback may require multiple pop operations, thus multiple cycles. The multiple cycle restoration process is usually acceptable because it takes several cycles until new instructions from the correct path reach the renaming stage of the pipeline. In Figure 2, checkpoint three (CP_3) is the bottom of the stack, and checkpoint six (CP_6) is the top. There is no need for a “top of stack” pointer—the closest copy (CP_6) to the active copy is always the top of the stack. The checkpointing stack is a little unusual because it also supports a removal operation from the bottom of the stack, which is used to commit and remove the oldest checkpoint. A checkpoint is committed by simply decrementing a counter that holds the number of checkpoints.

In the example in Figure 2, the checkpoint buffer stores four valid checkpoints, in which checkpoint CP_3 is the oldest and CP_6 is the newest. When a rollback occurs, the content of a checkpoint must suffice to restore the program state. The following list describes the fields saved in a checkpoint. Assume checkpoint CP is taken at instruction I_{CP} .

PC. The PC field contains the address of I_{CP} , with the exception of branch instructions, in which case it contains address of the following instruction (i.e., $PC+4$). This optimization saves re-executing the branch instruction itself after rollback.

Branch target address. If I_{CP} is a branch, this field contains the target address. Indirect branches update the target address field at a later time when they are resolved. This update may conflict with the insertion of a new checkpoint, in which case the new checkpoint is given priority. Table II shows which address field will be used for recovery, PC or the branch target address.

Mapping table. A list of registers corresponding to the architectural registers. In this example, the value of the first architectural register (R_1) is located in the first register (P_1) for checkpoints CP_3 and CP_4 , and in the 125th register (P_{125}) for checkpoints CP_5 and CP_6 . In a similar manner, the values of architectural registers R_2 , R_3 , and R_4 are located in registers P_{126} , P_3 , and P_4 , respectively, for CP_3 . The list format described here matches the mapping table structures that are based on a RAM [Sangireddy 2006]. Other structures, such as CAM-based mapping tables, save equivalent information [Cristal et al. 2004].

RegMapped flags. A flag per register indicating if it is part of the mapping table. The RegMapped flags are part of the early register release mechanisms the microarchitecture uses to attain satisfactory performance with a reduced-size

Table II. Selecting the Starting Address of the Recovery

Branch _{CP}		Branch _{CP+x} prediction correct	Recovery begins from
Prediction	Prediction correct		
Taken	No	N/A	Address in PC field
Not taken	No	N/A	Branch target address
N/A	Unknown	No	Recovery cannot complete before Branch _{CP} is resolved
Taken	Yes	No	Branch target address
Not taken	Yes	No	Address in PC field

The checkpoint CP was taken at branch instruction Branch_{CP}. Down the instruction stream, there is another branch, Branch_{CP+x} and CP is the closest checkpoint to it. The same checkpoint CP is used for recovery for both branches.

register file [Akkary et al. 2004]. The first RegMapped flag in the bit vector indicates P_1 , and according to the example in Figure 2, it will represent an architectural register if either checkpoint CP_3 or CP_4 will be recovered, but not for CP_5 or CP_6 . The information in this field is a subset of the information available from the mapping table field. The baseline processor uses two data structures, the mapping table for renaming and the RegMapped flags for register release management. Checkpoints also hold this information in two separate fields in order to refrain from the need to read the entire content of the mapping table and calculate the RegMapped flags.

3.2 Adding Optional Information

In the example in Figure 2, checkpoints contain information required for functionality. Checkpoints may contain additional information intended to achieve better performance, such as predictor structures that reflect the state of recent history at the time the checkpoint was saved. Restoring the state of predictors on rollback removes updates made by wrong-path instructions. Another solution is to avoid wrong-path updates by delaying the predictor's modification until the commit stage, but this requires a fine-grained bookkeeping method, such as a ROB, which our checkpoint-based microarchitecture does not have.

Skadron et al. [1998, 2000] studied restoring the state of the return address stack (RAS) and branch predictor in ROB-based processors. Such processors use a dedicated queue for outstanding branches, a structure whose size is proportional to the instruction window. On the other hand, checkpoint-based microarchitectures, typically use a small number of checkpoints, thus the number of stored states is small. Moreover, when the global history of the predictor is part of the checkpoint, it is saved and restored using the existing checkpoint management. (Following previous work [Skadron et al. 2000], we save only the global history for branch predictors and confidence estimators.)

Table III summarizes the miss rate and IPC speedup as a function of restoring three predictors: RAS, TAGE [Seznec and Michaud 2006], and JRS [Jacobsen et al. 1996]. The state of the RAS predictor consists of a pointer

Table III. Miss Rate and IPC Speedup after Restoring the State of Several Predictors

Predictor		Integer		Floating point		Cost per checkpoint (bits)
		Not restored	State restored	Not restored	State restored	
RAS index	Miss rate	12.4%	9.2%	9.9%	6.0%	3
	Speedup		2.0%		0.6%	
RAS index and values	Miss rate	12.4%	5.0%	9.9%	3.8%	256
	Speedup		4.6%		1.1%	
Branch predictor	Miss rate	4.1%	4.0%	0.8%	0.8%	131
	Speedup		0.2%		0.0%	
Confidence estimator	(1—SPEC)	19.0%	20.2%	35.6%	43.3%	12
	(1—PVN)	87.3%	87.5%	92.3%	92.4%	

The quality of the confidence estimator is measured using two indicators, *SPEC* (specificity), which is the fraction of branch mispredictions that were estimated as low-confidence predictions, and *PVN* (predictive value of a negative test), which is the fraction of low-confidence predictions that were in fact mispredictions. For consistency with miss rates, we show (1—SPEC) and (1—PVN), so that a lower value is better.

to the top of stack and the stack content. Restoring the state of the RAS improves performance. It reduces the RAS miss rate from 12.4% to 9.2% when the index is included in the checkpoint, and to 5.0% when both the index and the values are restored. This reduction translates to 4.6% mean speedup for the integer benchmarks. The floating point benchmarks show similar trends but are less affected because they have fewer mispredictions.

The branch predictor state includes a long global history (refer to the rightmost column of Table III). Restoring it results in mixed results, for instance the miss rate and IPC improve in Gcc, Mcf, and Parser but deteriorate in Bzip, Perlbnk, and Vortex.

The last predictor we evaluate is the confidence estimator. The confidence estimator state includes the global history shift register. Restoring it during rollback actually lowers accuracy, reducing both PVN and SPEC. Therefore, instead of restoring the old state, we simply reset the global history shift register. This approach reduces the (1—SPEC) from 19.0% to 17.3% and the (1—PVN) from 87.3% to 84.1% for the integer benchmarks, and of course, it saves the space that would be required to save the confidence estimator state.

Considering that the performance gain and implementation cost (rightmost column of Table III) greatly varies from one predictor to another, we do the following for the remainder of this article. We restore the state of the RAS and reset the confidence estimator history (when the estimator is used) but leave the branch predictor state unchanged.

4. CHECKPOINT ALLOCATION

Checkpoint-based processors save the state of the program periodically, according to a checkpoint allocation policy, providing that a free checkpoint exists. The policies we examine allocate a checkpoint at the occurrence of any one of the following events:

Table IV. Breakdown of Checkpoints Allocated by the Different Checkpoint Allocation Policies

Benchmarks	Checkpoint policy	Branches	Instruction threshold	Store threshold
Integer	LowConfBr	98.0%	1.0%	1.0%
	MinDist	99.6%	0.0%	0.4%
	Oracle	87.5%	6.6%	5.9%
	Adaptive	99.9%	0.1%	0.0%
Floating point	LowConfBr	71.1%	22.8%	6.1%
	MinDist	96.4%	0.3%	3.3%
	Oracle	22.5%	62.9%	14.6%
	Adaptive	96.1%	3.5%	0.4%

- (1) at certain branches, as described later,
- (2) at the first branch after a rollback,
- (3) when the number of instructions after the last checkpoint exceeds $Max_{threshold}$, and
- (4) when the number of store instructions after the last checkpoint reaches $St_{threshold}$.

The objectives of items 2 through 4 in the previously mentioned list are to: guarantee forward progress (Item 2), set an upper bound on the maximal re-executed code segment (Item 3), and prevent deadlocks in which a single checkpoint occupies all store-queue entries, execution halts, a consecutive checkpoint is never allocated, and as a result the store-queue entries are not recycled and execution cannot continue (Item 4). The $St_{threshold}$ could also have been set to lower values in order to recycle store queue entries more frequently. Nevertheless, experimenting with such values did not improve the average performance.

The first item in the list (branches) is the most frequent reason for allocating a checkpoint (refer to Table IV). Therefore, we name each checkpoint allocation policy by the way it selects the branches at which checkpoints will be taken. Although every policy is different, they all share the same objective—to prevent the checkpoint buffer from clogging while keeping the average re-executed code segment reasonably short.

Our baseline policy *LowConfBr* allocates checkpoints at branches that are likely to be mispredicted. The *LowConfBr* policy uses a branch prediction confidence estimator to allocate checkpoints on hard-to-predict branches. An *Oracle* policy is also used to understand the potential speedup that can be obtained by any checkpoint policy. It takes a checkpoint at mispredicted branches and at the occurrence of any of the events listed in Items 2 through 4. It is essentially a *LowConfBr* policy that uses a perfect (oracle) confidence estimator. Finally, we model another known policy *MinDist*, for completeness. The *MinDist* policy does not use a confidence estimator, instead it filters branches by setting a minimum distance ($Min_{threshold}$) between consecutive checkpoints.

Table IV illustrates how the checkpoint allocation policies are translated to actual checkpoints. The integer benchmarks are characterized by many nontrivial branches, and even the oracle policy allocates checkpoints mainly for branch mispredictions. In the floating point benchmarks, however, branch prediction is

Fig. 3. IPC speedup of the MinDist and oracle checkpoint allocation policies, normalized to the performance of the LowConfBr policy.

more accurate and oracle allocates fewer checkpoints to mispredicted branches, leaving in the buffer-free checkpoints that are used when crossing instruction- and store-count thresholds.

The performance of the LowConfBr, MinDist, and Oracle checkpoint allocation policies is compared in Figure 3. LowConfBr policy significantly outperforms the MinDist policy for the integer benchmarks and is nearly equivalent for floating points benchmarks. This justifies choosing LowConfBr as the baseline policy. However, LowConfBr is far from reaching its potential, represented by the oracle policy. The potential mean IPC speedup is 9.79% and 3.88% for the integer and floating point benchmarks, respectively. This is despite the fact that 57.5% of LowConfBr recoveries were to checkpoints taken at the mispredicted branches, thus requiring no re-execution.

Trying to narrow the performance gap between LowConfBr and the oracle by allowing the JRS [Jacobsen et al. 1996] to train longer, enlarging the JRS-table size, or replacing the confidence estimation algorithm did not succeed. Quadrupling the JRS table size or switching to an enhanced JRS estimator [Grunwald et al. 1998] did not improve the mean performance results by more than 0.25%. We did not quantify LowConfBr with more complex structures, such as a perceptron-based estimator [Akkary et al. 2004].

Designing an accurate confidence estimator becomes harder when branches are predicted better (see Figure 4). The reason for this is that predicting the branch direction and estimating its confidence level are related issues [Grunwald et al 1998]. For example, reliable confidence estimation algorithms can be made to part of the branch predictor by reversing its original prediction if estimated to be wrong. The confidence estimator, however, will no longer be as reliable when estimating the outcome of the aforementioned enhanced branch predictor. Our simulation model uses a state-of-the-art TAGE [Seznec and Michaud 2006] branch predictor, which explains the low quality of the confidence estimator (as presented in Table III). Confidence estimation is going to

Fig. 4. Quality of the branch prediction confidence estimator as a function of the branch predictor hit rate, as measured on three different instruction intervals for the integer benchmarks. The quality of the confidence estimator is measured using SPEC and PVN (defined in Table III), for both indicators—higher values mean better quality. For clarity of comparing the trends on a single scale, all results are normalized to the results of rightmost configuration.

Fig. 5. An example scenario of a rename stage halt due to a lack of an available checkpoint. Note that the instruction window is guaranteed to be larger than $Max_{threshold}$.

be even harder predictors are invented. For this reason, our attempt at closing the gap with the Oracle policy follows a different path, one that does not use a confidence estimator.

4.1 Adaptive Checkpoint Allocation Policy

In this section, we introduce a new adaptive checkpoint allocation policy. The adaptive policy is based on the observation that a checkpoint allocation policy should not maintain a large average distance between consecutive checkpoints. A large distance is desirable only when a very large instruction window is beneficial or when the registers are overutilized and are not reclaimed efficiently due to excessive use of checkpoints. Otherwise, checkpoints are taken at each branch and indirect jump, and at the events listed in Items 2 through 4 in the list at the beginning of Section 4.

The adaptive policy identifies three scenarios indicating that a large distance between consecutive checkpoints is beneficial. In all three scenarios, the detection and the response are simple to implement. The detection of a scenario requires an additional counter at most, and the response is limited to changing $Min_{threshold}$.

Second-Level Cache Misses. The baseline microarchitecture provides a reasonably large instruction window regardless of the checkpoint allocation policy. The instruction window size consists of two parts (Figure 5), a segment of

Table V. The Adaptive Policy Parameters and their Values

Symbol	Value	Available in baseline policy	Comments
$Max_{threshold}$	256	yes	
$St_{threshold}$	32	yes	
$RF_{utilization}$	90%	no	used to vary $Min_{threshold}$
$CB_{utilization}$	50%	no	used to vary $Min_{threshold}$
$Mark_1$	512	no	used to vary $Min_{threshold}$
$Mark_2$	1024	no	used to vary $Min_{threshold}$

$Min_{threshold}$ value changes dynamically between 0, 64, 128, and 256.

instruction execution marked by checkpoints and a segment of up to $Max_{threshold}$ instructions executed after the checkpoint buffer is exhausted. The number of instructions in the first segment is determined by the average distance between the checkpoints and the checkpoint buffer depth. By allowing the processor to continue after the checkpoint buffer is full, we support a large instruction window even though the distance between checkpoints in the first segment may be quite small. With $Max_{threshold}$ set to 256, as in Table I, the instruction window is large enough to exploit the available instruction-level parallelism (ILP) and to overcome execution unit and cache latencies, with the exception of second-level cache misses.

Second-level cache misses may take hundreds of cycles to resolve. Therefore, once a second-level cache miss is detected, the adaptive checkpoint allocation policy changes $Min_{threshold}$ from 0 to 64, which is the value used in the MinDist policy. $Min_{threshold}$ returns to 0 after all the second level cache misses were resolved. Processors with smaller checkpoint buffers or longer memory access times can set $Min_{threshold}$ higher.

Resource Awareness. Unnecessary checkpoints may cause renaming to halt due to lack of registers. The reason for this is that every checkpoint prevents registers that are part of its mapping table from being reused until the checkpoint itself is freed. Nevertheless, predicting unnecessary checkpoints is a difficult task, as we have seen in the LowConfBr checkpoint allocation policy. Moreover, increasing the interval between checkpoints results in longer checkpoint lifetime, which means that registers belonging to checkpoints are held longer. There is a trade-off between sparse checkpoints, which increase register pressure by holding checkpointed registers longer, and dense checkpoints, which may also cause register pressure by increasing the number of checkpointed registers. Checkpointed registers are registers that are pointed by the mapping table saved in a checkpoint, but may be freed as soon as the checkpoint itself is released.

In order to minimize the allocation of unneeded checkpoints, the adaptive policy combines both a direct and indirect approach. The direct approach is resource conscious. It constantly monitors the availability of registers and checkpoints. If the utilization of the register file is larger than $RF_{utilization}$, and the utilization of the checkpoint buffer is larger than $CB_{utilization}$ (refer to Table V), the adaptive checkpoint allocation policy tries to improve register reclamation by taking fewer checkpoints. Taking fewer checkpoints is achieved by changing

Fig. 6. Illustration of checkpoint allocation when rollback history is taken into account. At first (I.), checkpoints are taken on every branch ($Min_{threshold} = 0$). After the $CPsSinceRbck$ counter reaches $Mark_1$ (II.), $Min_{threshold}$ changes so that a new checkpoint can be allocated only after exceeding half $Max_{threshold}$ instructions. When the $CPsSinceRbck$ counter reaches $Mark_2$ (III.), the checkpoint allocation policy changes so that a new checkpoint is taken exactly every $Max_{threshold}$ instructions. The $CPsSinceRbck$ counter is reset and the checkpoint allocation policy returns to its first state (I.) upon a new rollback.

$Min_{threshold}$ from 0 to 64. $Min_{threshold}$ returns to 0 when the register file or checkpoint buffer utilization decreases below $RF_{utilization}$ and $CB_{utilization}$, respectively.

Rollback History Approach. The indirect approach to improve register reclamation is based on rollback history. Most programs have code segments that are highly predictable. During the execution of these segments, the processor almost never performs a rollback, and the exact number of the few instructions to be reexecuted has an unnoticeable impact on performance. On the other hand, taking fewer checkpoints improves register reclamation, which may be especially meaningful when prediction is successful and resources are needed to exploit the available parallelism in the predicted instruction trace.

To implement the indirect approach, we define a counter $CPsSinceRbck$ that counts checkpoints taken since the last rollback. We also define two marks ($Mark_1$ and $Mark_2$, see Table V), and raise $Min_{threshold}$ when $CPsSinceRbck$ reaches one of them. Figure 6 shows that $Min_{threshold}$ is set to $\frac{Max_{threshold}}{2}$, when the checkpoint count reaches $Mark_1$. Reaching $Mark_2$ sets $Min_{threshold}$ to $Max_{threshold}$, which is the highest value possible.

The direct and indirect approaches reduce the number of checkpointed registers by 2.2% and 7.0% for the integer and floating point benchmarks, respectively. The limited success in reducing the number of checkpointed registers is caused to some extent by the previously mentioned trade-off between the number and lifetime of checkpointed registers. There is also another factor to be considered. The number of different checkpointed registers between two adjacent checkpoints increases with the distance between the checkpoints. This factor negates some of the effect of reducing register pressure by increasing the distance between checkpoints.

4.2 Performance of the Adaptive Policy

Figure 7 shows the performance of the adaptive checkpoint allocation policy. The adaptive policy achieves a mean IPC speedup of 6.80% and 2.20% for the integer and floating point benchmarks, respectively. The achieved speedup is most to the potential shown by the oracle policy, which is 9.79% and 3.88%. The adaptive policy improves the performance for all the integer and most of the floating point benchmarks.

Fig. 7. IPC speedup of the adaptive and oracle checkpoint allocation policies.

Fig. 8. Breakdown of the IPC speedup of the adaptive checkpoint allocation policy.

The contribution of each adaptive method to the performance of the adaptive checkpoint allocation policy is presented in Figure 8, beginning with a trivial policy that takes a checkpoint at every conditional branch, and adding the three methods that construct the adaptive policy, one at a time, by the order of their nonoverlapping contribution. The performance of the integer programs is dominated by mispredictions and re-execution, and therefore taking a checkpoint every branch works well for most integer benchmarks but not for the floating point benchmarks. Changing the checkpoint allocation policy in the presence of second-level cache misses and clean rollback history records are both useful and account for nearly all of the speedup achieved by the adaptive policy. This, however, is not the case for the resource awareness method. The benefits of resource awareness are clear when the method is applied on top of the trivial policy (0.76% speedup, not shown in Figure 8), but vanish when the method is applied on top of the other two methods. In the latter case, the contribution

Fig. 9. Checkpoint allocation policies performance as a function of the checkpoint buffer depth. All measurements are normalized to the results of the LowConfBr policy with parameters as presented in Table I (checkpoint buffer depth of eight).

Fig. 10. Checkpoint allocation policies performance as a function of the memory access latency. All measurements are normalized to the results of the LowConfBr policy with parameters as presented in Table I (memory access time of 200 cycles).

overlaps that of the other two methods and even though the mean speed-up is positive, there are several benchmarks that experience a minor slowdown.

Figure 9 shows that, in the range of 4 to 16 buffer entries, the performance gap between the adaptive policy and the two other policies we consider (LowConfBr and MinDist) grows with the depth of the checkpoint buffer. The reason for this is that during certain time intervals (when $Min_{threshold}$ is zero) the adaptive policy allocates checkpoints at every branch instruction, and is able to take advantage of deeper checkpoint buffers. Figure 10 shows that, in the range of memory latency between 100 to 600 cycles, the adaptive policy is always superior. The performance gap, however, between adaptive and the other two policies, LowConfBr and MinDist, diminishes when the memory latency grows. For the integer benchmarks, the relative advantage starts at 8.1% at 100 cycles and is reduced to 7.1%, 5.8%, and 4.9% at 200, 400, and 600 cycles, respectively. The reason for this is that the adaptive policy adjusts in response to L2 miss events, which are discovered more than a dozen cycles after the missing load instructions were renamed. In that period of time, checkpoints were allocated using the previous $Min_{threshold}$ value.

Fig. 11. Number of re-executed instructions as a function of the checkpoint buffer depth. All measurements are normalized to the results of the LowConfBr policy with parameters, as presented in Table I.

The benefit of the adaptive policy comes primarily from the substantial reduction in the number of re-executed instructions. Figure 11 shows that for the integer benchmarks, at a buffer depth of eight, the number of instructions re-executed by the adaptive policy is less than $\frac{1}{8}$ and $\frac{1}{3}$ of those re-executed by the MinDist and the LowConfBr policies, respectively. The reported reduction is comprised of both a 25% reduction in the average length of the code segments and 83% and 58% reduction in the number of re-executed segments for the MinDist and LowConfBr policies, respectively. Another characteristic of the LowConfBr policy is the large variation in the length of the re-executed code segments. Of the re-executed code using LowConfBr 9.7% consist of segments longer than 80 instructions, whereas in the adaptive policy, long 80+ instruction segments are much less frequent (2.4%). Considering again Figure 11 and the integer benchmarks, the MinDist and LowConfBr policies are not successful in using a checkpoint buffer deeper than eight entries to decrease the number of re-executed instructions. The floating point benchmarks show similar improvement trends for the adaptive policy. Most floating point programs re-execute an order-of-magnitude fewer instructions (11% compared to the integer benchmarks), hence the minor effect on speedup.

Another contributor to the performance of the adaptive policy is a reduction in the number of wrong-path instructions. The number of instructions executed while following a mispredicted path is reduced by 18.7% and 12.9% for the integer benchmarks, relative to the MinDist and LowConfBr policies, respectively. These numbers are comprised of both a 4% reduction in the average length of wrong path code segments and 14.8% and 9.5% reduction in the number of recoveries. The main reason for the reduction in wrong-path instructions is the adaptive changes in the instruction window size. The adaptive policy runs code segments, which are prone to mispredictions, with a *CPsSinceRbck* counter value that is lower than the *Mark₁* threshold. In most cases, this results in a *Min_{threshold}* value of zero. Recalling that the instruction window has two parts (refer to Figure 5) and that the size of its first part depends on the average distance between checkpoints, it is evident that a small *Min_{threshold}* will lead to a smaller window size. In code segments that are prone to mispredictions, a

Fig. 12. Example scenario in which a later allocated checkpoint (CP_4) is ready to be released before the earlier checkpoint (CP_3) can be committed.

smaller window size is beneficial in two ways: It can prevent additional wrong-path instructions from being fetched and executed, or it can prevent a branch misprediction by fetching the branch at a later time, when the predictor is better trained. This by-product of the adaptive policy resembles the pipeline gating method described in Manne et al. [1998] to reduce the processors dynamic power consumption. Although pipeline gating and the adaptive policy are very different mechanisms, they are both effective because they reduce the instruction window size in code segments that are prone to mispredictions.

5. CHECKPOINT RELEASE

The in-order checkpoint management scheme presented in Figure 2 commits (and releases) checkpoints from the bottom of the stack and discards wrong-path checkpoints from the top (by doing one or more shifts into the active copy). However, it is unable to release other checkpoints, such as CP_4 in Figure 2, that are not at the bottom of the stack. CP_4 becomes useless (will never be used for recovery) when all the instructions belonging to CP_4 (i.e., starting with CP_4 and up to but not including CP_5) have finished execution without causing a rollback.

Most often checkpoints become ready for release in order. Figure 12, however, shows a scenario in which CP_4 is ready to be released before CP_3 . CP_3 has two pending instructions, a load to register `R1` and a data-dependent branch. On the other hand, instructions that belong to CP_4 (`Inst4A` to `Inst4Z`) have all finished execution; none of these instructions had a data dependence on register `R1`. Note that CP_4 can be released, but the instructions belonging to CP_4 cannot commit because a branch (`BEQ R1`) preceding CP_4 is still unresolved. If the branch is mispredicted, the program has to be restored from CP_3 .

Not releasing useless checkpoints is inefficient, it prevents reclaiming two resources: checkpoints and checkpointed registers. We now present two solutions to deal with this inefficiency: (i) *OOO checkpoint release* and (ii) a streamlined version of it, in which registers but not checkpoints are released out-of-order.

Fig. 13. IPC speedup of the adaptive policy with OOO checkpoint release, compared with the adaptive and oracle policies of in-order checkpoint management.

5.1 Out-of-order Checkpoint Release

Recall the in-order checkpoint buffer structure presented in Figure 2. Supporting OOO checkpoint release requires a random access structure rather than a stack. This increases the implementation cost (area, power) and the capacitance on the active copy, which may affect the critical timing path. OOO checkpoint release also complicates the management logic itself. When a rollback occurs, it must determine which checkpoints are valid (preceding the misprediction) and which should be purged. To release a checkpoint *CP*, it must identify the most-recent valid checkpoint preceding *CP*. One way to simplify the management issues is by introducing physical to logical checkpoint mapping tables so that the same physical checkpoint can be reused under different logical names. Other than the additional rename logic and table, this solution typically requires an extra bit per checkpoint tag, throughout the pipeline.

The OOO checkpoint release speed-up (Figure 13) primarily derives from reducing the number of re-executed instructions (by 34.7% for the integer benchmarks), and it varies across the benchmarks. Eon, Gap, Twolf, Art, and Sixtrack benchmarks gain the most from OOO checkpoint release. These are also the benchmarks in which OOO release reduces the highest numbers of re-executed instructions, and all of them also benefit from a deeper in-order checkpoint buffer.

5.2 Early Release of Checkpointed Registers

In this section, we consider early release of checkpointed registers while maintaining the in-order checkpoint management. Figure 14 illustrates the baseline register file and additional information used to manage checkpoints and release registers. The register file is a checkpoint-enhanced version of the proposal of Moudgill et al. [1993]. Associated with each register there is a *RegMapped* flag and *N* checkpointed copies of it. A register usage (*RegUse*) counter is used in order to track the number of instructions that will read the register value. Unlike

Fig. 14. Baseline register file and management information.

some ROB-free microarchitectures [Akkary et al. 2004], the *RegUse* counter is not incremented and decremented by all instruction, but only by the subset carrying the same checkpoint tag. This scheme works well because every instruction from a later checkpoint is represented by a checkpointed copy of the *RegMapped* flag. A register is deallocated when the *RegUse* counter, the *RegMapped* flag, and its checkpointed copies are all zero. As Figure 14 shows, the register file management can reset the *RegUse* counter and a checkpointed copy of the *RegMapped* flag in order to flush checkpoints taken during the wrong path. Aiming at freeing as many checkpointed registers as possible, we use a subset of the described flush mechanism for another purpose. Once all the instructions that belong to a checkpoint have finished execution and regardless if the checkpoint is located at the head of the checkpoint buffer, we reset the relevant checkpointed copy of the *RegMapped* flag (leaving the *RegUse* counter value as is). This may allow some checkpointed registers to be marked as free. The checkpoint resource itself is not freed and cannot be reused until it commits in an in-order fashion.

Simulation of the proposed OOO checkpointed-register release scheme achieved a small fraction of the speedup of the OOO checkpoint release scheme. The small IPC speedup has two main reasons (i) only a small subset of the checkpointed registers can be freed by OOO checkpoint release, and (ii) register availability is not a dominant performance bottleneck in the baseline configuration (Table I). In order to evaluate the second reason, we repeated the simulation for another configuration, using half-size register files. In this configuration, registers were the main bottleneck and the streamlined scheme achieved most of the speedup achieved by the more complex OOO checkpoint release scheme.

6. LEAKAGE POWER OF CHECKPOINTED AND RELEASED REGISTERS

Power consumption of register files has been an active research topic in recent years. Although power is not the primary objective of this article, there are two aspects of checkpoint-based microarchitectures, namely early register reclamation and checkpointed registers (which are only used for recovery), that present an opportunity for saving leakage power that has not been previously investigated. The motivation for reducing the power consumption of the register file is twofold: the register file consumes a large fraction of the total power (anywhere

Fig. 15. Integer register file average usage. Register breakdown: 32 (Alpha ISA) registers that are in the rename mapping table; checkpointed registers; other active registers (needed by dependent instructions, but excluded from the current mapping table); and free registers.

between 10% and 25% according to Ergin et al. [2004]), and it does so on a relatively small area, thus creating a high-power density spot. Leakage power consumption accounts for nearly half of the total power in some manufacturing technologies [Maliniak 2007], hence our focus on its reduction.

Many registers are not accessed for long periods of time. Checkpointed registers will only be used if, following a rollback, they are made part of the speculative mapping table and new instructions read them as operands. Free registers will not be accessed until they are allocated to instructions and these instructions reach the writeback stage. Checkpointed and free registers provide an indication that they may be accessed again soon, either during rollback (checkpointed registers) or when allocated at the rename stage (free registers). Several leakage power reduction techniques, such as gating the supply voltage [Powell et al. 2000] or reducing the potential difference [Kim et al. 2004], require a cycle or more to return to the regular mode that allows the cell to be accessed. Getting a warning several cycles before free or checkpointed registers may be accessed is important in order to maintain performance.

Figure 15 depicts the average register file usage, showing that many registers are free or checkpointed. On average, there are 44.8 and 52.8 free registers and 11.8 and 16.3 checkpointed registers for the integer and floating point benchmarks, respectively. The high number of free registers does not imply that the selected register file is too large. It only indicates that there is a high variance in register demand across benchmarks and across code segments of the same benchmark.

The last issue we consider before presenting our scheme is the cost of detecting the register type. The baseline register management (shown in Figure 14) already detects free registers. Detecting checkpointed registers, on the other hand, requires some additional logic in the baseline microarchitecture, but is straightforward in other checkpoint-based microarchitectures, such as CPR [Akkary et al. 2004].

Fig. 16. (a) Per register management logic and (b) the way it is attached to the register memory cells. The logic added to support the off and drowsy states are two (rather large) transistors shown in the shaded area.

The proposed leakage power reduction scheme is fine grained. Each register can be in one of three states: active, drowsy, and off, and the transition between states is controlled by the existing early register reclamation mechanism. Figure 16(a) shows the register management logic, the upper part is the baseline logic that calculates if the register is free and could be reused. A new n -type transistor is used to create a virtual ground that is connected to the register's cells instead of the regular ground. If the register is not free, the transistor will be on, connecting the cells to the ground. But when the register is free and until *RegMapped* flag is set, the transistor is off, suppressing leakage from both the cell's supply voltage and from the charged bit lines. The proposed scheme is not free of disadvantages; the added transistor adds area and access latency. Powell et al. [2000] presented several methods to minimize these negative effects. When the n -type transistor is off, a p -type transistor connected in parallel may place the register in drowsy state by setting the virtual ground to V_t .

We use the following formulas to estimate the power reduction:

$$(1) \quad LP_{base} = LP_{dec} + LP_{sa} + LP_{dr} + LP_{wl} + LP_{bl}$$

$$(2) \quad LP_{reduced} = LP_{dec} + LP_{sa} + LP_{dr} + LP_{wl} + (LP_{bl} \times (1 - \%RF_{off} - \%RF_{drowsy} \times k))$$

The variables in the previous equations are defined as follows:

LP_{dec} - Leakage power dissipation for the address decoder.

LP_{sa} - Leakage power dissipation for the sense amplifier.

LP_{dr} - Leakage power dissipation for the data output drive.

LP_{wl} - Leakage power dissipation for the word line.

LP_{bl} - Leakage power dissipation for the bit line.

LP_{base} - Leakage power dissipation for the baseline register file.

$LP_{reduced}$ - Leakage power dissipation for the proposed register file.

$\%RF_{off}$ - The fraction of registers that are in the off state.

$\%RF_{drowsy}$ - The fraction of registers that are in drowsy state.

k - The fraction of leakage power saved in drowsy state.

Table VI specifies the leakage power reduction of the integer and floating point benchmarks. The power reduction is significant, the leakage power is reduced to 47.66% and 51.44% of the original value for the integer and floating point benchmarks, respectively.

Table VI. Leakage Power Reduction Achieved by Making Checkpointed Registers Drowsy and by Turning Free Registers Off

			Integer benchmarks	Floating point benchmarks
Base		[mW]	90.071	90.071
Power saved	Checkpointed registers	[mW]	3.550	8.254
	Free registers	[mW]	43.588	35.481
	Total	[mW]	47.139	43.735
Power after reduction		[mW]	42.932	46.336
		[%]	47.66	51.44

The data for power saved by putting registers in drowsy mode is based on $k = 0.9$. Note that the major part of the power reduction is coming from turning free registers off, hence even a 10% error in the value of k will have a minor effect on the bottom line.

7. RELATED WORK

Most of this section surveys microarchitectures that use checkpoints, with emphasis on their goal, the way the checkpoints are allocated, used, and released. At the end of this section, we discuss previous work on reducing the leakage power consumption of register files. We begin our survey at the left side of Figure 1. In IBM's Power6 processor [McGhan 2006] certain parts of the pipeline and control are protected using parity and fault screening methods [Racunas et al 2007]. Every cycle, a new checkpoint is created so that the processor state can be restored if a soft or hard error is detected. The checkpoint-based recovery technique is inspired by highly reliable mainframe systems [Meaney et al. 2005].

Cherry [Martinez et al. 2002; Kirman et al. 2005] is a resource-reclamation ROB-checkpoint hybrid that employs a single checkpoint to speculatively reuse resources, such as registers and load/store queue entries, prior to committing the instruction they were allocated to. Cherry is based on a conservative approach to reuse. For example, it does not initiate a reuse beyond unresolved branches. A less conservative recycling approach was studied by Ergin et al. [2006], who proposed an early register reclamation mechanism implemented using a "checkpointed register file." In a checkpointed register file, every register has a shadow register that can be used to back-up values that may be needed in case of misspeculation. The shadow registers are used in conjunction with the ROB to restore the processor state.

High-frequency processors require a high number of cycles to access the main memory [Wilkes 2001]. Load instructions that miss the cache hierarchy, and their data-dependent instructions clog the ROB, which causes the processor to halt until the miss is resolved. Memory wall hybrids divide into two major categories, depending on the action they take to unclog the ROB. Mutlu et al. [2003] introduced the RunAhead microarchitecture that unclogs the ROB by skipping the long latency load and its data-dependent instructions. RunAhead has two modes: regular and prefetch. It switches to the prefetch mode on a L2 cache miss and returns to the regular mode when that load is resolved. In its basic form, RunAhead does not use checkpoints for speculation, the checkpoint taken on switching from regular to prefetch mode is always restored, flushing all

the results calculated during the prefetch mode. The goal of the prefetch mode is to train the caches and branch predictor. Ramirez et al. [2006] compared the runahead and checkpoint-based microarchitectures and found that the latter achieves significantly better performance on a lower power budget.

Memory wall hybrids that belong to the second category unclog the ROB by employing a load value predictor [Lipasti and Shen 1996]. Kirman et al. [2005] and Ceze et al. [2006] predict the load value upon L2 misses, use 4- and 1-deep checkpoint buffers, respectively, and perform a rollback for load value mispredictions.

Many modern processors have one or two mapping table instances, a speculative copy, used at the rename stage and optionally a nonspeculative copy, updated at the commit stage. These mapping table instances are in sync with the ROB's tail and head pointers, respectively. On recovery, the ROB's tail is adjusted and a new speculative mapping table is required. One way to achieve this is to delay the recovery until the mispredicted instruction reaches the commit stage. Better performance is achieved, however, by initiating the recovery as soon as the misprediction is detected. The new speculative mapping table is created in two phases: (i) one of the two existing table instances is copied and then (ii) it is adjusted by traversing all ROB entries from the reference point to the point of misprediction. Hybrid processors periodically allocate checkpoints, and have more than two mapping table instances that can be used as reference points. This reduces the average number of ROB entries to traverse, which in turn reduces the recovery time.

A faster recovery hybrid approach was implemented in two processors: the MIPS R10000 [Yeager 1996] and the Alpha 21264 [Kessler 1999]. These processors allocate a checkpoint at every branch. Such aggressive checkpoint allocation policies avoid the need to traverse the ROB but do not scale to large instruction windows. To solve this scalability limitation, Moshovos [2003] describes a more economic checkpoint allocation policy that takes checkpoints only at hard-to-predict branches. Following this direction, Akl and Moshovos [2006] presented BranchTap, a microarchitecture that avoids further speculation when the newly fetched instructions are at high risk of being on the wrong path. The adaptive pipeline gating decreases the mean instruction window, so the limited number of checkpoints available is better dispersed and the number of ROB-entries that have to be traversed following a rollback is reduced. The authors have also evaluated OOO checkpoint release, reaching similar conclusion to ours.

Checkpoint allocation policies used by ROB-free and faster recovery hybrids share a similar purpose—to reduce the number of instructions that need to be processed following a misprediction. Figure 17 illustrates the recovery process such microarchitectures perform following the detection of the same branch misprediction (BEQ_i). The hybrid performs a rollback to BEQ_i and updates the mapping table starting at either CP_2 or CP_3 and adjusting it by traversing $Inst_g$ to BEQ_i or $Inst_m$ to BEQ_i . ROB-free processors on the other hand, perform a rollback to $Inst_g$ but can immediately reexecute from there. In both cases, the overall recovery penalty depends on the number of instructions that need to be processed (traversed or re-executed). There are also significant differences.

Fig. 17. A recovery scenario comparison between ROB-free processors (F) and faster recovery hybrids (H).

Checkpoints in faster recovery hybrids affect neither the instruction window size nor the efficiency of register reclamation. On the other hand, checkpoints in ROB-free processors are an integral part of the bookkeeping method. Allocating them consumes checkpointed registers and can reduce the size of the instruction window. Furthermore, instructions processed through the pipeline carry the tag of the checkpoint they belong to. Releasing checkpoints OOO requires setting new values to these tags.

Reaching ROB-free microarchitectures on the right side of Figure 1, Hwu and Patt [1987] were the first to propose the use of checkpoints to recover the processor state following a misspeculation. Two main microarchitectures stand at the basis of recent work on coarse-grained bookkeeping: the Kilo-instruction processor [Cristal et al. 2004] and CPR [Akkary et al. 2004]. Both microarchitectures are resource efficient and are designed for an order-of-magnitude larger instruction window. A larger instruction window enables the execution of more instructions, while an L2 miss is processed and better exploits the available ILP. The Kilo-instruction microarchitecture [Cristal et al. 2004] has a static checkpoint allocation policy that attempts to balance the depth of the required checkpoint buffer, the re-execution penalty, and the store queue utilization. It allocates a checkpoint at the following points in the program: At the first branch after 64 instructions, at the 64th store instruction and when the number of instructions after the last checkpoint exceeds 512. The *MinDist* policy, which is used in this article for the purpose of comparison and evaluation of the proposed adaptive policy, is conceptually close to the Kilo-instruction microarchitecture. The CPR microarchitecture [Akkary et al. 2004] uses a dedicated branch prediction confidence estimator. CPR allocates a checkpoint at the following points in the program: at a hard-to-predict branch (a branch predicted

with a low-confidence estimation level), at the first branch after a rollback, and when the number of instructions after the last checkpoint exceeds 256. The same checkpoint allocation policy approach was used in Srinivasan et al. [2004] and Gandhi et al. [2005]. The baseline microarchitecture used in this article (LowConfBr), also uses a confidence estimator for allocating checkpoints.

Changing the supply voltage of registers in order to reduce the leakage power consumption has been an active research topic in recent years. Ayala et al. [2003] place all registers in a drowsy state by default, awaking registers as by demand prior to the read or write cycle. Their baseline model is an in-order processor using an architectural rather than unified register file. Khasawneh and Ghose [2005] manage the supply voltage, at a coarse-grained level (register zones). One of their proposed schemes also works by waking up drowsy registers on demand. A unified register file is used, thus an extra cycle delay between the scheduler select and register read is required in order to switch to the regular supply voltage. The authors assume that cycle is hidden because they assume the scheduler selects the instruction more than a cycle before it reads its operands.

A different approach is taken by Shieh and Hsu [2006], who use register access frequency information, obtained by the compiler, to assign infrequently used registers to infrequently used register banks, which are placed in drowsy state. The downside is that high-traffic registers are mapped into the same banks, which may lead to port collisions and performance slowdown. The work closest to the leakage power reduction scheme presented in this article is the work of Goto and Sato [2004]. They switch registers to the drowsy state after they are committed and until they are allocated, provided that they are no longer part of the mapping table. These are the registers that are considered free in a ROB-based microarchitecture with no early register release mechanisms. Our scheme is different in several ways: (i) it has greater potential for reduction (owing to early release of registers); (ii) it has minimal overhead logic for managing the state per register; (iii) it reduces the leakage between the bit lines and the ground line, which in a multiported structure such as the register file often accounts for a large proportion of the leakage power; and (iv) it also reduces the power consumed by checkpointed registers.

8. CONCLUDING REMARKS

Beginning with a highly efficient ROB-free microarchitecture, we have investigated checkpoint buffer organization trade-offs, checkpoint allocation and release policies, and ways to exploit specific features of checkpoint-based microarchitectures to reduce the leakage power of the register file. The motivation for the first part of this research was provided by results showing that an oracle checkpoint allocation policy performs significantly better than the two previously proposed policies, which are based on setting a minimal distance between consecutive checkpoints (MinDist) and allocating them at hard-to-predict branches (LowConfBr). For the integer benchmarks, the oracle policy performs 20.7% and 9.8% better than the MinDist and LowConfBr policies, respectively. A smaller performance gap also exists for the floating-point benchmarks.

We have presented a novel adaptive checkpoint allocation policy, which achieves most of the speed-up potential shown by the oracle policy. The adaptive policy changes according to dynamic events, such as second-level cache misses and rollback history. The adaptive policy is based on the observation that checkpoint allocation policies should only maintain large distances between consecutive checkpoints, either when a very large instruction window is beneficial or when the registers are overutilized and are not reclaimed efficiently due to excessive use of checkpoints. The adaptive checkpoint allocation policy achieves 6.8% and 2.2% speed-up for integer and floating point benchmarks, respectively. The adaptive policy does not require a branch confidence estimator and maintains its benefit for various checkpoint buffer depths and memory-access latencies.

Checkpoints provide an efficient infrastructure for restoring the state of a predictor. Efficiency derives from having a limited set of known points in the program, which the processor can rollback to, regardless of the size of the instruction window. Speculatively updated predictors become contaminated by wrong-path instructions. We have evaluated whether the state of the return address stack (RAS), branch predictor, and confidence estimator should be restored or reset during rollback. Results vary depending on the predictor type. Restoring the index and values of the RAS always boosts performance (4.6% speedup for the integer benchmarks), while restoring the global history of the branch predictor leads to mixed results. As for the confidence estimator, which is used only in the LowConfBr policy, we have found that resetting the global history shift register is better than either restoring it or leaving it as is.

Finally, we have shown that checkpoint-based microarchitectures present opportunities for reducing the leakage power of the register file. A substantial amount of leakage power is saved by releasing registers early and turning them off. Additional power is saved by placing checkpointed registers in drowsy mode. These two techniques add up to save about one half of the leakage power of the register file.

ACKNOWLEDGMENTS

Thanks to the anonymous reviewers for helpful feedback on this article and to Omer Heymann and Noam Jungmann for fertile discussions on leakage dissipation.

REFERENCES

- AKKARY, H., RAJWAR, R., AND SRINIVASAN, S. 2004. An analysis of a resource efficient checkpoint architecture. *ACM Trans. Archit. Code Optim.* 1, 4, 418–444.
- AKKARY, H., SRINIVASAN, S., KOLTUR, R., PATIL, Y., AND REFAAI, W. 2004. Perceptron-based branch confidence estimation. In *Proceedings of the 10th IEEE International Symposium on High-Performance Computer Architecture*. IEEE, Los Alamitos, CA, 265.
- AKL, P. AND MOSHOVOS, A. 2006. Branchtap: Improving performance with very few checkpoints through adaptive speculation control. In *Proceedings of the 20th Annual International Conference on Super-Computing*. ACM, New York, 36–45.
- AYALA, J., LOPEZ-VALLEJO, M., VEIDENBAUM, A., AND LOPEZ, C. 2003. Energy aware register file implementation through instruction predecode. In *Proceedings of the International Conference on Application-Specific Systems, Architectures, and Processors*. IEEE, Los Alamitos, CA, 86–96.

- BURGER, D. AND AUSTIN, T. 1997. The simple scalar tool set. *SIGARCH Comput. Archit. News* 25, 3, 13–25.
- CEZE, L., STRAUSS, K., TUCK, J., TORRELLAS, J., AND RENAULT, J. 2006. Cava: Using checkpoint-assisted value prediction to hide L2 misses. *ACM Trans. Archit. Code Optim.* 3, 2, 182–208.
- CRISTAL, A., SANTANA, O., VALERO, M., AND MARTINEZ, J. 2004. Toward kilo-instruction processors. *ACM Trans. Archit. Code Optim.* 1, 4, 389–417.
- ERGIN, O., BALKAN, D., GHOSE, K., AND PONOMAREV, D. 2004. Register packing: Exploiting narrow-width operands for reducing register file pressure. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, New York, 304–315.
- ERGIN, O., BALKAN, D., PONOMAREV, D., AND GHOSE, K. 2006. Early register deallocation mechanisms using checkpointed register files. *IEEE Trans. Comput.* 55, 9, 1153–1166.
- GANDHI, A., AKKARY, H., RAJWAR, R., SRINIVASAN, S., AND LAI, K. 2005. Scalable load and store processing in latency tolerant processors. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*. ACM, New York, 446–457.
- GOLANDER, A. AND WEISS, S. 2007. Re-execution and selective reuse in checkpoint processors. *HiPEAC J.* 2, 3, 129–152.
- GOLANDER, A. AND WEISS, S. 2008. Hiding the misprediction penalty of a resource-efficient high-performance processor. *ACM Trans. Archit. Code Optim.* 4, 4.
- GOTO, M. AND SATO, T. 2004. Leakage energy reduction in register renaming. In *Proceedings of the 24th International Conference on Distributed Computing Systems Workshops*. IEEE, Los Alamitos, CA, 890–895.
- 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*. IEEE, Los Alamitos, CA, 122–131.
- HWU, W. AND PATT, Y. 1987. Checkpoint repair for high-performance out-of-order execution machines. *IEEE Trans. Comput.* 36, 12, 1496–1514.
- JACOBSEN, E., ROTENBERG, E., AND SMITH, J. 1996. Assigning confidence to conditional branch predictions. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*. ACM, New York, 142–152.
- KALLA, R., SINHARAY, B., AND TENDLER, J. 2004. IBM POWER5 chip: A dual-core multi-threaded processor. *IEEE Micro* 24, 2, 40–47.
- KESSLER, R. 1999. The Alpha 21264 microprocessor. *IEEE Micro* 19, 2, 24–36.
- KHASAWNEH, S. AND GHOSE, K. 2005. An adaptive technique for reducing leakage and dynamic power in register files and reorder buffers. In *Proceedings of the 15th International Workshop on Integrated Circuit and System Design, Power and Timing Modeling, Optimization and Simulation*. Springer, Berlin, 498–507.
- KIM, N., FLAUTNER, K., BLAAUW, D., AND MUDGE, T. 2004. Circuit and microarchitectural techniques for reducing cache leakage power. *IEEE Trans. VLSI Syst.* 12, 2, 167–184.
- KIRMAN, M., KIRMAN, N., AND MARTINEZ, J. 2005. Cherry-mp: Correctly integrating checkpointed early resource recycling in chip multiprocessors. In *Proceedings of the 38th Annual International Symposium on Microarchitecture*. ACM, New York, 245–256.
- KIRMAN, N., KIRMAN, M., CHAUDHURI, M., AND MARTINEZ, J. 2005. Checkpointed early load retirement. In *Proceedings of the 11th IEEE International Symposium on High-Performance Computer Architecture*. IEEE, Los Alamitos, CA, 16–27.
- LIPASTI, M. AND SHEN, J. 1996. Exceeding the dataflow limit via value prediction. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*. ACM, New York, 226–237.
- MALINIAK, D. 2007. Stanch the bleeding of leakage power at 65nm. Tech. rep. <http://electronicdesign.com/Articles/ArticleID/17402/17402.html>
- 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*. IEEE, Los Alamitos, CA, 132–141.
- MARTINEZ, J., RENAULT, J., HUANG, M., PRVULOVIC, M., AND TORRELLAS, J. 2002. Cherry: Checkpointed early resource recycling in out-of-order microprocessors. In *Proceedings of the 35th Annual International Symposium on Microarchitecture*. ACM, New York, 3–14.
- MCGHAN, H. 2006. Power: The sixth generation. *Microprocessor Rep.* 10.

- MEANEY, P., SWANEY, S., SANDA, P., AND PAINHOWER, L. 2005. IBM z990 soft error detection and recovery. *IEEE Trans. Device Mater. Reliab.* 5, 3, 419–427.
- MOSHOVOS, A. 2003. Checkpointing alternatives for high-performance, power-aware processors. In *Proceedings of the International Symposium on Low-Power Electronics and Design*. ACM, New York, 318–321.
- MOUDGILL, M., PINGALI, K., AND VASSILIADIS, S. 1993. Register renaming and dynamic speculation: An alternative approach. In *Proceedings of the 26th Annual International Symposium on Microarchitecture*. ACM, New York, 202–213.
- MUTLU, O., STARK, J., WILKERSON, C., AND PATT, Y. 2003. Run ahead execution: An alternative to very large instruction windows for out-of-order processors. In *Proceedings of the 9th IEEE International Symposium on High-Performance Computer Architecture*. IEEE, Los Alamitos, CA, 129–140.
- POWELL, M., YANG, S., FALSAFI, B., ROY, K., AND VIJAYKUMAR, T. 2000. Gated-vdd: A circuit technique to reduce leakage in cache memories. In *Proceedings of the International Symposium on Low-Power Electronics and Design*. ACM, New York, 90–95.
- RACUNAS, P., CONSTANTINIDES, K., MANNE, S., AND MUKHERJEE, S. 2007. Perturbation-based fault screening. In *Proceedings of the 13th International IEEE Symposium on High-Performance Computer Architecture*. IEEE, Los Alamitos, CA, 169–180.
- RAMIREZ, T., PAJUELO, A., SANTANA, O., AND VALERO, M. 2006. Kilo-instruction processors, run ahead and prefetching. In *Proceedings of the 3rd Conference on Computing Frontiers*. ACM, New York, 269–278.
- SANGIREDDY, R. 2006. Reducing rename logic complexity for high-speed and low-power front-end architectures. *IEEE Trans. Comput.* 55, 6, 672–685.
- SEZNEC, A. AND MICHAUD, P. 2006. A case for (partially) TAGged GEometric history length branch prediction. *J. Instruc.-Level Paral.* 8.
- SHIEH, W. AND HSU, S. 2006. Power-aware register assignment for multi-banked register files. In *Proceedings of the International Computer Symposium*. ACM, New York, 99–104.
- SKADRON, K., AHUJA, P., MARTONOSI, M., AND CLARK, D. 1998. Improving prediction for procedure returns with return-address-stack repair mechanisms. In *Proceedings of the 31st Annual International Symposium on Microarchitecture*. ACM, New York, 259–271.
- SKADRON, K., MARTONOSI, M., AND CLARK, D. 2000. Speculative updates of local and global branch history: A quantitative analysis. *J. Instruc.-Level Paral.* 2.
- SMITH, J. AND PLESZKUN, A. 1988. Implementing precise interrupts in pipelined processors. *IEEE Trans. Comput.* 37, 5, 562–573.
- SRINIVASAN, S., RAJWAR, R., AKKARY, H., GANDHI, A., AND UPTON, M. 2004. Continual flow pipelines. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York, 107–119.
- TARJAN, D., THOZIYOOR, S., AND JOUPPI, N. 2006. Cacti 4.0. Tech. rep. HPL-2006-86, HP Laboratories Palo Alto.
- WILKES, M. 2001. The memory gap and the future of high-performance memories. *SIGARCH Comput. Archit. News* 29, 1, 2–7.
- YEAGER, K. 1996. The MIPS R10000 super-scalar microprocessor. *IEEE Micro*. 16, 2, 28–40.

Received September 2007; revised June 2008; accepted March 2009