Checkpointing Alternatives for High Performance, Power-Aware Processors

Andreas Moshovos
Electrical and Computer Engineering
University of Toronto
moshovos@eecg.toronto.edu

ABSTRACT

High performance processors use checkpointing to rapidly recover from branch mispredictions and possibly other exceptions. We demonstrate that conventional checkpointing becomes unattractive in terms of resource and power requirements for future generation processors. We propose out-of-order checkpoint release and checkpoint prediction, two alternatives that require significantly less resources and power while maintaining high-performance. We demonstrate their utility at the register alias table (RAT). Our methods reduce the number of RAT checkpoints to 1/3 (from 48 down to 16) for an aggressive, 8-way superscalar processor with a 256-entry instruction window. Using a 0.18um process model we estimate that RAT power is reduced by 24%.

Categories and Subject Descriptors

C.1 Processor Architectures, B.8 Performance and Reliability.

General Terms

Algorithms, Performance, Design, Experimentation.

Keywords

Checkpointing, renaming, out-of-order execution, power-aware, power density.

1. INTRODUCTION

Power-aware work for high performance processors to date has focused on significant, overall on-chip power reduction methods that maintain competitive performance. Naturally, existing work was thus targeted on those units that dissipate a large fraction of the overall on-chip power such as the level-one caches and the scheduler. Reducing the power of other units that may *not* dissipate a large fraction of overall power dissipation is also becoming necessary. These are units that exhibit extremely high *power density*, i.e., units whose overall power dissipation is disproportionately large compared to their area. Because silicon is not a good heat conductor these units create *hot spots* where the temperature tends to significantly exceed that of other on-chip areas. A high temperature can lead to transient or permanent faults.

In modern microprocessors the *register rename unit* exhibits both high power density and dissipates a considerable fraction of on-chip power [5]. For example, the Pentium Pro rename unit dissipates 4% of the overall on-chip power but its power-density is the fourth highest on-chip, just short of the power density of the integer execution, the floating-point execution and the decode units [5]. The current architectural trends towards deeper and wider instruction windows will further increase the power density of the renaming unit.

In this paper, we focus on reducing the power density at the rename unit by reducing the overall power dissipated by it while maintaining high performance. We propose methods that require fewer control-flow related checkpoints. As we show, using conventional checkpointing becomes impractical since the checkpoint store and its power grows very large for wider and deeper processors. We propose two orthogonal checkpoint reduction techniques: (1) *checkpoint prediction*, and (2) *out-of-order checkpoint release*. Both methods require virtually no additional resources as they utilize information that is already readily available.

We study the checkpoint requirements of integer and floating-point applications from SPECcpu2000 and of a multimedia application. We set a goal of reducing power while maintaining performance within 1% (worst case) of a conventional rename design. We demonstrate that our methods can reduce the number of checkpoints from 48 down to 16 (or to eight for a worst case slowdown of 2.1%). We further demonstrate that our methods can reduce maximum power by 24% (including regular rename activity in addition to checkpointing). To the best of our knowledge no previous work exists on checkpoint prediction and out-of-order checkpoint release.

The rest of the paper is organized as follows: In section 2 we introduce our methods. In section 3 we study their performance and power impact. We comment on related work in section 4 and summarize our findings in section 5.

2. RENAME CHECKPOINTING

Register renaming (or simply *renaming*) removes name dependences (WAR or WAW) with the goal of exposing additional instruction-level parallelism. This is done by mapping architectural registers to physical registers during execution. Figure 1(a) shows an example of renaming a code sequence assuming a MIPS-like ISA with three architectural registers (see, for example, [9] for additional information).

The core of the renaming activity takes place in the *register alias table* (RAT) which holds the current mapping of architectural to physical registers. Conceptually, the RAT is a table that is indexed by architectural register names and returns physical register names. Here we assume an SRAM-based RAT, a multiported register-file like structure (for example, for an ISA with 32 architectural registers and an implementation with 256 physical registers the RAT contains 32 8-bit wide entries). For an N-way processor and a MIPS-I ISA, the RAT has 3xN read ports (2xN for reading the current

ISLPED'03, August 25–27, 2003, Seoul, Korea.

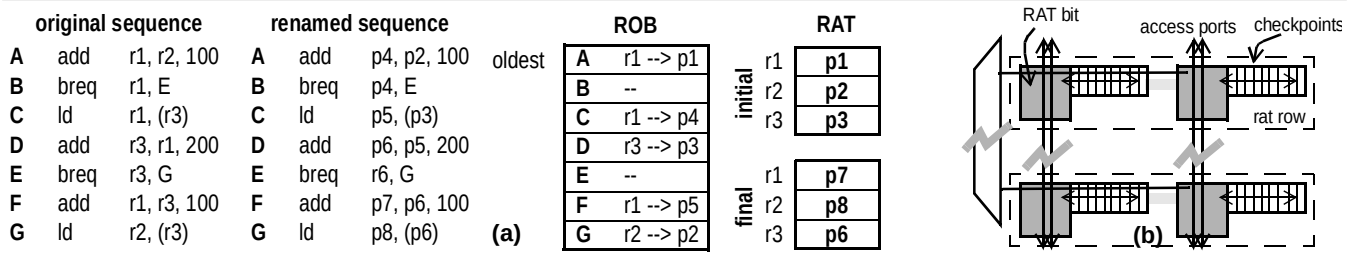

Figure 1: (a) Renaming example showing how the RAT and the ROB are used. Starting from the initial RAT state shown, we illustrate how a MIPS-like code sequence (all branches predicted not taken) is renamed. Shown are the final ROB and RAT state and the renamed code sequence. Names "r" and "p" refer to architectural and physical registers respectively.We assume an ISA with just three architectural registers. (b) RAT Checkpointing.

mappings for the source registers and N for doing the same for the destination register) and N write ports.

The Need for Checkpointing: Since renaming occurs prior to execution a renamed instruction may later be squashed. In which case we need to restore the RAT to the values it had immediately before the specific instruction was renamed. For example, in figure 1, load C may cause a page-fault in which case the RAT has to be restored to (p4, p2, p3). This is the functionality provided by *checkpointing*.

Checkpointing is implemented via the *re-order buffer (ROB)*, a circular queue which works as follows. Each renamed instruction allocates a ROB entry where it records the *previous* mappings for all its destination registers. For example, in figure 1, instruction C records that prior to renaming register r1 to p5, r1 was mapped to p4. That is, the ROB is a complete, ordered record of all mapping changes done by active instructions. Starting from the most recent ROB entry, we can reverse all register renames up to the oldest squashed instruction one-by-one. For example, in figure 1 if C causes a page-fault, we would start from the entry for G going backward reversing all RAT changes for G down to C.

While the ROB can restore the RAT to *any* instruction (it is a *fine-grain* checkpointing mechanism) it may require several cycles to do so if multiple instructions are squashed. Typically, the ROB is capable of restoring the renames of as many instructions as the processor can rename per cycle (e.g., eight instructions per cycle for an 8-way processor). Consequently, the penalty for restoring the RAT is proportional to the number of instructions that are squashed.

2.1 RAT Checkpointing

Virtually all modern high-performance processors use *control-flow speculation* (also known as *branch speculation*) to improve performance. When speculation is incorrect a *mispeculation* occurs and all instructions down the mispeculated path are squashed. Because such mispeculations are relatively frequent processors incorporate a second, *coarse-grain* checkpointing mechanism where the RAT is capable of checkpointing and restoring *all* register mappings *instantaneously*. This facility is implemented as a set of circular buffers that are physically embedded into the RAT next to each cell (figure 1(b)). When a branch is renamed, a complete copy of all RAT bits is made into the next available entry of this buffer. The depth of the buffer limits the number of checkpoints and hence it sets an artificial limit on the number of branches that can be simultaneously unresolved. For example, MIPS R10K had four such checkpoints [1]. As we show in section 3.1, many more are required in the future.

2.2 Intelligent Checkpointing

RAT checkpointing is a performance optimization. The ROB provides a fail-safe yet potentially slow recovery mechanism. A lower power alternative would be to avoid RAT-checkpointing altogether but, as we show in section 3.1, the performance penalty is high. We propose several alternatives that offer most of the performance advantages of RAT checkpointing with much lower resources and power.

For simplicity, conventional checkpointing mechanisms allocate checkpoints for *all* predicted branches. Moreover, they use *in-order checkpoint release* where a checkpoint is held until the corresponding and all preceding branches are *resolved* (i.e., their direction is calculated). This way the checkpoint store can be implemented and managed as a circular queue and the control logic is very simple (e.g., just a pointer). There are two considerations: First, when the number of checkpoints becomes large (as it would be the case for larger window processors) it may take a prohibitively large number of cycles to restore the RAT from the circular queue. This is because we have to shift in the appropriate checkpoint (it may be faster to restore from the ROB). Second, sufficient checkpoints must be provided to maintain high performance. As we move towards deeper and wider instruction windows this number of checkpoints increases. As we show in section 3.1, for a 256-entry window, we may need even up to 48 checkpoints to sustain acceptable performance for all the programs we studied. Creating RAT bit cells containing each a 48-entry[1] circular buffer is questionable in power and performance terms. To reduce the checkpoints we propose using *checkpoint prediction* and *out-of-order checkpoint release.*

Out-of-Order Checkpoint Release: In this scheme, a checkpoint is released as soon as the corresponding branch is resolved, hence checkpoints may be released *out-of-order*. This scheme uses the checkpoint store more efficiently. To understand why this scheme is correct let us consider the two possible scenarios in figure 1(a) where branch E is allowed to resolve and discard its checkpoint before branch B. In the first, branch B resolves correctly. In the second, branch B causes a mispeculation and squashes all subsequent instructions using its own checkpoint to restore the RAT. In both cases E's checkpoint was not needed. This argument generalizes for any number of branches. There are some implementation implications however. Specifically, the checkpointing store can no longer be managed as a circular queue. Instead, a

[1] This figure is in par with current designs. For example, the Alpha 21264 uses a 20-entry checkpoint store while it has a window of about 100 instructions.

Table 1: Base processor configuration.

Branch Predictor	16k GShare+16K bi-modal 16K selector 2 branches per cycle	**Stage Latencies**	8 cycles from branch predict to decode 6 cycles for decode and renaming 6 cycles from writeback to commit 10 cycles branch misprediction penalty + overhead to restore from ROB if applicable.	**Fetch Unit**	Up to 8 instr. per cycle 64-entry Fetch Buffer 2 branches per cycle Non-blocking I-Cache
DL1/IL1 Geometry	64Kbyte/32byte blocks/4-way SA			**Load/Store Queue**	64 entries, 4 inst/cycle Perfect disambiguation
Issue/Decode/Commit	any 8 instructions / cycle	**FU Latencies**	same as MIPS R10000	**Main Memory**	Infinite, 100 cycles
Instr. Window	256 entries	**UL2 Geometry**	1Mbyte/64byte blocks/8-way SA	**L1/UL2 Latencies**	3/16 cycles

register-file like scheme is needed. Still it is possible to embed the checkpointing store near each RAT bit using vertical read and write control lines per checkpoint cell along with appropriate control logic.

Checkpoint-Prediction: We also propose allocating checkpoints *selectively* via *checkpoint prediction.* When a mispeculation occurs, recovery proceeds as follows: if the mispeculated branch has a checkpoint, recovery is immediate. Otherwise, we use a two-step process. First, we use the nearest, subsequent in program order checkpoint to restore the RAT. Then using the ROB we reverse all intervening renames (this may require multiple cycles). For example, in figure 1(a), if we allocated a checkpoint for branch E and a mispeculation occurs at branch B, we would use the checkpoint for E and then the ROB to restore the RAT.

We propose two checkpoint prediction methods:

Anyweak: We propose allocating checkpoints only for *weak* (hard to predict) branches since mispeculations are more likely on them. The *anyweak* method uses information that is readily available in a combined predictor to identify weak branches. Specifically, it allocates a checkpoint if *any* of the two sub-predictors reports a weakly-biased value. Anyweak is compatible with both in-order and out-of-order checkpoint release. For simplicity, we allocate checkpoints for all indirect branches. Since we use readily available information, anyweak has virtually no power or resource overheads.

Lazy Anyweak: As we show in section 3.1, performance with anyweak alone levels off after a few checkpoints are provided. To take advantage of additional checkpoints we propose the *lazy anyweak* method which works as follows. Checkpoints are allocated for all branches with conventional checkpointing, however, a *weak* branch is allowed to *steal* (i.e., overwrite) the checkpoint of a preceding *non-weak* (or *strong)* branch. The front-end pipeline stalls only on *weak* branch if all checkpoints are currently held by preceding weak branches. Since the physical order of checkpoints does not match that of the program, this method requires out-of-order checkpoint release.

The performance tradeoffs with checkpoint prediction are complex. With conventional checkpointing, recovering the RAT takes a single cycle while with ROB-only checkpointing a number of cycles that is proportional to the number of instructions squashed is required. With checkpoint prediction, when a mispeculation occurs on a branch with a checkpoint recovery is as fast as with conventional checkpointing. Recovering from a branch with no checkpoint. however, may be faster compared to ROB-only checkpointing. This is because we may first restore the most recent, subsequent checkpoint if any exists, and then use the ROB to restore only the remaining instructions.

3. EVALUATION

3.1 Performance

We used Simplescalar v3.0 [6] to simulate the aggressive, deeply pipelined superscalar processor detailed in table 1. We use *gzip* (gzp), *mcf*, *twolf* (twf), *swim* (swm), *equake* (eqk) (mpe), *gcc*, *parser* (prs), *mesa* (mes), *bzip2* (bzp) which are integer and floating-point programs from SPECcpu2000 and *mpeg2encode* from mediabench. We simulated up to 1 billion instructions per benchmark after skipping the initialization. The binaries were compiled for the MIPS-like PISA architecture using GNU's gcc v2.9.

3.1.1 Conventional Checkpointing

Figure 2 compares the performance of ROB-only checkpointing with that of conventional in-order checkpointing (i.e., allocating checkpoints for all branches). In all performance graphs we report slowdowns compared to a processor with an infinite number of RAT checkpoints. For the conventional checkpointing we vary the number of checkpoints from 48 down to 4 in the steps shown. The ROB can restore eight instructions per cycle. ROB-only checkpointing offers competitive performance only for swim which exhibits a low branch frequency and high prediction accuracy. For the other applications, slowdowns with ROB-only checkpointing vary from approximately 10% (bzip2) to as much as 35% (gzip). Conventional checkpointing is worse than ROB-only checkpointing when there are few checkpoints. Most programs require 32 or more checkpoints to achieve performance that is within 1% of the base. Even with 48 checkpoints, mcf exhibits a slowdown of 1.7%.

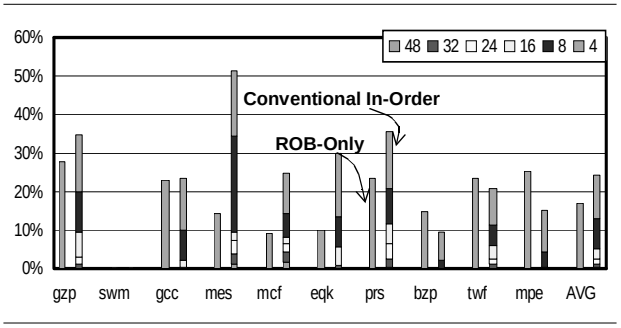

Figure 2: ROB-only checkpointing vs. conventional RAT in-order release checkpointing for various numbers of checkpoints.

3.1.2 Checkpointing Alternatives

Figure 3(a) reports performance slowdowns for the following combinations: *conventional* with *in-order release, anyweak* with *in-order release, conventional* with *out-of-order release, anyweak* with *out-of-order release.* Out-of-order release (third bar from left) offers significantly better performance compared to in-order release (first bar). Anyweak offers superior performance with both in-order (second bar) and out-of-order (fourth bar) checkpoint release when there are very few checkpoints (i.e., four or eight). It fails to offer additional improvements when more checkpoints are introduced. In some cases, anyweak with out-of-order release performs worse than with in-order release. This is possible when a checkpoint for a resolved branch is used to reduce the number of ROB entries

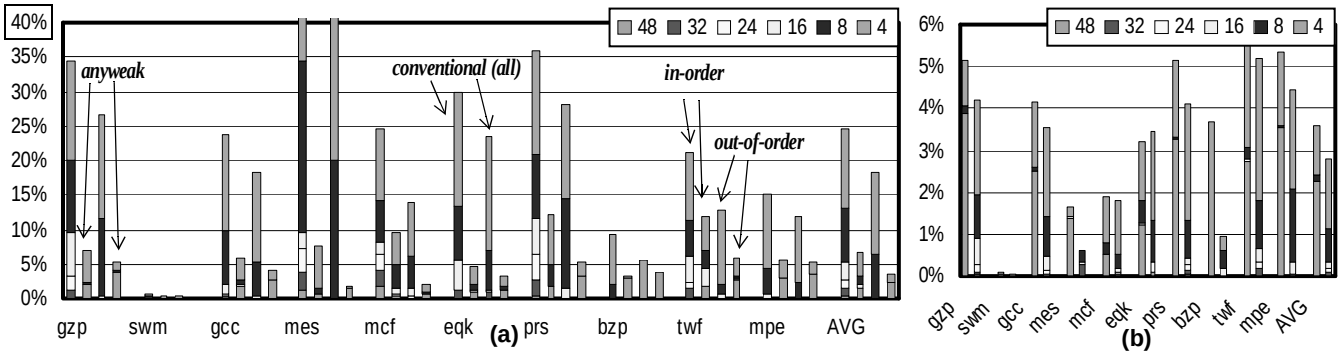

Figure 3: (a) Slowdowns with in-order (left) and out-of-order control-flow resolution policies as a function of the number of available checkpoints. (b) Slowdowns with anyweak (left bar) vs. anyweak lazy (right bar) under out-of-order checkpoint release. Lower is better.

that are restored on a mispeculation of a preceding branch. In figure 3(b) we compare *anyweak* (left bar) and *lazy anyweak* (right bar) prediction under out-of-order release. Anyweak lazy is superior in all cases and can also take advantage of additional checkpoints. It achieves slowdowns of 2.1% or less for all programs with just eight checkpoints. The slowdowns drop below 1% with just 16 checkpoints.

3.2 Power Reduction

We adopted the widely-used WATTCH [4] power models to estimate the maximum power dissipated by the RAT. We started from the simple *cache* model since modern register files use partial bitline discharging to reduce power. We modified this model to account for embedded checkpointing. We used WATTCH's 0.18um process model with a 1.8v power supply and a 1Ghz clock. We did not attempt to scale this process model to smaller feature sizes since its scalability has not been demonstrated. We assumed a single checkpoint read or write per cycle and that all 24 and 16 read and write ports of the RAT are used per cycle. These are pessimistic assumptions that may downplay the savings from our technique primarily because programs rarely need all the read and write RAT ports. A 24 read port and eight write port RAT for 32 architectural registers, 256 physical registers and 48 checkpoints (this is our base configuration based on table 1 and the results of section 3.1.1) dissipates approximately 996mW. With 16 checkpoints the same RAT dissipates 754mW or a 24% reduction in power (compared to a 32 checkpoint RAT the reduction is 15%).

4. RELATED WORK

There are two primary methods for implementing renaming based on an SRAM array (e.g. [1]) or based on a CAM array (e.g., [3]). Our methods are also applicable with modification to the CAM implementation. Parlacharla *et al*. developed delay models for the RAT SRAM array [2] and Brooks *et al*., extended these models to include maximum power dissipation [4]. Our checkpoint predictors rely on what can be thought of as branch confidence methods. Several such methods have been proposed for other purposes where different tradeoffs apply. Grunwald *et al*. proposed, among others, the *both strong* confidence estimator for pipeline gating purposes [8]. Our anyweak predictor can be thought as its reverse. An alternative would be to use explicit confidence predictors (e.g., [7]).

5. CONCLUSION

As a step towards power-density optimizations we focused on the register renaming unit and proposed checkpoint reduction via *selective checkpoint allocation* and via *out-of-order checkpoint release*. We proposed two *checkpoint prediction methods* and demonstrated that it is possible to drastically reduce the checkpoints necessary. We considered the obvious alternative of not using checkpointing at all. We modeled the power and performance of our methods and found that it is possible to achieve performance that is on the average within 0.4% of conventional RAT checkpointing by reducing the number of checkpoints from 48 to just 16 (or to eight for a worst case slowdown of 2.1%). Our methods can reduce overall rename power by 24%.

ACKNOWLEDGEMENTS: I'd like to thank the anonymous referees for their insightful comments. This work was supported by the National Sciences and Engineering Research Council of Canada, the Semiconductor Research Corporation and by the University of Toronto. This research was performed while the author was serving a full, mandatory term of service with the Hellenic Armed Forces which are in no way affiliated with this work. I wish to thank Doug Carmean, Herbert Hum, Stephan Jourdan, Avi Medleson, Farid Najm and Ronny Ronen for discussions about power density.

6. REFERENCES

[1] K. C. Yeager, The MIPS R10000 Superscalar Microprocessor, IEEE MICRO, April, 1996.

[2] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *Proc. of the 24th Int'l Symposium on Computer Architecture*, June 1997.

[3] R. P. Colwell and R. L. Steck, A 0.6um BiCMOS Processor with Dynamic Execution, In *Proc. International Solid State Circuits Conference*, Feb. 1995.

[4] D. Brooks, V. Tiwari M. Martonosi Wattch: A Framework for Architectural-Level Power Analysis and Optimizations, *Proc of the 27th Int'l Symposium on Computer Architecture*, 2000.

[5] P6 Power Data, Intel Corp.

[6] D. Burger and T. Austin, *The Simplescalar Simulation Environment*, Univ. of Wisconsin-Madison, Computer Sciences Dept. Technical Report.

[7] E. Jacobsen, E. Rotenberg, and J. E. Smith. *Assigning Confidence to Conditional Branch Predictions.* In Proc. Int'l Symposium on Microarchitecture, December 1996.

[8] D. Grunwald, A. Klusser, S. Manne and A. Plezkun, *Confidence Estimation for Speculation Control*, In Proc. of the 25th Int'l Symposium on Computer Architecture, June, 1998.

[9] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach, 2nd Edition, Morgan Kaufman Publishers.*