University of Pennsylvania
ScholarlyCommons

Departmental Papers (CIS) | Department of Computer & Information Science

1-1-2008

Physical Register Reference Counting

Amir Roth
University of Pennsylvania, amir@cis.upenn.edu

Copyright 2008 IEEE. Reprinted from *IEEE Computer Architecture Letters*, Volume 7, Issue 1, pages 9-12.

This material is posted here with permission of the IEEE. Such permission of the IEEE does not in any way imply IEEE endorsement of any of the University of Pennsylvania's products or services. Internal or personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution must be obtained from the IEEE by writing to pubs-permissions@ieee.org. By choosing to view this document, you agree to all provisions of the copyright laws protecting it.

This paper is posted at ScholarlyCommons. http://repository.upenn.edu/cis_papers/380
For more information, please contact repository@pobox.upenn.edu.

Physical Register Reference Counting

Abstract
Several recently proposed techniques including CPR (Checkpoint Processing and Recovery) and NoSQ (No Store Queue) rely on reference counting to manage physical registers. However, the register reference counting mechanism itself has received surprisingly little attention. This paper fills this gap by describing potential register reference counting schemes for NoSQ, CPR, and a hypothetical NoSQ/CPR hybrid. Although previously described in terms of binary counters, we find that reference counts are actually more naturally represented as matrices. Binary representations can be used as an optimization in specific situations.

Comments
Copyright 2008 IEEE. Reprinted from *IEEE Computer Architecture Letters*, Volume 7, Issue 1, pages 9-12.

This material is posted here with permission of the IEEE. Such permission of the IEEE does not in any way imply IEEE endorsement of any of the University of Pennsylvania's products or services. Internal or personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution must be obtained from the IEEE by writing to pubs-permissions@ieee.org. By choosing to view this document, you agree to all provisions of the copyright laws protecting it.

This journal article is available at ScholarlyCommons: http://repository.upenn.edu/cis_papers/380

Physical Register Reference Counting

Amir Roth
Department of Computer and Information Science, University of Pennsylvania

***Abstract*—Several recently proposed techniques including CPR (Checkpoint Processing and Recovery) and NoSQ (No Store Queue) rely on reference counting to manage physical registers. However, the register reference counting mechanism itself has received surprisingly little attention. This paper fills this gap by describing potential register reference counting schemes for NoSQ, CPR, and a hypothetical NoSQ/CPR hybrid. Although previously described in terms of binary counters, we find that reference counts are actually more naturally represented as matrices. Binary representations can be used as an optimization in specific situations.**

I. Overview

Dynamically-scheduled processors with unified physical register files use a simple algorithm to free physical registers: registers are unconditionally added to a free list at instruction commit and squash. Several proposed techniques require a more flexible physical register reclamation mechanism: *physical register reference counting*. Here, instruction processing stages do not unconditionally free registers. Instead, they increment and decrement register reference counts. Registers become free when their reference count reaches zero. These techniques fall into two general classes: ones that use reference counting to track register reads [1, 2, 6, 7, 10] and ones that use it to track register "writes" [3, 8, 9]. Here, we use CPR (Checkpoint Processing and Recovery) [1] and NoSQ (No Store Queue) [9] to represent these two classes, respectively.

Despite acting as an enabling mechanism for a number of techniques, register reference counting itself has received little attention. The earliest description of a reference counting scheme we found uses per-physical register multi-input up-down binary counters [6]. Subsequent proposals either point to this design, describe their own scheme as "a table of reference counts", or skirt the issue altogether. Insufficient detail about the requirements and implementation of register reference counting has led to confusion and to concern about the feasibility of proposals that use it. This paper attempts to de-mystify register reference counting by providing a set of design recipes for reference counting mechansims. As examples, we describe reference counting mechanism designs for NoSQ, CPR, and a hypothetical NoSQ/CPR hybrid.

II. Register Reference Counting Applications

Register management is related to, but separate from, register renaming. Register renaming is the mapping of source operands and uses a map table. Register management is the allocation and freeing of registers and uses a free list. Reference counting is a register management scheme. This section uses CPR and NoSQ to motivate register reference counting and to demonstrate the two basic ways in which it is used.

Conventional processor. Figure 1 shows register management for a conventional processor with three logical registers (r1–r3) and eight physical registers (p1–p8). For each of five dynamic instructions (A–E), the figure shows the state of the register map table before renaming (RMap), the renamed instruction itself, and the instruction's register management actions. At rename, an instruction allocates a new physical register for its destination. It also remembers the physical register it over-writes in the map table—it frees this register at commit. For example, instruction A allocates p4 for its destination r1. p4 is over-written by instruction D and is freed when D commits. Figure 1 refers to these actions as "ROB" actions because they parallel reorder buffer (ROB) management actions.

	raw insns	RMap r1	RMap r2	RMap r3	renamed insns	ROB alloc ren	ROB free cmt
A	r1= r3+1	p1	p2	p3	**p4** =p3+1	**p4**	p1
B	m[r2] = r1	**p4**	p2	p3	m[p2] = p4		
C	r3 = m[r2]	p4	p2	p3	p5 = m[p2]	p5	p3
D	r1 = r1+1	**p4**	p2	p5	p6 = p4+1	p6	**p4**
E	r3 = r1+r3	p6	p2	p5	p7 = p6+p5	p7	p6
		p6	p2	p7			

Figure 1. Register management for a conventional processor.

This simple algorithm is enabled by a one-to-one mapping between register-writing instructions and registers they manage. At rename, every register-writing instruction allocates exactly one register and every register is allocated by exactly one instruction. At commit, every register-writing instruction frees exactly one register and each register is freed by exactly one instruction.

NoSQ. NoSQ (No Store Queue) is a microarchitecture that does not have a conventional store queue, performing all would-be in-flight store-load communication using speculative memory bypassing (SMB) [5]. When NoSQ renames a load that it predicts will communicate with an older in-flight store, it does not dispatch that load to the execution engine. Instead, it maps the load's destination to the physical register that holds the store's data input. This action effectively connects the load's consumers directly to the store's data producer, collapsing the DEF-store-load-USE chain into a speculative DEF-USE chain. Figure 2a uses the same five instruction sequence to show NoSQ's register management scheme. Load C communicates with store B and so NoSQ maps its output r3 to B's data input p4 rather than to a new register. This action links E (C's consumer or USE) to A (B's producer or DEF), bypassing the store-load pair B-C.

NoSQ breaks the one-to-one instruction-to-register mapping that enables conventional register management. Not every renaming instruction allocates a new physical register—here C reuses p4, effectively "sharing" it with A. Similarly, not every committing instruction frees a register—here D doesn't free p4, p4 is only freed when both D and E commit. NoSQ's register management algorithm can be formulated in a simple way in terms of reference counting. Allocations and instances of sharing or reuse become increments and frees become decrements. These actions track references by in-flight instructions and parallel ROB management. This algorithm is not specific to NoSQ. It applies to any implementation of SMB on a processor that has both a unified physical register file and a ROB.

NoSQ-style physical register sharing is what we mean by register "write" tracking. When multiple instructions share a single register as their output—here A and C share p4—both logically write that register. However, because both would write the same value, only one—here A—executes and physically performs the write. We track logical writes and over-writes, not physical writes.

Manuscript submitted: 21-Aug-2007. Manuscript accepted: 24-Sep-2007. Final manuscript received: 28-Sep-2007.

	raw insns	RMap r1	RMap r2	RMap r3	renamed insns	ROB +ren	ROB –cmt
A	r1=r3+1	p1	p2	p3	**p4**=p1+1	**p4**	p1
B	m[r2]=r1	**p4**	p2	p3	m[p2]=p4		
C	r3=m[r2]	**p4**	p2	p3	**p4**=m[p2]	**p4**	p3
D	r1=r1+1	**p4**	p2	p4	p6=p4+1	p6	**p4**
E	r3=r1+r3	p6	p2	**p4**	p7=p6+p4	p7	**p4**
		p6	p2	p7			

(a) NoSQ

RMap r1	RMap r2	RMap r3	renamed insns	IQ +ren	IQ –exc	Ckpt +ren	Ckpt –cmt
p1	p2	p3	p4=p1+1	p1	p1	p1,2,3	p1,2,3
p4	p2	p3	m[p2]=**p4**	p2,**4**	p2,**4**		
p4	p2	p3	p5=m[p2]	p2	p2		
p4	p2	p5	p6=**p4**+1	**p4**	**p4**	**p4**,2,5	**p4**,2,5
p6	p2	p5	p7=p6+p5	p6,5	p6,5		
p6	p2	p7					

(b) CPR

RMap r1	RMap r2	RMap r3	renamed insns	IQ +ren	IQ –exc	LSQ +ren	LSQ –cmt	Ckpt +ren	Ckpt –cmt
p1	p2	p3	p4=p1+1	p1	p1			p1,2,3	p1,2,3
p4	p2	p3	m[p2]=**p4**			p2,**4**	p2,**4**		
p4	p2	p3	p4=m[p2]			p2,**4**	p2,**4**		
p4	p2	p4	p6=**p4**+1	**p4**	**p4**			**p4**,2,**4**	**p4**,2,**4**
p6	p2	p4	p7=p6+**p4**	p6,**4**	p6,**4**				
p6	p2	p7							

(c) NoSQ/CPR

Figure 2. Register management for three processors that use reference counting.

CPR. CPR uses aggressive register reclamation to reduce the active lifetime of individual registers and improve register file scalability. CPR frees a register after both the last instruction to read it executes and the instruction that over-writes it renames implying no future readers. Under conventional instruction-granularity speculation, squash recovery can undo map table over-writes and retroactively render a given register reclamation premature. CPR sidesteps this problem by supporting speculation and mis-speculation recovery only at the granularity of checkpoints. A register can be freed only if it doesn't appear in either the current map table or in any checkpoint. Figure 2b illustrates. The current map table—corresponding to state after instruction E is renamed—contains p6, p2, and p7. Checkpoints corresponding to instructions A and D map p1, p2 and p3 and p4, p2 and p5, respectively.

CPR also breaks one-to-one instruction-to-register mapping. Multiple reader instructions are involved in the freeing of a single register and a single instruction may be involved in the freeing of two registers. Checkpoint creation and freeing also participate in register management. CPR's register management algorithm can be formulated using reference counting. A register's reference count is incremented when a reading instruction renames and decremented when a reading instruction executes. Reference counts are also incremented when a map table checkpoint is created—the reference count of each register that appears in the checkpoint is incremented—and decremented when a checkpoint is freed. Figure 2b refers to these sets of actions as IQ and Ckpt actions, respectively, denoting correspondence to instructions entering and leaving the issue queue and to checkpoint creation and freeing.

NoSQ/CPR. NoSQ and CPR are a potentially attractive combination. CPR improves register file scalability while NoSQ improves load and store queue scalability. Merging NoSQ and CPR requires merging their reference counting schemes. The simplest way of doing that is to extend CPR's read-counting scheme to account for NoSQ. As part of eliminating the store queue, NoSQ moves store execution from the out-of-order core to the in-order back-end. NoSQ also (logically although not physically) re-executes all loads in the back-end for verification. From a reference counting standpoint, this requires moving the decrements associated with CPR loads and stores from execute to commit. Decrements associated with other instructions still take place at execute. Figure 2c shows a register management diagram for NoSQ/CPR. LSQ actions are reference count updates for loads and stores.

Squash recovery. Physical registers are freed not only at commit (for NoSQ) or execute (for CPR), but also during mis-speculation recovery. In a processor with conventional register management, recovery involves freeing the registers allocated to the destinations of squashed instructions. This can be done in one cycle for an arbitrarily large number of instructions starting at the tail of the window by exploiting the observation that squash recovery frees registers in reverse allocation order. The free list is organized as a circular queue and registers allocated to squashed instructions are reclaimed by moving the free list's head pointer.

In a processor with register reference counting, recovery involves undoing the increments that were performed by squashed instructions *if* the corresponding decrements had not already occurred. Accomplishing this in one cycle is not strictly necessary because instruction squashing and register freeing are not on the mis-speculation recovery critical path [1]. If possible, however, support for single-cycle squash recovery is desirable.

III. Basic Design: Unary Reference Counts

This section describes the basic structure and mechanics of unary reference counting using NoSQ as a detailed inline example. It then describes unary reference counting mechanisms for CPR and a NoSQ/CPR hybrid. Unary reference counting schemes for all three applications are shown in Figure 3.

Unary reference counts are two dimensional matrices and are naturally implemented as RAMs. Following the layout of Figure 3, the horizontal dimension corresponds to physical registers and so each column represents the reference count for that register. For register allocation purposes, exact reference counts don't matter; what matters is whether a given reference count is zero or not. To interface with register allocation, OR gates reduce each reference count—the bits in each column—to a single free/allocated bit. Collectively, these form a bitvector-style free list. Registers are allocated directly out of this bitvector using encoders. The reference counting mechanisms we describe are identical along their horizontal dimension and all present this interface to register allocation.

Reference counting schemes for different applications differ along the vertical dimension of the matrix. Here, rows correspond to entities that reference (i.e., point to) physical registers. In NoSQ, physical registers are referenced by (the destinations of) in-flight instructions and by architected logical registers. Figure 3a shows NoSQ's reference count matrix as containing one row for each ROB entry and one row for each logical register in the commit map table (CMap). Because matrix rows parallel the rows in some other existing structure, row management is essentially "free". If rows correspond to entries in two independently-managed structures then the matrix is also organized with multiple independently-managed banks. NoSQ's matrix is organized as two banks. One bank parallels the ROB—the reference count information for instruction A is at the same index as instruction A occupies in the ROB. A second bank parallels the CMap—the reference count information for logical register R is in row R.

Reference counts are updated on a row basis. Incrementing reference counts for the register(s) corresponding to a given entity means writing a bitvector into the corresponding matrix row. In NoSQ, the bitvector written by a renamed instruction into the ROB bank is the decoded representation of its destination physical register. For exam-

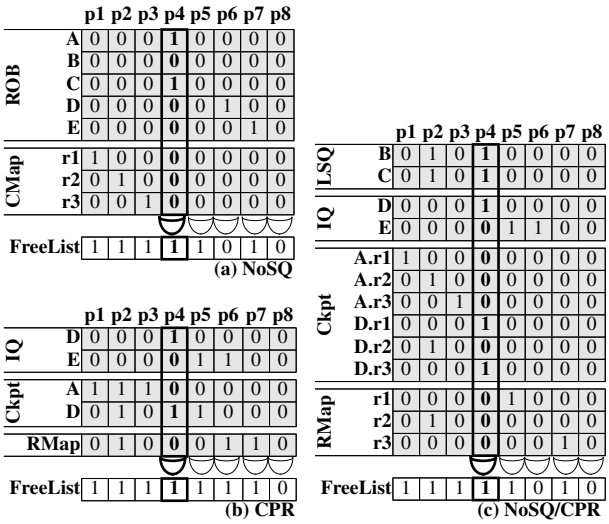

Figure 3. Unary reference counting mechanisms.

ple, instruction A writes a bitvector in which only p4's bit is set. At commit, an instruction writes this same bitvector into the appropriate row in the CMap bank. Each reference count matrix bank needs write bandwidth that matches that of the structure it parallels. In NoSQ, ROB bank write bandwidth matches rename bandwidth and CMap bank write bandwidth matches commit bandwidth.

Whereas unary matrix increments use write ports, decrements are even simpler, requiring only that the corresponding row be cleared or reset. Because resets don't require full ports—they require only wordlines, not bitlines—a reference count bank can support an arbitrary number of logical decrements per cycle using a bitmask. Unary reference counts can support single-cycle squash recovery this way. During normal NoSQ operation, the ROB bank's reset bitmask corresponds to committing instructions. On a squash, the reset mask corresponds to squashed instructions. The CMap bank does not support resets because every architectural register must be mapped to a physical register at all times.

Notice, unary reference count matrices do not need to support reads. Only row writes and resets and column ORs are needed.

CPR. Figure 2b shows CPR's unary reference counting scheme. In CPR, physical registers are referenced by (the sources of) un-executed in-flight instructions, by the rename map table, and by map table checkpoints. To track which registers appear in the rename map table (RMap), CPR maintains an auxiliary per-physical register bitvector of the same name. The RMap bitvector is updated at rename—bits corresponding to newly allocated physical registers that are mapped to instruction destinations are set, bits corresponding to over-written physical registers are cleared.

CPR's reference count matrix consists of two banks. The IQ bank parallels the issue queue. A renamed instruction writes a bitvector that encodes its source physical registers into its row. For example, instruction E's bitvector has the p5 and p6 bits set. An executed or squashed instruction clears its row. The Ckpt bank parallels the RMap checkpointing structure. Checkpoint creation—which happens at rename—writes the current RMap bitvector into the corresponding row. For example, the checkpoint corresponding to instruction D includes physical registers p2, p4, and p5. Checkpoint release—which happens at commit or squash—resets the row.

The reader may wonder why CPR can represent the physical register contents of a map table using a single bitvector (RMap) while NoSQ must represent the same information using per-logical register bitvectors (CMap). In NoSQ, the sparse representation is necessary because a physical register may appear in a map table multiple times. For example, in Figure 2a, the map table corresponding to instruction D maps both r1 and r3 to p4. Representing map table membership as a single bitvector is challenging in this scenario as bits for over-written registers can only be reset if those registers are not named elsewhere in the map table. In CPR, a given physical register appears at most once in any map table.

The reader may also recognize the RMap bitvector and its checkpoints as components of the Alpha 21264's CAM-style register renaming and map table checkpointing mechanisms, respectively [4]. Here, we use them for reference counting. Renaming and map table checkpointing are separate from reference counting. Our reference counting mechanisms can be used together with any renaming scheme. In fact, techniques like NoSQ which can map different logical registers to the same physical register cannot use the 21264's CAM-style renaming scheme, requiring the more traditional RAM-style scheme instead.

NoSQ/CPR. Figure 2c shows a unary reference counting scheme for NoSQ/CPR. There are three banks. The LSQ bank tracks reads by loads and stores and resembles NoSQ's ROB bank. The IQ bank tracks reads by un-executed non-memory instructions and resembles CPR's IQ bank. The Ckpt bank tracks physical registers named in map table checkpoints and resembles CPR's Ckpt bank. However, because a NoSQ/CPR hybrid allows map tables to name the same physical register multiple times, the contents of each map table are represented as they are in NoSQ—not as one bitvector, but as a bitvector per logical register. When a checkpoint is created, all RMap bitvectors are copied *en masse* to a block in the Ckpt bank. This inefficiency motivates the use of binary reference count representations.

IV. Optimization: Binary Reference Counts

A binary reference count is implemented as an n-bit register. Attached to the register is a tree of n-bit adders that processes increments and decrements. Aside from the contents of the counter register, the adder tree has one input for each instruction that can increment or decrement a reference count per cycle. For instance, if reference counts are incremented at rename and decremented at commit and the processor can both rename and commit two instructions per cycle, then a given reference count adder tree has five inputs. Figures 4a and 4b each show binary counters with two increment inputs (i0, i1) and two decrement inputs (d0, d1). Carry-save adders (cs+ blocks in the figure) can be used to reduce the height of this tree. As with unary reference counts, an OR gate reduces the register to a single free/allocated bit. A processor has one register-adder block for each physical register.

The same bitvectors that are written into the matrix in a unary representation are used as the inputs to the adder trees in a binary representation. In NoSQ, the bitvector each renamed instruction writes into the unary ROB matrix is the decoded form of its destination physical register. In a binary counter implementation, each bit in this bitvector is an input to the adder tree of the corresponding register. In a two-way superscalar processor, the first renamed instruction's bitvector supplies the i0 inputs; the second renamed instruction's bitvector provides the i1 inputs. The counters corresponding to these instructions' destination registers are incremented

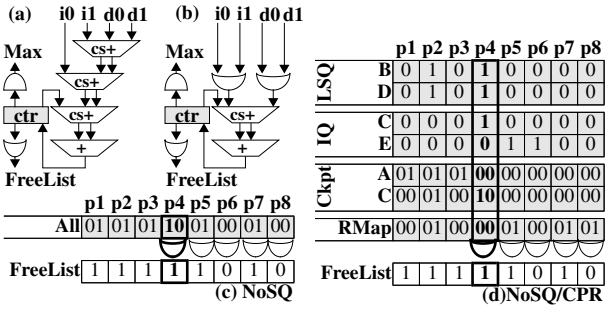

Figure 4. Binary and hybrid reference counting mechanisms.

by one along the i0 and i1 inputs, respectively. Counters corresponding to other registers are not incremented. The bitvectors for the two committing instructions act as the d0 and d1 inputs. For decrements, each bit in the bitvector is sign-extended to match adder width.

Binary counters can support resets, but this is not sufficient to support single-cycle squashing which requires decrementing each counter by some arbitrary amount. Because computing the arbitrary decrement amounts is difficult, squash-related decrements are processed by walking the squashed instructions. This generally has little performance impact [1].

Even ignoring non-support for single-cycle squashing, binary counters are at a disadvantage relative to unary matrices because register storage is larger per-bit than RAM storage and because of adder overhead. In practice then, binary counters provide an advantage over unary matrices only under certain conditions.

One such condition is that the maximum practical value of any reference count is much smaller than the theoretical maximum, allowing counters and adders to be made narrow. In NoSQ, the theoretical maximum value of a reference count is equal to the number of logical registers plus the number of in-flight loads. In practice, however, most NoSQ counter values are either zero or one and values over three—corresponding to at least three in-flight loads communicating with one store—are rare. And so NoSQ's binary counters can be implemented with only two bits. ANDing the bits in a counter (the Max output in Figures 4a and 4b) detects whether that counter has reached its maximum value and can no longer accept increments.

A second condition that favors binary representations is when the number of individual increments and decrements to a given reference counter can be restricted to one (of each) per cycle in a way that is both easy to implement and doesn't degrade performance. For example, in a two-way superscalar NoSQ processor the reference count of a single register could theoretically be incremented by both renamed instructions. However, instruction sequences that would bring this about—e.g., consecutive loads communicating with the same store—are both rare and easy to detect. Once detected, multiple same-cycle increments to the same register are avoided by forcing the loads to rename in consecutive cycles. With only one increment and one decrement per cycle, the adder tree collapses to a single three-input carry-save adder. The increment input to the adder is formed by bitwise ORing the bitvectors of the renaming instructions. The decrement input is formed similarly by ORing the bitvectors of the committed instructions. Figure 4b illustrates this optimization. The adder tree in Figure 4a can process up to two increments and two decrements per cycle—it has a separate input for each increment and decrement. In Figure 4b, increments and decrements are limited to one each per cycle and the adder tree has one increment input (i0 OR i1) and one decrement input (d0 OR d1).

Figure 4c shows a binary reference counting scheme for NoSQ. A single 2-bit binary counter replaces an entire column bitvector that spans both ROB and CMap banks.

CPR. Figure 4 doesn't include a binary reference counting design for CPR because CPR does not benefit from such a design. For IQ reference counts, which track register reads, increments and decrements are not naturally limited to one per cycle—multiple instructions in a rename group commonly read the same register. Meanwhile, Ckpt reference counts are quite dense even in a unary representation. With 16 checkpoints, it doesn't pay to replace a 16-bit vector with a 4-bit counter and an adder tree.

NoSQ/CPR. Binary reference counters can dramatically reduce the cost of reference counting for NoSQ/CPR. Most of the cost of the unary scheme comes from the RMap and Ckpt banks which represent registers named in map tables. Because of NoSQ's use of physical register sharing, each logical map table must be represented as a collection of bitvectors, one per logical register.

In Figure 4d, RMap is implemented as an array of 2-bit binary counters with associated three-input adders as in NoSQ. In contrast with NoSQ, NoSQ/CPR's RMap counter vector is both incremented *and* decremented at rename—decrements correspond to over-written registers. Each Ckpt bank entry is also an array of 2-bit counters. However, these do not have associated adders because they are never incrementally modified, only written and restored. As in CPR, the Ckpt bank is organized as a high-density RAM. Assuming 64 logical registers, the use of binary counters reduces the cost of the Ckpt bank by a factor of 32—from 64 bits per checkpoint per register to 2. The LSQ and IQ banks are left as unary matrices. Overall, NoSQ/CPR's reference counting scheme is a unary/binary hybrid in which every free/allocated bit is computed by ORing multiple unary bitvectors and 2-bit counters.

ACKNOWLEDGMENTS

The author thanks Adam Butts, Milo Martin, Vlad Petric, and Tingting Sha for discussions about physical register reference counting and for comments on this manuscript. This work was supported by NSF grants CCR-0238203 and CCF-0541292.

REFERENCES

[1] H. Akkary, R. Rajwar, and S. Srinivasan. "Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors." In *MICRO-36*, pages 423–434, Dec. 2003.

[2] A. Al-Zawawi, V. Reddy, E. Rotenberg, and H. Akkary. "Transparent Control Independence (TCI)." In *ISCA-34*, pages 448–459, Jun. 2007.

[3] S. Jourdan, R. Ronen, M. Bekerman, B. Shomar, and A. Yoaz. "A Novel Renaming Scheme to Exploit Value Temporal Locality Through Physical Register Reuse and Unification." In *MICRO-31*, pages 216–225, Dec. 1998.

[4] J. Keller. "The 21264: An alpha processor with out-of-order execution." In 9th Annual Microprocessor Forum, Oct. 1996.

[5] A. Moshovos and G. Sohi. "Streamlining Inter-Operation Communication via Data Dependence Prediction." In *MICRO-30*, pages 235–245, Dec. 1997.

[6] M. Moudgill, K. Pingali, and S. Vassiliadis. "Register Renaming and Dynamic Speculation: An Alternative Approach." In *MICRO-26*, pages 202–213, Dec. 1993.

[7] D. Oehmke, N. Binkert, T. Mudge, and S. Reinhardt. "How To Fake 1000 Registers." In *MICRO-38*, pages 7–18, Nov. 2005.

[8] V. Petric, T. Sha, and A. Roth. "RENO: A Rename-Based Instruction Optimizer." In *ISCA-32*, pages 98–109, Jun. 2005.

[9] T. Sha, M. Martin, and A. Roth. "NoSQ: Store-Load Communication without a Store Queue." In *MICRO-39*, pages 285–296, Dec. 2006.

[10] S. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton. "Continual Flow Pipelines." In *ASPLOS-11*, pages 107–119, Oct. 2004.