

## Flexible Register Management using Reference Counting

Steven Battle  
Drexel University  
sjb328@drexel.edu

Andrew D. Hilton  
IBM Corporation  
adhilton@us.ibm.com

Mark Hempstead  
Drexel University  
mhempstead@coe.drexel.edu

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

### Abstract

*Conventional out-of-order processors that use a unified physical register file allocate and reclaim registers explicitly using a free list that operates as a circular queue. We describe and evaluate a more flexible register management scheme—reference counting.*

*We implement reference counting using a bit-matrix with a column for every physical register and a row for every entity that can hold a physical register, e.g., an in-flight instruction. Columns are NOR'ed together to create a bitvector free list from which registers are allocated using priority encoders.*

*We describe reference counting designs that support micro-architectural techniques including register file power gating, dynamic register move elimination, register file checkpointing, and latency tolerant execution. Performance and circuit simulation show that the energy cost of reference counting is low and is easily recouped by the savings of the techniques it enables.*

### 1. Introduction

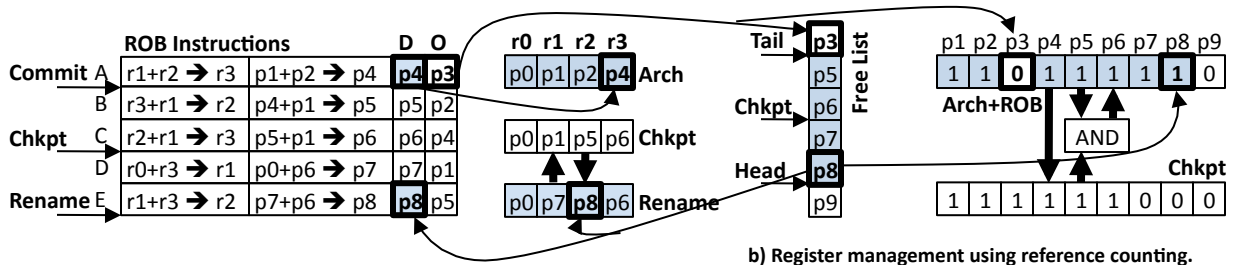
Most contemporary out-of-order processors use a unified physical register file to hold both architectural and speculative register state. This design requires fewer register value copies than one that uses a separate architectural register file and a ROB that holds destination register values. Processors manage registers—for the rest of the paper, we use *register* to mean physical register—using a free list that operates as a circular queue. Dispatching instructions dequeue (allocate) registers at one end. Committing instructions enqueue (free) the registers they overwrite at the other end. The destination registers of squashed instructions are freed in bulk by moving the free list head pointer.

This paper describes an alternative register management algorithm, *reference counting*, in which processor actions decrement reference counts rather than explicitly freeing registers. A register becomes free when its ref-

erence count becomes zero. Reference counting is more general and powerful than circular queue driven register management. In addition to conventional retirement-driven register reclamation, it supports execution-driven register reclamation used in latency-tolerant, scalable instruction window designs like CFP (Continual Flow Pipelines) [23] and BOLT (Better Out-of-Order Latency Tolerance) [8]. It also supports register sharing which allows multiple logical registers that contain the same value to share the same (physical) register, reducing execution overhead and (potentially) the height of the dynamic dataflow graph. Register sharing has been used in previously proposed techniques Unified Renaming [10], RENO (Rename Optimizer) [18], and NoSQ (No Store Queue) [22]. We show how reference counting supports these and other features, both alone and in combination.

The first description of reference counting used per-register multi-input up-down counters [16]. More recent designs use a two-dimensional bit-matrix [20]. The matrix has a column per register and a row per entity that can hold a register, e.g., an in-flight instruction or the rename map table. The bit at row  $e$  and column  $p$  is set if entity  $e$  holds register  $p$ . The bits in each column are NOR'ed together to create a bitvector-style free list from which registers are allocated using priority encoders. This structure is similar to the register renaming and map table checkpointing mechanism used in the Alpha 21264 [11], but is used only for register reclamation and can operate in conjunction with any renaming and checkpointing scheme. Reference counts are manipulated by operating on matrix bits. Different uses of reference counting require different matrix organizations. Depending on the type of register-holding entity and on manipulation bandwidth requirements, a given matrix row may be implemented using RAM or latches.

We use cycle-level performance simulation and circuit simulation, described in Section 2, to evaluate the performance benefits and area and power costs of various register reference counting configurations. Where analogous free-list configurations exist, we eval-



**a) Conventional queue-based register management.** There are four logical registers (r0–r3) and ten physical registers (p0–p9). Hardwired logical register r0 is mapped to hardwired register p0. The circular queue free list has six slots. Instruction A commits and frees overwritten (O) physical register p3 by adding it to the tail of the free list. It also writes its destination (D) physical register p4 into the architectural map table. Instruction E dispatches and allocates destination physical register p8 from the head of the free list. It also writes p8 into the rename map table. The processor took a checkpoint immediately after renaming instruction C. The checkpoint includes a copy of the map table and a copy of the free list head pointer. Restoring the checkpoint involves restoring the map table copy and the free list head pointer.

**b) Register management using reference counting.** The circular queue free list is replaced by a vector with one bit per physical register. When instruction A commits, it clears the bit corresponding to its overwritten register p3. When instruction E dispatches, it sets the bit corresponding to its destination register p8. This destination was selected from the free list using an encoder. Instruction C's checkpoint includes a copy of the map table and a copy of bitvector. Restoring the checkpoint involves restoring the map table copy and ANDing the checkpointed and active bitvectors.

**Figure 1.** Implementations of ROB register management.

uate those side-by-side. Section 3 introduces reference counting and shows how it can re-implement traditional register management. Section 4 presents register-file power-gating, a new application of reference counting that exploits its ability to allocate registers in any order. Section 5 discusses reference counting support for register sharing with an application to dynamic register move elimination. Although reference counting consumes more area and power than a traditional free list, the gating and move elimination optimizations it supports offset its energy cost. Sections 6 and 7 demonstrate reference counting support for speculative retirement and execution-driven register reclamation.

## 2. Methodology

We evaluate reference counting and its applications using both performance and circuit simulation.

**Performance evaluation.** Our cycle-level performance simulator executes user level x86\_64 code, breaking x86 instructions into RISC-like three-register micro-operations. The simulated core is modeled roughly after Nehalem. It is 4-wide issue with a 23-stage pipeline, 128-entry reorder buffer, 36-entry issue queue, and 96 rename registers. We model both a single-threaded configuration with 160 total registers and a dual-threaded configuration with 224 registers. The branch predictor is a 16 Kbyte 3-table PPM predictor [15]. The three-level cache hierarchy has 32 Kbyte 8-way set-associative, 3-cycle access instruction and data caches, a 256 Kbyte, 8-way set-associative, 10-cycle access L2 and an 8 MByte, 16-way set-associative 40-cycle L3. Main mem-

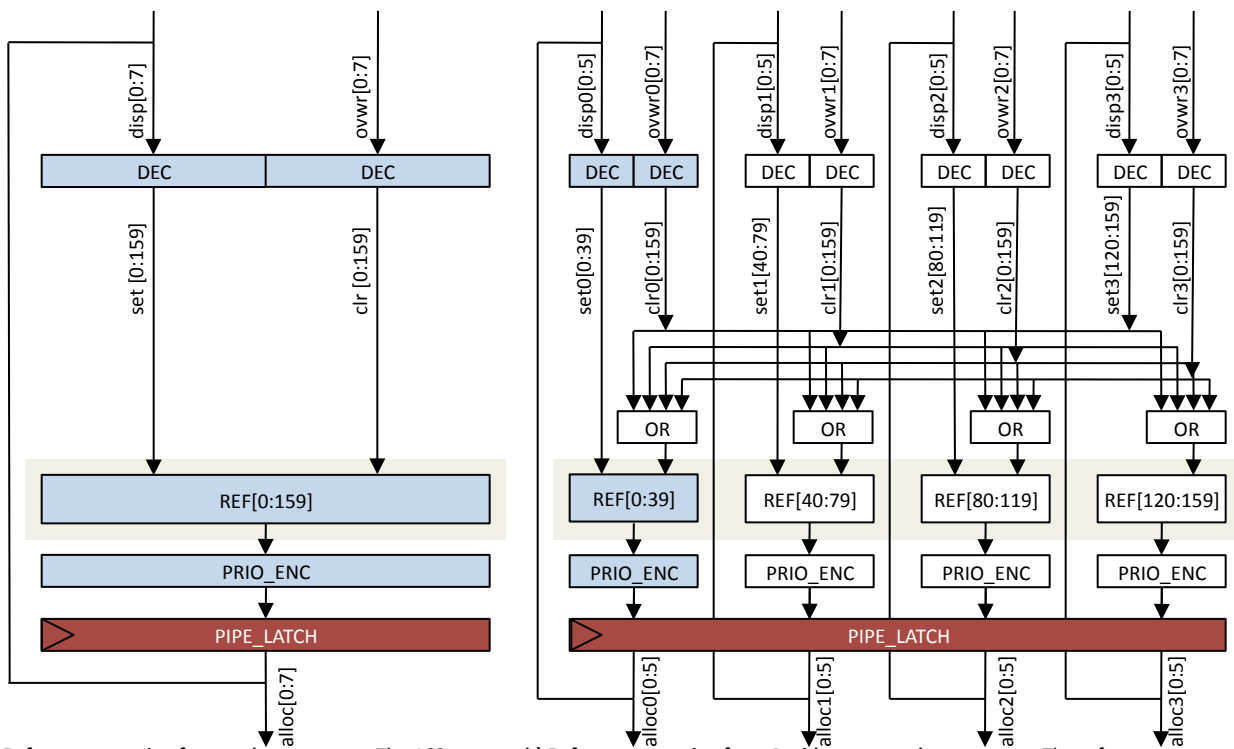
ory latency is 150 cycles—corresponding to a 3.2GHz clock—and bandwidth is 12.8 Gbytes/s.

We compiled the SPEC2006 benchmarks using gcc 4.3 at optimization level -O3. We simulate the benchmarks to completion on their training inputs, sampling 10 million of every 500 million instructions with 10 million instructions of cache and branch predictor warmup.

**Power and area estimation.** We created custom RTL for the reference counting structures and used the NCSU FabScalar memory generator [4] to create netlists for the register file, free list, and checkpoint SRAMs, all in NCSU's 45nm FreePDK CMOS technology. We derived power estimates for synthesized netlists using the power modeling features in Synopsys PrimeTime and also used HSPICE for transistor-level netlists. Our input activity traces included both the average port access rate and the average bit toggle rate to accurately reflect energy consumption. Static power is scaled to bring the P(st):P(dy) ratio in line with the typical 1:3 ratio [27] measured in commercial 45nm processors.

## 3. Conventional Register Reclamation

Out-of-order processors with unified register files use a variant of the MIPS R10000 register management algorithm [26]. The rename map table is a multi-ported RAM containing (physical) register numbers indexed by logical register number. The free list is a RAM of register numbers managed as a circular queue. For P registers and L logical registers the free list contains P–L entries as L registers are always mapped. Each entry is  $\log_2 P$ -bits wide. At rename, a register-writing instruction is al-



a) Reference counting for a scalar processor. The 160-register reference count vector supports one bit set (from dispatch) and one bit clear (from commit/overwrite) per cycle. Registers are allocated using a priority encoder. The allocated register is the dispatched destination register.

b) Reference counting for a 4-wide super-scalar processor. The reference counting vector is divided into 4 disjoint segments. 4  $\frac{1}{4}$  width priority encoders allocate 4 registers, 1 from each segment, per cycle. Because the dispatched destination registers are the allocated registers, those decoders are  $\frac{1}{4}$  wide as well. The 4 decoders for the overwritten registers are full width.

**Figure 2.** Reference counting mechanism.

located a destination register  $D$  from the head of the free list. A parallel map table lookup also notes the register previously allocated to the instruction's logical destination register—this is the overwritten physical register  $O$ . Both destination and overwritten registers are noted in the instruction's ROB entry. When an instruction commits, overwritten register  $O$  is freed and added to the free list at the tail. When an instruction is squashed, destination register  $D$  is freed and added to the free list at the head—this action only moves the free list head pointer.

Figure 1a shows an abstract processor with ten registers ( $p_0$ – $p_9$ ), four logical registers ( $r_0$ – $r_3$ ), and a six-slot free list. Logical register  $r_0$  is hardwired to the value zero and mapped to hardwired “register”  $p_0$ . The figure shows the architectural and rename (speculative) map tables and five instructions (A–E) in the ROB. Each instruction is shown in raw (logical register) and renamed (physical register) form along with its destination ( $D$ ) and overwritten ( $O$ ) registers. The figure shows two actions. Instruction A commits—it writes its destination register ( $p_4$ ) to the architectural map table and frees its overwritten register  $p_3$  by writing it at the tail of the free

list. Instruction E renames and dispatches—it allocates destination register  $p_8$  from the head of the free list and writes that register number into the rename map table.

**Reference counting formulation.** Figure 1b shows the corresponding reference counting implementation which uses a vector with one bit per register. Hardwired “register”  $p_0$  does not need to be reference counted because it is not allocated or freed. The vector has a 1 in position  $p$  if register  $p$  is mapped to either an architectural register or the destination of an instruction in the ROB. An instruction with destination register  $D$  and overwritten register  $O$  sets bit  $D$  when at dispatch and clears bit  $O$  when at commit. If it is squashed, it clears bit  $D$ . The bitvector is essentially an inverted free list.

**Scalar implementation.** A scalar processor allocates and frees at most one register every cycle, so a traditional free list needs one read port for allocation and one write port for reclamation. Reference count bitvectors support set/clear operations rather than read/write operations and so we implement the bits using set-reset (S-R) latches and the set/reset “ports” using encoders and decoders. A scalar processor with reference counting uses

two P-wide decoders, to both set and clear one bit per cycle. The allocator is a P-wide priority encoder. Figure 2a shows this design for a processor with 160 registers.

Because reference counting bits are implemented using latches and not flip-flops, the identification of free registers and the bitvector updates to indicate that the registers are no longer free cannot take place in the same pipeline stage. We avoid a cycle by “pre-selecting” a free register one pipeline stage (cycle) ahead. If the register is subsequently not used, the corresponding bit is not set and the register remains free.

**Superscalar implementation.** An N-wide superscalar processor allocates and frees up to N registers per cycle. Rather than supporting N-wide operation with N read and N write ports, a traditional free list is organized with one read port, one write port, and N entries per row. The queue discipline makes this organization conflict free, while latching recent results reduces accesses and facilitates management.

A superscalar processor with reference counting must allocate N registers per cycle and also support setting and clearing up to N bits per cycle. To support N-wide allocation, we divide the register space into N sets and make each encoder responsible for only P/N registers, *i.e.*, P/N-wide. The decoders that set bits corresponding to the destinations of dispatched instructions are divided in a similar way. The assignment of sets to superscalar renaming slots is rotated on a per cycle basis to avoid stalling dispatch when set 0 is empty. The decoders that clear bits corresponding to registers overwritten at commit cannot be divided without inducing conflicts—registers are not overwritten in allocation order—and are replicated at full P-width. Figure 2b shows reference counting for a 4-wide superscalar processor.

**Support for fast squash recovery.** The MIPS R10000 supported fast single-cycle recovery to a small number of checkpoints—snapshots of the rename map table that can be created and restored in a single cycle but do not support incremental updates. Checkpoint recovery frees an arbitrary number of in-flight destination registers in one cycle. The free list supports high-bandwidth squash-triggered reclamation by checkpointing and restoring the head pointer. Figure 1a shows a checkpoint taken after instruction C was renamed.

Reference counting bulk register release is implemented by adding a P-column C-row RAM where C is the number of checkpoints. When a checkpoint is created, the contents of the latch vector are written into the table in the corresponding row. When a checkpoint is restored, the table row is read and its contents are ANDed with those of the current reference count bitvector. Fig-

Component	$\mu\text{m}^2$	mW(dy)	mW(st)	mW(av)
<b>Single-threaded</b>				
Regfile, 160x64b, 6r3w	108733	22.70	2.49	9.97
Freelist, 27x32b, 1r1w	2279	2.08	0.06	0.82
<b>Regfile + freelist</b>	111012	<b>24.78</b>	<b>2.55</b>	<b>10.79</b>
Refcount, 1x160b, 4r4w	7622	1.24	0.08	0.72
Checkpoint, 4x160b, 1rw	2639	1.33	0.06	0.19
<b>Regfile + refcount</b>	118994	<b>25.27</b>	<b>2.63</b>	<b>10.88</b>
<b>Multi-threaded</b>				
Regfile, 224x64b, 6r3w	152226	24.50	3.49	11.00
Freelist, 27x32b, 1r1w	2279	2.08	0.06	0.82
<b>Regfile + freelist</b>	155190	<b>27.45</b>	<b>3.56</b>	<b>11.82</b>
Refcount, 2x224b, 4r4w	14515	1.54	0.14	1.00
Checkpoint, 4x224b, 1rw	3658	1.91	0.08	0.27
<b>Regfile + refcount</b>	170399	<b>27.79</b>	<b>3.71</b>	<b>12.27</b>

**Table 1.** Register management area and power for single- and multi-threaded processors.

ure 1b shows this action.

**Support for multi-threading.** Many contemporary out-of-order processors employ SMT (Simultaneous Multi-Threading) [12, 25]. Extending reference counting for SMT requires replicating the reference count bitvector on a per thread basis. Instructions update bits in the vector corresponding to their thread. Thread reference counts are maintained in separate vectors so that they can be checkpointed separately. However, the reference count checkpoint RAM is shared among the threads. Figure 4a shows reference counting support for an SMT processor with two threads. An additional 64 registers hold the architected state of a second thread.

**Area and power results.** Table 1 compares the power and area of the free list and reference counting designs for single- and dual-threaded processors. We use a low-leakage register file design [1] with 6 read ports and 3 write ports, typical for a 4-wide issue processor. In the table, dynamic power assumes maximum activity whereas average power is weighted by dynamic activity factors, collected from the SPEC benchmarks, and incorporates leakage power. For instance, the reference counting checkpoint RAM is a wide, high-powered structure but is written only once every ten cycles on average and read less frequently than that.

Our free list has 108 total entries—more than the 96 rename registers our simulated core has—to support simple register sharing [24] described in Section 5. For the single-threaded core, reference counting occupies 3 times more area and consumes 11% more power than a conventional free list, although the marginal overhead is small relative to the area and power consumption of the register file—7% and 1%, respectively. In the single-threaded configuration, reference counting and a conventional free list have identical performance.

For the multi-threaded core, we retain the 108 entry free list and add a set of head/tail pointers to statically

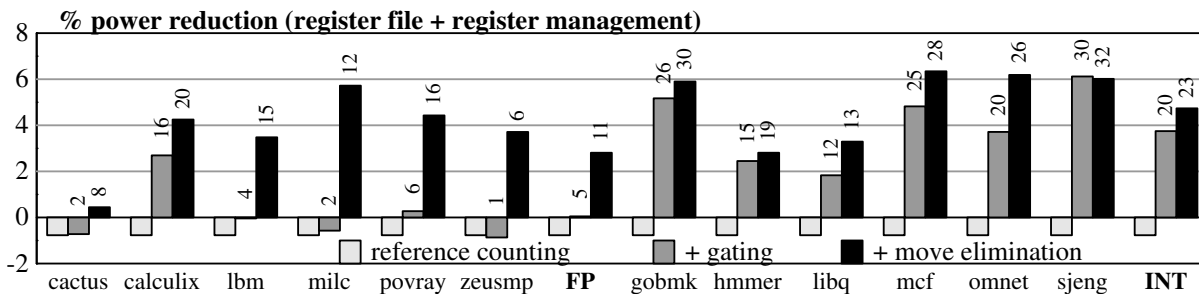


Figure 3. Power reduction due to register file power-gating. Percent of gated registers is shown above each bar.

partition it—and consequently register file—between the two threads. This setup also allows all 96 rename registers to be used by a single thread when one thread is quiesced. Dynamically partitioning the register file between two threads requires a more complicated setup—3 free lists with 108 slots each.

With reference counting, multi-threading requires additional hardware over a single-threaded design, both an additional vector and more bits per vector. In a multi-threaded processor, reference counting consumes 54% more power than a free list—4% of register file power. However, reference counting implements dynamic register partitioning which, our experiments show, improves throughput by 1%.

#### 4. Register File Power-Gating

Register files are typically sub-banked to reduce dynamic power consumption and access latency. Sub-banking chops up the bitlines into shorter, lower-capacitance segments. This enhances clock-gating by disabling pre-charge not only for unused ports but also for ports that are used on all banks except the one that contains the register accessed. The cost of banking is an additional set of lines that collect bank outputs using tri-states and distribute bank inputs. These incur both leakage and dynamic overheads. Our simulations show that banks of eight registers minimize total power by balancing the benefits of clock gating with the cost the additional lines. Even with banking and clock-gating, the register file consumes 15–20% of core power [6, 27].

Clock-gating reduces dynamic power consumption. Reference counting allows register file banks to be power-gated (V<sub>dd</sub>-gated), reducing leakage power as well. Register file power-gating is difficult to implement with a conventional free list. Even if the processor can dynamically resize the free list and take some registers out of circulation, the remaining registers are likely to be distributed evenly across the register file because the “un-disciplined” register overwriting continuously shuffles the contents of the free list. “Packing” the regis-

Component	$\mu\text{m}^2$	mW(dy)	mW(st)	mW(av)
<b>No gating</b>				
Regfile, 160x64b, 6r3w	108733	22.70	2.49	9.97
Refcount, 1x160b, 4r4w	7622	1.24	0.08	0.72
Checkpoint, 4x160b, 1rw	2639	1.33	0.06	0.19
<b>Regfile + refcount</b>	<b>118994</b>	<b>25.27</b>	<b>2.63</b>	<b>10.88</b>
<b>Gating</b>				
Regfile, 160x64b, 6r3w	111320	22.70	2.55	9.68
<b>Regfile + refcount</b>	<b>121581</b>	<b>25.27</b>	<b>2.69</b>	<b>10.59</b>

Table 2. Register file power-gating.

ter is unattractive as it involves a power-intensive and potentially slow procedure that adds overhead. Reference counting naturally packs registers, allowing power-gating to be applied opportunistically.

**Gated register file design.** We extend the existing banks to support power-gating with a PMOS gating transistor and driver. Our circuit simulations show that a gating transistor that is 2% the width of the gated SRAM bank minimizes leakage while maintaining switching performance [9, 19]. With this design, a register may be written two cycles after its bank is powered on. Because our simulated processor has a minimum of three cycles between issue and writeback, this design allows us to power on a bank when an instruction which writes a register in it issues. Table 2 shows the area and power of the power-gated register file. Power-gating support introduces area and leakage overheads of 2.4%.

**Reference counting formulation.** For power-gating, we add OR gates to the free-list, these determine if banks are “used”, *i.e.*, have any registers allocated. One important issue is the relationship of gating banks to allocation banks. Recall, in a 4-way superscalar processor, the register file is divided into 4 sets. To maximize the number of empty banks, the allocation sets are interleaved relative to the banks—bank 0 covers registers 0–7, allocation set 0 covers registers 0, 4, 8, etc.

**Gating algorithm and results.** Our HSPICE simulations show that a register bank must remain powered down for 15 cycles in order to offset the energy cost of power-up. This behavior requires that we build hysteresis into the power-gating algorithm to avoid powering



ROB				Rename Map Table				Reference Counts							
	Raw	Renamed	D	O	r0	r1	r2	r3	p1	p2	p3	p4	p5	p6	p7
A	r1+r2 → r3	p1+p2 → p4	p4.0	p3.0	"p0"	p1.0	p2.0	p4.0	1/0	1/0	0/0	1/0	0/0	0/0	0/0
B	r3 → r2		p4.1	p2.0	"p0"	p1.0	p4.1	p4.0	1/0	1/0	1/0	1/1	0/0	0/0	0/0
C	r0 → r3		"p0"	p4.0	"p0"	p1.0	p4.1	"p0"	1/0	1/0	1/0	1/1	0/0	0/0	0/0
D	r2 → r1	p4 → p5	p5.0	p1.0	"p0"	p5.0	p4.1	"p0"	1/0	1/0	1/0	1/1	1/0	0/0	0/0
E	r1+r3 → r2	p5+p0 → p6	p6.0	p4.1	"p0"	p5.0	p6.0	"p0"	1/0	1/0	1/0	1/1	1/0	1/0	0/0

There are five instructions (A-E), four logical registers (r0-r3), and 0 physical registers (p0-p7). Logical register r0 and "physical register" p0 are hardwired to the value 0. For every instruction, the figure shows the state of the Rename Map Table and Reference Counts after renaming. **Simple register sharing.** The processor supports "simple register sharing" on register p0, which does not require reference counting. Instruction C, which loads the constant 0 to register r3 exploits this sharing mode. Rather than allocating a new register (say p7) to r3 and dispatching an instruction that moves p0 to p7, r3 is set to point to p0 in the map table and no instruction is dispatched. **Move elimination using physical register sharing.** To support two-way sharing on the other registers, reference counts are extended to two bits and every register number in the map table and ROB is appended with a bit indicated which of the two reference count bits this entity "holds." Instruction B is a register move that is eliminated because only one of the two reference count bits on source register p4 (bit 0) is set. Instruction B is not dispatched, note the empty renamed instruction slot which would otherwise have contained the identity operation "p4 → p4". Its destination register, r2, is set to p4, and the second reference count bit for p4 is set. r3, written by instruction A and overwritten by C uses bit 0. r2, "written" by instruction B and overwritten by E uses bit 1. Instruction D is a move that is not eliminated because sharing for register p4 is already saturated.

**Figure 5.** Physical register sharing with an application to dynamic register move elimination.

hanging and surprisingly abundant fruit. Other sharing opportunities require speculation to identify and/or execution units modifications to exploit [18, 22].

**Simple register sharing.** Simple register sharing refers to sharing of hardwired values in registers—or pseudo-registers—that are never explicitly allocated or freed [24]. Our simulated core implements simple register sharing for the value zero, held in hardwired "register" p0. Our simulated core's micro-ISA has an explicit zero register, r0, and translates the common x86\_64 idiom of XOR'ing a register with itself to a move from r0. Simple register sharing can be implemented without reference counting, but does require additional free list entries to hold registers that would otherwise be allocated to architected registers. Our simulated core uses a 108 entry free list to allow up to 12 instructions to share p0.

In Figure 5, instruction C is a move from the zero register r0 that exploits simple register sharing. C is eliminated and its destination, r3, is mapped to p0. Being hardwired, p0 is not reference counted.

**General purpose register sharing.** Sharing of general purpose registers—even the restricted form associated with a single register move—is difficult to implement without reference counting. The reason is that responsibility for freeing the shared register has to be transferred from the instruction that overwrites the older sharer to the instruction that overwrites the younger one. This transfer is difficult to orchestrate. It may also have to be undone if the younger sharer is squashed. This complexity is illustrated using Figure 5. Eliminating register move B involves assigning the register allocated

Component	$\mu\text{m}^2$	mW(dy)	mW(st)	mW(av)
<b>No move elimination</b>				
Regfile, 160x64b, 6r3w	111320	22.70	2.55	9.68
Refcount, 1x160b, 4r4w	7574	1.24	0.08	0.72
Checkpoint, 4x160b, 1rw	2639	1.33	0.06	0.19
<b>Regfile + refcount</b>	<b>121533</b>	<b>25.27</b>	<b>2.69</b>	<b>10.59</b>
<b>Move elimination</b>				
Regfile, 160x64b, 6r3w	111320	22.70	2.55	9.25
Refcount, 2x160b, 4r4w	12317	1.32	0.12	0.84
Checkpoint, 4x320b, 1rw	5186	1.80	0.11	0.29
<b>Regfile + refcount</b>	<b>128823</b>	<b>25.82</b>	<b>2.75</b>	<b>10.38</b>

**Table 3.** Register management area and power for register sharing.

to r3, p4, to r2 as well. Without sharing, responsibility for freeing p4 resides with instruction C, the instruction that overwrites r3. With sharing that responsibility shifts to E, the instruction that overwrites r2. If E is squashed freeing responsibility returns to C, unless C has already committed in which case it remains with E and p4 must be freed on the spot.

Reference counting simplifies the book-keeping by tracking register occupancy in a central location and eliminating the need to assign register freeing responsibility to any specific instruction *a priori*.

**Reference counting formulation.** Figure 5 shows a reference counting formulation that supports a maximum sharing degree of two—a single register can be shared by at most two entities, where an entity is either an architected register or the destination of an in-flight instruction. This restricted implementation captures most sharing opportunities because any sharing group that comprises more than two entities is broken up into multiple sharing pairs.

The design replaces every reference count bit with

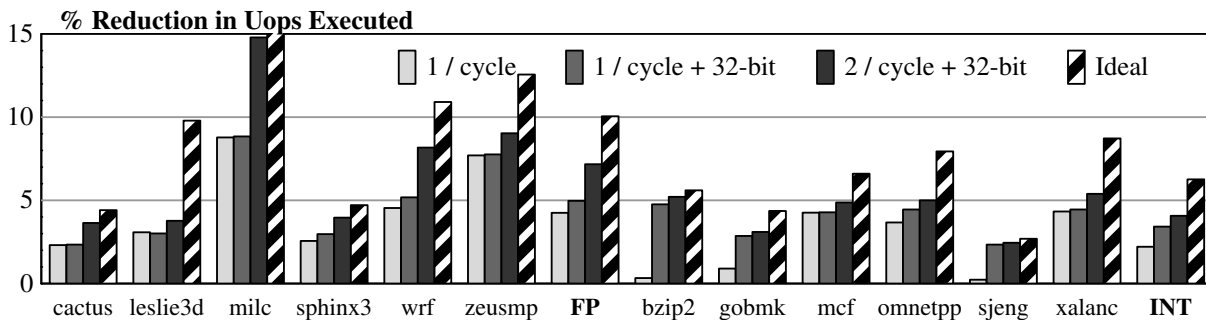


Figure 6. Reduction in  $\mu$ ops executed with move-elimination.

two bits—shared registers have both bits set, newly allocated registers have only bit 0. Reference count checkpoints are similarly extended. Register number fields in the map table and ROB are extended by one bit to track which of the two reference counting bits the architected register or in-flight instruction is using, respectively. This additional bit indicates which bit to clear at commit and allows the bit clearing operation to be a write as opposed to a read-modify-write. In Figure 5, instructions A (writing r3) and B (writing r2) share register p4—r3 uses bit 0 of p4’s reference count and r2 uses bit 1. Noting this information in the map table allows it to pass to instructions C and E which overwrite r3 and r2 respectively—note, C’s and E’s overwritten (O) register fields in the ROB. At commit, C clears bit 0 in p4’s reference count, and E clears bit 1.

Figure 4b shows the reference counting circuit for a scalar processor with register sharing. Register moves have to read the reference count bits of their input register to know whether sharing is already saturated and if it isn’t which bit to write. In Figure 5, instruction D is a move that is not eliminated because register p4 is already shared by A and B. Because register sharing serializes map table read with map table write, it requires rename to be pipelined over two stages with map table writes taking place in parallel with dispatch [18]. However, inserting a reference count read—which is a combinational read from latches—into the map table read-write sequence should not induce additional delays. If the move *is* eliminated, then the register speculatively allocated for it can be returned to the free list without disturbing younger instructions in the rename group—this would not be possible with a queue-style free list. If the move is *not* eliminated, it is not necessary to repair the inputs of younger instructions in the rename group that may depend on the move—it does not matter whether those instructions read the move’s source register or new destination register. Because only register moves read reference counts and fewer than 1/4th of dy-

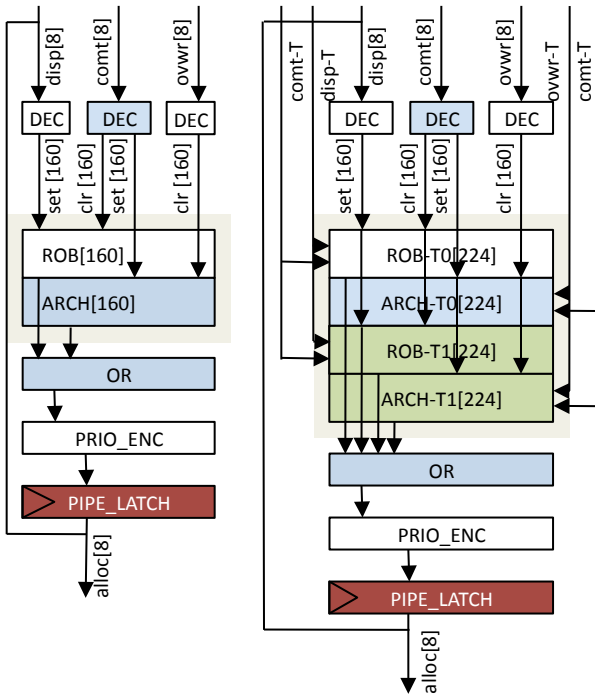
namic instructions are moves, our design uses a single reference count “read port”. If a rename group contains more than one move, the younger moves are simply not considered for sharing, they are allocated new registers and dispatched like non-move instructions.

Table 3 shows the area and power cost of reference counting support for register sharing with support for a single move elimination per cycle. The marginal cost over vanilla reference counting is an extra bitvector and read port in the primary circuit, and wider rows in the checkpoint RAM. No additional decoders are needed.

**Performance results.** Figure 6 shows reduction in  $\mu$ ops executed over the simple register sharing baseline. The first configuration (light grey) can detect and eliminate one register move per cycle, and permits two simultaneous sharers for any general purpose register.

The second configuration (grey) adds support for eliminating 32-bit moves. In x86.64, a 32-bit move (*e.g.*, `mov eax, ebx`) copies the low bits and clears the high bits of the 64-bit destination register. We extend move elimination to capture this case by adding an extra bit to map table entries and register specifiers in the issue queue. Renaming/eliminating a 32-bit register move sets this bit which consumer instructions propagate to the source register specifiers in their issue queue entries—the upper bits of these sources are treated as zero during execution. This feature benefits SpecINT programs which manipulate 32-bit data, *e.g.* `bzip2` and `sjeng`. Overall, this configuration eliminates 4% of dynamic  $\mu$ ops, offsetting the cost of reference counting by reducing register file accesses—register file power drops from 9.68 mW to 9.25 mW in Table 3. And by reducing register file occupancy, it also enhances register-file power-gating as shown in the right (dark) bar in Figure 3. Move elimination allows power-gating 6% more registers on SpecFP and 3% more registers on SpecINT, and much as 10% more registers on individual programs.

The third configuration in Figure 6 (dark grey) retains 32-bit move elimination support but can detect and elim-



**a) Reference counting for retirement stage checkpointing (left).** The reference vector is replicated to split tracking of registers bound to architectural state from those bound to in-flight destinations. A third decoder clears the bit for the committing instruction’s destination register in the in-flight vector while setting it in the architectural vector.

**b) Reference counting for retirement stage checkpointing and multithreading (right).** This combination uses four reference count vectors—in-flight and architectural for each thread. “Port” decoders bits grow with dispatch/commit width, not thread count.

**Figure 7.** Reference counting for scalar processors with speculative retirement and SMT.

inate two moves per cycle. The fourth (black stripes) models unlimited sharing on both per cycle and per register bases. Unlimited sharing doubles the number of instructions eliminated to 10% on SpecFP and over 6% on SpecINT, but at a larger hardware cost—additional reference count bitvectors, a wider reference count checkpoint table, and multiple reference count read ports.

## 6. Speculative Retirement

Several recently proposed architecture and microarchitecture techniques including TM (Transactional Memory) [3,7,14,17] rely on speculative retirement, *i.e.*, the ability to execute, complete, and release the pipeline resources of large numbers of instructions, and then either commit or abort their effects in bulk. The register state of speculatively retired instructions is collapsed into a checkpoint created at (speculative) retirement.

Component	$\mu\text{m}^2$	mW(dy)	mW(st)	mW(av)
<b>Single-threaded</b>				
Regfile-sb, 160x64b, 6r3w	130828	24.40	3.00	11.30
Freelist, 27x32b, 1r1w	2279	2.08	0.06	0.82
<b>Regfile-sb + freelist</b>	<b>133107</b>	<b>26.48</b>	<b>3.06</b>	12.12
Regfile, 160x64b, 6r3w	108733	22.7	2.49	9.97
Recount, 2x160b, 4r4w4rw	<b>10641</b>	<b>1.38</b>	<b>0.11</b>	<b>0.86</b>
Checkpoint, 4x160b, 1rw	2639	1.33	0.06	0.19
<b>Regfile + recount</b>	<b>130828</b>	<b>25.41</b>	<b>2.66</b>	<b>11.02</b>
<b>Multi-threaded</b>				
Regfile-sb, 224x64b, 6r3w	185900	26.60	4.27	12.10
Freelist, 27x32b, 1r1w	2279	2.08	0.06	0.82
<b>Regfile-sb + freelist</b>	<b>188179</b>	<b>28.68</b>	<b>4.33</b>	<b>12.92</b>
Regfile, 224x64b, 6r3w	152226	24.50	3.49	11.00
Recount, 4x224b, 4r4w4rw	23373	2.54	0.22	1.68
Checkpoint, 4x224b, 1rw	3658	1.91	0.08	0.27
<b>Regfile + freelist</b>	<b>179257</b>	<b>28.95</b>	<b>3.71</b>	<b>12.95</b>

**Table 4.** Register mamangement area and power for speculative retirement.

Implementing speculative retirement in a processor with a unified register file requires a retirement map table. It is this map table—not the rename map table—whose contents are checkpointed and restored to support checkpoint/abort operations. Additionally, it is necessary to either checkpoint the contents of the register file itself or to “pin down” the registers named in the retirement map table, temporarily taking them out of circulation. With a conventional free list, checkpointing register file values is the only option. Value checkpointing is implemented with low area and latency overheads and moderate power overhead using “shadow bitcells” [5]. It also allows a checkpointed thread to use the full complement of rename registers. In an SMT processor, tracking thread register usage in a bitvector—a simple form of reference counting—allows threads to checkpoint independently using a single set of shadow bitcells.

**Reference counting formulation.** Reference counting supports the pinning approach. Pinning does not add area, power, or latency to the register file, but reduces the number of rename registers a checkpointed thread can use. This is not as bad as it first appears because only registers mapped to logical registers that are redefined are lost. Redefinition of all logical registers in a one speculative episode is unlikely, *e.g.*, an integer program is unlikely to redefine many floating point registers.

Figure 7a shows reference counting support for speculative retirement on a scalar processor. The reference vector is replicated—one replica tracks register usage by the ROB, the second tracks usage by the retirement map table (ARCH). These two vectors are manipulated using three P-wide decoders. One decoder sets the bit for the dispatched destination register in the ROB vector. A second decoder clears the bit for the overwritten register in the ARCH vector. The third decoder “moves” the bit for the destination register of the committing instruc-

tion from the ROB vector to the ARCH vector. Speculative retirement checkpoints can share the RAM used to hold checkpoints for single-cycle branch misprediction recovery. Figure 7b shows reference counting which supports both speculative retirement and SMT.

Table 4 shows the hardware cost of implementing retirement-stage register file checkpointing for speculative retirement for both single-threaded and dual-threaded 4-wide superscalar processors. In the single-threaded core, the reference counting/pinning implementation has 10% lower power consumption and 2% lower area than the freelist/shadow bitcell implementation. In the multi-threaded core, the power advantage of reference counting disappears because of the increased cost of reference counting multiple threads. In both cores, reference counting also supports register-file power-gating, physical register sharing, and other optimizations whose benefits are not accounted for here.

## 7. Execution-Driven Register Reclamation

In retirement-driven reclamation, a register is reclaimed when the instruction that overwrites the corresponding logical register commits. Some proposed uses of register reference counting implement execution-driven register reclamation. Here, a register is reclaimed earlier, when the overwriting instruction is dispatched and all readers have executed [16].

Execution-driven register reclamation scales the register file, breaks the instruction-granularity state recovery mechanism that supports branch speculation and precise traps. CPR (Checkpoint Processing and Recovery) [2] marries execution-driven register reclamation with precise state. CPR creates periodic rename map-table checkpoints. A checkpoint holds all registers named in it until it is released. CPR then implements execution-driven register reclamation between checkpoints. Precise recovery to any inter-checkpoint instruction is implemented by rolling back to an older checkpoint and executing forward—the additional execution is called checkpoint overhead.

Execution-driven register reclamation supports non-blocking execution. A non-blocking processor uses dependence-tracking (*e.g.*, wakeup/select) to identify instructions that depend on long-latency cache misses, drain them from the window, and release their issue queue entries and (physical) registers, freeing up these resources for use by younger instructions. The miss-dependent instructions wait in a buffer until the miss returns, at which time they reacquire registers and issue queue entries, re-enter the window, and execute. CFP (Continual Flow Pipeline) [23] is a non-blocking

Component	$\mu\text{m}^2$	mW(dy)	mW(st)	mW(av)
<b>CPR/CFP</b>				
Regfile, 224x64, 6r3w	152226	24.50	3.49	11.66
Refcount, 2x224b, 4r4w	14515	1.54	0.14	1.00
Checkpoint, 8x224, 1rw	5196	2.11	0.11	0.32
Issue queue, 32x224, 4c4w	37831	5.06	1.13	2.55
<b>Regfile + refcount</b>	<b>209768</b>	<b>33.21</b>	<b>4.87</b>	<b>15.53</b>
<b>BOLT</b>				
Regfile, 224x64, 6r3w	152226	24.50	3.49	11.70
Refcount, 4x224b, 4r4w4rw	23373	2.54	0.22	1.68
Checkpoint, 4x224, 1rw	3658	1.91	0.08	0.27
Issue queue, 32x224, 2c2w	18256	1.78	0.55	0.59
<b>Regfile + refcount</b>	<b>197513</b>	<b>30.73</b>	<b>4.34</b>	<b>14.24</b>

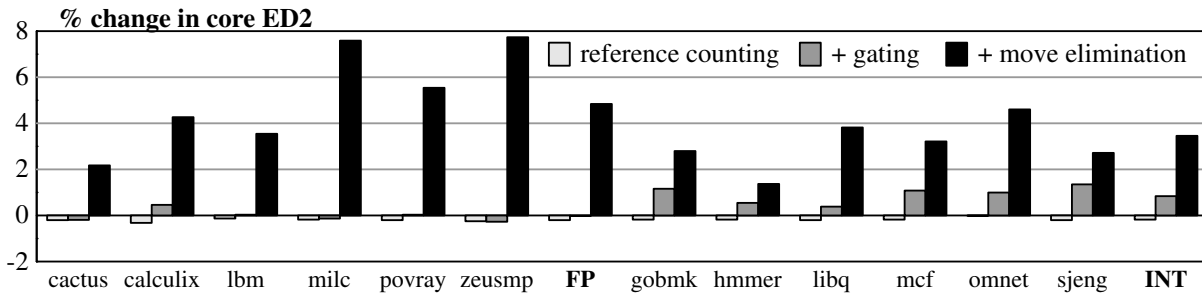
**Table 5.** Register management area and power for execution-driven register reclamation.

microarchitecture based on CPR. BOLT (Better Out-of-Order Latency Tolerance) proposal [8] hybridizes execution-driven with retirement-driven reclamation to eliminate checkpoint overhead, reduce checkpointing requirements, and simplify re-execution of deferred instruction slices by storing those slices in program order.

Execution-driven reclamation is difficult to implement using traditional free lists, which require assigning responsibility for freeing a given register to a particular instruction *a priori*. With execution-driven reclamation, the instruction responsible for freeing a given register may be the last reader of the register to execute. Because execution occurs out of-order, this instruction cannot be easily identified in advance.

**Reference counting formulation.** We describe two reference counting designs—one supports CPR/CFP, the second supports BOLT. The CPR/CFP design uses a reference vector implemented using latches and two reference RAMs. The P-element reference vector tracks the register contents of the rename map table. It is identical to the one that implements retirement-driven register reclamation described in Section 3 and shown in Figure 2 except that the overwritten registers are those that are overwritten at rename rather than those that are overwritten at commit. The RAMs represent reference counts of registers held by checkpoints and by issue queue entries, respectively. The checkpoint RAM resembles the RAM used to implement fast recovery in a ROB microarchitecture, except that its contents are NOR’ed into the free-list column wise. The issue queue RAM has one row per issue queue entry, one write port per dispatch slot and one “clear” port per issue slot—a clear port is a wordline-only port that resets an entire row. Again, the columns of the issue queue reference RAM are NOR’ed into the free list.

BOLT is a latency-tolerant microarchitecture that uses both speculative retirement and multithreading. It has a hybrid register management algorithm. Primary



**Figure 8.** Processor ED<sup>2</sup> reduction due to reference counting enabled optimization.

(*i.e.*, first-pass) execution uses retirement-driven reclamation with support for speculative retirement. Deferred instructions re-execute using an available thread context and use execution-driven reclamation. BOLT reference counting combines the mechanism for speculative retirement and SMT (Section 6, Table 4) with the issue queue reference counting RAM used for CPR/CFP. BOLT’s issue queue RAM needs fewer write and clear ports than CPR’s because BOLT uses execution-driven reclamation only during re-execution.

Table 5 shows the area and power of the reference counting structures required to implement CPR/CFP (top) and BOLT (bottom). For ease of comparison, both micro-architectures are implemented on a 224-register SMT substrates although only BOLT exploits SMT. CFP cannot reuse the additional map tables present in an SMT to re-allocate re-executing instructions, requiring a bespoke physical-to-physical map table instead. One of the previously unstated advantages of BOLT over CPR/CFP—previously stated advantages include better performance and lower implementation complexity, and a lower dynamic instruction execution count—is its greatly reduced reliance on issue queue based reference counting. The issue queue reference counting structure is large and power hungry, every dispatching instruction requires a 224-bit write. CPR/CFP write and clear rows in this structure for *every* dynamic instruction, incurring a power surcharge that is equal to 23% of register file power. BOLT uses issue queue reference counting to track only re-executing instructions and incurs a surcharge of only 5% as a result.

## 8. ED<sup>2</sup> Impact

Although reference counting has a small energy (and area) overhead relative to a traditional free list implementation for a vanilla microarchitecture, the techniques it enables—specifically register-file power-gating and dynamic move elimination—conserve enough energy to turn it into a net energy reduction technique. Further, by improving performance in addition to reducing energy consumption, dynamic move elimination makes refer-

ence counting an ED<sup>2</sup> reduction technique, *i.e.*, a technique that is more effective at trading power for performance than dynamic voltage and frequency scaling (DVFS) and which is suitable for a processor that implements DVFS [13].

Whereas we have detailed models of the register file and reference counting logic, we don’t have similar models for the rest of the processor and therefore must make some assumptions. Specifically, we use published data from the Power7 microprocessor [27] to scale our results. In the Power7, the register file accounts for 20% of total core power—specifically, 14% of its leakage power and 24% of its dynamic power. Also, register file dynamic power is three times higher than register file leakage power. Given these we compute  $E_{l\_base} = E_{l\_base\_RF}/0.14$  and  $E_{l\_base\_nonRF} = E_{l\_base} - E_{l\_base\_RF}$ . Similarly,  $E_{d\_base} = E_{l\_base\_RF} \times 3/0.24$  and  $E_{d\_base\_nonRF} = E_{d\_base} - E_{d\_base\_RF}$ .  $E_{base} = E_{l\_base} + E_{d\_base}$ . For a non-baseline,  $E_l = (E_{l\_base\_nonRF} + (P_{l\_RF} / P_{l\_base\_RF}) \times E_{l\_base\_RF}) \times CC / CC_{base}$ .  $P$  is power and  $CC$  is cycle count. Essentially, we leave the non-register file portion untouched and scale the register file portion using the circuit modeled ratio of old to new register file—here ‘register file’ includes register management structures, free list or reference counting. We then scale by relative execution time. We apply the same technique to dynamic power, scaling this time by relative instruction execution count. We discount the execution of eliminated register moves by a factor of 0.5 because elimination does not affect fetch, decode, or commit.  $E_d = (E_{d\_base\_nonRF} + (P_{d\_RF} / P_{d\_base\_RF}) \times E_{d\_base\_RF}) \times (IC - 0.5 \times IC_{elim\_move}) / IC_{base}$ .  $IC$  is instruction count.

Figure 8 shows the relative ED<sup>2</sup> of 4-wide superscalar processors with different reference counting configurations and reference counting enabled optimizations. The first bar corresponds to a configuration that replaces a conventional free list with reference counting but implements no other optimization. This bar is essentially invisible because there is no performance change and negligible energy overhead. The next bar adds register file gating. Here again, the changes are small although they

generally are for the better. Register-file gating does not affect performance, and while it does reduce leakage energy, this benefit is small at 45nm. The energy savings will be more significant in smaller, leakier technologies. Move elimination has a larger affect, reducing ED<sup>2</sup> by 3.5% and 4.8% for SpecINT and SpecFP by improving register file gating opportunities and reducing execution time. Latency tolerant execution, another technique enabled by reference counting also reduces ED<sup>2</sup> [8].

## 9. Conclusion

Reference counting is a flexible register management scheme for processors that enables optimizations not possible with a traditional free list. We presented a set of reference counting designs that support a range of micro-architectures. Circuit simulation shows that reference counting has a slight area and energy overhead over a traditional free list for a vanilla microarchitecture. However, reference counting supports optimizations like register-file power-gating and dynamic register sharing that offset this overhead, even turning it to an advantage. Finally, the real power of reference counting is that it supports designs in which traditional free lists cannot function at all, specifically scalable-window latency tolerant designs.

## 10. Acknowledgments

We thank the reviewers for their comments. This work was supported by multi-institution NSF grant CCF-1017184 (UPenn) and CCF-1017654 (Drexel).

## References

- [1] A. Agarwal, R. Kaushik, and R. Krishnamurthy. A leakage-tolerant low-leakage register file with conditional sleep transistor. In *Proc. 2004 Intl. SOC Conference*, Sep. 2004.
- [2] H. Akkary, R. Rajwar, and S. Srinivasan. Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors. In *Proc. 36th Intl. Symp. on Microarchitecture*, Dec. 2003.
- [3] C. Blundell, M. Martin, and T. Wensich. InvisiFence: Performance-Transparent Memory Ordering in Conventional Multiprocessors. In *Proc. 36th Intl. Symp. on Computer Architecture*, Jun. 2009.
- [4] N. K. Choudhary, S. V. Wadhavkar, T. A. Shah, H. Mayukh, J. Gandhi, B. H. Dwiell, S. Navada, H. H. Najaf-abadi, and E. Rotenberg. Fabscalar: Composing synthesizable rtl designs of arbitrary cores within a canonical superscalar template. In *Proc. 38th Intl. Symp. on Computer Architecture*, 2011.
- [5] O. Ergin, D. Balkan, D. Ponomarev, and K. Ghose. Increasing Processor Performance Through Early Register Release. In *Proc. 22nd Intl. Conf. on Computer Design*, Oct. 2004.
- [6] X. Guan and Y. Fei. Reducing power consumption of embedded processors through register file partitioning and compiler support. In *Proc. 2008 Intl. Conf. on Application-Specific Systems, Architectures and Processors*, Jul. 2008.
- [7] M. Herlihy and J. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proc. 20th Intl. Symp. on Computer Architecture*, Jun. 1993.
- [8] A. Hilton and A. Roth. BOLT: Energy-Efficient Out-of-Order Latency-Tolerant Processing. In *Proc. 16th Intl. Symp. on High Performance Computer Architecture*, Jan. 2010.
- [9] Z. Hu, A. Buyuktosunoglu, V. Srinivasan, V. Zyuban, H. Jacobson, and P. Bose. Microarchitectural techniques for power gating of execution units. In *Proc. 2004 Intl. Symp. on Low Power Electronics and Design*, 2004.
- [10] 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 *Proc. 31st Intl. Symp. on Microarchitecture*, Dec. 1998.
- [11] J. Keller. The 21264: An alpha processor with out-of-order execution. In 9th Annual Microprocessor Forum, Oct. 1996.
- [12] D. Koufaty and D. Marr. HyperThreading Technology in the NetBurst Microarchitecture. *IEEE MICRO*, 23(2), Mar. 2003.
- [13] A. Martin, M. Nystroem, and P. Penzes. ET2: A Metric for Time and Energy Efficiency of Computation. Technical Report CSTR:2001.007, CalTech, 2001.
- [14] J. Martinez, J. Renau, M. Huang, M. Prvulovic, and J. Torrellas. Cherry: Checkpointed Early Resource Recycling in Out-of-Order Microprocessors. In *Proc. 35th Intl. Symp. on Microarchitecture*, Nov. 2002.
- [15] P. Michaud. A PPM-like, Tag-Based Branch Predictor. *Journal of Instruction Level Parallelism*, 7(1), Apr. 2005.
- [16] M. Moudgill, K. Pingali, and S. Vassiliadis. Register Renaming and Dynamic Speculation: An Alternative Approach. In *Proc. 26th Intl. Symp. on Microarchitecture*, Dec. 1993.
- [17] N. Neelakantam, R. Rajwar, S. Srinivas, U. Srinivasan, and C. Zilles. Hardware Atomicity for Reliable Software Speculation. In *Proc. 34th Intl. Symp. on Computer Architecture*, Jun. 2007.
- [18] V. Petric, T. Sha, and A. Roth. RENO: A Rename-Based Instruction Optimizer. In *Proc. 32nd Intl. Symp. on Computer Architecture*, Jun. 2005.
- [19] M. Powell, S.-H. Yang, B. Falsafi, K. Roy, and T. N. Vijaykumar. Gated-vdd: a circuit technique to reduce leakage in deep-submicron cache memories. In *Proc. 2000 Intl. Symp. on Low-Power Electronics and Design*, 2000.
- [20] A. Roth. Physical Register Reference Counting. *Computer Architecture Letters*, 7(1), Jan. 2008.
- [21] S. Roy, N. Ranganathan, and S. Katkooori. State-Retentive Power Gating of Register Files in Multi-core Processors featuring Multithreaded In-Order Cores. *IEEE Transactions on Computers*, PP(99), 2010.
- [22] T. Sha, M. Martin, and A. Roth. NoSQ: Store-Load Communication without a Store Queue. In *Proc. 39th Intl. Symp. on Microarchitecture*, Dec. 2006.
- [23] S. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton. Continual Flow Pipelines. In *Proc. 11th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Oct. 2004.
- [24] L. Tran, N. Nelson, F. Ngai, S. Dropsho, and M. Huang. Dynamically Reducing Pressure on the Physical Register File Through Simple Register Sharing. In *Proc. 2004 Intl. Symp. on Performance Analysis of Systems and Software*, Mar. 2004.
- [25] D. Tullsen, S. Eggers, and H. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proc. 22nd Intl. Symp. on Computer Architecture*, Jun. 1995.
- [26] K. Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, Apr. 1996.
- [27] V. Zyuban, J. Friedrich, C. J. Gonzalez, R. Rao, M. D. Brown, M. M. Ziegler, H. Jacobson, S. Islam, S. Chu, P. Kartschoke, G. Fiorenza, M. Boersma, and J. A. Culp. Power optimization methodology for the ibm power7 microprocessor. *IBM Journal of Research and Development*, 55(3), May–Jun. 2011.