CRIB: Consolidated Rename, Issue, and Bypass

Erika Gunadi
Intel Corporation
Santa Clara, CA
erika.gunadi@gmail.com

Mikko Lipasti
Department of Electrical and Computer Engineering
University of Wisconsin - Madison
mikko@ece.wisc.edu

ABSTRACT

Conventional high-performance processors utilize register renaming and complex broadcast-based scheduling logic to steer instructions into a small number of heavily-pipelined execution lanes. This requires multiple complex structures and repeated dependency resolution, imposing a significant dynamic power overhead. This paper advocates in-place execution of instructions, a power-saving, pipeline-free approach that consolidates rename, issue, and bypass logic into one structure—the CRIB—while simultaneously eliminating the need for a multiported register file, instead storing architected state in a simple rank of latches. CRIB achieves the high IPC of an out-of-order machine while keeping the execution core clean, simple, and low power. The datapath within a CRIB structure is purely combinational, eliminating most of the clocked elements in the core while keeping a fully synchronous yet high-frequency design. Experimental results match the IPC and cycle time of a baseline out-of-order design while reducing dynamic energy consumption by more than 60% in affected structures.

Categories and Subject Descriptors

B.0 [**Hardware**]: General.

General Terms

Performance, Design

1. INTRODUCTION

High-performance core microarchitecture is stagnating: one can convincingly argue that today's high-end processor cores are largely evolutionary derivatives of the breakthrough, canonical designs from the mid-1990s. For example, Intel's Nehalem (Core i7) bears a strong similarity to the Pentium Pro design of 1995, while IBM's Power7 and AMD's recent Bulldozer look remarkably familiar to fans of Digital's late-90s Alpha 21264. Of course, substantial improvements have been made in areas such as clock frequency, power efficiency, branch prediction accuracy, and scaling of instruction window structures to better tolerate memory latency. Nevertheless, the logical operation of these machines has not changed much. Instructions are fetched and decoded in parallel groups. Architected sources and destinations are renamed into a large physical pool of rename registers that eliminates false dependences. Instructions are then dispatched into reservation stations where they wait until data and structural hazards are resolved. From there instructions issue onto a limited set of execution pipelines that fetch operands from a register file, a reorder buffer, or the bypass network. Finally, instructions wait in the reorder buffer to commit in program order. These operations are heavily pipelined, though limits on power consumption have forced recent designs to back off on this to some extent. Fundamentally, these machines are organized around their ALUs and cache ports, and implement incredibly complex control and operand delivery structures to maximize the utilization of these heavily-pipelined execution units.

This paper raises fundamental questions about the suitability of this organization for future processors. While device density continues to increase, thermal, power delivery, and energy supply constraints will severely complicate any effort to scale performance up via either frequency improvements or further attempts at refining the canonical 1990s out-of-order core to extract additional instruction-level parallelism (ILP). The industry trend toward many-core processors further limits the power budget for each processing core. While certain application domains can tolerate low-power in-order cores by compensating for their low single-thread performance with abundant thread-level parallelism [32], general-purpose systems still require the single-thread performance and high ILP provided by aggressive, out-of-order processors. For this class of systems, an ideal many-core processor core would provide the seemingly contradictory attributes of modest area, low power consumption, high ILP, and competitive frequency.

To meet these objectives, we set out to design the simplest possible execution core that still fully exposes ILP by eliminating false dependences and enforcing in-order execution only in case of true data dependences. Inspired by the Ultrascalar proposal [38], instructions are executed *in place* at distributed execution stations that consolidate renaming, issue logic, and bypassing in a single *CRIB* structure. As shown in Figure 1, the execution and communication resources are laid out in two dimensions, placing logical register names in horizontally-spaced columns and instructions ascending vertically in program order. The design avoids pipelining and other temporal reuse features that impose dynamic power overheads. There is no register file per se, as logical register values simply flow from the bottom of the CRIB to the top, and, once ready, are captured in a simple rank of latches. Each new definition of a logical register value simply inserts itself into this vertical flow, so that any horizontal bisection of the CRIB provides a precise snapshot of the architected state at the corresponding instruction boundary. Scheduling decisions are made locally in response to wake-up signals, so it is trivially easy to tolerate variable execution latency. This lets us incorporate power-efficient techniques like cache banking and line buffers without any additional control complexity or overhead from latency misspeculation.

This deceptively simple and straightforward design concept for an execution core poses many challenges for microarchitects, but the detailed design presented in this paper demonstrate that is feasible. Relatively straightforward extensions and enhancements of this fundamental design philosophy result in an elegant design that is eminently realizable and delivers excellent performance and cycle time along with dramatic reductions in power consumption.

The rest of this paper is organized as follows: Section 2 explains in-place execution and the CRIB concept; Section 3 describes our methodology; Section 4 explains details of the hardware design; Section 5 shows performance and energy results; Section 6 discusses prior work in this area; and Section 7 concludes the paper.

2. IN PLACE EXECUTION IN CRIB

To eliminate the power overheads that are inherent in conventional out of order processors, we sought inspiration from an unlikely source: the Ultrascalar microarchitecture [38], a massively wide 64-issue design that was proposed during the heyday of high-ILP

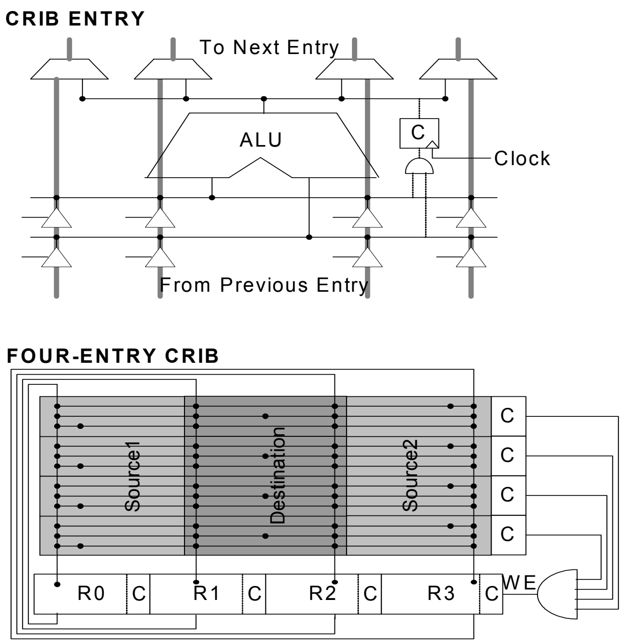


**Figure 1. Simplified Example of the CRIB Concept. **

research. Ultrascalar's key innovation was a conceptually simple yet scalable approach for out-of-order scheduling and operand delivery. Instead of scheduling logic that steered a handful of instructions per cycle from a large pool of waiting instructions to a narrow set of execution resources, instructions were inserted at dispatch time into a large pool of execution stations, were they were woken up and executed *in place*, with false dependences resolved via a localized renaming algorithm. A scalable parallel prefix network spanned the large pool of execution stations, providing low latency wake-up signals and operand delivery and resulting in potentially impressive levels of ILP with competitive cycle time, but offset by significant area increases and unknown power cost.

The key idea from Ultrascalar that inspires our proposal is *in-place execution*. Rather than delivering instructions and operands to a small set of heavily-pipelined execution units constantly evaluating complex broadcast-based scheduling logic, instructions simply wait at pre-assigned stations for their operands to arrive. As shown in the Ultrascalar work, this drastically simplifies dependence resolution and eliminates complex and poorly-scalable operations like renaming and broadcast-based instruction scheduling, and avoids the need for a heavily multiported register file. However, most importantly, it substantially reduces *activity* in the processor, since instructions are idle until their wake-up signal arrives, then they evaluate and directly pass along their results to waiting dependent instructions. Since activity determines overall power consumption, the net effect is that instructions are spending more of their allotted energy budget performing the actual work of computation, and much less on overhead for resource management, scheduling, and data movement. Furthermore, the fact that instructions are assigned dedicated execution stations eliminates the need for pipelining access to the execution units. In fact, very little in the datapath needs to be pipelined, leading to significant area and clock power savings from eliminated pipeline latches.

In the CRIB processor, as in Ultrascalar, the RAT, the RS, and the ROB are consolidated into one structure, which we call the consolidated rename/issue/bypass block, or *CRIB*. Figure 1 shows a simplified example of a CRIB, with four entries and four architected registers. Each logical register has its own column that vertically spans the CRIB. Instructions from the front end are placed into the

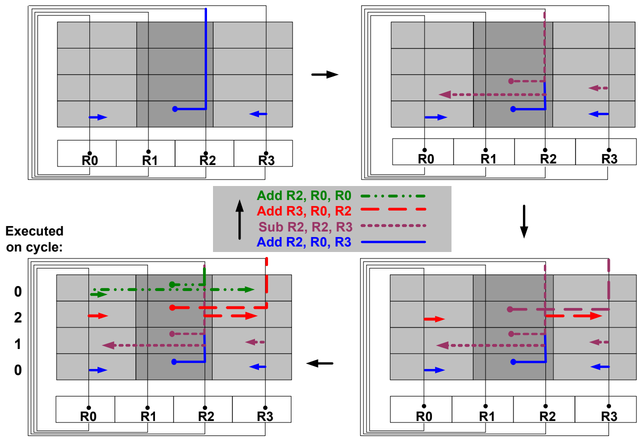


Figure 2. CRIB Example.

CRIB in program order, starting from the bottom. Each CRIB entry contains routing logic that connects logical register columns, which are spaced horizontally, to an ALU. It also connects the ALU result back to the appropriate register column. Each instruction in the CRIB taps its source operands from the register columns. It then overwrites its destination register column accordingly. An example of instruction execution in CRIB is shown in Figure 2. Once inserted, instructions stay in the CRIB until all entries finish executing. Data propagation inside the CRIB is done combinationally without latching. This is possible since an ALU serves the instruction until it leaves the CRIB, at which point the result is latched into the ARF. A completion bit is added to each register column and each CRIB entry to maintain synchronous wake-up and completion. When the completion bits in all entries are set, the ARF is clocked and new instructions are inserted into the CRIB.

Since our performance objectives are more modest than Ultrascalar's, we only consider configurations with relatively few instructions, so we have no need for the parallel prefix network for operand delivery. Instead, we only provide simple linear connectivity, which scales comfortably up to the instruction window sizes we need to achieve comparable performance to today's high-end processors. As we will show, this design delivers competitive cycle time and performance per clock cycle while requiring 60% less energy than a conventional out-of-order design. The energy saving mainly comes from eliminating redundant structures, reducing data and tag movement, and the smaller instruction window that is enabled by CRIB's shallower execution pipeline.

2.1. Towards a More Realistic CRIB Processor

In this section we explain how CRIB concept is expanded into a realistic execution core. With only the four entries described above, CRIB will have a very limited reach for extracting instruction level parallelism. Thus, scaling the CRIB up is necessary for high performance. One way to scale up the CRIB is by simply increasing the number of CRIB entries. However, because all instructions have to stay in the CRIB until all of them finish executing, simply increasing the number of CRIB entries will lead to low utilization and will delay dispatch of later instructions into the window. Instead, we add entries and partition the CRIB, as shown in Figure 3. The partitioned CRIB is maintained in a circular fashion. Instances of the ARF latches are inserted between CRIB partition. Only the ARF at the head of the partition has the committed state of the program. As a partition completes execution, the head or commit pointer is moved to the next ARF instance. To allow register values to travel through the ARF to the next partition without getting latched, transparent flip-flops are used [6][7]. The ARF flip flops that are not holding the

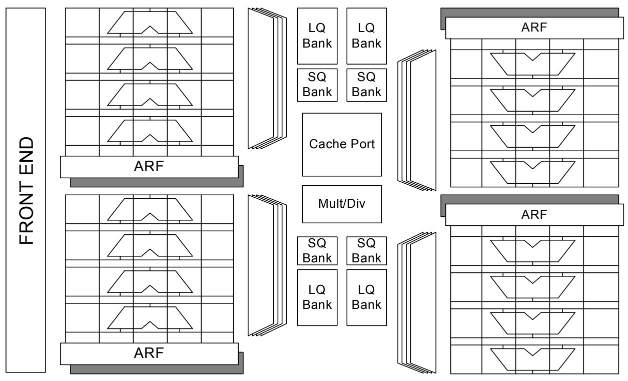


Figure 3. Four-Partition CRIB.

committed state of the program are left transparent, reducing latch power. Ready instructions across multiple CRIB partitions can evaluate concurrently, exposing additional parallelism. Data values take an additional cycle to travel from one partition to the next, adding a cycle of delay between dependent instructions that are in adjacent CRIB partitions. While data is not latched within each CRIB partition, completion bits are latched at every entry to ensure a fully-synchronous design. A partitioned CRIB with a few entries per partition can prevent dispatch bottlenecks due to improved utilization, but it incurs extra cost due to additional ARF latches between partitions.

So far we have assumed simple integer instructions with matching ALU resources that are easily replicated in each CRIB entry. However, for other instructions, full replication is too expensive and alternative approaches must be considered. We will explain how different instruction types are handled in CRIB in the following paragraphs. For simplicity, all cases are discussed assuming a four-entry partitioned CRIB.

Memory Instructions. CRIB entries do not latch instruction results; instead each CRIB entry's ALU continuously drives its result into its logical register destination column. Applying the same approach for loads would require holding cache ports for the entire evaluation of a CRIB partition. This would create too much demand for cache ports. Instead, CRIB uses the load queue to latch the data and holds the read port to the load queue as long as load instructions stay inside the CRIB. When a load is dispatched into the CRIB, it is assigned a load queue entry. As the load becomes ready, its address is written into the load queue and its address valid bit is set. A simple FIFO arbiter checks the load queue every cycle to find ready instructions to be executed. The load is then executed where it access the cache and associatively searches the store queue. The data is then written to the data portion of the load queue. To drive the data up the destination register wire, the read port of the load queue data is held open as long as the load resides in the CRIB.

To avoid the need for an excessive number of full read ports, the load queue is banked to match the number of CRIB partitions. Each load queue bank only needs as many read ports as the number of loads in each CRIB partition. Because load instructions compose no more than 40%-50% of total instructions in our workloads, we limit the number of loads in each CRIB partition to two. When the limit is reached, the dispatch logic stalls. A similar limitation is also imposed on store instructions: only one store is allowed per partition. When a store instruction is ready and issued, it writes its data into the store queue. When the CRIB partition finishes, the store address and data are sent to the write buffer or cache.

Cache Banking and Line Buffers. While widely used for lower level caches to provide high bandwidth, cache banking is not commonly used in L1 caches of out-of-order machines, since it introduces an element of uncertainty in cache latency due to bank conflicts. Because modern out-of-order scheduling logic relies on speculative wake-up of load dependents, delays from bank conflicts create additional complexity to squash the pipeline and replay the instructions. However, CRIB does not separate dependence and data linking, thus it can tolerate the variable latency introduced by banking. Address bits identifying the required cache bank are presented to the select logic to prevent collisions. With banking, CRIB can easily double the load bandwidth without the cost of replay on bank conflicts. Crossbars are used to connect the load queue dispatch ports and cache banks. Doubling load bandwidth also requires doubling store queue associative search capability. However, since the store queue is fairly small, it is feasible to do without much effect on overall execution core power consumption.

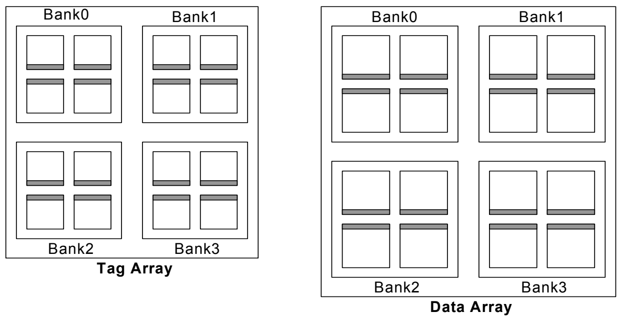


Figure 4. Cache Organization with Line Buffers.

Line buffers have been proposed as a power efficient filtering scheme for cache access, but line buffer misses also induce variable access delay [39]. While prior work maintains line buffers as separate entities, we utilize the pipeline latches that already exist in the cache. In pipelined caches with sub-arrays, the sense amp outputs in each sub-array are captured in a pipeline latch; we simply use these latches as a line buffer, and can save one cycle off the access time whenever there is a hit. This cache organization with line buffers is shown in Figure 4 (line buffers are shown as shaded areas).

Load and Store Ordering. Loads and stores are ordered aggressively, as in conventional out-of-order machines. Ready younger loads execute eagerly even when there is an older incomplete store instruction. In case of ordering violations, conventional designs require a flushing recovery, while CRIB can get by with a simple re-execution. As shown in Figure 5, an invalidation signal from a misordered older store causes the violating load queue entry to deassert its completion bit. This leads its dependent, the SUB instruction in Figure 5, to deassert its own completion bit, and so on. Once the load instruction retrieves the correct data, the completion bit is asserted once again. As illustrated in Figure 5, the ADD instruction does not need to deassert its completion bit because it does not depend on the output of the load instruction. Since the recovery mechanism is very lightweight, CRIB does not require an sophisticated memory dependence predictor to govern load-store ordering [10]. Instead, CRIB simply assumes that no misordering will occur and issues load instructions as soon as they are ready, and pays virtually no penalty when violations are resolved.

Complex Integer Instructions. Complex integer resources, such as multiplication and division units, are shared since they are too expensive to replicate. The units are pipelined as in conventional machines. However, since CRIB does not latch their results, the final pipeline stage has to be occupied by the instruction until the result is latched back into the ARF. Once the result is written back, the next instruction in the pipeline moves to the last stage and drives its output to its destination register column. To avoid deadlock, complex

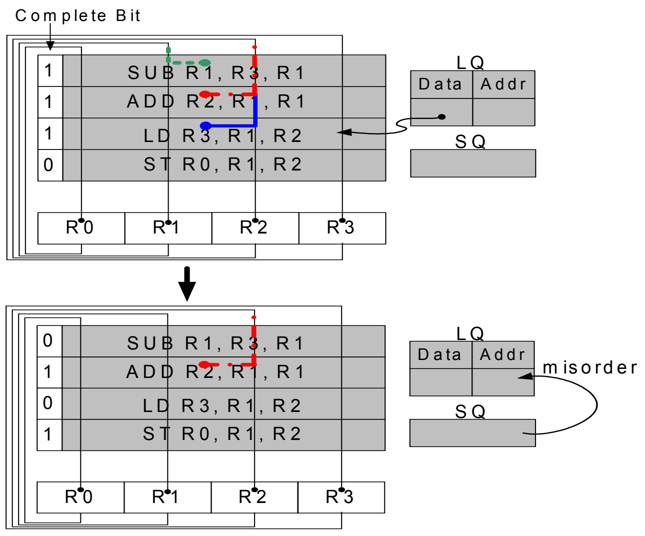


Figure 5. Recovery on Load-Store Mis-speculation.

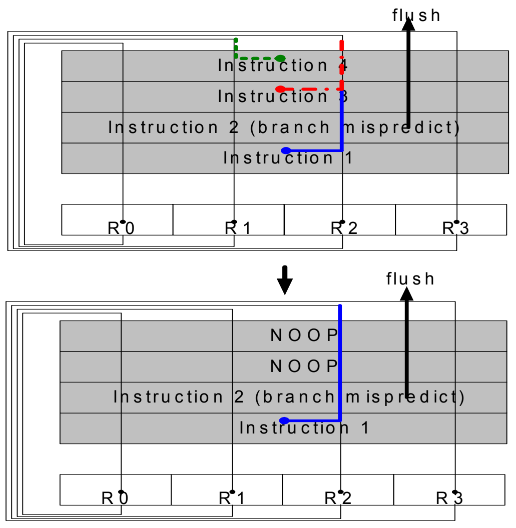


Figure 6. Branch Misprediction Example.

integer instructions per CRIB are limited to the number of units (e.g. one multiply per CRIB partition), and they must be issued in order.

Branch Instructions. Without explicit register renaming, branch misprediction handling in CRIB is much simpler than in conventional out-of-order machines, since no rename registers have to be reclaimed. On a branch misprediction, the mispredicted branch drives a global signal up the CRIB that forces younger instructions to transform into NOPs. The NOPs stop overwriting their logical destination registers. Once the execution finishes, the ARF captures the correct state of the program at the branch boundary, as shown in Figure 6. A few additional cycles are needed to clean up the store queue and the load queue from younger instructions. The same approach is also used for precise exception handling.

Floating-point instructions. Floating point (FP) instructions are handled in the same way as integer instructions. An FP CRIB is placed side by side with the integer CRIB. This FP CRIB is able to handle floating point execution and other non-integer execution such as SIMD extensions. FP instructions are placed into FP CRIB. Integer SIMD units are replicated for each entry of the CRIB. FP units such as addition, multiplication, division, square-root, and others are shared in the same way as the complex integer units in the integer CRIB. A similar limitation to the number of FP instructions in a CRIB partition is also imposed, i.e. if there are two FP add-multiply, one FP division, and one SSE mul/div, then the number of FP add-multiply instructions in the CRIB is limited to two, and FP division instruction is one, and so on.

The machine state is now defined by both INT head and FP head. However, only one of the head is incremented during retirement as only one CRIB (Integer or FP) can retire each cycle. Communication between integer and FP occurs through load and move operations. These communication instructions are split into two operations that will reside in both integer and FP CRIB. The first half will write into the communication buffer of the consumer and the second half will use the buffer to drive its logical register output. The communication buffers are allocated in order at dispatch to ensure consistent naming for both operations.

The block diagram is shown in Figure 7. As shown, integer and FP CRIB are decoupled. The FP CRIB need not have the same number of entries as integer CRIB. A total commit order between FP and integer CRIBs is maintained with a simple labeling scheme.

2.2. CRIB with Deeper Pipeline

In a realistic design, area constraints will limit the number of CRIB partitions to a small enough number that CRIB occupancy will often stall dispatch. Since a CRIB partition is not reclaimed until all instructions in it have finished, the occupancy of that partition is determined by the latency of the longest path through the partition's dependence graph. One way to reduce that latency, and improve occupancy, is to more deeply pipeline the CRIB hardware and clock it at a faster rate. Deeper pipelining comes at a high cost in conventional out-of-order machines, since the data has to be latched in every pipeline stage, leading to additional area and power. But more importantly, the latch latency overhead, latch setup time, and clock propagation, all tend to reduce the benefit of increased clock rate. As the pipeline grows deeper, the total latency to finish the same computation increases due to latch overhead.

However, since CRIB only latches the control/status bits, which are few and off the critical path, CRIB is able to implement deeper pipelining without suffering the same overhead as conventional machines. Figure 9 shows completion bit propagation for CRIB and the modifications necessary to implement a deeper pipeline. The shaded areas are transparent latches. As seen in the left picture, the data propagation path in CRIB mostly consists of two segments. The first is the propagation of data from the ARF to the CRIB entry and execution. The second segment is the propagation of the result to the next ARF. There is a vast imbalance between the first and second segments, with the first segment being the critical one. In the right picture, the critical path is now separated into two segments, in which the first one is the data propagation from the ARF to the CRIB entry while the second one is the execution itself. Completion bits are no longer latched in the ARF. Instead, they are propagated using transparent latches. The ALU latency is used as the critical path to determine the new cycle time. Analysis in Section 4 shows that we can achieve a 2x frequency gain with this pipelining technique. We found that the result can propagate as far as the next four entries in the CRIB within a single clock period, reducing the length of the critical dependence path and improving CRIB partition utilization.

Figure 8 shows pipeline diagram comparison between CRIB and an out-of-order machine. It also shows how the pipeline changes as deeper pipelining is added. As shown, CRIB consolidates the issue, register file read, and execute stages into one. While within CRIB dependent instructions can issue back-to-back, an output propagation cycle is added when dependences cross CRIB partitions. With a deeper pipeline, CRIB is clocked at a 2x rate as shown in the shaded

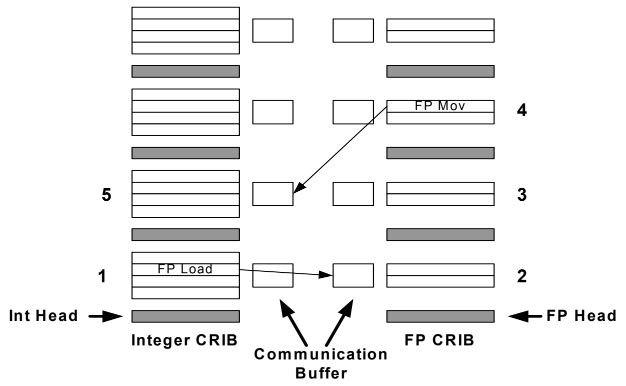


Figure 7. FP CRIB. \

areas. Input propagation, ALU execution, and output propagation each take one minor cycle. Communication with the rest of the processor, dispatch and writeback, is done at major cycle boundaries.

3. EVALUATION METHODOLOGY

Physical Modeling. To assess latency, area, and power consumption of our proposed design, we model each structure in the out-of-order window as well as structures in our CRIB execution core using a combination of synthesis and CACTI 5.3 [8]. Structures that requires small storage such as the RAT, the RS, and the CRIB are modeled using Verilog and synthesized using Synopsys Design Compiler with TSMC tcbnptc 0.65nm CMOS standard cell library. Large structures that consists mostly of SRAM such as the ROB and the PRF are modeled using CACTI 5.3[8]

Because we need comparable energy and delay for both regular flip-flops and transparent flip-flops, we use results from [6] rather than flip-flops from the standard library. We have verified the energy and delay between the two flip-flop designs and they have very similar characteristics. We use a port reduction technique [12][13] to model the PRF in the baseline machine, reducing the number of read ports from eight to four. However, this reduction is not taken into account in the architectural timing simulation; eight ports are assumed.

Architectural Modeling. To obtain performance and activity counts, we use an execution-driven x86 simulator derived from Bochs [9]. An internal RISC ISA is designed to crack x86 instructions into uops that run in the timing part of the simulator. We use two baselines, a small out-of-order core with a 24-entry ROB and an Intel Nehalem-like [36] machine, as described in Table 1. Baseline1 is configured to have similar window size as a 7-partition CRIB (justified in Section 5), while Baseline2 is configured to represent

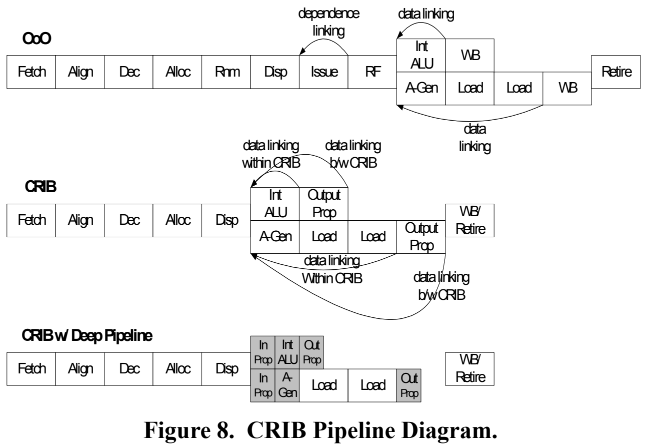


Figure 8. CRIB Pipeline Diagram.

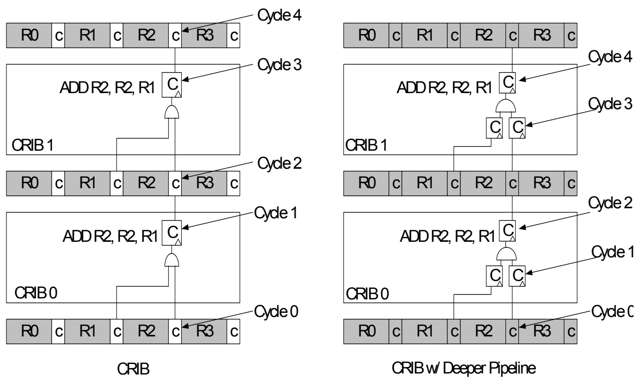


Figure 9. CRIB with Deeper Pipeline. .

today's microprocessor configurations. Activity counts are added to the simulator in a similar manner as Wattch [11].

Benchmarks. The SPEC 2006 benchmark suite is used for results presented in this paper. All benchmarks were compiled with gcc version 3.3.2 with -O3 optimization. We only use six benchmarks from SPEC INT 2006 and eight benchmarks from SPEC FP 2006 based on the redundancy analysis in [12]. The Pinpoint tool [15] is used to get simpoints [16][17] for each of these benchmarks. The benchmarks are fast-forwarded into the first simpoint and run for 100M instructions. Throughout the paper the baseline 1 with a configuration shown in Table 1 is used for any normalized results. The IPC for both baseline is presented in Table 2.

4. CRIB PHYSICAL DESIGN

This section provides additional details related to the physical design we are proposing for CRIB.

Architected Register File Size. In this study, we chose to use the x86-64 architecture as it is the most commercially significant architecture today. The x86-64 ISA has 16 general purpose registers and an EFLAGS register. For the internal uop support, 3 temporary registers are added. To support x87 floating point, there are 8 FP register, a control register, a status register, and a tag register. MMX technology requires 8 MMX registers that are aliased to the x87 FP registers. SSE adds 16 XMM registers and an MXSCR. This specification results in 1248 bits and 1904 bits of spatial register state overlaid on the integer CRIB and the FP CRIB.Given this specification this section describes the design of the CRIB with an objective of reaching a competitive cycle time without sacrificing back-to-back execution of dependent instructions.

Routing and Glitch Avoidance. Each entry of CRIB consists of an ALU, input router, output router, and flip-flops for storing opcodes and tags. Tags are stored using one-hot encoding to simplify input and output routing. Input routing consists of tri-state buffers enabled by the source tags driving the input to ALU, as shown in Figure 1. For output routing, a 2-to-1 mux is used to choose whether to overwrite the register value to the next entry with its ALU result or to continue the previous value. Since the datapath in CRIB uses few if any pipeline latches to minimize power consumed by flip-flops and clock-tree, it can be prone to glitching. To avoid glitch power, a tri-state buffer is added to the output of the ALU. The tri-state is controlled by completion bit and is enabled on the clock cycle following the execution. Thus the output of the ALU is exposed to its consumers when the result is stable. The output router with glitch avoidance is shown in Figure 10.

Delay, Cycle Time, and Energy. The baseline cycle time is derived from a 36-entry reservation station as described in [5]. The combinational delay for the reservation station according to the synthesis

Table 1.Baseline Out-of-Order Machine Configurations

	Baseline 1, Small OoO processor	Baseline 2, Nehalem-like OoO	CRIB
Branch Predictions	Combined bimodal (16k entry) / gshare (16k entry) with a selector (16k), 32-entry RAS, 4-way 2k-entry BTB		
Out-of-order Execution	4-wide fetch/commit, 4-wide issue, 24 ROB, 16 IQ, 8 LQ, 4 SQ, 32 Int-PRF, 36 FP-PRF, 11-stage pipeline, speculative scheduling with squashing recovery, aggressive memory reordering with store set predictor (4k ssit, 128 lfst) and flush recovery, runahead on L2 miss	4-wide fetch/commit, 6-wide issue, 128 ROB, 36 IQ, 48 LQ, 32 SQ, 96 Int-PRF, 64 FP-PRF, 11-stage pipeline, speculative scheduling with squashing recovery, aggressive memory reordering with store set predictor (4k ssit, 128 lfst) and flush recovery, runahead on L2 miss	7 x 4-entry integer CRIB 7 x 2-entry FP CRIB aggressive memory reordering without predictor, light CRIB recovery runahead on L2 miss
Functional Units	2 integer ALU (1-cycle), 1 integer mult/div (4-cycle), 1 LD (1+2-cycle), 1 ST-addr (1-cycle),1 ST-data (1-cycle), 2 SIMD units (1-cycle), 2 FP add/mult (6-cycle), 1 FP div/square-root (12-cycle)	3 integer ALU (1-cycle), 1 integer mult/div (4-cycle), 1 LD(1+2-cycle), 1 ST addr (1-cycle),1 ST-data (1-cycle), 2 SIMD units (1-cycle), 2 FP add/mult (6-cycle), 1 FP div/square-root (12-cycle)	integer ALU per integer CRIB entry SIMD unit per FP CRIB entry 1 LD(1+2-cycle), 1 ST addr (1-cycle),1 ST-data (1-cycle), 2 FP add/mult (6-cycle), 1 FP div/square-root (12-cycle) **Banking**: 2 LD per cycle **Line Buffers**: 1-cycle on buffer hit
Memory System (latency)	L1 I-Cache: 64KB, 2-way, 64B line size (2-cycle) L1 D-Cache: 32KB, 4-way, 64B line size (2-cycle) L2 Unified: 2MB, 8-way, 128Bline size (12-cycle)	Off-chip memory: 168-cycle latency 32-entry prefetch buffer, stream prefetching on DL1 miss	

with tcbn 65nm library from TSMC is 0.90ns. Adding FF delay of 0.14 and FF setup time of 0.11, the baseline cycle time is 1.15 ns.

Table 3 shows the delay, energy, and area of CRIB. We categorize the combinational delay of our proposed CRIB entry into three categories: the input router, the ALU, and the output router as shown in Table 3. The delay of the input and output router is separated into the driver delay, 0.08 ns, and the tri-state delay, 0.07 ns for input router and 0.05 ns for output router. It is important to make this distinction as the driver delay only affects the delay when the instructions are first moved into the CRIB.

For CRIB with deep pipeline, the cycle time is determined using the following equations:
1) *cycle time= max (data latency, control latency)*

2) data latency = max (ALU lat., data propagation lat.)

3) control latency = max (ready lat., control propagation lat.)

The data and control propagation latency for equation (2) and (3) depends on how many CRIB entries the propagation spans. So to set the cycle time we choose the maximum of ALU latency and ready latency. Ready latency is calculated as an AND gate delay plus FF setup and delay time. However, since the datapath is not latched, the ALU latency is simply the ALU latency of 0.56 ns. Using a cycle time of 0.56 ns, the number of entries that the complete bit can propagate is determined to be four entries per the following calculation.

*0.56 = 0.25 (FF overhead) + x * 0.05 (output router delay) + 0.04 (trans FF) + 0.07 (input router delay)*

Thus, CRIB cycle time is set at 0.56 ns. The data and ready bit can propagate to the next four entries within a particular cycle. Using this cycle time, the CRIB is clocked at twice the rate of baseline machine.

For the integer CRIB, the input and output router has to route 21 registers (16 general purpose registers + 3 temporary registers + 1 flag + 1 PC). Correspondingly, the floating-point CRIB also has to route 18 registers (8 x87 registers + 1 control/status/tag/ + 8 XMM registers + 1 MXCSR), though these registers are 128 bits wide, rather than 64 bits for the integer side. Energy is calculated using the assumption that only three out of the 21 tri-states in the input routers are switching as each input router only reads one out of sixteen registers. A similar assumption is applied to the output router energy calculation. For clocked elements, we assume 136 bits of storage for integer CRIB consisting of 64 bits of PC, 20 bits of first input source tag, 20 bits of second input source tag / immediate values, 20 bits of destination tag, 10 bits of opcode, and 2 status bits. Because the PC is not necessary for the FP CRIB, 72 bits of storage are assumed. The energy listed is consumed when a a group of instructions is inserted into the CRIB, assuming a 50% switching factor.

We do not have detailed delay, energy, and area estimates for the MMX/SSE integer vector unit that is integrated into each CRIB entry. However, we believe that the delay should be approximately the same as the integer ALU because the short vector arithmetic operations do not have to propagate a full 64-bit carry. The area and energy is estimated as twice the integer unit because the XMM registers are 128 bits wide, vs. 64 bits in the integer ALU.

Table 2.Baseline IPC and Energy per Instruction (EPI)

Benchmark	IPC		EPI (pJ)	
	Baseline 1	Baseline 2	Baseline 1	Baseline 2
SPECINT2006				
astar	0.679	0.780	123.310	219.315
bzip2	0.960	1.410	111.381	148.962
gcc	0.432	0.565	196.080	316.246
libquantum	2.118	3.000	70.799	84.056
mcf	0.170	0.176	352.960	834.315
omnetpp	1.681	1.998	76.188	93.140
perlbench	0.804	1.127	120.274	173.517
xalancbmk	0.589	0.670	138.049	267.823
SPECFP2006				
cactusADM	0.541	0.558	113.744	129.953
calculix	1.356	1.603	80.584	103.154
dealII	0.933	1.080	88.372	102.312
GemsFDTD	0.639	0.651	102.882	115.674
lbm	0.385	0.435	164.361	192.945
leslie3d	0.618	0.624	100.382	116.619
povray	0.912	1.353	103.313	132.889
soplex	0.639	0.756	130.724	181.767

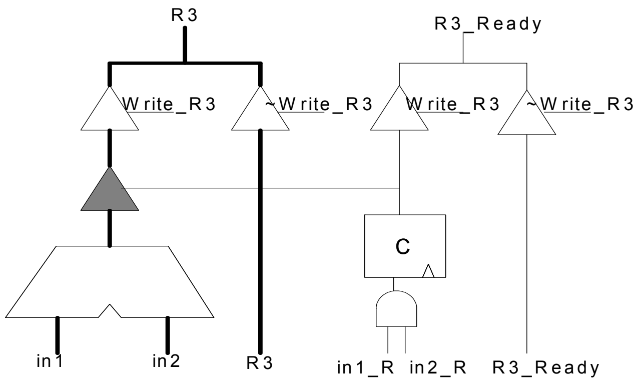


Figure 10. Output Router with Glitch Avoidance. Thick line represents 64-bit wires**.**

Area Estimates. We also consider the dimensions of the CRIB with 1312 (integer) and 1904 (FP) register wires laid out horizontally. We choose metal layer 4 for the register columns. Following the layout rule specified by the library, metal layer 4 requires a minimum width of 0.1 um and a spacing of 0.1 um. Thus, the total width of the register columns is 262.4 um and 380.8 um, which is less than the square-root of the area of a 4-entry integer and a 2-entry FP CRIB, which are 105732 um^2 and 192600 um^2. Hence, the CRIB area should not be wire-dominated, despite the seemingly large number of wiring channels that span it.

Wire Delay. While the delay calculation above already takes cell capacitance into account, it has not included the time for the signal to travel through the wire itself. The wire delay is calculated using a distributed RC model [30], *1/2RwireCwire*, for 380.8 um x 0.1 um metal layer 4 is 0.024 ps. This calculation is done with Rwire equal to 0.14 ohms per square distance and Cwire equal to 0.000232 pF per square distance. Thus, the wire delay can be considered negligible in the cycle time delay calculation. Note that though register columns appear to be long global wires, they are in fact short point-to-point links, as they are repeated at each CRIB entry.

ARF Area and Delay. The architected register file is implemented using transparent flip-flops to allow data and ready bit to propagate to the next CRIB. The bypass transparent flip-flop [6] is chosen due to its fast transparent delay of 0.04 ns. There are 1312 bits for integer ARF and 1904 bits for FP ARF. Each ARF has its shadow copy to support checkpointing for runahead. The energy for one register write assuming 50% duty cycle is 0.2 pJ. The transparent FF has a higher delay latency of 0.25 ns compared to a regular FF delay of 0.15ns. However, it does not affect the cycle time. Finally, an integer ARF with its shadow copy has an area of 0.023 mm^2, while the FP ARF has an area of 0.034 mm^2.

5. EXPERIMENTAL RESULTS

Partition size and count. We conducted extensive sensitivity studies to determine an attractive CRIB partition size and count. In the interests of space, we can only summarize our findings here. In terms of overall size, we were surprised to find that while some applications benefited from larger window sizes, the sweet spot for CRIB occurred at only 32 entries. We attribute this to a combination of runahead on L2 misses [33][34] effectively exposing memory level parallelism, as well as CRIB's relatively short pipeline and quick turnaround of CRIB partitions due to deep pipelining. We also evaluated smaller (2-entry) and larger (8-entry) partitions in various configurations. A smaller partition size has the advantage of faster CRIB turn around time, and is thus able to uncover distant independent instructions. However, a smaller partition size has higher ARF overhead, as ARF latches have to be inserted between each partition. Also, the propagation distance is shorter due to ARF delay. On the other hand, a bigger partition can propagate data further in the same cycle but has a slower CRIB turn around rate. Ultimately, we found that the 4-entry partition for integer CRIB and 2-entry partition for FP CRIB consistently provided the best performance. Hence, for the remainder of the results, we consider only a 7-partition CRIB configuration with 4-entry integer CRIB and 2-entry FP CRIB.

Table 3: Delay, Energy, and Area for CRIB

	Integer CRIB			Floating-Point CRIB		
	Delay (ns)	Energy (pJ)	Area (um^2)	Delay (ns)	Energy (pJ)	Area(um2)
Input Router	0.08 + 0.07	1.43	10240	0.11 + 0.07	2.01	18576
ALU	0.56	5.84	10214	0.56	11.68	20428
Output Router	0.08 + 0.50	0.75	4561	0.08 + 0.5	0.86	8395
Flip-Flops	0.14 + 0.11	0.30	1418	0.14 + 0.11	0.16	751

Energy and Area Results. We compare the area and energy usage between CRIB and two out-of-order baselines. For comparison purposes, we model the relevant structures in the both Baseline1 and Baseline2, such as register alias table, reservation station, payload RAM, ROB, register file, ALU with the bypass network, SQ, LQ, and DL1 cache. We model the integer CRIB, FP CRIB, architectural register file, SQ, LQ, and DL1 cache for the CRIB design.

Energy and area comparisons among these structures are shown in Table 4. The energy listed is energy per access while the area shown is the total area for each structure. For the base machine, the configuration in Table 1 is used to determine the dimensions of each structure. We assume that complex floating-point units remain the same without any modification so we do not model those structures.

The reorder buffer in the base machine has 96 bits per entry, consisting of PC, logical and physical destination identifier, load/store queue index, opcode, and various status bits [37]. It is modeled as a 4-wide buffer, thus only one write port is needed to insert four

Table 4: Energy and Area Comparison

	CRIB		Baseline1		Baseline2	
	Energy *pJ*	Area *mm^2*	Energy *pJ*	Area *mm^2*	Energy *pJ*	Area *mm^2*
Int RAT	-	-	3.29	0.058	3.29	0.058
FP RAT	-	-	3.25	0.062	3.25	0.062
ROB	-	-	9.16	0.100	11.83	0.302
RS	-	-	10.90	0.071	13.71	0.322
Immediate RAM	-	-	3.53	0.025	3.75	0.056
Int ARF vs Int PRF	0.02	0.147	8.04	0.151	9.82	0.282
FP ARF vs FP PRF	0.04	0.313	17.03	0.308	18.5	0.481
Int CRIB vs ALU+bypass	13.57	0.733	7.73	0.050	7.73	0.075
FP CRIB vs FP+bypass	20.73	0.680	12.63	0.026	12.63	0.026
Load Queue	4.06	0.023	2.32	0.015	13.33	0.204
Store Queue	2.03	0.012	1.16	0.008	10.56	0.093
DL1 Cache	42.39	0.564	51.06	0.575	51.06	0.575
Total Area	N/A	2.512	N/A	1.449	N/A	2.536

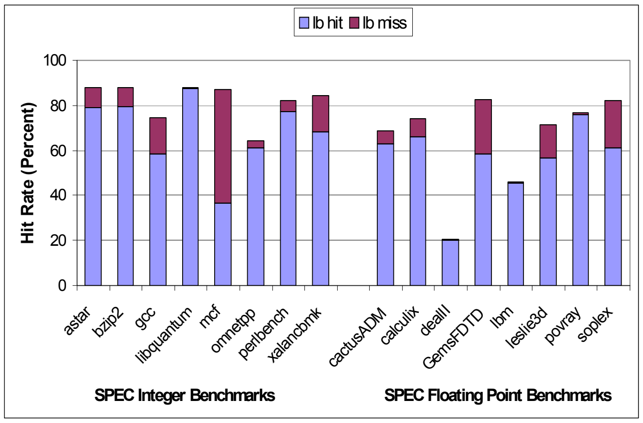


Figure 11. Line Buffer Hit Rate.

instructions into the window and one read port is needed to commit four instructions from the window. We assumed a port reduction technique [13][14], so the integer PRF is modeled with only four read ports rather than eight read ports (our timing model optimistically eliminates this constraint). ALU energy includes the energy for the ALU itself, 0.55 pJ, and a 2-level bypass network.

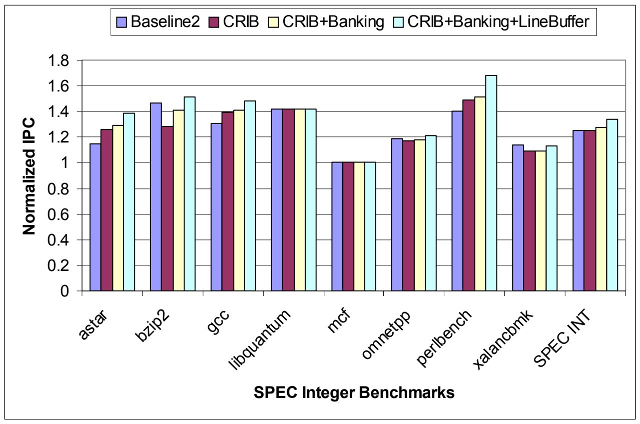
The ALU energy for CRIB includes the two input routers and eight output routers, assuming on average each destination result has a lifetime of eight instructions before being overwritten. The ARF in CRIB is modeled as flip-flops, without conventional register file ports, thus the energy per access is very small, only the energy to write 64 flip-flops.

Table 4 shows that CRIB has an area of 2.512 mm^2, comparable with 2.536 mm^2 area of Baseline2, while being 73% larger than the low-performance Baseline1. Table 1 also shows that several major structures are eliminated in CRIB. The area estimation can also be used to estimate the leakage power of the CRIB design, as it is largely proportional to area. Besides removing major structures, CRIB also removes many pipeline latches. We removed an estimated 1420 flip-flops, resulting in additional area savings of 0.017 mm^2 and energy saving of 3.935 pJ. per cycle. Further power reduction, which is hard to quantify, can be obtained from the simpler clock-tree needed for the CRIB design, which requires far fewer clock loads.

IPC and Total Energy Results. Figure 12 shows the IPC comparison between CRIB and baseline machines. IPC of CRIB and Baseline2 are normalized to Baseline1. The bars in each group show the IPC gained by different CRIB features. CRIB has two fundamental advantages: a shorter execution pipeline and the ability to aggressively disambiguate loads and stores due to its lightweight recovery. Performance benefit from a shorter execution pipeline has also been observed by Borch et al. [31]. Due to these benefits, CRIB's IPC exceeds Baseline1's, approaching Baseline2 IPC. Adding DL1 banking provides extra load bandwidth while adding line buffers provides latency reduction on buffer hit to 1-cycle. The additional bandwidth and latency reduction further increase the IPC.

The line buffers are surprisingly effective, returning hit rates exceeding 80% for most benchmarks, as shown in Figure 11. The hit rate is further categorized into 'lb hit' where the line buffer hit results in cache hit, providing 1-cycle cache access latency. The 'lb miss' category is where the line buffer contains the right set but the tag match fails; this results in early cache miss detection. Both cases result in energy saving since the sub-array does not need to be accessed.

We can see that for integer and FP benchmarks, CRIB's final IPC is quite comparable to Baseline2, although there are some outliers benchmarks that suffer 15%-20% slowdowns and speedups. On



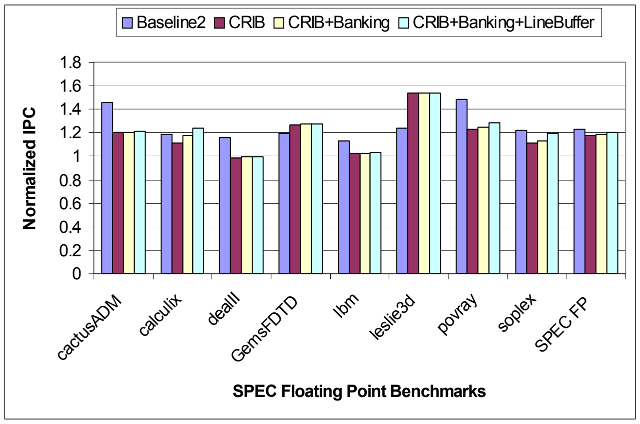
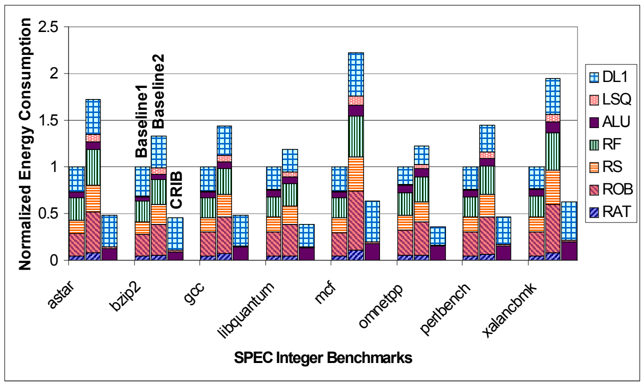


Figure 12. IPC Comparison for various CRIB optimizations.

average, CRIB's IPC for integer and FP benchmark are 6% higher and 3% lower than Baseline2.The gains are largely due to subtle pipeline effects in the processor models (e.g. faster recovery from squashes), higher memory bandwidth, and lower average memory latency. Losses are more prominent in FP benchmarks with the characteristic of a short chain of long latency instructions following independent loads. In these case, a bigger window like in Baseline2 is able to discover the next independent load and more efficiently pipeline the execution of the long latency instruction slice. Careful instruction scheduling in an optimizing compiler could be used to address this bottleneck, but this is left to future work.

Figure 13 and Figure 14 show energy consumption for Baseline1, Baseline2, and CRIB, normalized to Baseline1 energy consumption. On average, optimized CRIB consumes 40%-45% of Baseline1 energy and consumes 25%-30% of baseline 2 energy. Most of energy savings comes from removal of various structures such as RAT, ROB, RS, and RF. CRIB consumes up to twice the ALU energy of Baseline1 due to the fact that ALU energy consumption of CRIB includes input routing and output routing. While the number of cache accesses are roughly the same, CRIB's energy per access is slightly less due to the banking. Thus, the total cache energy of CRIB is slightly less than the total cache energy of both Baseline1 and Baseline2. The LSQ energy consumption of CRIB and Baseline1 are roughly the same, or about 15% of Baseline2 LSQ energy due to the difference in queue sizes. Banking helps reducing cache access energy.by 15%. The addition of line buffers helps significantly for some benchmarks by avoiding subarray access energy.

Summarizing the results, we find that CRIB, when augmented with deep pipelining, a banked cache, and line buffers, can easily match the per-cycle performance of an aggressive, current-generation out-



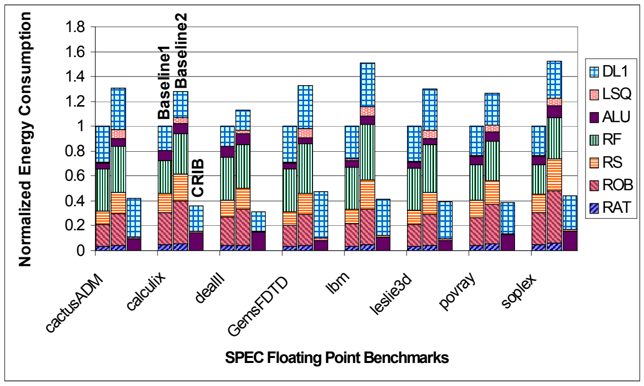
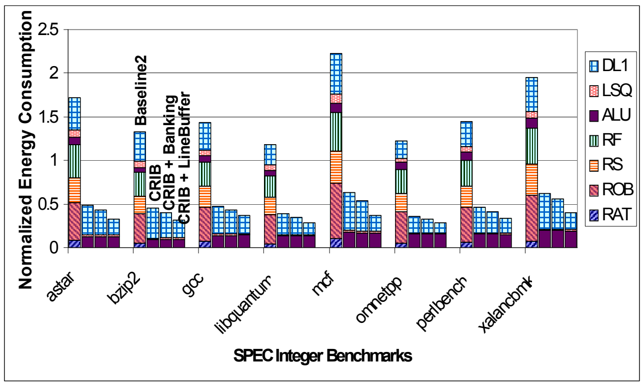


Figure 13. Energy Consumption Comparison. Energy is normalized to Baseline1.



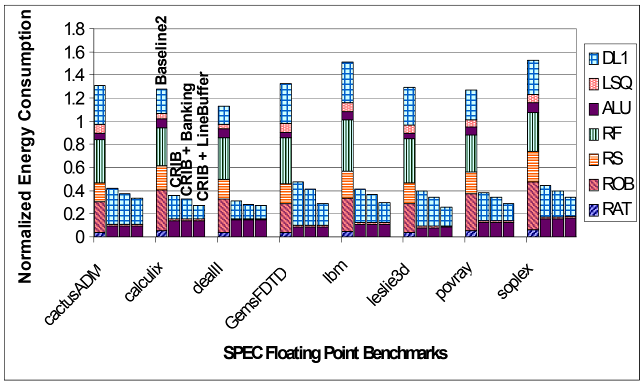


Figure 14. Energy Comparison for Various CRIB Optimizations. Energy is normalized to Baseline1**.**

of-order processor, while consuming only a fraction of the core power of the baseline machine. In-place execution reaps significant savings by eliminating redundant structures, accesses, and pipeline latches without compromising parallelism and performance.

6. RELATED WORK

In recent years, the concept of removing large centralized structures and replacing them with smaller distributed functional units has been explored in Raw, TRIPS, and Wavescalar. Raw [18][19] microprocessors are made up of simple tiles, each with a portion of the register set, a portion of the instruction and data cache, and one of the functional units. These tiles communicate via a pipelined two-dimensional interconnect mesh. Each tile contains a simple five-stage pipelined to maximize the clock rate and the number of tiles on chip. The hardware is fully exposed to the compiler to construct the schedule and spatial placement for instructions. The main difference between Raw microprocessors and our proposal is that Raw machines still depend on register renaming algorithm to resolve data dependences. It compensates for the scalability issue of the register file by distributing it across many tiles, creating non-uniform register access latencies (NURA).

In a similar spirit, TRIPS [20][21] uses an array of ALUs, each with limited control, connected by a scalar operand network. Each array consists of instruction and operand buffer, a functional unit, and router for input to and output from the array. With support from the compiler, programs are divided into groups of instructions. Each group occupies one array. Most data communication is done directly from producer to a consumer without involving the register file, thus reducing the read and write bandwidth to the register file significantly. The physical locations of consumers are explicitly encoded within producer instructions, thus no broadcast is necessary. The producer simply deliver its result to the listed consumers. To build the data-flow like execution, support from the compiler is required. Besides not requiring compiler support, one major difference between TRIPS and CRIB is how the data is passed from a producer to a consumer. In TRIPS, a producer has to deliver its result to consumers. while in our scheme, a producer only needs to write to its own destination register path.

Wavescalar [22] goes further to build its decentralized data-flow processors by embedding processing elements (PE) into the cache, called the WaveCache. The WaveCache is a grid of PEs arranged into clusters. Each PE contains logic to control instruction placement and execution, input and output queues for instructions operands, communication logic, and a functional unit. The goal of WaveScalar is to break the serialization point of the von Neumann model, namely the program counter, and to guarantee load-store ordering. By distributing the processing elements, WaveScalar also eliminates all the large hardware structures that make superscalars non-scalable.

In the vein of making the register file more efficient, a vast amount of prior work has also been done. These differ from our approach, which eliminates the physical register file completely due to the removal of renaming. Instead, these related works focus more on a better renaming and allocation algorithms. As the PRF is considered one of design bottlenecks in current microprocessors, these works try to relax the physical register file constraint with a better renaming algorithm. One of the earliest work in this area is the alternative approach to register renaming [23], which releases physical register early without waiting for the next write of the architected register to commit. The register is deallocated as soon as the register has been unmapped and all reads of the register are complete. This scheme is implemented by adding an unmap flag, a counter, and a complete flag for each physical register. Cherry [24] and CPR [25] uses a similar approach combined with checkpointing schemes to enable early release of physical registers and other resources. Physical register inlining [26] releases physical registers containing narrow-width value and inlines them in the rename table. A scheme to exploit value temporal locality through physical register reuse is proposed in [27]. Virtual physical registers [28] proposes a late allocation of physical register by assigning a virtual one during rename, thus reducing the effective register lifetime. Kilo processors [29] combines checkpointing, late allocation, and early release mechanism to create PRF that can support thousands of in-flight instructions.

A description of an earlier version of the CRIB design, along with many additional design details and an in-depth performance analysis, are available to the interested reader in [40].

7. CONCLUSIONS

This paper set out to design the simplest possible execution core that still fully exposes ILP, and achieved that objective with a novel spatially-oriented design that directly overlays data dependences on the execution hardware. The proposed CRIB processor eliminates redundant dependence-resolution activity by consolidating the RAT, the RS, and the ROB are into one CRIB structure. Explicit register renaming is replaced with local renaming within the CRIB. A large multiported physical register file is replaced by a small, spatially-organized architected register file which consists of a simple rank of latches. Data forwarding between instructions in the CRIB is accomplished without latches while keeping the design fully synchronous by latching only the control bits. Scheduling is local, so variable-latency optimizations like cache banking and line buffers are feasible and attractive. The CRIB execution pipeline is considerably shorter and faster, enabling a much smaller and more power-efficient instruction window to reap most of the available instruction-level parallelism, while memory-level parallelism is uncovered via runahead execution [33][34]. Finally, the core is amenable to power-efficient deep pipelining, since only control paths are latched, eliminating the power and area overhead of wide data path latches.

As we show through an extensive design analysis, this novel microarchitecture delivers competitive cycle time and performance per clock cycle while requiring 60% less energy than a conventional out-of-order design for the affected structures. Performance matches an aggressive Nehalem-like out-of-order baseline for integer and FP benchmarks. Finally, we achieve this with comparable core area with conventional out-of-order design.

In future work, we plan to explore improvements to floating-point performance, in particular changes to code generation and optimization that could alleviate the long-dependence-chain issues that inhibited ILP extraction for some benchmarks like *povray*. Also, the CRIB design better distributes activity across the entire core area, avoiding hot spots. We plan to evaluate this benefit in terms of its impact on leakage power and device aging, both of which are exacerbated by high temperatures.

ACKNOWLEDGEMENTS

This work was supported in part by NSF award CCF-0702272, financial support from IBM, and equipment donations from HP.

REFERENCES

[1] L.Gwenapp. Intel's P6 uses decoupled superscalar design. In *Microprocessor Report*, pages 9-15, 1995.

[2] J.M. Tendler et al. Power4 System Microarchitecture. *IBM Journal of Research and Development*, vol. 46, no. 1, 2002.

[3] R.E. Kessler et al. The Alpha 21264 Microprocessor Architecture. In *ICCD*, 1998.

[4] G. Hinton et al. The Microarchitectgure of the Pentium 4 Processor. *Intel Technology Journal Q1*, 2001.

[5] S. Palacharla, et al. Complexity-Effective Superscalar Processors. In *ISCA-24*, 1997.

[6] E.L.Hill and M.H. Lipasti. Transparent Mode Flip-Flops for Collapsible Pipelines. In I*CCD-25*, 2007.

[7] H.M. Jacobson. Improved clock-gating through transparent pipelining. In *ISLPED*, August 2004

[8] S. Thoziyoor, N. Muralimahonar, and N.P. Jouppi. CACTI 5.0. *HPL-2007-167*, 2007.

[9] Bochs: the Cross Platform IA-32 Emulator, http://bochs.sourceforce.net.

[10] G.Chrysos and J.Emer. Memory Dependence Prediction using Store Sets. In *ISCA-25*, 1998.

[11] D.Brooks et al. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. In *ISCA-27*, 2000.

[12] A.Phansalkar, A. Joshi, and L.K. John. Analysis of Redundancy and Application Balance in the SPEC CPU2006 Benchmark Suite. In *ISCA-34*, 2007.

[13] I. Park, et.al., Reducing Register Ports for Higher Speed and Lower Energy. In *MICRO-35*, 2002.

[14] I. Kim and M.H. Lipasti, Half-Price Architecture. In *ISCA-30*, 2003.

[15] H. Patil, et al. Pinpointing representative portions of large Intel Itanium programs with dynamic instrumentation. In *MICRO-37*, 2004.

[16] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *PACT*, 2001.

[17] T. Sherwood, et.al. Automatically characterizing large scale program behavior. In *ASPLOS-10*, 2002.

[18] E. Waingold, et al. Baring It All to Software: Raw Machines. *Computer*, pages 86-93, September 2007.

[19] W. Lee et al. Space-time scheduling of instruction-level parallelism on a Raw machine. In *ASPLOS-6*, 1998.

[20] R. Nagarajan et al. A design space evaluation of grid processor architectures. In *MICRO-34*, 2001.

[21] K. Sankaralingam et al, Exploiting ILP, TLP, and DLP with the polymorphous trips architecture. In *ISCA-30*, 2003.

[22] S. Swanson et al. Wavescalar. In *MICRO-36*, 2003.

[23] M. Moudgill et al. Register renaming and Dynamic Speculation: an Alternative Approach. In *MICRO-26*, 1993.

[24] J. Martinez et al. Cherry: Checkpointed early resource recycling in out-of-order microprocessors. In *MICRO-35*, 2002.

[25] H. Akkary et al. Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors. In *MICRO-36*, 2003

[26] M. Lipasti et al. Physical Register Inlining. In *ISCA-31*, 2004.

[27] S. Jourdan et al. A novel renaming scheme to exploit value temporal locality through physical register reuse and unification. In *MICRO*, 1998.

[28] A. Gonzalez et al. Virtual-physical registers. In *HPCA-4*, 1998.

[29] A. Cristal et al. Kilo-Instruction Processors: Overcoming the Memory Wall. In *IEEE Micro*, vol. 25, no. 3, May/June 2005.

[30] R. Ho, K.W. Mai, M.A. Horowitz, The Future of Wires. In *Proceedings of the IEEE, vol. 89, no. 4*, April 2001, pp. 490 - 504.

[31] E. Borch et al., Loose Loop Sink Chips. In *HPCA-8*, 2002.

[32] P. Kongetira, K. Aingaran and K. Olukotun, Niagara: A 32-Way Multithreaded SPARC Processor. In *IEEE Micro*, vol. 25, no. 2, 2005.

[33] J. Dundas and T. Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *International Conference on Supercomputing,* pages 68-75, 1997

[34] O. Mutlu, et.al., Techniques for Efficient Processing in Runahead Execution Engines. In *ISCA*, 2005.

[35] J.A. Kahle, et al., Introduction to the Cell multiprocessor. *IBM Journal of Research and Dev't*, vol. 49, no. 4/5, 2005.

[36] Anand Lal Shimpi, Nehalem - Everything You Need to Know about Intel's New Architecture. In *http://www.anandtech.com/cpuchipsets/intel/showdoc.aspx?i=3382.*

[37] ACS Simulation Tools About IVM version 1.0. In *http://www.crhc.illinois.edu/ACS/tools/ivm/about.html*

[38] D. Henry et al., "The Ultrascalar Processor-an asymptotically scalable superscalar microarchitecture," in ARVLSI 1999.

[39] C-L. Su and A.M. Despain, "Cache designs for energy efficiency," in HICSS-28, p. 306-, 1995.

[40] E. Gunadi, "CRIB: Consolidated Rename, Issue, and Bypass," Ph.D. Thesis, Univ. of WI-Madison, May 2010.