Computer Science and Artificial Intelligence Laboratory
Technical Report

MIT-CSAIL-TR-2006-066 September 18, 2006

RingScalar: A Complexity-Effective Out-of-Order Superscalar Microarchitecture

Jessica H. Tseng and Krste Asanovic

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

RingScalar: A Complexity-Effective Out-of-Order Superscalar Microarchitecture

Jessica H. Tseng and Krste Asanović
MIT Computer Science and Artificial Intelligence Laboratory
32 Vassar Street, Cambridge, MA 02139
{jhtseng,krste}@csail.mit.edu

Abstract

RingScalar is a complexity-effective microarchitecture for out-of-order superscalar processors, that reduces the area, latency, and power of all major structures in the instruction flow. The design divides an N-way superscalar into N columns connected in a unidirectional ring, where each column contains a portion of the instruction window, a bank of the register file, and an ALU. The design exploits the fact that most decoded instructions are waiting on just one operand to use only a single tag per issue window entry, and to restrict instruction wakeup and value bypass to only communicate with the neighboring column. Detailed simulations of four-issue single-threaded machines running SPECint2000 show that RingScalar has IPC only 13% lower than an idealized superscalar, while providing large reductions in area, power, and circuit latency.

1 Introduction

Early research in decentralized or clustered architectures [18, 8, 7, 16] was motivated by the desire to build wider superscalar architectures (8-way and greater) while maintaining high clock frequencies in an era where wire delay was becoming problematic [10]. Clustered architectures divide a wide-issue microarchitecture into disjoint clusters each containing local issue windows, register files, and functional units. Because each cluster is much smaller and simpler than a monolithic superscalar design, the circuit latency of any path within a cluster is significantly lower, hence allowing greater clock rates than a centralized design of the same total issue width. However, any communication across clusters incurs greater latency, and so a critical issue is the scheme used to map instructions to clusters.

Ranganathan and Franklin [14] grouped decentralized clustering schemes into three categories. Execution unit based dependence schemes (EDD) map instructions to clusters according to the types of instructions (e.g., floating-point versus integer). Control Dependence based schemes (CDD), such as Multiscalar [18], and Trace Processors [16], map instructions that are contiguous in program order to the same cluster. Data Dependence based schemes (DDD), such as PEWS [8] and Multicluster [7], try to map data dependent instructions to the same cluster.

EDD schemes are widely used, dating back to early out-of-order designs [19], and can also be employed locally within clusters of other schemes. However, EDD schemes do not scale well to large issue widths [14]. DDD schemes were found to be more effective than CDD schemes except for the largest issue widths, but both these schemes incur significant area and complexity overheads for small increases in total IPC [14].

In recent years, two trends have significantly changed processor design optimization targets. First, power dissipation is now a primary design constraint, and designs must be evaluated based on both performance and power. Second, the advent of chip-scale multiprocessors has placed greater emphasis on processor core area, as total chip throughput can also be increased by exploiting thread-level parallelism across multiple cores. We believe these trends favor techniques that reduce the complexity of moderate issue-width cores, rather than techniques that use large area and complex control to increase single-thread performance.

In this paper, we introduce "RingScalar", a new centralized out-of-order superscalar microarchitecture that uses banking to increase area and power efficiency of all the major components in the instruction flow without adding significant pipeline control complexity. RingScalar builds an N-way superscalar from N columns, connected in a unidirectional ring. Each column contains a bank of the issue window, a bank of the physical register file, and an ALU. Communication throughout the microarchitecture is engineered to reduce the number of ports required for each type of access to each bank within each structure. The restricted ring topology reduces electrical loading on latency-critical communications between columns, such as instruction wakeup and value bypassing. We exploit the fact that most decoded instructions are waiting on only one operand to use just a single source tag in each issue window entry, and dispatch instructions to columns according to data dependencies to reduce the performance impact of the restricted communication. Detailed simulations of the SPECint2000 benchmarks on four-issue machines show that a RingScalar design has an average IPC only 13% lower than an idealized superscalar, while having much reduced area, power, and circuit latency.

2 RingScalar Microarchitecture

The RingScalar design builds upon earlier work in banked register files [23, 4, 2, 11, 20], tag-elimination [5, 9], and dependence-based scheduling [8, 10]. For clarity, this section describes RingScalar in comparison to a conventional superscalar. Detailed comparison with previous work is deferred to Section 5.

2.1 Baseline Pipeline Overview

RingScalar uses the same general structure as the MIPS R10K [24] and Alpha 21264 [12] processors, with a unified physical register file containing both speculative and committed state. The following briefly summarizes the operation of this style of out-of-order superscalar processor, which we also use as a baseline against which to compare the RingScalar design.

Instructions are *fetched* and *decoded* in program order. During decode, each instruction attempts to allocate resources including: an entry in the reorder buffer to support in-order commit; a free physical register to hold the instruction's result value, if any; an entry in the issue window; and an entry in the memory queue, if this is a memory instruction. If some required resource is not available, decode stalls. Otherwise, the architectural register operands of the instruction are *renamed* to point to physical registers, and the source operands are checked to see if they are already available or if the instruction must wait on the operands in the issue window. The instruction is then *dispatched* to the issue window, with a tag for each source operand to hold its physical register number and readiness.

Execution occurs out-of-order from the issue window, driven by data availability. As earlier instructions execute, they broadcast their result tag across the issue window to *wake up* instructions with matching source tags. An instruction becomes a candidate for execution when all of its source operands are ready. A *select* circuit picks some subset of the ready instructions for execution on the available functional units. Once instructions have been selected for *issue*, they read operands from the physical register file and/or the bypass network and proceed to *execute* on the functional units. When instructions *complete* execution, they write values to the physical register file and write exception status to the reorder buffer entry. When it is known an instruction will complete successfully, its issue window entry can be freed.

To preserve the illusion of sequential program execution, instructions are *committed* from the reorder buffer in program order. If the next instruction to commit recorded an exception in the reorder buffer, the machine pipeline is flushed and execution continues at the exception handler. As instructions commit, they free any remaining machine resources (physical register, reorder buffer entry, memory queue entry) for use by new instructions entering decode.

Memory instructions require several additional steps in execution. During decode, an entry is allocated in the memory queue in program order. Memory instructions are split into address calculation and data movement sub-instructions. Store address and store data sub-instructions issue independently from the window, writing to the memory queue on completion. Store instructions only update the cache when they commit, using address and data values from the memory queue. Load instructions are handled as a single load address calculation in the issue window. On issue, loads calculate the effective address then check for address dependencies on earlier stores buffered in the memory queue. Depending on the memory speculation policy (discussed below), the load will attempt to proceed using speculative data obtained either from the cache or from earlier store instructions in the memory queue (the load may later require re-execution if an address mis-speculation or a violation of memory consistency is detected). If the load cannot proceed due to an unresolvable address or data dependency, it waits in the memory queue to reissue when the dependency is resolved. Loads reissuing from the memory queue are given priority for data access over newly issued loads entering the memory queue.

2.2 RingScalar Overview

RingScalar retains this overall instruction flow and uses the same reorder buffer and memory queue, but drastically reduces the circuitry required in the issue window, register file, and bypass network by restricting global communication within these structures. These restrictions exploit the fact that most instructions enter the issue window waiting on one or zero operands.

The overall structure of the RingScalar microarchitecture is shown in Figure 1. RingScalar divides an N-way issue machine into N columns connected in a unidirectional ring. Each column contains a portion of the issue window, a portion of the physical register file, and an ALU. Physical registers are divided equally among the columns, and any instruction that writes a given physical register must be dispatched and issued in the column holding the physical register. This restriction means each bank of the regfile needs only a single write port directly connected to the output of the ALU in that column.

A second restriction is that any instruction entering the window while waiting for an operand must be dispatched to the column immediately to the right of the column containing the producer of the value (the leftmost column is considered to be to the right of the rightmost column in the ring). This restriction has two major impacts. First, when an instruction executes, it need only broadcast its tag to the neighboring column which reduces the fanout on the tag wakeup broadcast by a factor of N compared to a conventional window. Second, the bypass network can be reduced to a simple ring connection between ALUs as any other value an instruction needs should be available from the register file. The bypass fanout is reduced by a factor of N, and each ALU output now only has to drive the regfile write port and the two inputs of the following ALU.

These restrictions are key to the microarchitectural savings in RingScalar, and as shown in the evaluation, have a

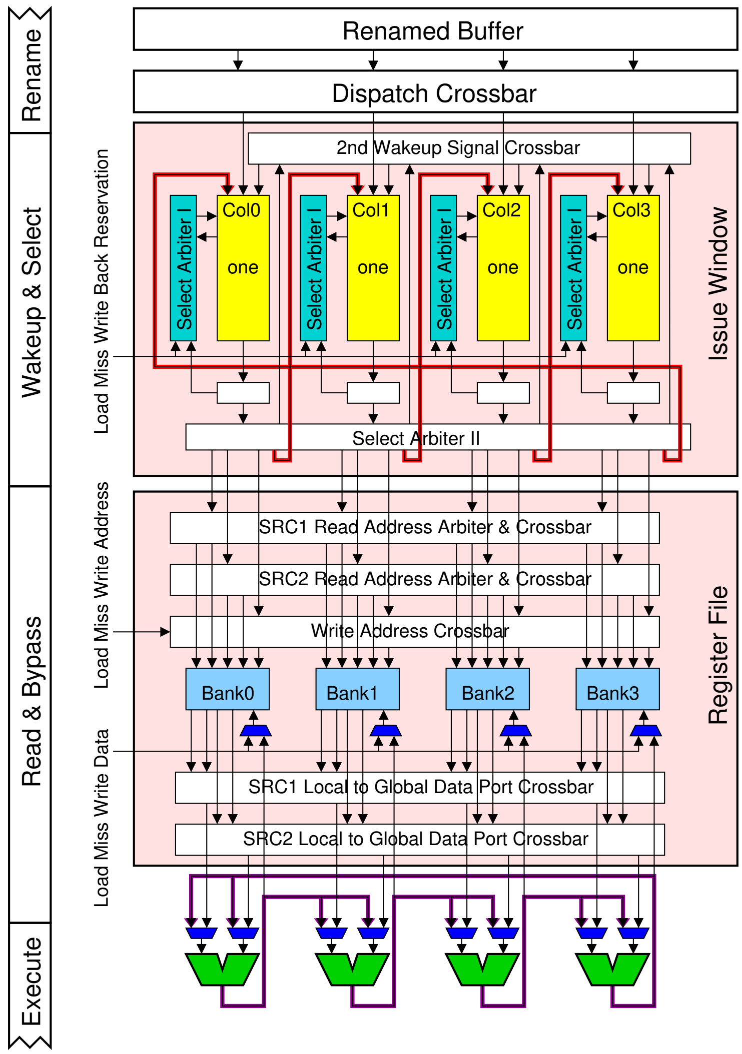

Figure 1. RingScalar core microarchitecture for a four-issue machine. The reorder buffer and the memory queue are not shown.

relatively small impact on instructions per cycle (IPC) while reducing area, power, and circuit latency significantly. The following subsections describe the major components of the machine in more detail.

2.3 RingScalar Register Renaming

The RingScalar design trades a little added complexity in the rename and dispatch stage (and some overall IPC degradation) to reduce the area, power, and latency of the remaining stages (issue window, regfile, bypass network). Figure 2 shows an example of the RingScalar renaming and dispatch process. We used the Alpha ISA in our experiments, where instructions can have zero, one, or two source register operands.

As RingScalar decodes each instruction, the source architectural registers are renamed into the appropriate physical registers and the readiness of these operands is checked, just as in a conventional superscalar processor. Instructions that are not waiting on any operand (*zero-waiting*) can be dispatched to any column in the window. Instructions that are waiting on one operand (*one-waiting*) must be dispatched to the column to the right of the producer column. Instructions waiting on two operands (*two-waiting*) are split into two parts that will issue sequentially, and each part must be dispatched to the column to the right of the appropriate producer column. A separate 2nd wakeup port is provided on each column to enable the first part of an instruction to wakeup the second part regardless of the column in which it resides. The second part sits in the window, but will not request issue until after the first part has issued and woken up the second part. The second part can be woken up one cycle after the first part. Previous work has considered the use of prediction to determine which operand will arrive last [5],

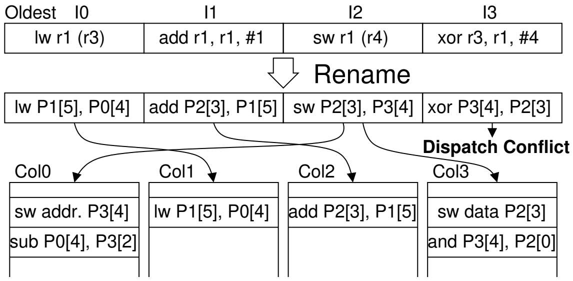

Figure 2. RingScalar rename and dispatch. The `sub` and `and` instructions were already in the window before the new dispatch group.

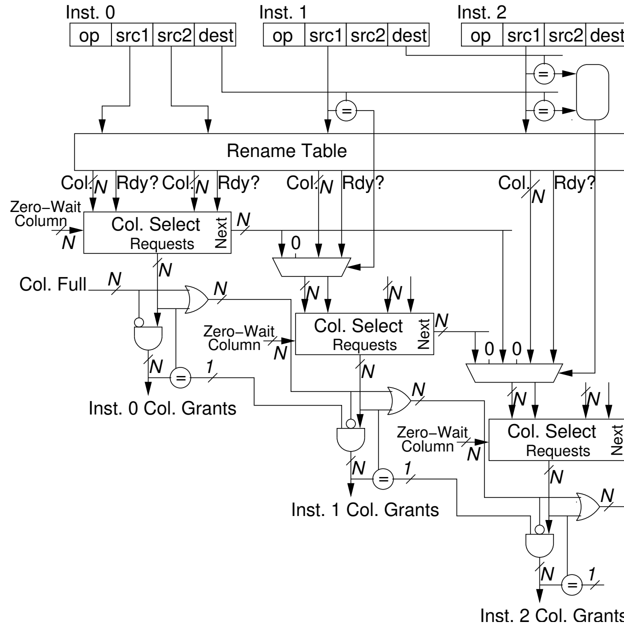

Figure 3. RingScalar register renaming and column dispatch circuitry. Only the circuitry for `src1` of instruction 1 and 2 is shown.

but in this work, we adopt a simple heuristic that assumes the first (left) source operand will arrive last for a two-waiting instruction. Store instructions are handled specially by splitting them into two parts (address and data) that can issue independently and in parallel.

To reduce complexity in both the dispatch crossbar and the issue window entries, there is only a single dispatch port for each column in the window. The two parts of a store or two-waiting instruction occupy two separate dispatch ports if they go to different columns, but can be dispatched together in one cycle to the same column. Note that a physical register tag is now two fields: the window column and the register within the column.

The RingScalar renamer has some flexibility in how any zero-waiting instructions (and any dependents) are mapped to columns. To reduce the complexity of the rename logic, we adopt a simple greedy scheme where instructions are considered in program order. Zero-waiting instructions select a dispatch column using a random permutation that changes each cycle. One-waiting and two-waiting instructions have no freedom and must be allocated as described above. When the next instruction cannot be dispatched because a required dispatch port is busy, dispatch stalls until the next cycle.

Figure 3 shows the renaming and column dispatch circuitry in detail. As with a conventional superscalar, the first step is a rename table lookup to find the current physical register holding each architectural register source operand together with its readiness, while, in parallel, the architectural source registers of later instructions in the group are checked for dependencies on the architectural destination registers of earlier instructions.

Each rename table lookup returns the column (*Col*) in which the physical register resides and a single bit (*Rdy?*) indicating if the value is ready or not, in addition to the physical register number. Below the rename table in Figure 3, we show only the circuitry responsible for column allocation and do not show the physical register number within the column. Column information is represented using a unary format with one bit per window column (i.e., *Col.* is an N-bit vector) to simplify circuitry.

For each instruction, the *Col. Select* circuitry calculates two N-bit column vectors: *Requests* and *Next*. The *Requests* vector has one or two bits set indicating which columns the instruction wants to dispatch into, and is calculated in two steps. First, if both of the operands are ready, an internal vector is set to the precomputed *Zero-Wait Column* vector which has a single bit pointing at the randomly assigned column for this instruction. If at least one of the operands is not ready, the internal vector is set to the bitwise-OR of the two input *Col* vectors. The internal vector is then rotated by one bit position to yield the *Requests* vector. The rotation is simple rewiring and so has no additional logic delay.

The *Next* output has a single bit set indicating the column into which this instruction will write its final result. First, the internal vector is assigned either *Zero-Wait Column* if both operands are ready, *Col* for the second source only if the instruction is one-waiting on the second operand, or otherwise *Col* for the first source (i.e., one-waiting on first operand or two-waiting). Second, the internal vector is rotated by one bit position to obtain the *Next* vector.

Any later instruction in the current dispatch group that has a RAW dependency on an earlier instruction in the group must mux in the *Next* vector from the earlier instruction in place of the stale column vector read from the rename table. In the worst case, a series of serially-dependent instructions requires the *Next* values to ripple across the different *Col. Select* blocks in the dispatch group, as shown in Figure 3. Fortunately, mux select lines are available early, and the ripple path always carries non-ready operands (*Rdy?*$=0$), which reduces worst-case latency to a few gate delays per instruction.

The dispatch column arbiter is implemented using the serial logic gates shown at the bottom of Figure 3. The leftmost input to the arbiter chain is a *Col. Full* vector indicating which columns cannot accept a new instruction, either because the issue window is full or because there are no free physical registers left in the column. The arbiter ensures that instructions must dispatch in program order, by preventing later instructions from dispatching if an earlier one did not get all requested columns (this is the purpose of the equality comparators). The ripple through the arbiter is in parallel with the slower ripple through the *Col. Select* blocks, so adds only a single gate delay before yielding the column grant signals.

The additional column latency in RingScalar is compensated by the reduced dispatch latency, as each dispatch port fans out to N times fewer entries than in a conventional superscalar, and each entry has one port rather than N.

Note that the column allocation circuity is a small amount of logic (dozens of gates on each of the N bit slices) and represents a very small power and area overhead compared to the savings in issue window and register file size.

The final step of renaming is to allocate a destination physical register for the instruction if required (stores and branches do not require destination registers, and neither does the first part of a two-waiting instruction). Each column has a separate physical register free list, and so any instruction that is dispatched to a column simply takes the head of the relevant free list.

2.4 Issue Window

The RingScalar issue window has several complexity reductions compared to a conventional superscalar. The primary savings come from the reduced port count in each column. A conventional superscalar window has N dispatch ports, $2N$ wakeup ports, and N issue ports on each entry. RingScalar has only a single dispatch port, two narrower wakeup ports, and one issue port.

Each column needs only two wakeup ports. The first wakeup port is used by the preceding column to wake up dependent instructions, while the second port wakes up the second part of a two-waiting instruction. Both of these ports are narrower than in a conventional superscalar as shown in Figure 4. The physical register tag requires $\lg(N)$ fewer bits because the consumer must be waiting for a value located in the preceding column. The second-part tag can be considerably narrower as it only needs to distinguish between multiple second parts mapped to the same column in the issue window.

The RingScalar instruction window design significantly reduces the critical wakeup-select scheduling loop. Wakeup has reduced latency because an instruction only has to broadcast its tag to one column, each entry has only one comparator, and each tag is narrower. A conventional design requires

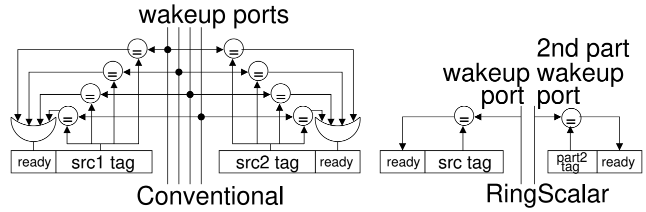

Figure 4. Wakeup circuitry.

the tag be driven across the entire window, with each entry having two comparators, leading to a $2N$ times greater fanout in total. The select arbiter also has considerably reduced latency, as each column has a separate arbiter, and each column can only issue at most one instruction. The conventional design has an arbiter with N times more inputs and N times more outputs.

Each entry only has a single issue port, which reduces electrical loading to read out instruction information after select. A conventional design has each entry connected to N issue ports, each with N times greater fanout.

The combination of a single dispatch port and a single issue port makes it particularly straightforward to implement a compacting instruction queue [3], where each column in the window holds a stack of instructions ordered by age with the oldest at the bottom. The fixed-priority select arbiter picks the oldest ready instruction for issue. To compact out a completed instruction from the window, each entry below the hole retains its value, each entry at or above the hole copies the entry immediately above to squeeze out the hole, while the previous highest entry copies from the dispatch port if there's a new dispatch (Figure 5). The age-ordered window columns also simplify pipeline cleanup after exceptions or branch mispredictions.

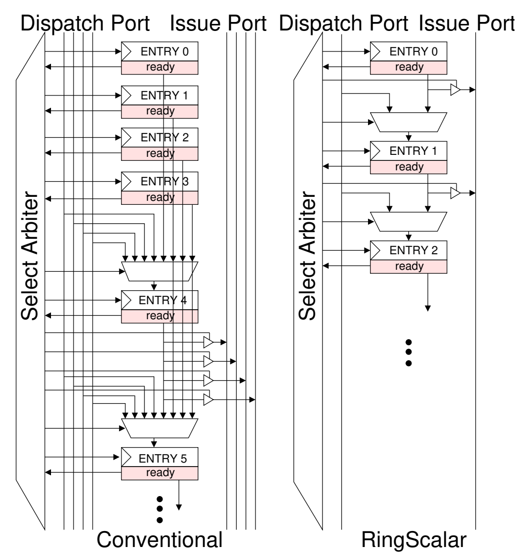

Figure 5. Compacting instruction queues.

In practice, instruction entries are not compacted out right after issue, but only after it is known they will complete successfully. In particular, dependent operations are scheduled assuming loads will hit in the cache and must remain in the window until the cache access is validated in case a miss requires the dependent instruction be replayed. As described in the following section, RingScalar uses the same technique if a banked register file with read conflicts is used.

Instructions are latched after issue, then undergo a second stage of select arbitration (*Select Arbiter II* in Figure 1) that is used to resolve structural hazards across columns. For example, our evaluation machine only allows a single load to issue per cycle. We also allow only a single first-part sub-instruction to wake-up a second-part sub-instruction across the *2nd Wakeup Signal Crossbar* each cycle. Instructions failing the second stage of arbitration remain in the issue latch and block further issue in the column until the structural hazard is resolved.

2.5 Register File

One of the greatest savings in the RingScalar design comes from the reduction in the number of write ports required on the register file. Each column has a separate physical register bank, which needs only a single write port.

We allow any column to read data from any register bank in the machine. In the simplest RingScalar design, we provide a full complement of read ports ($2N$) on every bank. To further reduce regfile power and area, we can reduce the number of read ports per bank and use the speculative read conflict resolution strategy previously published in [20]. For example, the four-issue machine shown in Figure 1 has four read ports and one write port per bank whereas a conventional machine would have eight read ports and four write ports. A local to global read port crossbar is required to allow any functional unit to read any register from any local bank's read port

The reduced-read-port design adds an additional arbitration stage to the pipeline as shown in Figure 6. Instructions continue issuing assuming there will be no conflicts. When read-port conflicts are detected, the instruction window must be repaired and execution must be replayed [20]. The number of read bank conflicts is reduced by not requesting read port accesses when a value was produced in the preceding cycle and hence will be available on the bypass ring (*conservative bypass-skip* [20]), and by implementing the *read-sharing* optimization [2], which allows a single bank port to send the same register to multiple requesters over the global ports.

Unlike the previous design [20], there is no need for a global write port network and an arbiter with bank conflict detection, as RingScalar associates one column with each write port and issues at most one instruction per column. This is a considerable saving, as it was also previously found that more than one write port per bank was required to reduce write port conflicts to an acceptable level [21].

Variable latency instructions, such as cache misses, also require access to the register file write ports to return their results. To avoid conflicts, a returning cache miss inserts a high priority request into the select arbiter for the target column, preventing another instruction from issuing while the cache miss uses the write port.

2.6 Bypass Network

RingScalar also provides a large reduction in bypass network complexity. An ALU can only bypass to its neighbor around the ring, not even to its own inputs. This bypass path is sufficient because the dependence-based rename and dispatch ensures dependent instructions are located immediately following the producer in the ring. If an operand was ready when the instruction was dispatched, it would have been obtained from the register file in any case. If a dependent instruction does not issue right after the producer, it must wait until the value is available in the regfile before issuing.

3 Evaluation

To characterize the behavior of RingScalar, we extensively modified SMTSIM [22], a cycle-accurate simulator that models an out-of-order superscalar processor with simultaneous multithreading ability. These modifications included changes to the Rename, Dispatch, Select, Issue, Regfile Read, and Bypass stages of the processor pipeline. A register renaming table that maps architectural registers to physical registers is added to monitor the regfile access from each instruction. To keep track of a unified physical register file organized into banks, extra arbitration logic is added to each regfile bank to prevent over-subscription of read and write ports when a lesser-ported storage cell is used. Since load misses are timing critical, a write-port reservation queue is also added to give them priority over other instructions. Additional changes are made to the register renaming policy, dispatch logic, wakeup-select loop, and issue logic, to model the RingScalar design.

This paper focuses on evaluating the performance of RingScalar within integer pipelines, so we choose the SPEC CINT2000 benchmark suite for its wide range of applications taken from a variety of workloads. The suite has long run times but expected performance can be well characterized without running to completion. To reduce the simulation run time to a reasonable length, the methodology described in [17] is used to fast-forward execution to a sample of half a billion instructions for each application. The benchmarks are compiled with optimization for the Alpha instruction set.

Given the design target for this microarchitecture, we compare RingScalar against idealized models of 4-issue centralized superscalars. Table 1 shows parameters common across the machines compared. We used a large reorder buffer of 256 entries and a large memory queue of 64 entries

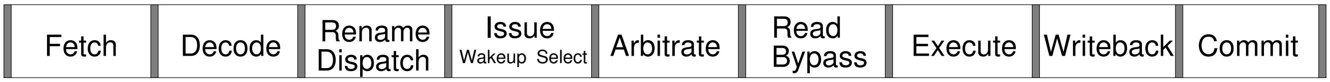

Figure 6. RingScalar pipeline structure.

L1 I-cache	16KB 4-way, 64-byte lines, 1 cycle
L1 D-cache	16KB 4-way, 64-byte lines, 1 cycle
L2 unified cache	1MB 4-way, 64-byte lines, 12 cycles
L3 unified cache	8MB 8-way, 64-byte lines, 25 cycles
Fetch width	8
Dispatch, issue, and commit width	4
Integer ALUs	4
Memory instructions	2 (1-Load and 1-Store)
Reorder Buffer	256 entries
Memory Queue	64 entries
Branch predictor	gshare 4K 2-bit counters, 12-bit history

Table 1. Common simulation parameters.

such that these would not limit performance. The simulator has an unrealizable memory queue model, with perfect prediction of load latency (hits versus misses) and perfect knowledge of memory dependencies (i.e. loads are only issued when they will not depend on an earlier store). Although we would obtain greater savings by comparing to wider issue machines, we did not observe a substantial increase in IPC that would justify more than 4-issue on these codes even with the optimistic memory system.

In this paper, each configuration is labeled with the following nomenclature: *(arch)(#iq):(size)R(#read)W(#write)*, where (*arch*) is either the monolithic baseline (*BL*) or the RingScalar (*RS*) architecture, (*#iq*) is the total number of instruction window entries, (*size*) defines the regfile size, (*#read*) and (*#write*) are the number of read ports and the number of write ports in each regfile storage cell.

Our idealized 4-issue superscalar baseline,*BL32:80R8W4*, contains a conventional monolithic issue window with 32 issue queue entries, a fully multiported register file with 80 registers, and a full bypass network. The issue window uses an oldest-first priority scheme to select among multiple ready instructions. For RingScalar, we assume entries are evenly distributed among columns, e.g., *RS48:128R8W1* is a RingScalar design where each of the four columns has 12 issue window entries, and a bank of 32 registers with 8 read ports and one write port. Any RingScalar with a speculatively-controlled banked register file [21], has an additional read port arbitration stage added as shown in Figure 6, and the branch misprediction penalty increases by one cycle.

3.1 Resource Sizing

The register file and issue window of RingScalar is spread evenly across the columns. Their utilization is less than a monolithic structure and so the optimal sizing needs to be re-evaluated. Our first experiments show the effect of increasing the regfile size while keeping the issue window fixed. Figure 7 shows diminishing performance improvements as we increase the regfile size for both the baseline superscalar *BL32:xR8W4* and the RingScalar *RS48:xR4W1* processor. For the baseline design, IPC saturates at 144 registers; performance remains the same if additional registers are added beyond this point. RingScalar, however, keeps improving as more registers are added. Because instructions can only be allocated to a particular column, an imbalance of registers across the columns can lower the total regfile utilization. Nevertheless, the diminishing returns do not justify implementing a regfile that is larger than 128 for issue queue sizes of 48 and 64. The performance of the BL128:256R8W4 configuration is also plotted to show the limit on IPC for these codes with this simulation framework.

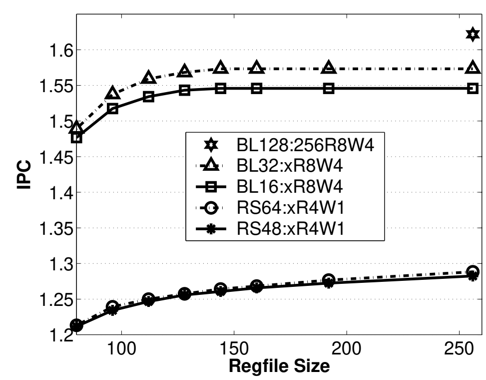

Figure 7. Average IPC comparison for different regfile size.

Our second set of experiments in Figure 8 shows how performance varies when increasing the size of the RingScalar issue window. For designs with 256 registers, IPC improvement tapers off beyond a window size of 64; for designs with 128 registers, it tapers off beyond a 48-entry window. This also demonstrates the importance of a balanced design, as increasing the resource in just a single area will not always lead to higher performance. We chose RS64:256 and RS48:128 as the two basic configurations for the RingScalar evaluation.

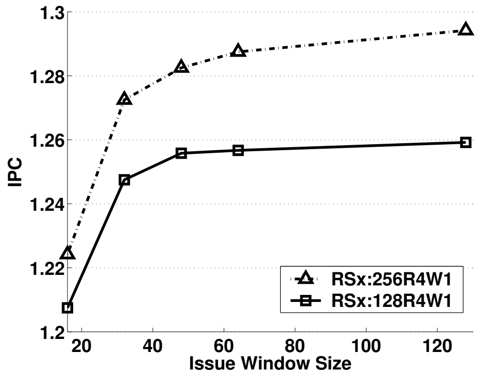

Figure 8. RingScalar average IPC sensitivity to instruction window size.

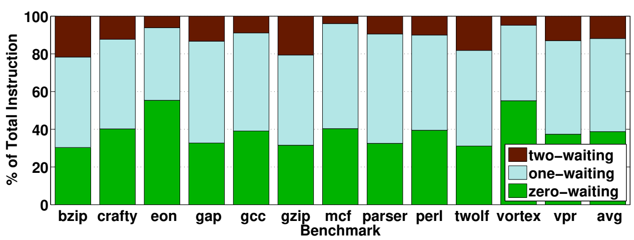

Figure 9. Distribution of zero-waiting, one-waiting, and two-waiting instructions for SPEC CINT2000 running on baseline superscalar.

3.2 Operand Availability

Previous work indicates that issue window source tags are underutilized and many instructions enter the issue queue with only zero or one outstanding register operands [5, 9]. Our results, shown in Figure 9, confirm this finding, with more than 80% of instructions waiting on one or zero operand for our four-way baseline superscalar processor running the SPEC CINT2000 benchmark suite. The percentage of two-waiting instructions ranges from 3.9% (mcf) to 21.9% (bzip) with an average of 11.8%.

As the second part of a two-waiting instruction will only be issued if woken up by the first part, the arrival timing of the two unmet operands impacts the performance of RingScalar. When the source operand of the second part arrives last, the waking of the second part is likely to finish before the instruction is ready to be issued. However, if the second part arrives first or arrives at the same time as the first part, the waking of the second part delays issue of the ready instruction for additional cycles. RingScalar uses a simple scheme where the left source operand is always predicted to arrive last. Figure 10 shows that the left source operand arrives last more than half of time in seven out of twelve programs (ratio ranges from 35.8% to 65.0% with an

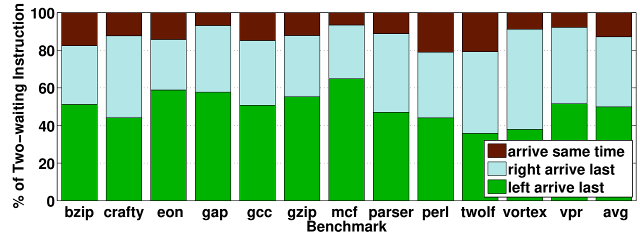

Figure 10. Percentage distribution of last-arrival operand for two-waiting instructions.

average of 50.0%). These numbers do not include store instructions, where the address calculation and data movement issue independently and in parallel.

3.3 IPC Comparison

We compared RingScalar performance against the baseline configuration. Simulations were run with a gshare branch predictor (Figure 11) and with a perfect branch predictor (Figure 12) to ascertain the effect of the extra pipeline stage and branch predictor inaccuracies. Relative performance differences remain reasonably consistent across the different branch predictor designs, with IPC increasing 12% on average with perfect branch prediction.

Despite their simplified window design and their much more realistic implementation parameters, the RingScalar results are quite competitive with the idealized superscalars. In comparison to the baseline (*BL32:80R8W4*), the performance of the small RingScalar design without regfile read-port conflicts (*RS48:128R8W1*) has an average IPC reduction of 12% with a maximum degradation of 24%. The performance impact is mainly due to delayed issuing of critical instructions, which can pile up in the same issue column. Extra regfile savings can be achieved in RingScalar with a lesser-ported banked structure. Figure 11 show that IPC drops only another 1% for *RS48:128R4W1* design but 4% for *RS48:128R2W1* design. Further comparing the *RS48:128R4W1* design to the large RingScalar design (*RS64:256R4W1*), only a 2% IPC difference is observed. The above data suggests that *RS48:128R4W1* is a good design point for a four-issue machine.

3.4 Two-waiting Queues

The performance loss from using a single-tag instruction window is evaluated by comparing the results to a variation of a RingScalar design where each issue queue column is divided into three banks. Figure 13 shows that instruction banks are of three types, depending on whether instructions are waiting on zero, one, or two source register operands. Unlike the original RingScalar design, instructions that wait on both operands are placed in the banks with two source

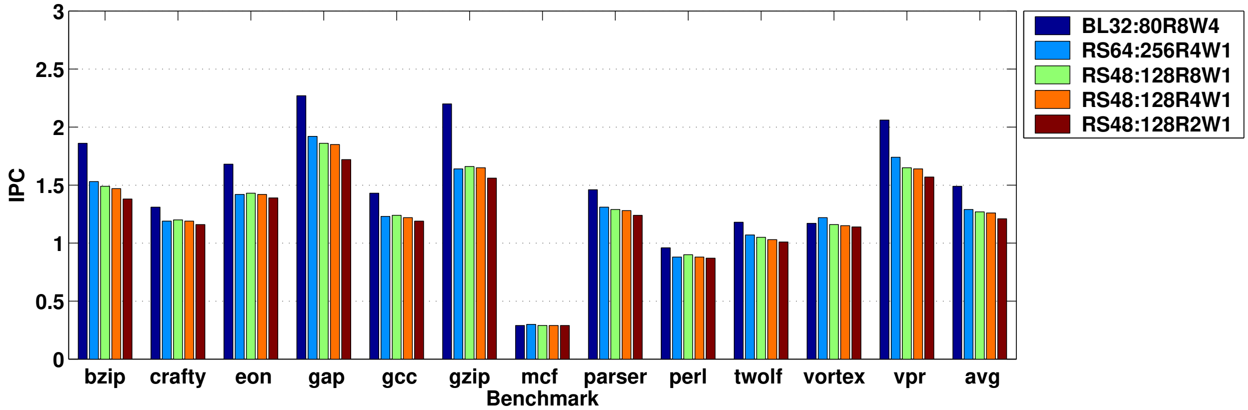

Figure 11. SPEC CINT2000 IPC with a gshare branch predictor.

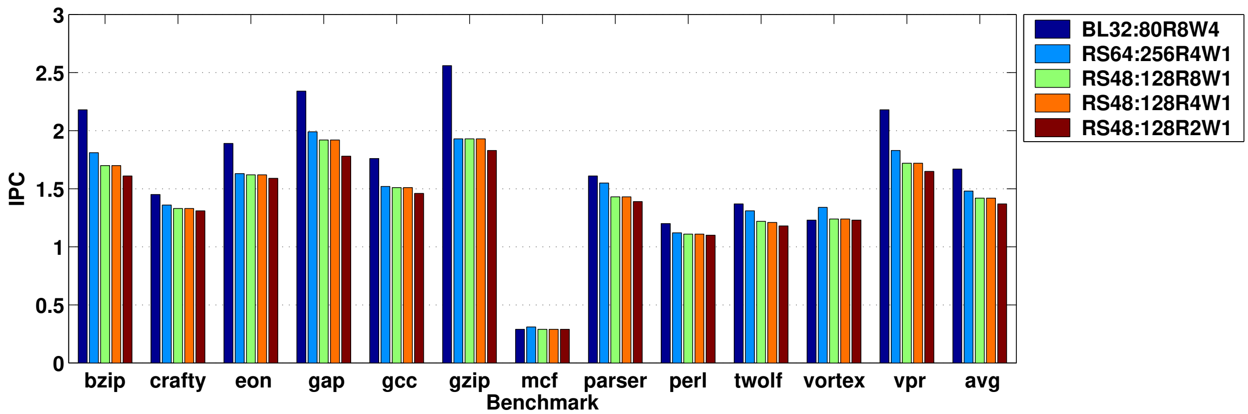

Figure 12. SPEC CINT2000 IPC with a perfect branch predictor.

tags. The two-waiting queues reduce the complexity in register renaming logic and eliminate the prediction on operand availability as two-waiting instructions are no longer split into two parts. Simulations show a 3% IPC improvement across the benchmarks after a 16-entry two-waiting queue is added to RingScalar (*RS48:128R4W1*). We believe this small improvement does not justify the additional area and power consumption of two-waiting queues.

4 Complexity Analysis

To determine the complexity effectness of RingScalar designs, we analyze the area, latency, and power reductions of key components in this section. The approach is to first compare required regfile die area by counting the number of occupied wire tracks. Then, we evaluate the factors that determine latency and power of issue windows.

The area of register file can be estimated by the number of bitlines and the number of wordlines [15, 20]. For regfiles with single-ended reads and differential writes, we use Equation 1 to approximate its grid area for a bit-slice. The width (w) and height (h) of a storage cell, including power and ground, is given in unit of wire tracks. R is the number of read ports, W is the number of write ports, and E is the number of physical entries in the regfile. The equation expresses that each regfile port calls for one wordline per entry, plus a single bitline for read ports and a pair of bitlines for write ports.

$$Area_{Regfile} = (w + R + W) \times (h + R + 2W) \times E \quad (1)$$

An issue window consist of dispatch ports, issue ports, wakeup port, comparators, tag broadcast network, and select arbiters. The speed of the wakeup-select loop is a function of wire propagation delay and the fan-in/fan-out delay. Its power consumption is proportional to switched capacitance, such as wire capacitance and transistor parasitic capacitance. RingScalar reduces these parameters by adopting lesser-ported banked structures. Power saving can also be achieved by minimizing the number of active components, such as comparators. Using Equation 2, the number of bit comparators in an issue window can be determined. T is the

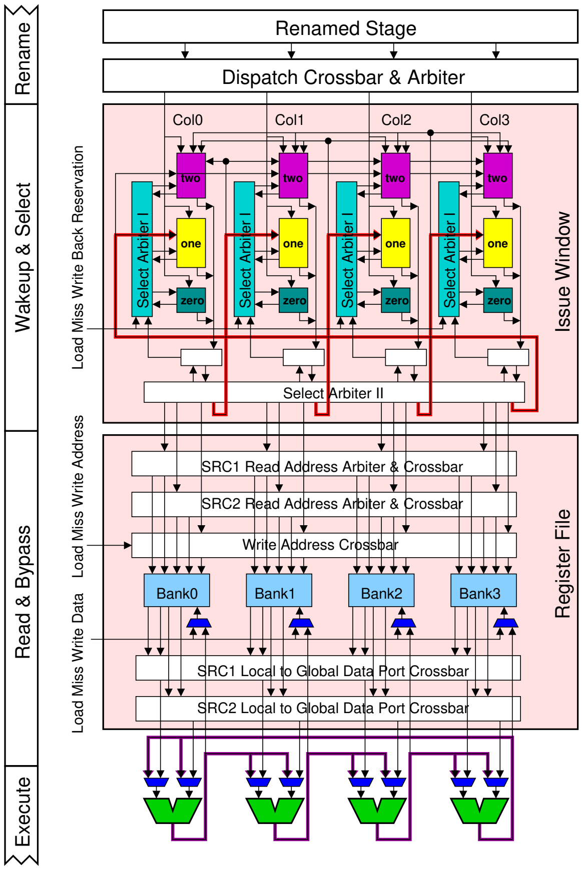

Figure 13. RingScalar architecture for designs with three issue banks per column.

number of tags per entry, B is the number of tag bits per entry (depends on the regfile size), Y is the number of wakeup port per tag, and E is the number of entries. For example, there are $2 \times 7 \times 4 \times 32 = 1792$ bit-comparators in the baseline (*BL32:80R8W4*) design.

$$Number_{bit-comparators} = T \times B \times Y \times E \qquad (2)$$

Table 2 provides a complexity and performance comparison across a few RingScalar designs and the baseline. In general, RingScalar designs are smaller and more power efficient than the conventional superscalar. Even the large RingScalar design (*RS64:256R4W1*) has a smaller regfile, a much simpler issue window (faster wakeup-select), and a smaller bypass network than the baseline, despite the increased number of regfile entries.

The *RS48:128R4W1* design point appears to be a sweet spot in the performance-complexity space. Regfile area is under half that of the baseline while the issue window is much smaller, and IPC is just 13.3% away from the baseline. Adding more read ports only increases IPC by 1% (*RS48:128R8W1*). The additional regfile area saving of moving from *RS48:128R4W1* to *RS48:128R2W1* is only 9% but this causes a 3% drop in IPC. Furthermore, a 48-entry RingScalar issue window requires only one fourth the number of dispatch, issue, and wakeup ports, and 21% of the tag comparators of a conventional 32-entry design. Table 2 also shows that RingScalar reduces wakeup delay because of its reduction in the wakeup broadcast fan-out. Comparing to the baseline configuration, *RS48:128R4W1* has faster select timing. *BL32:80R8W4* has one arbiter that selects four instructions out of a pool of 32 instructions while RingScalar has four arbiters, each independently selecting only one out of 12 instructions. The reduced bypass network decreases the ALU fan-out by a factor of three while the bypass mux fan-in is cut from seven to four for the RingScalar bypass networks.

In this evaluation, the baseline architecture was idealized in several respects. More realistic superscalar models should have a reduced IPC advantage over RingScalar. For example, most designs approximate the oldest-first select arbitration to reduce circuit complexity; real designs have less than fully orthogonal functional unit issue; and they will experience load-hit mispredictions and memory dependence misspeculations. These additional stalls will tend to reduce the IPC advantage of existing superscalar designs.

Although, due to the large engineering effort required, we are unable to complete a complete analysis using full-custom circuit implementation of both RingScalar and the conventional design, we believe the complexity analysis above shows that RingScalar is a promising approach which could enable significant cycle time, power, and area reductions.

5 Related Work

In this section, we describe how RingScalar relates to earlier work in banked register files, tag-elimination, and dependence-based scheduling [8, 10]. We also discuss the relationship to clustered designs.

RingScalar improves on previous work in banked register files by using the rename stage to steer instructions such that only a single write port per bank is required, avoiding write conflicts while supporting the simple pipeline control structure proposed in [20] to also reduce read ports per cell.

The fact that many operands are ready before an instruction is dispatched has inspired the tag-elimination [5], half-price architecture [9], and banked issue queue [3] designs. In comparison to the conventional scheduler, these schemes reduce the electric loading of the wakeup ports and the number of comparators by keeping the number of tag checks to a minimum. However, they still require N wakeup ports, N dispatch ports, and N issue ports. In contrast, RingScalar reduces both the number of tag comparisons and the number of wakeup ports. In addition, each issue queue entry of RingScalar requires only a single dispatch port and a single issue port.

Earlier dependence-based scheduling techniques have exploited the fact that it is unnecessary to perform tag checks

Configuration	Regfile Area	Issue Window		Wakeup Fan-out	Select Arbiter	Bypass Networks		IPC
		# Dispatch/Issue/ Wakeup Ports	# Comparators			ALU Fan-out	MUX Fan-in	
BL32:80R8W4	100.0%	4/4/8	100.0%	64	4 from 32	9	7	100.0%
RS64:256R4W1	80.8%	1/1/2	28.6%	16	1 from 16	3	4	89.7%
RS48:128R8W1	87.6%	1/1/2	21.4%	12	1 from 12	3	4	87.6%
RS48:128R4W1	40.4%	1/1/2	21.4%	12	1 from 12	3	4	86.7%
RS48:128R2W1	31.4%	1/1/2	21.4%	12	1 from 12	3	4	84.2%

Table 2. Total complexity comparisons. Results in percentage are normalized to the baseline (BL32:80R8W4).

on a consumer instruction prior to the issue of the producer, allowing instructions in the same dependency chain to be grouped to reduce wakeup complexity [8, 10, 13, 6]. By keeping track of the dependency chains and only issuing the instructions that reach the head of issue queues, these schemes eliminate tag broadcast and simplify the select logic. However, these designs either add complexity in tracking the dependency chain or require a large interconnect crossbar to connect the distributed ports. For example, the FIFO-based approach presented by Palacharla et al. [10] requires that N dependent instructions can be steered into the tail of a FIFO at dispatch time. This results in an $N \times N^2$ interconnect crossbar to allow any of the N dispatched instructions to connect to any of the N dispatch ports on any of the N FIFOs. In contrast, RingScalar spreads dependent instructions across different columns, reducing the dispatch interconnect requirements and the number of dispatch ports per entry. To reduce wakeup costs, RingScalar restricts the possible wakeup paths across windows, and to reduce select costs, only a subset of instructions are considered for each issue slot. RingScalar's integrated approach also reduces the cost of the register file and bypass network.

Although the techniques are similar, clustering can be distinguished from banking in two ways. First, the components that are most closely connected differ. A clustered architecture first tightly couples a local issue window, a local regfile, and local execution units within each cluster, then provides looser coupling between these clusters. A banked architecture first tightly couples the banks within an issue window or a regfile, often with combinational paths, then connects these major microarchitectural structures using conventional registers between pipeline stages. Second, clustering is generally used in larger scale designs, where each cluster is itself a moderate issue-width core, whereas banking reduces the complexity of a moderate issue-width core.

Multiscalar [18] was the earliest CDD clustered scheme to exploit control flow hierarchy, and uses compiler techniques to statically divide a single program into a collection of tasks. Each task is then dynamically scheduled and assigned to one of many execution clusters at run time. Aggressive control speculation and memory dependence speculation are implemented to allow parallel execution of multiple tasks. To maintain a sequential appearance, each task is retired in program order. This scheme requires extensive modification to the compiler and the performance depends heavily on the ability to execute tasks in parallel. Alternative CDD schemes include Trace Processors [16], which build traces dynamically as the program executes and which dynamically dispatches whole traces to different clusters. This approach, however, requires a large centralized cache to store the traces and the algorithm used to delineate traces strongly influences the overall performance. Both Multiscalar and Trace processors attempt to reduce inter-cluster communication by localizing sequential program segments to an individual cluster but rely largely on control and value predictions for parallel execution.

PEWs [8] and Multicluster [7] are DDD schemes that try to assign dependent instructions to the same cluster to minimize inter-cluster communication, with the assignments determined at decode time. Although performance is comparable to a centralized design for small numbers of clusters, these algorithms inherently have poor load balancing and cannot effectively utilize a large number of clusters. The Multicluster authors [7] suggest the possibility of using compiler techniques to increase utilization by performing code optimization and code scheduling. The fine-grained banking approach in RingScalar gives satisfactory performance with a simple fixed instruction distribution algorithm, whereas these larger clustered approaches often require complex instruction distribution schemes.

Abella and Gonzalez [1] recently published a related approach for clusters which distributes the workload across all the clusters by placing consumer instructions in the cluster next to the cluster that contains the producer instructions. The processor is laid out in a ring configuration so that the results of a cluster can be forwarded to the neighboring cluster with low latency. Each partition of the register file can be read only from its own cluster but can be written only from the previous neighboring cluster. This design still requires long latency inter-cluster communication to move missing operands to appropriate clusters. RingScalar uses a simpler variant of this ring scheme, although with the goal of reducing complexity rather than providing good load balancing.

Several of the proposals mentioned above have attempted to reduce the cost of one component of a superscalar architecture (e.g., just the register file or just the issue window), but often with a large increase in overall pipeline control complexity or possibly needing compensating enhancements to other portions of the machine (e.g., extending the bypass network to forward values queuing to use limited regfile write ports). The RingScalar architecture is engineered to simplify all of the major components simultaneously.

6 Conclusion

The RingScalar design provides complexity reduction throughout the major components of an out-of-order processor, in exchange for a small increase in complexity in the rename and dispatch stage. Compared with idealized superscalar architectures, there is only a small (10.3-13.3%) drop in IPC but with a large reduction in area, power, and latency of the issue window, register file, and bypass network. RingScalar should be even more competitive against realistic conventional superscalar processors, and should provide a suitable design point for CMP cores that need both high single thread performance and lower power and area.

References

[1] J. Abella and A. Gonzalez. Inherently workload-balanced clustered microarchitecture. In *IPDPS-19*, Long Beach, CA, April 2005.

[2] R. Balasubramonian, S. Dwarkadas, and D.H. Albonesi. Reducing the complexity of the register file in dynamic superscalar processors. In *MICRO-34*, Austin, TX, December 2001.

[3] A. Buyuktosunoglu, D. Albonesi, P. Bose, P. Cook, and S. Schuster. Tradeoffs in power-efficient issue queue design. In *ISLPED'02*, Monterey, CA, August 2002.

[4] J.-L. Cruz, A. Gonzalez, M. Valero, and N. P. Topham. Multiple-banked register file architectures. In *ISCA-27*, Vancouver, Canada, June 2000.

[5] D. Ernst and T. Austin. Efficient dynamic scheduling through tag elimination. In *ISCA-29*, Anchorage, AK, May 2002.

[6] D. Ernst, A Hamel, and T. Austin. Cyclone: A broadcast-free dynamic instruction scheduler with selective replay. In *ISCA-30*, San Diego, CA, June 2003.

[7] K. I. Farkas, P. Chow, N. P. Jouppi, and Z. G. Vranesic. The Multicluster architecture: Reducing cycle time through partitioning. In *MICRO-30*, Research Triangle Park, NC, December 1997.

[8] G. A. Kemp and M. Franklin. PEWs: A decentralized dynamic scheduler. In *ICPP'96*, Bloomingdale, IL, August 1996.

[9] I. Kim and M. Lipasti. Half-price architecture. In *ISCA-30*, San Diego, CA, June 2003.

[10] S. Palacharla, N. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *ISCA-24*, Denver, CO, June 1997.

[11] I. Park, M. D. Powell, and T. N. Vijaykumar. Reducing register ports for higher speed and lower energy. In *MICRO-35*, Istanbul, Turkey, November 2002.

[12] D. Ponomarev, G. Kucuk, O. Ergin, K Ghose, and P. Kogge. The Alpha 21264 microprocessor. *IEEE Transactions on Very Large Scale Integration Systems*, 11(5), October 2003.

[13] S. Raasch, N. Binkert, and S. Reinhardt. A scalable instruction queue design using dependence chain. In *ISCA-29*, Anchorage, AK, May 2002.

[14] N. Ranganathan and M. Franklin. An empirical study of decentralized ILP execution models. In *ASPLOS-8*, San Jose, CA, October 1998.

[15] S. Rixner, W. Dally, B. Khailany, P. Mattson, U. J. Kapasi, and J. D. Owens. Register organization for media processing. In *HPCA*, Toulouse, France, 2000.

[16] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. E. Smith. Trace processors. In *MICRO-30*, Research Triangle Park, NC, December 1997.

[17] S. Sair and M. Charney. Memory behavior of the SPEC2000 benchmark suite. Technical report, IBM Research Report, Yorktown Heights, New York, October 2000.

[18] G. S. Sohi, S. Breach, and T. N. Vijaykumar. Multiscalar processors. In *ISCA-22*, Santa Margherita Ligure, Italy, June 1995.

[19] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal*, 11(1), January 1967.

[20] J. Tseng and K. Asanović. Banked multiported register files for high-frequency superscalar microprocessors. In *ISCA-30*, San Diego, CA, June 2003.

[21] J. Tseng and K. Asanović. A speculative control scheme for an energy-efficient banked register file. *IEEE Transactions on Computers*, 54(6), June 2005.

[22] D.M. Tullsen. Simulation and modeling of a simultaneous multithreading processor. In *The 22nd Annual Computer Measurement Group Conference*, San Diego, CA, December 1996.

[23] S. Wallace and N. Bagherzadeh. A scalable register file architecture for dynamically scheduled processors. In *PACT-5*, Boston, MA, October 1996.

[24] K. C. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, 16(2), April 1996.