Half-Price Architecture

Ilhyun Kim and Mikko H. Lipasti
Department of Electrical and Computer Engineering
University of Wisconsin–Madison
{ikim,mikko}@ece.wisc.edu

Abstract

Current-generation microprocessors are designed to process instructions with one and two source operands at equal cost. Handling two source operands requires multiple ports for each instruction in structures--such as the register file and wakeup logic--which are often in the processor's critical timing paths. We argue that these structures are overdesigned since only a small fraction of instructions require two source operands to be processed simultaneously. In this paper, we propose the half-price architecture *that judiciously removes this overdesign by restricting the processor's capability to handle two source operands in certain timing-critical cases. Two techniques are proposed and evaluated: one for the wakeup logic is sequential wakeup, which decouples half of the tag matching logic from the wakeup bus to reduce the load capacitance of the bus. The other technique for the register file is sequential register access, which halves the register read ports by sequentially accessing two values using a single port when needed. We show that a pipeline that optimizes scheduling and register access for a single operand achieves nearly the same performance as an ideal base machine that fully handles two operands, with 2.2% (worst case 4.8%) IPC degradation.*

1. Introduction & Motivation

A compelling argument in favor of RISC design was that simpler instruction sets could be realized more efficiently, giving them an inherent performance advantage over complex instruction sets. Many current-generation microprocessors follow a similar philosophy in their hardware designs. Even in processor implementations for complex instruction sets such as IA-32, complex instructions are broken into micro-ops running on RISC-style cores. The principles of these designs can be summarized as *regularity*, *orthogonality* and *composability* [15], implying that a collection of homogeneous and atomic instruction primitives perform complex operations. The regularity in their hardware and software interface has been regarded as a key factor to fast and efficient designs with simpler controls over instruction and data flow. Being faithful to this paradigm, the user-visible ISA usually adopts two source and one destination operands as its basic design since a reasonable amount of useful work can be expressed in this format. Accordingly, hardware has also been built to handle this two-to-one operand configuration as a minimal processing unit, which many hardware designers have considered intuitive and reasonable, since execution eventually occurs at *instruction granularity.*

In this paper, we argue that current-generation microprocessors are overdesigned to process instructions with one and two source operands at equal cost. This design was originally chosen to avoid structural hazards, by assuming that the vast majority of instructions have two source operands. It simplifies the processor's control logic in a classic streamlined pipeline since the hardware is guaranteed to satisfy the worst-case requirements of an instruction. Hence, arbitrating resources to handle source operands is unnecessary. This decision was made when building extra hardware (such as a dual-ported register file that deals with two source operands) was feasible.

As the machine width increases in modern superscalar, out-of-order processor designs, the complexity of the processor grows exponentially to process multiple instructions in parallel. This requires heavily multi-ported structures in many pipeline stages; such structures can limit the processor's cycle time. However, they are still designed based on the premise of two source operands per instruction, hence requiring twice the machine bandwidth for processing source operands. Figure 1 shows the base machine model used in this paper, and pipeline stages built to handle two source operands (illustrated as shadowed boxes).

Although the static and dynamic count of these two-source instructions still accounts for a significant portion of total instructions, in fact, it is infrequent that such overdesigned structures are fully utilized since only a small fraction of instructions require two source operands to be processed simultaneously in the out-of-order execution window. In later sections, we will show that less than 3% of instructions have their two source operands awakened in the same clock cycle, and that less than 4% of instructions require two register read port accesses for source values. Hence, handling two-source instructions is no longer the common case, and the benefit of simplified control over instructions is not likely to justify the cost for the overdesigned hardware. Rather, changing the design of at least a portion of the pipeline structures from instruction granularity to *operand granularity* would be desirable so that the processor optimizes the common case: zero or one-source instructions.

We propose the *half-price architecture* to reduce hardware complexity by restricting the processor's capability to handle two source operands. In this architecture, structures such as wakeup logic and the register file are configured to handle instructions with zero or one source operand without any restrictions; in contrast, instructions with two source operands may execute more slowly. Specifically, this paper proposes and evaluates two solutions for the

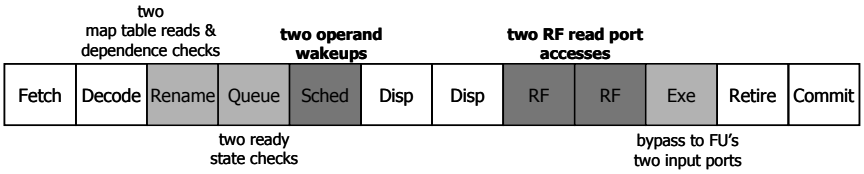


Figure 1. The base machine model and pipeline stages that handle two source operands. Instructions with two source operands complicate many stages in the pipeline, which are illustrated as shadowed boxes. This paper discusses possible solutions for the scheduling (Sched) and register file access (RF) stages.

scheduling and register access stages that are likely in the processor's critical path. One targets the wakeup logic: *sequential wakeup* decouples half of the tag matching logic from the wakeup bus to reduce the load capacitance of the bus. The other technique is for the register file: *sequential register access* halves the register read ports by sequentially accessing two values using a single port when needed.

The rest of the paper is organized as follows: Section 2 describes the simulation methodology and identifies instructions with two source operands that the half-price architecture targets. Section 3 and 4 characterize dynamic instruction behaviors in the out-of-order execution core, and propose techniques that reduce wakeup logic and register file complexity. Section 5 provides a detailed performance evaluation of those techniques and shows that the half-price architecture can reap most of the performance of a conventional pipeline that fully handles two source operands.

2. Simulation Methodology

2.1. Processor model

Our execution-driven simulator used in this study is derived from the *SimpleScalar / Alpha* 3.0 tool set [1], a suite of functional and timing simulation tools for the Alpha AXP ISA. Specifically, we extended *sim-outorder* to perform speculative scheduling with non-selective recovery, similarly to Alpha 21264 [2]. In this pipeline, instructions are scheduled in the scheduling stage assuming load instructions have fixed execution latency and any latency changes due to e.g. cache misses, cause all subsequently issued instructions to be rescheduled. We modeled a 12-stage out-of-order pipeline with 4- and 8-instruction machine width. The pipeline structure is illustrated in Figure 1. The detailed configurations of each machine model are shown in Table 1.

Since our techniques focus on instruction scheduling, it is worth noting that our scheduler employs an oldest-instruction-first selection policy with load and branch instructions having higher priorities than other types. Older instructions in program order have higher priority in each priority group. This selection policy is similar to the one employed by the base SimpleScalar model.

2.2. Benchmarks

The SPEC CINT2000 integer benchmark suite is used for all results presented in this paper. All benchmarks were compiled with the DEC C and Fortran compilers under the OSF/1 V4.0 operating system using -O4 optimization.

Table 1: Machine Configurations

	4-wide	8-wide
Out-of-order Execution	4-wide fetch/issue/commit, 64 RUU, 32 LSQ, speculative scheduling w/ non-selective recovery	8-wide fetch/issue/commit, 128 RUU, 64 LSQ, speculative scheduling w/ non-selective recovery
Functional Units (latency)	4 integer ALUs (1), 2 floating ALUs (2), 2 integer MULT/DIV (3/20), 2 floating MULT/DIV (4/12), 2 memory ports	8 integer ALUs (1), 4 floating ALUs (2), 4 integer MULT/DIV (3/20), 4 floating MULT/DIV (4/12), 4 memory ports
Branch Prediction	Combined bimodal (4k entry) / gshare (4k entry) with a selector (4k entry), 16 RAS, 1k-entry 4-way BTB, at least 11 cycles for misprediction recovery, fetch stops at first taken branch in a cycle	
Memory System (latency)	64KB 2-way 32B line IL1 (2), 64KB 4-way 16B line DL1 (2), 512KB 4-way 64B line unified L2 (8), main memory (50)	

Table 2: Benchmarks

Benchmarks	input sets	inst count	Base IPC (4 / 8-wide)
bzip	lgred.graphic	2.64B	1.74 / 2.16
crafty	crafty.in	3B	1.92 / 2.65
eon	chari.control.cook	3B	2.00 / 2.41
gap	ref.in	3B	1.99 / 2.43
gcc	lgred.cp-decl.i	5.12B	1.52 / 1.95
gzip	lgred.graphic	1.79B	1.84 / 2.11
mcf	lgred.in	0.79B	0.71 / 0.93
parser	lgred.in	4.52B	1.24 / 1.42
perl	lgred.markerand	2.06B	1.36 / 1.58
twolf	lgred.in	0.97B	1.45 / 1.65
vortex	lgred.raw	1.15B	2.02 / 2.95
vpr	lgred.raw	1.57B	1.64 / 1.88

Table 2 shows the benchmarks, input sets, the number of instructions committed, and IPC on 4 and 8-wide base machines. The large reduced input sets from [16] were used for all benchmarks except for *crafty, eon* and *gap*. These three benchmarks were simulated with the reference input sets up to 3 billion instructions since the reduced inputs are not yet available.

2.3. Identifying 2-source instructions

Our base architecture is the Alpha ISA, which is a load / store RISC architecture. There are four major instruction format classes that contain 0, 1, 2 or 3 register fields, supporting up to 2 source register operands and 1 destination register operand. Since the goal of this study is to reduce complexity of structures that handle multiple source operands, we present how many instructions require two valid

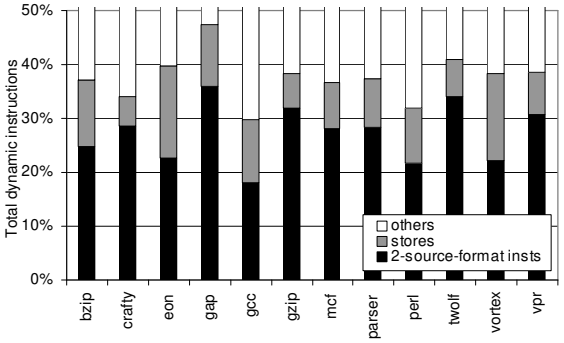


Figure 2. Percentage of 2-source-format instructions. 18~36% of instructions have two source operands in their format. Stores are presented in their own category since they are handled differently inside the processor core.

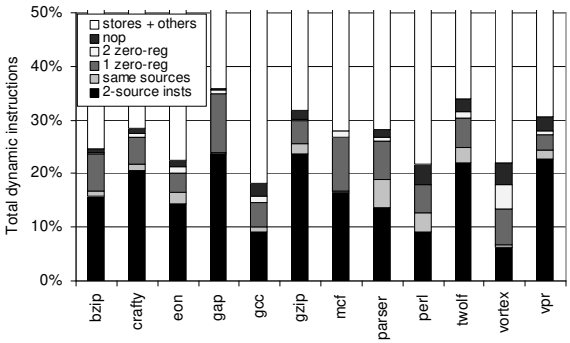


Figure 3. Breakdown of 2-source-format instructions. Only instructions in the bottom-most bars have 2 unique source operands that create dependences. They will be referred to as *2-source* instructions in the later sections.

source registers across the benchmarks with given input sets. The data in this section shows program characteristics and does not depend on machine configuration. Characterization of dynamic behavior within the processor core which is dependant on machine configuration will appear in later sections.

Figure 2 indicates the percentage of dynamic instructions with two source operands (referred to as *2-source format*). The figure shows that 18~36% of dynamic instructions use 2-source format, indicating that this is by far not the common case. We present stores in their own category in the figure because even though store instructions have two source operands, they do not behave as 2-source-format instructions for the following reasons: first, the actual store operation that accesses the cache is usually scheduled separately and performed in the commit stage where dynamic instruction scheduling is not involved. Second, stores can be handled as two operations, such as an address generation and a move that copies the register value into the store scheduler. Since the Alpha ISA does not support a MEM[reg + reg] indexing mode for memory operations, neither address generation nor this move operation requires two source operands.

Figure 3 shows a breakdown of 2-source format instructions as a function of the number of unique source operands. 2-source-format instructions may require less than two source registers for their inputs since many instructions have one or two zero-registers (`r31` and `f31` in Alpha ISA) that do not create register dependences (e.g. `add r1← r2, r31`), or two identical source operands (e.g. `add r1← r2, r2`). We note that Alpha binaries have a measurable number of 2-source-format nops that write to zero-registers in order to satisfy instruction alignment restrictions; they are eliminated from the instruction stream by the decoder without execution [2], and are shown separately in the figure. Across the benchmarks tested, 6~23% of dynamic instructions in the bottom-most bars are measured to have two unique, non-zero source operands (referred to as *2-source* instructions).

2-source instructions will be the basis of further analysis in later sections. Section 3 and 4 will present that only a few of them require two operands to be processed simultaneously in the out-of-order execution core despite the fact that 2-source instructions are still a significant portion of all dynamic instructions.

3. Reducing scheduling logic complexity

The wakeup logic is responsible for waking up instructions in the instruction window when all of their source operands become available. The select logic selects instructions to be issued among a pool of ready instructions; the selected instructions then broadcast their destination tags to all instructions in the window in the next wakeup cycle. From the physical design perspective, broadcasting tags is quite expensive because all wakeup buses should be connected to all source tag comparators in all issue queue entries, which significantly increases the load capacitance of the bus the wakeup logic drives. Moreover, wakeup and select operations in conventional schedulers should be atomic (i.e. they cannot be easily pipelined [17]) in order to achieve consecutive instruction scheduling in back-to-back cycles. For these reasons, the instruction scheduling logic is likely to be a bottleneck in both current and future microprocessors running at a high operating frequency.

Each issue queue entry is usually equipped with two tag comparators because the scheduler needs to handle 2-source instructions. However, this design is not fully utilized even by 2-source instructions because some of the input operands may be ready when the instruction is inserted into the scheduler, and therefore no "wake-up" operation is required for these operands. Figure 4 presents the number of ready operands of 2-source instructions when they are inserted into the scheduler. Many instructions already have one or two ready operands, and only 4~16% of instructions (shown in the bottom-most bars as *0 ready*) have *2-pending-source* operands that require two tag comparators.

Even though tag comparisons for all pending-source operands are required for correct instruction scheduling, it is only the last-arriving operand that initiates instruction issue. Our half-price technique, *sequential wakeup* exploits this observation and eliminates half of the load capacitance incurred by tag comparators from the wakeup bus by connecting only one critical operand per instruction directly to the wakeup bus, while other operands are decoupled from

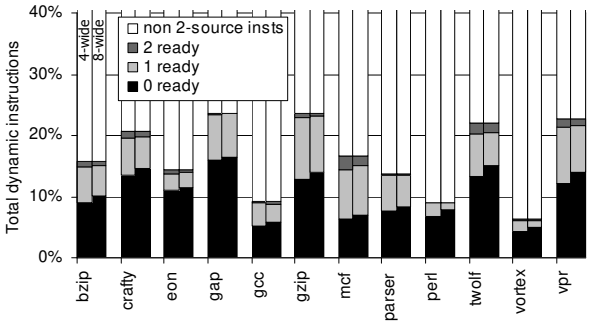


Figure 4. Breakdown of 2-source instructions, as a function of the number of ready operands when they are inserted into the scheduler. Only 4~16% of instructions have 2 unresolved operands at insert time.

the wakeup bus. It enables the wakeup logic to operate at a higher frequency with minimal performance impact.

3.1. Related work

Palacharla et. al. [12] proposed the clustered microarchitecture where a collection of small windows work as a wider and deeper instruction window. Michaud and Seznec [3] proposed the data-flow based instruction window that reorders instructions before they enter the issue queue according to data dependences. Lebeck et. al. [13] studied the effect of cache misses on the instruction window and explored scheduler designs that re-insert the load and dependent instructions after the cache miss is resolved. Hrishikesh et. al. [14] proposed to pipeline the instruction wakeup logic that manages segmented instruction windows. Most of these studies focus on making the instruction window look bigger, by either giving different priorities to different instruction groups, or keeping long-latency instructions in a separate structure before they enter the small window. On the other hand, our technique tries to reduce the cost of tag matching for multiple source operands and therefore is orthogonal to those techniques.

Most recently, Ernst and Austin [10] proposed *tag elimination*, a combined scheme that uses specialized windows and *last-tag speculation* to achieve wakeup logic cycle time improvement by reducing load capacitance on the wakeup bus; this is similar in spirit to our sequential wakeup scheme. Specifically, their last-tag speculation performs tag comparisons only for operands predicted to be last-arriving, and removes tag comparators for other operands, leaving a single tag matching logic per issue queue entry. This approach does not guarantee correctness of instruction scheduling and instructions may be incorrectly issued before all operands become ready (due to last-arriving operand mispredictions). Therefore, a scoreboard must check the readiness of operands that were not connected to the wakeup bus and flag a misprediction to the scheduler, causing instructions to be re-issued. For scheduling recovery, their technique relies on an Alpha 21264-style, non-selective recovery mechanism that invalidates all dependent and independent instructions issued following the mispredicted instruction.

Tag elimination differs fundamentally from sequential wakeup due to its speculative nature. That is, our sequential wakeup approach does not require any scheduling recovery from mispredictions since no instruction is issued speculatively with regard to readiness of source operands. On the other hand, mispredictions in their tag elimination scheme may affect performance severely, especially on a wider machine where the rescheduling penalty grows as the number of instructions to be invalidated increases. In Section 5.1, we will show that the performance of the tag elimination scheme does not scale well with increasing misprediction penalty.

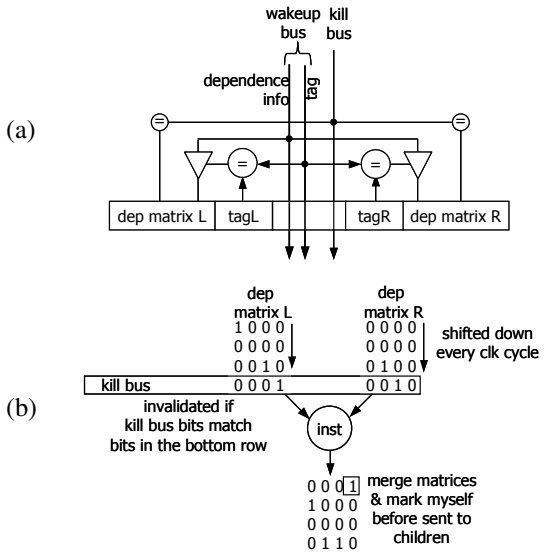


Figure 5. A possible implementation of selective recovery logic. (a) the list of issued parent instructions in the pipeline is propagated to dependent instructions along with tag broadcasts. (b) the source operand with matching bits in the kill bus is invalidated, indicating that misscheduling of its parent instruction is detected.

Many recent processor designs have adopted a selective recovery mechanism that significantly reduces the rescheduling penalty by invalidating and replaying only data-dependent instructions on a misprediction. Although one might expect that this selective recovery would also reduce the rescheduling penalty in the tag elimination scheme, unfortunately, implementing tag elimination on a machine with selective recovery is impractical. This can be easily explained by showing how the selective recovery mechanism keeps track of data dependences among instructions. Figure 5 shows a possible implementation of selective recovery logic, where the wakeup bus is used not only to wake up dependent instructions, but also to propagate the dependence information to child instructions in matrix form. In these matrices, rows and columns imply pipeline stages and issue slots, where current locations of issued parent instructions are marked. When a child instruction receives matrices from parent instructions, it merges matrices from both source operands, and marks its own location on the matrix before it forwards the matrix to its dependent instructions. During this propagation, all instructions get the list of all parent instructions that they depend on. Bits in the matrix are shifted down every clock cycle to be in sync with the pipeline flow until all bits are phased out, as all parent instructions in the pipeline reach functional units

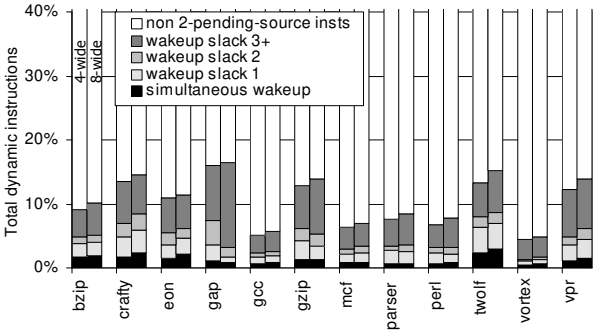


Figure 6. Slack between two operand wakeups. Less than 3% of instructions have two operands that become ready in the same clock cycle.

and execute. If a mis-scheduling is detected in the execute stage due to e.g. a cache miss, it is signaled through the kill bus that invalidates all operands with the corresponding bit (the bit in the same column as the faulty issue slot in the last row) in their matrices.

In the tag elimination scheme, removing tag matching logic prevents early-arriving operands from participating in this dependence propagation, and therefore the instruction cannot correctly determine if it is dependent on rescheduling events. In contrast, our approach is fully compatible with selective recovery since both operands observe dependence information, while the wakeup logic can still benefit from the reduced bus capacitance.

3.2. Characterization of operand wakeups

Although 2-pending-source instructions need two wakeups inside the scheduler before they are issued, it is relatively infrequent that two source operands are awakened in the same clock cycle. Figure 6 shows the breakdown of 2-pending-source instructions as a function of wakeup slack (the number of clock cycles between two source operand wakeups). The data shows that the vast majority of 2-pending-source instructions have at least one clock cycle slack between wakeups, and that two source operands are awakened simultaneously for less than 3% of instructions. This wakeup slack can be exploited to reduce the wakeup bus capacitance by prioritizing operand wakeups and hiding the delay of tag comparisons for operands with a lower priority.

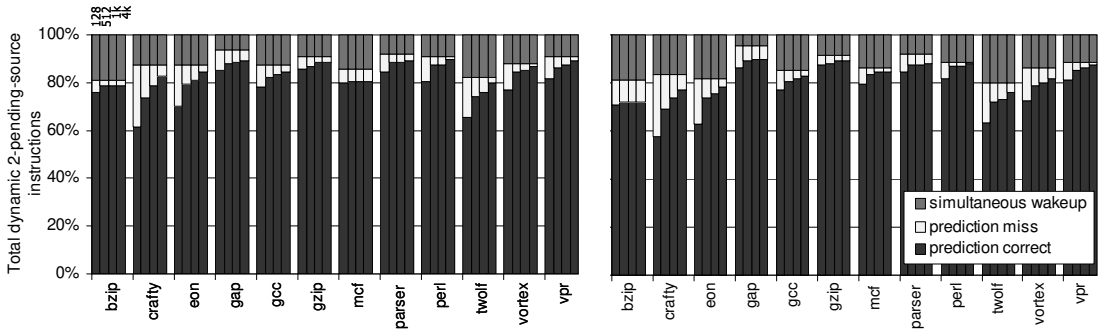
To achieve these benefits, our wakeup logic needs a

Table 3: Characterization of operand wakeup order and last-arriving operand

Bench-marks	4-wide		8-wide	
	% wakeup order (same / diff)	% last-arriving operand (left / right)	% wakeup order (same / diff)	% last-arriving operand (left / right)
bzip	86.9 / 13.1	51.3 / 48.7	82.5 / 17.5	50.0 / 50.0
crafty	88.4 / 11.6	49.0 / 51.0	82.4 / 17.6	53.9 / 46.1
eon	91.3 / 8.7	49.2 / 50.8	86.1 / 13.9	53.1 / 46.9
gap	88.3 / 11.7	49.7 / 50.3	84.9 / 15.1	49.4 / 50.6
gcc	86.8 / 13.2	43.8 / 56.2	90.0 / 10.0	50.3 / 49.7
gzip	90.1 / 9.9	43.4 / 56.6	92.0 / 8.0	49.0 / 51.0
mcf	81.4 / 18.6	44.4 / 55.6	91.6 / 8.4	61.5 / 38.5
parser	93.0 / 7.0	44.2 / 55.8	93.4 / 6.6	48.5 / 51.5
perl	98.1 / 1.9	72.9 / 27.1	98.9 / 1.1	80.3 / 19.7
twolf	87.6 / 12.4	46.4 / 53.6	88.5 / 11.5	50.7 / 49.3
vortex	93.4 / 6.6	28.5 / 71.5	88.8 / 11.2	30.4 / 69.6
vpr	92.5 / 7.5	62.7 / 37.3	92.5 / 7.5	65.5 / 34.5

prediction scheme to determine the operand that matches the last-arriving tag to initiate instruction issue. Table 3 shows the predictability of last-arriving operands of 2-pending-source instructions measured on the base 4 and 8-wide machines. The second and fourth columns present the percentage of same / different operand wakeup orders of 2-source instructions, compared with the last history at the same PC. The third and fifth columns in the table present the percentage of left / right last-arriving operands. The numbers do not include *simultaneous wakeup* cases where both operands become ready in the same clock cycle. Although last-arriving operands are randomly distributed between operands on both sides, the order of operand wakeups is stable, showing that around 90% of the time a static instruction has the same last-arriving operand as its previous execution. This implies that a history-based predictor can easily determine which operand would arrive last.

There are several studies on the design of last-arriving operand predictors [10][17] for various purposes in different contexts. We examined several predictors and determined that a simple PC-based, direct-mapped bimodal predictor with 2-bit saturating counters achieves similar prediction accuracy with a relatively small table size, compared with other more sophisticated designs. Figure 7



(a) 4-wide (b) 8-wide

Figure 7. Last-arriving operand prediction accuracy on 4 and 8-wide machines

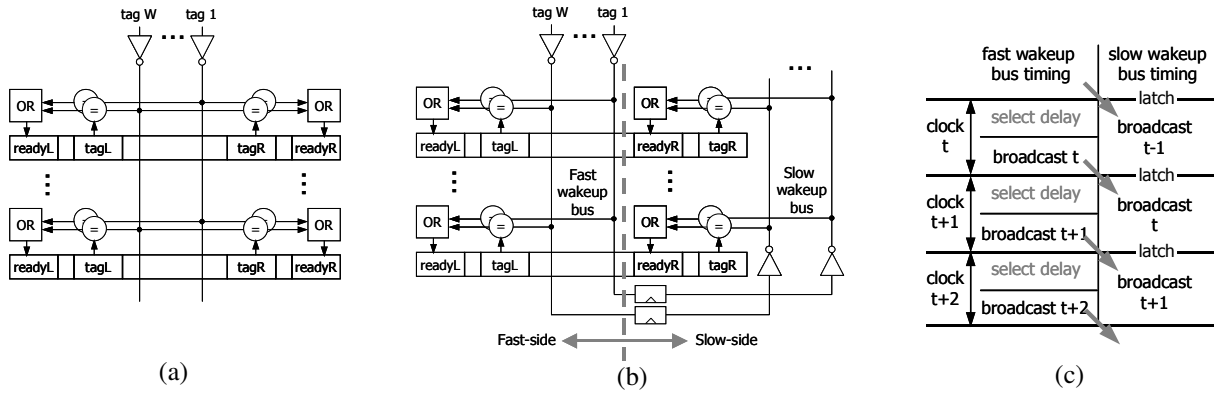


Figure 8. Reducing the fanout of wakeup buses. (a) conventional wakeup logic, (b) sequential wakeup logic, and (c) broadcast timings of fast / slow wakeup buses.

shows the accuracy of the bimodal predictor on 4 and 8-wide machines, varying the number of the table entries from 128 to 4096. Simultaneous wakeups of both operands are shown in top-most bars, which can be interpreted as either mispredictions or correct predictions, depending on the wakeup logic that we present in the later sections.

3.3. Sequential wakeup

Figure 8 illustrates the structure of our proposed sequential wakeup logic. Unlike the conventional design shown in (a), only one side of operands are directly wired to the *fast wakeup* bus that broadcasts tags of issued instructions in the same clock cycle. Then, the tags on fast wakeup buses are latched and re-broadcast in the next clock cycle using the *slow wakeup* bus, sequentially waking up operands on the other side. The size of the wakeup logic is important because the wire delay of the wakeup bus increases as the logic size grows. Although sequential wakeup requires two wakeup buses, it would not increase the size of the fast-side wakeup logic since broadcasting tags using the slow wakeup bus can extend over the selection logic delay as shown in Figure 8c, and therefore the slow-side wakeup logic can be located far from the fast side.

Based on the last-arriving operand predictions performed in the instruction fetch stage, the scheduler puts the last-arriving operand in the fast side and the other operand in the slow side when instructions are inserted into the issue queue entries. Since the actual physical register specifiers are acquired from the payload RAM [18] in later stages (*Disp* in Figure 1) in our base machine, scheduling instructions with swapped tags does not affect correctness. The operands in the slow side always see the tags one clock cycle later than those in the fast-side wakeup logic. If the prediction is correct, the broadcast delay through the slow wakeup bus is hidden by the wakeup slack and does not affect performance; however, if the prediction is incorrect, i.e. the last-arriving operand is placed in the slow side, the instruction issue will be delayed by one clock cycle since the slow bus wakes up the instruction one clock cycle later than the base case. This delay can be also hidden by issuing another ready instruction in the best case. An example of sequential wakeup is illustrated in Figure 9, where all instruction are issued without penalty.

One disadvantage of sequential wakeup is that simultaneous wakeups, in which both operands become ready in the same clock cycle, always incur one clock cycle penalty because the broadcast delay in the slow-side wakeup logic cannot be hidden behind the wakeup slack. However, less than 3% of dynamic instructions fall into this category, as shown in Figure 6. In Section 5, we will illustrate that the performance degradation incurred by simultaneous wakeups is negligible.

The biggest advantage of this approach over previous work [10] is that our approach requires neither a detection mechanism nor scheduling recovery for mispredictions. Furthermore, it can be easily integrated into a pipeline with selective recovery because early-arriving operands still observe dependence information propagated through the wakeup bus. In contrast, we discussed in Section 3.1 that combining tag elimination with selective recovery is impractical. In addition, sequential wakeup logic maintains the benefits of reduced wakeup bus load capacitance by decoupling half of the tag matching logic from the fast wakeup bus, enabling a higher clock frequency in the scheduling logic. Consistent with the circuit analysis presented by Ernst and Austin [10], we believe the total delay of a 4-wide, 64-entry scheduler with sequential wakeup can be reduced from 466ps to 374ps[1], which is a 24.6% speedup over a conventional scheduler that fully handles two source operands.

4. Reducing register file complexity

A multiple-issue, high performance microprocessor requires a register file with many read and write ports. This highly multiported structure becomes challenging to design because the area of a register file increases quadratically and the latency increases approximately linearly as the number of ports grows [6][7][8].

Conventional register files in RISC-style microprocessors have two register read ports and one write port per issue slot. Even though this 2X read port configuration guarantees register accesses for all types of instructions, it

1. The wakeup bus length and the number of tag comparators of the sequential wake logic on the given machine parameters is equivalent to 16/32/16 configuration in [10].

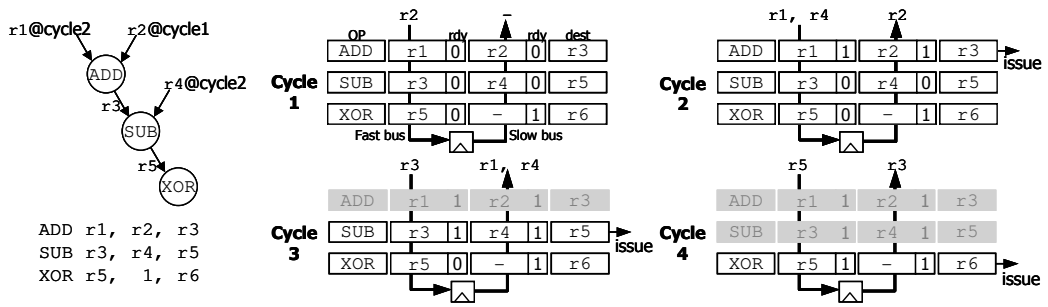


Figure 9. An example of sequential wakeup

is wasteful because many of 0 or 1-source instructions do not need two register read accesses. The only reason each issue slot has two register read ports is to handle 2-source instructions that require two source values from the register file without stalling the pipeline. However, many 2-source instructions do not fully utilize two read ports since they frequently get source values from the bypass logic, and only 4% of instructions require two read port accesses to the register file, which is far from the common case. *Sequential register access* eliminates half of the register read ports with a minimal impact on performance by making instructions sequentially access a single port when two source values are needed from the register file, which enables a more compact and faster register file design. Using a register access model derived from CACTI 3.0 [19], we found that the access time for a 160-entry register file at 0.18µ technology is reduced from 1.71ns to 1.36ns, which is a 20.5% drop when the number of register ports decreases from 24 to 16 on an 8-wide machine.

4.1. Related work

To tackle the problem of the large multi-ported register file, several researchers have studied hierarchical or clustered register file structures. Cruz et. al. [5] and Balasubramonian et. al. [4] studied two-level hierarchical organizations with different caching policies for the smaller register file so that it can be accessed in a single clock cycle. Balasubramonian [4] and Park [23] proposed techniques to reduce the number of register read ports. Although their schemes also exploit the fact that many instructions do not require two register accesses and many values are read from the bypass, they still service two input ports for all functional units by placing a fully-connected crossbar into the register file. Moreover, arbitration for register file ports is performed globally. On the other hand, our approach does not require such a crossbar with global arbitration logic. Hoogerbrugge and Corporaal [9] studied a compiler-controlled register file with a smaller number of ports in the context of VLIW processors. As a real processor implementation, Alpha 21264 [2] uses dual-banked register files to reduce the number of read ports. Our approach is orthogonal to such a replicated register file and can be applied in conjunction with other complexity-effective register file techniques.

It is important to note that incomplete bypass on the clustered architecture would reduce the efficiency of the sequential register access scheme if a result value of one cluster is passed to others only through the register file, because more instructions require two read port accesses from the register file as the bypass opportunity decreases. We did not study sequential register access on a clustered microarchitecture and leave it as future work because performance is highly dependent on the cluster steering policy. However, many researchers [12][21][22] have studied steering policies for reducing inter-cluster communication and hiding its delay on such microarchitectures; the same policies can be used in favor of sequential register access to minimize performance impact.

4.2. Characterization of register read port accesses

Bypass logic is responsible for broadcasting result values to input ports for all functional units so dependent instructions can execute in the very next clock cycle before the result value is written back to the register file. Since many result values can be read off the bypass, and many instructions do not require two source values, the average port utilization is much lower than the peak register file port utilization [4][5].

If an issue slot is restricted to only one register read port, 2-source instructions can still be handled by accessing a single port twice to satisfy both source operands. This sequential register access increases the instruction latency by one clock cycle but reduces the number of physical register ports required. However, interfacing sequential register access with scheduling logic is not straight-forward: speculative scheduling does not allow variable-latency execution of instructions because dependent instructions cannot react to dynamic changes once they are scheduled [11]. To avoid this problem, the scheduling logic should be able to detect if a 2-source instruction will require one extra clock cycle for sequential register access, and to reflect it in the schedule for the following clock cycle.

The key observation is that where the instruction reads its source values is determined by when it is issued [4]. If the parent and dependent instructions are scheduled in back-to-back clock cycles, the dependent instruction is guaranteed to read the result value of the parent instruction off the bypass without accessing the register file; if the dependent instruction is not issued in the very next clock cycle, the value is no longer available from the bypass network and should be read from the register file. We conservatively assume only one clock cycle of "bypass window" (i.e. the result of a parent instruction is available on a

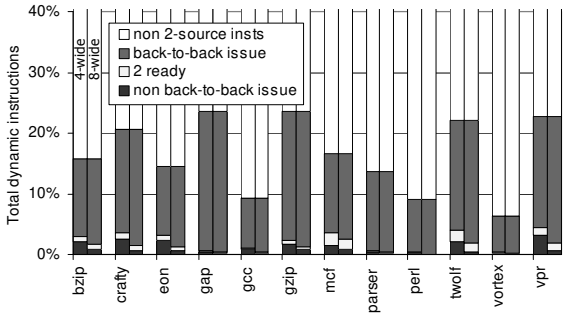


Figure 10. Characterization of register accesses. Two stacked bars from the bottom (*non back-to-back issue* + *2 ready*) show that less than 4% of instructions require two port accesses to the register file in most benchmarks.

bypass path for only a single cycle). This assumption could be relaxed to give a larger bypass window on machines with multi-cycle register file access, which requires additional bypass paths.

Figure 10 characterizes the register accesses of 2-source instructions. Most 2-source instructions are issued as soon as both operands become ready (shown as *back-to-back issue*). These instructions do not need two register ports since at least one source value is available off the bypass. Otherwise, instructions require two register reads when both source operands are ready at insert time (shown as *2 ready*), or they are not issued immediately due to structural hazards (shown as *non back-to-back issue*). The bottom two stacked bars show that less than 4% of dynamic instructions require two register file ports across all benchmarks.

4.3. Sequential register access

In Figure 11, we show wakeup and select logic that detects and schedules sequential register access. Back-to-back instruction issue is determined by simply looking at whether an instruction is awakened and selected in the same clock cycle. An issue queue entry shown in Figure 11a has single-bit *nowL/R* fields that are identical to *readyL/R* except that *readyL/R* are sticky bits while *nowL/R* fields stay set only during the cycle when tags match. If any of *nowL/R* fields is set, which implies that the value will be available off the bypass, the *seq_reg_access* signal is cleared, indicating that the instruction does not need two register ports. If the *seq_reg_access* signal is set when the instruction is selected and issued, it adds one extra clock cycle to the instruction latency (*delay* field) for sequential register access. Since a sequential register

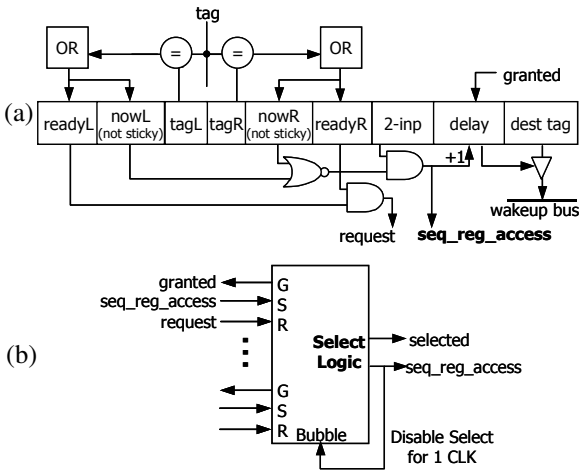


Figure 11. (a) the wakeup logic and (b) the select logic for sequential register access

access is a non-pipelined operation, the pipeline must be stalled so that the instruction accesses the register file twice. The select logic shown in Figure 11b creates a bubble to prevent other instructions from issuing through this issue slot by disabling itself for one clock cycle when an instruction with a *seq_reg_access* bit is selected. Since each issue slot has its own select logic, only the issue slot that requires sequential register access is blocked. Therefore, the penalty of sequential register access is 1 clock cycle latency + 1 issue slot, which can be interpreted as issuing an extra register read operation implicitly.

Figure 12 shows an example of sequential register access. Suppose the ADD instruction sequentially accesses the register file because both source operands are ready at insert time. At cycle 1, ADD is issued and the select logic is disabled for the next clock cycle. ADD will wake up the dependent instruction (SUB) one clock cycle later than the base case. At cycle 2, no instruction is issued following ADD instruction. At cycle 3, SUB is awakened and selected in the same clock cycle. Since ADD and SUB are scheduled in back-to-back cycles, the result value r3 of ADD is guaranteed to be read off the bypass and hence SUB does not require two register ports. At cycle 4, XOR with single source operand is issued and it reads r5 off the bypass at cycle 6.

Figure 13 illustrates the changes in the pipeline for sequential register access compared with a conventional pipeline. In this configuration, a single register read port is directly connected to both functional unit input ports. An issued instruction gets its opcode and physical register specifiers from the payload RAM in the base pipeline. In

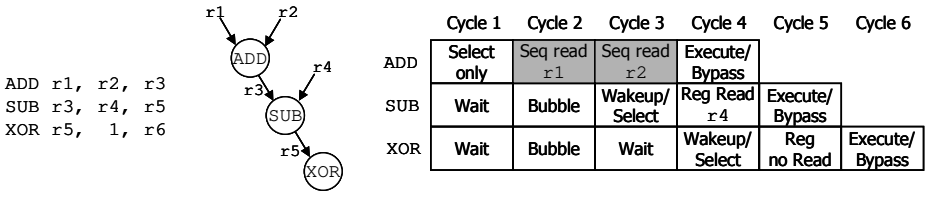


Figure 12. An example of sequential register access. Only three stages (Sched, RF and EXE) are shown for the simplicity of presentation.

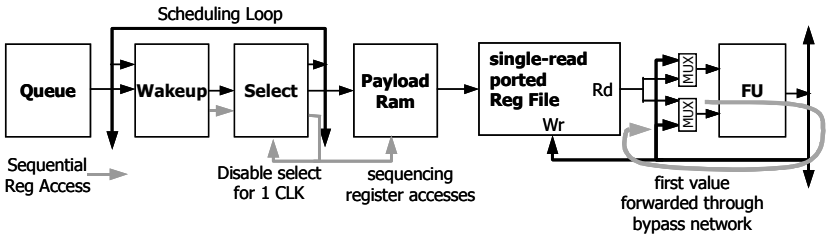


Figure 13. The changes in the pipeline for sequential register access. Unlike an conventional pipeline with 2 register read ports per issue slot, the register file has only one register port per issue slot.

addition, our technique requires the payload RAM to sequence register accesses when an instruction with *seq_reg_access* signal is issued. In order for both source values to arrive at the functional unit simultaneously when the register file is accessed twice, the pipeline needs storage for the first source value until the other source value is accessed. This can be accomplished by either adding latches to the input ports of the functional unit, or forwarding a source value accessed earlier through the bypass to the functional unit input port again, as if a move instruction is executed. Since the functional unit is idle when the first source value arrives, the functional unit can easily put the value on the bypass. In the next clock cycle, the instruction starts execution when both source values, from the register file and the bypass logic, arrive at the functional unit.

Performance evaluation of sequential register access which eliminates half of register-read ports will be presented in Section 5.2 and also combined with sequential wakeup in Section 5.3.

5. Performance Evaluation

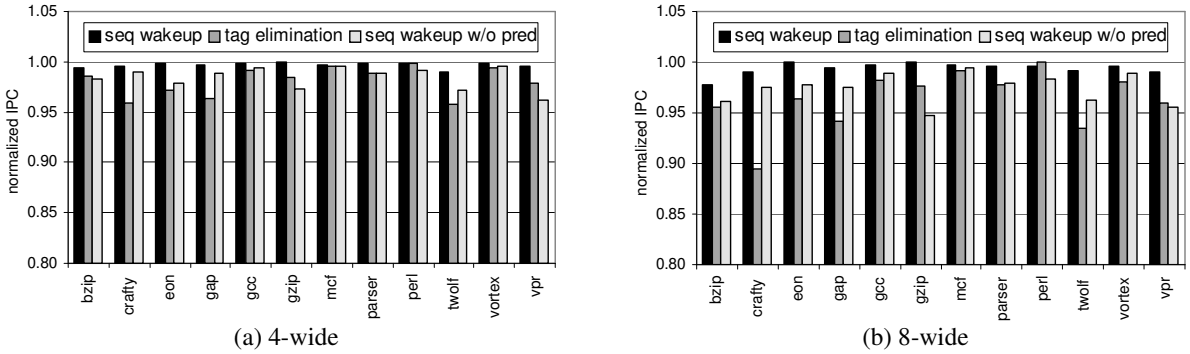
5.1. Evaluation of sequential wakeup

Figure 14 presents IPCs of the sequential wakeup and tag elimination schemes discussed in Section 3, normalized to the base performance. The base IPCs on 4 and 8-wide machines are presented in Table 1. For the last-arriving operand predictor, a 1k-entry, direct-mapped bimodal predictor is used for both sequential wakeup and tag elimination schemes.

The performance degradation due to sequential wakeup with a last-arriving operand predictor, shown in the left bars, is measured to be on average 0.4% and 0.6% on 4 and 8-wide machines respectively. These slight IPC drops come from at most 4% of dynamic instructions that experience the issue penalty from operand mispredictions and simultaneous wakeups across all benchmarks. Since mispredicted instructions do not consume issue bandwidth, mispredictions can be overlapped by other ready instructions that take the issue slot instead.

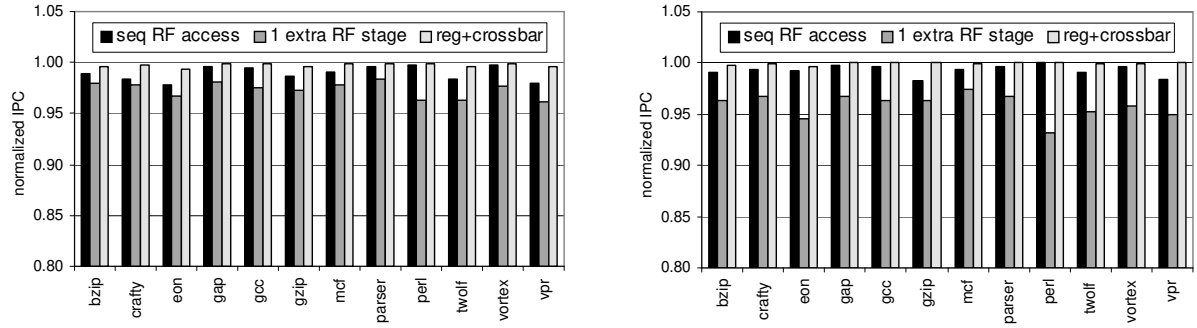
As a point of reference, the performance of the tag elimination scheme [10] is shown in the middle bars. We use the same last-arriving operand predictor as the one used for sequential wakeup, and assume that the scoreboard is located right next to the schedule stage in order to minimize the detection delay. Although the performance degradation is not drastic, it is measured to be worse than sequential wakeup in most benchmarks, and does not scale well with increasing misprediction penalty, especially on an 8-wide machine (worst case 10.6% in *crafty*). Although the tag elimination scheme is not restricted by simultaneous wakeups, the misprediction penalty in non-selective recovery is so severe that it significantly affects performance.

Interestingly, sequential wakeup without a last-arriving predictor, as shown in the right bars, outperforms the tag elimination scheme with a predictor in many benchmarks. In this configuration, the right-hand side source operands are statically assumed to be last-arriving, which has less than a 50% chance to be correct on average (as shown in Table 3), causing up to 10% of dynamic instructions to experience issue penalties. However, the average performance degradation is still measured to be 1.6% and 2.6% on 4 and 8-wide machines. This result clearly shows that the performance of sequential wakeup is relatively insensitive to the predictor accuracy because the misprediction



(a) 4-wide (b) 8-wide

Figure 14. Performance of sequential wakeup. Note that sequential wakeup without a predictor outperforms the tag elimination scheme with a predictor in many benchmarks, showing that the misprediction penalty in sequential wakeup is slight.



(a) 4-wide (b) 8-wide

Figure 15. Performance of sequential register access. From left to right, sequential register access with one read port per issue slot (*seq RF access*), a conventional register file with one extra pipeline stage (*1 extra RF stage*), and a register file with half the number of read ports + a fully connected crossbar to all functional units (*reg + crossbar*).

penalty is very small.

In summary, the sequential wakeup scheme enables reducing the wakeup bus load capacitance with a minimal performance impact.

5.2. Evaluation of sequential register access

The characterization of register file accesses and the structure of the sequential register access logic are discussed in Section 4. Figure 15 presents IPCs from various register file configurations normalized to those on the 4 and 8-wide base machines. Instructions that require sequential register accesses are presented in Figure 10.

Performance of sequential register access is presented in the left bars in Figure 15 (*seq RF access*). Overall, degradation on a 4-wide machine is slightly greater than on an 8-wide machine because the narrower issue bandwidth leads to more structural hazards that prevent 2-source instructions from issuing in consecutive clock cycles. The degree of slowdown is in general correlated to the number of sequential register accesses, and *bzip, crafy, eon, gzip, twolf* and *vpr* exhibit noticeable performance drops on a 4-wide machine, as up to 4% of instructions experience the penalty of 1 clock cycle + 1 issue slot for two register port accesses. However, the worst performance degradation is observed to be only 2.2% in *eon*. On average, 1.1% and 0.7% IPC drops on 4 and 8-wide machines are measured.

The middle bars in Figure 15 (*1 extra RF stage*) show performance of a conventional pipelined register file with two read ports per issue slot when one extra pipeline stage is added for a shorter access latency. Although it does not show drastic performance drops compared to the base machine performance, this design may not be attractive since an extra register access stage requires another level of internal bypass paths within the register file.

The right bars (*reg + crossbar*) show performance from the same half-read-ported register file used in sequential register access with a fully-connected crossbar to all functional unit inputs, which is similar to a single-bank configuration with limited register read ports proposed in [4]. Although this scheme achieves most of base performance since the actual port requirement rarely exceeds the issue bandwidth, it incurs mux delays for the crossbar, and complexity in the select logic to arbitrate contention in register ports across all issued instructions. A common optimization in conventional schedulers to ease implementation is having separate select logic for each execution pipeline or cluster [2][12][20]. We emphasize that this arbitration for register file ports cannot be carried out locally (within a single scheduler) but must be across all issued instructions. Sequential register access, on the other hand, does not require such a crossbar or global arbitration logic (arbitration for the single physical register port is a local scheduler decision, as described in Section 4.3) while it shows comparable performance in many benchmarks.

In conclusion, we see that sequential register access enables reducing the number of register read ports with a minimal impact on performance.

5.3. Evaluation of combined techniques

In this section, we discuss the performance impact of combining both sequential wakeup and sequential register access. The combined techniques require a minor change in the wakeup logic: the wakeup logic does not require the *nowR* field in Figure 11a, and only *nowL* is able to clear the *seq_reg_access* signal. Although these techniques are orthogonal and no substantial complication is involved, negative interference may occur when both techniques are performed together because the issue penalty from simultaneous wakeups and last-arriving operand mispredictions always forces 2-source instructions to sequentially access the register file. The combined penalty is 2 clock cycles of delay + 1 issue slot.

Figure 16 shows the performance of the combined techniques on 4 and 8-wide machines with a 1k-entry, direct-mapped last-arriving operand predictor as used in Section 5.1. As expected, the performance degradation from the combination of sequential register access and sequential wakeup is slightly worse than the direct sum of performance drops from the techniques applied individually. However, the overall performance degradation of the combined techniques is not drastic, and the worst case (*bzip* on an 8-wide machine) performs within 4.8% of the baseline. On average, 2.2% performance degradation is observed.

In summary, our proposed half-price architecture reduces complexity in the wakeup logic and the register file by using narrower structures and restricting the pipeline's capability to handle 2-source instructions, resulting in a minimal impact compared to a conventional pipeline

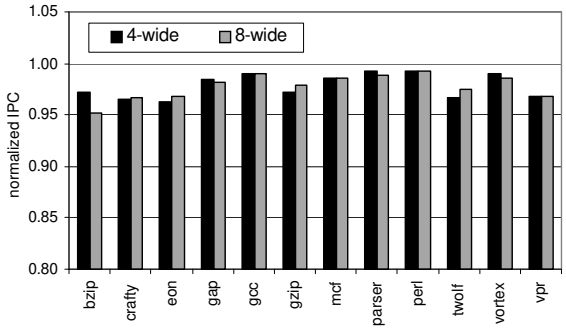


Figure 16. Performance of sequential register access combined with sequential wakeup

that fully handles two source operands. At the same time, it enables higher-frequency implementations of both the wakeup logic and register file by reducing the size and easing the physical design of these structures.

6. Conclusion & Future work

We make four main contributions in this work. First, we described the pipeline stages that introduce overdesigned hardware structures due to instructions with multiple source operands. Second, we characterize the dynamic behavior of instructions with two source operands and present that only a small fraction of instructions require their two source operands to be processed in the wakeup logic and the register file. Third, we introduce the half-price architecture that reduces hardware complexity by limiting the processor's capability of handling instructions with two source operands. Fourth, we propose two complexity-effective techniques: sequential wakeup and sequential register access, and demonstrate that these techniques can significantly reduce the complexity in the wakeup logic and the register file with a minimal IPC degradation.

Although we discuss only two major components of the processor in this work, the basic concept of half-price architecture can extend to all pipeline stages that handle two source operands. We are developing half-price techniques for register renaming, ready information check and bypass logic. The final goal of half-price architecture is to achieve an *operand-centric* processor design that enables a higher clock frequency with a minimal performance impact by eliminating redundancy incurred by instruction-centric design.

7. Acknowledgements

This work was supported in part by the National Science Foundation with grants CCR-0073440, CCR-0083126, EIA-0103670 and CCR-0133437, and generous financial support and equipment donations from IBM and Intel. We especially thank Kevin Lepak for his insights and comments on this work. We would also like to thank the anonymous reviewers for their many valuable comments.

8. References

[1] D. C. Burger and T. M. Austin, *The Simplescalar tool set, version 2.0*, Technical Report CS-TR-97-1342, University of Wisconsin, Madison, June 1997.

[2] Compaq Computer Corporation, *Alpha 21264 microprocessor hardware reference manual*, July 1999.

[3] P. Michaud and A. Seznec, *Data-flow prescheduling for large instruction windows in out-of-order processors*, in Proc. of 7th International Symposium on High Performance Computer Architecture, 2001.

[4] R. Balasubramonian et. al, *Reducing the complexity of the register file in dynamic superscalar processors*, in Proc. of 34th International Symposium on Microarchitecture, 2001.

[5] J. L. Cruz, A. Gonzalez, M. Valero and N. P. Topham, *Multiple-banked register file architecture*, in Proc. of 27th International Symposium on Computer Architecture, 2000.

[6] B. Black and J. P. Shen, *Scalable register renaming via the Quack register file*, Tech report, CMuART-2000-01, Carnegie Mellon University, Pittsburgh, 2000.

[7] M. Tremblay, B. Joy and K. Shin, *A three dimensional register file for superscalar processors*, in Proc. of 28th Hawaii International Conference on System Sciences, pp. 191-201, 1995.

[8] K. I. Farkas, N. P. Jouppi and P. Chow, *Register file design considerations in dynamically scheduled processors*, WRL Research Report 95/10, 1995.

[9] J. Hoogerbrugge and H. Corporaal, *Register file port requirements of transport triggered architectures*, in Proc. of 27th International Symposium on Microarchitecture, 1994.

[10] D. Ernst and T. M. Austin, *Efficient dynamic scheduling through tag elimination*, in Proc. of 29th International Symposium on Computer Architecture, 2002.

[11] I. Kim and M. H. Lipasti, *Implementing optimizations at decode time*, in Proc. of 29th International Symposium on Computer Architecture, 2002.

[12] S. Palacharla, N. P. Jouppi and J. E. Smith, *Complexity-effective superscalar processors*, in Proc. of 24th International Symposium on Computer Architecture, 1997.

[13] A. R. Lebeck et. al, *A large, fast instruction window for tolerating cache misses*, in Proc. of 29th International Symposium on Computer Architecture, 2002.

[14] M. S. Hrishikesh et. al, *The optimal useful logic depth per pipeline stages is 6-8 FO4*, in Proc. of 29th International Symposium on Computer Architecture, 2002.

[15] W. A. Wulf, *Compilers and computer architecture*, IEEE Computer, 14 (8):41-47, 1981.

[16] A. KleinOsowski, J. Flynn, N. Meares and D. J. Lilja, *Adapting the SPEC2000 benchmarks suite for simulation-based computer architecture research*, Workshop on Workload Characterization in International Conference on Computer Design, 2000.

[17] J. Stark, M. Brown and Y. Patt, *On pipelining dynamic instruction scheduling logic*, in Proc. of 33rd International Symposium on Microarchitecture, 2000,

[18] M. Brown, J. Stark and Y. Patt, *Select-free instruction scheduling logic*, in Proc. of 34th International Symposium on Microarchitecture, 2001.

[19] P. Shivakumar and N. P. Jouppi, *CACTI 3.0: an integrated cache timing, power, and area model*, WRL Research Report 2001/2, 2001.

[20] J. M. Tendler, S. Dodson, S. Fields, H. Le and B. Sinharoy, *POWER4 system microarchitecture*, IBM Server Group Technical White Paper, 2001.

[21] B. Fields, S. Rubin and R. Bodik, *Focusing processor policies via critical-path prediction*, in Proc. of 28th International Symposium on Microarchitecture, 2001.

[22] E. Tune, D. Liang, D. M. Tullsen and B. Calder, *Dynamic prediction of critical path instructions*, in Proc. of 7th International Symposium on High-performance Computer Architecture, 2001.

[23] I. Park, M. Powell and T. Vijaykumar, *Reducing Register Ports for Higher Speed and Lower Energy*, in Proc. of 35th International Symposium on Microarchitecture, 2002.