Appears in MICRO 2002

Reducing Register Ports for Higher Speed and Lower Energy

Il Park, Michael D. Powell, and T. N. Vijaykumar
School of Electrical and Computer Engineering, Purdue University
{parki, mdpowell, vijay}@purdue.edu

Abstract

The key issues for register file design in high-performance processors are access time and energy. While previous work has focused on reducing the number of registers, we propose to reduce the number of register ports through two proposals, one for reads and the other for writes. For reads, we propose bypass hint to reduce register port requirements by avoiding unnecessary register file reads for cases where values are bypassed. Current processors are unable to avoid these unnecessary reads due to timing constraints. For writes, we use register file banking. Current banking schemes assign different banks to instructions that are renamed together, which does not necessarily avoid conflicts among instructions that writeback together. We use decoupled rename, a technique which separates dependence and physical tagging of register operands. Decoupled rename allows us to perform physical register allocation just before writeback, avoiding bank conflicts. Our results show that combining bypass hint and write banking, our 1-cycle register file with 6 read ports, and two 4-write-ported banks achieves a 9% processor energy-delay savings over a system using a perfectly-pipelined, 2-cycle register file with 16 read ports and 8 write ports.

1 Introduction

In high-performance, wide-issue processors, the register file is in the critical path for clock speed and accounts for a significant fraction of overall processor energy. Register files are large and multiported to keep multiple instructions in flight and to enable wide issue. Because of cycle time impact, Alpha 21264 split the register file and pipeline back-end into two parts [9]. Both large size and high numbers of ports result in slow access and high energy dissipation.

The key difference between previous work and ours is that previous research on register files focuses on reducing the *number of registers* (either via 2-level register files [12, 1] or via caching [3]*),* and we focus on reducing the *number of ports*. Previous research on reducing the number of registers has required substantial changes to the pipeline such as searching the active list every cycle [1], or storing register values in the issue queue and handling coherence in register caches [3]. These changes create significant complications, which we discuss in the next section. In contrast, reducing the number of ports is a simpler solution. We reduce register file port requirements through two proposals, one for read ports and the other for write ports.

For the first proposal, we observe that modern pipelines *anticipatorially* read the register file to obtain instructions' source operands, *before* ascertaining that the source register value comes from the register file and not bypass paths. About 50-70% of register operands come from bypass paths, and the corresponding anticipatorially read values are discarded, wasting substantial bandwidth and energy. Previous research [1] has assumed that bypass operands can be determined prior to reading the register file. However, processors such as the Pentium 4 [6] and Alpha 21264 [14] determine bypass condition during the register read stage while anticipatorially reading the register file.

There are two options for reducing anticipatory register reads and consequently reducing demand for read ports. The first option is straightforward and identifies bypass operands in an extra pipeline stage inserted between out-of-order issue and register read. For the second option we propose a novel technique, *bypass hint,* based on the observation that when a consumer's source operand is woken up in the issue queue by the producer's result broadcast, the consumer often issues *before* the producer writes back. Consequently, the consumer obtains the operand via bypass except for the cases in which it is delayed due to structural hazards or unavailability of another operand. By leveraging off wake-up logic, bypass hint isolates operands that are woken up by in-flight producers from the operands whose producers have written back. Because the exceptional cases are rare, bypass hint is quite accurate. With virtually no extra hardware or extra stage delay, bypass hint avoids anticipatory reads, enabling a faster, lower-energy register file. Thus, while previous techniques [1, 12, 3] use banking and caching to increase bandwidth supply, we opt to reduce demand.

Reducing the number of read ports introduces the problem of mapping instructions to ports. Conventional pipelines, however, do not have this mapping problem because they provide enough register ports for a one-to-one hard-wired mapping between register file ports and functional units. Because bypass hint is available at the same time as wakeup *before* select occurs, select can be augmented to map instructions to functional units and ports, *in parallel*. Unlike complications caused by previous techniques, bypass hint's timing makes our solution simpler to implement.

While read bandwidth demand can be reduced, we found

that write bandwidth demand cannot be easily reduced and optimizing writes requires more changes. We propose to supply high write bandwidth at low cost via banking. While banking is a well-known bandwidth technique, the novelty is in our second proposal for reducing the inevitable bank conflicts. Conventional banking performs round-robin allocation among the instructions in rename, hoping to evenly distribute bank assignments to avoid conflicts at read and writeback. This strategy is ineffective: instructions that rename in the same cycle are often different from instructions that read or write back in the same cycle due to the dynamic nature of out-of-order execution and variable-latency instructions. Our solution is that while conventional rename uses the same mapping for dependence tags and physical registers, we *decouple* the two. We assign dependence tags as usual in rename, but assign physical registers in a round-robin fashion (perhaps different than dependence tags) in the ample time between issue and writeback, avoiding conflicts among writes. [8] proposes such decoupling to reduce the number of registers but not bank conflicts.

[1, 12] use banking for both reads and writes. Because bypass hint sufficiently reduces read bandwidth demand, we use banking for writes but not for reads; for reads, we use true ports and bypass hint to reduce the demand for those ports.

The main results of this paper are:

- On average, the SPEC2K benchmarks obtain about 66% of source operands from bypasses; bypass hint correctly predicts 98.5% of the bypassed operands. Using only 6 read ports, bypass hint reduces read energy-delay by 66% with 1.8% performance degradation compared to an unrealizable, 16 read-port, 1-cycle register file.
- Using 2 banks for write banking (4 write-ports per bank) results in little performance degradation. While increasing the number of banks reduces energy under the constraint of maintaining the same number of total ports, the constraint forces fewer ports per bank causing more bank conflicts. Decoupled rename effectively alleviates this conflict problem. 4 banks (2 write-ports per bank) with decoupled rename has energy-delay savings of 65% compared to 56% for two banks. Using 4 banks without decoupled rename degrades performance by 7%; with decoupled rename degradation is only 3%.
- The previously-proposed 2-level register file has a performance degradation of 3.1% assuming 1-cycle searches of the active list, but *increases* energy-delay by 115%. In contrast, bypass hint has a performance degradation of 1.8% with a register file energy-delay savings of 61%
- Combining bypass hint and write banking, our 1-cycle register file with 6 read ports, and two 4-write-ported banks achieves a 9% processor energy-delay savings over a system using a perfectly-pipelined, 2-cycle register file with 16 read ports and 8 write ports.

In the next section, we contrast our techniques from previous proposals. In Section 3, we describe techniques for reducing register read port demand, and increasing write supply. In Section 4 we discuss access time and energy impact of our techniques. Section 5 covers experimental methodology and Section 6 contains experimental results. We conclude in Section 7.

2 Related work

2.1 Complications with previous proposals

Reducing the number of registers decreases register file access time by reducing bitline delay but introduces substantial complications to the pipeline. While [12] proposes the possibilities for reducing the number of registers in the abstract, [1] discusses the implementations details. Because it is difficult to compare performance and energy in the abstract, we contrast our ideas against [1]. The 2-level register file proposed in [1] places the registers that will not be reused, except in the event of misspeculation, in the second level. Upon misspeculation, this technique requires searching for and replacing moved registers.

In order to identify which register operands will not be reused, the technique requires that during register renaming the entire active list be searched to match use of physical registers in L1 (tracked in the usage list) with existing active instructions. Searching the active list every cycle adds significant complication to the out-of-order pipeline. The reason for separating yet-to-issue instructions in the small issue queue and all in-flight instructions in the large active list is to avoid searching the active list. As such, conventional active lists are non-searchable structures, which facilitates high clock speeds and avoids wiring complexity. The 2-level register file also uses a copy list that maintains information needed to restore entries from L2 to L1 after misspeculation. While misspeculation rollback is already a complex aspect of conventional pipelines, the additional restoration of register state required by the 2-level register file requires more resources, and will both increase energy and time for the recovery.

Register caching, as proposed in [3], maintains small register caches close to the functional units and reads register operands from one of four sources: 1) pre-read before the issue queue, 2) extended forwarding (bypass) logic, 3) register caches, and 4) the register file after a register cache miss. Because operands may be read before the issue stage, their technique requires storing operands in the issue queue. Storing register values in the issue queue increases the size of issue queue entries as register values are sub-

stantially larger than the dependence tags stored in conventional issue queues. The additional area and wiring required for such an issue queue are primary reasons why modern issue queues do not store register values.

If the operand is not pre-read into the issue queue, [3] next tries to obtain it from bypass logic or the register cache. The operand may not be present (i.e. a register cache miss), or it may be invalid (i.e. a coherence miss). In either case, the valid operand is not available in the register cache. The instruction issues and produces an invalid result. The correct register value must be brought to the register cache, and the instruction must be reissued. It takes several cycles to identify register cache misses, and stalling issue for these many cycles to ensure a valid operand would degrade performance. Therefore, the authors assume the capability to reissue selectively only the instructions dependent on the unavailable operand. Supporting selective reissue would add substantial complexity to the issue queue. Many conventional pipelines do not implement selective reissue for similar circumstances. For example upon a load miss many conventional designs simply squash and reissue all instructions between issue and cache stages.

Another register file technique is banking, which is an inexpensive means of supplying high bandwidth. [1, 12] use banking for both reads and writes. For register reads, reducing the demand for ports is more desirable than supplying bandwidth through banking. Register read bank conflicts must be resolved *before* reads are initiated. Bank conflict detection must occur in an additional pipeline stage inserted between issue and register read. However, the additional stage can only *detect* bank conflicts and stall offending instructions; it cannot *prevent* conflicts because the instructions have already been issued. Prevention of read bank conflicts would require augmenting the select logic in the issue stage to map instructions to specific banks, *in addition* to mapping instructions to functional units. For example, if there were four banks with two read ports each, an individual register would be accessible from only two ports. The select logic would be required to select and map up to eight instructions with sixteen source registers to the correct banks to avoid conflicts. Conceptually, the select logic would need to treat each bank as a functional unit and ensure that the number of instructions reading a bank does not exceed the number of read ports per bank, much as the select logic ensures that no more than 4 add instructions are selected if there are only 4 ALUs. Of course, this change implies that the select logic is more complex and may impact the clock speed [13].

2.2 Our proposals

Unlike previous schemes, our proposals do not introduce complications like searching the active list, storing operands in the issue queue, and selective reissue. Table 1 summarizes previous proposals and our techniques. Much like read banking needs to map instructions to banks, reducing read ports introduces the problem of mapping instructions to ports. Bypass hint's timing helps us solve the mapping problem without adding complications to the pipeline, as we explain in Section 3.2. Apart from the mapping problem, there are two special cases in our proposals: bypass hint misprediction and write bank conflicts in decoupled renaming. For these special cases, we advocate simple stalls of the relevant back-end stages. Because these special cases are infrequent and resolving them takes only a cycle, stalling incurs little performance loss. For simplicity, we stall entire stages and not specific instructions. Because we advocate completely stalling the issue stage (and later stages, if need be), there are no complications with the out-of-order scheduler, as we explain in Section 3.2 and Section 3.3.

It is worth noting that proposals such as the 2-level register file and register caching for reducing the number of registers may be implemented orthogonally to our proposals for reducing the number of ports. However, because our proposals adequately reduce the register file size without the pipeline complications discussed above, combining the techniques may be unnecessary.

2.3 Other related work

Decoupled rename was proposed in [8] to reduce the number of registers but not bank conflicts. Other register file proposals include compiler-controlled two-level register files for VLIW processors [17]. Several non-hierarchical, partitioned register files have been proposed to support clusters of functional units [5,7]. There are a few circuit-level techniques for low-energy register files. For ultra low energy dissipation, the NRERL register file is an adiabatic register file clocked at less than 1 MHz [11]. In [16], the

Table 1: Register file technique pipline interaction summary: previous proposals (left) and ours (right).

Previous proposals	Ours
2-level:	**Bypass hint:**
Rename: Search active list to update usage	Issue: Bypass bit matches bypass operands. Map reads
Rollback: Search copy list to restore L2 entries to L1	Register read: stall if inadequate b/w
Register Caching:	**Extra bypass-check stage:**
Issue: Pre-read operands and store in issue queue.	Extra stage: identify bypass, map ports, stall if needed
Selectively reissue instructions that miss and successors	**Write banking via Decoupled Renaming:**
Read/Write banking:	Added stage: non-bypass operands map to physical tags
Reg read: added stage maps ports, ids conflicts, stalls	Mem: Non-conflicting assignment of physical tags

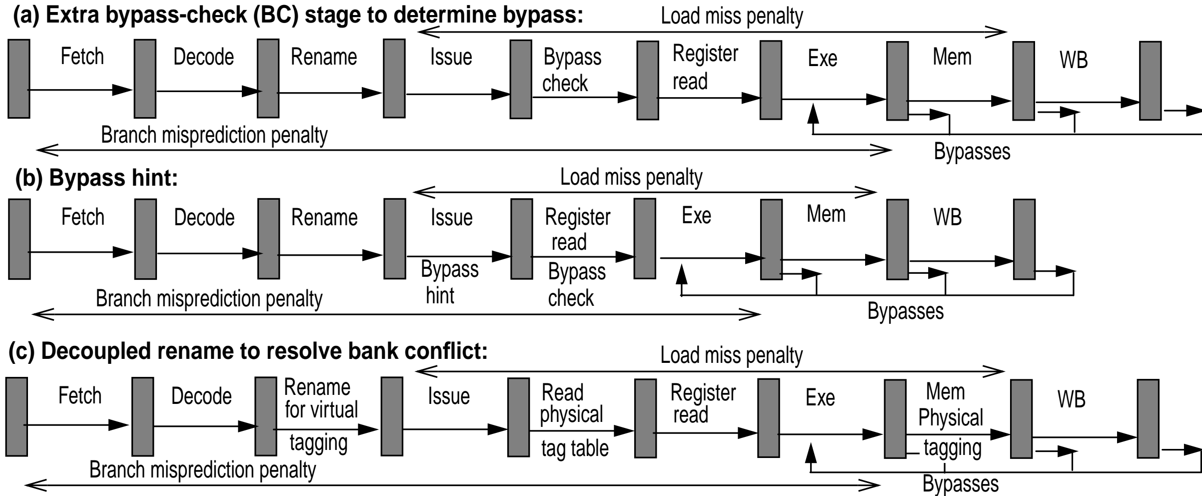

FIGURE 1: Out-of-order pipelines: (a) extra stage to determine bypass, (b) bypass hint, and (c) decoupled rename to resolve bank conflicts.

authors present several circuit-level techniques to reduce energy dissipation. The paper avoids register reads for bypassed values by determining the bypass conditions and *then* accessing the register file for non-bypassed values, *within* the same stage. The paper considers only a simple in-order-issue, five-stage pipeline, and cannot be applied to wide-issue, out-of-order pipeline where determining bypass may require an additional stage due to high clock speeds.

3 Reduce read demand; increase write supply

In the register read stage, the conventional pipeline knows *what* instructions are issued but does not know from *where* they will obtain the source operands. Bypass conditions are determined using destination register information from in-flight instructions, and at the same time the potential (non-bypass) operands are anticipatorially read from the register file. Note that we show bypasses from the end of writeback because the register file in wide-issue pipelines is too big to be written in the first half cycle and read in the next half cycle, as suggested by DLX [10].

3.1 Extra stage for bypass determination

One solution to avoid anticipatory reads is to separate bypass checking and register read, as shown in Figure 1(a). Bypass conditions are determined in an extra stage, called bypass-check (BC) stage, between issue and read. Only non-bypass operands are read from the registers in the read stage. Thereby, (1) the bandwidth demand on the register file reduces, enabling faster register with significantly fewer ports, and (2) no energy is wasted on unnecessary anticipatory reads.

BC has obvious performance concerns. Separating bypass check and register read into two stages increases the branch misprediction penalty. Similarly, the squash penalty incurred during a load miss increases [3]. The performance penalties of BC are similar to those of a perfectly pipelined register file. However, a pipelined register file extends writeback into multiple stages and introduces the substantial additional complexity of requiring additional bypass paths.

Apart from the performance concerns, BC interacts with the pipeline in other ways. In conventional pipelines, because there are as many ports as the issue width, there is a one-to-one hardwired mapping between read ports and functional units, as shown in Figure 2(a). The select logic maps selected instructions to functional units *without* regard for availability of read ports. Because of the hardwiring, the read ports are guaranteed to be available.

Reducing the number of ports violates this one-to-one mapping and raises four issues. The first issue is that the issued instructions have to be steered to the read ports via muxes. Because bypass conditions are determined in the BC stage *after* select, select is unaware of the bypass conditions and cannot map the instructions to the ports. Upon issue, any of the instructions may need to access the ports, and the mapping of instructions to ports can be determined only after the bypass conditions are known in the BC stage. Therefore, the muxes may have to be n-to-1 muxes for an issue width of n. It is likely that the substantial reduction in the number of register file ports makes up for the space and time overhead of these muxes.

The second issue is that select cannot avoid issuing instructions requiring more read ports than are available. There are two options to deal with the excess instructions. One is to stall both the excess instructions and the issue stage, and use the extra muxes to map the excess instructions to read ports in the next cycle when no more instructions are coming from issue. The other is to squash the

excess instructions and reissue, similar to load miss squash and reissue in modern pipelines. For the first option to be viable, the bypass conditions have to be determined well before the next clock edge so that issue can be stalled from pushing more instructions into register read in the next cycle. This timing requirement is likely to be easily met because bypass condition determination entails equality-checks of physical register tags and can be done fast. When stalling issue, the structure monitoring functional unit schedules must also be stalled to maintain correct state regarding functional unit availability. The second option may not be desirable because instead of stalling the excess instructions for just a cycle, several cycles (e.g., 5 cycles between issue and writeback) are lost in squash.

The third issue is that even if the number of read ports are adequate for a set of instructions issued, some control logic has to map the instructions to the read ports. Only those instructions that obtain their operands from the register file, and not from bypass, need to be steered to the ports. The other instructions need not. Again, because the bypass conditions are determined later in the BC stage, this mapping cannot be done in select. Instead, control logic for this mapping has to be built into the read stage.

The fourth issue is that the output of each read port has to be fanned-out to multiple functional units (in reality, to muxes choosing between bypass inputs and register inputs in front of the functional units) to accommodate the excess instructions. This requirement adds wires, but not muxes.

3.2 Bypass hint

The lack of knowledge of the bypass conditions in select is a major drawback of BC. This drawback is overcome by bypass hint, which is an alternative means of eliminating anticipatory register reads. Bypass hint is based on the fundamental observation that when a consumer's source operand is woken up in the issue queue by the producer's result broadcast, the consumer often issues *before* the producer writes back. Bypass hint avoids BC's performance penalties.

We use one extra bit, called the *bypass bit*, per source operand in the issue queue slot. At the time of instruction issue, the bypass bit indicates whether an operand had been woken up while the instruction was waiting in the issue queue or whether the operand was ready even before the instruction entered the issue queue. If an operand is ready at the time the instruction enters the issue queue, its bypass bit is clear. At the time an operand is woken up, the bypass bit is set *in parallel* with the setting of the conventional operand-ready bit. Thus, there is no extra delay associated with setting of the bypass bit, as indicated by bypass hint being in parallel with issue in Figure 1(b).

The select logic sees the bypass bits and selects instructions so that the number of read ports needed do not exceed the number of ports available. Essentially, some of the control logic that maps instructions to ports (described in Section 3.1) is absorbed into select. This mapping is a little simpler than that of read banking (end of Section 2.1). Because bypass hint employs ports instead of banks, select logic has to merely restrict the total number of source operands to less than the total read ports. Hence, select can treat all of the register file as one type of functional unit, as opposed to treating each bank as one type of functional unit. This difference is a minor advantage for bypass hint over read banking if the number of banks is large.

Even if bypass hint were perfectly accurate, the selected instructions have to be steered to the ports via muxes, exactly as in BC. The only difference is that because bypass hint is known earlier than in BC, select can separate instructions into "no or partially bypassed" and "fully bypassed" categories. This separation helps in reducing the size of the muxes. For example, in Figure 2(b), we show mapping 4 instructions into 3 ports. Assuming bypass hint is perfect and these 4 instructions do not need more than 3 ports, the combinations in which 3 ports would be needed by 4 instructions are: (a) 1 instruction needs 2 ports, another needs 1, and the rest need no ports. (b) 3 or fewer instructions need 1 port each, and the rest need no ports. In either case, at least one instruction needs no ports, so select can separate out that instruction under "fully bypassed", as shown in the far right in Figure 2(b). Similarly the rest of the instructions can be separated as shown in the figure. The solid lines in the figure show the muxing for the above combinations. Thus, if bypass hint were perfect, only 2-to-1 muxes are needed in our example.

In Section 6, we show that 6 ports are sufficient for 8-way issue. In that case, the design would include 2 replicas of our example, and 2-to-1 muxes would still suffice. In reality, however, we need 3-to-1 muxes to handle bypass hint mispredictions, but still the mux overhead is reasonably low.

Upon issue, instructions read the register file for only those operands that have their bypass bit clear. As in conventional pipelines, the bypass control logic determines the real bypass conditions for the operands, in parallel with the register reads. The control logic detects mispredictions corresponding to the rare exceptional cases when a source operand is not bypassed but the bypass bit is set. These cases occur under two possibilities: (1) The other source operand(s) of the instruction became ready much later, by which time the previously-woken-up operand has already been written back to the register file (i.e., the operand will not be bypassed). (2) Although all source operands have been previously woken up, the instruction is not issued due to structural hazards. For both possibilities, although the previously-woken-up operands have their bypass bits set, the operand values should be read from the register file.

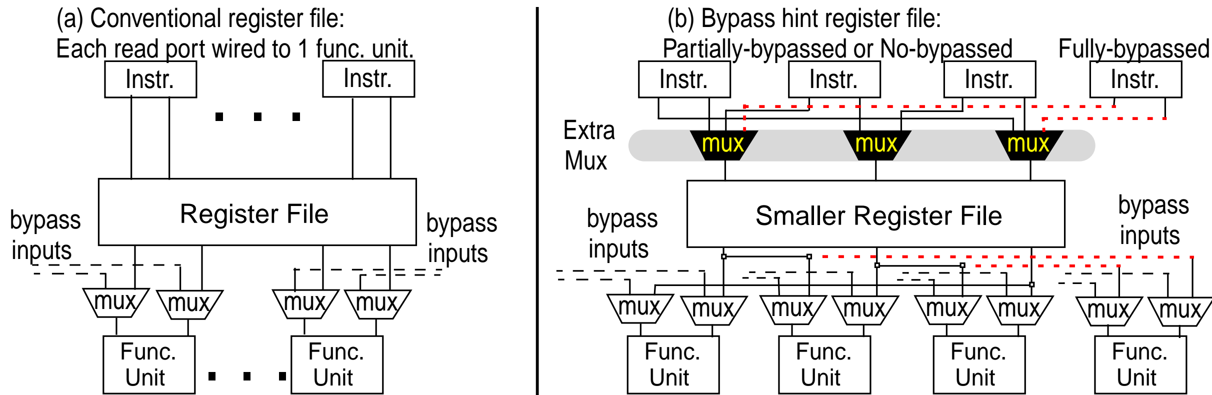

FIGURE 2: Wiring for read ports: (a) conventional register file. (b) bypass hint register file.

Fortunately, as the real bypass conditions are determined in parallel with register read, such mispredictions can be detected in the register read stage, as shown by bypass check being in parallel with register read in Figure 1(b).

Bypass hint mispredictions correspond to excess instructions in BC. Unlike BC's excess instructions, bypass hint mispredictions are infrequent. Also, the mispredictions are detected early in register read. Hence, simply stalling issue, like BC, works without much performance loss here as well. As before, when stalling issue, the structure monitoring functional unit schedules must also be stalled to maintain correct state regarding functional unit availability. Much like the excess instructions, the mispredicted instructions are steered through the muxes in the next cycle. It is this steering that requires the muxes to be 3-to-1. Figure 2(b) shows this steering in broken lines. Note that the instructions to functional unit mapping done by select remains the same for mispredicted instructions.

It is also possible that a source operand is bypassed but the bypass bit is clear. This possibility occurs due to the operands whose producer has already done wake-up before the consumer enters the issue queue, but the producer has not written back by the time the consumer is issued. Such operands enter the issue queue with their operand-ready bit set, and are not woken up while in the issue queue. Consequently, such an operand has its bypass bit clear, even though the operand is bypassed. However, as in a conventional pipeline, the bypass control logic determines that the operand should come from bypass and discards the register value. This lost opportunity is insignificant in practice.

3.3 Decoupled rename for write banking

Reducing write demand is difficult. It may seem that register value death may be exploited for this purpose. Many values are dead even before the instruction producing them reaches writeback (i.e., the values are consumed via bypasses and are not read from the register file). It may seem that such dead values need not be written back, allowing energy savings.

Unfortunately, register value death determination is speculative because of branch mispredictions. A register getting overwritten does not guarantee death because the overwriting instruction may be the result of a misprediction. Consequently, registers cannot be deemed to be dead until all previous speculations are validated. Deeming a register dead before such validation will require resurrecting the dead value, incurring complications. It is such resurrection that forces [1] to search the active list. Therefore, we do not pursue this idea.

Instead, we utilize banking to supply write bandwidth for the register file. Conventional banking performs round-robin allocation among the instructions in rename. Instructions that rename in the same cycle are often different from instructions that read or write back in the same cycle, rendering this strategy ineffective. We use *decoupled rename* for write banking to reduce bank conflicts. [8] proposes such decoupling to reduce the number of registers but not bank conflicts.

Conventional rename combines the dependence tag and physical register number to be the same. Decoupled rename separates the dependence tagging of operands performed at the rename stage from the physical register assignment which determines banks. By delaying physical register assignment until just before writeback, it is possible to make non-conflicting assignments for registers that write back at the same time. We identify the dependence tag as the *virtual tag* and the physical register (and bank) assignment as the *physical tag*; the number of virtual and physical tags, and physical registers are *all* equal. In [8], the physical registers are fewer than the virtual tags, causing deadlock problems. But we do not have that problem.

The virtual tag is assigned in rename, stored in the rename table, and used for wakeup and bypass condition, similar to dependence tags in conventional rename. As shown in Figure 1(c), decoupled rename assigns the physical tag in the mem stage, just before writeback. Instructions in writeback update the *physical tag table* with their physical tag, so that later consumer instructions know which physical register holds their value. The physical tag table is

Table 2: System parameters.

Issue width	8
RUU	128 entries
Int. reg. and ports	180 regs. 16 rd and 8 write
FP reg. and ports	180 regs. 16 rd and 8 write
L1 i-cache	64K 2-way, 2-cycle
L1 d-cache	64K 2-way, 2-cycle, 2-port
L2 cache	2M 8-way, 12-cycle
Memory latency	80 cycles
Branch predictor	2-level hybrid. 8K entry L2
Mispredict penalty	6 cycles

Table 3: Register relative energy and overhead.

Energy components for one 64 bit register file copy	Relative Energy
Read to 1 of 4 ports	1.00
Write to 1 of 8 ports	0.32

read for all source register operands in an additional stage before register read, as shown in Figure 1(c). Because of the additional pipeline stage, decoupled rename introduces additional branch misprediction and load-miss penalty similar to BC.

For instructions exiting the issue queue, two cases are possible for each operand. The first is that operands, whose producers have written back, obtain their physical tag from the physical tag table and then read the register file. The second is that operands, whose producers have not yet reached writeback, will *not* have a physical tag in the physical tag table. However, these operands are *guaranteed* to get their values from bypass via virtual tag match, making a physical tag unnecessary. In the event of a load miss, branch misprediction, or exception, the physical tag table is repaired in the same way as the rename table.

Unavoidable conflicts simply stall the pipeline back-end from issue till writeback. Because we do not use read banking, decoupled renaming presents no additional complications in issue and register read. Both virtual and physical tags are freed when the next instruction writing the same logical register commits, just like conventional renaming.

The physical tag table may be optimized by utilizing bypass hint for *energy-optimized* decoupled rename. Because the physical tag table need only be read for operands coming from the register file, the bypass bit may be used to reduce the number of physical tag table read ports in the same way it is used to reduce register read ports. This optimization is substantial from an energy standpoint; avoiding reading the physical tag table for the high number of bypass operands substantially reduces the porting and energy overhead of the physical tag table.

A real implementation of our register file techniques would combine write banking with one of the read techniques. It would seem that combining decoupled rename with bypass hint would annul bypass hint's advantage of avoiding the extra BC stage, and that combining decoupled renaming with BC (and doing the physical renaming in the BC stage) would work as well as combining decoupled renaming with bypass hint. In reality, the lack of knowledge of bypass conditions during select is a major drawback of BC. Therefore, combining decoupled rename with bypass hint is more advantageous than combining decoupled rename with BC.

4 Access time and energy impact

Typical register file implementations employ multiple copies so that each copy provides a reasonable number of true ports, and together the copies provide a large number of ports [15]. True multiporting (i.e., multiple bitlines per cell) makes register file cells slow beyond a certain number of bitlines per cell. The optimum number of copies for a given system is specific to technology and access-time requirements; in general having more copies (hence fewer bitlines per copy) improves access speed unless the increased wire lengths due to more copies offsets the advantage of fewer bitlines. For example, to get 8 read and 4 write ports, implementations may use 2 copies where each copy provides 4 read ports and 4 write ports. Because there are fewer read ports using our techniques, the number of copies needed is also reduced. Fewer copies lead to a smaller and faster register file.

The effect of banking on access time is similar to that of reducing the number of registers or using subarrays because banks shrink the size of the bitlines and reduce the number of bitlines per cell. Because our designs use only write banking due to timing concerns with read banking (Section 2.1) and the fact that we reduce read bitlines by reducing demand for read ports, our banks only reduce the number of write-port bitlines per cell. However, both read and write access time (and energy) benefit because the size of the bitlines is reduced.

4.1 Reducing energy

Bypass hint reduces read energy by avoiding most of the unnecessary anticipatory reads. Because the register file is read less often, read energy reduces significantly. Fewer *read* ports reduces *write* energy due to the way copies in multiported register files are updated. To keep the register file copies coherent, writes to a register are done to all the copies at once. Consequently, each write consumes significant energy because it must update multiple copies.

Energy overhead of the additional structures needed for register file techniques must also be taken into consideration. Bypass hint, BC, and banking require additional wires outside the register file to map ports to functional units (discussed in Section 3 and Figure 2), but these wires do not add substantial energy as they carry the same oper-

Table 5: Statistics for bypass hint with 8 and 16 stage pipelines.

Benchmarks	crafty	eon	equake	fma3d	gcc	gzip	lucas	mesa	mgrid	twolf	vortex	wupwise	Average
% from bypass 8 stage	63	70	76	78	61	68	57	66	51	69	70	67	66
% misprediction 8 stage	0.5	1.0	1.1	0.9	0.1	0.2	6.2	2.6	2.2	1.2	0.4	1.8	1.5
% from bypass 16 stage	68	77	85	85	64	73	66	74	60	72	78	75	72
% misprediction 16 stage	0.1	0.2	0.1	0.2	0.1	0.1	1.0	0.5	0.2	0.6	0.1	0.2	0.2

ands as in a conventional system. Bypass hint maintains the bypass bit for each entry in the issue queue and extra muxes to map instructions to register read ports. The energy overhead of the extra bit in the issue queue entries is negligible compared to the energy of the register file, and the energy overhead of the muxes is negligible because they only carry register numbers (8 bits), not data (64 bits), and the muxes are only one extra pass gate.

In contrast, the 2-level register file in [1] has substantial energy overhead. The authors state that the total overhead of the usage table and the copy table accesses (not to mention the active list search) is likely to counter any potential energy savings from the 2-level register file.

5 Methodology

We modify Wattch [4] to replace the architectural register file with a physical register file of the simulated size, architecture, latency, and number of ports. Wattch associates physical register energy with the Register Update Unit (i.e., instruction window) and Load/Store queue. We removed Wattch's physical register components from both the Register Update Unit and Load/Store queue in the total energy. Instead, we added our physical register file to the total energy. For register file energy dissipation, we assume that, similar to unused banks in energy-efficient caches [2], unused register ports do not consume energy from bitline swing. An example register file energy calculation, based on values reported by our modified Wattch, is shown in Table 3. We retain Wattch's energy models for the remaining components.

Table 2 shows the base configuration for the simulated systems. We assume 180 integer and floating-point registers in the baseline architecture because it is a reasonable value based on today's processors and does not inflate the energy impact of the register file. The 2-level register file [1] in our study uses 100 registers for L1 and 80 registers for L2. We assume 16 read ports and 8 write ports to support our 8-issue processor which can issue any combination of eight instructions every cycle, constrained by the two-ported L1 d-cache. For each SPEC2K application, we use ref inputs, fastforward 2 billion instructions, and run for 500 million instructions. In the interest of space, we show results for a subset of the SPEC2K applications which are representative of our results over the entire SPEC2K suite. We simulated the entire suite and carefully chose this subset ensuring that the average behavior of this subset closely matched that of the suite.

Table 4: Register file copies and write energy.

Ports and copies (all 8 write ports)	read ports/ copy	bitlines/ cell	Relative write e-delay
16 read, 4 copies	4	12	1.0 (base)
6 read, 3 copies	2	10	0.76
6 read, 2 copies	3	11	0.51

6 Results

In each subsection, we show performance and energy-delay relative to a processor using an idealized 1-cycle, 16 read-port, 8 write-port register file. It is important to note that this base case is intended to be an unrealizable upper bound for performance; therefore all of our results show performance *degradation* relative to this ideal. The idealized base case provides a reasonable comparison against our techniques which facilitate an achievable 1-cycle register file. We report energy-delay, a widely used metric in low power research because it considers both the energy and performance impact of a technique. If energy *alone* were used as a metric, results would be biased toward techniques with large energy savings at the cost of large performance degradations.

Section 6.1 through Section 6.3 show energy delay calculated only from register file energy and overhead from our techniques while Section 6.4 shows processor energy-delay. In Section 6.1 we show that bypass hint and BC are effective techniques for reducing read ports and reducing register read energy. Section 6.2 shows the effect of reducing read ports on write energy and the effectiveness of write banking. Section 6.2 also shows that decoupled rename effectively reduces the performance degradation caused by each bank having fewer ports when there are a large number of banks. In Section 6.3 we compare the performance of bypass hint to the 2-level technique proposed in [1]. Finally, in Section 6.4 we combine our read and write techniques to evaluate our proposals in terms of overall processor energy-delay.

6.1 Register Read Techniques

In this section, we evaluate the effectiveness of using BC and bypass hint to reduce register read ports by avoiding anticipatory reads. Table 5 presents the percent of oper-

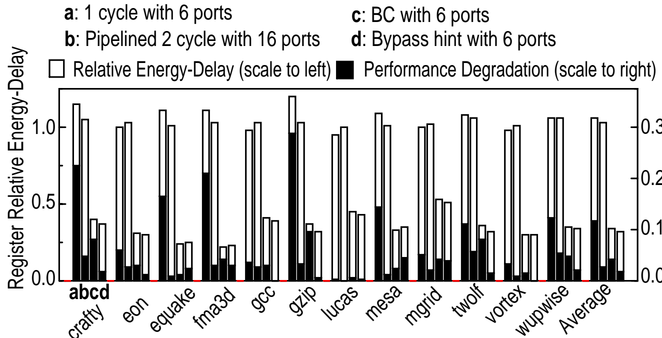

FIGURE 3: Read energy delay and performance.

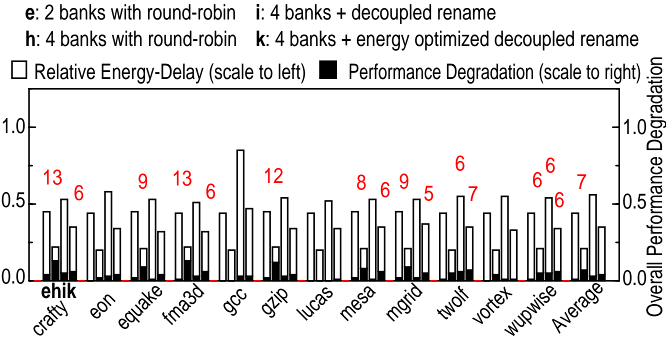

FIGURE 4: Write energy delay and performance.

ands from bypass, and bypass hint accuracy for both 8 and 16 stage pipelines. For the 8-stage pipeline, between 51 and 78 percent of operands come from bypass, with an average of 66%. Bypass hint has a low average misprediction rate of 0.9%. As pipeline depths increase, more stages of bypass are needed, increasing the opportunity for bypass operands. For the 16- stage pipeline the average percent of operands from bypass is 72%. The increased opportunity for bypass operands reduces the misprediction rate to 0.2%. The only application with an unusually high misprediction rate for either pipeline is *lucas,* which has few bypass operands because its d-cache miss rate is over 18%. From this point, all results use the 8-stage pipeline.

Figure 3 shows results for register reads. From left to right, the bars represent a 1-cycle register file with only 6 read ports, a *perfectly*-pipelined 2-cycle register file with 16 read ports, a 1-cycle register file with 6 read ports using BC to eliminate anticipatory reads, and a 1-cycle register file with 6 read ports using bypass. All register files in this section have 8 write ports. The results are relative to our ideal, but unachievable 16 read port, 1-cycle register file. The black sub-bars (right scale) represent performance degradation relative to the base case. The full-height bars (left scale) represent relative register read energy delay (i.e. taking into account only register read energy). We show the 2-cycle pipelined register file, which may be difficult to implement due to complications with pipelining SRAMS, for comparison.

Using BC and bypass hint to eliminate anticipatory reads effectively reduces the number of required register read ports. A conventional processor using only 6 read ports *without avoiding* anticipatory reads incurs a performance degradation of 12%, as shown in the leftmost bars. In contrast, BC and bypass hint incur average performance degradations of 4% and 2% respectively despite only having 6 read ports for an 8-issue processor. The pipelined 2-cycle register file has degradation of 3%.

Though BC and bypass hint perform similarly, recall BC's complications (Section 3.2). Also, BC incurs the largest absolute degradations because of branch misprediction and load miss penalties as well as the inability of the scheduler to avoid conflicts. *Crafty, gzip,* and *twolf* all exceed 7%. The maximum performance degradation with bypass hint is less than 5%.

BC and bypass hint perform comparably in energy delay, achieving average savings of 65% and 67% respectively. Although BC is deterministic, bypass hint's predictive nature is acceptable because of its high accuracy. Bypass hint also avoids the extra penalties associated with BC. In contrast to bypass hint and BC, the 2-cycle, pipelined register file experiences an average energy delay savings of *minus* 3% (this is an *increase* in energy-delay) because it does not avoid anticipatory reads and incurs performance degradation over the 1-cycle register file.

Unlike pipelined, banked, or 2-level register files that attempt to satisfy ever-increasing demand for read ports, bypass hint eliminates much of the demand, resulting in substantial energy savings with minimal performance degradation. In contrast to register reads, register writes cannot be eliminated, as we said in Section 3.3. We address register writes in the next subsection.

6.2 Register Write Techniques

6.2.1 Effect of bypass hint and write-banking

In this section, we evaluate techniques for meeting register write demand and reducing write energy. We expect the use of bypass hint for register reads to substantially reduce write energy by mitigating the need for register file copies (Section 4.1). We also apply banking to register files to reduce energy. We expect decoupled rename to mitigate the performance degradation from write banking by avoiding write bank conflicts.

Bypass hint alone substantially reduces write energy. Each register write must occur to every copy of the register file. Because bypass hint reduces the number of necessary read ports, the number of copies are reduced. As discussed in Section 4, the number of copies is technology dependent and based on the number of bitlines (i.e. ports) per cell. Reducing copies decreases access time and saves write energy because each write occurs to fewer locations. We assume that our base register file with 16 read and 8 write ports, will require 4 copies. We compare our base to bypass hint configurations using either 2 or 3 copies in Table 4.

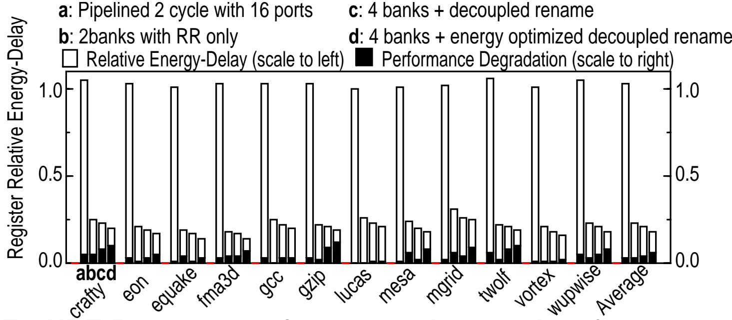

FIGURE 5: Overall regfile energy delay and performance.

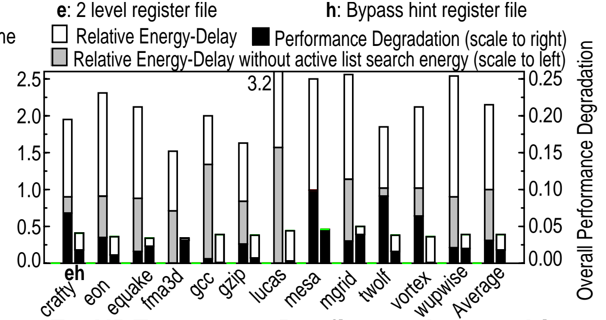

FIGURE 6: 2-level Regfile. vs. bypass hint.

(The relative energy delay is computed for the same group of benchmarks as in the previous section.) These configurations have comparable numbers of bitlines per cell, but those with fewer copies will have faster access time due to wire delay. The write energy-delay savings are proportional to the number of copies eliminated. We use 2 copies for further experiments.

We also apply write banking as discussed in Section 3.3 to satisfy write bandwidth requirements and reduce energy. Figure 4 depicts performance degradation and relative write energy-delay for 4 configurations: using 2 and 4 banks with conventional round-robin bank allocation, using 4 banks with decoupled rename, and using 4 banks with energy-optimized decoupled rename. Applications with performance degradations of at least 5% have their degradations noted above the bars. Note that the scale for the performance degradation bars is different from Figure 3. (The scale change is needed to avoid the black bars overlapping the white bars.) These results use 16 read ports and include *only* write banking.

Dividing the register file into two banks (4 write-ports per bank) causes minimal performance degradation. While increasing the number of banks reduces energy under the constraint of maintaining the same number of total ports, the constraint forces fewer ports per bank causing more bank conflicts. The average performance degradation with 2 banks is 1%, compared to 7% with 4 banks. However, from 2 to 4 banks, the average energy-delay savings increases from 56% to 79%. In addition to the energy benefits associated with physically smaller banks, using 4 banks may be preferable to using 2 banks due to access time considerations if the performance degradation due to conflicts can be reduced.

We use decoupled rename, as discussed in Section 3.3, to prevent write bank conflicts when using 4 banks. As shown by the third set of bars in Figure 4, decoupled rename reduces the average performance degradation with 4 banks from 7% to less than 3% even though the technique adds an additional pipeline stage to read the physical tag table before register read. Unfortunately, the large number of accesses to the physical tag table adds substantial energy overhead, as shown by the reduced energy delay savings. The worst case is *gcc,* with an energy-delay over 75%. *Gcc* experiences many squashed instructions and an unusually high register read to write ratio of about 6 to 1. Due to the large number of reads requiring physical tags, the energy overhead of the physical tag table dominates the register write energy.

We use bypass hint to reduce the overhead of decoupled rename's physical tag table. As mentioned in Section 3.3, our optimization uses information from bypass hint to reduce accesses to the physical tag table; bypassed operands do not need to look up the table. The results show that energy-optimized decoupled rename increases energy-delay savings from 44% to 65%. However, the optimization also increases average performance degradation to 4% due to bypass mispredictions.

6.2.2 Register read and write techniques combined

In this section, we combine bypass hint with decoupled rename. We do not combine BC with decoupled rename because of the arguments discussed at the end of Section 3.3. When bypass hint and decoupled rename are combined, register file energy delay is reduced by over 75%. The register file with 2 banks achieves substantial energy savings, while additional savings may be extracted using more banks and decoupled renaming. Figure 5 combines read and write techniques and depicts register file energy-delay (i.e. read energy, write energy, and overhead). The performance degradation values of the first three bars are also shown in Figure 7 using a larger scale. The 2-cycle pipelined register file has an energy delay savings of *minus* 3% (this is an *increase* in energy-delay*)* with a 2.8% performance degradation. The 2-banked register file has an energy delay savings of 77% with a performance degradation of 2.9%. The 4-banked register file with decoupled rename has an energy delay savings of 79% with a performace degradation of 4.2%. Energy-optimizing the physical tag table increases energy delay savings to 82% while increasing performance degradation to 6.3%.

6.3 Comparison with 2-level

In this section, we compare bypass hint to the 2-level register file proposed in [1]. We expect the 2-level register

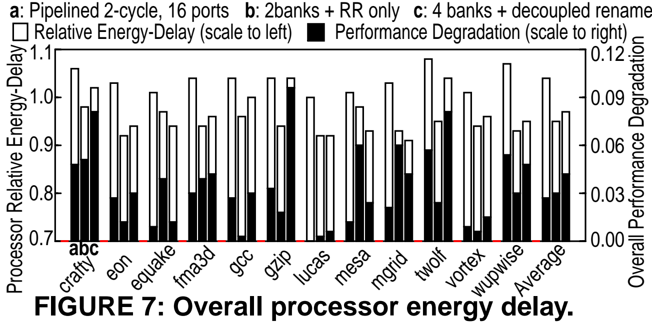

FIGURE 7: Overall processor energy delay.

file to suffer more performance degradation than bypass hint because of the additional misspeculation penalty caused by moving values from the L2 to the L1 and register shortages in the L1. As mentioned in Section 4.1, we expect the 2-level register file to incur substantial energy overhead from its usage table, copy list, and active list searches.

In our comparisons against the 2-level register file, our technique only uses bypass hint with no banking. In [1], the authors state that banking would provide little benefit to the 2-level register file (because their L1 and L2 are already fairly small), but banking provides substantial benefits for bypass hint. To be fair, our comparison does not include write banking, but our energy-delay numbers do include both register read and write energy as well as overhead from the techniques. We also assume that the 2-level register file can search the active list every cycle to update the usage table.

Figure 6 shows the 2-level register file with a 100 entry L1 and 80 entry L2, compared to our idealized 1-cycle 16 read-ported 180 entry register file. The figure also shows our bypass hint technique. The black sub-bars represent performance degradation (scale on right; different from scale in previous graphs). The full bars represent register file energy-delay (scale on left; different from scale in previous graphs). For the 2-level register file, the top of the gray bars represents relative energy-delay *without* the energy overhead of searching the active list. The top of the full bars represents relative energy-delay including the active list search overhead.

The 2-level register file experiences greater performance degradation than bypass hint. We idealistically assume the active list can be searched each cycle, and we assume that 8 usage table entries may be checked each cycle during misspeculation recovery. The average performance degradation for 2-level is 3.1% compared to 1.8% for bypass hint. Low IPC applications such as *lucas,* which use few registers, suffer the least with 2-level. However, searching the active list every cycle is not realistic. Less frequent searches of the active list mean fewer chances to move registers to L2. If the active list can be searched only every three cycles, average performance degradation increases to 5.4% (not shown in figure).

The 2-level register file has large energy overhead. Without considering the overhead of searching the active list, our estimates of the energy overhead of the 2-level register file are actually slightly lower than those in [1]. Compared to bypass hint reducing energy delay by 61%, the 2-level register file reduces energy-delay by *minus* 1% (this is an *increase* in energy-delay*)*. When the overhead of searching the active list is included, the energy-delay increases to 215% that of the base case.

6.4 Processor Energy-Delay

In this section we evaluate the effects of bypass hint and write banking on overall processor energy. Because modern register files represent about 10% to 15% of processor *energy*, we expect to see processor *energy-delay* savings of less than 10% when accounting for performance loss.

Figure 7 depicts overall processor energy-delay and performance degradation for register files using bypass hint and using write banking with either 2 banks or 4 banks plus decoupled rename. Results are relative to the idealized 1-cycle register file. The perfectly-pipelined, 2-cycle register file with 16 read ports, is also shown. Note that both the energy and performance degradation scales are different from previous graphs.

The 2-bank register file performs slightly better than the 4-bank register file. Processor energy-delay savings is 5% for the 2-bank register file and 3% for the 4-bank register file with decoupled rename. The energy-saving potential of bypass hint is evident in a comparison against the pipelined 2-cycle register file, which is a more realistic design point than the 16 port, 1 cycle register file. Compared to using the 2-cycle register file, bypass hint and 2 write banks achieves a processor energy-delay savings of 9% with virtually no performance loss (0.1%).

Performance degradations are 2.9% and 4.2% respectively for 2 banks with RR and 4 banks with decoupled rename when compared to the 1-cycle, 16 read-port register file. Two problem cases when using 4-banks are *twolf* and *gzip*. For *twolf,* the energy overhead of the physical tag table eliminates energy-delay savings with 4-banks. Though it's energy-delay and performance are slightly worse, the 4-bank register file, however, might be necessary in some designs for faster access time.

Energy savings from bypass hint and write banking increase with the number of physical registers. If we increase the number of registers from 180 to 256, the energy-delay savings of bypass hint with 2 write banks over the 1-cycle, 16-read-port register file is 8% with no change in performance loss (not shown in figures).

7 Conclusions

We proposed to reduce the number of register ports

through two proposals, one for reads and the other for writes. For reads, we proposed bypass hint to reduce register port requirements by avoiding unnecessary register file reads for cases where values are bypassed. Current processors are unable to avoid these unnecessary reads due to timing constraints. For writes, we used register file banking. Current banking schemes assign different banks to instructions that are renamed together, which does not necessarily avoid conflicts among instructions that writeback together. We used decoupled rename, which separates dependence and physical tagging of register operands. Decoupled rename allows us to perform physical register allocation just before writeback, avoiding bank conflicts.

Unlike previous register file techniques, our proposals do not require complications such as searching the active list, storing register operands in the issue queue, maintaining coherence among register caches, and selective reissue upon a register cache miss. Bypass hint requires that we map instructions to a limited number of read ports, but bypass information is available early enough to perform this mapping in parallel with issue. While read bandwidth demand can be easily reduced via bypass hint, we found that write bandwidth demand cannot be easily reduced. Optimizing writes required banking, but mitigating bank conflicts required introducing decoupled rename.

On average, the SPEC2K benchmarks obtain about 66% of source operands from bypasses; bypass hint correctly predicts 98.5% of the bypassed operands. Using only 6 read ports, bypass hint reduces read energy-delay by 66% with 1.8% performance degradation compared to an unrealizable, 16 read-port, 1-cycle register file. While using 2 write banks (4 write-ports per bank) results in little performance degradation, 4 banks (2 write-ports per bank) with decoupled rename has energy-delay savings of 65% compared to 56% for two banks. Using 4 banks without decoupled rename degrades performance by 7%; with decoupled rename degradation is only 3%. The 2-level register file has a performance degradation of 3.1% assuming 1-cycle searches of the active list, but *increases* energy-delay by 115%. In contrast, bypass hint has a performance degradation of 1.8% with a register file energy-delay savings of 61%. Combining bypass hint and write banking, our 1-cycle register file with 6 read ports, and two 4-write-ported banks achieves a 9% processor energy-delay savings over a system using a perfectly-pipelined, 2-cycle register file with 16 read ports and 8 write ports.

Acknowledgements

This research is supported in part by NSF under CAREER award number 9875960-CCR, SRC under contract 2000-HJ-768, and an Intel Ph.D. Fellowship.

References

[1] R. Balasubramonian, S. Dwarkadas, and D. Albonesi. Reducing the complexity of the register file in dynamic superscalar processors. In *Proceedings of the 34th International Symposium on Microarchitecture (MICRO 34)*, pages 237–249, Dec. 2001.

[2] D. Boggs. Breathing life into a paper tiger. In *Keynote speech: 33rd International Symposium on Microarchitecture*, Dec. 2000.

[3] E. Borch, E. Tune, S. Manne, and J. Emer. Loose loops sink chips. In *Eighth International Symposium on High Performance Computer Architecture*, pages 299–310, Feb. 2002.

[4] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 83–94, June 2000.

[5] A. Capitanio, N. Dutt, and A. Nicolau. Partioned register files for vliws: A preliminary analysis of tradeoffs. In *Proceedings of the 25th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 292–300, Oct. 1992.

[6] D. Carmean. Personal communication. Oct. 2001.

[7] K. I. Farkas, P. Chow, N. P. Jouppi, and Z. Vranesic. The Multicluster architecture. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 30)*, pages 149–159, December 1997.

[8] A. Gonzalez, J. Gonzalez, and M. Valero. Virtual-physical registers. In *Proc. 4th International Symposium on High Performance Computer Architecture*, Feb. 1998.

[9] L. Gwennap. Digital 21264 sets new standard. In *Microprocessor Report*, Oct. 1996.

[10] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1990.

[11] J.-H. Kwon, J. Lim, and S.-I. Chae. A three-port nrerl register file for ultra-low-energy applications. In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 161–166, July 2000.

[12] K. lorenzo Cruz, A. Gonzalez, M. Valero, and N. P. Topham. Multiple-banked register file architectures. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 316–325, June 2000.

[13] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, June 1997.

[14] M. Reilly. Personal communication. Oct. 2001.

[15] J. E. Smith and S. Weiss. Powerpc 601 and alpha 21064: A tale of two riscs. *IEEE Computer*, 27(6):46–58, June 1994.

[16] J. Tseng and K. Asanovic. Energy-efficient register access. In *Proc. XIII Symposium on Integrated Circuits and Systems Design*, Sept. 2000.

[17] J. Zalemea, J. Liosa, E. Ayguade, and M. Valero. Two-level hierarchical register file organization for vliw processors. In *Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 33)*, pages 137–146, Dec. 2000.