Reducing the Complexity of the Register File in Dynamic Superscalar Processors *

Rajeev Balasubramonian†, Sandhya Dwarkadas†, and David H. Albonesi‡
† Department of Computer Science
‡ Department of Electrical and Computer Engineering
University of Rochester

Abstract

Dynamic superscalar processors execute multiple instructions out-of-order by looking for independent operations within a large window. The number of physical registers within the processor has a direct impact on the size of this window as most in-flight instructions require a new physical register at dispatch. A large multi-ported register file helps improve the instruction-level parallelism (ILP), but may have a detrimental effect on clock speed, especially in future wire-limited technologies. In this paper, we propose a register file organization that reduces register file size and port requirements for a given amount of ILP. We use a two-level register file organization to reduce register file size requirements, and a banked organization to reduce port requirements. We demonstrate empirically that the resulting register file organizations have reduced latency and (in the case of the banked organization) energy requirements for similar instructions per cycle (IPC) performance and improved instructions per second (IPS) performance in comparison to a conventional monolithic register file. The choice of organization is dependent on design goals.

1 Introduction

Modern high-performance processors use an out-of-order dynamic superscalar core to extract instruction-level parallelism (ILP) from applications. These processors examine a large window of in-flight instructions to find multiple ready and independent instructions every cycle. The size of this window is one of the key determinants of the degree of ILP that can be achieved. However, supporting a large window of in-flight instructions also requires large structures within the processor, namely, a large register file, issue queue, and reorder buffer (ROB). Since in high frequency designs microarchitects try to set the clock speed of the processor based on the execution speed of simple integer instructions, a large multi-ported register file can potentially compromise clock cycle time.

*This work was supported in part by NSF grants EIA-9972881, EIA-0080124, CCR–9702466, CCR–9701915, CCR–9811929, CCR-9988361, and CCR–9705594; by DARPA/ITO under AFRL contract F29601-00-K-0182; and by an external research grant from DEC/Compaq.

The register file size has a direct impact on the number of in-flight instructions since every dispatched instruction that has a destination register is assigned a new physical register. Hence, once the free registers run out, the dispatch stage gets stalled, causing the processor to look for ILP within a restricted window until the oldest instructions commit and free their registers. The growing gap between memory and processor speeds results in an increasing number of long latency instructions, causing the commit stage to be frequently stalled and further necessitating a large number of registers. In addition, the large issue widths in such processors also require a large read/write bandwidth to the register file. Implementing a large number of registers with many ports for the sake of increased ILP poses a number of challenges in terms of both performance and energy.

The register file is a heavily-ported RAM structure. A processor capable of issuing eight integer instructions each cycle may need an integer register file with sixteen read ports (corresponding to two source operands per instruction) and eight write ports. Using a register file access time model derived from CACTI-2.0 [28], we found that the access time for an 80-entry 24-ported register file can exceed 1.5ns at 0.18μ technology, potentially being on critical paths determining the cycle time. The current trends of increased frequencies, dominating wire delays at smaller technologies [16, 19], and increased register requirements because of simultaneous multithreading [26] make it harder to implement a register file that can be accessed in a single cycle. Having a large register file with a multi-cycle access time poses problems of its own. For example, a 3-cycle register file access time would require three levels of bypassing among the functional units, thereby increasing the bypassing delay, another cycle-time critical path [19]. A multi-cycle register file access time would also degrade instructions per cycle (IPC) by increasing the branch mispredict penalty and the register file pressure by increasing register lifetimes. Furthermore, pipelining the register file is not a trivial task as it is a RAM structure.

Given these constraints, the register files in modern dynamic superscalar processors have been very modestly sized. The Alpha 21264 [12] has as many as 80 integer physical registers, but requires a clustered organization to

reduce the number of ports and hence the access time. Clustering the register file can potentially have a detrimental effect on IPC because of inter-cluster communication. Farkas et al [9] showed that larger register file sizes resulted in improved IPC even as the sizes were increased beyond 128 entries, but modern dynamic superscalar processors do not support that large a size because of cycle time constraints. Modern processors are also limited by problems relating to power consumption. The register file consumes a non-negligible portion of chip power, around 10% according to the power models based on Wattch [2].

In this paper, we address the problem of designing the register file in a more complexity-effective manner in the context of dynamic superscalar processors while maintaining IPC and significantly improving instructions per second (IPS) performance. We follow the complexity-effective design approach of Palacharla et al. [19] in that we seek to reduce the access time of critical processor structures (in our case, the register file) even if this involves introducing other structures that are not on cycle time critical paths. We achieve this goal via two orthogonal approaches:

- Reducing the number of required registers in the register file on the critical path by using a more efficient register allocation policy.
- Reducing register file port complexity without unduly sacrificing register read/write bandwidth.

In terms of energy dissipation, we find that the first technique imposes an energy penalty comparable to its performance improvement, while the second technique provides a significant reduction in register file energy consumption in addition to improved performance.

We achieve the first objective via a hierarchical division of registers into those with active consumers and those waiting for precise conditions. Our design differs from previous approaches to register file partitioning (described in Section 6) in being hardware-based rather than relying on compiler support. Registers are allocated from the first-level (L1) register file at the time of dispatch. When a register value has been completely consumed by all instructions that source the value, it is moved to the second-level (L2) register file. These values are retained in the L2 since they might be needed in the event of a branch misprediction or an excepting instruction. Since the L1 register file now contains only those values that will be sourced by the functional units, it contains fewer registers than a single-level register file, and therefore its access time is considerably smaller. However, the additional structures that are introduced to keep track of the status of various registers consume non-trivial amounts of energy, reducing the potential of such an organization as an energy saving technique.

To reduce register file port requirements, we propose a banked organization that bears similarities to that proposed for data caches [10, 20] as well as other previously proposed banked register file organizations (discussed in Section 6). Our approach differs from these prior efforts in that our banks have a *single* read port and a *single* write port (which we call *minimally ported*), despite the fact that our processor is capable of reading sixteen registers and writing eight registers each cycle; and we model a dynamic, single-cluster, superscalar processor as opposed to the VLIW or clustered superscalar processor models of other approaches. The result is a more scalable alternative to a large, monolithic register file that operates considerably faster while dissipating significantly less energy, even with the additional address predecoding and output multiplexing required.

We also show that combining the two techniques by using a smaller banked L1 in conjunction with an L2 register file does not result in further improvements in IPS. This is due to the fact that the access time improvement of splitting a banked organization into two levels is overridden by the small but additive (but no more than additive) IPC degradation of the two techniques at the evaluated technology point.

The organization of the rest of this paper is as follows. Section 2 outlines the operation of a conventional register file. Section 3 describes the proposed two-level register file organization while Section 4 describes the banked register file approach. We evaluate the proposed designs in terms of IPC, access times, and energy in Section 5. Section 6 compares and contrasts our approaches with existing related work. Finally, we make concluding remarks in Section 7.

2 Conventional Register File Organization

The register file is typically a RAM structure consisting of a fixed number of registers with as many write and twice as many read ports as the maximum number of instructions that can issue in any cycle. In addition, dynamic superscalar processors like the Alpha 21264 [12] and the MIPS R10000 [29] use a physical register allocation policy similar to the one illustrated here by an example:

```
    Original code        Renamed code
1:  lr5 <- ...           pr18 <- ...
2:  ... <- lr5           ...  <- pr18
3:  branch to x          branch to x
4:  lr7 <- lr3           pr22 <- pr24
5:  lr5 <- ...           pr27 <- ...
    ...                  ...
6:  x: ... <- lr5        x: ...  <- pr18
```

At dispatch, the first write to logical register 5 (lr5) causes it to get mapped to physical register 18 (pr18). This value is read by the next instruction, after which a branch is encountered. The branch is predicted to be not taken and subsequently, another write to lr5 occurs. At this point, lr5 gets mapped to a different free physical register, pr27. However,

the value in pr18 can still not be freed as the branch may have been mispredicted, in which case, there would be another read from lr5 (instruction 6), which actually refers to pr18. Further, if the write to lr7 (instruction 4) were to raise an exception, to reflect the correct processor state, lr5 would have to be mapped to the value in pr18. Hence pr18 cannot be released back into the free list until the next write to lr5 (instruction 5) commits, which guarantees that all previous branches have been correctly predicted and all previous instructions have not raised an exception. This mechanism to release registers back into the free pool is easily implemented in hardware - the ROB keeps track of the old physical register mapping for each instruction's logical register and releases it at the time of commit. However, it leads to long register lifetimes since long latency operations (loads from memory, for example) could hold up the commit stage for many cycles.

3 A Two-Level Register File

Our two-level register file uses an allocation policy that leaves values that have potential readers in the level one (L1) register file and transfers other values into level two (L2). This significantly reduces the number of required L1 entries for a given level of IPC performance, thereby reducing register file access time. Details of the register allocation policy and the required microarchitectural changes are discussed next.

3.1 Microarchitectural Changes

Figure 1 shows a block diagram of our proposed microarchitecture, outlining its essential features. We assume an 8-way issue processor in the following discussion.

During rename, register names correspond only to L1 physical registers; L2 registers are hidden from the rename process. We introduce a new hardware structure, shown in Figure 1, that monitors the usage statistics for the L1 physical registers. For every L1 physical register, this *Usage Table* maintains the following information:

- A *Pending Consumers* counter that keeps track of the number of pending consumers of that value. During rename, an instruction that sources the register increments it. During issue, the same instruction would then decrement it[1].
- A single bit (called the *Overwrite* bit) that is set when the physical register is no longer the latest mapping for its logical register.
- Another bit that indicates if a result has been written into the physical register.
- The sequence number for the branch immediately following the instruction that writes to this physical register (sequence number 1).
- The sequence number for the branch immediately preceding the next instruction that writes to the same logical register (sequence number 2).

Most of the information required to update this table is readily available during the rename stage. The sequence number counters identify the various in-flight branches and may be as many bits as $log_2(ROBsize)$. When the number of L1 physical registers falls below a pre-set threshold, registers that have a *Pending Consumers* count of zero, have a result in them, and have their *Overwrite* bit set are copied into the L2 (provided there are free L2 physical registers). The corresponding L1 registers are released into the free pool. A single *L2 ID valid* bit, added to each ROB entry, indicates that the destination register ID in that entry corresponds to an L2 register. At the time of commit, the register is released back into the L2 free pool instead of the L1 free pool.

The *Copy List*, which keeps track of L1-L2 copies for recovery from a branch mispredict, contains the following information for each L2 register:

- The L1 physical register name that had earlier contained the value.
- The sequence number for the branch immediately following the instruction that writes to this physical register.
- The sequence number for the branch immediately preceding the next instruction that writes to the same logical register.

These values are copied from the *Usage Table* when the transfer is made.

The two branch sequence numbers stored indicate the 'live' period of a physical register value, *i.e.*, the period during which instructions sourcing this value are dispatched. If a branch with a sequence number between the two sequence numbers (both inclusive) for an entry mispredicts, then the L2 register value is reinstated back to L1, as instructions along the correct path may need to source that value. All such L2 values (referred to as the 'live' set) are copied back into the L1. The original L1 registers of the 'live' set are guaranteed to be available for the following reasons. An L1-L2 copy can occur only when the *Overwrite* bit is set, that is, when a newly renamed instruction (call it instruction RI) has the same logical destination register as the copied instruction (CI). Thus, this is the point at which CI's L1 physical register can be reused. However, a branch that mispredicts and causes the value of CI to be restored back to L1 by definition must have occurred before instruction RI. Thus, instruction RI and its successors will

[1] An instruction being squashed as a result of a branch mispredict also decrements the counter.

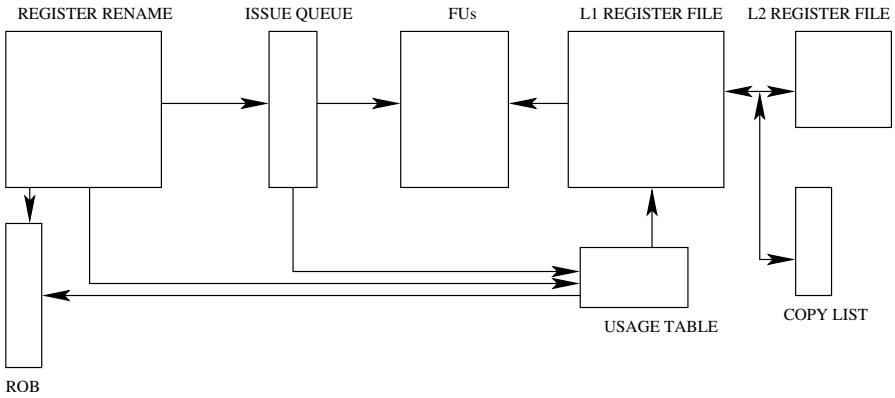

Figure 1. The Two-Level Register File Organization.

be squashed, thereby guaranteeing the availability of CI's original L1 register. In addition, this mechanism requires no modifications of the register map table checkpoint and restoration process.

The processor also needs to recover to a valid register file state on an exception. Since exceptions are not as frequent as branch mispredicts, most designs (the MIPS R10000 [29], for example) simply traverse the ROB in reverse order to restore old register mappings. For the two-level register file, values are also restored from the L2 to the L1 as follows. The 'live' set of the branch closest to the ROB head (the excepting instruction) is reinstated as in a mispredict. Then, the ROB is traversed starting at the branch instruction and moving toward the excepting instruction. Each entry whose *L2 ID valid* bit is set has its L2 value restored to the L1. Because of the traversal of the ROB, this process is likely to take a number of cycles, even though branches are often fewer than 10 instructions apart. Since exceptions are infrequent and the process of recovering the register mappings is of comparable complexity, the overhead of this copying operation should be negligible.

3.2 Complexity of the Proposed Structures

The *Usage Table* has as many entries as the L1. Each entry requires $log_2(ROBsize)$ bits for each sequence number (or no more than eight bits each for current processors) and only a few bits for the *Pending Consumers* counter, in addition to the two single-bit fields. As a result, we have found that the *Usage Table* access time is much less than that of the L1 register file. Note that the *Overwrite* bit is checkpointed on every branch so it could be recovered in case of a mispredict. The table look-up to determine L1 registers that are candidates for copying to the L2 requires simple combinational logic for each entry. The *Copy List* and the few-ported L2 register file are also small structures compared to the L1. We have modeled these as well and found their access times to be less than that of the L1.

In terms of energy, the frequent access and modification of the various *Usage Table* fields adds non-trivial amounts of overhead. In a given cycle, up to eight instructions across two basic blocks can be dispatched. Hence, a number of registers could update their sequence number fields within the usage table, although the value being written into these fields could be only one of two values (as all instructions belong to one of two possible basic blocks). As a result, the structure would have two sets of bitlines and wordlines, but many decoders. Up to eight instructions could update sequence number 1 and eight more could update sequence number 2. Given the small size of the fields, the decoder energy dominates the energy consumption for this structure. To reduce the energy consumption, this structure could be integrated with a CAM implementation of the rename table, or with the free list, thereby doing away with any additional decoding to identify the registers being renamed that cycle.

For the *Pending Consumers* counters, in a given cycle, up to eight instructions can dispatch and eight can issue, resulting in many possible counter updates. We also noticed that most registers that were copied into the L2 only had a single consumer. Restricting the L2 to only such registers resulted in almost no performance degradation. Hence, the counter could be a single bit, with another bit to indicate overflow and the register's non-candidacy for copying into the L2. The number of possible values that can be written, and therefore the word and bit line energy, is reduced by this mechanism. Again, the decoding process to identify the counter dominates the energy consumption.

The *Copy List* has as many entries as the L2. It consists of a RAM part that stores the various fields (not exceeding 24 bits). It also consists of a CAM part for improved efficiency as the entries would have to compare their branch sequence numbers with that of the mispredicted branch while copying values back into the L1. Since the CAM is invoked only on a mispredict, its energy consumption is negligible compared to that of the RAM part. The energy consumption of the RAM structure is also low as only a single copy is performed each cycle, requiring a single read/write port.

The copying process need not require additional ports in the L1. The L1 register ports are often not maximally

utilized because there aren't enough ready instructions or instructions have fewer register source operands. The copy from the L1 is made during these periods when spare read ports are available.

Our modified ROB has an extra *L2 ID valid* bit and $log_2(max(L1size, L2size))$ bits for the register identifier. Since a comparable two-level organization is likely to have fewer L1 registers, the size of each entry is practically unaffected. The number of accesses to the ROB goes up by the number of copies to L2, but is unlikely to increase contention or energy consumption significantly.

4 A Minimally-Ported Banked Register File

This section tackles the second source of complexity: the large number of register file ports in a wide-issue processor. In a processor capable of issuing eight integer instructions, as many as 16 operands could be read from, and as many as eight operands could be written to, the integer register file each cycle (see Figure 2). Meeting this high bandwidth requirement via true multiporting is costly in terms of access time, power dissipation, and scalability. A similar problem exists for high bandwidth data caches, and the alternatives to true multiporting that have been proposed in the literature [20, 24] are double-pumping, replication of the arrays, and banked organizations. Double pumping can be employed if the access time for an array structure is much smaller than the cycle time. It is not very scalable and can usually only be employed to help reduce area as halving the number of ports usually reduces the access time by a factor of less than half. To reduce the complexity of the register file, the Alpha 21264 [12] implements a replicated register file, one in each cluster, so as to reduce the number of read ports. Replication results in a penalty in terms of IPC because of the added communication cost between the clusters. We explore the benefits of banking to reduce multiporting requirements in the following sections.

4.1 Register File Port Requirements

Although a processor capable of issuing eight integer instructions and simultaneously writing back eight integer instructions theoretically could use as many as 24 integer register ports in a cycle, the number of ports required on average are a lot fewer for several reasons:

- Many operands are read off of the bypass network, not from the register file.
- Many instructions only have a single register operand.
- A number of instructions produce results that are not written to the register file (branches, stores, effective address computation part of a load or store).

Using the processor model described in Section 5, we evaluated the average port requirements for the benchmark programs. We found that for every issued instruction, only 0.64 values were read from the register file and 0.73 were read off the bypass network. In terms of actual performance, we observed that using four read and four write ports caused very few instructions to stall due to a conflict for a port and the resulting IPC degradation was only 2% on average. This is a three-fold reduction in the number of register ports, but comes at the cost of some additional complexity in the issue stage. Along with various other structural hazards, the issue stage with this organization has to take into account the port requirements of the ready instructions and postpone the issue of instructions that do not have sufficient ports. The issue queue is already aware of which registers can be read off the bypass network — these are the same registers involved in the wakeup logic that cycle. The changes in the select logic are described later. To handle the limited write bandwidth, arbitration logic is required before functional units can write results onto the result bus. Since destination registers of instructions are known in advance, this arbitration can occur a cycle in advance of writing the result. Additional registers have to be provided at the functional units to buffer results that fail to use the result bus right away, or the pipeline for the functional unit has to be stalled.

This additional logic overhead is small compared to the drastic register file energy, area, and access time savings in going from a 24-ported structure to an 8-ported structure. The most significant overhead, which we quantify in a later section, is the cost of drivers/multiplexors used to direct data from the eight ports to the 24 datapaths. The values from the read ports have to now be distributed to multiple functional unit inputs. As a worst-case scenario, we assume that the value read from any of the ports can be sourced by any of the functional unit inputs. Figure 2 shows the structure of the limited-port organization being considered. We start with this base case as it represents an attractive design point and see if we can further reduce its complexity.

4.2 Register File Banking

In an N-banked register file, the various registers are distributed among N banks, with each bank having p ports. Hence, as many as $N \times p$ values can be read in any cycle, with the added restriction that only p values can be read from any one bank. If the operands being read in a cycle are evenly distributed among the various banks, there is almost no IPC degradation compared to a central register file with $N \times p$ ports, yet complexity is greatly reduced as each structure has fewer registers and fewer ports.

We evaluate the use of a banked register file with a single read and write port per bank. Figure 2 shows a 4-banked organization. Here, an instruction may have both its source operands in a single bank, making it impossible for both to be read in the same cycle. Hence, we must allow 'partial reads', *i.e.*, if an instruction cannot issue because of bank conflicts, but can read one of its operands, it does so and

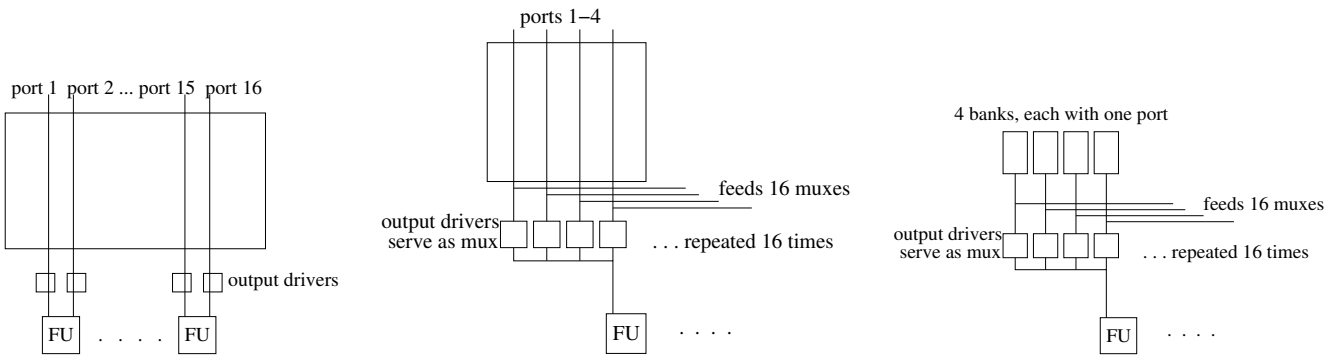

Figure 2. A conventional monolithic register file for an 8-issue processor, a monolithic register file organization with a limited number of ports, and a banked, single-port-per-bank organization (only read ports shown here).

saves the operand in the latch at the input to the functional unit. The instruction continues to remain in the issue queue, but it marks the corresponding operand as 'read'. In subsequent cycles, the instruction continues to compete for the bank corresponding to its second operand, while holding up its functional unit. When the instruction finally reads its second operand, it starts executing. While this is necessary to avoid deadlock, this phenomenon has a minimal impact on performance as it occurs very infrequently due to operands being frequently read off the bypass network. The maximum percentage of 'partial reads' out of all issued instructions was found to be only 4% for a 4-banked organization while running *em3d*.

The select logic in the issue queue has to take into account the contention for the ports and the functional units, and different implementations can trade-off select logic access time with IPC. One possible implementation would be to resolve conflicts for ports and functional units independently (using Palacharla's tree of request-grant blocks [19]) and allow an instruction to issue only if it was able to procure its functional unit and at least one register file port. While this implementation minimally impacts the latency of the select logic, there could be instances where an instruction could have issued, but does not, potentially degrading IPC. An alternative implementation could take into account port and functional unit availability at each request-grant block, before allowing a request to propagate up the tree. This could increase the delay of the select logic, but would improve the allocation of resources to ready instructions. We assumed the latter implementation in our simulations.

5 Evaluation

5.1 Simulation Methodology

We used Simplescalar-3.0 [3] for the Alpha AXP instruction set to simulate a dynamically scheduled superscalar processor with the simulation parameters summarized in Table 1. The simulator has been modified to model the memory hierarchy in great detail (including interleaved access, bus and port contention, writeback buffers, *etc*). We model issue queues that are smaller than the ROB size (in Simplescalar, the issue queues and the ROB constitute one single unified structure called the Register Update Unit (RUU)), a physical register file and mapping of logical registers to them, and split integer and floating-point issue queues and physical register files, similar to the Alpha 21264 microprocessor [12] but enhanced for wider issue. We also chose our ROB and issue queue sizes in order to ensure that they did not introduce an additional bottleneck so as to focus the results on the register file.

Fetch queue size	16
Branch predictor	comb. of bimodal and 2-level gshare; bimodal size 2048; Level1 1024 entries, history 10; Level2 4096 entries (global) Combining predictor size 1024; RAS size 32; BTB 2048 sets, 2-way
Branch mispredict cost	11 cycles
Fetch, dispatch, commit width	8
int,fp issue width	8,4
ROB and Ld/St queue	200 and 100
Issue queue size	64 (int and fp, each)
L1 I and D-cache	64KB 2-way, 32-byte lines, 2 cycles
L2 unified cache	1.5MB 6-way, 64-byte lines, 15 cycles
TLB	128 entries, 8KB page size
Memory latency	70 cycles for the first chunk
Memory ports	4 (interleaved)
Integer ALUs/mult-div;	8/4
FP ALUs/mult-div	4/4

Table 1. Simplescalar simulator parameters.

As benchmarks, we use a wide variety of programs, from the Olden [22], SPEC2000, SPEC95, UCLA Mediabench [14], and NAS parallel benchmark [7] suites. The benchmark set represents a mix of both integer and floating-point programs, as well as a mix of memory-intensive low IPC programs (that tend to run out of registers because of long latency operations that stall the commit stage) and non-memory-intensive high IPC programs (that tend to be

Benchmark	Input set	Instrs simulated	L1 mrate	Base IPC
em3d (Olden), FP	20K, 20	1000-1010M	28%	0.86
sp (NAS-uniproc), FP	A,	2500-2525M	20%	1.44
gzip (SPEC2k), Int	ref	2000-2050M	1%	2.04
vpr (SPEC2k), Int	ref	2000-2050M	2%	1.49
crafty (SPEC2k), Int	ref	2000-2050M	1%	2.48
art (SPEC2k), FP	ref	300-350M	26%	1.53
gcc (SPEC95), Int	ref	300-325M	1%	1.68
perl (SPEC95), Int	ref	500-525M	0%	2.73
cjpeg (Mediabench), Int	test	200-225M	0%	1.70
djpeg (Mediabench), Int	test	150-175M	0%	3.87

Table 2. Benchmark description and L1 D-cache miss rates. Base IPC represents a processor model with a monolithic register file with 160 entries and 24 ports.

constrained by register file bandwidth). To reduce simulation time for all programs, we studied cache miss rate traces to identify smaller instruction intervals that were representative of the whole program. The simulation was fast-forwarded past the initial warm-up phases and another one million instructions were simulated in detail to prime all structures before doing the performance measurements over the chosen interval. Details on the benchmarks are listed in Table 2. The programs were compiled with Compaq's cc, f77, and f90 compilers for the Alpha 21164 at the highest optimization level. The program code uses 32 integer and 32 floating-point logical register names.

To quantify the complexity of the baseline and proposed register file organizations, we used the access time and energy models of CACTI-2.0 [28] at 0.18μ technology as a baseline. We modified it to model a register file (similar to that done by Farkas [9]). Additional changes were made to model our proposed organizations, details of which appear in the next subsections.

5.2 Two-level Register File Evaluation

Our base case consists of a monolithic single-level register file with four read ports and four write ports. As shall be seen in the next section, this has almost the same IPC as a base case with 16 read and 8 write ports. To this, we add an L2 register file with a single read and a single write port. For our initial experiments, the sum of the registers in the L1 and L2 equals 160 (int and fp, each), which is roughly the maximum number of required registers for a ROB size of 200. We do not add any additional ports to the L1 – copies to the L2 are made only when there are free ports available. We also attempt copies only if there are fewer than eight registers in the L1 free register pool. When a mispredict is discovered, register values need to be copied back into the L1. We assume that up to four transfers can be made without adding to the mispredict penalty, *i.e.*, that it takes at least four cycles for instructions from the correct path to reach the issue stage and that one copy can be made in each of these cycles. These are rather pessimistic assumptions as typical superscalar pipelines today usually have more than four stages before the issue stage. If more than four copies need to be made, we stall the fetch stage by an extra cycle for every additional copy.

	100-L1	60-L1, 40-L2
L1	258	197
L2	0	17
L1-L2 bus	0	17
usage counters	0	22
sequence number storage in usage table	0	39
copy list	0	8
Total	258	300

Table 3. Energy breakdown for the monolithic and two-level register files. Energy is shown as the arithmetic mean of pJ/instr across all programs.

We start by assuming that the register file access time is the critical path and determines the clock speed. To compare various organizations, we use two metrics, IPC and instructions per second (IPS), which is derived by dividing the IPC by the access time for the register file. Figure 3 shows overall performance results (using the harmonic mean (HM)) for various register file organizations. The graph on the left shows the variation in IPC with the size of the L1 register file. The solid line shows IPCs for single-level register files, while the dotted line shows IPCs when these organizations are augmented with a second level (with the sum of the L1 and L2 register files held constant at 160 registers). The gap between the two lines represents the speedup possible by the addition of a second level. An overall IPC of 1.67 is the maximum possible for a ROB size (in-flight instruction window) of 200 and the two-level organization quickly saturates to this value, having an IPC of as high as 1.63 with just 80 L1 registers. The single-level organization requires as many as 140 registers to attain an IPC of 1.65. This suggests that out of 140 physical registers, only about 80 are 'active' at any given time. The remaining 60 don't have any consumers unless there is a misprediction or exception and they can be moved away to the L2.

Assuming that in high frequency designs the register file access time determines the clock speed, a designer would use the IPS metric to pick the best design point. The graph on the right in Figure 3 shows how IPS varies with the size of the L1 register file. For the single-level register file, this value peaks for a 100-entry register file. The corresponding peak for the two-level organization is seen for a 60-entry L1. The gap between the two curves illustrates that the two-level organization strikes a better balance between IPC and access times - its optimal IPS is 17% better than the optimal IPS with a single-level register file. For the two-level structure with a 60-entry L1, we also studied the effect of varying the L2 register file size and found that a 40-entry L2 yielded IPC within 1% of a 100-entry L2.

The use of a smaller L1 register file could also poten-

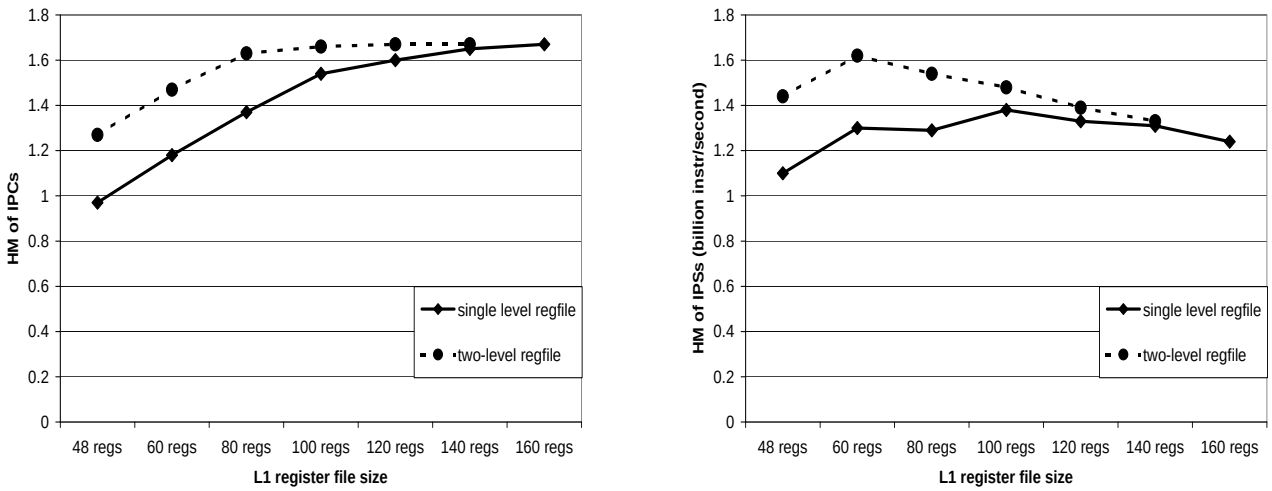

Figure 3. Graphs showing IPC and IPS with varying L1 register file sizes for the single and two-level organizations.

tially result in energy savings. The energy per access was estimated using CACTI-2.0. For each port that was not accessed in a cycle, we assumed that it consumed 10% of its maximum energy. For the two-level organization, we also considered the cost of transfers across the bus between the L1 and L2. We also attempted to model the additional structures (usage table, copy list) with CACTI-2.0. It must be pointed out that modeling these auxiliary structures as RAMs represents one design point, which might not necessarily be the most optimal in terms of energy efficiency.

Table 3 shows the various components of the average energy consumption for the 100-entry monolithic register file and the two-level register file. When the auxiliary structures are not considered, the two-level organization consumes 11% less energy. The L2 register file is a single ported structure and it does not add significant energy overhead. Rather, there is a drastic L1 energy savings due to the reduction in the size of the heavily ported L1 structure. When the energy from the other structures is taken into account, the two-level organization ends up consuming 16% more energy than the monolithic base case. Most of this energy comes from the various decoders in these structures, which emphasizes the need to design them carefully, so that decoders from other stages can be integrated with them (as described in Section 3).

To show behavior on individual applications, we also show IPS numbers for three of the organizations in Figure 4. The first two bars show IPSs for single-level register files with 60 and 100 registers, while the last bar shows IPSs for a two-level organization with 60 registers in the L1 and 40 in the L2. All the programs show an IPC improvement in going from a 60-entry L1 to a 100-entry L1, though the increased access time does not always translate into higher IPS. The two-level organization does a very good job identifying 'inactive' registers and moving them to the L2, often

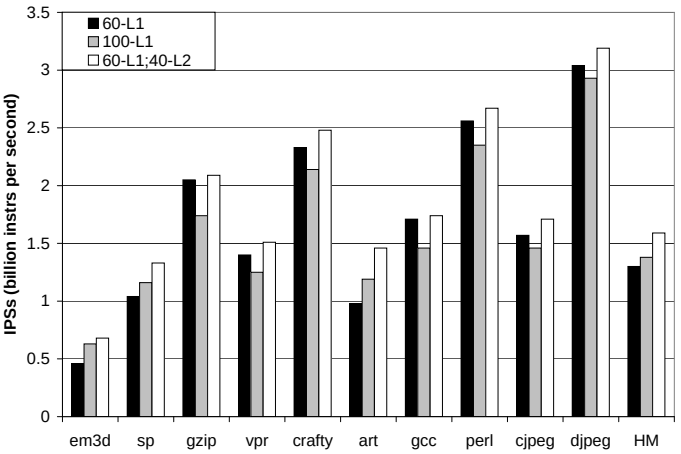

Figure 4. IPSs for individual applications for single-level register files of sizes 60 and 100 and a two-level organization with a 60-entry L1 and 40-entry L2.

achieving IPCs comparable to the larger single-level register file, while maintaining a low access time and exceeding its IPS. The L1-L2 copies accounted for about 18% of all accesses to the L1. Very few L2-L1 copy-backs were required on each mispredict and in most cases, these were effected without stalling the front-end. The program *vpr* was the only exception - in a 50M instruction simulation, it effected 22.5M copies from L1 to L2, of which 1.6M had to be copied back on mispredicts, resulting in as many as 0.8M front-end stalls, and resulting in a minor 0.016 CPI loss.

5.3 Banked Register File Evaluation

We now study the implications of a register file that has a single read and a single write port, but is organized into N banks. For $N = 4$, it has the same peak read and write bandwidth as the base case, but incurs an IPC degradation

because of the added constraint that two values cannot be sourced in the same cycle if they lie in the same bank. The banks are high-order interleaved, *i.e.*, the high-order operand address bits select the correct bank to read or write.

We use a processor with the parameters described in the earlier section and use a 160-entry register file (int and fp, each) in order to study a high IPC model with the most potential for bank conflicts. For the banked structure, we show results with four and eight banks, with the registers distributed equally among these. At the time of rename, free registers are picked out of the banks in a round-robin order to ensure that there is a fair distribution of registers among the banks. There can be at most eight outstanding partial reads at any given time (one for each integer functional unit) and at most eight results can be buffered due to a failure to get access to the write ports.

Figure 5 shows IPC results for various organizations. The first bar shows a conventional organization with 24 ports. The second bar shows the chosen base case that has a single bank, allowing four reads and four writes in a cycle. As can be seen, the chosen base is within 2% of the 24-ported register file. The third bar shows the effect of using four banks, each with one read port and eight write ports, while the fourth bar also has four banks, but only a single read and a single write port. Thus, the third bar shows the penalty imposed by conflicts for read ports and the fourth bar shows the additional penalty because of write port conflicts. When compared with the organization with 24 ports, there is a 1% drop in IPC because of read conflicts. The degradation increases to 5% when write conflicts are also taken into account. (However, the IPC degradation when compared with the organization with the same read/write bandwidth is only 3%.) The most significant IPC degradations are seen for some of the high ILP programs, like *djpeg, perl, crafty,* and *gzip* - the greater the number of instructions issuing every cycle, the greater the number of bank conflicts. The IPC for *djpeg* is about 10% worse than the non-banked register file with the same bandwidth.

For the 4-banked organization, each functional unit input multiplexes one of the four values read from the register file (Figure 2). If more than one functional unit attempts to read the same register in the same cycle, this can be done without having to read that value twice, *i.e.*, the value is read once and multiplexed to both functional units without any added logic. Instead, if this value is read twice, it leads to a great number of bank conflicts, resulting in a further 4% IPC loss. This happens because some registers have many consumers in the same cycle, most notably, the stack pointer.

To reduce bank conflicts, we attempted simple schemes where register mappings were steered to specific banks to avoid conflicts. Steering the two operands of the same instruction to different banks did not yield much benefit. Since one of the operands is usually read off the bypass net-

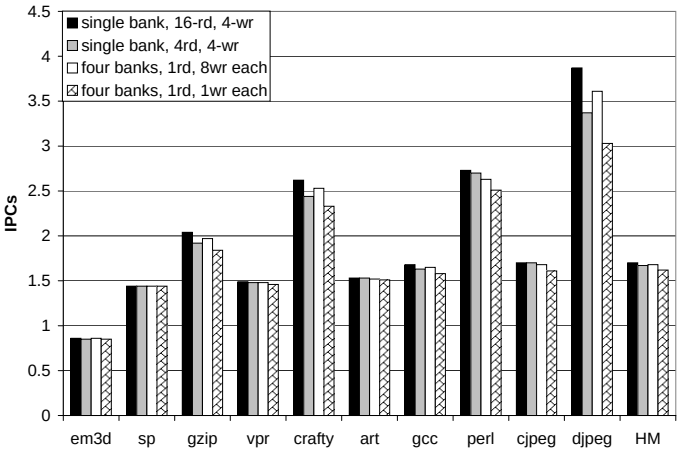

Figure 5. IPCs for the conventional and the base case (single bank with four read and four write ports), and for organizations with four banks. The third bar shows the effect of limited read ports, the fourth shows the effect of limited read and write ports.

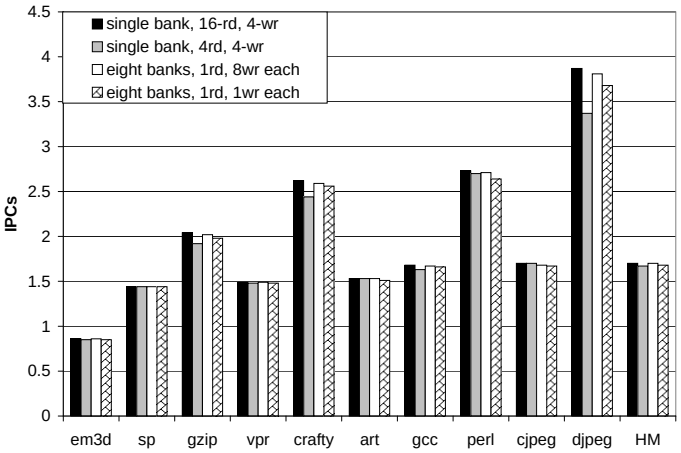

Figure 6. IPCs with eight bank organizations.

work, this phenomenon is not a source for conflicts. We tried to see if two instructions issued in the same cycle on a regular basis. If such instructions were identified, the source (and also their destination) registers could be mapped to different banks to help reduce the chances of a conflict. However, due to the unpredictable nature of scheduling because of cache misses, resource conflicts, *etc*, we observed that two instructions that issued together in a cycle were likely to do so again during their next instantiation with a probability of only 20%. Simple predictors that exploited this property to steer registers to specific banks showed negligible improvements. More complicated predictors could possibly do a better job, but because the maximum improvement possible was only 5%, we did not attempt these.

The easiest way to reduce conflicts is to simply implement more banks. Figure 6 repeats the experiments in Figure 5, but with eight banks. There is almost no degradation because of read port conflicts. The write port conflicts result

Organization			IPC	Access	IPS	Energy
L1/L2	banks	ports/bank		time (ns)	(BIPS)	pJ/instr
160	1	16-r, 8-w	1.70	2.51	0.68	1524
160	1	4-r, 4-w	1.67	1.35	1.24	368
100	1	4-r, 4-w	1.54	1.12	1.38	258
60	1	4-r, 4-w	1.18	0.91	1.30	187
60/40	1	4-r, 4-w	1.45	0.91	1.59	300
160	4	1-r, 1-w	1.62	0.97	1.67	84
160	8	1-r, 1-w	1.68	0.98	1.71	107
100	4	1-r, 1-w	1.49	0.94	1.59	73
60/40	4	1-r, 1-w	1.39	0.91	1.53	183

Table 4. Summary for various organizations.

in a 2% IPC loss when compared with the 24-ported register file. However, the cost of an eight-banked structure is a potential increase in access time, which we now evaluate.

In determining the access time of the monolithic structure with fewer ports (four read and four write), we have to take two additional delays into account. First, the signal read off the bitline has to be distributed via a driver to as many as 16 possible datapaths (eight integer units, two operand inputs each). At each of these datapaths, there exists a multiplexor that then selects the data read out of one of the four read ports and forwards it to the functional unit input. We modified CACTI-2.0 to take these two effects into account. The conventional organization simply has an output driver that transmits the data to the functional unit. The fewer-ported structure has a buffer that feeds 16 output drivers. The output drivers (which are tristate buffers and serve as the multiplexors) also have a greater delay because four of them drive the same bus.

The four-banked organization has a similar output structure as the fewer-ported organization. Once the four values are read out, they follow the same path as in the latter. However, access time is reduced because the delay to read data out of each bank is smaller (each structure is one-quarter the size and has one-quarter the read and write ports). We also take into account the time taken to propagate a signal across the breadth of all the banks.

Table 4 summarizes the features of the four organizations evaluated. According to the access times obtained from CACTI-2.0, reducing the number of ports in the monolithic structure from 24 ports to 8 reduces the access time from 2.51ns to 1.35ns, a 46% drop, even when accounting for the additional delay of the buffer and output multiplexors. By further splitting the register file into 4 banks, each with one read and one write port, the access time is reduced by an additional 28% to 0.97ns. Of this delay, 0.11ns was because of the buffer and the mux and 0.24ns was because of the propagation delay across the breadth of all the banks. With the 8-banked structure, the access time increases slightly. Even though the access time for an individual bank decreases, it takes longer to propagate a signal across all banks. Given that these drastic access time reductions are possible with almost negligible IPC penalties, the IPS metrics for the banked organizations are correspondingly much higher - the 8-banked register file has an IPS that is 38% higher than the single-banked register file.

In terms of register file energy, the 24-ported structure consumes 1524pJ per instruction on average. The single-bank 8-ported structure achieves more than a factor of four lower energy consumption (368pJ/instr). The four-banked structure shows a further reduction by a factor of 4.4 for a per instruction consumption of 84pJ. Finally, by using the eight-banked structure, energy increases slightly to 107pJ/instr. This occurs because the additional decoders, bitlines, and wordlines of the eight-banked structure still dissipate energy under our model even when idle, although the energy of the selected bank is reduced.

The use of fewer ports introduces some logic in the select stage of the issue queue and some arbitration logic at the functional units. Our analysis has not taken into account the extra energy consumed within these structures. Given that the proposed register file organizations consume about 18 times less energy than the base case, we expect that these overheads would be comparably negligible. The power models based on Wattch [2] attribute very little power to the select logic when compared with the register file.

5.4 Combining the Two Techniques

So far, we have studied the two orthogonal aspects of the register file in isolation - the number of entries and the bandwidth. In this subsection, we see the effect of combining the two, *i.e.*, using a smaller banked L1 in conjunction with an L2 register file. The banked organization reduces access time as well as energy consumption for a marginal IPC loss, while the two-level organization also reduces access time but with a potential increase in energy consumption due to auxiliary structures.

Figure 7 shows the IPS of the combined two-level, banked approach as well as that of the individual techniques for each benchmark. Table 4 provides a breakdown of the performance numbers as well as average energy. In comparing the two-level, banked, and combined organizations, we find that IPS performance actually degrades slightly when the techniques are combined. The reason is that the access time improvement of splitting a banked organization into two levels is overridden by the IPC degradation incurred. With such a small number of registers in each bank to begin with, the bitline delay ceases to dominate the access time to the point where further reducing the number of registers in each bank via splitting into two levels has diminishing returns. Thus, even though the IPC degradation effects of combining the two techniques are additive (but no more than this), the reductions in access time are not. We also found that this held true for the larger register files likely to be implemented in simultaneous multithreaded processors.

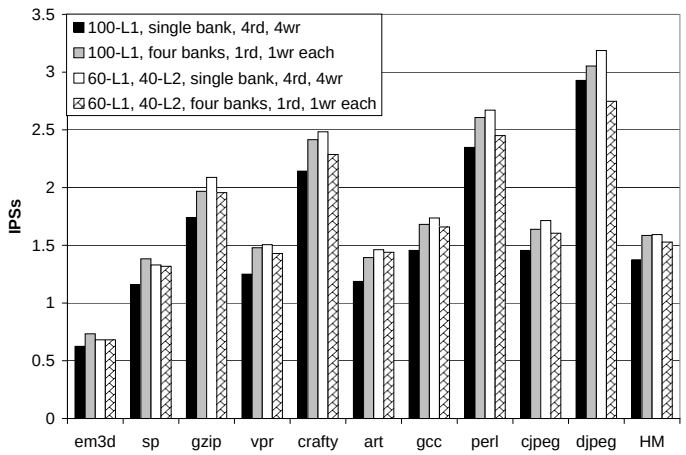

Figure 7. IPSs for the single-level base case, for the single-level banked organization, for the two-level non-banked, and the two-level banked organizations.

For example, with 512 total registers, an eight-way banked register file has an access time of 1.05ns in 0.18μ technology, while a two-level eight-way banked organization with 256 registers each in L1 and L2 has only a slightly lower access time of 0.99ns. Thus, we conclude that at least for the 0.18μ parameters that we used in our analysis, combining the techniques does not afford any advantage.

We also find that for a given number of registers (100 in this case), the two-level and banked organizations perform identically. The advantage of the two-level organization is its simpler layout compared to the banked organization, which requires many wires to span the breadth of the register file and many output multiplexers. If layout considerations are the overriding concern, then the two-level organization is the most effective means to reduce register file access time and increase IPS. If energy considerations are paramount, then the banked approach provides a significant energy savings in addition to a marked performance improvement.

6 Related Work

Cruz et al [6] use a two-level hierarchical inclusive register file organization (where the second level contains all values). In comparison, our organization uses an exclusive caching policy that avoids the IPC loss from missing in the first level. However, the penalty is a potentially larger size and access time for the L1. Hence, the choice of which organization works better would depend on the target frequency, the process parameters (the register file size that can be supported in a single cycle), and the benchmark set.

Zalamea et al [31] proposed a two-level register file that is compiler-controlled for reduced register spilling in the context of VLIW processors. The Cray-1 [23] also implemented a software-controlled two-level hierarchical register file. Yung and Wilhelm [30] explored the possibility of caching part of the register file with an LRU replacement policy in the context of an in-order processor. Swensen and Patt [25] proposed a hierarchical non-inclusive register file, where different banks have different sizes and speeds.

Processor implementations, such as the HP PA-8000 [13], maintain a logical register file that holds committed values, and the rename registers are maintained in a separate bank (perhaps in the ROB). Since a functional unit could source values in either bank, this partitioning into two banks does not result in a reduction in access time.

The conditions under which a register can be deallocated have been dealt with in detail by Moudgill et al [18]. Wallace and Bagherzadeh [27] and Monreal et al [17] propose delaying the allocation of registers until the time to actually write the value, thereby improving its utilization.

Partitioned non-hierarchical register file organizations have been proposed in the past [1, 4, 5, 8, 12, 15, 21]. These organizations have clusters of functional units, with each cluster having its own private register file. While these organizations reduce porting requirements per cluster, they still provide dedicated ports per functional unit, and they incur additional latency (in extra cycles) when values from other clusters need to be communicated. In our banked organization, the banks are adjacent and are treated as one structure. As a result, we pay a penalty in terms of a slightly longer access time as an operand could be sourced from any of the banks, which requires a multiplexor and the added delay of having to cross multiple banks. However, this choice makes it possible to have as few as a single read and single write port per bank. Such an organization was also proposed by Janssen and Corporaal [11] in the context of a VLIW processor. Their scheme requires compiler support and incurs a non-trivial IPC degradation. In comparison, our scheme does not require compiler support and uses a wider issue processor. We also quantify the effect of the added circuitry on access time and energy, and evaluate its impact on the performance of a dynamic superscalar processor.

7 Conclusions

The register file is a key bottleneck in modern dynamic superscalar processors. Both a large number of registers and many ports are necessary to support a large window of in-flight instructions and extract enough ILP. The access time of the register file is, however, critical in determining cycle time, requiring that its design be as simple as possible. The register file may also be a significant contributor to overall power consumption.

In this paper, we address the latency and energy consumption of the register file using two orthogonal approaches that can be combined. The novel contributions of the paper are: a hierarchical division of registers into those with active consumers and those waiting for precise

conditions (different from earlier partitioning proposals in being hardware-based and not compiler-based); the use of minimally-ported register file banks, which has not been studied in the context of dynamically scheduled processors; and a thorough evaluation of IPC, access time, and energy.

Our results show that the use of a two-level structure helps reduce the access time of the first-level register file in comparison to a single-level register file for roughly the same IPC. When using the instructions per second metric, the two-level organization performs 17% better than the best single-level organization. Using a banked single-port-per-bank register file organization reduces access times by a factor of more than two and energy consumption by a factor of more than 18 when compared to a conventional organization. These improvements are obtained without a significant degradation in IPC. The choice of technique — two level or banked — is dependent on design goals.

References

[1] A. Baniasadi and A. Moshovos. Instruction Distribution Heuristics for Quad-Cluster, Dynamically-Scheduled, Superscalar Processors. In *Proceedings of MICRO-33*, pages 337–347, Dec 2000.

[2] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. In *Proceedings of ISCA-27*, June 2000.

[3] D. Burger and T. Austin. The Simplescalar Toolset, Version 2.0. Technical Report TR-97-1342, University of Wisconsin-Madison, June 1997.

[4] R. Canal, J. M. Parcerisa, and A. Gonzalez. Dynamic Cluster Assignment Mechanisms. In *Proceedings of HPCA-6*, 2000.

[5] A. Capitanio, N. Dutt, and A. Nicolau. Partitioned Register Files for VLIWs: A Preliminary Analysis of Trade-offs. In *Proceedings of MICRO-25*, 1992.

[6] J.-L. Cruz, A. Gonzalez, M. Valero, and N. P. Topham. Multiple-Banked Register File Architectures. In *Proceedings of the ISCA-27*, pages 316–325, 2000.

[7] D. Bailey, et al. The NAS Parallel Benchmarks. Technical Report TR RNR-94-007, NASA Ames Research Center, March 1994.

[8] K. Farkas, P. Chow, N. Jouppi, and Z. Vranesic. The Multicluster Architecture: Reducing Cycle Time through Partitioning. In *Proceedings of ISCA-24*, 1997.

[9] K. Farkas, N. Jouppi, and P. Chow. Register File Considerations in Dynamically Scheduled Processors. In *Proceedings of HPCA*, 1996.

[10] L. Gwennap. PA-8500's 1.5M cache aids performance. *Microprocessor Report*, 11(15), November 17, 1997.

[11] J. Janssen and H. Corporaal. Partitioned Register File for TTAs. In *Proceedings of MICRO-28*, 1995.

[12] R. Kessler. The Alpha 21264 Microprocessor. *IEEE Micro*, 19(2):24–36, March/April 1999.

[13] A. Kumar. The HP PA-8000 RISC CPU. *IEEE Computer*, 17(2), March 1997.

[14] C. Lee, M. Potkonjak, and W. Mangione-Smith. Mediabench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In *Proceedings of MICRO-30*, pages 330–335, 1997.

[15] P. Lowney, S. Freudenberger, T. Karzes, W. Lichtenstein, R. Nix, J. O'Donnell, and J. Ruttenberg. The Multiflow Trace Scheduling Compiler. *Journal of Supercomputing*, 7(1-2):51–142, May 1993.

[16] D. Matzke. Will Physical Scalability Sabotage Performance Gains? *IEEE Computer*, 30(9):37–39, Sept 1997.

[17] T. Monreal, A. Gonzalez, M. Valero, J. Gonzalez, and V. Vinals. Delaying Physical Register Allocation through Virtual-Physical Registers. In *Proceedings of MICRO-32*, pages 186–192, Nov 1999.

[18] M. Moudgill, K. Pingali, and S. Vassiliadis. Register Renaming and Dynamic Speculation: an Alternative Approach. In *Proceedings of MICRO-26*, 1993.

[19] S. Palacharla, N. Jouppi, and J. Smith. Complexity-Effective Superscalar Processors. In *Proceedings of ISCA-24*, 1997.

[20] J. Rivers, G. Tyson, E. Davidson, and T. Austin. On High-Bandwidth Data Cache Design for Multi-Issue Processors. In *Proceedings of MICRO-30*, pages 46–56, 1997.

[21] S. Rixner, W. Dally, B. Khailany, P. Mattson, U. Kapasi, and J. Owens. Register Organization for Media Processing. In *Proceedings of HPCA-6*, Jan 2000.

[22] A. Rogers, M. Carlisle, J. Reppy, and L. Hendren. Supporting Dynamic Data Structures on Distributed Memory Machines. *ACM TOPLAS*, Mar 1995.

[23] R. Russell. The Cray-1 Computer System. In *Readings in Computer Architecture*, 2000.

[24] G. Sohi and M. Franklin. High-Bandwidth Data Memory Systems for Superscalar Processors. In *Proceedings of ASPLOS*, pages 53–62, 1991.

[25] J. Swensen and Y. Patt. Hierarchical Registers for Scientific Computers. In *Proceedings of ICS*, pages 346–354, 1988.

[26] D. Tullsen, S. Eggers, and H. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proceedings of ISCA-22*, pages 392–403, 1995.

[27] S. Wallace and N. Bagherzadeh. A Scalable Register File Architecture for Dynamically Scheduled Processors. In *Proceedings of PACT*, Oct 1996.

[28] S. Wilton and N. Jouppi. An Enhanced Access and Cycle Time Model for On-Chip Caches. Technical Report TN-93/5, Compaq Western Research Lab, 1993.

[29] K. Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, 16(2):28–41, April 1996.

[30] R. Yung and N. Wilhelm. Caching Processor General Registers. In *Proceedings of the International Conference on Circuits Design*, 1995.

[31] J. Zalamea, J. Llosa, E. Ayguade, and M. Valero. Two-Level Hierarchical Register File Organization for VLIW Processors. In *Proceedings of MICRO-33*, Dec 2000.