URCS Technical Report 745
A High Performance Two-Level Register File Organization *

Rajeev Balasubramonian†, Sandhya Dwarkadas†, and David Albonesi‡
† Department of Computer Science
‡ Department of Electrical and Computer Engineering
University of Rochester
April, 2001

*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.

Abstract

Dynamic superscalar processors execute 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 are assigned a new physical register. A large register file helps improve the instruction-level parallelism (ILP), but has a detrimental effect on clock speed, especially at future technologies. In this paper, we propose a two-level register file organization, where the first level only contains values that potentially have active consumers in the pipeline. The second level contains those values that are going to be used only in the event of a branch mispredict or an exception, and has minimal port requirements. Adding the second level shows overall speedups of 1.22, 1.06, and 1.19 relative to an architecture without a second-level cache for three different processor models for a varied benchmark set. A small first-level register file supported by a second-level register file can support as much ILP as a much larger single-level register file, thus having favorable implications for clock speed and power.

1 Introduction

Most modern high-performance processors use an out-of-order dynamic superscalar core to extract instruction-level parallelism (ILP). These processors examine a large window of in-flight instructions to find ready and independent instructions every cycle. The size of this window is one of the key determinants of the performance that can be achieved. A large window not only increases the probability of finding a ready instruction, it also allows cache misses to be fetched well in advance of the loaded data being used. 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).

There are three potential bottlenecks that can limit the extraction of ILP in an application. The register file size has a direct impact on the number of in-flight instructions within the processor as 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. Similarly, dispatch is also stalled when the issue queue gets full and this condition persists until an instruction is issued. Finally, the ROB (which is usually larger than the register file size) can occasionally also get full and dispatch has to wait till the oldest instructions commit.

The register file size can often be the bottleneck as it lies on critical paths determining the cycle time. A 4-wide processor typically has at least 10 register file ports[1], allowing only a limited set of registers to be accessible in a single cycle. In the future, frequencies will become much higher and delays of wires will dominate even more at smaller technologies [16, 13], thereby further exacerbating the problem. Having a large register file with a multi-cycle access time does not solve the problem either. For example, a 3-cycle register file access time would require three levels of bypassing among the functional units, which then goes on to add to the bypassing delay, another cycle-time critical

[1]The number of ports is usually lower than the 12 that would be required for a 4-wide issue processor in order to avoid issue stalls. This decision is based on the probability of needing all 12 ports.

path [16]. A multi-cycle register file access time would also degrade ILP by increasing the branch mispredict penalty and the register file pressure by increasing register lifetimes. Further, pipelining the register file is not a trivial task as it is a RAM structure. The register file also consumes a significant portion of chip power, around 10% according to the power models based on Wattch [1].

Given these constraints, the register files in modern dynamic superscalar processors have been very modestly sized. The Alpha 21264 [11] has as many as 72 physical registers (integer (int) and floating point (fp), each), 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 ILP because of the inter-cluster communication. Farkas et al [8] showed that larger register file sizes resulted in improved ILP even as the sizes were increased beyond 128 entries, but no modern dynamic superscalar processor has come close to supporting that large a size because of cycle time constraints.

These constraints suggest the need for a more efficient register allocation policy in order to support as large an instruction window as possible. In a conventional processor design, a physical register is allocated at the time an instruction is dispatched and this mapping remains active until the next instruction with the same logical destination register is committed. During this register lifetime, there is a long phase initially when the register contains no value at all and there is an even longer phase later when the register value does not get read. The register lifetimes and hence the register pressure can be reduced firstly by delaying the allocation of the actual physical register and secondly by eagerly freeing the register once it has been consumed. The former technique has been evaluated by Monreal et al [14] by using a new set of register names that they call virtual-physical registers. In this paper, we exploit the latter opportunity.

We propose a two-level exclusive register file organization. Registers are allocated from the first-level (L1) register file at the time of dispatch. As soon as a register value is completely consumed by all instructions that source the value, we release the physical L1 register back into the free pool. However, in the event that the value is needed because of a branch misprediction or an excepting instruction, it must still be retained and is stored in the second-level (L2) register file. The L2 register file therefore contains all values that will not be read unless there is a misprediction or an exception. The L1 register file now contains a smaller set of only those values that will be sourced by the functional units, thus enabling a faster clock. The L2 register file has minimal port requirements and is smaller than the first level. If there is a misprediction, values are copied back from the second level into the first. Since this can be overlapped with the fetch of instructions along the correct path, there is usually no addition to the branch mispredict penalty.

The organization of the paper is as follows. Section 2 describes the proposed register file organization in more detail. Section 3 provides a quantitative evaluation of the architecture. Section 4 compares and contrasts our design to the extensive literature describing prior work in order to identify our contributions. Finally, we make concluding remarks in Section 5.

2 The Two-level Register File

2.1 A conventional register file organization

Dynamic superscalar processors like the Alpha 21264 [11] and the MIPS R10000 [22] use a simple physical register allocation policy, 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 <- ...
    ...                     ...
    end                     end
    x:                      x:
6:  ... <- lr5              ...  <- 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 is encountered. 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 and releases it at the time of commit. However, this also leads to long register lifetimes. If the instruction writing to pr22 (instruction 4) were a long latency load from memory, it would hold up the commit stage for potentially 100's of cycles, which would imply that pr18 would not be freed even if its value had already been consumed by instruction 2.

2.2 A two-level register file organization

With the two-level register file, we modify the allocation policy so that the L1 only contains values that have potential readers. A physical register from the L1 is allocated when an instruction is dispatched. For each physical register in the L1, we keep track of the number of subsequently dispatched instructions that use the same physical register as source operand. This value gets

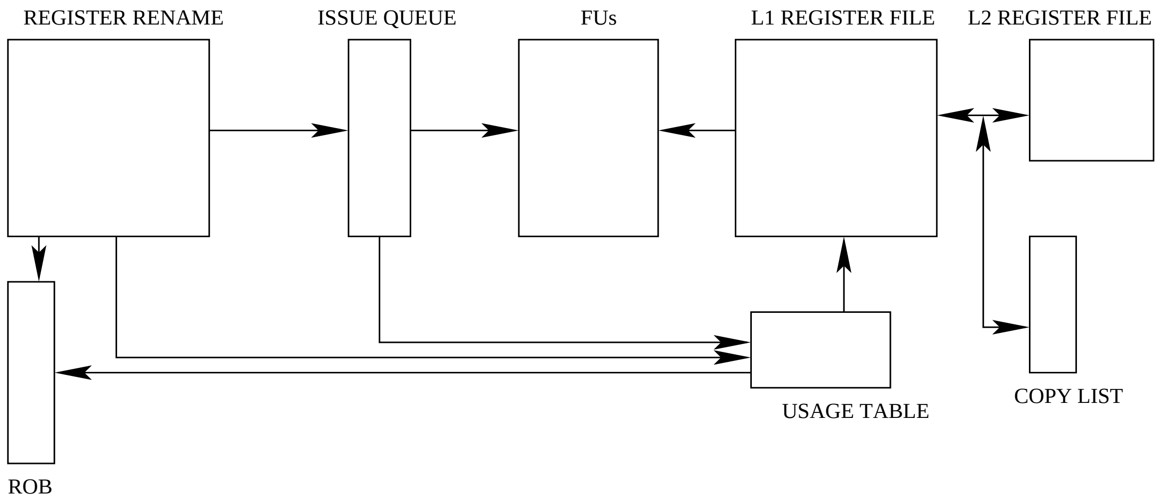

Figure 1: The two-level register file organization

decremented when the sourcing instruction actually gets issued. In addition, we also keep track of whether the physical register is the latest mapping for its corresponding logical register. A subsequent instruction that remaps the logical register to some other physical register indicates that the previously mapped physical register is no longer the latest mapping. This means that no more instructions sourcing that physical register will be dispatched (assuming all branches have been predicted correctly). Once a physical register has been written, has no more readers, and the corresponding logical register has been remapped, the value is then copied to a free physical register in the L2 and the L1 register is released back into the free list. The ROB is updated so that when the corresponding instruction commits, it releases the L2 physical register into the free list. An additional hardware structure keeps track of this copy that was made. If a branch misprediction is detected, this structure is looked up to identify those values that are still 'live' and need to be copied back into the L1, i.e, values that might yet have consumers along the correct path. This copy is overlapped with the fetch of instructions along the correct path and does not add to the mispredict penalty. Details of these operations are discussed next.

2.3 Microarchitectural changes

The rename stage of the pipeline behaves just as in a conventional processor. During this stage, the register map table is looked up to find the physical register mappings for the source operands and the previous mapping for the logical destination register. The instruction is entered in the ROB along with this previous mapping (the physical register to be released back into the free list when the instruction commits). A new physical register is removed from the free list to map the logical destination register for the instruction and this is entered in the register map table. At every branch, the current logical to physical register mappings are checkpointed, so that they can be quickly recovered in the event of a mispredict.

We introduce a new hardware structure, shown in Fig 1, that monitors the usage statistics for the L1 physical registers. For every L1 physical register, this *Usage Table* maintains the following information -

- The corresponding logical register name.
- A 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[2].
- A single bit (called the *Overwrite* bit) indicating if the physical register is the latest mapping for its logical register. This bit is set when it is not the latest mapping.
- 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.
- The sequence number for the branch immediately preceding the next instruction that writes to the same logical register.

Most of the information required to update this table is readily available during the rename stage. Note that we have introduced a sequence number to identify the various in-flight branches. This would require a small counter with as many bits as log(ROB_size). If the processor is running out of L1 physical registers, the *Usage Table* is examined and those registers that have their counter at zero, have a result in them, and have their *Overwrite* bit set are deemed useless. These values are then copied into the L2 (provided there are free L2 physical registers) and the L1 registers are released into the free pool. We add another field, the L2 register id, to the ROB. This gets written into when the copy is made. At the time of commit, if this id is non-zero, the L2 register is released back into the free pool instead of the L1 register.

We introduce another structure that keeps track of all the copies made, so that quick recovery can be initiated in the event of a branch mispredict. This *Copy List* maintains the following information -

- The logical register name.
- The L1 physical register name that had earlier contained the value.
- The L2 physical register name.

[2] The instruction would also decrement the counter if it were being squashed as a result of a branch mispredict.

- 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., this is the period during which instructions sourcing this value are dispatched. If a branch with a sequence number between these two sequence numbers (both inclusive) is determined to be a mispredict, then the corresponding physical register will have to be reinstated back in the L1 as instructions along the correct path may want to source that value. Over the subsequent cycles, all such values (referred to as the 'live' set) are copied back into the L1. There will necessarily be free L1 registers to enable this copy, as illustrated here.

Assume instruction A writes to physical register X, instruction B writes to the same corresponding logical register, and subsequently, the value in X is copied into the L2 and X is released back in the free pool. X can now only be mapped to instructions that follow instruction B, as the dispatch of B is a necessary condition for the copy into the L2 and the release of X. After a branch mispredict is discovered, if we need to reinstate the value produced by A, then it means that the branch would have to come before instruction B. Hence B and all its successors would have to be squashed and physical register X would be back in the free pool. Hence, the same physical register as before could be used to copy the value back. This also means that the checkpoint mechanism in the register map table need not be modified. Since the next instructions to source the values in L2 would have to be fetched from the correct path, it would be a few (typically more than 5) cycles before they are actually read. This allows sufficient time to do the copy, making it possible to have very few read ports within the L2.

The processor should also recover to a valid register file state on an exception. Since exceptions are not as frequent as branch mispredicts, most designs (the MIPS R10000 [22], for example) choose to go with a slow recovery process in order to keep the hardware simple. The ROB is usually traversed in reverse order and register mappings are gradually undone. In the case of the two-level register file, in addition to undoing register mappings, values may have to be copied from the L2 to the L1. The steps to be taken are illustrated here with an example.

The instruction raising the exception would be at the head of the ROB. We start by looking at the branch immediately following this instruction. Its 'live' set is first reinstated (as if it were a mispredict). Further, the effect of instructions between the excepting instruction and the branch would also have to be undone. The ROB is traversed starting at the branch instruction and moving backward (in time) toward the excepting instruction. For each entry, if the earlier mapping for the logical register resides in the L2, it would have to be copied back into the L1. Because of the traversal of the ROB, this process is likely to take a number of cycles, even though branches are usually fewer than 10 instructions apart. Since exceptions are infrequent and the process of recovering the register mappings is of comparable complexity, we expect that this would not be a large overhead.

2.4 Complexity of the proposed structures

The introduced mechanisms do not interfere with the register rename process. The register map table could be either a CAM or a RAM structure [16]. On a branch mispredict, checkpointed mappings are reinstated, just as in a conventional processor.

The *Usage Table* that monitors the status of the L1 physical registers has as many entries as the L1. Each entry in the table is likely to not exceed five or six bits. The counter for the number of active consumers is the most complex component, but can be quite easily approximated with a smaller structure. In the worst case, in a 4-wide processor, as many as four consuming instructions could be dispatched and as many as four could be issued, requiring eight inputs to the counter. Since this is a rare event, even as few as 2-4 inputs would suffice for most cases. If there are more consumers that cycle, an overflow bit could be set, indicating that the counter is inaccurate and the register should not be considered as a candidate for copying into the L2. Similarly, the size of the counter could be restricted to 3-4 bits. Note that the *Overwrite* bit would have to be checkpointed on every branch so it could be recovered in case of a mispredict. Correspondingly, the number of active consumers would be decremented as instructions are removed from the issue queue (squashed) due to the mispredict. The table look-up to determine L1 registers that are candidates for copying to the L2 would simply require combinational logic for each entry.

The *Copy List* must be as large as the L2 size. Most fields within it are also not likely to exceed five bits. The L2 register name would be the largest field (log(L2_size bits, which is less than seven for all the cases considered in this paper) as the L2 could be quite large. We propose a CAM structure as the entries would have to compare their branch sequence numbers with that of the mispredicted branch while copying values back into the L1.

The L2 register file could have more entries than the L1. As we shall show in Section 3, its port requirements are quite minimal. Even allowing for as few as one read and one write port is enough for near-optimal performance. Hence its access time and energy requirements are likely to be a lot less than the heavily ported L1. Even the *Copy List* requires just a single port as only a single copy is done each cycle.

The copying process need not require additional ports in the L1. Very often, the L1 register ports end up not being used 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.

We have also added an extra field to the ROB elements to store the L2 register tag when applicable. The ROB is a simple CAM structure with relatively small fields (as many as log(L1_size)

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 penalty	8 cycles
Fetch, dispatch, issue, and commit width	4
L1 I and D-cache	64KB 2-way, 2 cycles
L2 unified cache	2MB 8-way, 15 cycles
TLB	128 entries, 8KB page size
Memory latency	66 cycles for the first chunk
Memory ports	2 (interleaved)
Integer ALUs/mult-div; FP ALUs/mult-div	4/2; 2/1

Table 1: Simplescalar simulator parameters

or log(L2_size) bits), is likely not a cycle time critical component, and does not affect branch mispredict penalty. Hence, we expect no impact on cycle time because of this modification.

3 Results

3.1 Simulation methodology

We used Simplescalar-3.0 [2] for the Alpha AXP instruction set to simulate a dynamically scheduled 4-wide superscalar processor. The simulation parameters are 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)). We model a physical register file and mapping of logical registers to them. We use a split integer and floating-point issue queue and register file. In all our experiments, the maximum number of in-flight instructions (the size of the ROB) is set to be slightly more than the number of physical registers, as some of the instructions (stores, branches, etc) do not have destination registers.

As benchmarks, we use a wide variety of programs, from the Olden [18], SPEC2000, SPEC95, and NAS parallel benchmark [7] suites. These include both memory and non-memory intensive programs. Note that memory-intensive programs are more likely to run out of physical registers because they tend to stall the commit stage with long latency operations. 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

Benchmark	Input dataset	Simulation window	L1 miss rate
em3d (Olden)	20000 nodes, arity 20	1000M-1010M instructions	23%
mst (Olden)	256 nodes	14M instrs (whole program)	9%
perimeter (Olden)	32Kx32K	1500-1520M instrs	9%
mg (NAS)	A, uniprocessor	2500M-2550M instrs	8%
sp (NAS)	A, uniprocessor	2500M-2550M instrs	13%
gzip (SPEC2k Int)	ref	2000M-2050M instrs	1%
vpr (SPEC2k Int)	ref	2000M-2050M instrs	2%
crafty (SPEC2k Int)	ref	2000M-2050M instrs	1%
parser (SPEC2k Int)	ref	2000M-2050M instrs	2%
art (SPEC2k Fp)	ref	300M-350M instrs	22%
lucas (SPEC2k Fp)	ref	2000M-2050M instrs	9%
apsi (SPEC95 Fp)	ref	200M-250M instrs	7%

Table 2: Benchmark description

'Small' base case	Issue queue - 20 (int) and 15 (fp), ROB - 100 L1 physical register file - 48 (int and fp, each)
'Medium' base case	Issue queue - 20 (int) and 15 (fp), ROB - 100 L1 physical register file - 72 (int and fp, each)
'Large' base case	Issue queue - 30 (int) and 30 (fp), ROB - 200 L1 physical register file - 72 (int and fp, each)

Table 3: Parameters for the three base cases

Alpha 21164 at the -O3 (highest non-architecture-specific) optimization level. The program code uses 32 integer and 32 floating-point logical register names.

3.2 Performance results

The sizes of the issue queue, register file, and ROB determine the number of in-flight instructions that can be supported. Depending on the technology parameters and the circuit-level implementations, the sizes of these structures may vary, resulting in different bottlenecks. Hence, we use three different base cases with differently sized core structures that are summarized in Table 3. We start by evaluating the two-level register file in the context of an existing modern processor (the 'medium' base case), like the Alpha 21264 [11]. The small size of the issue queue often limits the performance that can be achieved with a second-level register file. Using a larger issue queue such as in the 'large' base case helps remove this bottleneck, allowing the second level register file to be fully exploited. We then repeat our experiments in the context of a 'small' base case that represents

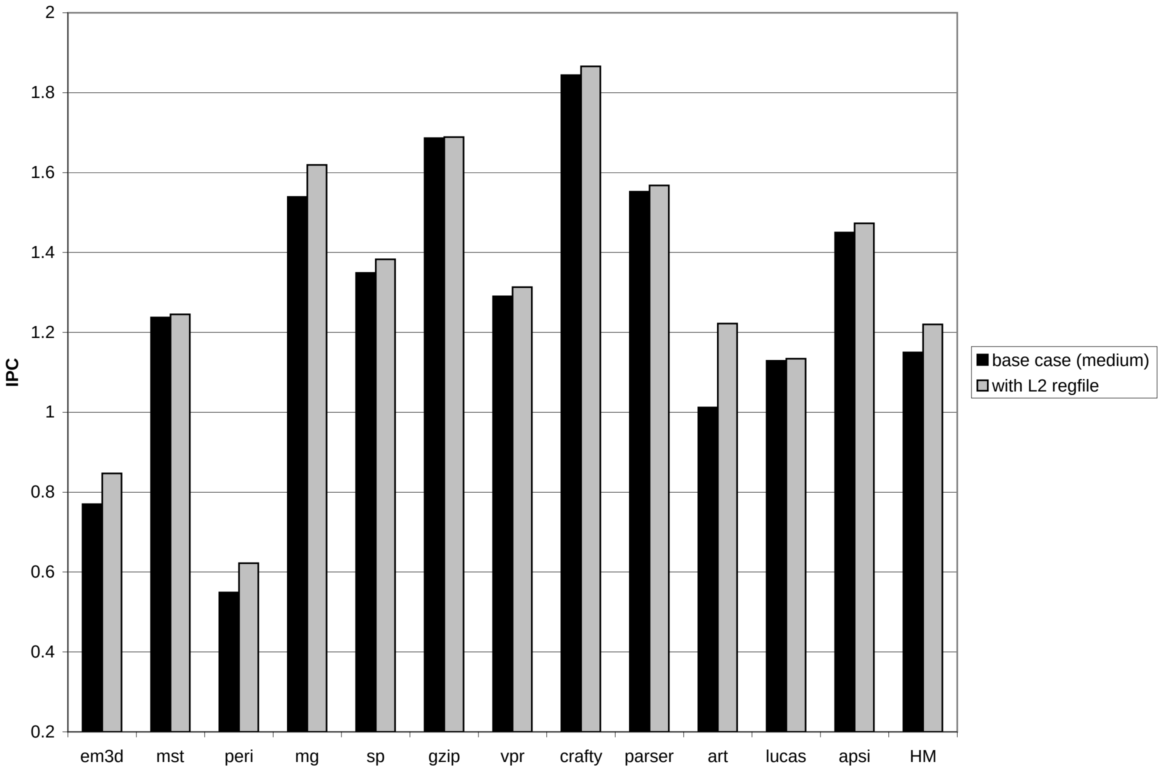

Figure 2: IPCs for the 'medium' base case. Also shows IPCs after adding an L2 register file of 40 entries (int and fp, each)

	Instrs simulated	Copies to L2 int, fp	ROB occupancy base / with L2	Br mispredicts/ br-predict rate	Copies to L1 int, fp	Surplus copies int, fp	Stalls
em3d	10M	1.7M, 0	70 / 89	31K, 95%	0.3K, 0	0, 0	0
mst	14M	0.1M, 0	31 / 31	27K / 99%	3K, 0	0.2K, 0	152
peri	20M	3.8M, 0	44 / 53	412K / 91%	88K, 0	0, 0	0
mg	50M	4.4M, 1.9M	58 / 78	22K / 100%	0, 0	0, 0	0
sp	50M	1.7M, 1.8M	70 / 79	219K / 89%	0.5K, 0	0, 0	0
gzip	50M	0.9M, 0	30 / 31	549K / 89%	85K, 0	1K, 0	698
vpr	50M	2.3M, 0	39 / 43	285K / 95%	196K, 0	10K, 0	6617
crafty	50M	1.8M, 0	32 / 34	451K / 93%	60K, 0	2.5K, 0	1402
parser	50M	1.4M, 0	33 / 34	584K / 93%	45K, 0	0.5K, 0	334
art	50M	11M, 0.1M	62 / 86	138K / 97%	36K, 0	0.3K, 0	268
lucas	50M	0.1M, 0	31 / 31	4K / 99%	0, 0	0, 0	0
apsi	50M	0.7M, 1.8M	48 / 51	34K / 98%	0, 0	0, 0	0

Table 4: Statistics for the various programs while using a 40-entry second level register file. The base case is the 'medium' organization.

a processor model geared towards faster clock speeds. This also helps us see if a small-sized L1 register file backed by an L2 can perform as well as a larger monolithic L1 in terms of IPC. Finally, we also evaluate how sensitive the two-level register file is to its various design parameters such as the number of ports and branch mispredict penalty.

We start with a base case that has an issue queue size of 20 (int) and 15 (fp), a physical register file size of 72 (int and fp, each), and a ROB of 100 entries. We refer to this base case as the 'medium' organization. To this base case, we add a 40-entry L2 register file, int and fp, each. Values are copied from the L1 only if fewer than 3 instructions were issued that cycle - this guarantees that there will be free register ports to perform the transfer[3]. We also limit the number of copies to two per cycle (to reduce port requirements in both the L1 and L2). We attempt the copy only if there are fewer than four registers in the L1 free register pool, i.e, there is a chance that we might run out of L1 registers in the subsequent cycles. When a mispredict is discovered, register values need to be copied back into the L1. We assume that up to eight 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 two copies 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 eight copies need to be made, we stall the fetch stage by an extra cycle for every two additional copies. Hence, we assume here that the L2 has two read and two write ports, while no additional ports are required in the L1.

Figure 2 shows instructions per cycle (IPC) for the 'medium' base case and for a processor with the two-level register file. Table 4 also lists various statistics pertaining to the latter organization that help us understand the results.

For half the programs, very marginal improvements in performance of under 2% are seen. These include the non-memory-intensive SPEC2k integer programs (gzip, vpr, crafty, and parser) and some of the memory intensive programs (mst, lucas, and apsi) as well. For these programs, the main bottleneck is the issue queue, with the dispatch stage being stalled for more than half the total execution time because the issue queues were full. This happens because these programs have few independent dependence chains and this causes instructions to wait for prolonged periods in the issue queue, causing it to fill up before the register file. Hence, providing the second-level register file did not alleviate the bottleneck. The average size of the window of in-flight instructions increased by only 1-4 instructions (as illustrated by the ROB occupancy in Table 4). The longer instructions wait in the issue queue, the longer it takes for values to be written into registers and for them to be read by their consumers. This decreases the number of valid candidates for copying into the L2 register file and fewer than 0.1M copies are made for mst and lucas.

[3]Some processors provide fewer register ports than the worst case requirement in the expectation that many instructions will not be sourcing operands from the register file. Hence, we use a more stringent criterion like lack of issued instructions to determine if there will be free ports.

For the remaining programs, the register file was the bottleneck, and using an L2 register file allowed the dispatch stage to make further progress. A large number of copies were made to the L2 and the average ROB occupancy increased by 9-24 instructions. A large number of copies is indicative of the fact that many register values have been consumed, which usually implies that there are many independent instructions in the window. Hence, looking in a larger window is usually fruitful as there are likely to be many independent instructions in the freshly fetched instructions as well. Art had the most number of copies, the largest increase in the ROB occupancy, and a resulting speedup as high as 1.21. The overall speedup for the entire benchmark set (while comparing the harmonic mean (HM) of IPCs) was 1.06.

Among these programs that showed reasonable improvements, two (perimeter and art) were also limited by poor branch prediction rates. This resulted in a relatively large number of copies back to the L1, but there were few copies required per misprediction. Perimeter required 88K copies for its 412K mispredictions and art required 36K copies for its 138K mispredictions, i.e, on average, fewer than 0.3 copies per prediction. Em3d and sp, which also had a number of mispredicts, required very few copies back into the L1 as the mispredicts were discovered quickly. For the organization modeled, as many as eight copies could be made without adding to the mispredict penalty and in most cases there were fewer than eight copies required on a mispredict. The number of copies in excess of eight in any cycle is referred to as the number of 'surplus copies' (the second-last column in Table 4) and is always a very small number. The additional number of stalls to the dispatch stage because of these 'surplus copies' is indicated in the last column. Even for the programs with the lowest prediction rates (SPEC2k int programs), the additional mispredict penalty does not exceed a few thousands of cycles (while simulating as many as 50M instructions). The most encouraging aspect of this study is the finding that mispredictions usually result in very few copies back to the L1.

In terms of performance, the overall improvement was only a modest 6%. One of the main reasons for this is the small size of the issue queue in our base case. Over the past few years, a number of proposals have been made to improve the design of the issue queue. Palacharla et al [16] describe a clustered organization for the issue queue, where successive instructions in a dependence chain are steered into the same cluster. By doing this, only the head instruction of each cluster needs to be checked for readiness every cycle. (Note that this kind of clustering is different from that where the issue queue, register file, and functional units are broken into multiple blocks with single-cycle communication within each block and multi-cycle communication between blocks.) Canal and Gonzalez [3] describe a hybrid issue queue structure where most waiting instructions look up a RAM structure to check for readiness, while a limited number of instructions have to go through an associative CAM-like search. Henry et al [9] propose a novel circuit design that reduces the cycle time for the associative search required in the issue queue. Given these developments, there is a strong likelihood that issue queues in future processors are going to be larger. In fact, the reason the issue queues in the Alphas are smaller than 20 entries is because using larger issue

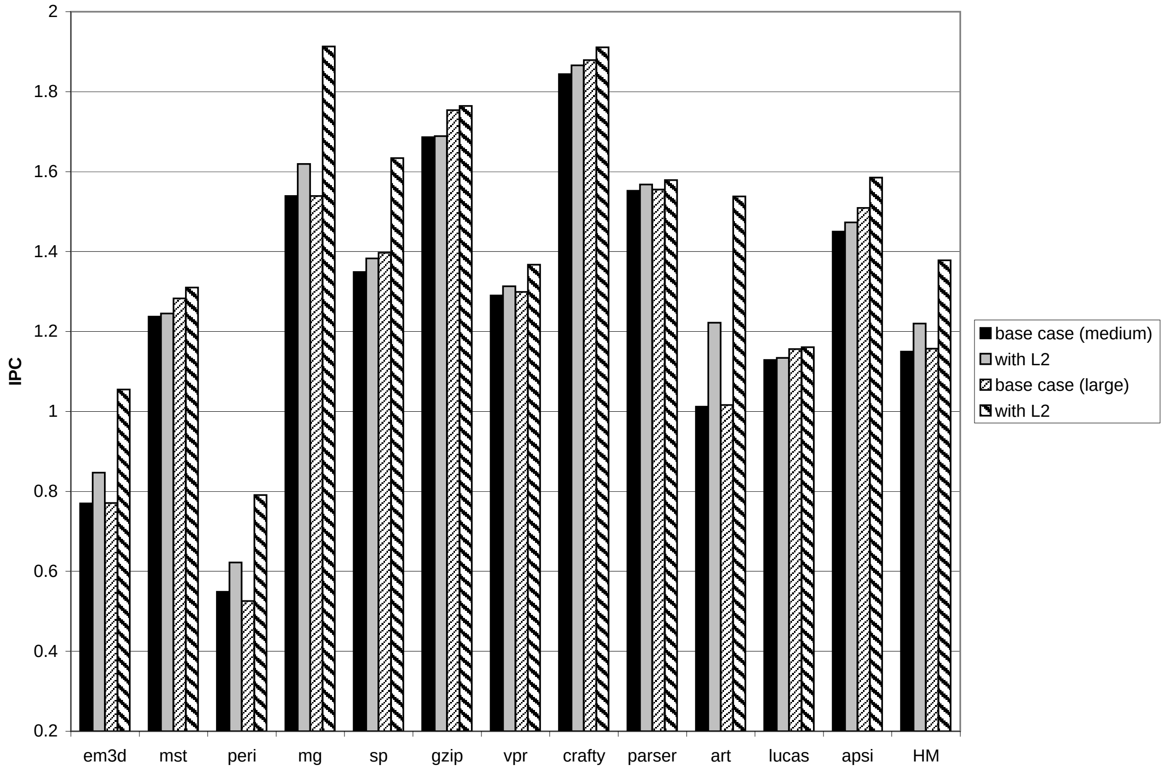

Figure 3: IPCs for the base case and with an L2 register file for two different processor models - 'medium' and 'large'.

queues does not result in better performance for the target register file sizes. Programs that have their issue queue as the bottleneck have few independent dependence chains. Hence, being able to look in a larger window does not buy much performance and increasing the issue queue size is a waste of energy. Programs that do have independent instructions within the window are limited by the size of the register file and hence increasing the issue queue does not result in any change in performance. As a result, the issue queue sizes of the Alphas are probably not limited by cycle time constraints and have been sized so as to be on the knee of the performance curve.

We therefore next evaluate the performance potential of using the two-level register file in a more aggressive processor design that allows for larger issue queues. Figure 3 starts by showing the results seen so far with the 'medium' base case. The latter bars then show results for the 'large' base case without and with the L2 register file. The 'large' base case has larger issue queues of 30 entries (int and fp, each), a larger ROB of 200 entries, and an L2 of 96 entries (int and fp, each). The L1 register file size remains at 72 entries as before. The first noteworthy point is that the 'large' base case is only marginally better than the 'medium' base case - about 1% better on average and never more than 5% better for any program. This confirms our hypothesis that increasing the issue queue size beyond that in the Alphas does not buy much performance. Adding an L2 register file allows

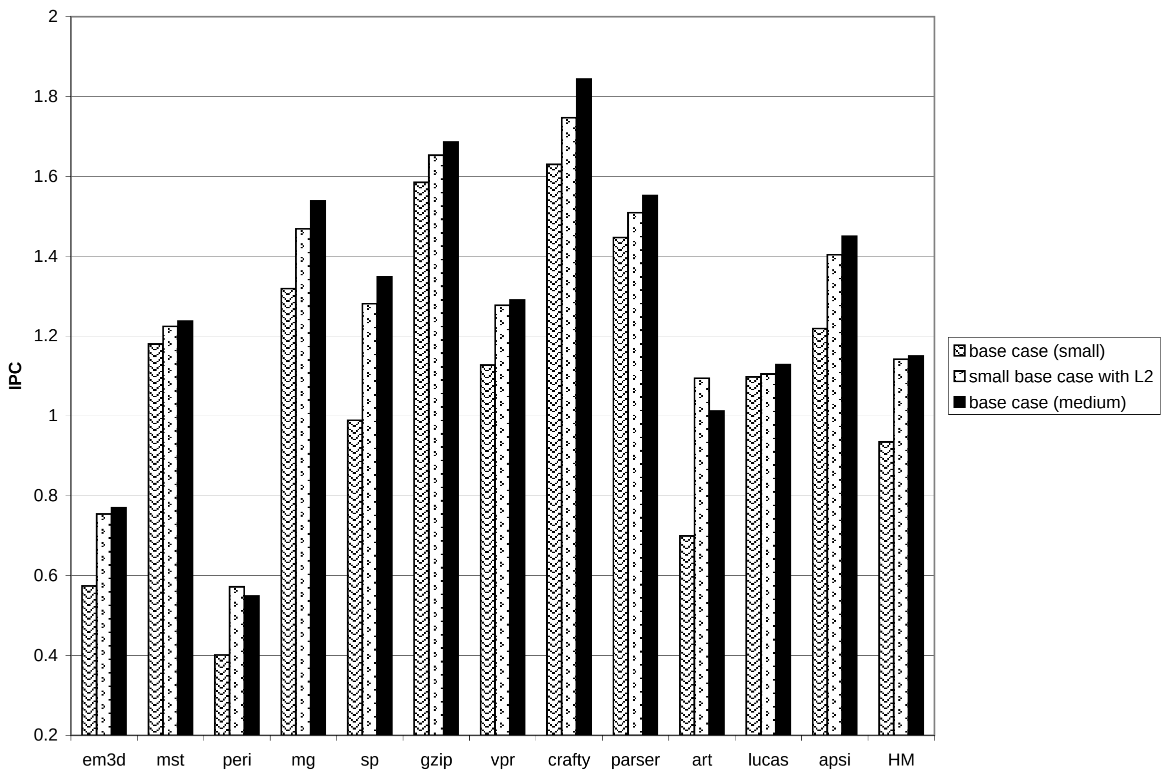

Figure 4: IPCs for the 'small' base case and with an L2 register file. Also shown is the 'medium' base case that has the same parameters except for a larger L1 of 72 entries (int and fp, each).

this more aggressive processor to fully exploit its potential. The programs that already showed reasonable improvements with the 'medium' processor model show drastic improvements with the more aggressive model. The slight increase in the issue queue size now causes the average ROB occupancy to go as high as between 82 to 134 for these 5 programs. This results in speedups as high as 1.37, 1.50, 1.24, 1.17, and 1.51. Even the other programs that had shown little improvement for the 'medium' processor model show slight speedups (up to 1.05 for apsi and vpr). The overall improvement for the 'large' processor model is 1.19.

Next, we start with the 'medium' processor model and shrink the L1 register file size to 48 entries to arrive at the 'small' processor model. The 'small' processor model is bound to have a much higher clock speed than the 'large' model as all structures have been shrunk down. Figure 4 shows IPCs for this 'small' base case without and with an L2 register file as well as IPCs for the 'medium' base case. Because of the smaller L1 register file size, there is a sharp drop in IPC when comparing the two base cases, even for the SPEC2k int programs. However, by using an L2 register file of 32 entries, we are able to quickly reuse the L1 registers and make much further progress. In fact, the HM of IPCs with the L2 is almost the same as that of the 'medium' base case. When comparing against the 'small' base case, the overall speedup is 1.22. Of the 48 L1 register entries,

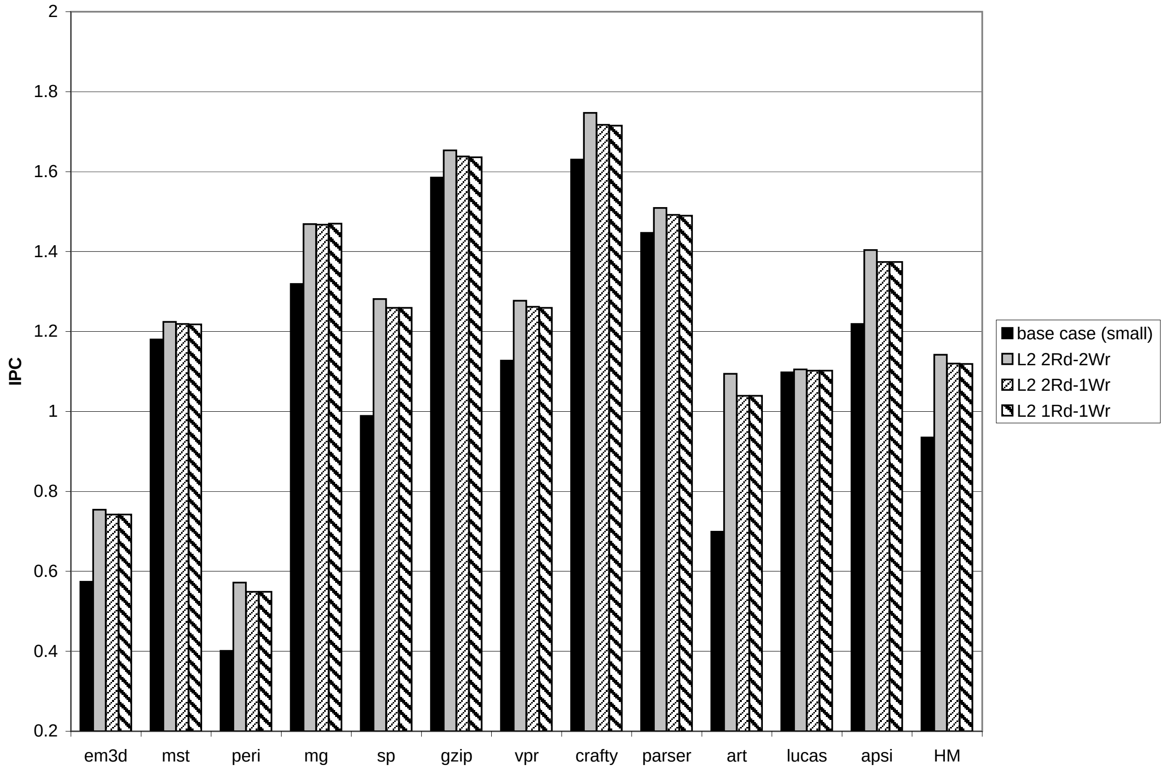

Figure 5: IPCs for the 'small' base case and with various L2 register file organizations added to it. The three organizations shown differ in their number of read and write ports.

32 of them can not be candidates for copying as they are the latest mappings for the 32 logical registers. Even with as few as 16 candidate registers, we are able to quickly find enough values that have been consumed and make as much forward progress as a base case with 24 more L1 registers. This implies that a register file with 72 entries usually has fewer than 48 active values, i.e, values with consumers in the pipeline. The use of an L2 allows the L1 to be downsized to this smaller size, thereby potentially saving power and greatly reducing the cycle time.

Finally, we evaluate the sensitivity of performance with respect to the number of L2 ports and the penalty during a mispredict due to 'surplus' copies. For brevity, we only show results with the 'small' processor model, which is the most sensitive to these parameters. Figure 5 shows results for the base case and with three L2 organizations that have 1-2 read and write ports. For the last organization that has only one read port, only one copy per cycle can be made back to the L1 in the event of a mispredict. We now assume that up to four copies can be made without adding to the mispredict penalty and every 'surplus' copy beyond that adds an extra stall cycle. The results show that there is a very marginal performance degradation by moving to these fewer-ported structures. The single read and write ported structure is only about 2% worse on average than the structure with two read and two write ports, with no program showing a degradation of more than 5%.

Since a single write port suffices, we can conclude that on average, not more than one candidate L1 register copy can be found every cycle. The non-effect of using a single read port comes as no surprise as we've already seen that on average, less than 0.3 copies need to be made per mispredict. The largest stall penalty was seen for vpr, which had 152,220 stall cycles in a 50M instruction simulation (about 0.003 additional cycles per instruction).

4 Related Work

There have been a few earlier proposals regarding a register file cache. Cruz et al [6] use a two-level hierarchical inclusive register file organization (where the second level contains all values). The first level is much smaller and only contains those values that are likely to be read in the near future. This includes values that were not read off the bypass network and possibly other values that were prefetched because their sourcing instruction is about to issue. For their processor model and benchmarks (Spec95), about a 10% and 2% IPC degradation was seen for Int and FP programs, when compared to a monolithic register file as large as the second level. This degradation comes about because a number of accesses miss in the first level. Even programs with relatively smaller register file requirements, like SPEC95 Int, see IPC degradation because of this. Yung and Wilhelm [23] also explored the possibility of caching part of the register file with an LRU replacement policy in the context of an in-order processor. Our organization uses an exclusive caching policy with different criteria for transferring data between levels. As a result, the first level has a 100% hit rate and only a marginal cost is occasionally paid for certain branch mispredicts. The penalty is that our first level is likely to be bigger than the first level proposed by Cruz et al, possibly implying a longer cycle time. 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 [24] also proposed a two-level hierarchical register file in the context of VLIW processors. The level 1 and level 2 registers are both visible to the compiler and it explicitly manages the transfer of values between the two levels. The improvement in performance comes about because of the reduced amount of register spilling, which reduces memory traffic, and because of the higher clock speed afforded by the small level 1. The Cray-1 [19] implemented a software-controlled two-level hierarchical register file. Swensen and Patt [20] proposed a hierarchical non-inclusive register file, where different banks have different sizes and speeds. Register allocation is performed by the compiler for the statically scheduled architecture in order to trade off size and speed according to instruction requirements. Our approach differs in that the two levels are managed automatically in hardware.

The performance and access time characteristics of a monolithic register file in a dynamically scheduled processor has been dealt with in detail by Farkas et al [8]. Rixner et al [17] give a taxonomy of various register file organizations and equations for their access time, area, and energy estimates. Partitioned non-hierarchical register file organizations have been proposed by them and by others in the past [5, 10]. These proposals resemble the clustered organizations seen in the Alpha 21264 [11], where the register file is replicated and each copy is accessible to only a few functional units so as to reduce the port requirements. A clustering organization proposed by Canal et al [4] also partitions the register file, but makes copies of values depending on whether a cluster is going to need it or not.

The conditions under which a register can be deallocated have been dealt with in detail by Moudgill et al [15] in the context of register renaming. The specific conditions under which we decide to copy a value from the L1 to the L2 have also been proposed by them in the context of a processor with imprecise exceptions. We describe a hardware mechanism to use these conditions in order to automate the transfer of register values between L1 and L2.

Processor implementations, such as the HP PA-8000 [12], maintain a logical register file that holds committed values, and the rename registers are maintained in a separate bank (possibly, as part of 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.

Another cause of inefficiency in the register allocation process is that physical registers are allocated at the rename stage, but it may take a number of cycles until the value is actually produced. Wallace and Bagherzadeh [21] and Monreal et al [14] propose mechanisms by which this allocation is delayed until the time to actually write the value.

5 Conclusions

The register file is a key bottleneck in modern dynamic superscalar processors. A large set of registers is necessary to support a large window of in-flight instructions and extract sufficient ILP. The access time of the register file is, however, critical in determining cycle time, requiring that it be as small as possible. In this paper, we have proposed and evaluated an exclusive two-level organization for the register file. The first level is similar to the one-level designs and is heavily ported, but is smaller in size as it only contains those values that potentially have active consumers in the pipeline. As soon as a value has no more consumers, it is copied into the second level. Copies from the L2 back to the L1 are necessary only if there is a branch mispredict or an exception is raised.

Our results show a modest overall speedup of 1.06 for our benchmark set for a processor with parameters resembling the Alpha 21264. The improvements are limited by the small size of the issue queue. With the assumption that the issue queue sizes in the Alpha are conservative and in anticipation that circuits for issue queues would improve and be able to support a larger queue, we then evaluated the two-level register file for two other processor models. One is geared towards higher IPC and has large structures and the other is geared towards faster clock speeds and has smaller structures. For these models, impressive overall speedups of 1.19 and 1.22 respectively were achieved. The larger model was able to support average window sizes as large as 134 instructions with only 72 physical registers (of which only 40 were candidates for reuse). With the smaller model, an L1 physical register file of 48 entries was able to find enough candidates to reuse and support as much computation as a register file of 72 entries. The discovery of a branch mispredict resulted in very few copies back from the L2 to the L1, which were overlapped with the fetch of instructions along the correct path, thus causing little or no performance degradation even in programs with high mispredict rates. A second-level register file with as few as a single read and a single write port had near-optimal performance. The additional hardware required to do the necessary book-keeping is relatively small and not on any latency critical path.

As future work, we would like to better quantify the performance difference between the two-level exclusive organization and other proposed register file caches. Such an evaluation would need detailed models for access times, area, and energy consumption. We would also like to see if our proposed model can help reduce the energy requirements of the register file. In terms of performance, we have already seen that a 48-entry (heavily-ported) L1 supported by a 32-entry (2-ported) L2 performs as well as a 72-entry (heavily-ported) L1 register file. In spite of the additional reads and writes because of the transfers between the L1 and L2 register files, the two-level organization is likely to consume less energy than the monolithic register file because the energy is dominated by the L1 register file and its size.

References

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

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

[3] R. Canal and A. Gonzalez. A Low-complexity Issue Logic. In *Proceedings of ICS*, 2000.

[4] R. Canal, J. Parcerisa, and A. Gonzalez. Dynamic Code Partitioning for Clustered Architectures. *International Journal of Parallel Programming*, 29(1):59–79, 2001.

[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 27th ISCA*, 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, N. Jouppi, and P. Chow. Register File Considerations in Dynamically Scheduled Processors. In *Proceedings of HPCA*, 1996.

[9] D. Henry, B. Kuszmaul, G. Loh, and R. Sami. Circuits for Wide-Window Superscalar Processors. In *Proceedings of ISCA*, 2000.

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

[11] R. Kessler. The Alpha 21264 Microprocessor. *IEEE Micro*, 19(2), March/April 1999.

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

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

[14] T. Monreal, A. Gonzalez, M. Valero, J. Gonzalez, and V. Vinals. Delaying Physical Register Allocation Through Virtual-Physical Registers. In *Proceedings of MICRO-32*, Nov 1999.

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

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

[17] 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.

[18] A. Rogers, M. Carlisle, J. Reppy, and L. Hendren. Supporting Dynamic Data Structures on Distributed Memory Machines. *ACM Transactions on Programming Languages and Systems*, Mar 1995.

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

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

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

[22] K. Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, 16(2), April 1996.

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

[24] 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.