Partitioning Register File to Reduce Access Time

Hyong-Youb Kim, Julie Rosser,
Kyle Bryson, Supratik Majumder

1 Introduction

In a wide superscalar processor, the amount of time it takes to execute an application depends on the instruction latency and the amount of instruction level parallelism (ILP) that can be extracted from the application. One important factor which influences the instruction latency is the number of cycles it takes to access the register file. Whether it is a CISC architecture or a RISC architecture, practically every instruction accesses the register file for one or more operands. Therefore, its hardly surprising that processor architects have over the years designed architectures which enabled a one cycle read/write access to the register file. Single-cycle accesses are also preferred because larger access times require deeper pipelines, and deeper pipelines induce more hazards, larger branch penalties, and more complex hardware for hazard detection and data forwarding.

Studies have shown that many of the ILP increasing techniques employed in wide-issue superscalar processors increase the demand for registers [3]. At the same time, an increased issue width of the processor requires additional register ports. Typically, a 4-wide superscalar processor needs to have at least 8 read ports and 4 write ports on its register file. Both of these affect the register file not only by making it much bigger but also much slower.

The access time of a register file consists of two distinct components: the wire propagation delay and the fan-in/fan-out delay. Register files typically contain long word-lines and bit-lines, which can take a long time to propagate a signal across their length. For the kind of register file structures considered here, the wire propagation delay is far greater than the fan-in/fan-out delay. Bigger register file and an increased number of ports result in a taller register file layout, which translates to longer word-lines and bit-lines [7], thereby increasing wire propagation delay. Also, wire delays do not at all scale with the silicon technology improvements. Thus as register files grow in size, with faster transistors (smaller feature sizes), its only exacerbates their delay problem.

Over the past decade, researchers have suggested a number of techniques for alleviating the problem of increased wire delay. Whenever a large block of silicon takes up a large fraction of the cycle time, it usually common to split the block up into smaller and more importantly faster pieces [6]. In the past, precious silicon area dictated logic reuse, but these days designers frequently duplicate logic to reduce wire lengths. We believe that these couple of ideas could be applied to the register files as well.

1.1 Hypothesis

We hypothesize that splitting up the register file would not only reduce its delay but also make it scale better with technology. At the same time the functional units could be duplicated to limit wire lengths from the register partitions to them. This effectively provides every partition of the register file with its own set of functional units. Obviously, inter-cluster communication would be costly, and this technique would be beneficial only if inter-cluster communication is rare. From the register usage patterns that are obtained from a few benchmarks programs, we conclude that this is indeed the case.

The extent of performance improvement that can be expected from our architecture is dependent directly upon this cluster locality in the instruction stream. It also depends upon how many cycles we attribute to accessing a monolithic register file. But we believe that there is room for some significant improvement by

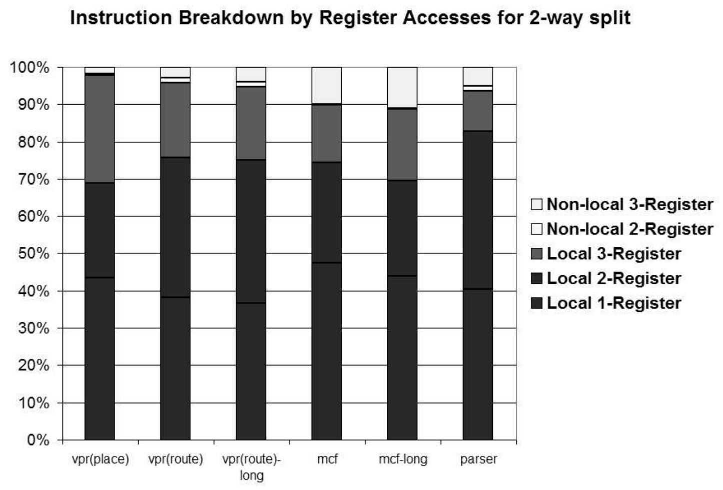

Figure 1: **Instruction breakdown of register accesses for a 2-way split of the register file.**

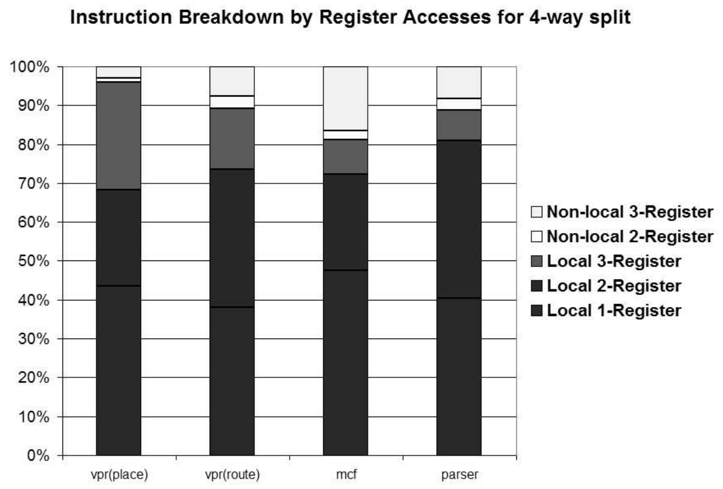

Figure 2: **Instruction breakdown of register accesses for a 4-way split of the register file.**

our technique. Our architecture would scale much better with process improvements, when the monolithic register file would become slower.

Figure 1 shows the register usage pattern of a few SPEC CPU2000 integer benchmarks for a two-way split of the register file. We split up the register file such that each half of the registers form a cluster (registers 0–15 in one cluster and registers 16–31 in the other). We then use the SimpleScalar tool set to obtain the register usage statistics. From the figure, it is apparent that almost 90% of the instructions, in these benchmarks, show high locality in their register accesses. In other words, almost 90% of the time we are likely to complete the instruction in just one cluster and would not need to pay the inter-cluster communication penalty. Figure 2 shows similar data for a four-way split of the register file in which each cluster has one quater of the registers.

Based on these preliminary findings, we hypothesize that splitting up the register file and clustering the

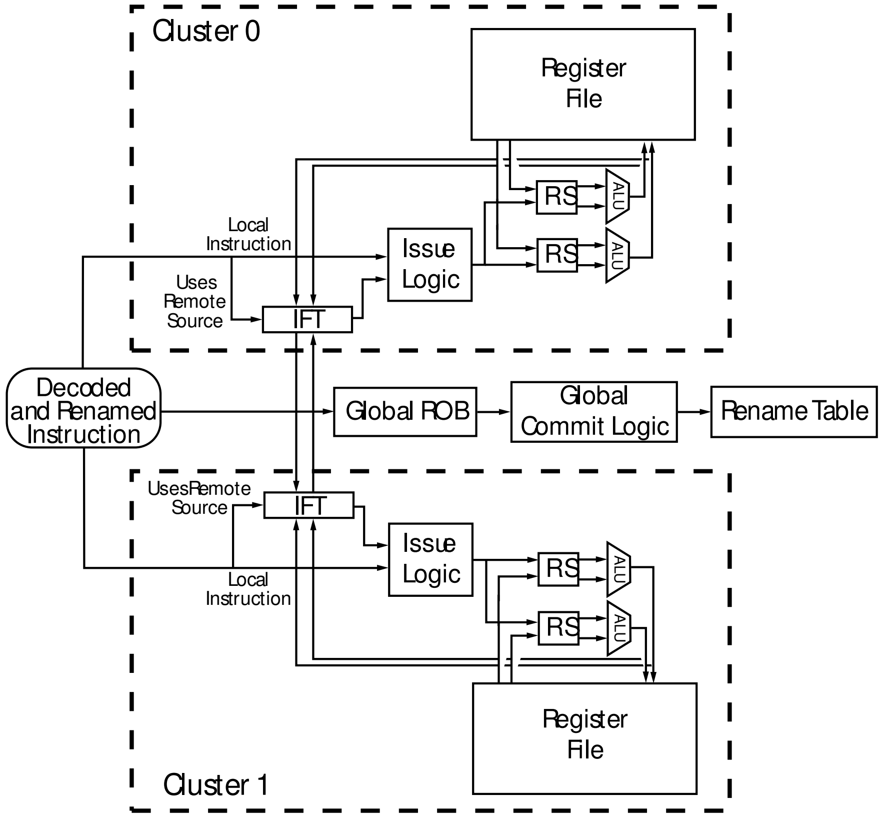

Figure 3: **Overview of the Architecture**

architecture would indeed be a performance win. Also, as the size of the register file grows, and the latency of wires increases, we expect our technique to perform even better.

In our project we are targeting mainly the integer register file. At the same time however there is absolutely nothing in our architecture that will prevent it from being applied to the floating point registers as well. In other words, we expect almost all applications to benefit from our scheme, but we show performance results for only SPEC integer benchmarks.

2 Architecture

Our design is based on a modified SimpleScalar architecture [2]. Unlike the standard SimpleScalar architecture, our design does not contain a register update unit (RUU). Our baseline architecture replaces the RUU with reservation stations, a limited reorder buffer, and a single register file. While functionally similar, the differences in the baseline are important for comparing parallel structures with our proposed architecture. The overall structure of our processor is shown in Figure 3. SimpleScalar parameters for the baseline architecture are shown in Table 2.

Our proposal is to partition elements of the architecture into clusters, splitting some of the resources and duplicating others. Figure 4 depicts the pipeline with detail as to which stages are divided. A single instruction stream is fetched and decoded as in a conventional architecture. The decode stage renames registers uniquely, but is slightly restricted. For a two cluster implementation (which will be assumed here for explanation purposes), register names in the lower half of the architected register file must be renamed in the lower half of the physical register file, and likewise for those in the upper half. An entry is then

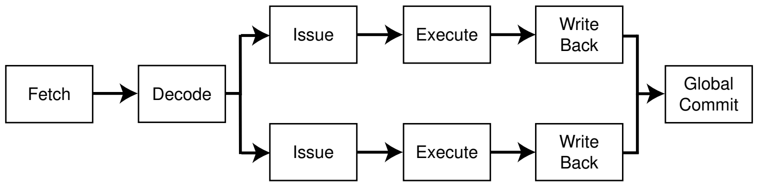

Figure 4: Logical Pipeline Structure of the Architecture

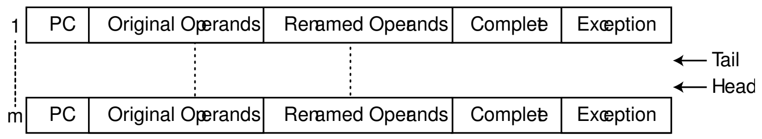

Figure 5: Reorder Buffer

allocated in a simplified reorder buffer. This buffer keeps the renamed and original register names and bits for tracking completion of execution and exceptions, as depicted in Figure 5. This buffer uses pointers for the head and the tail to insert entries in order and remove them in order when instructions commit. The decoded and renamed instructions are then sent to both clusters identically along with their corresponding address in the reorder buffer. This address must be kept with the instructions to properly match them with their entry again when they are ready to commit.

Issue logic in each cluster independently, but identically, determines the proper action for that instruction. If the destination (or only) register accessed by the instruction is in that cluster and no source registers are in another cluster, that instruction may be issued. Issued instructions go into reservation stations for the functional units in that cluster, reading values from the register file appropriately as in a conventional system. If the instruction requires one or two sources from another cluster, a special table entry is allocated to coordinate the data forwarding. This table, the inter-cluster forwarding table (IFT), is described in detail below.

Similarly, if a cluster contains only sources, but not the destination of the instruction, it must either forward the needed data or allocate a table entry in the IFT and forward the data when it is available. If the cluster contains none of the registers used by the instruction, the instruction is discarded. For special instructions that do not use registers (e.g., absolute jump), the first cluster will execute the instruction and all others will discard it.

Once instructions are issued to a reservation station, they read their operands from the register file and proceed through the execution units in that cluster as available. Execution units are fully duplicated in the two clusters, as per the motivating principle that this logic is small and easily duplicated. Once an instruction completes, it sets a bit in the reorder buffer to indicate that it has completed. At this point, the cluster's work is complete. Commit logic reads instructions from the tail of the reorder buffer and commits them in order by changing the mapping in the rename logic to indicate the location of the new value of the result register. Exceptions are detected and handled as in any processor; in-flight instructions are flushed, and the rename logic maintains a map of the correct location of all register values.

Gains in register file latency are due to a decrease in the number of registers in each file. In order to achieve this gain, it is important that design changes do not require additional ports, which contribute to access latency more than additional. Because instructions are sent to all clusters, issue logic will consider instructions that other clusters are to execute. Issue logic must decide whether to issue an instruction to one of its reservation stations, send one or two source operands to another cluster, or discard the instruction. Register ports are available for the full issue width, as in the baseline architecture. If operands need to be forwarded for an instruction, the read ports that could have been used to issue that instruction are instead used to read the operands for forwarding. The other situation where inter-cluster communication is needed

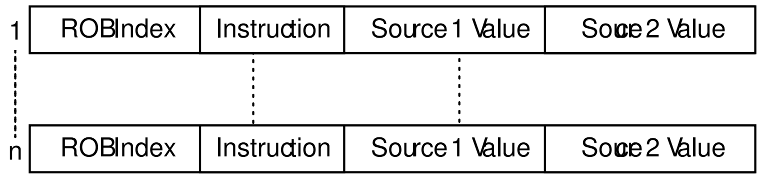

Figure 6: Inter-cluster Forwarding Table

occurs when data needs to be forwarded from the result of a calculation to a waiting instruction in another cluster. In this case, data is directly forwarded from the output of the ALU as controlled by a special table.

This table for forwarding operands between clusters is an additional hardware structure required for this partitioning scheme. When a needed operand is ready at the time an instruction is sent to the issue stage, it is immediately sent from the source cluster. After the inter-cluster communication delay, the correct cluster receives the operand and can issue the instruction to a reservation station. When the needed operand is the result of an instruction that has not completed, an entry is allocated on both sides in an inter-cluster forwarding table (IFT). This table contains entries as depicted in Figure 6. Each entry contains the entire instruction, including the name of the register(s) to be forwarded for a specific instruction; two register name locations are needed, as it is possible to have to forward both source registers. Each table entry also contains the index of the reorder buffer where the corresponding instruction is located. When the operand is available, the value and the name of the register are sent to the destination cluster. When the operand (or both operands, if two needed) has been sent, the entry is deallocated and available for another instruction to use. Likewise, when the last operand has been received by the cluster executing the instruction, the entry on that side is deallocated, and the instruction consuming the operands may be issued. This occurs a number of cycles after the operand has been sent equal to the inter-cluster communication delay.

Because only instructions that are not local to a single cluster and require a source not yet calculated need an entry in the IFT, IFT usage will be quite low, especially in a two cluster implementation. Because a realistic table size would be quite small, the control for the IFT should be simple, and the table should be fast enough not to impact the cycle time. In an actual implementation, the processor would stall if space were not available for an IFT allocation; in our simulation, this rare case is not modeled.

Extending this design to a larger number of clusters should be possible with little modification. IFT interaction will complicate because communication must be directed to the correct path. Each cluster must have a dedicated connection in each direction to every other cluster because they cannot arbitrate usage of a bus in a single cycle. Divisions between the lower and upper halves of the register file extend to quarters or eighths easily. Extending beyond this point is probably not reasonable without changing other major elements of the architecture; dividing an architecture with 32 architected registers even eight ways eliminates most locality with only four registers in each cluster.

3 Experimental Methodology

3.1 SimpleScalar Simulator

A modified SimpleScalar version 2.0 models both the baseline and the proposed architectures described in Section 2. This section briefly describes operations of both the original and modified versions of SimpleScalar.

As mentioned earlier, SimpleScalar uses the RUU in order to handle out-of-order issue and in-order commit. During the dispatch stage, SimpleScalar allocates one RUU entry per instruction. Then, it decodes instructions, updates the register rename table, and executes instructions. During the issue stage, SimpleScalar consults the RUU to determine ready instructions, allocates hardware resources such as ALU, and schedules an event to occur when instructions are supposed to complete execution. By executing instructions in the dispatch stage and modeling functional unit latencies through events, SimpleScalar implicitly models the execution stage. During the writeback stage, SimpleScalar checks completed instructions and modifies the RUU in order to wake up dependent instructions. During the commit stage, it retires instructions in order through the RUU and updates the register rename table to reflect that results are stored in the architected register file. The commit stage also flushes pipeline upon branch mispredictions.

The baseline architecture described in Section 2 differs from the SimpleScalar architecture in two significant ways. First, the baseline architecture has one physical register file that stores values produced by both committed instructions and instructions in flight. Second, it has the simplified reorder buffer that stores neither source nor result values. Hence, the number of physical registers can be arbitrarily greater than the architected registers. Since the purpose of the simplified reorder buffer is to maintain instruction dependencies and ensure in-order commit of instructions, the modified SimpleScalar reuses the RUU in the original SimpleScalar in order to model the reorder buffer. To model one physical register file, the modified SimpleScalar has a register rename table, separate from the rename table associated with the RUU that is only used to handle instruction dependencies. During the dispatch stage, the modified SimpleScalar explicitly renames registers through this new rename table and stalls pipeline if there is no free register. During the issue stage, register access time is simulated by delaying execution completion events by register access time. So, the modified SimpleScalar can model arbitrary register access times. During the commit stage, the modified SimpleScalar updates the rename table to reflect the most recent locations of the architected registers in the physical register file.

To model the proposed architecture with multiple clusters described in Section 2, SimpleScalar is further modified as follows. The dispatch stage now renames registers such that logical register names that map to the same cluster are assigned physical registers that also belong to the same cluster. Also, the dispatch stage determines which cluster is going to execute each instruction. In the proposed architecture, each cluster independently determines whether to execute a given instruction. This logic is moved to the dispatch stage in SimpleScalar only for the ease of programming and does not alter the architecture. The issue stage delays instructions that require operands from remote clusters to model inter-cluster communication delays. Hardware resources per cluster are modeled by allocating data structures specific to cluster during the initialization of the simulator. Finally, the commit stage postpones retiring instructions to model a multiple-cycle commit stage.

3.2 Benchmarks

Three applications from the SPEC CPU2000 [4] integer benchmark suite are used to evaluate the impact of partitioning register file on overall processor performance. The three applications are 175.vpr (FPGA circuit placement and routing), 181.mcf (minimum cost network flow solver), and 197.parser (natural language processing). For each input, 175.vpr executes twice. It first produces circuit placements without routing. Then, the same program reads the placement results, determines routing, and generates final circuit placements. 175.vpr exhibits very different characteristics during these two stages, and the program in each stage is considered a distinct application. For brevity, the subsequent sections use VPR, VPR2, MCF, and PARSER to refer to 175.vpr (the first stage), 175.vpr (the second stage), 181.mcf, and 197.parser benchmarks. All benchmarks use the reduced input data sets [5].

3.3 Processor Configurations

Table 1 shows the processor configurations used for evaluation. All configurations have 64-entry simplified reorder buffer and 128 physical registers. BASE configurations model the baseline architectures with one and two-cycle register access time. BASE(1) represents the ideal baseline architecture that pays no extra cycle for register access. These are used to measure performance impact of increasing register access time. BASE(C) is same as BASE(1) except the two-cycle commit stage. The purpose of BASE(C) is to assess the impact of multi-cyle commit stage on processor performance. PROPOSED configurations model the proposed architectures with two or four clusters with intercluster communication delays varying from one to four cycles. PROPOSED2(1) and PROPOSED4(1) configurations represent ideal versions of the proposed architecture with only one-cycle intercluster communication delay. These configurations are used to show

Configuration	Clusters	Register Latency	Intercluster Latency	Commit Latency
BASE(1)	1	1	N/A	1
BASE(2)	1	2	N/A	1
BASE(C)	1	1	N/A	2
PROPOSED2(1)	2	1	1	2
PROPOSED2(2)	2	1	2	2
PROPOSED2(3)	2	1	3	2
PROPOSED2(4)	2	1	4	2
PROPOSED4(1)	4	1	1	2
PROPOSED4(2)	4	1	2	2
PROPOSED4(3)	4	1	3	2
PROPOSED4(4)	4	1	4	2

Table 1: Processor configurations used for evaluation of both the baseline and the proposed architectures.

Parameter	Value	Parameter	Value
-fetch:ifqsize	4	-bpred	bimod
-fetch:mplat	3	-bpred:bimod	2048
-fetch:speed	1	-bpred:btb	512 4
-decode:width	4	-issue:width	4
-commit:width	4	-issue:inorder	false
		-issue:wrongpath	true
-ruu:size	64 (the number of entries in the simplified reorder buffer)	-lsq:size	8
-cache:dl1	dl1:128:32:4:l	-cache:dl1lat	1
-cache:dl2	ul2:1024:64:4:l	-cache:dl2lat	6
-cache:il1	il1:512:32:1:l	-cache:il1lat	1
-cache:il2	dl2	-cache:il2lat	6
-cache:flush	false	-cache:icompress	false
-mem:lat	18 2	-mem:width	8
-tlb:itlb	itlb:16:4096:4:l	-tlb:dtlb	dtlb:32:4096:4:l
-tlb:lat	30		
-res:ialu	4 (per cluster)	-res:imult	1 (per cluster)
-res:memport	2	-res:fpalu	4 (per cluster)
-res:fpmult	1 (per cluster)		
-bugcompat	false		

Table 2: Processor configuration parameters common to all configurations shown in Table 1

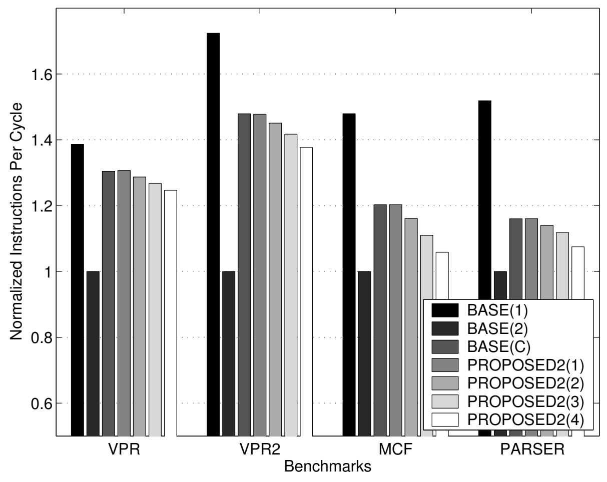

Figure 7: Instruction throughputs achieved by BASE(1), BASE(2), BASE(C), PROPOSED2(1), PROPOSED2(2), PROPOSED2(3), and PROPOSED2(4) configurations shown in Table 1. Instruction throughputs are normalized to that of BASE(2).

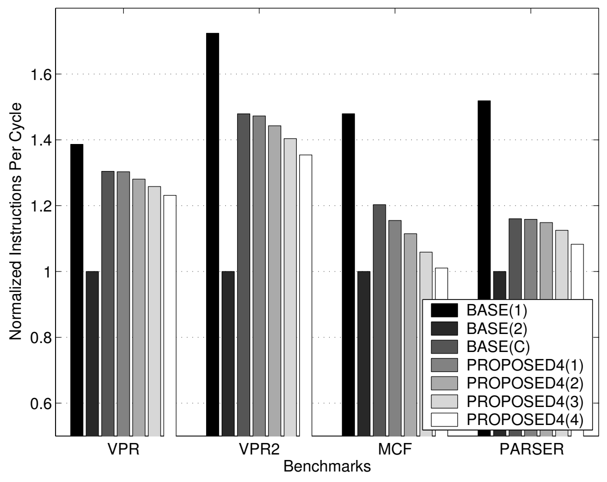

Figure 8: Instruction throughputs achieved by BASE(1), BASE(2), BASE(C), PROPOSED4(1), PROPOSED4(2), PROPOSED4(3), and PROPOSED4(4) configurations shown in Table 1. Instruction throughputs are normalized to that of BASE(2).

the upper bound of performance of the proposed architecture. PROPOSED configurations have two-cycle commit stage to account for increased access times of the global structures that must be updated during the commit stage.

Table 2 shows SimpleScalar parameters common to all configurations shown in Table 1. The parameters have default values of SimpleScalar version 2.0. For descriptions of each parameter, refer to the SimpleScalar manual [1].

4 Experimental Results

The experimental results show that reducing register access time through partitioning can improve instruction throughput measured in instructions per cycle. As predicted by Figures 1 and 2 in Section 1.1, performance penalty due to intercluster communication does not offset benefits from reduced register access time.

Figure 7 shows the instruction throughputs achieved by the baseline architecture and the proposed architecture with two clusters. The X axis shows the names of benchmarks (refer to Section 3.2 for their descriptions). For each benchmark, seven bars show instruction throughputs achieved by BASE(1), BASE(2), BASE(C), PROPOSED2(1), PROPOSED2(2), PROPOSED2(3), and PROPOSED2(4) configurations. The Y axis shows instructions per cycle normalized to that of BASE(2). As described in the previous section, BASE(1) represents the ideal baseline architecture with one-cycle register access time. Instruction throughputs of BASE(1) are about 38–72% higher than those of BASE(2), showing that increased register access time can significantly degrade instruction throughput. All of PROPOSED2 configurations outperform BASE(2). PROPOSED2(2) achieves 15–45% improvement over BASE(2). PROPOSED2(3) and PROPOSED2(4) show that instruction throughput decreases as intercluster communication delay increases. This result is expected since the number of cycles spent waiting for remote register accesses increases proportional to intercluster communication delay. Table 3 shows actual instruction throughputs and the ratio of remote and total register accesses. Remote register accesses acount for only 9–15% of all register accesses, as predicted by Figures 1 and 2. Instruction throughput of PROPOSED2(2) is within 7–25% of BASE(1)'s instruction throughput.

The upper bound of instruction throughput of the proposed architecture, shown by PROPOSED2(1), is only about 5–14% higher than that of PROPOSED2(4). This result shows that the proposed architecture can effectively tolerate intercluster communication delays. However, the instruction throughput of PROPOSED2(1) is up to 24% worse than BASE(1), while BASE(C)'s instruction throughput is almost identical to that of PROPOSED2(1). When a mispredicted branch is detected, both baseline and proposed architectures update branch prediction table during the commit stage. The two-cycle commit stage adversely affects the branch prediction accuracy, and the prediction accuracy of jump register instruction decreases by roughly 5%.

The proposed architecture with four clusters also achieves higher instruction throughput than the baseline architecture. Figure 8 shows the instruction throughputs achieved by BASE(1), BASE(2), BASE(C), PROPOSED4(2), PROPOSED4(3), and PROPOSED4(4) configurations. The figure is in the same format as Figure 7. PROPOSED4(2) improves instruction throughput by 10–41% over BASE(2). These improvements are lower than the improvements achieved by PROPOSED2(2) due to the increased remote register accesses. Table 3 shows that with four clusters, remote register accesses acount for 14–17% of all register accesses, as opposed to 9–15% with two clusters. Despite the increased performance penalty due to remote register accesses, all of the PROPOSED4 configurations still outperform BASE(2). PROPOSED4(1) again achieves only about 6–14% higher instruction throughput than PROPOSED4(4), showing that the proposed architecture can tolerate various intercluster communication delays. The proposed architecture assumes two-cycle commit stage to account for the need to update global structures during the stage. As mentioned earlier, the two-cycle commit stage and its adverse impact on branch prediction accuracy, rather than multi-cycle intercluster communication delays, account for most of the performance penalty of the proposed architecture. Earlier detection of branch misprediction and updates of branch prediction table would bring the performance of the proposed architecture close to the ideal baseline architecture.

5 Conclusion

We have investigated a novel technique to alleviate increasing register file access latencies. We hypothesize that splitting up the register file and duplicating execution resources can gain us a performance win over architectures which employ multi-cycle register files.

Our results show that our hypothesis is indeed correct. For the benchmarks we simulated, we were able to get a performance improvement ranging from 14-45% with two clusters, depending upon the intercluster communication delay that we attribute to our architecture. We have simulated our proposed architecture with intercluster communication delays of two, three and four cycles. Our results show that even with a 4-way split of the register file, our technique still improves processor performance over the multi-cycle register file. In this case, however, the improvement is less as compared to the two cluster case. This is reasonable since with four clusters, the locality obtained in instructions is reduced and hence there is more inter-cluster communication penalty.

As architects try to come up with techniques to counter wire delay, a clustering approach has been proposed by many. Our architecture exposes a different approach to clustered architectures. As the register files grow in size, and wire delay becomes more of a problem, we believe the merits of an architecture such as ours will become even more apparent.

References

[1] *The SimpleScalar Tool Set, Version 2.0.*

[2] Doug Burger, Todd M. Austin, and Steve Bennett. Evaluating Future Microprocessors: The Simplescalar Tool Set. Technical Report CS-TR-1996-1308, 1996.

Configuration	Benchmark	IPC	Remote Register Access Rate (Remote//Total)
BASE(1)	VPR	0.9466	0 (0//90575471)
	VPR2	1.4554	0 (0//24123950)
	MCF	1.3400	0 (0//325525986)
	PARSER	1.6390	0 (0//568325969)
BASE(2)	VPR	0.6827	0 (0//94545395)
	VPR2	0.8442	0 (0//25211340)
	MCF	0.9059	0 (0//349821612)
	PARSER	1.0793	0 (0//641305913)
BASE(C)	VPR	0.8904	0 (0//88792692)
	VPR2	1.2485	0 (0//23585370)
	MCF	1.0899	0 (0//319574116)
	PARSER	1.2522	0 (0//535657435)
PROPOSED2(1)	VPR	0.8923	0.14 (12468100//87923838)
	VPR2	1.2476	0.15 (3587788//23480772)
	MCF	1.0899	0.11 (35517931//313551131)
	PARSER	1.2526	0.09 (47384014//530006631)
PROPOSED2(2)	VPR	0.8786	0.14 (12521509//88318934)
	VPR2	1.2246	0.15 (3615363//23704141)
	MCF	1.0520	0.11 (35915806//318225825)
	PARSER	1.2304	0.09 (47987068//540818945)
PROPOSED2(3)	VPR	0.8654	0.14 (88935764//12617553)
	VPR2	1.1965	0.15 (23810856//3626989)
	MCF	1.0055	0.11 (320351217//36094339)
	PARSER	1.2068	0.09 (552018823//49703883)
PROPOSED2(4)	VPR	0.8512	0.14 (12710394//89635310)
	VPR2	1.1621	0.15 (3638990//23897217)
	MCF	0.9588	0.11 (36457290//322664175)
	PARSER	1.1604	0.09 (49862427//559294329)
PROPOSED4(1)	VPR	0.8895	0.16 (13439282//84423331)
	VPR2	1.2431	0.17 (3920065//22852901)
	MCF	1.0465	0.14 (42627085//293668103)
	PARSER	1.2503	0.14 (75962979//519208473)
PROPOSED4(2)	VPR	0.8743	0.16 (13473394//84608032)
	VPR2	1.2180	0.17 (3929889//22909658)
	MCF	1.0100	0.14 (42603782//294364094)
	PARSER	1.2397	0.14 (74039906//518511083)
PROPOSED4(3)	VPR	0.8590	0.16 (13520269//84865801)
	VPR2	1.1850	0.17 (3936381//22962632)
	MCF	0.9592	0.14 (42454506//294498720)
	PARSER	1.2146	0.14 (76690363//529976577)
PROPOSED4(4)	VPR	0.8407	0.16 (13581940//85209955)
	VPR2	1.1432	0.17 (3942378//23008245)
	MCF	0.9153	0.14 (42485568//295701561)
	PARSER	1.1685	0.14 (77912382//537429157)

Table 3: Processor performance achieved by the configurations shown in Table 1

[3] Keith I. Farkas, Norman Jouppi, and Paul Chow. Register File Design Considerations in Dynamically Scheduled Processors. In *Proceedings of the Second IEEE Symposium on High-Perfomance Computer Architecture*, February 1996.

[4] John L. Henning. SPEC CPU2000: Measuring CPU Performance in the New Millennium. *Computer*, 33(7):28–35, July 2000.

[5] AJ KleinOsowski and David J. Lilja. MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Ccomputer Architecture Research. *Computer Architecture Letters*, 1, June 2002.

[6] Subbarao Palacharla, Norman P. Jouppi, and J. E. Smith. Complexity-Effective Superscalar Processors. In *Proceedings of the 24th International Symposium on Computer Architecutre*, pages 206–218, 1997.

[7] Scott Rixner, William J. Dally, Brucek Khailany, Peter Mattson, Ujval J. Kapasi, and John D. Owens. Register organization for media processing. In *Proceedings of the 6th International Symposium on High Performance Computer Architecture*, pages 375–386, January 2000.