Journal of Instruction-Level Parallelism 8 (2006) 1-17 Submitted 10/05; published 2/06

Asymmetric Clustering using a Register Cache

Roger Morrison MORRISON@EECS.ORST.EDU
Ben Lee BENL@EECS.ORST.EDU
School of Electrical Engineering and Computer Science
Oregon State University
1148 Kelley Engineering Center
Corvallis, OR 97331-5501 USA

Shih-Lien Lu SHIH-LIEN.L.LU@INTEL.COM
Microarchitecture Research Lab
Intel Corp.
2111 N.E. 25th Avenue,
Hillsboro, OR 97124 USA

Abstract

Conventional register files spread porting resources uniformly across all registers. This paper proposes a method called *Asymmetric Clustering using a Register Cache (ACRC)*. ACRC utilizes a fast register cache that concentrates valuable register file ports to the most active registers thereby reducing the total register file area and power consumption. A cluster of functional units and a highly ported register cache execute the majority of instructions, while a second cluster with a full register file having fewer read ports processes instructions with source registers not found in the register cache. An 'in-cache' marking system tracks the contents of the register cache and routes instructions to the correct cluster. This system utilizes logic similar to the 'ready' bit system found in wake-up and select logic keeping the additional logic required to a minimum. When using a 256-entry register file, this design reduces the total register file area by an estimated 65% while exhibiting similar IPC performance compared to a non-clustered 8-way processor. As the feature size becomes smaller and processor clocks become faster, the number of clock cycles needed to access the register file will increase. Therefore, the smaller register file area requirement and subsequent smaller register file delay of ACRC will lead to better IPC performance than conventional processors.

1. Introduction

High performance processors simultaneously issue and execute multiple instructions to exploit instruction level parallelism (ILP). This requires the register file to be able to provide register access to several instructions at once. For example, a processor with an issue-width of four is capable of executing four instructions per clock cycle and requires a register file with eight read ports and four write ports. A conventional processor contains a single, multi-ported register file that provides data to all the instructions. This allows an instruction to access any register value so the issue logic can assign instructions to the proper functional unit (FU) without any consideration of register file limitations. However, as the issue width increases to take advantage of ILP, so do the number of read and write ports the register file must contain. This causes a dramatic increase in physical size of the register file because the area of the register file increases quadratically with the issue width [1]. Moreover, processors require register files with a large number of entries to maximize performance, which also increases the register file area [1].

Large register file area leads to long wires to access data, which translates into slower access time and lower overall processor performance [2]. This has led to several design ideas to improve performance by reducing the size and access time of register files. One method is *register file caching*, which provides fast access to a small subset of registers. If an instruction requires a register not available in the register cache, then it accesses the full set of registers. This design necessitates additional logic to manage the register cache and guarantee instructions have access to needed registers. Another method to reduce the register file size divides the register file and FUs into separate *clusters*, which reduces the porting requirement and the size of each register file. For example, the Alpha 21264 processor has two similar clusters, each with a full copy of the register file and FUs write to both clusters simultaneously [3]. In addition, each register file only supplies four read ports allowing it to be smaller and faster than one unified register file with eight read ports. An alternative design uses clusters of different size and function that leads to *asymmetric clustering* [4]. The characteristics of the clusters are complementary and create a more powerful system. However, a clustered design reduces the processor's instructions per cycle (IPC) performance by limiting instructions to a subset of the processor resources.

This paper proposes a hybrid method called *Asymmetric Clustering using a Register Cache* (*ACRC*) that combines register file caching and asymmetric clustering techniques to concentrate porting resources to the most active registers. ACRC separates the register file and FUs into two clusters. To reduce the total register file size, one of the clusters contains only a small subset of the full register set. Since the register cache satisfies the majority of register read requests, it has a high number of read ports. This reduces the porting requirements and corresponding size of the full register file, which contains all register values, located in the remaining cluster. The resulting two register files are smaller and faster than a single, large register file, which leads to reduced power consumption and better overall processor performance. The design requires logic to determine if instructions will be able to use the cluster with the register cache or need to execute in the cluster with the full register set. Fortunately, this additional logic is minimal and is similar to that already found in many superscalar processors.

The organization of the paper is as follows. Section 2 presents the proposed ACRC design and its comparisons to conventional superscalar processor. Section 3 discusses how the proposed design impacts the register file area and processor performance. Section 4 presents the related work. Finally, Section 5 concludes the paper and discusses possible future work.

2. The Proposed Method

The proposed ACRC design shown in Figure 1 splits the issue window, register file, and FUs into two specialized clusters: Cache Cluster and Full Cluster. Since the *Cache Cluster* contains only a partial register file, instructions that require register values not found in the register cache must execute in the *Full Cluster* with the full register file. By default, the rename stage steers instructions into the Cache Cluster. When the rename stage recognizes that the register cache will not contain the needed register values for some of the instructions, these instructions are steered to the Full Cluster.

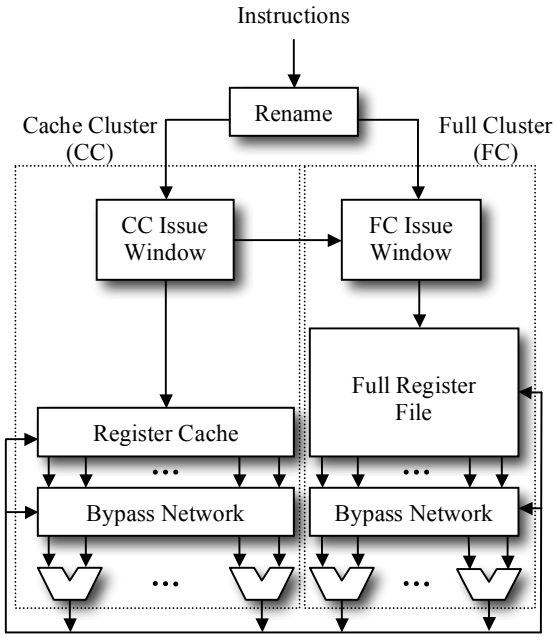


Figure 1: The proposed asymmetric clustering design.

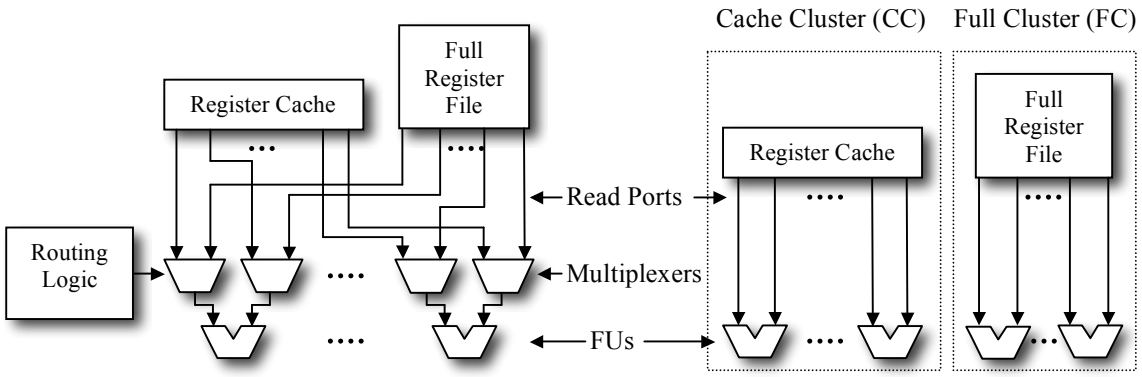
The issue window in each cluster contains logic similar to the issue window of a conventional superscalar processor. In addition, the issue window in the Cache Cluster has the ability to forward instructions to the issue window of the Full Cluster. This allows instructions to change clusters as the contents of the register cache change. The connections among the register file, bypass network, and FUs are all comparable to the conventional superscalar processor. That is, the register files connect to the FUs and the bypass network passes the outputs from FUs straight to the inputs of other FUs.

The following two subsections discuss the different clustering choices and the instruction steering mechanism required to route instructions to the clusters.

2.1. Clustering

The register cache supplies register values to the majority of instructions. However, since the register cache does not contain all of the registers, there are instructions that must read from the full register file. One approach to accomplish this is to allow the FUs access to both the full register file and the register cache by way of multiplexers and routing logic, as shown in Figure 2a. Since the full register file contains a limited number of read ports, a multiplexed design has to consider read port limitations when choosing instructions to issue. However, the issue window is a complex stage in a wide-issue processor's pipeline and integrating this extra logic into the issue window is impractical [5]. An alternative technique is to add a stage after the issue logic to recognize when the selected instructions exceed the number of available read ports in the full register file. This arbitration stage squashes and then reissues instructions when they exceed the number of full register file read ports. This involves an additional pipeline stage and wastes processor resources when squashing instructions, thus lowering performance [6].

For these reasons, the proposed ACRC design employs clustering shown in Figure 2b that ties FUs and register ports to a specific issue window. The first cluster contains the register



(a) Multiplexed register files *(b) Clustered register files*

Figure 2: Two clustering options.

cache and executes the majority of instructions requiring the register cache to be highly ported. The second cluster contains the full register file with the ability to process any instruction. This cluster executes only a small number of instructions allowing the full register file to have fewer read ports. FUs in the Cache Cluster will be idle if no instructions in the Cache Cluster issue window are ready, even if the issue window in the Full Cluster is full of ready instructions. This limitation reduces processor IPC slightly, but simplifies the design of the issue logic.

Clustering also requires consideration of instruction type such as simple integer, complex integer or load/store. The cluster with the full register file must be able to execute any type of instruction; otherwise, an instruction that cannot execute in the Cache Cluster would never execute and lead to a deadlock. This mandates that at least one of each type of FU must be in the Full Cluster. Two options are available for designing the Cache Cluster. The first option is to limit the types of instructions that can execute in the Cache Cluster. For this option to function properly, an additional routing system is needed to send any instructions that cannot execute in the register cache directly to the full register file. The second option is to ensure the Cache Cluster also has at least one of each type of FU. This increases the minimum number of required FUs, but eliminates the need for any additional instruction routing based on instruction type. The proposed design uses the second option and supports all instruction types in the Cache Cluster.

2.2. Instruction Steering Logic

Steering instructions to the correct cluster is a crucial part of the proposed system, and the mechanism must be simple and fast to avoid additional pipeline stages. The basic idea is to maximally exploit the register cache. Therefore, instructions are steered to the Cache Cluster instruction window whenever possible. However, since the register cache does not contain sufficient number of entries to hold all registers, entries will be replaced as new register values are produced. An instruction cannot execute in the Cache Cluster if either of its source registers have been replaced in the register cache. For this reason, the rename stage sends these instructions to the Full Cluster issue window. Likewise, if an instruction's source register is replaced in the register cache while the instruction is waiting in the Cache Cluster issue window, then the instruction is steered to the Full Cluster issue window.

In the proposed design, instructions with both source registers in the register cache must execute in the Cache Cluster. However, since the Full Cluster contains a full register file, these instructions could also execute in the Full Cluster. This can be implemented in several ways.

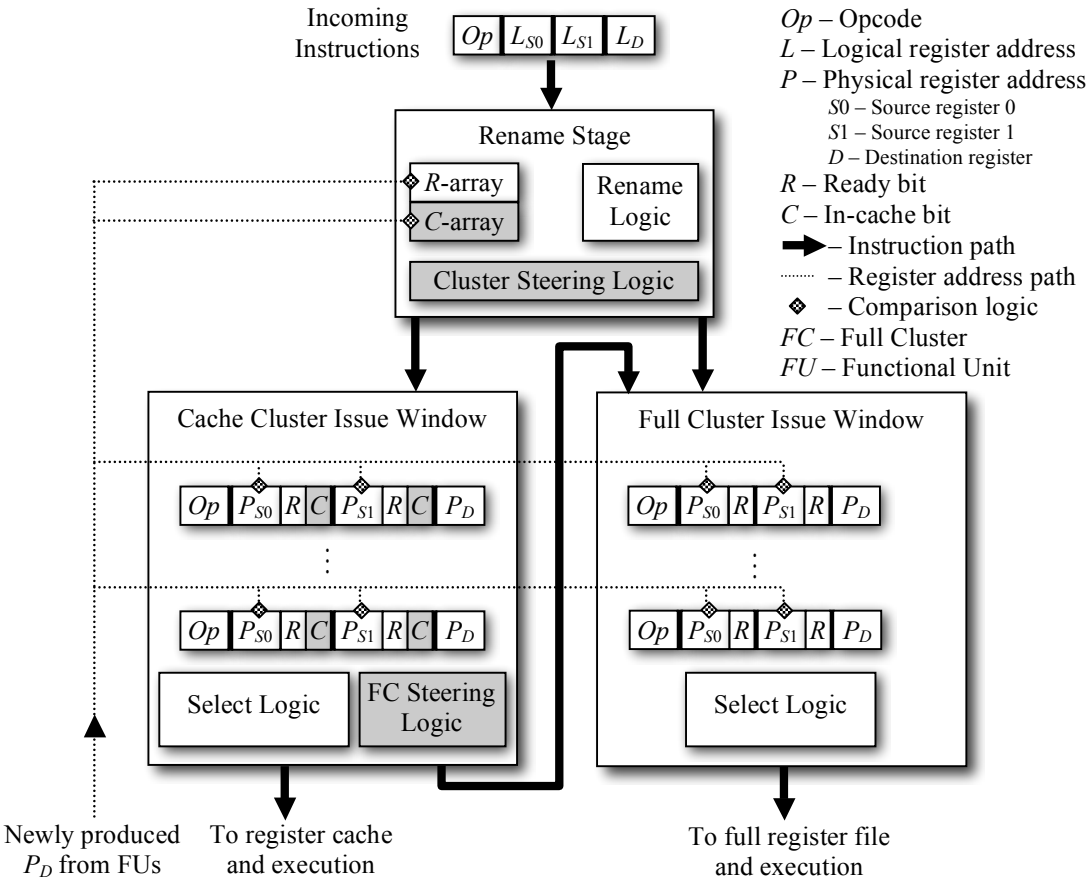


Figure 3: The rename and issue stages. New logic required is shown in grey.

One approach is to allow the Full Cluster issue logic to issue instructions from the Cache Cluster issue window. A second approach is to place each incoming instruction in the issue queue of both clusters. Instructions that cannot be issued in the Cache Cluster are deleted from the Cache Cluster issue queue, but remain in the Full Cluster issue queue. However, in both methods, the select logic for the two clusters must coordinate so that an instruction is not executed twice: once in each cluster. Also, the Full Cluster issue logic must scan more instructions, which increases complexity. For simplicity, this proposal limits the Full Cluster issue logic to only instructions in the Full Cluster issue window.

The steering logic in the rename stage and the Cache Cluster issue window must stay cognizant of the register cache's contents. In order to track whether or not register values are in the register cache, special 'in cache' bits are employed. These 'in-cache' bits work together with 'ready' bits already used in conventional superscalar processor. 'Ready' bits indicate when a register value has valid data in the register file. When both source registers of an instruction have valid data, the instruction is ready to execute. Maintaining these 'ready' bits requires two systems, one in the rename stage and the other in the issue window. Therefore, in addition to the 'ready' bits, each system also maintains 'in-cache' bit for each source register.

The modifications necessary for the rename and issue stages are shown in Figure 3. The rename stage contains an array of 'ready' (*R*-array) and 'in-cache' (*C*-array) bits, one for each physical register. When the source registers of incoming instructions are renamed, their 'ready'

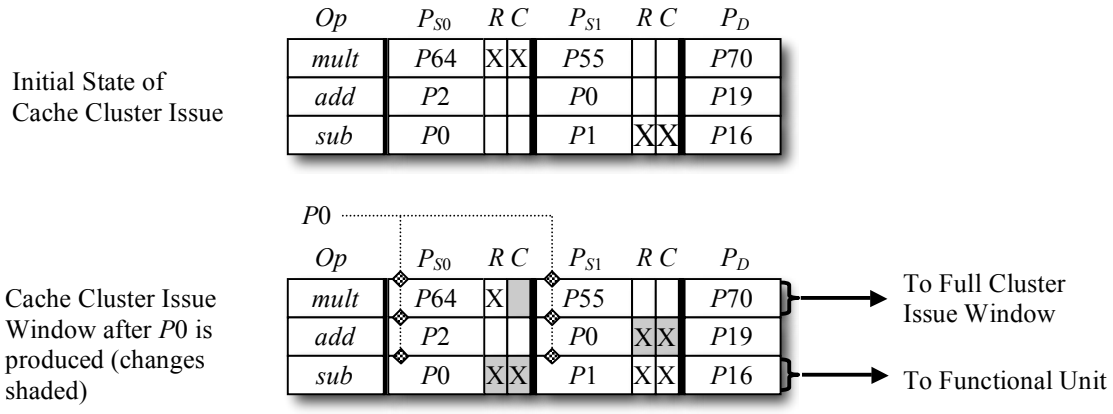


Figure 4: An example of issue process in the issue window of the Cache Cluster using a 64-entry register cache. In this case, a functional unit produces a new value for register P0. First, wake-up logic sets and clears the appropriate ready and in-cache bits. Then, the select logic chooses entries to send to the functional units and Full Cluster issue window.

bit and 'in-cache' bits are also assigned their initial bit values. The *C*-array in the rename stage and the wake-up logic in the issue window keep these bits up to date while the instructions are waiting for execution.

After an instruction executes, the register result value is written to both the register cache and the full register file. The rename logic and issue windows monitor this activity and set the register's corresponding 'ready' and 'in-cache' bits. Thus, when the 'ready' bit is set, it indicates the corresponding register contains a valid data. This is also true for the 'in-cache' bit, which indicates the corresponding register value is in the register cache. However, since a newer register replaces an existing register in the register cache, the replaced register's 'in-cache' bit in the *C*-array and the Cache Cluster issue window is cleared. Therefore, the replaced register is identified by a set 'ready' bit and cleared 'in-cache' bit, and instructions with replaced source registers are rerouted to the Full Cluster issue window. An example of this is shown in Figure 4.

In this paper, the ACRC system utilizes a direct-mapped register cache. Other mapping policies, such as set-associative, and replacement algorithms may lead to a more efficient utilization of the register cache and thus improve performance. However, set-associative caches are more complex, which increases the amount of additional logic and delay required for the ACRC design. For simplicity, this paper does not examine other register cache mapping methods.

The direct-mapped register cache used in the proposed system separates the register address identifier into a tag and an index. The index of a register's address determines where to store the register value in the register cache. Consider, for example, the configuration with 256-entry full register and 64-entry register cache shown in Figure 5. Registers with the address of '000', '064', '128' and '192' all share the same index, *000000*, but have different tags: **00**, **01**, **10** and **11**. Therefore, the mapping policy stores these four registers in the same location in the register cache. When an instruction writes a result to the register '000', the previous value stored in that register cache location is replaced. In this case, it can be register '064', '128' or '192'.

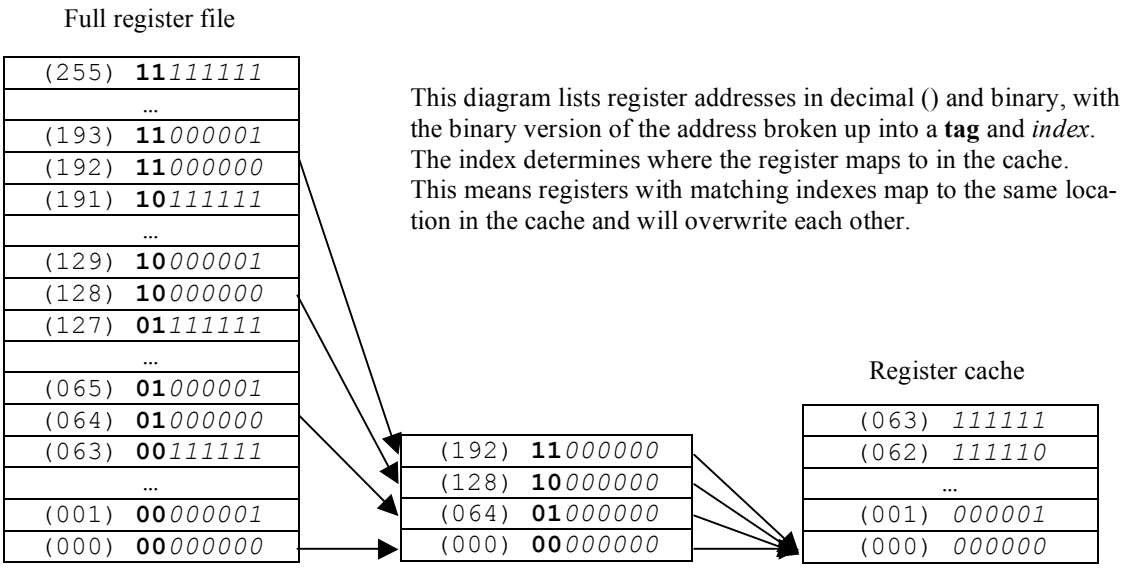


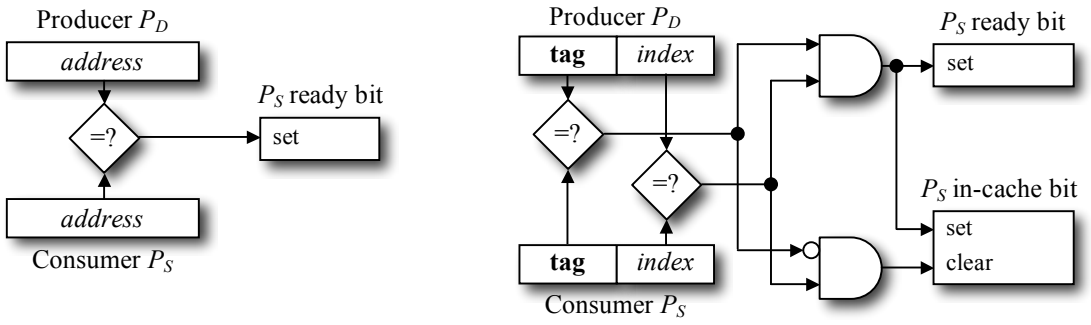
Figure 5: Direct mapping of registers to the register cache. This example uses a 256-entry full register file and 64-entry (1/4 size) register cache.

Both the rename stage and the Cache Cluster issue window track this behavior by clearing the 'in-cache' bits of instructions' source registers whenever a result value is generated for a register with a matching index, but non-matching tag. For example, when a result for the destination register **00***000000* is generated, the rename stage and the Cache Cluster issue window scans for any source register entries with index of *000000*. Of these entries, if its tag is **00**, then its 'ready' and 'in-cache' bit is set. Other entries with tag of **01**, **10** or **11** have their 'in-cache' bits cleared. The Full Cluster issue window does not require this additional 'in-cache' bit logic because its register file contains all of the registers. However, instructions write result values to the register files in both clusters, so each issue window and the rename stage must monitor values produced by both clusters.

Figure 6 shows a possible design for setting and clearing the 'ready' and 'in-cache' bits in the rename stage and Cache Cluster issue window. The processor compares the physical addresses of the destination register (P_D) and the corresponding source register (P_S). A match causes the 'ready' bit and the 'in-cache' bit to be set. In addition, the 'in-cache' bit is cleared when index matches but tag mismatches.

A special case occurs when one register result value sets the 'in-cache' bit and, simultaneously, another results register value causes the 'in-cache' bit to clear. For example, with a 64-entry register cache, both registers '000' and '128' could be produced during the same clock cycle. This causes a conflict in the register cache because two values will be written to the same location. A 'tie-breaker' method could be used to decide which value to store in the register cache; however, such a solution would require an appropriate logic to set the 'in-cache' bit. For simplicity, this proposal assumes neither register value is written to the register cache in the case of a tie. The 'in-cache' bit system tracks this by clearing the bit when both the clear and set input signals are enabled.

The 'in-cache' bit has an additional timing requirement not needed for a 'ready' bit. When the issue logic sets the 'ready' bit it remains set until the instruction executes. This is because registers in a full register file are not overwritten until all dependant instructions are done exe-



Conventional ready bit logic

Logic with in-cache bit

Figure 6: An example of logic to set and clear the ready and in-cache bits.

cuting. In the register cache, however, a register can be overwritten while dependant instructions are waiting in the Cache Cluster issue window. The design must guarantee that the select logic does not incorrectly issue instructions based on expired 'in-cache' bit information. For example, suppose an instruction's source registers are both marked as 'in-cache' and, during the same clock period, another instruction executes and overwrites a needed source register value in the register cache. The select logic will send the instruction to a FU for execution, but the register cache will no longer contain the source register. Checking for this error is possible, but may not be simple or fast.

Fortunately, register files of modern superscalar processors require multiple cycles to perform a write followed by a read. That is, the register file cannot have a value written into a register and read back out during the same clock cycle. Instead, buffers store values during reads and writes. This allows an instruction to read an old register value while another instruction simultaneously writes a new value to the same register cache location. So long as reading a value from the register file takes longer updating the 'in-cache bit', the system will work correctly.

3. Performance

The goal of ACRC is to increase performance by reducing register file area and access delay. To obtain performance, the design must offer the processor a faster clock rate and/or increase the processor's instructions per clock (IPC). Since the processor's clock rate must be slow enough to allow each stage in the pipeline to complete, fast pipeline stages are critical. Slow pipeline stages, like register file access, can be split into multiple stages. This allows the clock rate to remain fast, but can increase the branch misprediction penalty. When a processor incorrectly predicts the outcome of a branch instruction, all of the instructions that follow the branch are wasted. The number of wasted instructions depends on the number of pipeline stages between the branch prediction and the actual branch execution. Thus, a large branch misprediction penalty reduces overall performance by lowering IPC.

The ACRC design reduces the size of the register file and splits the issue window and bypassing network stages into smaller parts. This may increase the processor clock rate by allowing these stages to be faster. However, this is only true if these stages are the slowest in the pipeline and limit the processor clock rate. Unfortunately, accurately predicting the delay of

Table 1: ALU and register file port requirements for the three simulated cluster configurations.

Configuration	Simple Integer ALU	Complex Integer ALU	Load/Store Unit	Read Ports	Write Ports	Total Ports
8-way conventional	8	4	4	16	8	24
7-way Cache Cluster	7	3	3	14	8	22
1-way Full Cluster	1	1	1	2	8	10
6-way Cache Cluster	6	3	3	12	8	20
2-way Full Cluster	2	1	1	4	8	12
4-way Cache Cluster	4	2	2	8	8	16
4-way Full Cluster	4	2	2	8	8	16

each stage is difficult without designing an entire processor. For this reason, this paper does not investigate the clock speed advantages that may be possible through the ACRC design.

The main advantage of the smaller register file is the ability to read register data quickly. Processors such as the Pentium® 4 require multiple clock cycles to access the register file [7] and these cycles increase the branch misprediction penalty. A faster register file reduces the number of pipeline stages between the branch prediction and actual branch execution, which increases IPC. Therefore, Subsection 3.1 presents the total register area, power, and access delay estimates for different register file configurations using previously derived formulas. Using these register file access delay estimates, Subsection 3.2 evaluates the effects of ACRC and its reduction of register access delays on processor IPC using micro-architectural simulation.

3.1. Register Cache Area, Power, and Delay Estimation

The area and power of a register file grows approximately linearly with the number of entries and quadratically with the number of ports [1]. Therefore, the total register file area and power for the ACRC design is proportional to $R_{RF}\,p_{RF}^2 + R_{RC}\,p_{RC}^2$, where R represents the number of registers, p represents the number of ports, and subscripts RF and RC indicate full register file and register cache, respectively. The three ACRC configurations shown in Table 1 split the 8-way execution capability and FUs between the two clusters. Each of the listed configurations requires register files with a different number of ports and entries. For example, a 6-way Cache Cluster requires a register cache with 12 read ports to supply values for up to 6 simultaneous instructions. A register file located in the Full Cluster will have 256 entries, while the number of entries in the Cache Cluster varies. Both register files have 8 write ports to store the results from both clusters.

Figure 7 shows the area and power estimates of different register cache configurations according to Table 1. The baseline register file consuming 100% of the area and power is a 16-read, 8-write, 256-entry register file. The groups of bars on the graph represent the three different methods of breaking the 8-way execution capabilities into separate clusters and the individual bars represent various register cache sizes.

As the feature size of processors continue to shrink, the wire interconnects become a greater source of delay [8]. Larger register files must have longer wires, increasing register file access time. This wire length becomes the dominant delay for register files large enough to serve a 10-way processor when using a 130 nm manufacturing process [1]. This paper assumes this trend will continue down through to 8-way processors for future manufacturing processes.

A formula from [1] estimates register file access delay, which is proportional to the square root of the register file area. In this proposal, estimation for register file access delay incorpo-

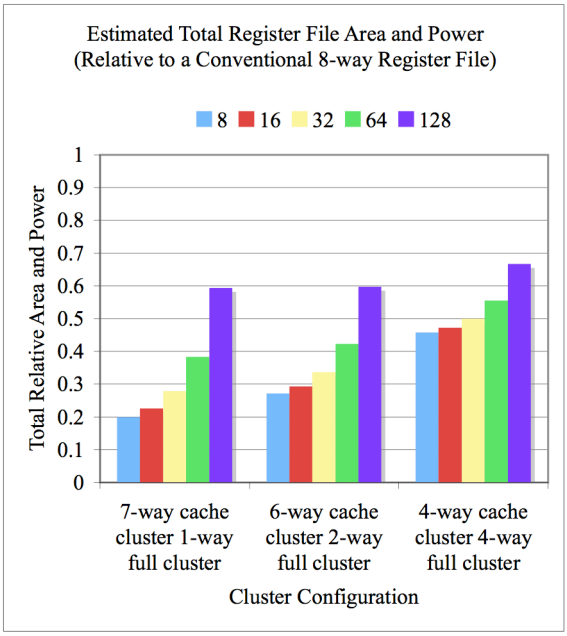


Figure 7: Estimated total register file area and power.

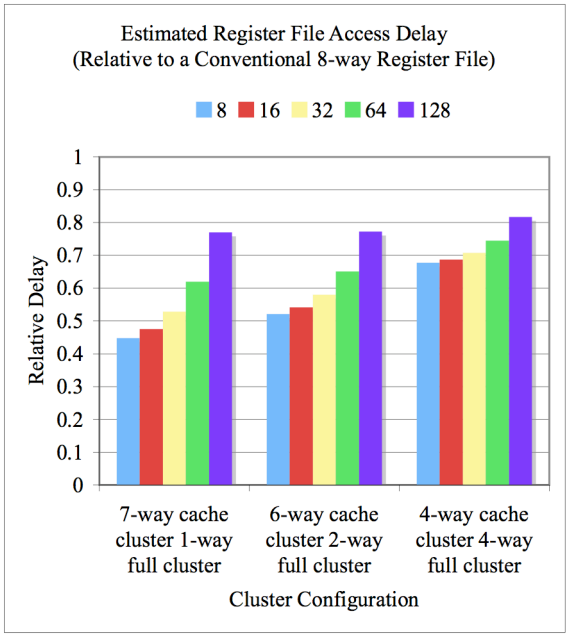


Figure 8: Estimated register file delay.

rates the register files in both clusters and assumes the layout of the two register files minimizes the longest distance to reach any register value. Therefore, the total register file area, not the area of individual register files, is used to estimate delay. This paper does not examine alterative layouts that provide faster access to the register cache. Figure 8 shows the delay estimates of different register cache configurations based on Table 1.

As can be seen by Figures 7 and 8, the configurations with the smallest register cache and a full register file with the fewest read ports result in minimum total area, power, and access delay. However, as will be shown in the next subsection the configurations with the smallest register files do not provide the best cluster load balancing, which results in low IPC performance.

3.2. Effects of Clustering on IPC

The SimpleScalar toolset [9], configured according to Table 2, provides the simulation environment to test the IPC of different processor configurations. The simulation study is based on the following eleven SPEC2000 integer benchmarks [10]: gzip, vpr, gcc crafty, parser, eon, perlbmk, gap, vortex, bzip2, and twolf. Integer benchmarks tend to produce a greater number of branch mispredictions, which highlight the ACRC design's smaller branch misprediction penalty. The SimpleScalar's out-order simulator fast-forwards each benchmark past the initialization phase and then records statistics for 500 million instructions [11].

As mentioned in Section 2, instructions requiring a source register not in the Cache Cluster must execute in the Full Cluster. Moreover, instructions that have both source registers in the Cache Cluster must execute in the Cache Cluster.

This paper also assumes passing instructions between clusters does not incur any additional penalty. Therefore, when the FC Steering Logic (see Figure 3) forwards an instruction from the Cache Cluster to the Full Cluster, the instruction can be selected for execution during the next

Table 2: SimpleScalar configuration.

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 + register read delay cycles
Fetch, dispatch, commit width	8
int,fp issue width	8,4
ROB and Ld/St queue	256 and 128
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

clock cycle. Also, the Full Cluster issue window may receive up to an 'issue-width' number of instructions from the Cache Cluster. For example, in the 6-way Cache Cluster, 2-way Full Cluster configuration, up to 2 instructions can be transferred from the Cache Cluster to the Full Cluster during any clock cycle. This is in addition to the issue window's ability to accept up to eight instructions from the rename stage during each cycle.

The design also requires that each cluster is able to execute any type of instruction. Otherwise, an instruction could end up in a cluster that is unable to execute it, and the processor would stall. Therefore, each cluster contains at least one simple integer, one complex integer, and one load/store functional unit.

The simulation results in Figures 9-12 show how the size of the register cache and the ratio of instructions flowing through the two clusters affect IPC. For the '7-way Cache Cluster, 1-way Full Cluster' configuration in Figure 12, the IPC performance decreases as the register file size decreases. Even with a large (128-entry) register cache, the IPC performance is well below that of the ideal 8-way processor. This is because the cluster with a full register file cannot keep up with demand and thus instructions back up in the issue window. The '6-way Cache Cluster, 2-way Full Cluster' configuration shown in Figure 10 helps relieve this bottleneck by increasing the number of instructions able to flow through the full register file, and attains an average IPC within 5% of the ideal 8-way processor. The configuration with a 32-entry register cache has the best workload balance between the two clusters and results in the highest IPC. In the case of the '4-way Cache Cluster, 4-way Full Cluster' configuration shown in Figure 11, the Cache Cluster becomes the bottleneck. This is because the Full Cluster can only execute instructions that cannot be executed in the Cache Cluster. Therefore, when the number of register cache entries increases, the instructions back up in the Cache Cluster, which starves the Full Cluster, and IPC decreases. Even with a small 8-entry register cache, enough instructions find source registers in the bypass network and register cache to overload the Cache Cluster.

The best performing design will not only attain good load balancing, but also result in a small and fast register file. A fast register file reduces the branch misprediction penalty, which boosts IPC. Figure 8 shows that the fastest register file belongs to the '7-way cache cluster 1-

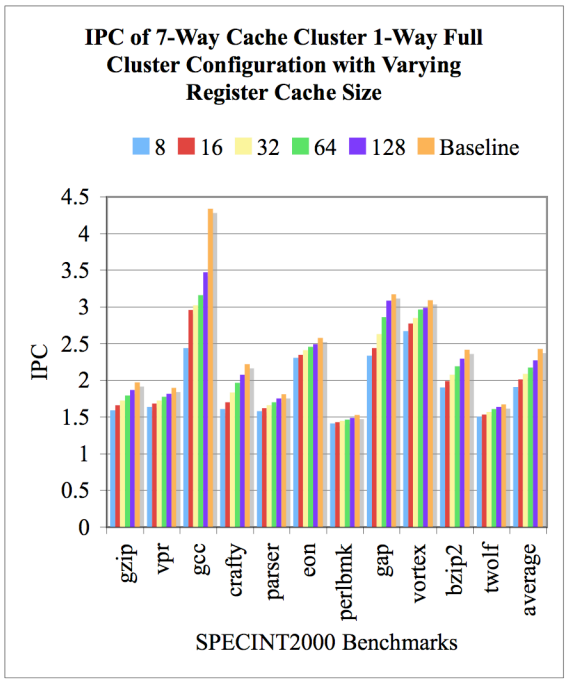


Figure 9: IPC using 7-way cache cluster 1-way full cluster configuration.

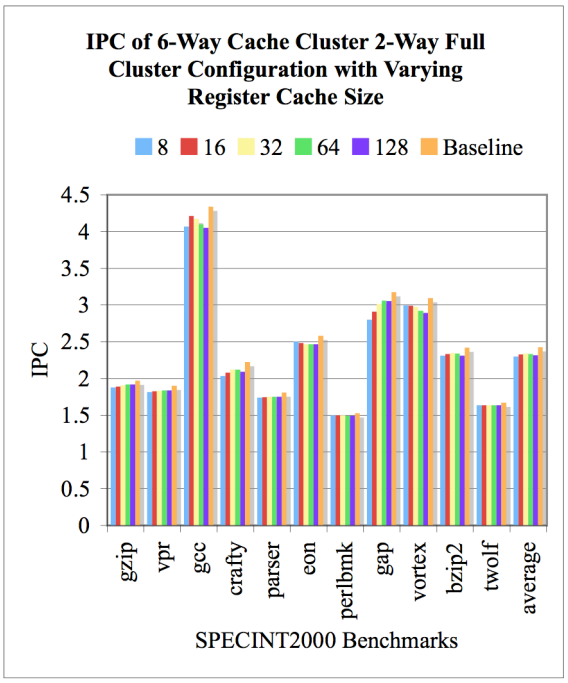


Figure 10: IPC using 6-way cache cluster 2-way full cluster configuration.

way full cluster' design with an 8-entry cache. However, this design results in poor load balancing. Finding the best compromise between a configuration with fast register file access time and good load balancing could require a complicated formula. Fortunately, load balancing has a far greater impact on IPC than register file access delay. For this reason, the next set of simulations use the '6-way Cache Cluster, 2-way Full Cluster' and 32-entry register cache configuration. This configuration not only provides the best cluster load balancing, but also results in one of the smallest total register file areas and results in the best overall IPC.

None of the cluster configurations can match the IPC obtained from a conventional 8-way processor when both designs have the same register file access delay. However, the ACRC design reduces branch misprediction penalty by creating a smaller and faster register file. When the proposed design reduces the branch misprediction penalty by multiple cycles it surpasses the IPC performance of a conventional processor.

This paper assumes the processor's branch misprediction penalty is the register file delay plus 11 clock cycles for tasks such as rename, issue and execute, which is similar to current processors such as the IBM Power 4 [12]. As a result, a processor using a register file with a 3-cycle access delay results in a 14-cycle branch misprediction penalty. As the register file requires more clock cycles to access, the branch misprediction penalty increases and IPC decreases. The proposed ACRC reduces the register file delay by an estimated 42% compared to a conventional processor's register file (see Figure 8). This means a conventional processor with a 3-cycle register file access delay reduces to approximately 1.7 cycles. This results in a 2-cycle register file access delay for the ACRC design and a one-cycle reduction in the branch misprediction penalty. As manufacturing processes shrink and register file access time increases the

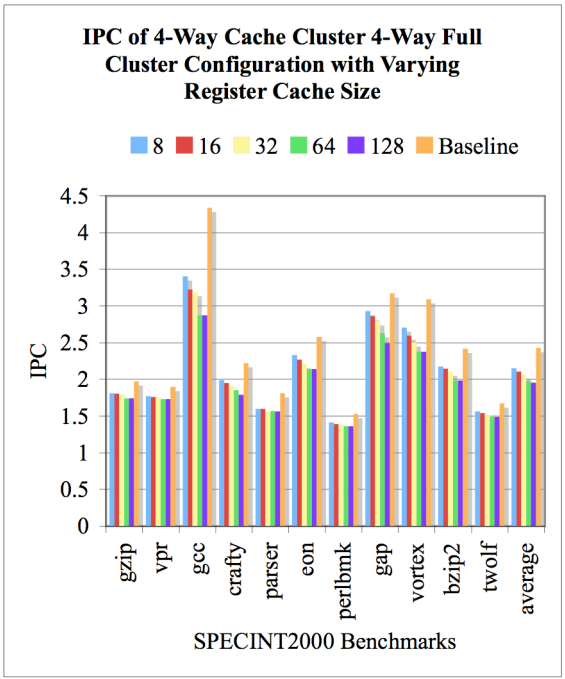


Figure 11: Benchmarks IPC using 4-way cache cluster 4-way full cluster configuration.

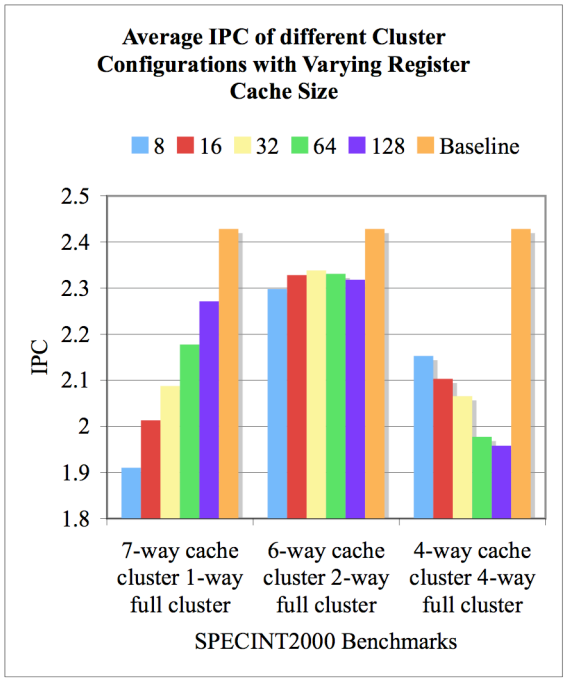


Figure 12: IPC average using different cluster configurations.

ACRC design reduces the branch misprediction penalty by an even greater amount. For example, a conventional processor with a 7-cycle register access delay reduces to approximately 4-cycles in an ACRC design. This, in turn, decreases the branch misprediction penalty from 18 cycles to 15 cycles.

The three comparisons shown in Figure 13 demonstrate the IPC advantage of ACRC for different register file access delays. The first example compares an 8-way conventional processor with a 3-cycle register file delay to an ACRC design with a 2-cycle register file delay. The conventional processor case is comparable to the 3-cycle register delay found in the 8-way Alpha EV8 design using a 125 nm manufacturing process [13]. Here the IPC of the ACRC design is slightly lower. The faster register file cannot makeup for the performance degradation caused by clustering. However, the ACRC reduces the register file area and breaks the issue window and bypassing network into smaller parts. Both of these stages can limit clock rate in a processor [5], and by breaking them into two smaller clusters is an additional performance benefit of the proposed method. The second configuration predicts the register file delay of an 8-way conventional processor using a smaller manufacturing process. This requires doubling the ports of the 2.5-cycle register file designed for a 4-way, 4 GHz processor using a 100 nm process [14]. Applying the register file delay formula results in an 8-way register file that requires 5 cycles to access. An ACRC design reduces the delay to 3 cycles and the resulting IPC closely matches the conventional processor. The third configuration uses a 7-cycle register file delay to simulate future designs. Once designs reach this point, the ACRC design achieves smaller register file size and higher IPC performance compared to the conventional processor.

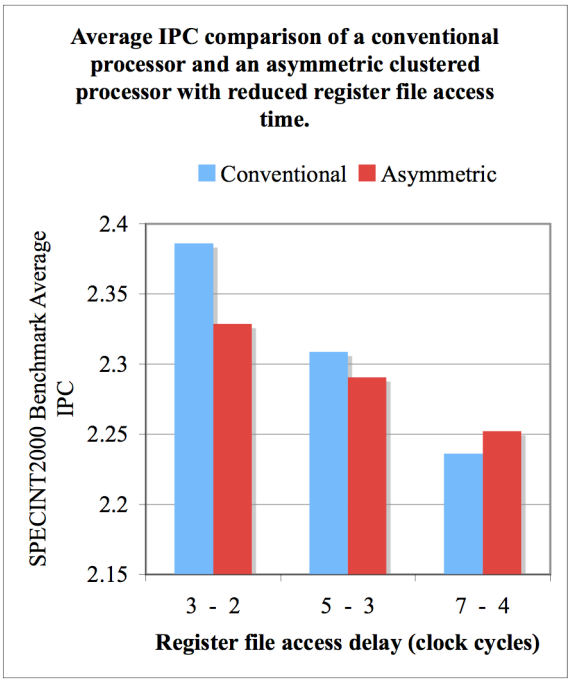


Figure 13: IPC comparison of conventional processor to asymmetric clustering with a reduced register access time and, therefore, branch misprediction penalty.

4. Related Work

The proposed ACRC does not require any changes to the compiler or software since the hardware is responsible for scheduling instructions. Other proposals utilize compiler-generated instructions, instead of hardware, to control register cache usage. The design presented in [15] uses a VLIW (Very Long Instruction Word) processor with a two-level hierarchical register file. The compiler controls activity between a large register file and a small, fast register file to reduce register spill. The Cray-1 also uses explicit software control to manage a two-level register file [16]. Again, the compiler controls instruction scheduling since they are processed in-order. In contrast, the proposed method targets processors that dynamically schedule instructions and exploits run-time information that is not available at compile-time.

Concepts described in [6] and [17] both concentrate on reducing the porting requirements of the register file. These proposals recognize that during most clock cycles FUs do not utilize the dedicated porting resources traditional register files provide. Reducing the number of register ports leads to a smaller register file, but the FUs must share the limited porting resources. This requires multiplexers between the register file and FUs to assign a limited number of register ports to the issued instructions. In addition, designs with limited register file ports also include logic in, or directly after, the issue stage to insure instruction requests do not exceed register file porting resources. This introduces an additional stage in the pipeline to arbitrate for register file resources or imposes a method of stalling the pipeline if the number of register requests exceeds

the register file ports. The ACRC design discussed in this paper does not require multiplexers and the control logic is an extension of the logic used in a conventional wake-up and select system.

Similar to the proposed method, the caching schemes described in [2] and [18] also reduce register file size. However, the register cache is exclusive to the larger secondary register file and FUs only access the register cache. These two-level caching systems are fundamentally different from the ACRC design. A two-level caching system moves register values between levels to serve the needs of the executing instructions. This involves a system to prevent instructions from executing when required registers are not in the register cache. In comparison, ACRC moves instructions between clusters to serve the needs of the register cache. This guarantees that instructions have the register resources needed when they execute. The advantage of the ACRC design is that the issue logic does not need to account for register file limitations, nor does it required any recovery system for register cache misses. The design presented in [2] employs a prefetching logic and a usage table to control the contents of the fully-associative register cache. It requires a copy list to keep track of register activity and pipeline stalls when register cache misses occur. In a similar fashion, the two-level caching system proposed in [18] uses a set of rules to predict which registers are stored in the register cache, and instructions must wait when register values are not in the register cache. In the ACRC design, all instructions write results to both the register cache and the full register file eliminating the need for register prefetching logic. In addition, the 'in-cache' bit system makes managing the register cache simple and straightforward.

The design presented in [14] separates the register file into fast and slow regions by using an asymmetric register file layout. Thus, half of the register file can be accessed in 2 clock cycles, while the other half requires 3 cycles. When possible, a register controller maps registers to fast portions of the register file using a method similar to a direct-mapped cache. The design allows the majority of physical register values to be in the faster portion of the register file; however, it does not reduce the register file size. Instead, speeding up the fast portion is at the expense of reducing the speed of the slow portion. In contrast, this proposal reduces the total register file area and thus the delays of all register accesses.

Each cluster in [19] contains an exclusive set of local registers and instructions execute in the cluster that contains the majority of the required registers. Inter-cluster register passing and global registers provide any missing register data to instructions. To improve performance, this design requires compiler optimizations to provide cluster load balancing. ACRC does not separate registers into global and local types and does not required compiler optimization to achieve high performance.

The use of asymmetric clustering is not limited to register file reduction. The design in [4] uses two specialized clusters: one small, high-speed system for latency-intolerant instructions and another large, slower system to find a large amount of ILP. This allows two clusters, each providing a valuable feature, to perform similarly to a single system with both features. However, [4] maximizes issue window performance and does not reduce register file area and delay.

5. Conclusion

To extract the greatest amount of ILP, processors require a physical register file with a large number of entries and read ports. A single register file with both of these traits becomes large and slow. Clustering two register files, one being a register cache providing a large number of read ports to the most active registers while the other being is a full register file providing a large

number of registers, but with few read ports, achieves a smaller register file area. This decreases the branch misprediction penalty, which increases performance.

Steering instructions to the correct cluster depends on an 'in-cache' bit system. This system monitors the contents of the register cache using a distributed logic similar to the 'ready' bit used in wake-up and select logic. This minimizes logic and delay overhead and allows the system to perform within 5% of a non-clustered system. Combining the decreased register file delay with a low overhead steering system results in a design of smaller size and, in future processors, better IPC than a conventional non-clustered processor.

For future work, a number of topics to increase IPC and further reduce register file area, power, and delay can be explored. For example, the proposed design relies on the hit rate of the register cache to provide load balancing between the clusters. Techniques to better balance the instruction load using static or dynamic methods will increase the performance of the system. In addition, the potential benefits of using set-associative mapping and replacement algorithms to improve the utilization of the register cache need to be further investigated.

References

[1] S. Rixner *et al.*, "Register Organization for Media Processing," *6th International Symposium on High Performance Computer Architecture*, pp. 375-386, January 2000.

[2] R. Balasubramonian, S. Dwarkadas, and D. H. Albonesi, "Reducing the complexity of the register file in dynamic superscalar processors," *34th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-34)*, Dec. 2001.

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

[4] E. Brekelbaum, *et al*., "Hierarchical Scheduling Windows," *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-35)*, pp. 27-36, November 2002.

[5] S. Palacharla, N.P. Jouppi, and J.E. Smith, "Complexity-effective superscalar processors," *24th Annual IEEE/ACM International Symposium on Computer Architecture (ISCA-24)*, 1997.

[6] J. Tseng and K. Asanovic, "Banked Multiported Register Files for High-Frequency Superscalar Microprocessors," *30th Annual IEEE/ACM International Symposium on Computer Architecture (ISCA-30)*, June 2003.

[7] G. Hinton *et al.*, "The Microarchitecture of the Pentium 4 Processor," *Intel Technology Journal*, Q1 2001.

[8] M. Bohr, "Interconnect Scaling – The Real Limiter to High Performance ULSI," *IEEE International Electron Device Meeting Technical Digest*, pp. 241-244, 1995.

[9] SimpleScalar LLC, http://www.simplescalar.com.

[10] SPEC 2000 benchmarks suites, http://www.spec.org.

[11] S. Sair and M. Chamey, "Memory Behavior of the SPEC2000 Benchmark Suite," *IBM Thomas J. Waston Research Center Technical Report RC-21852*, October 2000.

[12] J. M. Tendler *et al.*, "Power 4 System Microarchitecture," *IBM Journal of Research and Development*, Vol. 46, pp. 5.

[13] R. P. Preston *et al*., "Design of an 8-Wide Superscalar RISC Microprocessor with Simultaneous Multithreading," *2002 IEEE International Solid-State Circuits Conference*, February 2002.

[14] S. Hsu *et al.*, "Dynamic Addressing Memory Arrays with Physical Locality," *35th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-35)*, pp. 161-170, November 2002.

[15] J. Zamalea *et al.*, "Two-level Hierarchical Register File Organization for VLIW Processors," *33rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-33)*, pp. 137-146, 2000.

[16] R. Russell, "The Cray-1 computer system," *Communications of the ACM*, Vol. 21, Issue 1, pp 63-72, 1978.

[17] Il Park, M. D. Powell, and T. N. Vijaykumar, "Reducing Register Ports for Higher Speed and Lower Energy," *35th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-35)*, November 2002.

[18] J. L. Cruz, A. González, and M. Valero, "Multiple-banked register file architectures," *27th Annual IEEE/ACM International Symposium on Computer Architecture (ISCA-27)*, pp. 316-325, 2000.

[19] K. I. Farkas *et al.*, "The Multicluster architecture: Reducing cycle time through partitioning," *30th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-30)*, pp. 149–159, 1997.