Exploiting Quiescent States in Register Lifetime

Rama Sangireddy
Dept. of Electrical Engineering
University of Texas at Dallas
Richardson, TX 75080, USA
rama.sangireddy@utdallas.edu

Arun K. Somani
Dept. of Electrical & Computer Engineering
Iowa State University
Ames, IA 50011, USA
arun@iastate.edu

Abstract

Large register file with multiple ports, but with a minimal access time, is a critical component in a superscalar processor. Analysis of the lifetime of a logical to physical register mapping reveals that there are long latencies between the times a physical register is allocated, consumed, and released. In this paper, we propose a TriBank register file*, a novel register file organization that exploits such long latencies, resulting in a larger register bandwidth and a smaller register access time. Implementation of the TriBank register file organization, as compared to a conventional monolithic register file in an 8-wide out-of-order issue superscalar processor reduced the register access time up to 34%, even while enhancing the throughput in instructions per cycle (IPC) by 3% and 14%, for SpecInt2000 and SpecFP2000, respectively.*

1 Introduction

Wide issue processors require multiple ports in the register file which has an adverse affect on the register access time. Besides, a wide-issue superscalar processor is effective only if maximum possible number of instructions are issued during each cycle, which implies that the processor has to view a larger instruction window to achieve sufficient amount of instruction level parallelism (ILP). Large instruction window implies a larger set of in-flight instructions requiring a larger number of physical registers. However, increasing the size of register file adversely affects the register access time.

Register access time plays a critical role in determining the processor clock cycle time [1]. Farkas *et al* [2] have shown that an 8-wide issue superscalar processor handling precise exceptions increases the average instructions per cycle (IPC) throughput as the register file size is increased up to 256. However, the processor loses performance in terms of average number of instructions per second (IPS) for a register file size beyond 128, due to the adverse impact of the large register access time on the processor cycle time.

From above observations, we develop a *TriBank register file*, a register file architecture that performs well in meeting the following two main goals: (a) provide a small register access time to enable a faster processor cycle time, (b) provide a large number of registers to enable dispatching maximum possible number of instructions to issue window for extracting higher ILP. These two goals are met by designing a register file that exploits long latencies involved, in between allocation of register to a logical value and actual consumption of the value by a functional unit, and then in between consumption of the value and actual freeing of physical register for next allocation.

The rest of the paper is organized as follows. Section 2 analyzes the register life time. Section 3 discusses related research. In Section 4 we present a detailed design of the proposed register file organization. The section also discusses the necessary modifications in the microarchitecture. Section 5 analyzes the performance of the architecture. Section 6 concludes the paper.

2 Register lifetime Analysis

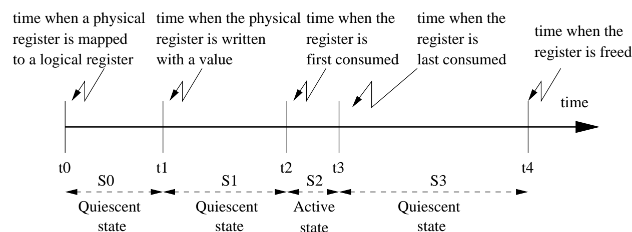

Figure 1: Various stages in the lifetime of a physical register, for a particular mapping to a logical register.

We first study and analyze the activity of a physical register during its lifetime of one logical to physical mapping. In Alpha 21264 processor pipeline [3], fetching and renaming of instructions are performed in-order, while issue and execution of instructions are performed out-of-order. The logical destination register for an in-

struction fetched is mapped to a free physical register at dispatch stage. Subsequent instructions with same logical register as their source operand are mapped to the assigned physical register. The scheme of logical register renaming eliminates false data dependencies. The allocated physical register is freed only when a subsequent instruction with same logical destination is committed, to enable recovery from precise exceptions. The conditions for freeing registers are described in more detail in [2]. The life cycle of a physical register is identified as the time between its allocation to a logical destination at the dispatch stage and the time when it is freed.

The various stages in the register lifetime, as illustrated in Figure 1, are: (i) t0: time at which a free physical register R_p is allocated to a logical destination register R_l of an instruction I_k, (ii) t1: time at which R_p is written with a value. This happens when the instruction I_k enters writeback stage, (iii) t2: time at which the value in R_p is first consumed. This happens when an instruction I_{k+x}, with a logical source operand of R_l, is executed, (iv) t3: time at which the value in R_p is consumed for the last time. This happens when an instruction I_{k+y} $(y > x)$, with a logical source operand of R_l, is executed. And, no further instructions use R_l as source operand until R_l becomes a destination register for another instruction I_{k+z}, where $z > y > x$, (v) t4: time at which the physical register R_p is freed and is ready for the next allocation. This happens when instruction I_{k+z} is committed.

At the microarchitecture level it is not easy to determine when a register is consumed for the last time. To do so, it requires a large overhead of keeping track of all the instructions that are potential customers for the operand with a counting mechanism to track the instructions as and when they are executed. In this study, we identify the time of last consumption of a register value only for the purpose of analyzing the register activity during lifetime. Using that we develop our architecture where we do not have to keep track of time of last consumption of a register.

To study a relationship among these various times, we used Simplescalar-3.0 [4] for the Alpha AXP instruction set to simulate a dynamically scheduled out-of-order issue superscalar processor with the simulation parameters depicted in Table 1. The instructions are traced along the various stages of the processor pipeline and the time intervals between various stages in the lifetime of a register are measured according to the above mentioned specifications. The time intervals measured are: [t1-t0:] time during which the register is waiting for the result to be written into it after it is allocated, [t2-t1:] time during which the register is waiting to be read by a functional unit after it is written into, [t3-t2:] time during which the register is active as supplier of an operand to functional units, [t4-t3:] time during which the register is waiting to be freed after it is consumed for the last time.

Table 1: Simplescalar simulation parameters.

Parameter	*Value*
Instruction cache *- latency*	32KB, 2-way, 32B line 1 cycle
Data cache *- latency*	32KB, 4-way, 32B line 1 cycle
Branch predictor *−mis − prediction latency* *−return address stack size*	bimodal, 2K table size 7 cycles 8
Instruction issue queue size	128
Load/store queue (LSQ) size	64
ReOrder buffer (ROB) size	64
Issue width	2/4/8/16
Commit width	2/4/8/16
L2 unified cache *−latency*	256KB, 4-way, 64B line 6 cycles
TLB *−D − TLB* *−I − TLB* *−latency*	 512KB, 128 entries 256KB, 64 entries 30 cycles
Memory *−latency first, next* *−bus width*	 70, 2 cycles 8B

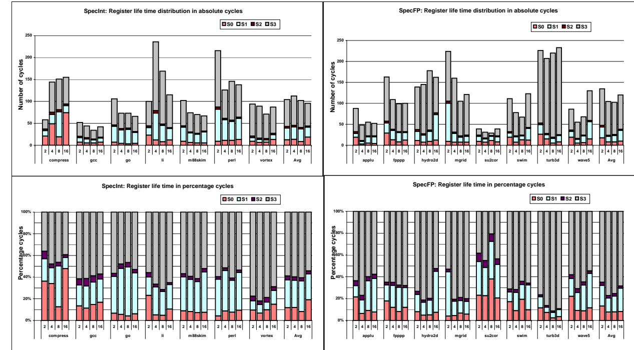

Figure 2: Physical register lifetime distribution in (above) absolute number of cycles (below) percentage of lifetime.

The average time interval between each stage of register lifetime is shown for various Spec benchmarks in Figure 2. The register lifetime can be classified, as illustrated in Figure 1, into an active state (S2) where the register is supplying the values to functional units, and quiescent states (S0, S1, and S3) where the physical register is inactive waiting for some action to take place. It can be observed that the average active time of the register is exceptionally small (around 1% to 5%). This is mainly due to two reasons. First, it is observed that around 85% of the time a register value is read at most once. Second, some registers are never read as the values they hold are either supplied to their consumers through the bypass logic or even not read at all.

The observations that emanate from the above analysis are: (a) physical registers are allocated at dispatch stage, early in the pipeline, and experience a long latency before consumption, (b) amount of duration when the register is an active supplier of values to consumers is very small as compared to its long lifetime, (c) after the last consumption by a functional unit, there is a long latency before a register is freed up. Hence, a

more aggressive logical to physical mapping at dispatch stage can be obtained by hiding the latency in freeing of registers, with the support of a mechanism to handle precise exceptions.

3 Related Research

Alpha 21264 microprocessor [3] uses a replicated register file organization to reduce the number of ports, wherein each copy can be accessed by only a few functional units. Tseng *et al* [5] have examined designs of such multiple bank with fewer ports to reduce power and area. Cruz *et al* [6] used a multiple-banked organization for implementing a two-level register file. Level two (L2) register file is a large bank that holds all register values and is used for logical to physical register mapping. Level one (L1) register file, a smaller bank and closer to the ALU, maintains a copy of those L2 registers that are potential consumer operands. This organization takes advantage of the latency in the quiescent states S0 and S1, and small active state S2. The scheme does not exploit the long latency in quiescent state S3, as the L2 register file still holds the values until the registers are freed. Also, implementation of a large L2 register file results in a large latency in access from L2 to L1.

Balasubramonian *et al* [7] proposed and evaluated two orthogonal designs - two level and multi-banked register file. In the two-level organization, L1 bank is used for logical to physical register mapping and holds values until they are consumed. After the consumption register is moved to L2 and maintained until it is freed. This exploits the quiescent state S3. Such implementation helps in organizing a relatively smaller L1 register file as compared to a conventional register file, and hence reduces the register access time. However, the microarchitecture keeps track for each physical register value, when it is consumed for the last time, with a large hardware overhead. The approach in this case and in [8] results in difficulty of managing the complexity and additional latency of the control logic required to handle read and write bank conflicts and the mapping of register ports to functional units.

Some of the other techniques proposed in the past for an effective utilization of register resources are as follows. Borch *et al* [9] have recently proposed caching of registers. To reduce register access time, A cluster based design of execution units and the extension of storage hierarchy for each cluster, in place of a global register file, is introduced by Dally [10]. Zyuban and Kogge [11] have developed energy models for multi-ported register files with a variety of architectural parameters, and assert that the centralized register files would become the dominating power component of next-generation superscalar computers. Gonzalez *et al* [12] proposed a *virtual registers* architecture with a strategy to reduce the pressure on register file by delaying the allocation of physical registers until instructions complete, instead of doing it in the decode stage.

4 TriBank Register File

The TriBank Register file organization consists of three banks of physical registers with a heterogeneous structure, as shown in Figure 3. Each register bank consists of a number of registers and ports according to the architecture requirements as discussed later. The bank RF1 is closer to ALU and consists of a small number of registers and a sufficient number of read and write ports to support issue width of the processor. Functional units are always supplied with the data only from the registers in RF1. The RF2 bank consists of a large number of physical registers and a few read and write ports. The physical registers in RF2 are used for logical to physical register mapping at dispatch stage. Results are always written to registers in RF2. The bank RF3 also consists of a large number of registers and a few ports. From Figure 2, we observe that the average time during intervals t2-t0 (= S0+S1) and t4-t3 (S3) is around 45-50% each. Hence we propose that there be an equal number of registers in RF2 and RF3 banks.

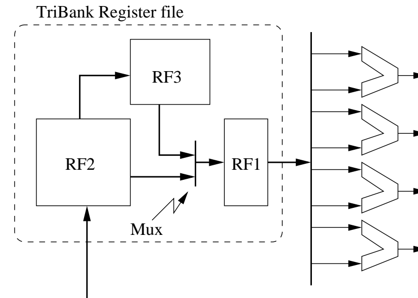

Figure 3: A TriBank Register file organization.

The RF1 bank obtains a copy of only those register values, from RF2 or RF3, that are soon to be consumed, and so holds the values in the active state S2. The register values in RF2 are transferred to RF3 whenever RF3 has free registers and thus simultaneously freeing the corresponding registers in RF2. Section 4.3 discusses in detail the process of transferring register values from RF2 to RF3, and conditions for freeing of registers in RF2 and RF3. The register in RF2 is freed up much earlier than that is done in a conventional monolithic register file. The following subsection discusses in detail the various mechanisms for register value copying to RF1 for consumption.

4.1 Register value fetching to RF1

The issue stage in the processor consists of wakeup logic and select logic. The instruction is said to be in wakeup

state when it is in the reservation station waiting for both its source operands to be ready. When an instruction has all its source operands ready, it sends a ready signal to the select logic. The select logic sends a grant signal to the instruction permitting it to be executed, when necessary functional unit is available. The copying of register values to RF1 from RF2 or RF3 occurs when the values are ready to be consumed by an instruction that is ready to be issued for execution. To avoid any delay, the source operands for an instruction are copied to RF1, when the instruction in the wakeup logic sends a ready signal to the select logic. This mechanism ensures that the values are fetched to RF1 in time to be consumed, and also not much in advance before being consumed.

It is necessary that all active registers, to be consumed soon, be kept in RF1. Thus we propose that RF1 be maintained as a small fully associative register file, similar to the Multi-banked register files proposed by Cruz *et al* [6]. In this case, the values are replaced according to the least recently consumed (LRC) policy. The registers in RF1 can also be easily marked as consumed, and thus enable replacing only those registers that are marked. A pitfall with this scheme is that the register value, though required, may not actually be read from the register file, but is supplied directly via the bypass logic. In that case, the register would not be marked as consumed and hence never replaced. To avoid this, instead of setting the flag for a register when it is read, the flags are set whenever the consumer instruction is executed, irrespective of whether the source operands are read from RF1 or obtained via the bypass logic.

At the microarchitecture level, it is not possible to know in advance when a register value is consumed for the last time without a large overhead of keeping track of all the instructions that are customers for the operand. Hence, retaining the value in RF1 until it is completely consumed cannot be guaranteed with 100% accuracy. However, the register value replacement in RF1 using LRC policy increases the chances that the value is held for long enough to be consumed more than once if needed. Even with a rare chance that the value is replaced before it is consumed for the last time, it can again be copied from RF2 or RF3 when that last instruction is to be issued for execution as per usual procedure described above.

4.2 Register Transfer from RF2 to RF3

At the register renaming stage, a free physical register in RF2 bank is used to allocate to a logical destination register at time t0. The value produced after the execution of the corresponding instruction is written into the physical register in RF2 at time t1. Since the two register banks RF2 and RF3 are maintained to be of same size, a direct-mapping (one-to-one) scheme is followed, i.e., transfer of values from RF2 to RF3 is done strictly in a one-to-one correspondence. The transfer of a physical register value in RF2 to RF3 bank will happen upon the satisfaction of both the following conditions:

1. The physical register in RF2 bank must have already obtained its value, i.e., time t1 must have elapsed in its current lifetime, and
2. the corresponding physical register in RF3 bank should be free.

When the above conditions satisfy, register value in RF2 is transferred to the corresponding register in RF3. Subsequently the register in RF2 is freed and is ready for a mapping to a new logical destination register. In direct-mapping policy, for applications in which latency of quiescent state S3 for a register is much larger compared to the latency in state S0+S1 during the lifetime of a logical to physical mapping, most of the time the register in RF2 will be moved to RF3 much after it is consumed. In this case, the effective lifetime of a register mapping in the processor's view is the latency in freeing the register in RF2, which translates to the long latency of freeing a register from RF3.

However, if the latency in state S0+S1 is larger than that in state S3, a register value might be moved from RF2 to RF3 even before it is consumed. This will not hinder the reading of operands as the value can still be fetched from RF3. However, there is a pitfall in this scenario. Consider a case when a value in physical register pr5 is moved from RF2 to RF3 even before it is consumed, and then the register pr5 in RF2 is freed for next allocation. Subsequently the register pr5 in RF2 obtains another value corresponding to the next logical renaming. A following instruction that sources the logical operand corresponding to pr5 in RF3 (former mapping) or the logical operand corresponding to pr5 in RF2 (later mapping) has to read the value correctly. To illustrate the scenario more clearly, consider the example shown in Figure 4.

Code with logical registers	Code with renamed registers
lr6 ← ... ;	pr5 ← ...
.... ;	....
.... ;	....
... ← lr6 ;	... ← pr5
.... ;	....
.... ;	....
lr4 ← lr2 ;	pr5 ← pr9
.... ;	....
... ← lr6 ;	... ← pr5 (in RF3)
... ← lr4 ;	... ← pr5 (in RF2)
.... ;	....

Figure 4: Example code.

Let us say, sometime in between the first two instructions shown, the value in pr5 is moved from RF2 to

RF3. In that case, the microarchitecture should make sure that the second instruction reads the value from pr5 register in RF3 bank. Besides, now pr5 in RF2 is free to be allocated and thus gets mapped to lr4. For the next two instructions shown, the reading of operands lr6 and lr4 happens to be from pr5 and thus operands have to be read from RF3 and RF2, respectively as per the correct mapping shown. These issues can be addressed by the mechanism shown in Figure 5.

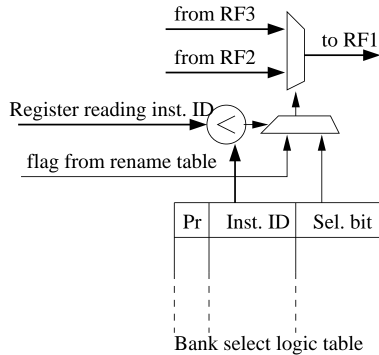

Figure 5: Selection of physical register value from correct register bank.

A *bank select logic table* with tuple [physical register, instruction id., select bit] is maintained at the register read stage in the processor. When the value in pr5 is moved from RF2 to RF3, the register mapping table in the dispatch stage is updated by setting a flag as one for the mapping $lr6 \longleftrightarrow pr5$ in the rename map table. Also, the instruction id. against pr5 in the *bank select logic table* is set to highest number and the select bit is set to one. For a subsequent mapping of $lr4 \longleftrightarrow pr5$ the corresponding flag in the rename map table is set to zero, and the id. of that instruction replaces the highest number set in the *bank select logic table* for pr5.

Now, for the second instruction shown in Figure 4, since the id. of that instruction is smaller than the instruction id. in the *bank select logic table* (it is assumed that id. for instructions are assigned in increasing order at decode stage in program order), the logic uses the select bit from the *bank select logic table* to read the register value from the appropriate bank. For the fourth and fifth instructions shown in Figure 4, since their instruction id. are larger than that in *bank select logic table*, the flag from rename map table is used to read the register value from appropriate bank. It is important to note that, the selection logic to read the operand value from appropriate bank functions in parallel to the reading of the operand values from both RF2 and RF3 banks, and so do not add any delay in the register access time. Hence, the addition of this small selection logic does not impact the overall register access time.

4.3 Freeing of registers in RF2 and RF3

As discussed above, a register in RF2 is freed whenever the register value is transferred to the corresponding register in RF3 bank. A register in RF3 is freed according to the conditions followed in the case of a conventional monolithic register file. That is, for a current logical to physical register mapping, the physical register in RF3 is freed when a subsequent instruction with same logical destination commits. The freed register in RF3 is again ready to assume another register value from RF2.

The proposed architecture recovers from branch mis-predictions as follows. In a conventional architecture, when a branch mis-prediction occurs, the instructions that are not yet committed following the mis-predicted branch are squashed from the pipeline along with the corresponding values written in register file and the logical to physical register mappings due to those instructions. Subsequently, for instructions that are issued the source operands are read according to the logical to physical register mapping performed before the branch instruction. In the proposed architecture, the corresponding values existing in RF2 and RF3 are squashed when branch mis-prediction occurs. For new instructions issued, the source operand values are obtained from either RF2 or RF3, where ever they exist. Consider the example shown in Figure 6.

Code with logical registers	Code with renamed registers
lr6 ← ... ;	pr3 ← ...
... ← lr6 ;	... ← pr3
branch to LOOP ;	branch to LOOP
lr5 ← lr4 ;	pr8 ← pr9
lr6 ← ... ;	pr2 ← ...
... ← lr6 ;	... ← pr2
.... ;	....
LOOP: ... ← lr6 ;	... ← pr3

Figure 6: Example code.

Initially, the logical register lr6 is mapped to physical register pr3, and thus is read as a source operand for the next instruction. When the branch is predicted to be not taken and the following instructions are fetched, the logical register lr6 is renamed to a physical register pr2, different from the earlier mapping. The subsequent dependent instructions read from physical register pr2. However, when the branch is realized to be mis-predicted, the instruction in the correct path after the branch requires the logical value from lr6 which actually refers to pr3. Thus when the recovery mechanisms are initiated and the instructions are processed on the correct program path, instruction requiring the value in physical register pr3 will find the register value either in RF2 or RF3, depending on the timing of transfer of the value from RF2 to RF3.

4.4 Impact on bypass logic

The design of a register file, also impacts the complexity of the bypass logic. A conventional monolithic register

file with one cycle latency will have one level of bypass. However, a large monolithic register file to support an 8-issue superscalar processor (requires around 128 registers with 16 read and 8 write ports) is unlikely to be implemented with one-cycle latency. Such a register file requires two levels of bypass logic which incurs a significant cost. For a register file with two-cycle latency, designing only one level of bypass logic further degrades the performance [6]. For the proposed architecture, since operands are supplied only from the RF1 register bank, complexity of the bypass logic is not affected and is the same as for a register file with single-cycle latency and a single level of bypass logic.

5 Performance Evaluation

Table 2: Configurations for various register file organizations simulated. Access time is measured at 0.18μ.

Index	*Configuration* ($IW = Issue\ Width$) read port (rp), write port(wp), access latency	$(IW) = 4$ *access time (ns)*	$(IW) = 8$ *access time (ns)*
C1: Base	RF = 128 registers rp = 2*IW , wp = IW	1.0614	1.4873
C2: two-level organization	RF1 = 16 registers rp = IW , wp = IW, 2 cycles	0.8046	0.9791
	RF2 = 128 registers rp = IW , wp = IW, 2 cycles	0.9428 0.9428	1.2302 1.2302
C3: TriBank organization	RF1 = 16 registers rp = 2*IW , wp = IW, 1 cycle	0.8046	0.9791
	RF2 = 64 registers rp = IW , wp = IW, 1 cycle	0.8922	1.1552
	RF3 = 64 registers rp = IW, 1 cycle number of buses from RF2 to RF3 = IW	0.8687	1.0844
C4: TriBank organization	RF1 = 16 registers rp = 2*IW , wp = IW, 1 cycle	0.8046	0.9791
	RF2 = 128 registers rp = IW , wp = IW, 2 cycles	1.0614	1.4873
	RF3 = 128 registers rp = IW, 2 cycles number of buses from RF2 to RF3 = IW	0.9428	1.2302

We used Simplescalar-3.0 [4] for the Alpha AXP instruction set to simulate a dynamically scheduled out-of-order issue superscalar processor with the simulation parameters summarized in Table 1, with a few modifications as below. In Simplescalar, instruction issue queues and the re-order buffer (ROB) constitute one single centralized circular structure called the Register Update Unit (RUU). The simulator has been modified to model the instruction issue queues and the ROB structures. Besides, an Alpha 21264 processor [3] based architecture is implemented with split integer and floating-point physical register files and issue queues for a 4-wide and an 8-wide out-of-order issue processor. The configurations for four different register file organizations are used for the analysis as shown in Table 2. The benchmark programs are simulated for 500-1000 million instructions depending on the characteristics of each program, and the simulation was fast-forwarded past the initial warm-up phases.

The configuration C1 is a base processor implementation. In C1 the monolithic register file is implemented as a single cycle register file with one-level bypass logic. The configuration C2 is implemented in line with the two-level register file design proposed by Cruz *et al* [6]. The configuration C3 is used to evaluate the performance of the TriBank register file organization with RF2 and RF3 constituting 64 physical registers as against a conventional monolithic register file with 128 physical registers. This constitutes an even-handed comparison of the TriBank scheme with C1 and C2 in terms of number of physical registers available for data storage. However, note that the physical register bandwidth available for logical register mapping in C3 will be half of that available in case of C1 and C2. Hence, to measure the performance of TriBank scheme with same logical to physical register mapping bandwidth, we also evaluate the configuration C4. In C2, C3 and C4, the small register bank closer to ALU is implemented as a single cycle one-level bypass register file. The RF2 and RF3 in C3 are implemented with a single cycle latency, while RF2 in C2, and RF2 and RF3 in C4 are implemented with a two-cycle latency.

We used SPEC2000 benchmarks, and evaluated both the integer and floating-point programs. The access time models of CACTI-2.0 [13] at 0.18μ technology are used with necessary modifications to generate cycle times for multiported register files, to evaluate the complexity of proposed register file structures in comparison to the baseline organization. The CACTI-2.0 was made to analyze caches with a fewer ports. We have greatly expanded the tool to analyze register files (which typically do not use sense amps like caches) with a larger number of ports. We compute the access time of the RF2 and RF3 register banks while accounting for the multiplexer logic used to select values from either of the banks.

5.1 Results and analysis

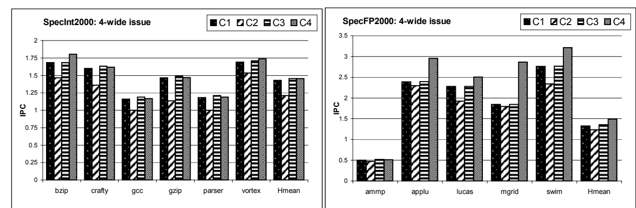

Figure 7: Instructions per cycle (IPC) throughput for various register file configurations in the 4-wide issue processor.

Figures 7 and 8 show the IPC throughput for various integer and floating-point Spec2000 programs for 4-wide and 8-wide processors, respectively. The degradation in IPC for configuration C2 as compared to configuration C1 is in line with the analysis given by Cruz *et*

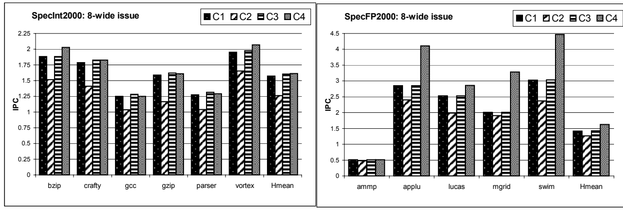

Figure 8: Instructions per cycle (IPC) throughput for various register file configurations in the 8-wide issue processor.

al [6]. It is observed, that the TriBank register file configurations C3 and C4 perform either similar or better as compared to the base configuration. An enhancement in IPC by 2% and 3% for a 4-wide processor, and by 2% and 1% for an 8-wide processor, is seen with configuration C3, for SpecInt and SpecFP programs, respectively. On the other hand Implementation of configuration C4 enhances IPC by 2% and 12% for a 4-wide processor, and by 3% and 14% for an 8-wide processor for SpecInt and SpecFP programs, respectively. For certain integer benchmarks like crafty, gcc, gzip, and parser, C3 is observed to be performing slightly better than C4. This is due to the larger access cycles for register banks RF2 and RF3 in C4 as compared to those in C3. Hence, this performance difference can vary in either way depending on the architecture implementation technology and other factors that govern access time of a register bank.

It can be observed from Table 2 that the register organization in C3 significantly reduces the register access time by 25%, while implementation of C4 register file architecture reduces the register access time by 34%. The advantage gained by the inclusion of RF3 register bank, used to retain the register values before being freed, is explained as follows. Cruz *et al* [6] have shown that a large RF2 (with large access time) and small RF1 results in IPC loss though instruction throughput per second is gained as it increases the pipeline latency. We have shown that splitting the large register bank into RF2 and RF3 provides the same register bandwidth for dispatch stage while reducing pipeline latency improving both IPC and the instruction throughput. This is a significant contribution.

6 Conclusions

An effective register file organization in a superscalar processor is one that provides a small register access time, even while providing a large number of registers and multiple ports. We developed *TriBank register file* organization, a novel architecture that exploits the long quiescent states in the lifetime of a logical to physical register mapping. Implementation of the TriBank register file organization, as compared to a conventional monolithic register file in an 8-wide out-of-order issue superscalar processor enhanced the throughput in instructions per cycle (IPC) by 3% and 14%, while reducing the register access time by 34% .

References

[1] S. Palacharla, N. P. Jouppi, and J. E. Smith, "Complexity-Effective Superscalar Processors", *Proc. 24th Annual International Symposium on Computer Architecture*, 1997, pp. 206-218.

[2] K. I. Farkas, N. P. Jouppi, and P. Chow, "Register File Design Considerations in Dynamically Scheduled Processors" *Proc. Second International Symposium on High-Performance Computer Architecture*, 1996, pp. 40-51.

[3] R. E. Kessler, "The Alpha 21264 microprocessor", *IEEE Micro*, Volume: 19, Issue: 2, Mar-Apr 1999, pp. 24-36.

[4] Doug Burger and Todd M. Austin, "The SimpleScalar Tool Set, Version 2.0", Computer Sciences Department Technical report # 1342, University of Wisconsin-Madison, June 1997.

[5] J. H. Tseng and K. Asanovic, "Banked Multiported Register Files for High-Frequency Superscalar Microprocessors", *Proc. 30th Annual International Symposium on Computer Architecture*, 2003.

[6] J. L. Cruz, A. Gonzalez, M. Valero, and N. P. Topham, "Multiple-banked Register File Architectures", *Proc. 27th Annual International Symposium on Computer Architecture*, 2000, pp. 316-325.

[7] R. Balasubramonian, S. Dwarkadas, and D. H. Albonesi, "Reducing the Complexity of the Register File in Dynamic Superscalar Processors", *Proc. 34th ACM/IEEE International Symposium on Microarchitecture, MICRO-34*, 2001, pp. 237-248.

[8] I. Park, M. D. Powell, and T. N. Vijayakumar, "Reducing Register ports for higher speed and lower energy", *Proc. 35th Annual International Symposium on Microarchitecture*, 2002.

[9] E. Borch, E. Tune, S. Manne, J. Emer, "Loose loops sink chips" *Proc. Eighth International Symposium on High-Performance Computer Architecture*, 2002, pp. 270-281.

[10] W. J. Dally, "Interconnect-limited VLSI architecture", *Proc. IEEE International Conference on Interconnect Technology*, 1999, pp. 15-17.

[11] V. Zyuban and P. Kogge, "The Energy Complexity of Register Files", *Proc. International Symposium on Low Power Electronics and Design*, 1998, pp. 305-310

[12] A. Gonzalez, M. Valero, J. Gonzalez, and T. Monreal, "Virtual registers", *IEEE Fourth International Conference on High-Performance Computing*, 1997, pp. 364-369

[13] S. E. Wilton and N. P. Jouppi, "An Enhanced Access and Cycle Time Model for On-chip Caches", DEC WRL Research 93/5, July 1994.