Multiple-Banked Register File Architectures

José-Lorenzo Cruz, Antonio González and Mateo Valero

Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya
Jordi Girona, 1-3 Mòdul D6
08034 Barcelona, Spain
{cruz,antonio,mateo}@ac.upc.es

Nigel P. Topham

Siroyan Ltd
Wyvols Court
Swallowfield
Berkshire RG7 1WY, U.K.
ntopham@siroyan.com

Abstract

The register file access time is one of the critical delays in current superscalar processors. Its impact on processor performance is likely to increase in future processor generations, as they are expected to increase the issue width (which implies more register ports) and the size of the instruction window (which implies more registers), and to use some kind of multithreading. Under this scenario, the register file access time could be a dominant delay and a pipelined implementation would be desirable to allow for high clock rates.

However, a multi-stage register file has severe implications for processor performance (e.g. higher branch misprediction penalty) and complexity (more levels of bypass logic). To tackle these two problems, in this paper we propose a register file architecture composed of multiple banks. In particular we focus on a multi-level organization of the register file, which provides low latency and simple bypass logic. We propose several caching policies and prefetching strategies and demonstrate the potential of this multiple-banked organization. For instance, we show that a two-level organization degrades IPC by 10% and 2% with respect to a non-pipelined single-banked register file, for SpecInt95 and SpecFP95 respectively, but it increases performance by 87% and 92% when the register file access time is factored in.

Keywords: Register file architecture, dynamically-scheduled processor, bypass logic, register file cache.

1. Introduction

Most current dynamically scheduled microprocessors have a RISC-like instruction set architecture, and therefore, the majority of instruction operands reside in the register file. The access time of the register file basically depends on both the number of registers and the number of ports [8]. To achieve high performance, microprocessor designers strive to increase the issue width. However, wider issue machines require more ports in the register file, which may significantly increase its access time [2]. Moreover, a wide issue machine is only effective if it is accompanied by a large instruction window [14] or some type of multithreading [13]. Large instruction windows and multithreading imply a large number of instructions in-flight, which directly determines the number of required registers [2]. However, increasing the number of register also increases the register file access time. On the other hand, technology evolution produces successive reductions in minimum feature sizes, which results in higher circuit densities but it also exacerbates the impact of wire delays [7]. Since a significant part of the register file access time is due to wire delays, future processor generations are expected to be even more affected by the access time problem.

Current trends in microprocessor design and technology lead to projections that the access time of a monolithic register file will be significantly higher than that of other common operations, such as integer additions. Under this scenario, a pipelined register file is critical to high performance; otherwise, the processor cycle time would be determined by the register file access time. However, pipelining a register file is not trivial. Moreover, a multi-cycle pipelined register file still causes a performance degradation in comparison with a single-cycle register file, since a multi-cycle register file increases the branch misprediction penalty. Besides, a multi-cycle register file either requires multiple levels of bypassing, which is one of the most time-critical components in current microprocessors, or processor performance will be significantly affected if only a single-level of bypassing is included.

In this paper we propose a register file architecture that can achieve an IPC rate (instructions per cycle) much higher than a multi-cycle file and close to a single-cycle file, but at the same time it requires just a single level of bypass. The key idea is to have multiple register banks with a heterogeneous architecture, such that banks differ in number of registers, number of ports and thus, access time. We propose a run-time mechanism to allocate values to registers which aims to keep the most critical values in the fast banks, whereas the remaining values are held in slower banks.

We show that the proposed organization degrades IPC by 10% and 2% with respect to a one-cycle single-banked register file for SpecInt95 and SpecFP95 respectively, assuming an infinite number of ports. However, when the register file cycle time is factored in and the best configuration in terms of instruction throughput (instruction per time unit) is chosen for each register file architecture, the proposed architecture outperforms the single-banked register file by 87% and 92% respectively. When compared

27th. Annual International Symposium on Computer Architecture (ISCA 2000), Vancouver, Canada, June 12-14, 2000

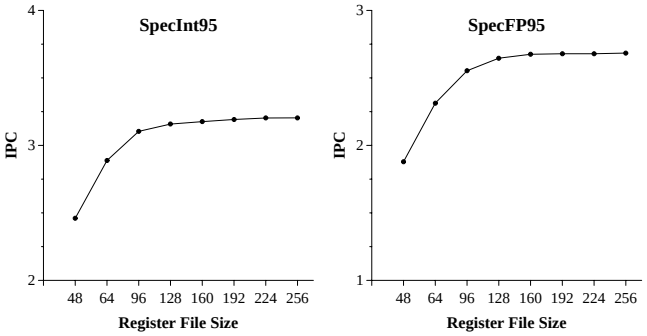


Figure 1: IPC for a varying number of physical registers. The harmonic mean for SpecInt95 and SpecFP95 is shown. (for this experiment we assume the architectural parameter described in section 4.1, but a reorder buffer and an instruction queue of 256 entries).

with a two-stage pipelined, single-banked register file with just one level of bypass, the proposed architecture provides a 10% (SpecInt95) and 4% (SpecFP95) increase in IPC for infinite number of ports and a 9% (SpecInt95) increase in instruction throughput for the best configuration. Moreover, the performance figures for the two-stage pipelined organization are optimistic since we assume that the register access can be pipelined into two stages of the same duration and without any inter-stage overhead.

The rest of this paper is organized as follows. Section 2 motivates this work by presenting some statistics about the impact of register file access time on processor performance and bypass logic. Section 3 presents different multiple-banked architectures and describes in detail the register file cache architecture, which is the main contribution of this work. Performance statistics are discussed in section 4. Section 5 outlines the related work and finally, section 6 summarizes the main conclusions of this work.

2. Impact of the Register File Architecture

The register file provides the source operands and stores the results of most instructions. Dynamically scheduled processors rename at run-time logical registers to physical registers such that each result produced by any instruction in-flight is allocated to a different physical register. In this way, name dependences are eliminated and instruction parallelism is increased. The cost of this technique is that a large number of registers may be required. Figure 1 shows the harmonic mean of the IPC of an 8-way issue processor with a varying number of physical registers for the SpecInt95 and SpecFP95 benchmarks. Details about the evaluation framework can be found in section 4.1. Across the whole paper we use the same architectural parameters with the exception that in Figure 1 we assume a reorder buffer and an instruction queue of 256 entries in order to evaluate larger register files. Note that the performance curves start to flatten beyond 128 registers.

The previous experiment assumed a one-cycle latency for the register file. However, a register file with 128 registers and 16 read ports and 8 write ports is unlikely to have such a low access time. However, a two-cycle register file has some important implications for processor performance and complexity compared with a single-cycle register file, as observed by Tullsen et al. [12]:

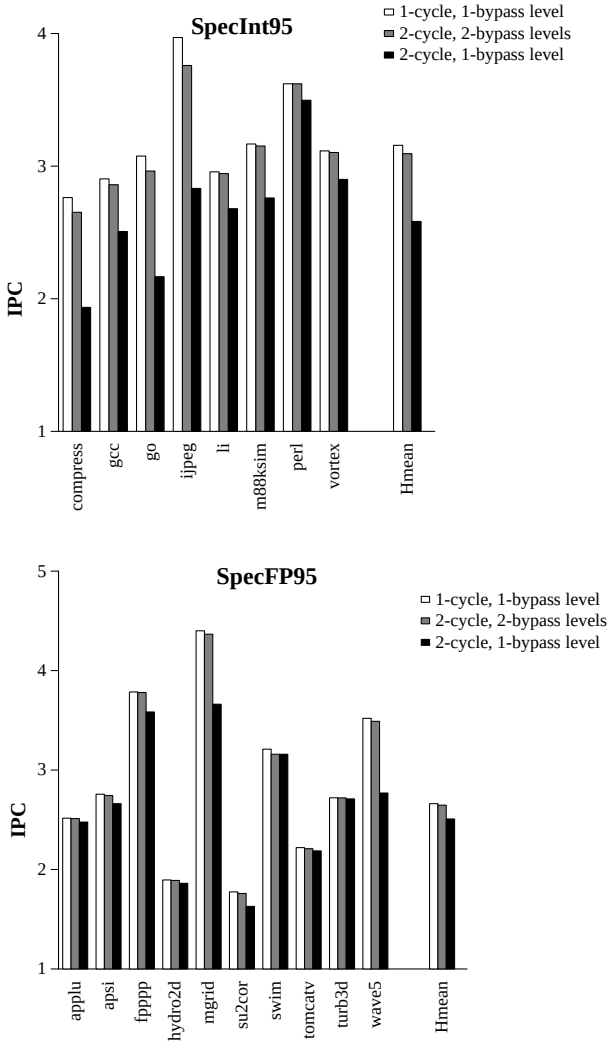


Figure 2: IPC for a 1-cycle register file, a 2-cycle register file and a 2-cycle register file with just one level of bypass.

- The branch misprediction penalty is increased since branches are resolved one cycle later.
- The register pressure is increased since the time that instructions are in-flight is increased.
- An extra level of bypass logic is required. Each bypass level requires a connection from each result bus to each functional unit input, if full bypass is implemented. This incurs significant complexity.

An alternative approach to reducing complexity, at the expense of a lower performance, is to keep just one level of bypass. In this case, only the last level of bypass is kept in order to avoid 'holes' in the access to register data. In this context a hole refers to the fact that a value is available in a given cycle (from the bypass network), then is not available in a subsequent cycle, and later on is available again (from the register file). Holes are undesirable since they would significantly increase the complexity of the issue logic.

Figure 2 shows the IPC for the whole SPEC95 benchmark suite comparing three different architectures of the register file

architectures: a) one-cycle latency and one level of bypass; b) two-cycle latency and two levels of bypass; and c) two-cycle latency with one level of bypass.

We can see that an additional cycle in the register file access time slightly degrades performance when all additional bypasses are implemented. Not surprisingly, performance significantly decreases if only a single level of bypass is available. The impact is higher for integer codes, due in part to their much higher branch misprediction rates. Moving from a two-cycle register file with one bypass level to a two-cycle register file with two bypass levels produces an average increase in IPC of 20% for SpecInt95. A one-cycle register file results in an average speedup of 22%. Note that all programs are significantly affected by the register file latency and the number of bypass levels. For SpecFP95, the differences are lower (6% and 7% respectively) but still quite significant. This results are consistent with the study of Tullsen et al. [12], who reported a less than 2% performance decrease when the register file latency increased from 1 to 2 cycles with two levels of bypass.

The register file optimizations proposed in this paper are also based on the observation that a very small number of registers would be required to keep the processor at its maximum throughput if they were more effectively managed. This is because many physical registers are "wasted" due to several reasons:

- Registers are allocated early in the pipeline (decode stage) to keep track of dependences. However, they do not hold a value until the instruction reaches the write-back stage.
- Some registers hold values that will be used by later instructions that have not yet entered the instruction window.
- For reasons of simplicity, registers are released late. Instead of freeing a register as soon as its last consumer commits, it is freed when the next instruction with the same logical destination register commits.
- Some registers are never read since the value that they hold is either supplied to its consumers through bypass paths or never read.

Figure 3 shows in solid lines the cumulative distribution of the number of registers that contain a value that is the source operand of at least one unexecuted instruction in the window. Only average numbers for SpecInt95 and SpecFP95 are shown. Note that 90% of the time about 4 and 5 registers are enough to hold such values for integer and FP codes respectively. If the processor provided low latency in the access to these operands, performance could not be degraded even if the remaining registers had a much higher latency. In fact, the number of required registers may be even lower, since some of these operands may not be useful to any instruction at that time, since the instructions that consume them are waiting for other operands. The critical values are those that are source operands of an instruction in the window that has all its operands ready. The cumulative distribution of this measure is shown in Figure 3 by means of a dashed line. Note that on average, the number of critical values is less than 4 (resp. less than 3) for 90% of the time in SpecInt95 (resp. SpecFP95).

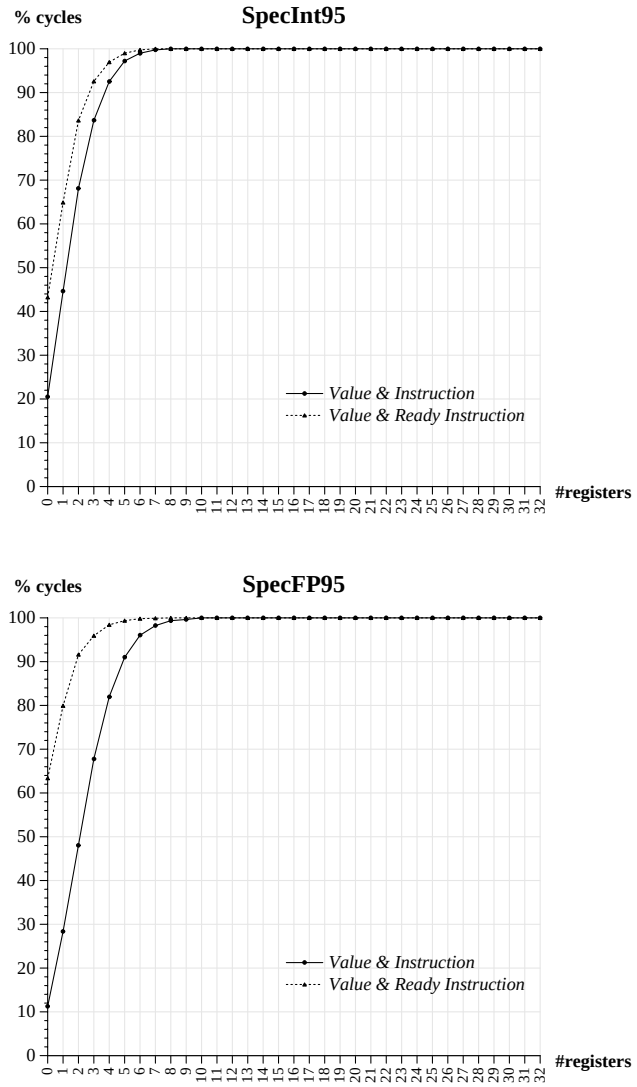
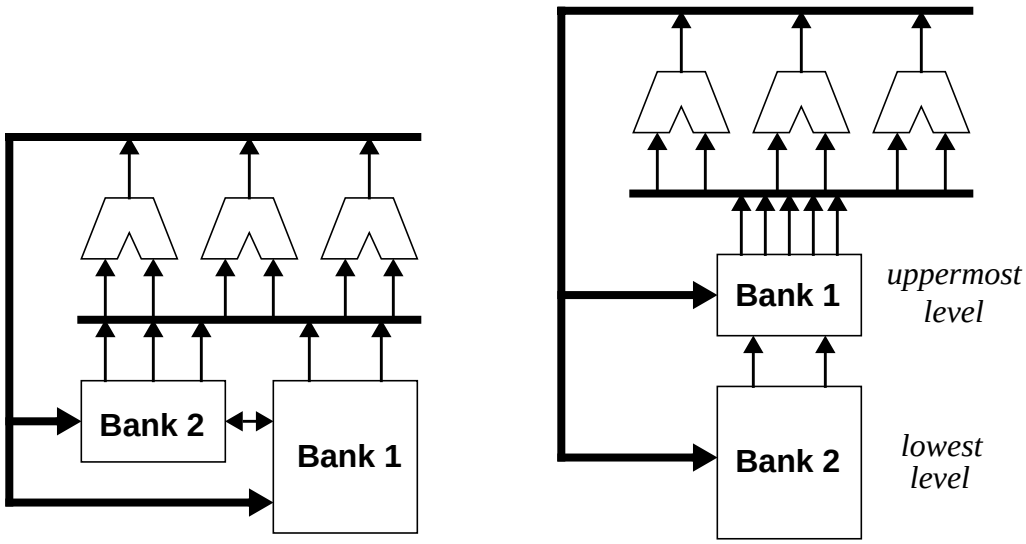


Figure 3: Cumulative distribution of number of registers.

3. A Multiple-Banked Register File

The main conclusion from the previous section is that a processor needs many physical registers but a very small number are actually required from a register file at a given moment. Moreover, register file access time has a significant impact on the performance and complexity of bypass logic. We propose to use a register file with multiple banks to tackle these problems.

A multiple-banked register file architecture consists of several banks of physical registers with a heterogeneous organization: each bank may have a different number of registers, a different number of ports and therefore, a different access time. A multiple-banked register file can have a single-level organization or a multi-level organization, as shown in Figure 4 for the particular case of two banks. In a single-level organization, each logical register is mapped to a physical register in one of the banks. All banks can provide source operands to the functional units, and each result is stored just in one of the banks. In a multi-level organization, only the uppermost level can directly provide source operands to the functional units. A subset of registers in the lower levels are cached in the upper levels depending on the expectations of being required in the near future. Results are always written to the lowest level, which contains all the values, and optionally to upper levels if they are expected to be useful in the near future. Since this multi-level organization has many similarities with a multi-level cache memory organization, we will also refer to it as a



a) one-level **b) multi-level (register file cache)**

Figure 4: Multiple-banked register file architectures.

register file cache. In this paper, we focus on this register file architecture.

A register file cache can have a bank at the upper level that has many ports but few registers, which may result in a single-cycle access time. Banks at the lower levels have many more registers and a somewhat lower number of ports, which may result in an increased latency. However, it requires the same bypass logic as a monolithic register file with one-cycle latency, since source operands are always provided by the uppermost level.

When results are produced they are cached in upper levels, based on heuristics described below. In addition, there is a prefetch mechanism that moves values from lower to upper levels of the hierarchy. Note that data is never moved from upper to lower levels since registers are written only once, and the lowest level is always written.

The approach to deciding which values are cached in the upper level of the hierarchy is a critical issue. Like in cache memories, a criterion based on locality seems appropriate, that is, upper levels should contain those values that are more likely to be accessed in the near future. However, the locality properties of registers and memory are very different. First of all, registers have a much lower temporal re-use. In fact, most physical registers are read only once, and there is even a significant percentage that are never read. Spatial locality is also rare, since physical register allocation and register references are not correlated at all.

We need then different criteria to predict which values are most likely to be accessed in the near future. We propose a caching policy based on the observation that most register values are read at most once. We have observed that this happens for 88% of the values generated by the SpecInt95 and 85% of the FP register values produced by SpecFP95. For a two-level organization, one option is to cache only those results that are not read from the bypass logic. These values will be written in both register banks, whereas bypassed values are written only in the lowest bank. We refer to this policy as *non-bypass* caching.

With *non-bypass* caching, we still cache some values whose first use does not occur for many cycles because they may be source operands of instructions whose other operands take a long time to be produced. In this case, we are wasting a precious space in the uppermost level to store a value that is not needed for many cycles. The second policy we have investigated tackles this problem by caching just those results that are source operands for an instruction that is not yet issued, but which now has all operands ready. In such cases we can be sure the value will be required soon, but will not be available via the bypass logic. We refer to this policy as *ready* caching.

Orthogonal to these two caching policies we have investigated two fetching mechanisms. The first one we call *fetch-on-demand.* In this policy, registers from the lower level are brought down to the upper level whenever an instruction has all its operands ready and some of them are in the lowest level (provided that the bus between both levels is available). This policy is very conservative since it brings to the uppermost levels only those operands that are ready to be used. However, it may delay the execution of some instructions for several cycles, since after identifying a ready instruction with some operands in the lower level, these operands must be read from that level, then written into the upper level, and then read from there to be issued.

A more aggressive fetching mechanism could prefetch the values before they are required. Like in cache memories, prefetching must be carefully implemented to prevent premature or unnecessary fetching from polluting the upper levels. In general, prefetching can be implemented by software or hardware schemes. In this paper we focus on the latter. For cache memories, hardware prefetching is based on predicting the addresses of future references before they are known. For a register file cache, prefetching can exploit knowledge about the instructions in-flight. In fact, the rename and issue logic of a conventional processor can identify all the operand communications between the instructions in the window. Therefore, the processor knows some of the future access to the register file.

We propose the following prefetching scheme that exploits this predictability of register references. Whenever an instruction is issued, it brings to the uppermost level of the hierarchy the other source operand of the first instruction that uses the result of the current one. For instance, in the following code (already renamed):

```
(1) p1 = p2+p3
(2) p4 = p3+p6
(3) p7 = p1+p8
```

when instruction (1) is issued, a prefetch of register *p8* is issued. In this way, part of the latency of bringing *p8* to the uppermost level of the hierarchy can be overlapped with the execution of instruction (1). We refer to this prefetching scheme as *prefetch-first-pair*.

4. Performance Evaluation

4.1. Experimental Framework

The performance of the proposed register file architecture has been evaluated through a cycle-level simulator of a dynamically-scheduled superscalar processor and an analytical model of the area and access time of the register file.

The processor simulator models a 6-stage pipeline (instruction fetch; decode and rename; read operands; execute; write-back; commit). Each stage takes one cycle except for the read and execute stages, which can take several cycles depending on the instruction and the particular architecture. The main features of the microarchitecture are described in Table 1.

Parameter	Value
Fetch width	8 instructions (up to 1 taken branch)
I-cache	64KB, 2-way set-associative, 64 byte lines, 1 cycle hit time, 6 cycle miss time
Branch predictor	Gshare with 64K entries
Instruction window size	128 entries
Functional units (latency in brackets)	6 Simple int (1); 3 int mult/div (2 for mult and 14 for div); 4 simple FP (2); 2 FP div (14); 4 load/store
Load/store queue	64 entries with store-load forwarding
Issue mechanism	8-way out-of-order issue loads may execute when prior store addresses are known
Physical registers	128 int / 128 FP
Dcache	64KB, 2-way set-associative, 64 byte lines, 1 cycle hit time, write-back, 6-cycle miss time if not dirty, 8-cycle miss time if dirty, up to 16 outstanding misses
Commit width	8 instructions

Table 1: Processor microarchitectural parameters

Based on the results presented in Section 2, our experiments use 128 physical registers at the lower level with a 16-register cache

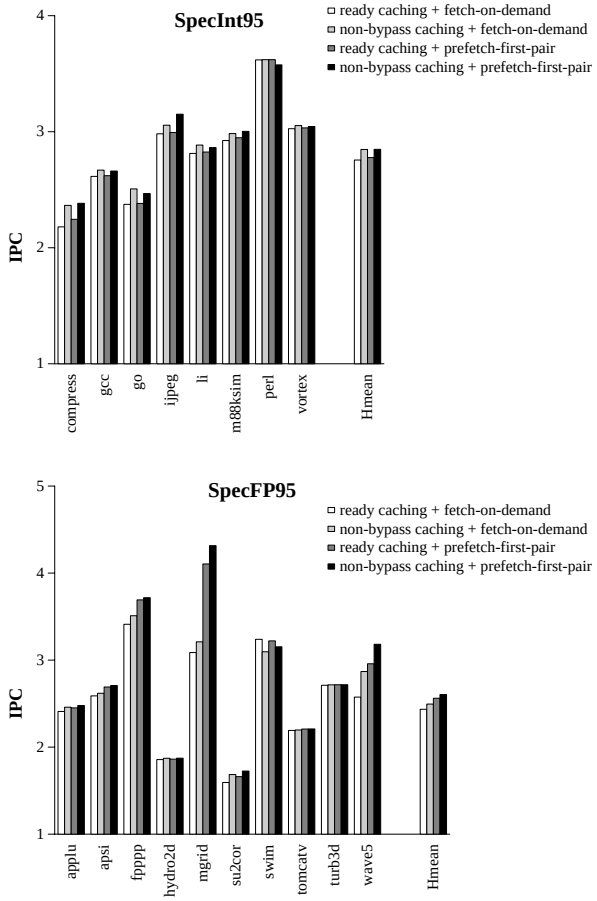


Figure 5: IPC for different register file cache architectures.

at the upper level. The upper level has a fully-associative organization with a pseudo-LRU replacement. The aim of this experimental evaluation is to analyze the effect of register file bandwidth, at each level, on area and performance.

The analytical models of the area and access time are described in [4]. The area model measures the area in λ^2 units and is generic for different technological processes. The access time model is an extension of the CACTI model [16]. The model is configured with the technology parameters corresponding to a process with λ=0.5 μm. This is somewhat old for current designs but is the most aggressive configuration that we have available. However, we have always compared different architectures assuming the same λ, and the performance gains are always reported as speedups relative to a base architecture.

Our experiments used the complete Spec95 benchmark suite. Programs were compiled with the Compaq/Alpha compiler using *-O4* and *-O5* optimization flags for integer and FP codes respectively. The resulting programs were simulated for 100 million instructions, after skipping the initialization part.

4.2. Performance results

We first evaluated the performance of the register file cache for an unlimited number of ports. In fact, the number of ports for maximum performance is bounded by the number of instructions that can simultaneously issue and complete. Figure 5 shows the IPC

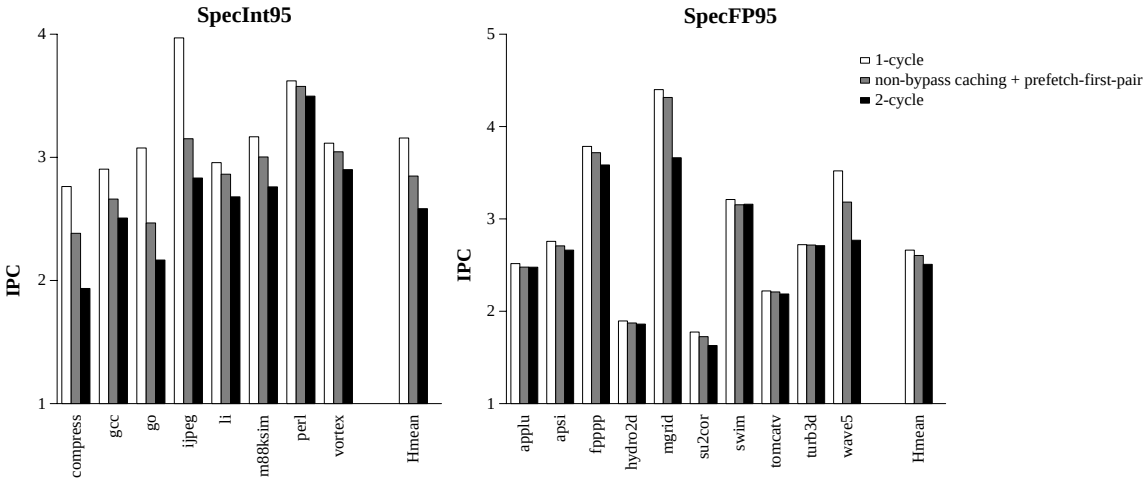


Figure 6: Register file cache versus a single bank with a single level of bypass.

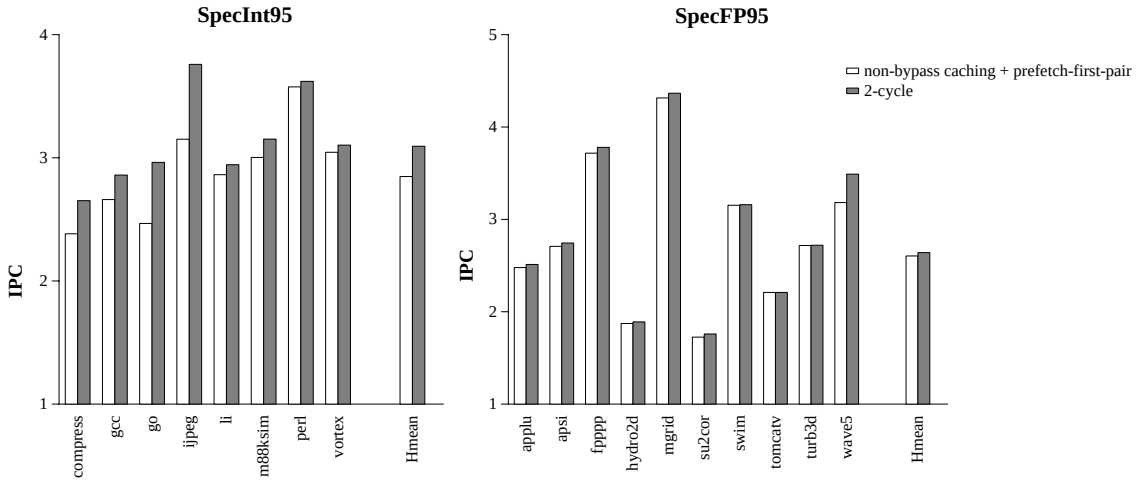


Figure 7: Register file cache versus a single bank with full bypass.

(instructions committed per cycle) for four register file configurations that arise from combining the two caching policies and the two fetch strategies presented in the previous section. Results show that *non-bypass* caching outperforms *ready* caching by 3% and 2% for integer and FP programs respectively. The *non-bypass* policy is also much easier to implement, since identifying which values are not bypassed is straightforward. The second conclusion of this graph is that the proposed prefetching scheme is only effective for a few programs: it provides significant speed-ups for `mgrid`, `fpppp`, and `wave5` and slight improvements for `ijpeg`, `apsi` and `applu`. However, it is important to point out that these figures refer to register files with an unrestricted bandwidth. We will lately show that for a limited number of register file ports the benefits of prefetching are more noticeable.

Figure 6 compares the IPC of the best register file cache configuration (*non-bypass* caching with prefetch-first-pair) with that of a single-banked register file with a single level of bypass and an access time of either 1 or 2 cycles. These three architectures all have the same bypass hardware complexity, but their performance characteristics differ significantly. In general, integer codes are more sensitive to register file latency than the FP codes. For integer codes, the register file cache has 10% higher IPC than the conventional two-cycle register file, on average. For FP codes the average benefit is 4%. Note that the register file cache exhibits significant speed-ups for almost all Spec95 programs. The IPC of the register file cache is still 10% and 2% lower than that of a one-cycle register file for integer and FP codes respectively, which suggests that further research on caching and fetching policies may be worthwhile. However, when the cycle time is factored in, the register file cache outperforms the one-cycle register file as shown below.

Figure 7 compares the IPC of the register file cache with that of a single bank with a two-cycle access time and full bypass. We can observe that the IPC of the register file cache is lower than that of the conventional register file (8% and 2% on average for integer and FP codes respectively). However, the register file cache requires a much simpler bypass network (a single level).

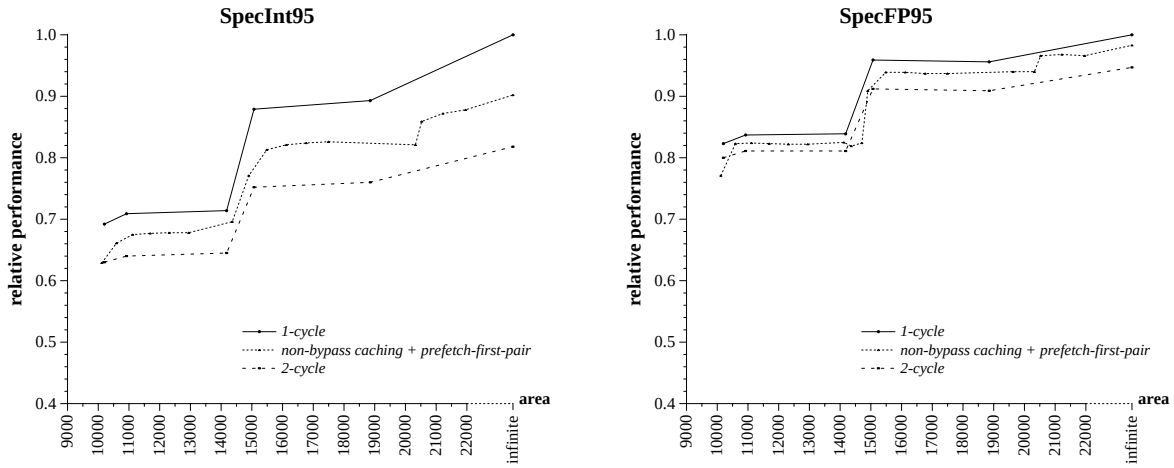


Figure 8: Performance for a varying area cost.

When the number of ports is varied, the different configurations provide different levels of performance as well as different implementation costs. Figure 8 compares the performance of the three register file architectures with a single level of bypass (one-cycle single-banked, two-cycle single-banked, register file cache). Performance is shown as IPC relative to the IPC of the one-cycle single-banked with an unlimited number of ports. For each particular area cost (in 10K λ^2 units) the best configuration in terms of number of read and write ports has been shown. In fact, for each register file architecture we have analyzed all possible combinations of number of read and write ports. Then, we have eliminated those configurations for which there is another configuration of the same register file architecture that has lower area and higher IPC. We can observe that the register file cache offers a significant speed-up over the two-cycle single-banked architecture, especially for integer programs, and its performance is close to that of the one-cycle single-banked architecture, especially for FP programs, for the whole range of area cost.

For some particular programs such as `mgrid`, `hydro2d` and `vortex`, the register file cache outperforms, in some cases, the one-cycle single-banked configuration with the same area. This is because for a given area cost, the register file cache can have a larger number of ports in the uppermost level at the expense of a lower number of ports in the lowest level.

Performance is ultimately determined by execution time. When the number of instructions and the cycle time does not vary, execution time is proportional to 1/IPC. In our experiments, the number of instructions of a given program is fixed but the cycle time depends on the register file access time, which in turn depends on its number of ports. Thus, when alterations to the micro-architecture are critical to cycle time, one must combine predicted cycles with simulated IPC values to obtain a more realistic model of expected performance.

Figure 9 compares the performance of the register file cache with that of a single-banked register file assuming that the register file access time determines the cycle time of the processor. For the single-banked configuration with 2-cycle access time, we have assumed that the processor cycle time is half the register file cycle time (i. e., we suppose that the register file can be pipelined into two stages of equal duration and without any inter-stage overhead, which is somewhat optimistic). We have chosen four different configurations that represent different costs in terms of area, as described in Table 2. For each area cost, the optimal number of ports for each architecture has been chosen. The area of the single-banked architectures and the register file cache are very similar for each configuration, although they are not exactly the same since it is impossible to find configurations with identical area due to their differing architectures. Performance is measured as instruction throughput (instructions committed per time unit) relative to the throughput of a one-cycle single-banked architecture with configuration C1.

We can see in Figure 9 that performance increases as the area rises, up to a certain point where a further increase in area degrades performance, This point is reached when an increase in cycle time is not offset by the boost in IPC provided by a larger number of ports. If we choose the best configuration for each architecture we can see that the speed-up of the register file cache over the single-banked architecture is very high, averaging 87% for SpecInt95 and 92% for SpecFP95. Comparing the register file cache with the two-cycle single-banked architecture, we observe an average speed-up of 9% for SpecInt95 and about the same performance for SpecFP95. However, note that the figures for the two-cycle single-banked architecture are optimistic as commented above.

5. Related work

The organization and management of the register file has been extensively researched in the past. However, there are very few proposals based on a multiple-banked organization for a single cluster architecture.

A two-level register file organization was implemented in the Cray-1 [10]. The Cray-1 processor had 8 first-level and 64 second-level registers for addresses (referred to as A and B registers respectively) and the same organization for scalars

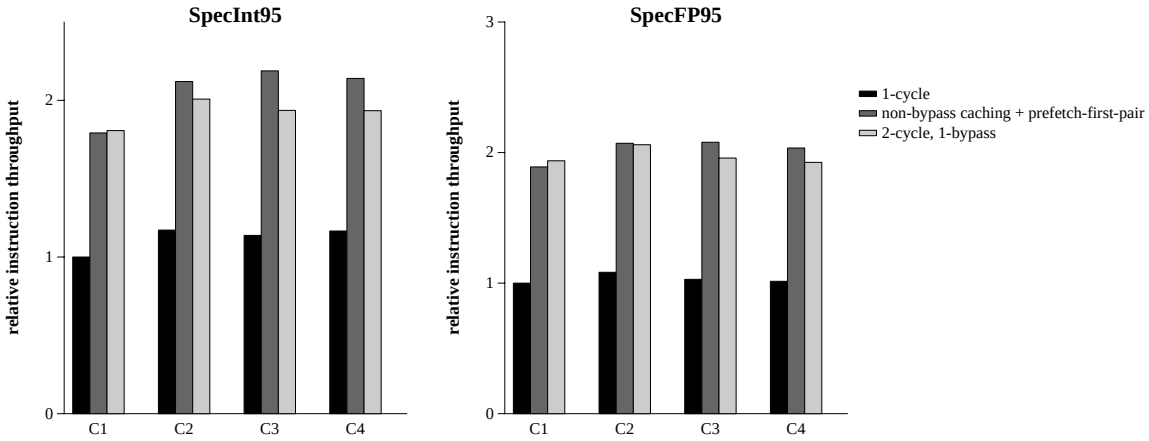


Figure 9: Performance of different register file architectures when the access time is factored in. The different architectures are described in Table 2.

conf.	one-cycle single-banked				two-cycle single-banked				register file cache					
	Area (10K λ^2)	cycle time	R	W	Area (10K λ^2)	cycle time	R	W	Area (10K λ^2)	cycle time	uppermost level		lowest level	B
											R	W	W	
C1	10921	4.71	3	2	10921	2.35	3	2	10593	2.45	3	2	2	2
C2	15070	4.98	3	3	15070	2.49	3	3	15487	2.55	4	3	3	2
C3	18855	5.22	4	3	18855	2.61	4	3	20529	2.61	4	4	4	2
C4	24163	5.48	4	4	24163	2.74	4	4	25296	2.67	4	4	4	3

Table 2: Number of read (R) write (W) ports of each configuration. For the register file cache, number of buses (B) between the two levels are also specified. Each bus implies a read port in the lowest level and an additional write port in the uppermost level.

(referred to as S and T respectively). Data movement between the first and second levels was done completely under software control by means of explicit instructions. Another hierarchical register file that is very similar to the Cray-1 organization was proposed in [11].

A replicated multiple-banked organization is a well-known technique to reduce the number of ports of each bank. This approach is used for instance by the Alpha 21264 microprocessor [3]. The integer unit of this processor has a two-bank register file with a one-level organization (both banks can feed the functional units) and full replication of registers. Each functional unit can be fed only by one of the banks and the results are always written in both, with a one-cycle delay for the non-local bank.

A multiple-banked register file organization was also proposed in [15]. This is a one-level organization with two read and one write ports per bank. Register allocation was done at the end of the execution in order to avoid conflicts in the write ports.

The non-consistent dual-register file [5] is a two-bank organization for VLIW architectures with partial replication of registers. It is a one-level organization with homogeneous banks, in which the allocation of registers to banks was done by the compiler.

The Sack [6] is a one-level two-bank architecture with heterogeneous organization (different number of ports) and no replication, which was proposed again for VLIW architectures. The allocation of registers to banks was done at compile-time.

A two-level register file organization for a windowed register file of an in-order issue processor was proposed in [1]. In this case, the uppermost level stored a small number of register windows, including the active one, whereas the lowest level held the remaining windows. Overflow/underflow in the uppermost

level (due to a call/return respectively) was handled by halting the processor and using a special mechanism to transfer an entire window to/from the lowest level. The idea of caching a subset of the registers in a fast bank was also proposed in [17]. That paper presented some statistics about the hit rate of such a cache assuming that every result was cached under an LRU replacement policy. They also derived some performance implications for an in-order processor based on some simple analytical models.

Several partitioned register file organizations for media processing were proposed in [9]. That work presents a taxonomy of partitioned register file architectures across three axes. Register files can be split along the data-parallel axis resulting in a SIMD organization, or along the instruction-level parallel axis resulting in a distributed register file organization, or along the memory hierarchy axis resulting in a hierarchical organization. They concluded that partitioned register file organizations reduce area, delay, and power dissipation in comparison to the tradicional central register file organization.

This paper is the first work, to the best of our knowledge, that proposes a register file cache for a dynamically scheduled processor and evaluates it with respect to other single-banked architectures with different levels of bypass. It also proposes a prefetching scheme for a register file cache.

6. Conclusions

This paper tackles the problem of the increasing impact on performance of the register file access time. The proposed solution is based on a multiple-banked register file architecture. Among the different multiple-banked organizations outlined in this paper we focus on a two-level organization that we call a register file cache. This organization allows a low latency register access and a single level of bypass, whilst supporting a large number of physical registers.

A multiple-banked register file architecture allows for a heterogeneous organization in which some banks have a higher bandwidth and/or lower latency than others. The effectiveness of the caching and fetching policies is critical to performance. In this paper we have proposed two caching policies and a prefetching scheme.

We have shown that the register file cache outperforms a non-pipelined single-banked architecture with the same bypass complexity by 87% and 92% for SpecInt95 and SpecFP95 respectively.

Different caching and prefetching policies as well as their extension to the one-level organization are currently being investigated.

7. Acknowledgments

This work has been supported by the projects CYCIT TIC98-0511 and ESPRIT 24942, and by the grant PN96-46655316. We would like to thank Jim Smith, Yale Patt, Andy Glew and Eric Sprangle for their comments and suggestions on this work. The research described in this paper has been developed using the resources of the European Center of Parallelism of Barcelona (CEPBA).

8. References

[1] B.K. Bray and M.J. Flynn, “A Two-Level Windowed Register File”, Technical Report CSL-TR-91-499, Stanford University, 1991.

[2] K.I. Farkas, N.P. Jouppi and P. Chow, “Register File Considerations in Dynamically Scheduled Processors”, in *Proc. of Int. Symp. on High-Performance Computer Architecture*, pp. 40-51, 1996.

[3] R.E. Kessler, “The Alpha 21264 Microprocessor”, *IEEE Micro*, 19(2):24-36, March 1999.

[4] J. Llosa and K. Arazabal, “Area and Access Time Models for Multi-Port Register Files and Queue Files”, Technical Report UPC-DAC-1998-35, Universitat Politècnica de Catalunya, *www.ac.upc.es/recerca/reports* (in Spanish), 1998.

[5] J. Llosa, M. Valero and E. Ayguade, “Non-Consistent Dual Register Files to Reduce Register Pressure”, in *Proc. of 1st. Int. Symp. on High-Performance Computer Architecture*, pp. 22-31, 1995.

[6] J. Llosa, M. Valero, J.A.B. Fortes and E. Ayguade, “Using Sacks to Organize Registers in VLIW Machines”, in *Proc. of CONPAR94 - VAPP VI*, pp. 628-639, 1994.

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

[8] A.S. Palacharla, N.P. Jouppi and J.E. Smith, “Complexity-Effective Superscalar Processors”, in *Proc. of Int. Symp. on Computer Architecture*, Edited by M. Hill, N. Jouppi and G. Sohi, pp. 206-218, 1997.

[9] S. Rixner et al., “Register Organization for Media Processing”, in *Proc. of Int. Symp. on High-Performance Computer Architecture*, pp. 375-384, 2000.

[10] R. M. Russell, “The Cray-1 Computer System”, in *Reading in Computer Architecture*, Morgan Kaufmann, pp. 40-49, 2000.

[11] J.A. Swensen and Y.N. Patt, “Hierarchical Registers for Scientific Computers”, in *Proc. of Int. Conf. on Supercomputing*, pp. 346-353, 1988.

[12] D.M. Tullsen et al., “Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor”, in *Proc. of the Int. Symp. on Computer Architecture*, pp. 191-202, 1996.

[13] D.M. Tullsen, S.J. Eggers and H.M. Levy, “Simultaneous Multithreading: Maximizing On-Chip Parallelism”, in *Proc. of the Int. Symp. on Computer Architecture*, pp. 392-403, 1995.

[14] D.W. Wall, “Limits of Instruction-Level Parallelism” Technical Report WRL 93/6 Digital Western Research Laboratory, 1993.

[15] S. Wallace and N. Bagherzadeh, “A Scalable Register File Architecture for Dynamically Scheduled Processors”, in *Proc. 1996 Conf. on Parallel Architectures and Compilation Techniques*, pp. 179-184, 1996.

[16] S.J.E. Wilton and N.P. Jouppi, “An Enhanced Cache Access and Cycle Time Model”, *IEEE Journal of Solid-State Circuits*, 31(5):677-688, May 1996.

[17] R. Yung and N.C. Wilhelm, “Caching Processor General Registers”, in *Proc. Int. Conf. on Circuits Design*, pp. 307-312, 1995.