

An Optimized Front-End Physical Register File with Banking and Writeback Filtering

Miquel Pericàs*, Ruben Gonzalez*, Adrian Cristal*,
Alex Veidenbaum^o and Mateo Valero*

*Technical University of Catalonia, ^oUniversity of California at Irvine
email: {mpericas,gonzalez,adrian,mateo}@ac.upc.edu, alexv@matrix.ics.uci.edu

Abstract

In recent years, processor manufacturers have converged on two types of register file architectures. Both IBM with its POWER series and Intel with its Pentium series are using a central storage for all in-flight values, which offers a high performance potential. AMD, on the other hand, uses an optimized implementation of the Future File for its line of Opteron processors. Both approaches have limitations that may preclude their application in future processor implementations. The centralized register file scales poorly in terms of power-performance. The Future File may be limited by the requirement of distributed reservation stations and by the branch misprediction recovery scheme.

This paper proposes to give processor designer teams another choice by combining a traditional future file architecture with the concept of a central physical register file. This new register file is used in the "front end" in combination with value storage in the instruction queue. Further, it is shown that, contrary to what happens with a centralized R10k-like architecture, the proposed architecture can use banking that scales well with the size of structures in the processor. Only a 0.3% IPC degradation was observed when using this technique. The energy savings due to banking are fully utilized in our proposal. Further, the implementation is shown to integrate well with a technique to early release short lived values that we call *writeback filtering*. Overall, the resulting energy delay product is lower than in previous proposals.

Keywords: *Register File, Reservation Stations, Register Distribution, IPC, Low Power*

1 Introduction

Memory-based structures in modern microprocessor have increasingly large energy requirements. This is especially true if access rates to such structures are high. One such component is the register file.

One approach is to use an architecture based on the future file, which uses an architectural register file and the future register file in the front-end and reservation stations in the back-end. The future file is not very large and is thus quite energy efficient. In case of branch misprediction, old register values need to be recovered from the reorder buffer. Increasing memory latencies can cause this approach to suffer

a large IPC penalty as each branch misprediction requires the mispredicted branch instruction to reach the head of the ROB before recovery can start.

An alternative is to make the whole set of physical registers directly accessible at each point in execution without implementing an architectural register file. This is the approach taken by the MIPS R10000 and many later processors. The difficulty is that this single register file has to be accessed to supply values for computation and for misprediction recovery. It is typically accessed after an instruction is scheduled to execute, even if source operand value(s) were available much earlier. As a result, this file needs to be both large and heavily multiported, which drives the energy consumption high.

Recent work has focused mainly on reducing the number of ports and the number of registers. The approach presented here proposes to increase the total number of registers via replication while heavily reducing the number of ports. Replicating registers can be done in several ways. One option is to implement multiple copies of the register file. This has been used in the Alpha21264 [1] or in the register file of the Netburst microarchitecture [2].

An alternative is to replicate registers depending on their status. With status we refer to several register properties such as whether it has a calculated value or whether it has been renamed.

In an approach based on a future file, we can distribute the registers among several register files using the register status:

- Logical registers, not renamed by later instructions, and with a calculated value can be kept in the front-end¹ (in a future file).
- All other registers that do not have a calculated value

¹In this paper we consider the front-end to include all stages until insertion into the issue queues/reservation stations. The queues are considered to be part of the back-end

must wait in reservation stations or be sourced from the bypass network.

- A third register file is needed to source registers that are necessary in the case of branch misprediction recovery or an exception [3]. We will make use of this idea in our proposal, but the registers will be kept together with the future file registers by assigning physical register numbers to all registers.

The approach proposed here uses a single register file containing all physical registers. Restarting execution after a mispredicted branch can then be done using a map recovery, assuming the processor stores the current rename map in a rename stack on each conditional branch. Thus it performs a faster recovery than the typical future file which requires all previous instructions to commit before the architectural state at the time the branch was decoded has been recovered.

The new approach proposed here starts by taking a future file architecture and replacing the future file by a physical register file. As source operand registers are renamed, it can be determined if a source register has a computed value. The front-end physical register file is only read in such case, significantly reducing the access frequency. Combined with the higher IPC due to faster branch recovery, it improves the energy-delay product compared to the two traditional approaches as will be shown later.

A new structure to hold such "early read" values is created in the instruction queue payload RAM. Its function is similar to that of reservation stations. It is smaller than the physical register file and thus consumes less energy. It is written into by each completing instruction, if the produced value is being waited for by one instruction.

This architecture has some similarities with the PowerPC620 [4]. The main difference is that PPC620 implements the set of physical registers as two separate register files (the logical registers plus the rename buffers) and our baseline allocates both types of registers in the same memory structure.

The contribution of this paper is not only an architecture with a physical register file in the front-end that has a better energy-delay product. The analysis presented also shows that this particular configuration is well suited for the application of power-performance optimizations that imply power-performance trade-offs. This is a result of the register distribution on which the architecture is based. As a result, the best energy-delay product is achieved.

We investigate the use of two optimizations with our front-end physical register file architecture (FPRF). The first is the well known technique of register file banking. We will compare our front-end banking proposal with a recent banking proposal in the back-end [5] where all operands are read after issue. The second is writeback filtering, a technique to eliminate unnecessary writebacks into the register file, conceptually similar to [6]. Writebacks are unnecessary when

the processor can determine that no new instruction entering the CPU will read the value from the front-end register file. While not exactly the same, *Writeback Filtering* is strongly related to *Early Release* techniques such as [7].

2 Related Work

The body of related work on register file energy optimization is large. Many recent papers have proposed mechanisms to reduce the number of the ports by means of modifying the register file architecture, such as [8] [9] [10] [11]. A reduced number of ports may be more efficient both in terms of energy and access time, which can improve performance. The mechanism proposed in this paper is completely orthogonal to these, and can also benefit from a reduced number of ports.

A different approach is to reorganize the registers into several files, concentrating most activity on small files with little power consumption. [12] is an example of this approach based on the isolation of narrow operands. Hierarchical register files, such as those presented in [13] [14] [15] and clustering techniques such as [16] [1] [9] are another example of this technique, which effectively trades size, speed and power consumption.

Another research direction has focused on changing the register allocation algorithm to reduce the register requirements of the architecture. *Early Release* (or *Early Recycling*) frees registers before the commit stage of the next instruction that writes to the same logical registers [7] [17] [18]. *Virtual registers* [19] try to delay the allocation of the physical register until the writeback stage of the instruction. Another approach to reduce registers is to exploit repeated values in the registers [20] [21].

Our approach is somewhat based on the *Future File* organization. The Future File dates back to 1985, when it was proposed by Pleszkun and Smith in their work on precise exceptions [22]. The original proposal only provided operands to instructions via a logical register file in the front-end which receives the name of *Future File*. Modern Future Files have the capability of reading operands from both the Future File and the Architectural Register File [23]. This is specially useful after an exception/misprediction, when a precise instruction state needs to be recovered. The main difference with the Future File is that our architecture is designed around the concept of physical registers to identify the state of the processor. Thus, while a Future File architecture can only recover from a mispredicted branch by draining the ROB, our proposed architecture can recover directly from the physical registers. Future File architectures are still being used in the form of the AMD K7 and K8 microarchitectures [24].

The future file can be extended with rename buffers to provide access to the full processor state at once. This has been implemented in the PPC620 [4] and POWER3 processors.

However, these two processors still require the architectural state to be copied from the rename buffers during retirement. Having an architectural register file in the front-end shortens the pipeline one stage (access can be performed in parallel to rename stage) but increases the number of register transfers in the chip.

A directly related work is the research by Tseng et al. on banked register files [5]. This work discusses in detail an efficient implementation of banking for the register file of a R10000-like architecture. As our proposal implements banking of the register file in the front-end we are interested to see how it compares to a proposal that uses similar techniques for a back-end physical register file.

Finally, the *Writeback Filtering* technique that we evaluate is tightly related to other research in microarchitecture. Freeing registers before the commit of the instruction that renamed the corresponding logical registers is generally known as *Early Release* in literature. This technique was first described by Moudgill et al. [7] and has been used extensively since. Our proposal, based on the release of short-lived values, is similar in concept to other work performed in the context of out-of-order architectures such as [6] and also in the context of VLIW architectures [25]. However, as will be seen there are significant differences between our proposal and these two papers.

3 Front-End Physical Register File

This section describes the *Front-End Physical Register File Architecture (FPRF)*. The FPRF pipeline tries to provide instructions with their operands as soon as they are available. Further, it implements a banked central physical register file in the front-end that allows for fast recovery with limited complexity.

The FPRF Architecture, like a Future File, reads available registers in the front end. However, in our approach the registers are accessed via a mapping into a centralized register file that contains all registers. This has two implications:

1. Access to computed values in the front-end needs to be delayed until the rename stage has completed
2. The number of registers in the front-end, being equal to the total number of registers, is much larger than it is in a Future File Architecture.

For this reason this file is now called the *Front-End Physical Register File*. As will be seen, maintaining this file in the front-end instead of the back-end has several benefits.

Figure 1 shows the full microarchitecture of the FPRF architecture. It can be seen that the instructions, after passing through the decode stage, enter the rename stage where

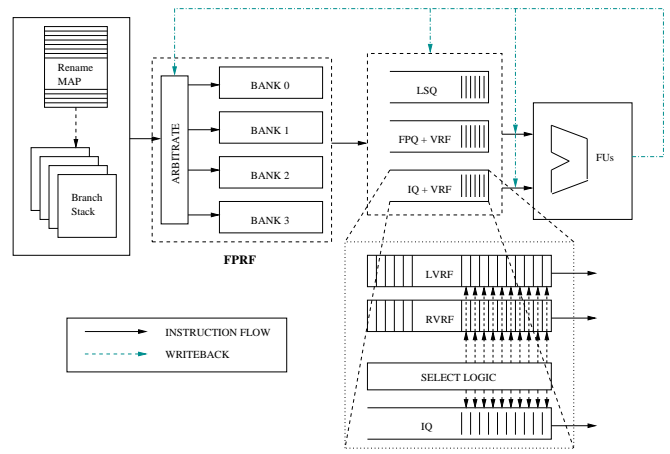


Figure 1: The Front-End Physical Register File Architecture

source and destination registers get renamed. Using this information the instruction accesses the FPRF, a two stage process consisting of arbitration and data access. After the available values have been given to the instruction, it is inserted into the instruction queue.

The back-end pipeline works as follows: When the functional units generate a result, this value is sent to all dependent instructions in the queue. In case there is an instruction waiting, the value is written into the corresponding entry of the Value Register File (VRF) which is part of the payload RAM of the instruction queue. The VRF is driven by the wakeup logic signals and thus works like a register file that does not require a decode stage. The value also gets written into the FPRF, as indicated by its physical register designator. There is also a possibility that a value is bypassed to a dependent instruction.

3.1 FPRF Pipeline

The pipeline of the FPRF Microarchitecture adds one stage to a typical 8-stage pipeline consisting of fetch, decode, rename, queue, issue, execute, operand read, writeback and commit. Due to the FPRF access in the front-end there are two additional stages in this part of the pipeline. In the back-end the operand read stage disappears. This reduces the length to a single additional cycle.

The FPRF access consists of two stages. In the first stage the source registers are analyzed to check for conflicts in the access of the FPRF. Since we are aiming for a banked FPRF, there can be conflicts if too many registers try to access the same bank in the same cycle. Whenever this condition occurs all stages until the FPRF stage halt.

We will also be comparing our banked FPRF against a single-banked configuration that is fully ported and therefore does not require an arbitration stage.

Figure 2 shows the pipeline modeled in this research. The main additions are the stages ARB and FPRF. The ARB stage represents the arbitration stage. During rename, an inquiry is made to know if the source registers that are being requested have computed values. This is implemented via a bit vector with as many entries as logical registers. In the case of the Alpha ISA, which we are modeling, this requires maintaining two 32-bit vectors, one for the integer datapath and another for the floating point datapath. Each entry of this bit-vector indicates whether the corresponding logical register has a computed value or not. In case the computed value is available a read to the corresponding integer or floating point FPRF is started. The ARB arbitration stage logic checks if access to the FPRF can proceed during the next cycle. The front-end, starting from the current instruction, is stalled if some value cannot be read the next cycle.

During the arbitration cycle, priority is given to older instructions to access the operands. This makes sure that the front-end does not dispatch instructions to the instruction queues out-of-order.

The number of ports for each bank has some constraints. The most notable is that there need to be at least two read ports per bank. Some instructions will obtain both operands from the FPRF. It's clear that the processor needs to be able to handle the case in which both these operands are in the same bank. If there were only one read port per bank, the registers would need to be obtained in different cycles. Although this can work, it increases considerably the complexity of the design. In the future we plan to evaluate mechanisms which would allow us to work with single-ported banks in the register file.

Once the arbitration has been done and the arbiter has granted access to the FPRF, in the next cycle the FPRF read occurs.

After the instruction has read the necessary values it is inserted along with them in the instruction queues. This happens during the *Queue* stage. The register values are inserted into a payload RAM associated to the instruction queues. This file is called the *Value Register File* (VRF).

There are no special issues in this stage. However, it's interesting to note that the access rate to the FPRF is lower than a back-end physical register file. Lower access rates mean that less conflicts will occur in the front-end and also that it will have less energy consumption. In particular we observed that the number of integer operands that are obtained from the FPRF is about 40% of the total while for floating point operands this number decreases to around 20%. This has been measured over Spec2000 using the configuration presented in Section 4. Figure 3 and Figure 4 show the exact distribution of integer and fp operand sources averaged over 100 million instructions for each Spec2000 benchmark. In these figures, the box labeled as *FPRF* accounts for those operands that have been read in the front-end from the FPRF. The box labeled as *WRITEBACK* refers to those operands

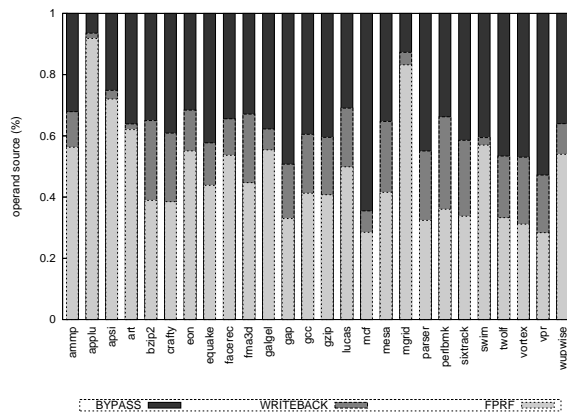


Figure 3: Distribution of Integer Operands

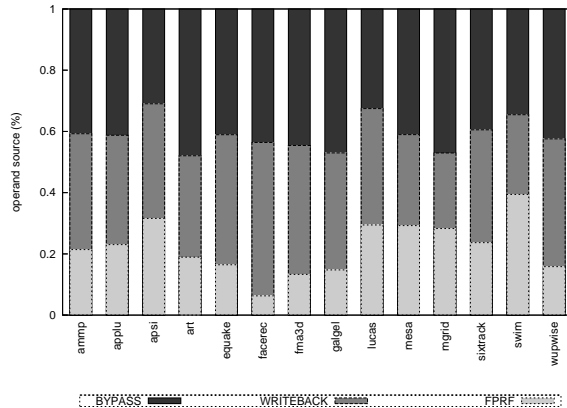


Figure 4: Distribution of FP Operands

that are obtained via writeback into the VRF. Finally, the label *BYPASS* refers to those values that are sourced from the bypass network.

In the event of a branch misprediction our architecture behaves exactly like the R10000. If a prediction turns out to be incorrect, the processor immediately aborts all instructions fetched along the mispredicted path, restores the mapping from the branch stack and starts fetching instructions from the correct path.

3.2 Read Sharing

Accesses to the same logical registers are often clustered during program execution. For example, many instructions have the same logical register for both register sources. On the other side it is also common that the same register is used by several instructions. For example, instruction sequences that manipulate objects on the stack will normally source the stack pointer register. These register accesses cannot be

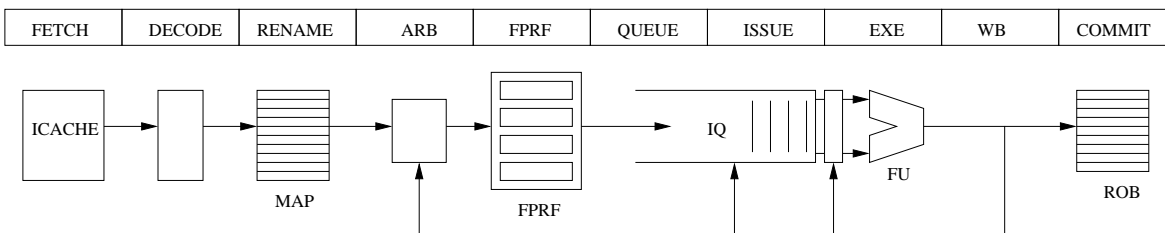


Figure 2: The Pipeline of the FPRF Architecture

distributed among different banks. When they appear there is a high probability that a register access will result in a bank conflict. This suggests that conflicts in the access to the FPRF can probably be effectively reduced by using the technique known as read sharing [14]. Read sharing allows multiple reads of a same register to happen using a single port which is *shared* among the instructions. Previous work on banked register files by Tseng et al. [5] has also used this approach. Read sharing will be evaluated later in the context of the FPRF architecture.

3.3 Writeback Filtering

In the FPRF implementation described so far all values are written back into their physical register in the FPRF during the writeback stage. However recall that a future file architecture is being used. The front-end of such an architecture needs only to maintain copies for those physical registers that may be needed in the future. There are two types of such registers. First, all computed registers belonging to the current architectural state must be in the FPRF. Second, all registers that may be needed in the case of a misspeculation or exception recovery need to be in the FPRF as well. The total number of registers that are generated by writeback is larger than these two numbers combined. Many registers that are renamed twice in a short interval do not appear in any of the current mappings and thus the first result will not be needed in the future. If this can be detected in time, then the writeback can be filtered and the write to the FPRF can be eliminated. To implement this strategy the processor needs to analyze the mapped registers in all rename checkpoints plus the current mapping and decide if the register that is being written back is necessary. Checkpoints need to be taken at all instructions that may cause a replay. There are many such instructions but the vast majority are conditional branches and load operations. Registers that are not referenced anywhere are candidates to be filtered out during writeback.

One very interesting property of the writeback filtering concept is that lazy implementations can be built that are out of the critical path. The information whether an operand needs to be written back or not can be computed right af-

ter the rename stage. However, the operand itself will not be produced until the execution stage has completed, which is at least 5 cycles in the future. In general we can delay the computation of the filter mask a number of cycles equivalent to the distance between rename and writeback. However it has to be noted that delaying this computation will allow many unnecessary writebacks to happen because a physical register that is no longer necessary may appear as mapped in the filter mask even though it does not belong anymore to the current mapping.

This lazy writeback scheme allows the designer to propose slower but pipelined hardware structures to compute the filtering mask. For example, one proposal would be to use a slow multistage OR structure to compute the OR of the several checkpointed rename maps, assuming that a CAM-style renamer is being used. We expect such a structure to be slower, but also less power-hungry compared to precise proposals based on counters such as [7].

In section 5 we will provide performance results on writeback filtering for a scheme that computes the filter mask immediately. We have not evaluated the lazy scheme but expect it to have very similar performance.

We will present some preliminary results in Section 5.

4 Experimental Setup

For the evaluation of the FPRF architecture we used a modified execution driven simulator based on SimpleScalar that models an out-of-order processor. The simulator executes binaries compiled for the Alpha ISA. Our benchmark suite consists of all benchmarks of the Spec2000 suite compiled with Digital *cc* using `"-O2"`. We simulated 100 million of committed instructions.

First a baseline out-of-order microarchitecture was simulated and then we extended it using the enhancements proposed in this paper. Finally, we implemented the model described in [5] for comparison. The common parameters of all configurations are shown in Table 1.

To evaluate our proposal with the read sharing technique we simulated the 6 configurations summarized in Table 2 and described below.

Fetch/Issue/Commit Width	4 instructions/cycle
Branch Predictor	Combined bimodal + 2-level
I-L1 size	32 KB, 4-way, 1 cycle latency
D-L1 size	32 KB, 4-way, 2 rd/wr ports, 1 cycle latency
D-L2 size	256 KB, 4-way, 2 rd/wr ports, 10 cycle latency
Memory Bus Width	32 bytes
Ports to the Register File	8 Read & 4 Write
Reorder Buffer Size	128
Memory latency	100
Integer Physical Registers	160
FP Physical Registers	160
Load/Store Queue	128 entries
Integer Queue	32 entries
FP Queue	32 entries
Integer Functional Units	4 (latency 1)
FP Functional Units	4 (latency 2)

Table 1: Common parameters for all configurations

1. *Base-SHORT* is our optimal baseline configuration. We have used two baselines to benchmark our proposal. Both *Base-SHORT* and *Base-LONG* simulate an out of order configuration with a full-ported centralized physical register file in the back-end. This architecture is based on the MIPS R10000 microarchitecture [26]. *Base-SHORT* simulates an architecture with a three-stage front-end and a six-stage back-end. This configuration has a pipeline that is one stage shorter than our FPRF pipeline. The performance will be better not only because it does not have conflicts in accessing the register file, but also because the branch misprediction penalty is smaller than in our proposal. This is the reason why we introduce *Base-LONG*.
2. *Base-LONG* is an architecture identical to *Base-SHORT*, but with an additional stage in the pipeline. This model is identical to our model in number of cycles paid when a branch misprediction happens. We introduce this model to evaluate how much IPC our proposal loses when the only difference lies in the front-end conflicts and subsequent stalls of our architecture.
3. *FPRF-8B2R2W* is the first configuration based on our proposal. It features an FPRF consisting of 8 banks, 2 read ports and 2 write ports each. We have chosen these numbers to match configurations that Tseng et al. present in their paper on banked register files [5]. This configuration is the base case where no optimizations are implemented. The pipeline depth is equal to *Base-LONG*.
4. *FPRF-8B2R2W-RS* is identical to the *FPRF-8B2R2W* configuration but featuring the read sharing optimization described in Section 3.2. Comparing configurations *Base-LONG*, *FPRF-8B2R2W* and *FPRF-8B2R2W-RS* we will be able to measure how effective the read sharing technique is when applied to our FPRF architecture.
5. *BMRF-OPT* is an optimistic implementation of the banking strategy described in [5]. This proposal needs to speculatively issue instructions to the functional units. If later a conflict occurs when the instruction wants to read its operands, a bubble is inserted in the pipeline while the correct state of the microarchitecture is recovered. We call this model optimistic because it assumes that there is no need to kill the next issue group and that the architecture can recover at once.
6. *BMRF-STALL* is the same implementation as *BMRF-OPT* but in this case we take into account that a bubble is inserted in the pipeline when a conflict occurs in the access to the register file and the following issue group is killed. This configuration approximates the work done in Tseng et al. [5] but, as will be verified later, it is still optimistic because some constraints have been left out of our architecture.

For the evaluation of *writeback filtering* we used the *FPRF-8B2R2W-RS* architecture.

5 Performance Evaluation

In this section we’re going to focus on the two main metrics: IPC and Energy. *Writeback Filtering* will be analyzed later in section 5.3.

Configuration	#Banks	Read Ports per Bank	Write Ports per Bank	Read Sharing	Bubble?	Pipeline Length
Base-SHORT	1	Unlimited	Unlimited	NO	-	9
Base-LONG	1	Unlimited	Unlimited	NO	-	10
FPRF-8B2R2W	8	2	2	NO	-	10
FPRF-8B2R2W-RS	8	2	2	YES	-	10
BMRF-OPT	8	2	2	YES	NO	10
BMRF-STALL	8	2	2	YES	YES	10

Table 2: Main differences between configurations

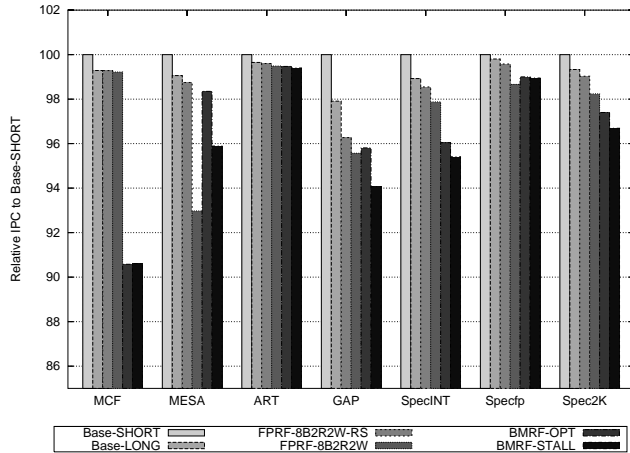


Figure 5: Relative IPCs

5.1 IPC

First we evaluate the amount of instruction level parallelism that our technique is able to exploit. On average, our architecture should be slower than both Base-SHORT and Base-LONG, because these architectures never stall and, in the case of Base-SHORT, the pipeline is shorter which makes branch recovery faster for this architecture.

We simulated the entire SPEC2000 benchmark suite for each of the 6 proposed configurations. For simplicity, only averages and a few selected benchmarks are shown.

Figure 5 shows the results. First, let us focus on the full SPEC average (rightmost column). Our FPRF architecture, with no optimizations, performs only 1.1% worse than Base-LONG, our target architecture. Applying the read sharing optimizations we get as close as 0.3%. Both differences are small. However, as can be seen, the behavior shows a lot of variance across benchmarks. For many applications the improvement is far from incremental.

The BMRF-OPT and BMRF-STALL implementations that model the banking strategy described in [5] perform 1.9% and 2.7% worse than Base-LONG, respectively. This

is because stalls in the back-end have higher impact on performance than stalls in the front-end. These configurations used the read sharing optimization. Compared to our FPRF-8B2R2W-RS implementation, the BMRF implementations lose 1.6% and 2.3% IPC. Our values for the BMRF-STALL implementation are 1.5% better than the values reported by the original paper. This variation is small and reasonable considering the differences between the modeled architectures: different numbers of physical registers and no partitioning between right and left register ports in our model.

Comparing the plots for SpecINT and SpecFP we observe that there is much more IPC loss for integer benchmarks than for floating point programs. There are many reasons for this. For the specific case of Base-SHORT and Base-LONG, the high rate of mispredicted branches in integer applications is the cause for the IPC loss. FP programs have fewer branches and they are more predictable. If we compare the other columns of the plot we see similar behavior. For this case, the reason FP programs have less IPC loss is because many more instructions are in the instruction queues and stalls in the front-end can be easily absorbed by the back-end. The fact that FP programs perform simultaneously FP and integer calculations helps to reduce the conflicts because both types of instructions access different register files.

Finally, we have selected four specific benchmarks (MCF, MESA, ART and GAP) to show how much the performance is dependent on program characteristics. The selected benchmarks show the most extreme variation we observed across all of the SPEC benchmark.

5.2 Energy performance

One of the main benefits of implementing banking is to reduce the power consumption of the register file. The banking technique has long been known as a means of reducing energy, but the complexity of control logic and potential loss in IPC, have precluded its use in high performance microprocessors. So we have proposed a banking scheme for the FPRF taking advantage of the reduced access rate to this structure when it is located in the front-end.

In this section we evaluate the energy requirements of the

register files of our architecture. To perform the energy evaluations we model the register file as proposed by Rixner et al. in [27]. This paper establishes equations to compute the dissipation of register files. We compute energy values by multiplying with the number of accesses to the register file. We present values averaged over all benchmarks of SPEC2000. We will evaluate only the banked architectures here but using two different organizations for the issue queues. The first type of back-end uses a centralized *Value Register File*. This is a heavily multiported structure (4 read ports and 10 write ports are needed). The results for this configuration are shown on the left side of Figure 6 relative to the power consumed by the centralized *Base-SHORT* architecture. The energy is given for both the FPRF and the VRF. The results show that our register distribution and banking techniques effectively reduce the energy consumed. Up to 94% of the Base-SHORT energy can be saved by combining our banked FPRF architecture with read sharing. On the other side, although the VRF has fewer accesses and only 32 entries, its large number of ports increases the energy consumption compared to [5], which lacks a VRF. A detailed study of the VRF is left as future work.

For this work we also modeled an alternative organization based on multiple queues as is done in the POWER4 microprocessor [28]. This microarchitecture uses multiple small issue queues instead of one large centralized queue. Each queue has smaller issue capabilities and less entries, which considerably reduces the number of ports (only 1 read port and 6 write ports, one for every source functional unit, are now required). We evaluated our scheme using 4 issue queues of 8 entries for both integer and FP datapaths. The energy of such a distributed scheme is shown on the right side of Figure 6. Using this approach the energy of the VRF is reduced by 82% and the total energy is now 18% below the power consumed by the BMRF model. Applying this optimization to the BMRF model is without effect as, like the Power4, this model still requires the presence of a centralized register file and cannot be distributed. The new distributed scheme will have some IPC loss, but this will affect all architectures equally. A detailed analysis of this technique is outside the scope of this paper.

5.3 Writeback Filtering

The energy of the FPRF can be reduced even more by using *writeback filtering*. This technique has no effect on IPC, but it can have an impact on the energy because it reduces the number of writes into the register file. We measured the number of writebacks that can be filtered out from Spec2000 and observed that about 22% of all integer writebacks could be filtered out using this technique. The number of floating point writebacks is somewhat smaller, with 18% of all floating point results being short-lived values, candidates for writeback filtering. The energy reduction is expected to be

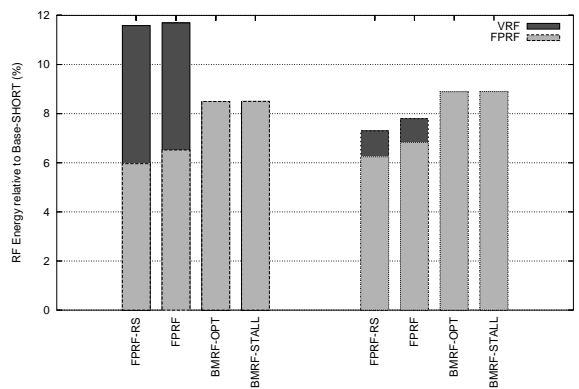


Figure 6: Relative energy with a centralized VRF (left) and a distributed VRF (right)

somewhat smaller as only some writes but no reads are removed. For the full SPEC suite, the energy due to accesses to the integer register file is reduced by 12.1%. For SpecFP the energy due to accesses to the floating point register file is reduced by 13%.

These improvements are not very good. The main reason is the way load misses are handled. The need to replay load misses forces us to add those registers that appear in load rename maps to the rename stack where writeback filtering is analyzed. The number of loads is large and so is the number of additional maps. Not all implementations force loads to replay starting from architectural state. Some processors use a scheme in which instructions that depend on a load are kept in the instruction queues until the load resolves in case they have to be reissued. This allows the processor to recover directly from the issue queues in case of a load scheduling miss. In such a scheme less registers will be written back into the FPRF because many checkpoints will be gone. We evaluated the impact of this alternative implementation on the FPRF. With this new model up to 55% of all integer writebacks and up to 67% of all FP writebacks can be filtered out. These values are much closer to the expected gains. In terms of energy, this means that up to 30% of the energy can be saved in the integer FPRF and up to 47% can be removed from the floating point FPRF. These values may seem very large but it has to be reminded that the FPRF provides only 20-40% of the operands, so the bulk of accesses to this register file are writes. This is what makes the writeback filtering technique so attractive. On the downside, maintaining load-dependent instructions in the queue until the resolution of the load will have a negative impact on IPC because the queues will fill sooner.

The use of *writeback filtering* is interesting because it allows us to attack the energy problem from two perspectives. Positioning the physical register file in the front end allows

to reduce the number of read accesses because many registers don't have computed values at this point. Writeback filtering, on the other side, reduces the number of writeback accesses. Thus this combination of techniques allows simultaneous reduction in both read and write accesses.

6 Conclusions

In this paper we have analyzed the energy properties of a simple architecture based on traditional concepts like the Future File and the centralized physical register file. We have shown that this architecture is well suited for the application of optimizations that involve power-performance trade-offs. The reason behind this is that registers are distributed across the pipeline and less pressure is applied on a single register file.

We applied two optimizations to our architecture: register file banking with read sharing and writeback filtering. We observed minimal IPC loss when using a banked register file, considerably less compared to a previous proposal where the physical register file is in the back-end [5]. Only 0.3% of IPC was lost due to the banking conflicts, down from 1.6% in [5].

We then applied the *Writeback Filtering* technique. We first discussed the different implementation options and analyzed two configurations. The first one used the architectural state to replay loads while the second replayed the loads directly from the instruction queues. We observed that the second configuration is able to remove about 60% of all writebacks, while the first only removed about 20%.

In the future we plan to analyze how our approach can be used to support large processor configurations with many more in-flight instructions.

Acknowledgements

This work has been supported by the Ministry of Science and Technology of Spain under contract TIC-2001-0995-C02-01 and by the CEPBA.

References

- [1] R. Kessler, "The alpha 21264 microprocessor," *IEEE MICRO*, vol. 19, Mar. 1999.
- [2] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel, "The microarchitecture of the penium 4 processor," vol. Q1, 2001.
- [3] R. Gonzalez, A. Cristal, M. Pericàs, A. Veidenbaum, and M. Valero, "Scalable distributed register file," in *Proceedings of the 4th Workshop on Complexity-Effective Design (WCED)*, 2004.
- [4] D. Levitan, T. Thomas, and P. Tu, "The powerpc 620 microprocessor: A high performance superscalar risc microprocessor," in *Proceedings of COMPCON'95*, March 1995, pp. 285–291.
- [5] J. Tseng and K. Asanovic, "Banked multiported register files for high-frequency superscalar microprocessors," in *30th Annual Intl. Symposium on Computer Architecture*, 2003.
- [6] D. Ponomarev, G. Kucuk, O. Ergin, and K. Ghose, "Reducing datapath energy through the isolation of short-leveled operands," in *Proceedings of the 12th Intl. Conference on Parallel Architectures and Compiler Techniques (PACT03)*, 2003.
- [7] M. Moudgill, K. Pingali, and S. Vassiliadis, "Register renaming and dynamic speculation: an alternative approach," in *26th. Intl. Symp. on Microarchitecture (MICRO-26)*, 1993, pp. 202–213.
- [8] V. Zyuban and P. Kogge, "The energy complexity of register files," in *Intl. Symp. on Low Energy Electronics and Design*, 1998, pp. 305–310.
- [9] A. Seznec, E. Toullec, and O. Rochecouste, "Register write specialization register read specialization: a path to complexity-effective wide-issue superscalar processors," in *35th Intl. Symp. on Microarchitecture (MICRO-35)*, 2002, pp. 383–394.
- [10] I. Park, M. D. Powell, and T. Vijaykumar, "Reducing register ports for higher speed and lower energy," in *35th Annual Intl. Symposium on Microarchitecture*, Dec. 2002.
- [11] N. S. Kim and T. Mudge, "Reducing register ports using delayed write-back queues and operand pre-fetch," in *Intl. Conf. of Supercomputing*, June 2003.
- [12] R. Gonzalez, A. Cristal, D. Ortega, A. Veidenbaum, and M. Valero, "A content aware integer register file organisation," in *31th Intl. Symp. on Computer Architecture (ISCA-31)*, 2004.
- [13] J. Cruz, A. González, M. Valero, and N. Topham, "Multiple-banked register file architecture," in *27th Intl. Symp. on Computer Architecture (ISCA-27)*, 2000, pp. 316–325.
- [14] R. Balasubramonian, S. Dwarkas, and D. Albonesi, "Reducing the complexity of the register file in dynamic superscalar processors," in *34th Intl. Symp. on Microarchitecture (MICRO-34)*, 2001.
- [15] J. Zalamea, J. Llosa, E. Ayguadè, and M. Valero, "Two-level hierarchical register file organization for vliw processors," in *33th Intl. Symp. on Microarchitecture (MICRO-33)*, 2000, pp. 137–146.

- [16] S. Palacharla, N. Jouppi, and J. Smith, "Complexity-effective superscalar processors," in *24th Intl. Symp. on Computer Architecture (ISCA-24)*, 1997.
- [17] R. Balasubramonian, S. Dwarkadas, and D. Albonesi, "Dynamically allocating processor resources between nearby and distant ilp," in *28th Intl. Symp. on Computer Architecture (ISCA-28)*, 2001, pp. 26–37.
- [18] J. Martinez, J. Renau, M. Huang, and M. P. and J. Torrellas, "Cherry: Checkpointed early resource recycling in out-of-order microprocessors," in *35th Intl. Symp. on Microarchitecture (MICRO-35)*, 2002, pp. 3–14.
- [19] A. González, J. González, and M. Valero, "Virtual-physical registers," in *4th Intl. Symp. on High-Performance Computer Architecture (HPCA-4)*, 1998.
- [20] M. H. Lipasti, B. Mestan, and E. Gunadi, "Physical register inlining," in *31th Intl. Symp. on Computer Architecture (ISCA-31)*, 2004.
- [21] S. Balakrishnan and G. Sohi, "Exploiting value locality in physical register files," in *36th Intl. Symp. on Microarchitecture (MICRO-36)*, 2003.
- [22] J. E. Smith and A. R. Pleszkun, "Implementation of precise interrupts in pipelined processors," *Proc. Computer Architecture*, pp. 34–44, 1985.
- [23] M. Johnson, *Superscalar Microprocessor Design*. Prentice Hall, 1990.
- [24] C. Keltcher, K. McGrath, A. Ahmed, and P. Conway, "The amd opteron processor for multiprocessor servers," *IEEE MICRO*, vol. 23, pp. 66–76, Mar. 2003.
- [25] M. Sami, D. Sciuto, C. Silvano, V. Zaccaria, and R. Zafalon, "Exploiting data forwarding to reduce the power budget of vliw embedded processors," in *Proceedings of the Conference on Design, automation and test in Europe*, 2001, pp. 252–257.
- [26] K. C. Yeager, "The mips r10000 superscalar microprocessor," *IEEE MICRO*, vol. 16, pp. 28–41, Apr. 1996.
- [27] S. Rixner, W. J. Dally, B. Khailany, P. R. Mattson, U. J. Kapasi, and J. D. Owens, "Register organization for media processing," in *High Performance Computer Architecture (HPCA-6)*, 2000, pp. 375–386.
- [28] J. Tendler, S. Dodson, S. Fields, and B. S. H. Le, "Power4 system microarchitecture," *IBM Journal of Research and Development*, vol. 46, no. 1, 2002.