Compiler Directed Early Register Release

Timothy M. Jones, Michael F.P. O'Boyle
Member of HiPEAC, School of Informatics
University of Edinburgh
t.m.jones@sms.ed.ac.uk, mob@inf.ed.ac.uk

Jaume Abella†‡, Antonio González†‡ and Oğuz Ergin‡
†Dept. Computer Architecture, ‡Intel Labs
Universitat Politècnica de Catalunya
{jaumex.abella, antonio.gonzalez, oguzx.ergin}@intel.com

Abstract

This paper presents a novel compiler directed technique to reduce the register pressure and power of the register file by releasing registers early. The compiler identifies registers that will only be read once and renames them to different logical registers. Upon issuing an instruction with one of these logical registers as a source, the processor knows that there will be no more uses of it and can release the register through checkpointing. This reduces the occupancy of our banked register file, allowing banks to be turned off for power savings.

Our scheme is faster, simpler and requires less hardware than recently proposed techniques. It also maintains precise interrupts and exceptions where many other techniques do not. We reduce register occupancy by 28% in a large register file and gain in performance too; this translates into dynamic and static power saving of 18%. When compared to state-of-the-art approaches for varying register file sizes, our scheme is always faster (higher IPC) and always achieves a greater reduction in register file occupancy.

1. Introduction

The register file in a modern superscalar processor facilitates out-of-order execution by eliminating false (WAR and WAW) dependencies between instructions. However, it is one of the most energy-consuming structures within the processor [1, 2] with a high latency and is a hotspot whose cooling system in future processors will increase non-linearly in cost compared to the amount of heat removed [3].

Previous research has noted that registers are idle for many cycles after their last use, before being placed on the free-list to be assigned to a new instruction [4]. This is because a register cannot be released until the instruction redefining its logical register commits, in order to maintain a precise processor state in the event of an exception, interrupt or branch mis-prediction occurring.

To remove this register idle time, early register releasing has been proposed [4, 5] which puts a register back on the free-list before the commit of its redefining instruction. Although the main focus of previous work has been to increase performance, releasing registers early can also save power. This can be achieved either through the use of a smaller, more power-efficient register file, or by using a banked register file where banks can be turned off for power savings.

However, hardware techniques for early register release all suffer from the fact that, without speculative releases, they must wait for the redefining instruction to enter the reorder buffer so that they can be certain that no more instructions will need to read the value.

This paper proposes compiler directed early register release. Its advantage is that frequently the compiler knows exactly when the last use of a register is and can guarantee that it will not be used again, allowing registers to be released much earlier. Our scheme employs data-flow analysis on pre-linked assembler code to determine the registers that are only read once and renames them to different logical registers. We focus on single-use registers as they occur frequently and their early release can be easily implemented in hardware. The processor, upon dispatching an instruction with one of these logical registers as its source, knows that this will be the last, indeed the only, use of the register and can release it early.

Our early releasing technique also has the advantage that we can release registers after the issue of the last user,

Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)
1089-795X/05 $20.00 © 2005 IEEE

rather than needing to wait for it to commit. This means that the registers released early experience far fewer cycles of idle time after their final read, allowing more efficient use of the available resources in the register file. This leads to increased IPC and much reduced register pressure and static/dynamic power. Furthermore, our technique recovers the precise processor state in the event of an interrupt or exception, a feature lacking in many hardware approaches.

The rest of this paper is structured as follows. Section 2 discusses previously proposed techniques to optimise the register file and other early releasing schemes. Section 3 shows an example of our proposal, then section 4 presents our compiler analysis. Section 5 describes our microarchitecture and the checkpointed register file whilst section 6 describes our early releasing algorithm. Section 7 compares our results with the state-of-the-art and finally section 8 concludes this work.

2. Related Work

The centralised register file [6] has been the target of many previous approaches whose aims have been to reduce its power consumption, delay, or simply make more efficient use of it. Many schemes take advantage of short-lived values to reduce each register's idle time.

Ergin *et al.* introduced the checkpointed register file to implement early register releasing with the ability to release before the redefining instruction was known to be non-speculative [5]. This is done by copying its value into the shadow bitcells of the register where it can be accessed easily if a branch mis-prediction should occur. This register file is the one we use and it is described in further detail in section 5.1. Ergin *et al.* then present two schemes to release short-lived registers early, either at the commit of the defining instruction or the dispatch of the redefiner provided that all consumers have started execution. The main problem with these schemes (as we show in section 7) is that they cannot release registers until the redefining instruction has dispatched, and therefore been renamed. This may be many cycles after the issue of the last consuming instruction, cycles where potential benefits are lost.

Another two techniques to release registers early come from Monreal *et al.* [4]. The first scheme waits for a redefining instruction to become non-speculative before releasing the previous version of its logical register. The second adds a new queue with multiple levels corresponding to the unconfirmed branches in the reorder buffer (ROB). Registers are released when the redefining instruction becomes non-speculative and the last instruction using the physical register has committed. On a branch mis-prediction the relevant levels in the release queue are squashed. The downside of these techniques is that no recovery mechanism is in place to retain values released early. In the event of an exception or interrupt it would be impossible to reconstruct the precise processor state. They also need to add many large structures to the processor so that the status of redefining and last-use instructions can be maintained, increasing the complexity of the pipeline.

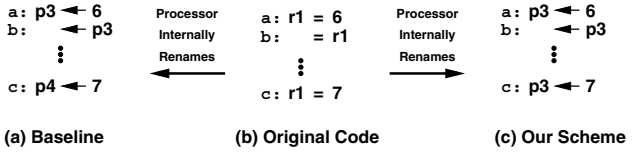


Figure 1. Baseline internally renames the registers, two are used. Our scheme: only one

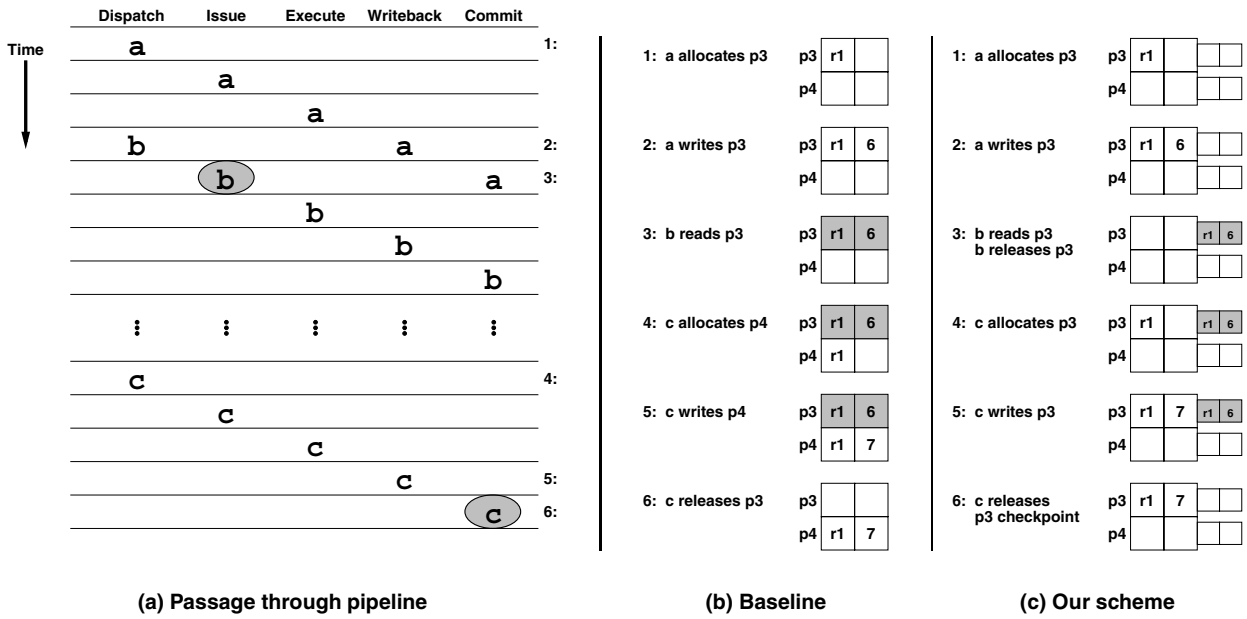
There have been attempts at using the compiler to help with early register releasing by indicating the last use of a register. Lo *et al.* present five schemes for SMT processors that use a mixture of OS and compiler support to relieve register pressure [7]. Unfortunately, these schemes do not allow for precise exceptions or interrupts at all either.

Dead value information (DVI) is introduced my Martin *et al.* [8] which is calculated by the compiler and passed to the processor to help with early register releasing. DVI can be passed via new explicit DVI (E-DVI) instructions which contain a bit-mask indicating the registers released, or implicitly on certain instructions. Procedure calls and returns use implicit DVI (I-DVI) such that when a dynamic call or return is committed the caller-saved registers are released early because the calling conventions implicitly state that they will not be live. The scheme we present in this paper also releases caller-saved registers at procedure boundaries using I-DVI. Once again, with this scheme by Martin *et al.* precise exceptions and interrupts are not maintained.

González *et al.* try to reduce the register idle time that occurs whilst defining instructions are waiting to execute by proposing virtual-physical registers [9]. They allocate a virtual tag to an instruction as its destination at rename and delay the allocation of a physical register until writeback.

Hu and Martonosi propose the Value Ageing Buffer to take advantage of short-lived values [10] whilst Savransky *et al.* consider lazy retirement from the reorder buffer [11]. Other proposals have tried to reduce the number of ports for power reduction [12, 13, 14] or have banked the register file [15, 16].

Other works have used hardware or software techniques to reduce the number of instructions entering the issue queue, indirectly saving power in the register file [17]. There have also been attempts to exploit narrow-width operands [18, 19] or to bank the register file [20, 15] to make better use of resources. The Cherry scheme [21] attempts to recycle all instruction resources early, rather than



(a) Passage through pipeline (b) Baseline (c) Our scheme

Figure 2. Passage of three instructions through pipeline and the state of the register file. Normally p3 is not released until c commits. We do this much earlier when b issues as this is the last use of p3. Other techniques must wait for c to dispatch because this redefines r1.

just the registers used.

However, these techniques often add extra hardware which adds complexity to the processor, whilst not tackling the root cause of register pressure - registers being idle for long periods of time.

3. Motivation

We wish to use early register releasing to reduce the number of physical registers occupied at any one time i.e. register pressure. This allows unused physical registers to be turned off, saving power. This technique can also be used to design processors with smaller register files without affecting performance.

To illustrate how our approach works, consider the example in figure 1 where, to aid readability, we have shown reads and writes to registers using pseudo-code. This example shows a simple assembly code fragment (figure 1(b)) where the value 6 is written into register `r1` in instruction `a`, read in instruction `b` and then a new value 7 is assigned to register `r1` in instruction `c` some time later. In the normal baseline scheme (figure 1(a)) the assembly program register is allocated by the processor at runtime to a hardware physical register, say `p3`, and the value 6 written to it by instruction `a` and read in instruction `b` as before. When `r1` is written to again, the baseline assigns a new unoccupied or unallocated physical register to `r1`, say `p4`, so as to eliminate the anti-dependence from `b` to `c` and the output-dependence from `a` to `c`, the intention being to prevent false storage based dependences slowing the program down. So, if instruction `b` were delayed for any reason, (e.g. waiting for another operand or functional unit contention), the out-of-order superscalar hardware could continue to execute `c` and later dependent instructions, without stalling.

However, if at runtime the read of `r1` occurs before the write in instruction `c`, then there is no need to allocate a new physical register and the same physical register `p3` could be reused, as is the case in our scheme (figure 1(c)) . The register `r1` is marked as being single-use so as soon as it has been read, the physical register it occupies is available for reuse, reducing the number of registers needed and potentially saving power. We apply compiler analysis to the prelinked assembly code to determine which registers are used only once before redefinition. We then rename all registers such that single-use registers have names which the processor knows are reserved for single-use and the physical register associated with it can be immediately released after being used. We focus on single-uses as they frequently occur and their early release is easy to support in hardware.

As we examine an out-of-order pipelined processor, we

consider what happens to the state of the registers as instructions pass through the pipeline as shown in figure 2.

Baseline Case The diagram in figure 2(a) shows the passage of the instructions through the pipeline whilst 2(b) shows the state of the baseline physical register file where we show a five stage pipeline to aid presentation. Physical registers are allocated when a defining instruction dispatches at the beginning of the pipeline and are released at the time of the redefining instruction's commit. So, at point 1, `p3` is allocated to logical register `r1`[1]. At point 2 the value (6) produced by `a` is written into the physical register `p3`. At point 3 it is read for the last time (denoted by the shaded box) as `b` issues. At point 4, possibly much later, `c` dispatches and so register `p4` is allocated. The value (7) generated by `c` is written into the physical register `p4` at point 5. Finally, at point 6, in the baseline case the register holding the previous version of `r1`, (`p3`) is released. The physical register `p3` needlessly retains the value of `r1` from point 3 to 6 in the baseline case.

Our Scheme In our approach (figure 2(c)) the same events happen at points 1 and 2, i.e. register `p3` is allocated to `r1` and then the value (6) is written in at the writeback stage. However, at point 3, once instruction `b` has read `p3`, it is immediately released allowing it to be reused later as it is known that there are no other consumers of this value.

Although, the value will no longer normally be used, rather than discarding it, it is copied into a cheap backup storage using the checkpointed register file described in [5]. So, if there were an exception or mis-speculation between instruction `b` and `c` the value of register `r1` could retrieved from the slow checkpointed store. The cost of recovering from speculation by flushing the pipeline or by handling an exception completely amortises any cost due to recovering the old value of `r1`

Continuing with our example, at point 4, the register is allocated to the now free physical register `p3` which is written to at point 5 with the value 7. At point 6 the old value of `r1` will never be needed as its new value has committed so the checkpoint just needs to be cleared. In reality this does not involve anything more complicated than marking the checkpoint invalid.

Thus we are able to release registers early and guarantee correct behaviour in the case of mis-speculation or exception handling with the support of a small amount of checkpointing. If `p3` and `p4` were in different register banks, `p4` could be gated off for the entire instruction sequence saving static and dynamic power.

Other Schemes Other early releasing schemes would release `p3` later than we propose. Ergin *et al.* [5] release when the redefining instruction (`c` in this example) has entered the pipeline, the original defining instruction has committed and all consumers have read the value. This is at point 4 in our diagram. Monreal *et al.* [4] release when the redefining instruction becomes non-speculative and all consumers have read the value. This would occur somewhere between points 4 and 6 in our diagram.

1 The logical register that each physical register corresponds to is not actually kept in the register file itself, but as a pointer in the reorder buffer - it is merely shown for clarity.

4. Compiler Analysis

This section describes the analysis to identify and rename registers single-use registers that can be released early. Our analysis is based on simple data-flow and liveness analysis with graph colouring. The first stage analyses the assembly program a procedure at a time to determine single-use registers. The next stage determines the number of registers that can be safely used as early release registers throughout the lifetime of the program. This information is passed to the hardware via a single special NOOP at the beginning of the program. Single-use registers are then renamed to use the early release registers wherever possible.

4.1. Single-use Identification

First of all, we construct the control flow and data dependency graphs then rename all registers to virtual registers to distinguish between multiple definitions and uses of the same hard register.

Using the control flow and data dependency graphs and liveness information, it is a simple task to recognise the registers used once, the single-use registers.

To determine whether a register is single-use we consider all its uses in the data dependency graph, their successor nodes in the control flow graph and the registers live into them. Each node in the control flow graph is an instruction. If the register we are considering is live into any successor of one of its uses then it cannot be a single-use register. This is because it will be used again after that use along some control path and hence is used more than once. If none of the successor nodes of its uses have the register in their live-in set then it is only used once.

If we define $uses[R]$ as the set of uses of register R in the data dependency graph and $liveIn[N]$ as the set of variables live immediately before a node N, then R is single-use if the following condition holds:

$$\forall U \in uses[R], \forall S \in succs[U] : R \notin liveIn[S]$$

4.2. Register Allocation

The first task of our register allocator is to recreate the interference graph which is then coloured with registers to

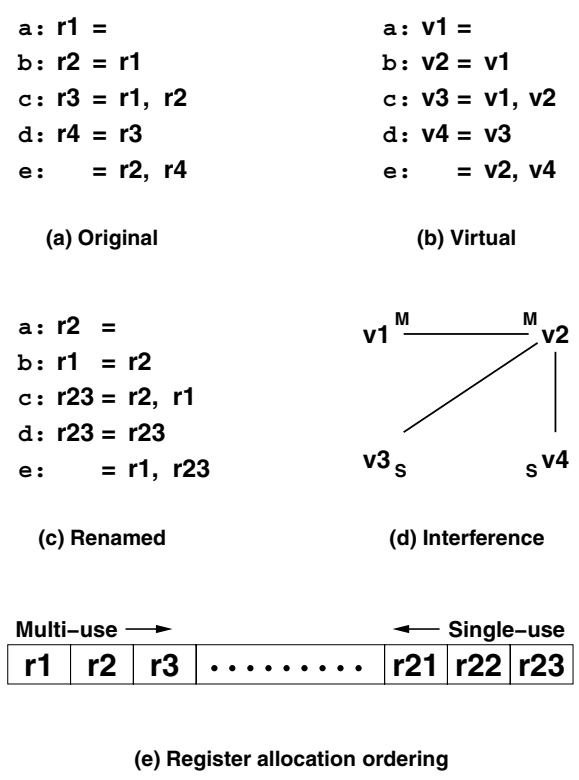


Figure 3. Overview of compiler pass

get our final code. This graph colouring can never introduce spill code because we have the same number of hard registers to allocate to as we did before we renamed them to virtual registers.

We use the standard graph colouring technique described by Appel [22] which creates a set of pairs of nodes (virtual registers) that share an edge in the interference graph. It also records the number of edges each node has, its degree.

The next stage is to determine the order in which nodes will be coloured: nodes with high degrees are first. We use a greedy algorithm to select the node with the highest degree and colouring proceeds from a fixed order of registers. Single-use registers are allocated from one end of the ordering, multi-use from the other. Registers close to the end of the ordering are preferred so as to keep the total number used to a minimum.

These two ordering allocations can, of course, meet and overlap. In this case the register is considered a multi-use register to guarantee correctness. We record the maximum number of multi-use registers ever needed across the entire program. From this, we can safely determine the number of early-release registers. Our experiments show that the entire ordering is never needed for multiple-use values and that there is always room for at least 5 single-use early-release registers for the Spec2000 integer benchmarks and Alpha ISA.

4.3. Example

Figure 3 shows an example of the whole process of single-use register identification and register allocation. The original instructions are shown in figure 3(a) and again after the registers have been renamed to virtual registers in figure 3(b).

After single-use registers have been identified, the register interference graph is constructed as shown in figure 3(d) where the virtual registers are annotated with an S if they are single-use or M otherwise (i.e. they are multi-use). For example `v1` is a multi-use register as it is used in instruction `b` and `c` while `v4` is only used in instruction `e`. As can be seen, register `v2` interferes with all registers so has degree 3. There is no other interference so the others have degree 1. Figure 3(e) shows the ordering that is used to allocate registers. Single-use registers are allocated from the right, whereas multi-use registers are allocated from the left.

Colouring proceeds with the highest degree first, so `v2` is allocated a hard register from the multi-use end. It becomes `r1`. All other nodes have degree 1 so we allocate in lexical order. Virtual register `v1` is multi-use and cannot be `r1` because of the interference with `v2`, so it becomes `r2`. Virtual register `v3` is single-use so gets a hard register from the opposite end of the ordering, `r23`. Likewise, `v4` is single-use and so is also allocated from this end. In this case there is no interference with `v3` so it can also be allocated to `r23`. The final code after allocation of all four registers can be seen in figure 3(c). Only 1 single-use register is needed here (`r23`) and the special inserted NOOP containing the value 1, informs the hardware of this.

4.4. Library Procedures and Register Control

As we do not compile any of the library procedures, all early releasing is stopped on entry to a library routine. This is achieved by inserting the special NOOP instruction with the value 0 turning all single-use registers into multi-use registers.

5. Microarchitecture

Our processor is an out-of-order superscalar with a centralised architectural register file. Decisions on early releasing are taken by the processor depending on the destination logical register number and the state of the register file when the last user issues. This section describes the changes made to each structure within the processor to allow early releasing to proceed correctly.

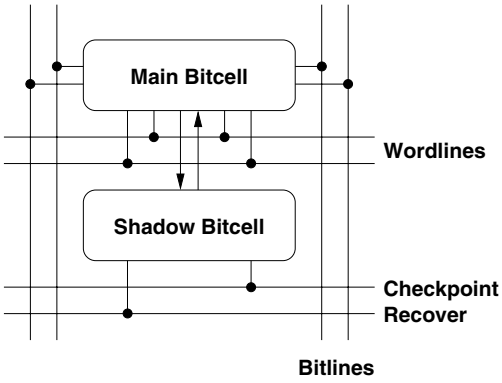


Figure 4. A dual-ported checkpointed bitcell

5.1. Register File

To allow early register releasing and still keep a consistent state in case of exceptions and interrupts, a checkpointed register file [5] is used. Here, a pair of cross-coupled inverters are added to every bitcell which are connected to the main bitcell using pass transistors. Two extra wires are needed to signal a store from the main cell's value into the shadow bits (a *Checkpoint* line) or to copy from the shadow cell into the main (*Recover*). An overview of the whole bitcell can be seen in figure 4 with arrows showing data transfer on a checkpoint and recovery.

The area overhead of the shadow bits is independent of the number of ports. For our register file with 16 read and 8 write ports, the area overhead is 19.4%. The delay overhead is less than 0.5% since no extra gate capacitance is added to the lines [5]. The extra width and height of the checkpointed bitcell increases the wordline and bitline energy consumption, but this affects the energy dissipated in a read or write by only a very small amount. This has been accounted for in our experiments.

In order to keep the additional static power dissipation to a minimum we have employed a super-drowsy circuit for the shadow bitcells [23]. When turned on, the supply voltage arrives through a wide-channel transistor, but when off, a long-channel transistor supplies a much lower supply voltage to preserve the state of the bitcell. With a drowsy voltage of 250mV the leakage energy of the circuit can be reduced by 98%. We apply this technique only to the checkpointed bits where a fast access time is not needed.

Each register needs to keep track of whether there is a valid value held in its shadow bits. We simply add a single bit to each register called the *checkpointed* bit which indicates that the value checkpointed is needed.

A further optimisation we make is to bank the register file using eight registers per bank. When a bank holds no valid data, even in checkpoints, it can be turned off for dynamic and static power savings.

5.2. Reorder Buffer

The reorder buffer keeps track of the instructions between dispatch and commit and so keeps information about the instructions' source registers and their early release status. To allow our technique to work the reorder buffer must store two extra bits per source register. The first is an *early_release* bit which is checked when the instruction issues to determine whether to release early or not. The second is a *did_early_release* bit which indicates whether the source register was released early or not.

5.3. Map Tables

Both the register dispatch map table and the register retirement map table need to keep track the status of the early releasing. The dispatch map table needs to record whether a logical register is allowed to be released early whereas the retirement map table, updated at commit, needs to remember whether the correct value for a logical register is held in the main physical register pointed to, or its shadow cells.

We augment the register dispatch map table with a bit for every entry, called the *early_release* bit, which indicates whether the logical register is allowed to be released early. These can be set through the use of the special NOOP if the number of one-use registers should change, as is the case for foreign code (see section 4.4).

For the register retirement map table we add a *checkpointed* bit to each entry which is used in the event of an interrupt or exception, as explained in section 6.2.

6. Early Releasing

This section describes how early releasing works within the processor using the additions to each structure described in section 5. The main algorithm is presented first, then the way in which branch mis-speculations, interrupts and exceptions are handled. A summary is given in figure 5.

6.1. Normal Execution

When an instruction is dispatched to the issue queue, register renaming takes place as usual. For each source operand, the *early_release* bit is copied from the dispatch map table to the new ROB entry. If the instruction's source registers are the same logical register, only one *early_release* bit should be set in the ROB entry to avoid early releasing twice later on.

After a defining instruction has finished execution its dependents can issue. This involves them reading their source

registers as usual (or obtaining the data through the bypass network). However, if the *early_release* bit of a source register is set in the ROB then the register can be released early.

A register's *checkpointed* bit can be read in parallel to reading the data held in the main part of the register. If this is unset then the register can be checkpointed in the following cycle by copying the value it contains into the shadow bits, putting the register identifier onto the free-list to be used again and setting the *checkpointed* bit. In this case, the *did_early_release* bit in the consumer instruction's ROB entry is set to indicate a successful early release.

When an instruction commits, the previous version of its logical destination register is released. If the *checkpointed* bit in the register retirement map table is set then only the register's *checkpointed* bit needs to be cleared because the register was released early. If unset then the register should be released in the normal way.

An instruction's source registers are also considered when it commits. The *did_early_release* bits in the instruction's ROB entry are copied to the relevant *checkpointed* bit in the register retirement map table to indicate whether the register's value can be found in the main or shadow bits of the physical register.

6.2. Mis-predictions, Interrupts and Exceptions

On a branch mis-prediction some instructions that released registers early may be squashed. By consulting the *did_early_release* bits of instructions being squashed, registers that were released early and checkpointed can be restored so that the correct user can read the right data.

When an interrupt or exception occurs the pipeline is emptied. Before the interrupt or exception handler can be invoked, all logical registers must be represented as non-checkpointed, physical registers in order to maintain a precise processor state. At this time there may be some registers that are checkpointed and of these, some may have other valid values in the main register bits. These need moving so that the checkpointed values can be safely restored.

To deal with this, the processor consults the retirement map table to determine registers that are in the main bitcells blocking checkpointed values. It then issues a `MOV` instruction for each one to place them in different physical registers. The checkpointed values can then be safely restored before executing the interrupt or exception handler. This ensures no registers are checkpointed when control passes over. Although this may take several cycles longer than in the baseline case, the infrequent nature of interrupts and exceptions compared to the savings gained from our technique make this worthwhile.

Dispatch

- Copy the *early_release* bit from the map table to the ROB entry for each source logical register

Issue

- Read the *checkpointed* bit of each source physical register in parallel with reading the data
- If this *checkpointed* bit is unset and the *early_release* bit is set in the ROB entry, the following cycle the register can be checkpointed
- If checkpointing occurs, set the register's *checkpointed* bit and the ROB's *did_early_release* bit

Commit

- Release the previous version of the destination register or remove the checkpoint, depending on the *checkpointed* bit in the retirement map table
- Copy the *did_early_release* bit of each source to the retirement map table's *checkpointed* bit

Figure 5. Main early releasing events

6.3. Impact on ISA

Our technique has no impact on the ISA. The number of early release registers can be fixed on a per-program basis through the use of a special NOOP instruction (which sets the *checkpointed* bits in the dispatch map table) and need not be changed after that, except for calls to foreign code.

7. Results

This section describes the results of our early register releasing scheme in terms of performance, power and register file occupancy. We evaluate how our early releasing scheme affects the performance (in terms of IPC) and power of the register file. We have also implemented two hardware techniques, those of Monreal *et al.* [4] and Ergin *et al.* [5], and the software technique by Martin *et al.* [8]

For the scheme by Monreal *et al.* we chose their extended scheme and called this *Monreal*. Their paper is described in section 2 but, to summarise, this technique releases registers when the redefining instruction becomes non-speculative and the last user commits, whichever is the latter.

From the paper by Ergin *et al.* we chose their combined scheme and called this *Ergin*. This is taken from the paper that proposed the checkpointed register file. The technique works by releasing a register at commit of the original defining instruction or the dispatch of a redefiner, whichever is the latter, providing all consumers have started execution.

Finally, from the paper by Martin *et al.* we chose their scheme which inserts E-DVI instructions before procedure calls to release registers and also releases caller-saved registers upon the commit of each call and return.

We do not have precise details for the power consumed by the three comparison techniques, especially *Monreal* because this introduces a number of large additional struc-

Table 1. Processor configuration

Parameter	*Configuration*
Fetch, decode and commit width	8 instructions
Branch predictor	Hybrid 2K gshare, 2K bimodal 1K selector
BTB	2048 entries, 4-way
L1 Icache	64KB, 2-way, 32B line, 1 cycle hit
L1 Dcache	64KB, 4-way, 32B line, 2 cycles hit
Unified L2 cache	512KB, 8-way, 64B line, 10 cycles hit, 50 cycles miss
ROB size	128 entries
Issue queue	80 entries
Int register file	112 entries (14 banks of 8)
FP register file	112 entries (14 banks of 8)
Int FUs	6 ALU (1 cycle), 3 Mul (3 cycles)
FP FUs	4 ALU (2 cycles), 2 MultDiv (4 cycles mult, 12 cycles div)

tures into the processor. Therefore we compare the register file occupancy of our technique, *Monreal*, *Ergin* and *Martin*, because the lower the occupancy, the more banks can be turned off meaning less power is consumed.

7.1. Compiler, Simulator and Benchmarks

Our compiler analysis was written as a pass in MachineSUIF [24] from Harvard. We used Wattch [25], based on SimpleScalar [26], to implement our processor whose configuration is shown in table 1.

We chose to run the Spec2000 integer benchmark suite, except for *eon* because it is written in C++ which SUIF cannot directly compile. We did not use any of the floating point benchmarks as SUIF cannot compile programs written in Fortran 90 or those with language extensions.

We ran the benchmarks with *ref* inputs for 100 million instructions, skipping the initialisation part and warming the caches and branch predictor for 100 million instructions.

7.2. Evaluation

Figure 6 shows the normalised IPC for each benchmark. On average (denoted by the bar labelled Compiler), there is a small increase in performance of about 0.3%. Some benchmarks experience a very small drop in performance which is due to changes in the branch predictor. Our technique dispatches a slightly greater number of instructions than the baseline (about 0.2%), all of which are along misspeculated branches. Our branch predictor is updated during the writeback stage so it gets slightly polluted by the updates from these extra mis-speculated instructions. However, for most benchmarks there is a very small increase in

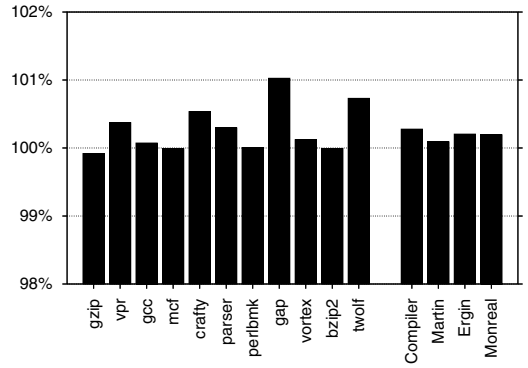


Figure 6. Normalised IPC

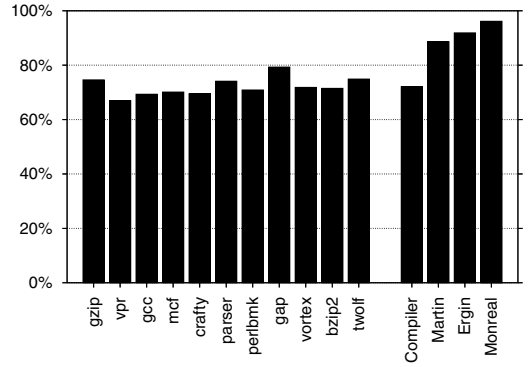
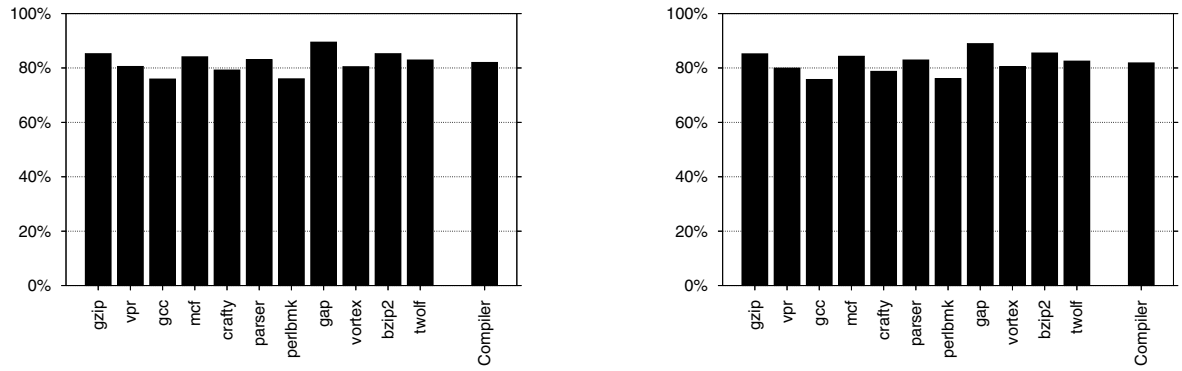


Figure 7. Normalised register file occupancy

performance due to lower register pressure. For comparison, the IPC of *Ergin*, *Monreal* and *Martin* is shown too.

The register file occupancy of our technique is shown in figure 7. There is an average reduction of 28% across all the benchmarks, with almost all achieving a 25% reduction. Again, in this figure we show the register file occupancies for *Ergin*, *Monreal* and *Martin*. The average occupancy reduction for *Ergin* is 8%, for *Monreal* is 4% and for *Martin* is 11% showing our technique performs much better. Through the implementation of a hardware oracle that releases registers immediately after their last use, we find that our technique achieves 80% of the available reduction.

The normalised dynamic and static power of our technique is shown in figures 8(a) and 8(b) respectively. The average dynamic power saving is 18%, although *gcc* and *perlbmk* achieve a 24% saving. There is a similar reduction in the static power of the register file too. Again, the average is an 18% saving with almost all benchmarks achieving a 15% reduction.



(a) Normalised dynamic power

(b) Normalised static power

Figure 8. Normalised dynamic and static register file power for our compiler technique

7.3. Register File Size Sensitivity

Although our technique gives a slight gain in performance, it is really the power savings that it can achieve which make it useful when the register file is large. These savings come about because our technique reduces the average number of registers needed by each program. Put another way, using a smaller register file and our technique we could run each benchmark without losing performance but getting the power savings and decreased access times that come from a smaller register file.

We ran our benchmarks with register files decreasing in size from 112 (as in our original configuration) to 40. In the last configuration, for the baseline, only 8 registers can be used at any time because the others are storing the committed values of the 32 architectural registers. As a comparison, we also simulated early releasing techniques proposed by Monreal *et al.* [4] and Ergin *et al.* [5].

Figure 9(a) shows the IPC for the baseline (called *Baseline*), the technique by Monreal *et al.* (*Monreal*), the technique by Ergin *et al.* (*Ergin*) and our own compiler directed scheme (*Compiler*). As the register file size decreases, so does the IPC, although our scheme is always better than the others. In fact, with a register file size of only 88 entries the IPC of our technique is still better than the baseline with 112 entries. The *Monreal* and *Ergin* schemes do manage to increase the IPC of the baseline, especially for small register file sizes, but our technique is consistently better. For example, when there are only 40 registers available then we increase the IPC from 1.1 to 1.7, an increase of 56%, whereas *Ergin* manages an increase of 20% and *Monreal* only 11%. When there are 64 registers, we increase by 14% (from 1.9 to 2.1), *Ergin* increases by 7% and *Monreal* by 3%.

We could not get any power results for the *Monreal* technique so instead decided to look at the average register file occupancy. This directly affects the amount of static and dynamic power dissipated and is thus a good indicator of how much power would be saved. The average occupancy of each technique for each register file size is shown in figure 9(b). Our technique is able to reduce register pressure, maintain higher IPC and allow greater power savings across all configurations.

8. Conclusions

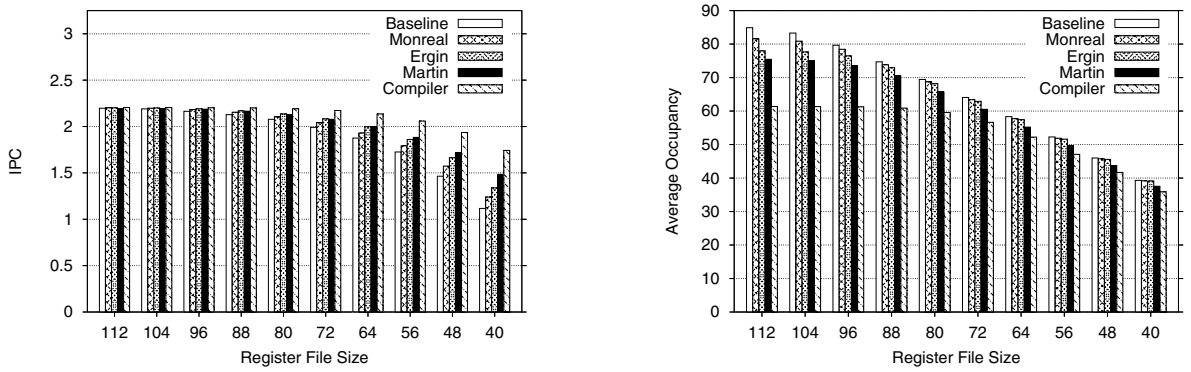
We have presented a novel scheme to dynamically release registers early using the compiler for support. Allowing registers to release early decreases the occupancy of the register file and increases performance. In fact, our scheme with 88 registers is still faster than the baseline with 112.

We bank our registers and turn off a bank when there are no valid registers or checkpointed values held in it, giving both static and dynamic power savings. Results from our experiments show average power savings of 18% with a very slight performance increase.

We have compared our technique to two recently proposed hardware approaches and found that our scheme shows a larger increase in performance and larger decrease in register file occupancy than either of them. In summary, our technique is faster, saves more power and requires a much lower number of registers than state-of-the-art approaches, and relies on less complex hardware. Future work will examine early-release of multi-use registers.

Acknowledgements

This work has been partially supported by The Spanish Ministry of Education and Science under grants TIC2001-0995-C02-01, TIN2004-03072, FEDER funds and Intel Corporation.



(a) IPC **(b) Register file occupancy**

Figure 9. Decreasing the register file size

References

[1] J. Emer. Ev8: The post-ultimate alpha. In *Keynote at PACT*, 2001.

[2] T. C. I. Center. http://bwrc.eecs.berkeley.edu/cic/.

[3] S. H. Gunther, F. Binns, D. M. Carmean, and Jonathan C. Hall. Managing the impact of increasing microprocessor power consumption. *Intel Technology Journal*, Q1, 2001.

[4] Teresa Monreal, Víctor Viñals, Antonio González, and Mateo Valero. Hardware schemes for early register release. In *Proceedings of ICPP*, 2002.

[5] Oguz Ergin, Deniz Balkan, Dmitry Ponomarev, and Kanad Ghose. Increasing processor performance through early register release. In *Proceedings of ICCD-22*, 2004.

[6] Mayan Moudgill, Keshav Pingali, and Stamatis Vassiliadis. Register renaming and dynamic speculation: an alternative approach. In *Proceedings of MICRO-26*, 1993.

[7] Jack L. Lo, Sujay S. Parekh, Susan J. Eggers, Henry M. Levy, and Dean M. Tullsen. Software-directed register deallocation for simultaneous multithreaded processors. *IEEE Transactions on Parallel and Distributed Systems*, 10(9), 1999.

[8] M. M. Martin, A. Roth, and C. N. Fischer. Exploiting dead value information. In *Proceedings of MICRO-30*, 1997.

[9] A. González, J. González, and M. Valero. Virtual-physical registers. In *Proceedings of HPCA-4*, 1998.

[10] Zhigang Hu and Margaret Martonosi. Reducing register file power consumption by exploiting value lifetime. In *Proceedings of WCED in conjunction with ISCA-27*, 2000.

[11] G. Savransky, R. Ronen, and A. González. Lazy retirement: A power aware register management mechanism. In *Proceedings of WCED in conjunction with ISCA-31*, 2004.

[12] Il Park, Michael D. Powell, and T. N. Vijaykumar. Reducing register ports for higher speed and lower energy. In *Proceedings of MICRO-35*, 2002.

[13] Rajeev Balasubramonian, Sandhya Dwarkadas, and David H. Albonesi. Reducing the complexity of the register file in dynamic superscalar processors. In *Proceedings of MICRO-34*, 2001.

[14] Nam Sung Kim and Trevor Mudge. The microarchitecture of a low power register file. In *Proceedings of ISLPED*, 2003.

[15] José-Lorenzo Cruz, Antonio González, Mateo Valero, and Nigel P. Topham. Multiple-banked register file architectures. In *Proceedings of ISCA-27*, 2000.

[16] Jessica H. Tseng and Krste Asanović. Banked multiported register files for high-frequency superscalar microprocessors. In *Proceedings of ISCA-30*, 2003.

[17] Timothy M. Jones, Michael F.P. O'Boyle, Jaume Abella, and Antonio González. Software directed issue queue power reduction. In *Proceedings of HPCA-11*, 2005.

[18] O. Ergin, D. Balkan, K. Ghose, and D. Ponomarev. Register packing: Exploiting narrow-width operands for reducing register file pressure. In *Proceedings of MICRO-37*, 2004.

[19] Mikko H. Lipasti, Brian R. Mestan, and Erika Gunadi. Physical register inlining. In *Proceedings of ISCA-31*, 2004.

[20] Steven Wallace and Nader Bagherzadeh. A scalable register file architecture for dynamically scheduled precessors. In *Proceedings of PACT*, 1996.

[21] José F. Martinez, Jose Renau, Michael C. Huang, Milos Prvulovic, and Josep Torrellas. Cherry: Checkpointed early resource recycling in out-of-order microprocessors. In *Proceedings of MICRO-35*, 2002.

[22] Andrew W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, 2002.

[23] Nam Sung Kim, Krisztián Flautner, David Blaauw, and Trevor Mudge. Single-V_{DD} and single-V_T super-drowsy techniques for low-leakage high-performance instruction caches. In *Proceedings of ISLPED*, 2004.

[24] Machine SUIF. http://www.eecs.harvard.edu/machsuif/-software/software.html.

[25] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of ISCA-27*, 2000.

[26] D. Burger and T. Austin. The simplescalar tool set, version 2.0. Technical Report TR1342, University of Wisconsin-Madison, 1997.