Exploring the Limits of Early Register Release: Exploiting Compiler Analysis

TIMOTHY M. JONES and MICHAEL F. P. O'BOYLE
University of Edinburgh
JAUME ABELLA and ANTONIO GONZÁLEZ
Intel Labs Barcelona—UPC
and
OĞUZ ERGIN
TOBB University of Economics and Technology

12

Register pressure in modern superscalar processors can be reduced by releasing registers early and by copying their contents to cheap back-up storage. This article quantifies the potential benefits of register occupancy reduction and shows that existing hardware-based schemes typically achieve only a small fraction of this potential. This is because they are unable to accurately determine the last use of a register and must wait until the redefining instruction enters the pipeline. On the other hand, compilers have a global view of the program and, using simple dataflow analysis, can determine the last use. This article evaluates the extent to which compiler analysis can aid early releasing, explores the design space, and introduces commit and issue-based early releasing schemes, quantifying their benefits. Using simple compiler analysis and microarchitecture changes, we achieve 70% of the potential register file occupancy reduction. By adding more hardware support, we can increase this to 94%. Our schemes are compared to state-of-the-art approaches for varying register file sizes and are shown to outperform these existing techniques.

Categories and Subject Descriptors: C.1.0 [**Processor Architectures**]: General; C.0 [**Computer Systems Organization**]: General—*Hardware/software interfaces*; D.3.4 [**Programming Languages**]: Processors—*Compilers*

General Terms: Experimentation, Measurement, Performance

Additional Key Words and Phrases: Low-power design, energy efficiency, compiler, microarchitecture, register file

This work has been partially supported by the Royal Academy of Engineering, EPSRC and the Spanish Ministry of Science and Innovation under grant TIN2007-61763 and the Generalitat de Catalunya under grant 2009 SGR 1250.
T. M. Jones, M. F. P. O'Boyle, and O. Ergin are members of HiPEAC (European Network of Exellence on High Performance and Embeded Architecture and Compilation).
Author's address: T. M. Jones, School of Informatics, 1.12 Informatics Forum, 10 Crichton Street, Edinburgh EH8 9AB, UK; email: tjones1@inf.ed.ac.uk.

DOI 10.1145/1582710.1582714 http://doi.acm.org/10.1145/1582710.1582714

ACM Reference Format:
Jones, T. M., O'Boyle, M. F. P., Abella, J., González, A., and Ergin, O. 2009. Exploring the limits of early register release: Exploiting compiler analysis. ACM Trans. Architec. Code Optim. 6, 3, Article 12 (September 2009), 30 pages.
DOI = 10.1145/1582710.1582714 http://doi.acm.org/10.1145/1582710.1582714

1. INTRODUCTION

Modern superscalar processors use register renaming to eliminate false (WAR and WAW) dependences between instructions and expose instruction level parallelism. However, a register file that supports wide issue and a large instruction window has a high latency and complexity due to its size and the number of ports required to read and write data each cycle. It is also a hotspot and one of the most energy-consuming structures within the processor [Emer 2001] whose cooling system cost in future processors will increase nonlinearly compared to the amount of heat removed [Gunther et al. 2001].

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 [Monreal et al. 2002; Butts and Sohi 2004]. 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 or interrupt.

Early register releasing has been proposed [Moudgill et al. 1993; Martin et al. 1997; Monreal et al. 2002; Ergin et al. 2004] to remove this idle time by releasing a register before the commit of its redefining instruction. This allows the register to be reused by a newly dispatching instruction and reduces the overall occupancy of the register file, meaning that a smaller register file can be used without a drop in performance. Hardware early releasing schemes suffer from the fact that, without speculative releasing, they can only release a register early when the redefining instruction enters the pipeline. This is to ensure that no future instructions will need to read the register that is being released. However, there may be many cycles between a register's last use and the dispatch of the redefining instruction, during which time the physical register is not accessed and an opportunity for early releasing is missed.

In this article, we quantify the benefits available using an idealistic oracle with full knowledge of the last use of each register. We find that there is potential for significant reductions in register file occupancy that would directly affect the amount of energy consumed. However, existing schemes are only able to achieve between 14% and 39% of this reduction. The reason for this poor performance is that the hardware, in practice, does not know the last use of any register and has to be conservative in its release policy. We consider the design space of compiler-supported approaches where early-released registers are backed up into shadow register cells, and we show that they are able to provide dramatic improvements. The compiler has global knowledge of the program and can guarantee when a register value will not be used again.

We show that a simple compiler-assisted scheme with minimal hardware support can achieve 70% of the oracle's occupancy reductions. A more aggressive implementation allowing backwards compatibility with existing binaries

achieves 87% of the maximum potential benefit. A scheme with tagged instructions that achieves 94% of the benefit of an oracle is also developed. Finally, we make an evaluation of the best techniques across varying register file sizes and show that they provide large occupancy reductions for large register file sizes and significant IPC gains for small register files, outperforming the best current techniques.

The rest of this article is structured as follows. Section 2 discusses previous work on early releasing and register file optimization techniques. Section 3 motivates the use of early releasing to reduce register pressure, and Section 4 shows the ability of the compiler to obtain the majority of the benefits available. Section 5 describes our compiler analysis, then Section 6 describes the changes to the microarchitecture that are common to all subsequent schemes. Sections 7 and 8 describe commit- and issue-based early releasing approaches, while Section 9 combines the best schemes from each. Section 10 evaluates compiler and hardware techniques across varying register file sizes, and finally, Section 11 summarizes this article.

2. RELATED WORK

There have been many previous approaches that attempt to optimize the centralized register file [Moudgill et al. 1993]. The checkpointed register file was first proposed by Ergin et al. [2004] to aid the implementation of early register releasing. At early release, data is copied into shadow bitcells so that it can be recovered in the case of a branch misprediction, interrupt, or exception. This is the register file that our techniques use. Two schemes are presented by Ergin et al. [2004] to take advantage of early register releasing. However, both must wait until the dispatch of the redefining instruction before releasing any registers and, as this may happen many cycles after the last use of a register, many potential benefits may be lost.

Monreal et al. [2002] also propose two techniques to implement early register releasing. The first scheme waits for a redefining instruction to become nonspeculative before releasing the previous version of its logical register. The second adds a new queue to the processor with multiple levels corresponding to the unconfirmed branches in the reorder buffer. Registers are released when the redefining instruction becomes nonspeculative and the last instruction using the physical register has committed. Unfortunately, however, the addition of the new queue increases the complexity of the pipeline. In addition to this, as no back-up copy is kept of the registers that are released early, precise processor interrupts and exceptions cannot be maintained.

Two compiler schemes have been proposed to help with early register releasing: by Martin et al. [1997] and Lo et al. [1999]. The former uses special compiler-inserted instructions to release registers before procedure calls while the latter targets SMT processors using a mixture of OS and compiler support, including the addition of special instructions or the use of free ISA bits to release registers. In this article, we also present schemes that use special instructions to release registers early and, furthermore, combine them with techniques that use the logical register number to indicate early release information.

In Jones et al. [2005], we present a single compiler-based technique that renames registers that are only used once, releasing them upon their issue and at the commit calls and returns. This current article, however, performs a full exploration of the compiler design space, showing that significant further improvements are available.

The Cherry scheme [Martinez et al. 2002] attempts to recycle all instruction resources early, rather than just the registers. A back-up register file is provided to store a checkpoint of the architectural registers, which can be recovered in the event of an exception. Physical registers are recycled when the architectural register's redefining instruction is nonspeculative and all consumers have executed. When a physical register is released, no back-up copy is made, so execution must restart at the checkpoint if an exception occurs. In contrast, in this article, we present schemes that can release registers even earlier by indicating the last use of each register. This means that we do not even have to wait for the redefining instruction to enter the pipeline before releasing a dead register. Registers that are released early are stored in checkpoint bits within the register file so that they can be quickly recovered on an exception, interrupt, or branch misprediction.

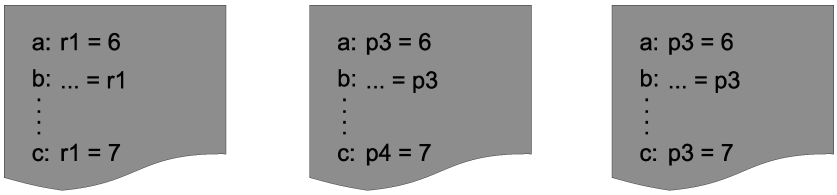
González et al. [1998] attempt to reduce the idle time before a register is written by proposing virtual physical registers. Virtual tags are associated with each destination in the issue queue; the physical register is not allocated until the writeback stage of the pipeline. Short-lived values can be optimized with a value ageing buffer [Hu and Martonosi 2000] and by Savransky et al. [2004] using lazy retirement from the reorder buffer. Narrow-width operands can also be exploited [Ergin et al. 2004; Lipasti et al. 2004] as can register sharing [Tran et al. 2004].

Other approaches create a multilevel register file [Cruz et al. 2000; Borch et al. 2002; Butts and Sohi 2004], bank the register file [Wallace and Bagherzadeh 1996; Tseng and Asanović 2003], or attempt to reduce the number of ports it requires [Balasubramonian et al. 2001; Park et al. 2002; Kim and Mudge 2003]. Some approaches optimize the issue queue, indirectly affecting the register file too [Abella and González 2003; Jones et al. 2005]. These schemes are orthogonal to the techniques we propose and could be used in conjunction with ours to further optimize the register file. Finally, it has been suggested that degree-of-use prediction could be used for early register releasing [Butts 2004].

3. MOTIVATION

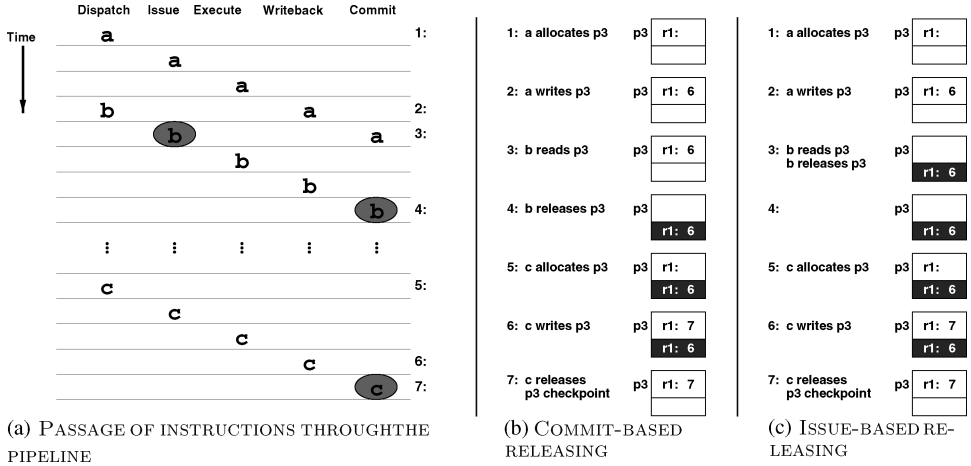
We wish to use early register releasing to reduce the number of physical registers occupied at any one time (i.e., register pressure). This allows unused physical registers to be turned off, saving power. This technique can also be used to design processors with smaller register files without affecting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

To illustrate how our approaches work consider the example in Figure 1, where to aid readability, we have shown reads and writes to registers using pseudocode. Figure 1(a) shows a simple assembly code fragment where instructions *b* and *c* are several instructions apart. In the normal baseline scheme



(a) ORIGINAL CODE (b) BASELINE (c) OUR SCHEME

Fig. 1. The processor internally renames the registers. In the baseline, two are used, whereas in our scheme, we only need one.



(a) PASSAGE OF INSTRUCTIONS THROUGHTHE PIPELINE (b) COMMIT-BASED RELEASING (c) ISSUE-BASED RELEASING

Fig. 2. Passage of three instructions through the pipeline and the state of physical register p3 when early releasing is employed. Normally p3 is not released until c commits. With commit-based releasing, we can release it when its last user, b, commits. In the issue-based releasing schemes, it can be released when b issues. Other hardware approaches must wait for c to dispatch since this redefines r1.

(Figure 1(b)), the assembly program register is allocated by the processor at runtime to a hardware physical register, say *p3*, when instruction *a* dispatches. When *r1* is written to again in *c*, the baseline assigns a new unallocated physical register to *r1*, say *p4*.

However, if at runtime the read of *r1* in instruction *b* 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)). Instruction *b* is marked as the last user of register *r1* so the physical register it occupies is available for reuse once *b* has read it, reducing the number of registers needed and potentially saving power.

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. We consider the baseline and two types of early register releasing, namely issue-based and commit-based.

3.1 Baseline

The diagram in Figure 2(a) shows the passage of the instructions through the pipeline where we show a five-stage pipeline to aid presentation. At point 1, *p3* is allocated to logical register *r1*, and at point 3, it is read for the last time as *b* issues. At point 5, possibly much later, *c* dispatches and so a new physical register is allocated. Finally, at point 7, in the baseline case the register holding the previous version of *r1* (*p3*) is released, shown by the grey oval around *c*.

As is apparent in this baseline case, the physical register *p3* needlessly retains the value of *r1* from points 3 to 7. This is because the processor does not know whether another instruction will need to read *p3* until *c* commits. However, using compiler knowledge that is guaranteed to be correct, it can release much earlier if it knows which instruction is the last user of a register. Physical registers can be released early either at the commit of this instruction, or when it issues. The former is referred to as a commit-based scheme, whereas the latter is known as an issue-based approach. These are illustrated in Figure 2(a) at points 3 and 4 with grey ovals around *b*. The potential benefits to be gained from both commit-based and issue-based early releasing schemes are evaluated in Section 4, but first we give an example of how each works in the following sections.

3.2 Commit-Based Releasing

Figure 2(b) shows the state of the physical register file[1] when we employ commit-based early register releasing. This occurs when a register is released at the commit of its last user instruction. We consider releasing in this way because we can guarantee that all consumer instructions have read the physical register once the last user has committed. We also do not then have to deal with the added complexity of coping with mispredicted branches, since they will only affect subsequent instructions, which are guaranteed not to need to read any registers that have been released early.

The passage of instructions through the pipeline remains exactly as in Figure 2(a) and the same events happen at points 1, 2, and 3. However, at point 4, as *b* commits, it releases register *p3* because we know that *b* is the last user of this register and thus no other instructions will read it. Although the value will no longer normally be used, rather than discarding it, it is copied into cheap back-up storage using the checkpointed register file described in Ergin et al. [2004] for recovery in the event of an exception or interrupt. At point 5, the register is allocated to the now free physical register *p3*. At point 7, the old value of *r1* will never be needed because its new value is committed, so the checkpoint just needs to be cleared.

[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.

3.3 Issue-Based Releasing

Although commit-based releasing has its advantages in terms of complexity, in actual fact, we can release registers even earlier than the commit of the last user instruction. We know that after all users have read a register, its value will no longer be needed and we can safely release the register early. As before, we save a copy of the register in the cheap back-up storage from which we can recover in case of an exception, interrupt, or branch misprediction. In this section, for simplicity, we assume that instruction *b* is the last user of register *r1* and the last dynamic instruction to read the physical register.

Figure 2(c) shows the state of the register file when issue-based early releasing is employed. This is similar to commit-based early register releasing except that the early releasing of registers occurs when an instruction issues rather than when it commits. This occurs at point 3, after *b* has read its source value, when we know that all consumers have read the value.

Thus, with both commit-based and issue-based releasing, we are able to release registers early and guarantee correct behavior in the case of misprediction or exception with the support of a small amount of checkpointing. By recycling physical registers much earlier than usual, register pressure is reduced. For large register files, where the number of physical registers is not a bottleneck to dispatch, unused registers can be turned off for static and dynamic energy savings. When a small register file is employed, the rapid register recycling allows more instructions to dispatch, increasing performance.

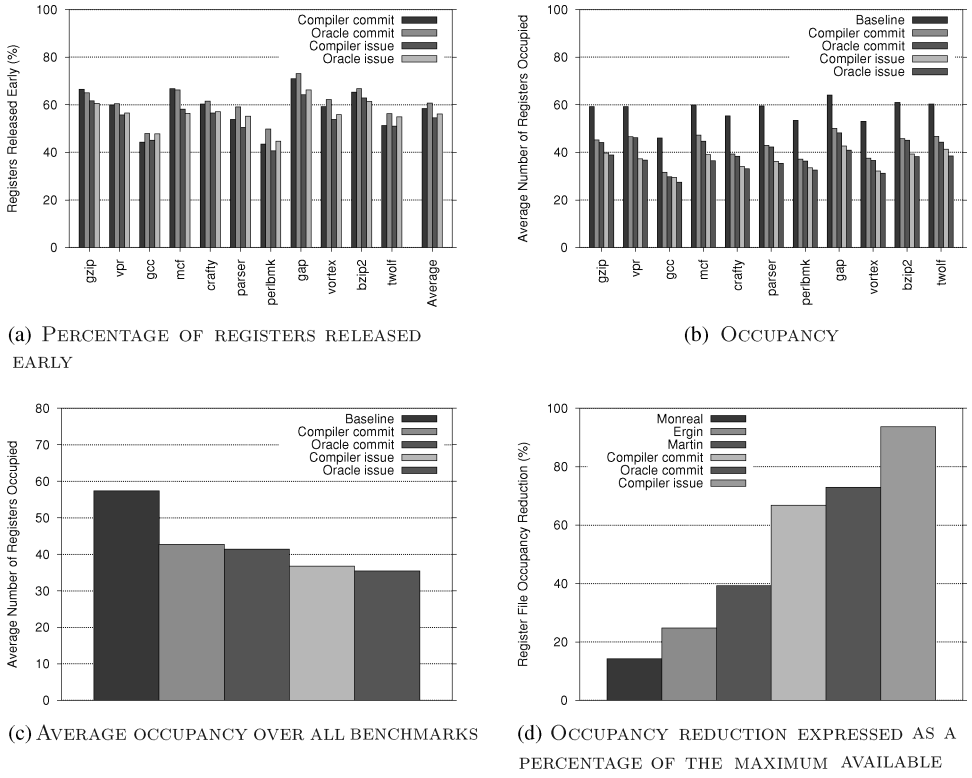
3.4 Other Schemes

Other early releasing schemes would release *p3* later than we propose. Ergin et al. [2004] 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 5 in our diagram. Monreal et al. [2002] release when the redefining instruction becomes nonspeculative and all consumers have read the value. This would occur somewhere between points 5 and 7 in our diagram.

4. QUANTIFYING THE BENEFITS

Having motivated the use of early releasing to reduce register pressure, we now consider the extent to which this can be achieved and the compiler's ability to help.

Consider the situation where the processor has knowledge of the future (i.e., an oracle). Specifically, this oracle knows the instructions that are the final consumers of each register. This is gained by executing each benchmark twice, generating a trace the first time, which can then be consulted by the second to determine the last consumer information. Using this information, the processor can release a register far earlier than usual without having to wait for the dispatch of a redefining instruction. This oracle, although unrealistic in practice, serves as the lower bound on the register file occupancy the processor alone can achieve.



(a) Percentage of registers released early

(b) Occupancy

(c) Average occupancy over all benchmarks

(d) Occupancy reduction expressed as a percentage of the maximum available

Fig. 3. The oracle and best compiler analysis with early register releasing at issue and commit. Figures (b) and (c) show that early releasing can significantly reduce the register file occupancy where an idealistic compiler scheme can achieve almost the same benefits as an oracle.

A second idealistic case considered is where the compiler is able to pass information about the last use of each register to the processor without considering the hardware cost or ISA impact. An instruction is tagged if it is the last consumer of one of its source registers, which allows the processor to release it early. Additionally, at the start of each basic block, tags are placed for each register that is live out of a predecessor but not live into the current block. This is to mark the last consumers of a register after a loop, where they will not normally be released. The compiler analysis considers all control flow paths through a program, including loops, and incorporates interprocedural analysis to mark the last register uses.

Figure 3 shows graphs of registers released early and register file occupancy (based on the architecture described in Section 6) for the oracle and compiler analysis. There is little change in the IPC of the benchmarks when early releasing occurs, hence we do not show this graph. Figure 3(a) shows the percentage of renamed registers that are released early with each scheme. On average, 58% and 61% are released early for the commit-based compiler and hardware schemes, respectively. For the issue-based approaches, it is 55% and 56%, on average. A smaller percentage of the renamed registers are released early with

issue-based schemes compared with commit-based approaches because more registers are renamed in the former case. The actual number of registers released is similar with each technique.

Combined with Figure 3(b) (and Figure 3(c) to show more clearly), releasing registers early can lead to a large reduction in the register file occupancy. More savings can be achieved by the oracle if registers are released in the issue stage of the pipeline (from 57 down to 35 registers), when all consumer instructions have read the data, rather than in the commit stage (when it is reduced to 41). Even though fewer registers are released early at issue, they are actually released earlier in the pipeline, meaning that the benefits last longer. This is because, after execution, instructions can wait many cycles in the reorder buffer before committing.

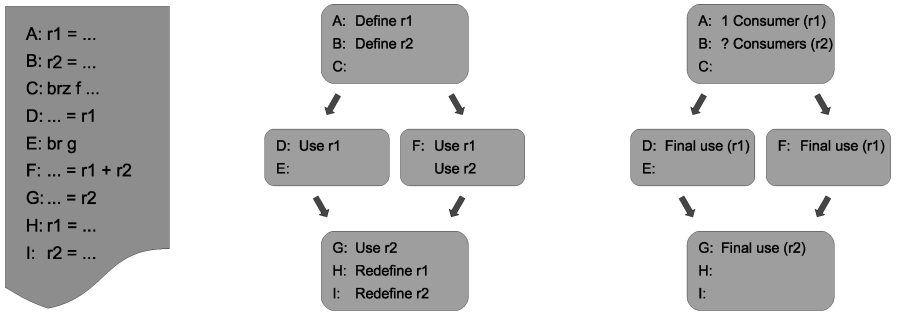
Figure 3(d) shows the average occupancy of the oracle (commit-based), compiler analysis (commit- and issue-based), and three state-of-the-art approaches in terms of the percentage of the total reduction possible (57 down to 35 registers) they have achieved. So, the baseline, by definition, has a 0% reduction (not shown), whereas the oracle with an issue-based releasing scheme achieves 100% of the possible reduction (again, not shown). The existing approaches of Monreal et al. [2002], labeled Monreal in Figure 3(d); Ergin et al. [2004], labeled Ergin; and Martin et al. [1997], labeled Martin, do not significantly reduce the occupancy. Monreal achieves a 14% reduction of the maximum possible (down to 54), Ergin achieves 25% (down to 52), and Martin achieves 39% (down to 49). This compares to the oracle releasing at commit, which achieves a 73% reduction of the maximum available (down to 41).

What is immediately obvious is that the compiler-based oracle scheme achieves almost the same savings as the actual oracle, that is, a 67% reduction for commit-based releasing and a 94% reduction for issue-based releasing. This shows that exploring realistic compiler-based implementations of early releasing is potentially worthwhile. The remainder of this article attempts to determine realistic compiler-based techniques, exploring the trade-off between microarchitecture modification and register file occupancy reduction.

5. COMPILER ANALYSIS

Having quantified the benefits of early register releasing, this section gives a brief overview of the compiler analysis used in this article to determine the last use of a register and the number of consumers it has. It also discusses the limits to the information the compiler can discover through purely static analysis.

Our compiler analysis for all schemes in this article is based on simple dataflow and liveness analysis. For each function in the program, we construct the control flow graph (CFG) and calculate liveness information using the sets of registers that are written and read by each instruction. We then iterate over the CFG, marking the last consumers of each register based on the liveness information. The last consumer of a register is identified by having the register in its $live_{in}$ set and only in its $live_{out}$ set if it defines the register. While iterating



(a) Pseudocode (b) Control flow graph (c) Register uses and last consumers

Fig. 4. An example of the compiler analysis used to determine a register's last consumer. In (a), we show some pseudocode using r1 and r2. Here, c is a conditional branch and e is an unconditional branch. In (b), we show the control flow graph and instructions that r1 and r2 are defined and used. Traversing the control flow graph postorder, we can easily determine the final uses of each register in (c). We know there is exactly 1 consumer of r1 but, depending on the path taken, there could be 1 or 2 consumers of r2.

over the CFG, we record the number of uses of each register between the definer and last consumer.

Figure 4 shows an example of this analysis. In Figure 4(a), we show some pseudocode that defines and uses *r1* and *r2*. Here, instruction *c* is a conditional branch and instruction *e* is unconditional. Figure 4(b) shows the CFG for this code. Here, we have annotated each instruction that defines or uses *r1* or *r2*. Using the liveness analysis and counting consumers along all paths in the CFG leads to Figure 4(c). It is easy to identify the two final uses of *r1* in the separate paths of the CFG, and the single final use of *r2* in instruction *g*. It is also clear to see that *r1* has only one consumer, no matter which path is taken.

Figure 4 also shows a limitation of a purely static approach. The number of uses of *r2* cannot be determined exactly. If the left-hand CFG path is taken, then there is 1 use; if the right-hand path is taken, then there are 2. In this situation, the compiler simply marks the number of uses as "unknown" and an opportunity for optimization could be missed. However, in practice, situations such as this rarely occur. Any scheme that could capture these corner cases would simply improve the results we obtain through using compiler analysis.

The final consumer and number of uses of each register are used to release registers early. The way in which this information is used varies between schemes, and more detail is given with each technique described in Sections 7 and 8.

6. MICROARCHITECTURE

Having discussed the compiler analysis performed for early register releasing, in this Section, we briefly describe the hardware configuration used throughout this article.

Table I. Processor Configuration

Parameter	Configuration	Parameter	Configuration
Machine width	4 instructions	ROB size	96 entries
Branch predictor	16K gshare	LSQ size	48 entries
BTB	2,048 entries, 4-way	Issue queue	32 entries
L1 Icache	32KB 8-way 64B line 1 cycle hit	Int register file	96 entries (12 banks of 8)
		FP register file	96 entries (12 banks of 8)
L1 Dcache	32KB 8-way 64B line 3 cycle hit	Int FUs	3 ALU (1 cycle), 1 Mul (3 cycles)
Unified L2 cache	2MB 8-way 64B line, 14 cycles hit, 250 miss	FP FUs	2 ALU (2 cycles), 1 MultDiv (4/12 cycles)

Our processor is an out-of-order superscalar with a centralized physical register file, as described in Table I. The register file used is described in Sections 6.1 and 6.2 and the details on early releasing in Section 6.3. Section 6.4 discusses interrupts and exceptions, and Section 6.5 describes our experimental set-up.

6.1 Checkpointed Register File

The checkpointed register file was proposed by Ergin et al. [2004] to aid their early releasing scheme. It can hold a copy of each register that is released early in cheap back-up storage that can be recovered easily and quickly. This enables the processor to maintain a consistent state and thus implement precise interrupts and exceptions and recover from branch mispredictions.

To implement back-up storage, an extra bitcell (called the shadow cell) is connected to the main bitcell. Two extra wires are needed to signal a store from the main cell's value into the shadow bit (a *Checkpoint* line) or to copy from the shadow cell back into the main bitcell (*Recover*).

The area overhead of the shadow bitcells is independent of the number of ports. For the register file used in this article with 8 read and 4 write ports, the area overhead is less than 20% [Ergin et al. 2004]. The delay overhead is less than 0.5%, since no extra gate capacitance is added to the lines [Ergin et al. 2004]. 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 small amount. There is also a small amount of energy consumed when checkpointing and recovering data from the shadow cells. This is further described in Section 6.5 and all additional energy consumption is accounted for in the experiments performed.

In order to keep the additional static power dissipation to a minimum, a super-drowsy circuit is employed for the shadow bitcells [Kim et al. 2004]. When turned on, the supply voltage arrives through a wide-channel transistor, but when turned off, a long-channel transistor supplies a 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% [Kim et al. 2004]. This technique is only applied to the checkpointed bits where a fast access time is not needed, and in later sections, we also report the results without using this scheme.

All our compiler schemes use the checkpointed register file, as does the Ergin technique implemented in Section 10. The baseline (in all experiments),

Monreal and Martin, schemes use a register file without checkpointing support.

6.2 Register File Banking

A further optimization is to bank the register file using eight registers per bank, as performed by Abella and González [2003]. When a bank holds no valid data, even in the shadow cells, it can be turned off independently of all the others to save energy. We assume that the baseline scheme uses a noncheckpointed, banked register file and keeps all banks turned on permanently.

Turning register file banks off and on incurs additional performance and energy overheads. We assume that each bank will take 1 cycle to turn off and another to turn on. During these extra cycles, we conservatively assume the bank is consuming full dynamic and static energy. We turn banks off the cycle after all data becomes invalid and back on again as soon as a register is renamed to be a destination of a dispatching instruction [Abella and González 2003]. As there are several cycles between dispatch and writeback of an instruction, the time taken for the bank to turn on again is hidden and thus does not impact on processor performance. However, all energy and cycle overheads are modelled in detail in our experiments.

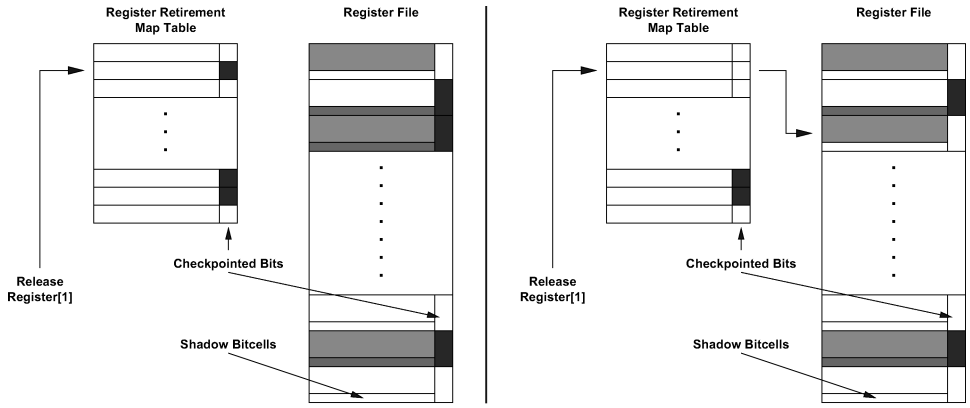
A secondary benefit of banking the register file is that it has a faster access time. On a read, bank selection is performed in parallel with the decoding and reading from each bank. The output logic selects the correct entry from all banks. On a write, bank selection is overlapped with wordline decoding, with the write being performed only in the selected bank. Only banks that are turned on are accessed during a read or write operation. Hence, turning banks off saves not just static energy, but also dynamic energy [Abella and González 2003]. In Section 9, we evaluate the energy savings from our approaches and show the contribution available through turning off banks alone.

6.3 Early Releasing

Each register needs to keep track of whether the value held in its shadow bitcells is valid. A single bit is added to each register, called the checkpointed bit, which indicates that this value is needed.

The register retirement map table needs to keep track of each logical register, whether it is held in the main or shadow bitcells of the physical register pointed to. A checkpointed bit is added to the register retirement map table (one per logical register) which, when set, indicates the required value can be found in the shadow bitcells of its physical register.

When an instruction commits, it releases the previous version of its logical destination register. The register retirement map table is consulted and the relevant checkpointed bit read. If it indicates the actual register is held in shadow bitcells, then the only tasks that need performing are to clear the checkpointed bits in the map table and the physical register pointed to. If the checkpointed bit shows that the actual register is kept in a main bitcells, then the previous version of the logical register is released in the normal way.



(a) Before commit (b) After commit

Fig. 5. An example of the release of a register when the instruction redefining its logical register commits. The checkpointed bit in the register retirement map table is consulted in (a), and we find that the logical register is kept in the shadow bitcells in the register file. In (b), we clear the checkpointed bits in the map table and the register file, which invalidates the value in the register's shadow bitcells.

Figure 5 shows an example of releasing logical Register 1. In Figure 5(a), the checkpointed bit for this register in the retirement map table is consulted. This is then cleared along with the checkpointed bit in the register file for the physical register in Figure 5(b). Clearing these bits invalidates the register's shadow bitcells.

The total overhead of using the checkpointed register file for early releasing is 32 bits in the register retirement map table (one per logical register) and 96 bits in the integer register file (one per physical register). Certain schemes presented in subsequent sections require extra bits to the reorder buffer and map tables, which are described in detail in the text.

6.4 Interrupts and Exceptions

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 noncheckpointed 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 their main register cells. These need to be moved so that the checkpointed values can be safely restored without overwriting these values.

To achieve this, the processor consults the retirement map table to determine registers that are blocking checkpointed values in the main bitcells. It then issues a `MOV` instruction for each one, placing them in different physical registers. It is guaranteed that there will always be a free physical register to move the checkpointed values into because there are more physical registers than logical. After this has occurred, the checkpointed values can be safely restored before executing the interrupt or exception handler. This scheme ensures no registers are held in shadow bitcells 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 these schemes make this worthwhile.

6.5 Compiler, Simulator, and Benchmarks

Our compiler analysis was written as a pass in MachineSUIF [Smith and Holloway 2000] version 2.02.07.15 from Harvard. We used Wattch [Brooks et al. 2000] version 1.02, based on SimpleScalar [Burger and Austin 1997] version 3.0d, to implement our processor whose configuration is shown in Table I. In addition to this, we added support for checkpoints and banking to the register file and the ability to release registers early, as described in Sections 6.1, 6.2, and 6.3.

We derived our dynamic and static register file energy values from Wattch and Cacti [Tarjan et al. 2006] version 4.1 for a 70nm technology. For the baseline (in all experiments) and the Monreal and Martin schemes (in Section 10), we modeled the banked register file described in Section 6.2. For all of our compiler-directed approaches and the Ergin scheme from Section 10, we modeled a banked register file but, in addition, increased the RAM cell sizes to account for the shadow bitcells in the checkpointed register file. The addition of these shadow bitcells increases the register file's static energy consumption (because there are more transistors) and dynamic energy consumption per access (because of the increased area and thus longer wires). An access to the shadow bitcells either through the Checkpoint or Recover lines (see Section 6.1) incurs the dynamic energy needed to access the correct register bank and read or write the relevant shadow bitcells.

For our benchmarks, we chose the Spec2000 integer 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 written in Fortran 77 with language extensions. We ran all benchmarks with the *ref* inputs for 100 million instructions after skipping the initialization part and warming the caches and branch predictor for 100 million instructions. We determined the initialization part of each benchmark by instrumenting the source code at the point where the main algorithm begins, starting functional warming after the instrumented instruction.

7. COMMIT-BASED RELEASING

This section describes and evaluates commit-based early releasing techniques. These schemes release registers after the commit of the instructions performing the release. These approaches are attractive as the microarchitecture requirements are relatively modest. Four schemes are evaluated with varying impact on the ISA. These are: using special NOOPs, releasing at branches, releasing at procedures, and releasing through tagging the last use of each instruction. This section describes the minimal architectural impact of each of the schemes and provides a short description of the compiler analysis and ISA modifications required.

7.1 Commit NOOPs

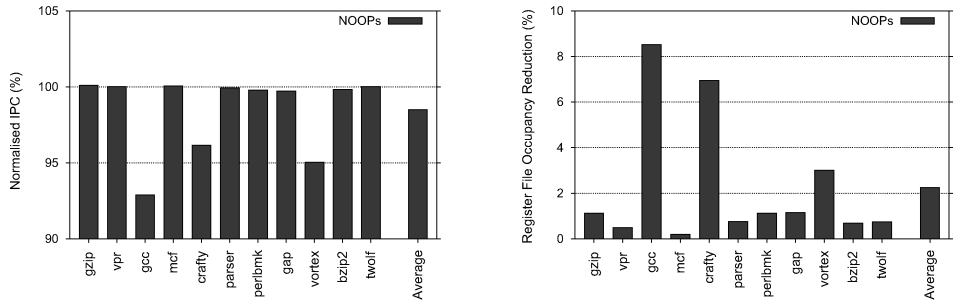
To implement early releasing techniques, ideally there should be as little impact on the processor and ISA as possible. One approach is to release registers early through a new instruction, to guarantee backward compatibility. We chose to use a special NOOP, called a commit NOOP. These special NOOPs do not affect the program's semantics but simply hold an encoding of the registers that are to be released. This allows the processor to release them early at the commit stage.

7.1.1 *Microarchitecture Changes and ISA Impact.* The commit NOOPs need to be dispatched in the same manner as any other instruction. They are stored in the reorder buffer along with the other instructions so that they can release a set of registers on commit. However, they are not executed, since they perform no operation. When a commit NOOP is committed, the processor attempts to release a set of registers early. To represent all 32 logical registers in the processor, only 5 bits are needed. Each NOOP has 25 bits free allowing it to release a maximum of 5 registers at commit. Unused slots in the NOOP encoding are used to free the zero register, which the processor ignores. Although the use of commit NOOPs adds a new instruction to the ISA, no existing instructions are altered, so backward compatibility is maintained. This is true for binaries containing NOOPs with the default encoding of the zero register in these 25 free bits.

For each encoded register the retirement map table is accessed, the correct physical register found, and the relevant checkpointed bit read. If the checkpointed bit for a register is clear then, in the following cycle, it can be released early after first copying its value to the checkpoint bitcells. If the checkpointed bit is set, then the value held in the shadow bitcells of the physical register is needed so the register cannot be released early. Each register that is released early is recorded in the register retirement map table by setting the checkpointed bit, as described in Section 6.3. This technique requires modest overheads: enough ports to access the checkpointed bits in the physical registers via the register retirement map table and check that they are set. Allowing two commit NOOPs to commit each cycle means that there needs to be 10 extra ports to the checkpointed bits.

7.1.2 *Compiler Analysis.* We use the compiler analysis described in Section 5 to determine the final users of each register. Commit NOOPs are inserted at the start of each basic block to release registers that have been used along all paths before the basic block but are not live into it.

7.1.3 *Results.* Figure 6 shows the effect of releasing registers early through this technique. As Figure 6(a) shows, several benchmarks experience a decrease in IPC due to the commit NOOPs taking up processor resources (e.g., the reorder buffer). Register file occupancy is normalized to be the difference between the baseline (at 0%) and the oracle (at 100%) in Figure 6(b) (similar to Figure 3(d)). The average register file occupancy is slightly reduced overall, bringing savings of 2% of the maximum possible. This translates into average dynamic energy



(a) NORMALISED IPC

(b) OCCUPANCY REDUCTION EXPRESSED AS A PERCENTAGE OF THE MAXIMUM AVAILABLE

Fig. 6. Releasing using commit NOOPs.

savings of 6% and average static energy savings of 16% in the register file, *gcc* benefiting the most.

Although this technique shows some register file occupancy reduction and energy savings, it loses performance and there remains significant room for improvement. Therefore, this approach is not considered further.

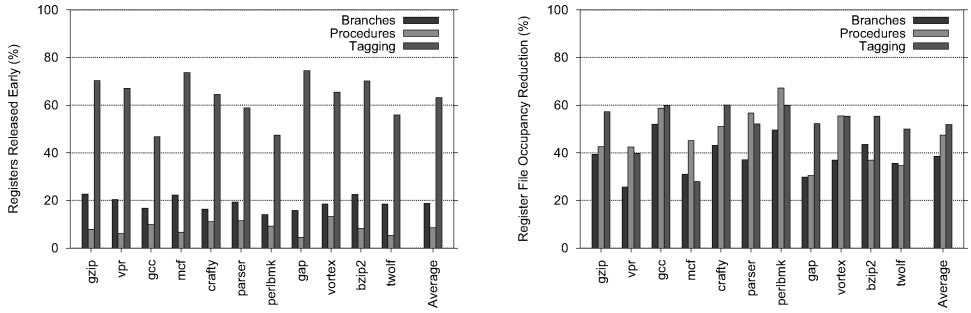
7.2 Branches

As discussed in Section 7.1.3, the downside of using commit NOOPs is that they take up processor resources. This scheme considers the case where branch instructions have spare bits that can be used to release a set of registers early, in much the same way as the commit NOOPs. The differences are that the branch instruction is actually executed and that it occurs at the end of a basic block rather than at the start, so it should release registers used in the block it belongs to. Using branches also means that some blocks will not have their registers released early because some do not terminate with a branch instruction.

The microarchitecture changes required to support releasing on branches are exactly the same as for commit NOOPs, using just the checkpointed bits of the physical registers and the register retirement map table. We release only 1 register, requiring 5 bits, at each branch instruction. The compiler analysis for this scheme is similar to that used when releasing with commit NOOPs.

7.3 Procedures

Another approach to early releasing is to release caller-saved registers at calls and returns because they are guaranteed not to be live afterward. The exact set can be altered depending on the calling conventions the compiler wishes to use and can be passed to the processor by a special NOOP, different to that mentioned in Section 7.1, which encodes the set of registers the compiler will use as caller saved. It can be placed at the start of the binary and whenever the compiler alters its set of caller-saved registers to inform the processor of the change. The processor can store the encoded registers in a small buffer and release these on each call and return. As in Section 7.1, these NOOPs add an instruction to the ISA but maintain backward compatibility.



(a) Percentage of registers released early

(b) Occupancy reduction expressed as a percentage of the maximum available

Fig. 7. Individual commit releasing schemes.

7.4 Instruction Tagging

The final commit-releasing technique has the largest ISA impact and requires two spare bits in each instruction, one for each source register. If either bit is set, it indicates that this instruction is the last consumer of the register. The microarchitecture changes needed to implement this scheme are the same as for commit NOOPs in Section 7.1. Additionally, the hardware must be able to check each source register's last consumer bit and release the corresponding physical register if set. Once again, the compiler analysis to find the last consumer of each register is used, as described in Section 5.

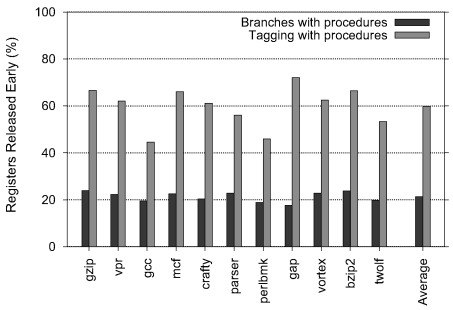
7.5 Combined

By combining several commit releasing techniques, it may be possible to take advantage of different approaches. Procedure releasing can be combined with all other schemes. However, it does not make sense to combine releasing on branches with tagging because they release the same registers: those that are tagged just release slightly earlier (they do not have to wait for the branch to commit). Hence, two new techniques are evaluated: branch and procedure releasing, and tagging and procedure releasing.

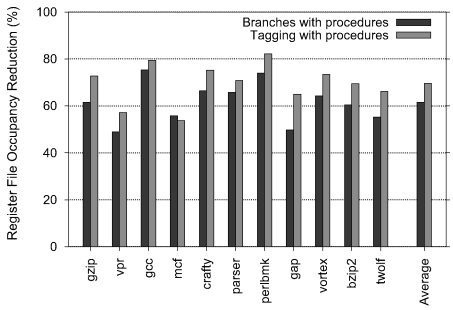
7.6 Results

The results from the commit releasing techniques are shown in Figure 7. We do not show performance because there is only a negligible change when using these schemes (less than 0.1%). Figure 7(a) shows the percentage of registers that are released early. On average, releasing at branches and at procedure boundaries means that 19% and 9% of all registers are released early. However, using the tagging approach means that 63% of all registers are released early at commit, freeing the main physical register, which can be reused by another instruction.

The impact of this is shown in Figure 7(b). This shows that tagging the last use of each register gives the most register file occupancy reduction, gaining 52% of the benefits, on average. This translates to 7% dynamic and 26% static



(a) PERCENTAGE OF REGISTERS RELEASED EARLY



(b) OCCUPANCY REDUCTION EXPRESSED AS A PERCENTAGE OF THE MAXIMUM AVAILABLE

Fig. 8. Releasing at procedures combined with other commit releasing schemes.

energy savings in the register file. Procedure releasing, on the other hand, saves a similar amount of dynamic energy (6% dynamic and 10% static) even though it has a lower occupancy reduction. This is because the tagging scheme checks the physical register checkpointed bits many more times than when only releasing at procedures, incurring a greater dynamic energy overhead.

For some benchmarks, however, releasing at procedure boundaries results in a higher occupancy reduction than releasing through tagging (e.g., *vpr*, *mcf*, *parser*, and *perlbmk*). In these applications, the time between final use and redefinition of the registers released at procedure boundaries tends to be longer than for other registers. Therefore, although fewer registers are released early, those that are have more of an impact on register occupancy because a larger fraction of register idle time is saved.

Combining several commit-based early releasing techniques can bring further benefits. Figure 8(a) once again shows that using tagging means that almost two thirds of all registers are released early. In terms of occupancy reduction (shown in Figure 8(b)), releasing registers through tagging the last use combined with procedure releasing gains 70% of the savings, almost the same amount as the oracle when it releases at commit (it achieves 73%). This shows that using the compiler with simple microarchitecture and ISA changes can reduce the register file occupancy almost to the limit of that which is achievable. This translates into a savings of 8% dynamic and 26% static energy in the register file. Again, there is only a negligible impact on performance.

Table II shows the ISA changes required from using each commit-based approach, benefits, in terms of occupancy reduction from baseline to oracle and register file energy savings that result. In terms of static energy savings, it shows the reductions achieved through using the super-drowsy shadow bits, described in Section 6.1, and without this circuit scheme (in brackets).

8. ISSUE-BASED RELEASING

As shown in the previous section, commit-based early releasing can be beneficial to register occupancy. One advantage of these schemes is that they require only modest hardware modifications. However, as we discovered in Section 4,

Table II. Summary of Commit Releasing Schemes

Scheme	Benefit	Dynamic	Static
Commit NOOPs	2%	6%	16% (10%)
Branches	38%	7%	24% (18%)
Procedures	47%	7%	26% (20%)
Tagging	52%	7%	26% (20%)
Branches & procedures	62%	8%	28% (22%)
Tagging & procedures	70%	8%	26% (20%)

We show the benefit in terms of available register file occupancy reduction and dynamic register file energy savings. Static register file energy savings are shown with (without) the superdrowsy circuit for the shadow bitcells.

releasing a register at the issue, rather than the commit, of its consumer potentially gives greater occupancy reduction at the cost of increased hardware resources. Hence, this section describes and evaluates issue-based early releasing techniques.

Figure 9 shows the cumulative benefits that can be gained from issue-based releasing schemes. The x-axis shows the maximum number of uses a register has before it is released early. So, for example, when releasing registers with 4 uses or fewer at commit, 46% of the register file occupancy savings are obtained. However, releasing at the issue stage of the pipeline achieves a 74% reduction.

Issue-based releasing allows the benefits to be quickly realized. In fact, releasing all registers that are only used once (one-use) gains 53% of the occupancy savings. Releasing two use as well gains 69%. This section only considers releasing one-use and two use registers. Although there are benefits to releasing three-use registers, our experiments have shown that in practice, these are hard to obtain.

In this section, we consider the use of register renaming to pass information from the compiler about the number of consumers each register has. Although we could have used schemes such as those proposed in Section 7 (i.e., special NOOPs or tagging), we chose to use register renaming because it requires only a minor ISA change (see Section 8.1.1) and incurs no performance penalties.

The remainder of this section is structured as follows. Section 8.1 describes the microarchitecture changes and compiler analysis required to release one-use registers early. Section 8.2 then extends this to two-use registers. Finally, Section 8.3 presents the results from these schemes.

8.1 One-Use Registers

Previous work [Franklin and Sohi 1992; Canal and González 2001] has discovered that many registers only have one consumer instruction. This can be exploited by releasing these registers (henceforth referred to as one-use registers) after the issue of their only consumer because they will not be read again along any control path without first being redefined. As Figure 9 shows, approximately 57% of the benefit could be achieved from releasing such registers.

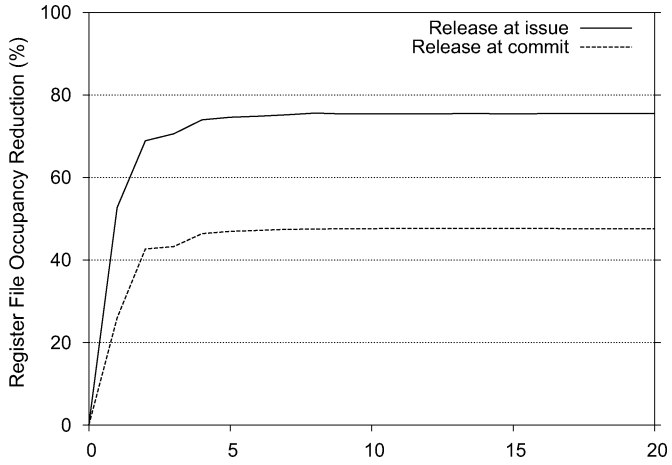


Fig. 9. The cumulative increase in occupancy reduction from baseline to oracle, achieved by releasing registers with a fixed number of consumers.

8.1.1 *Compiler Analysis.* The compiler analysis for identifying one-use registers is described in Section 5. After this identification pass has been performed, we rename all registers so that the one-use registers use a predefined set of logical registers, and all others use a different set. The processor knows which registers belong in the different sets. Therefore, when it decodes an instruction that uses a register, it immediately knows whether this register is one-use or not.

The first task is to rename all registers to virtual registers to separate out different uses of the same logical register. This is because the first time a register is defined it might be used just once, but the second time, it may be used many times. Renaming to virtual registers assigns a unique virtual register number to each new definition, completely removing the link between unrelated register definitions.

The next step is to recreate the interference graph, which is then colored with registers to get the final code. This graph coloring can never introduce spill code because the same number of logical registers are available for allocation as there were before they were renamed to virtual registers. For this, we use a standard graph coloring technique, as described by Appel [2002].

Nodes with high degrees are colored first. A greedy algorithm is used to select the node with the highest degree and coloring proceeds from a fixed order of registers. One-use registers are allocated from one end of the predefined ordering (e.g., *r0*, *r1*, . . .), multiuse from the other (e.g., *r31*, *r30*, . . .). Registers at 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 multiuse register to guarantee correctness. The maximum number of multiuse registers ever needed across the entire program is recorded. From this, the compiler can safely determine the number of one-use registers. Experiments show that the entire ordering is never needed for

multi-use values and that there is always room for at least five one-use registers. However, if the compiler wanted to change this, it could easily do this through the use of a special NOOP, called a usage NOOP, which could encode the new number of one-use registers.

8.1.2 *Microarchitecture Changes.* Only simple changes to the microarchitecture are needed to support the early release of one-use registers in the issue stage of the processor. The register dispatch map table has a single bit added to each logical register entry, which is called the early_release bit. This is used to determine, at dispatch, whether a register can be released early or not. These bits are set through the use of a usage NOOP, which is stripped out of the instruction stream at dispatch.

The reorder buffer is augmented with two extra bits per source register to enable early releasing: an early_release bit and a did_early_release bit. When an instruction dispatches, the processor copies the early_release bit for each of its source registers from the dispatch map table into the early_release bits in the reorder buffer.

When the instruction issues, the checkpointed bit of each source physical register is read at the same time as the relevant early_release bit from the reorder buffer. This is in parallel to reading the data held in the main part of the register. For early releasing to take place safely, the early_release bit should be set and the checkpointed bit unset (to show that there is no valid data in the shadow cells of the physical register). If early releasing takes place, then in the following cycle, the did_early_release bit is set for the source registers that are released early.

When an instruction commits, the did_early_release bit for each source register is copied to the register retirement map table for use in the event of an interrupt or exception. The previous version of its logical destination register is also released, as described in Section 6.3.

On a branch misprediction, 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 can be restored for the correct user to read.

The overhead of this scheme, in addition to those described in Section 6.3, is 2 bits per source register in the reorder buffer. This equals 512 bits in total.

8.1.3 *Static and Dynamic One-Use Registers.* The most basic technique for releasing one-use registers early is to provide a certain number of fixed, one-use registers. Through analysis of the register requirements of the benchmark programs, five of the registers can be reserved for use as one-use registers. This is called a static number of one-use registers because these five are only ever used as one-use registers to be released early, never as multiuse registers.

A more complex but flexible scheme adapts the number of one-use registers to the changing requirements of the program. The number of these registers is fixed on a per-procedure basis through the use of a usage NOOP. The number encoded in them sets the number of registers from the predefined ordering that will be one-use until another usage NOOP alters them again. This approach is

called a dynamic number of one-use registers because the number changes as the program executes.

8.2 Two-Use Registers

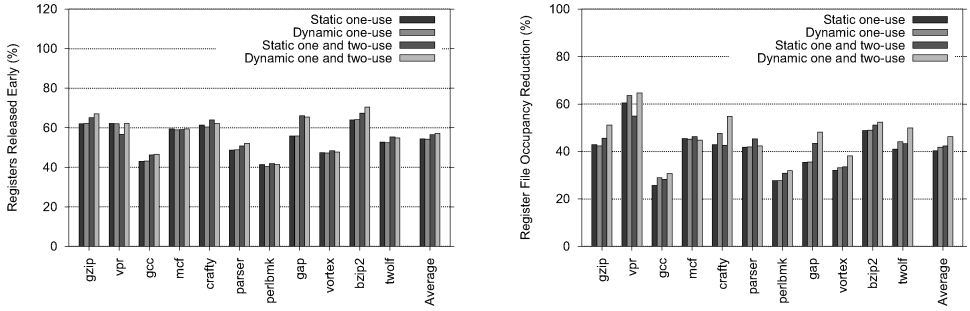
With the ability to release one-use registers early, the natural next step is to increase this to allow two-use registers to also be released early. The compiler analysis must change to identify these registers. However, the changes are simple and only require that along every path in the control flow graph from a defining instruction, the register defined is used exactly twice before being redefined. Both a static and dynamic number of two-use registers are considered. In the static approach, four registers are used for one-use values and a single register for two-use. In the dynamic case, the number of one- and two-use registers is encoded in a usage NOOP as before.

8.2.1 *Compiler Analysis.* The compiler analysis to find two-use registers is described in Section 5. Logical registers are renamed to virtual registers, one-use and two-use registers are identified and the interference graph created, as in Section 8.1.1. Coloring proceeds from the fixed ordering, as before, with two small changes. When a dynamic number of one- and two-use registers are being allocated, the one-use registers are allocated first from one end of the ordering, then the two-use, with the condition that no register that has been designated one-use anywhere in the procedure can be allocated to a two-use register. This is to ensure that a register is either one-use, two-use, or multiuse for the whole procedure. In the case of allocating a static number of one- and two-use registers, specific registers are designated one-use or two-use and these are used, if there is no interference, when allocating a register of the respective type.

8.2.2 *Microarchitecture Changes.* The changes to the microarchitecture needed to support the early release of two-use registers in addition to one-use are simple. The register map tables and reorder buffer are altered in a similar way to that described in Section 8.1.2, and some further additions are also made. The register dispatch map table, instead of having an early_release bit, is given two extra bits per entry to indicate whether a register is one-use, two-use, or multiuse. It also gets an extra bit, called the last_user flag, which indicates whether an instruction is the last user of a physical register. It is set to one for a newly renamed one-use register, or zero otherwise.

When an instruction dispatches the early_release bits in reorder buffer entry are set if the corresponding logical source register is one-use or two-use (as indicated in the dispatch map table) and the last_user flag is set as previously described. Once the instruction has dispatched, the last_user flag is reversed. This means that a two-use register will only be released early once, by its second consumer instruction.

Along with a checkpointed bit for each physical register, several bits are needed to count the number of consumer instructions that have to issue. This is called the consumer_counter, and for releasing two-use registers, it only needs to be 1 bit. When an instruction issues, it can release the physical register



(a) Percentage of registers released early

(b) Occupancy reduction expressed as a percentage of the maximum available

Fig. 10. Releasing one-use and two-use registers.

early if the checkpointed bit is unset, the consumer_counter is zero, and the early_release bit in its reorder buffer entry is set. This means that only the second consumer of a two-use register can release a physical register early and only if it issues after the first consumer. This may lead to missed opportunities to release a register early but is important so that recovery from branch mispredictions is not too complex. In fact, recovery from branch mispredictions is the same as described in Section 8.1.2 for one-use registers.

The overhead of this scheme is 3 bits per entry in the dispatch map table, 2 bits in the reorder buffer per source, as before, and a bit per physical register. Again, there is only a minor ISA impact for this early releasing scheme due to the addition of a usage NOOP.

8.3 Results

The results from releasing one- and two-use registers are shown in Figure 10. Figure 10(a) shows the percentage of registers that are released early. In general, all four schemes perform well, releasing over half of all registers early, on average. This translates into register file occupancy changes, as can be seen in Figure 10(b). It is clear that releasing one-use registers is beneficial. However, it appears that dynamically changing the number of one-use registers does not give much of an improvement over having a static number. This is because the majority of procedures only need a few one-use registers so providing only five in the static case achieves most of the benefits.

There is a negligible performance impact from using these schemes, apart from in gcc and vortex. These benchmarks lose 2% and 4% IPC, respectively, when using the dynamic one-use and dynamic two-use schemes. This loss of performance is due to the usage NOOPs that are inserted to change the register mappings.

Releasing a static number of one- and two-use registers brings some benefits. However, when a dynamic number are set, further improvement occurs. For example, crafty experiences an increase from 43% of the difference in occupancy in the static case to to 55% in the dynamic case. This is because some procedures

Table III. Summary of Issue Releasing Schemes

Scheme	Benefit	Dynamic	Static
Static one-use	40%	6%	24% (18%)
Dynamic one-use	42%	6%	24% (18%)
Static one- and two-use	42%	6%	24% (19%)
Dynamic one- and two-use	46%	6%	25% (19%)

We show the benefit in terms of available register file occupancy reduction and dynamic register file energy savings. Static register file energy savings are shown with (without) the super-drowsy circuit for the shadow bitcells.

have more than the single two-use register that is provided and this can be exploited when usage NOOPs are used to alter the number available.

Not all the benefits that are available from releasing one-use registers, suggested by Figure 9, are obtained, as discussed in Section 5. Some registers are one-use along one path through the control-flow graph but have several uses along another. They cannot be classified as one-use since it is not known at compile time, which path will be taken, so if the one-use path is actually the correct one when the program runs, an opportunity to release early is missed.

A summary of the issue-based releasing schemes is given in Table III. All techniques gain at least 40% of the register file occupancy savings, with dynamic two-use releasing reaching 46%. Dynamic energy savings in the register file are 6% for each scheme and static register file energy savings range from 24% to 25%.

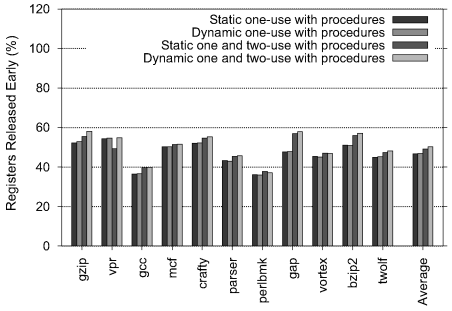
9. COMBINED APPROACHES

This section evaluates techniques that combine issue and commit releasing, aiming to further improve occupancy reduction. The three commit-based and four issue-based schemes described in Sections 7 and 8 are combined together. The combinations vary in terms of their performance and hardware/ISA impact. It then recommends three compiler-based schemes that could be used depending on your design requirements. It presents the dynamic and static energy savings that they achieve in the register file. The combined schemes work by releasing registers both at the issue and commit stages of the pipeline.

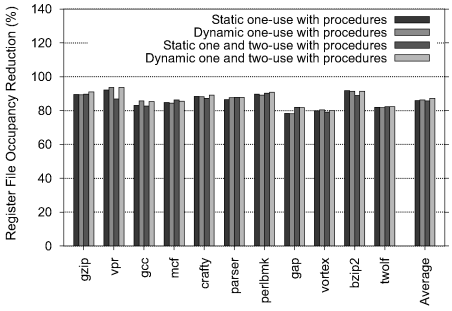
9.1 Results

Figure 11(b) shows the reduction in occupancy when commit-based procedure releasing is added to each issue-based scheme. The results are within 13% of the limit of benefits when combining with dynamic two-use registers. Releasing static one-use registers and procedure releasing achieves within 14% of the limit, or gains 86% of the available benefits, and this requires little modification to the microarchitecture.

Combining commit-based tagging to the issue-based schemes achieves less impressive results, giving, on average, around 56% of the available register file occupancy reduction, as shown in Figure 12(b). However, by using both commit releasing techniques with one-use and two-use releasing, savings close to 6%

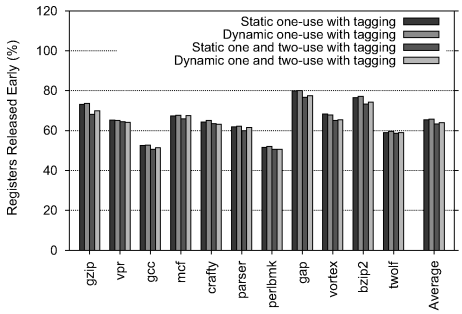


(a) PERCENTAGE OF REGISTERS RELEASED EARLY

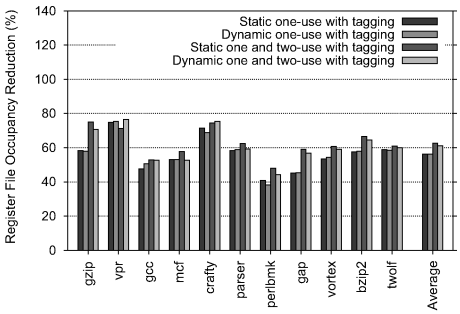


(b) OCCUPANCY REDUCTION EXPRESSED AS A PERCENTAGE OF THE MAXIMUM AVAILABLE

Fig. 11. Combining issue releasing with procedure commit releasing.



(a) PERCENTAGE OF REGISTERS RELEASED EARLY



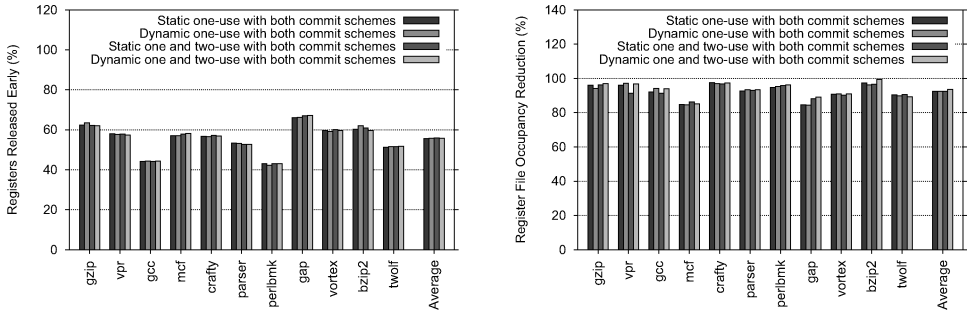
(b) OCCUPANCY REDUCTION EXPRESSED AS A PERCENTAGE OF THE MAXIMUM AVAILABLE

Fig. 12. Combining issue releasing with tagging commit releasing.

of the limit can be achieved, or 94% of the benefits gained. This is shown in Figure 13(b). This requires major changes to the ISA, and microarchitecture should be used only if reducing the register file occupancy is a major concern. Otherwise, early releasing implemented with static one-use issue releasing and commit releasing on procedures gives a good trade-off between savings and changes to the microarchitecture.

In general, tagging registers means that dynamic energy savings are limited due to the overhead in accessing the checkpointed bits frequently to determine whether a register can be released early. Although a smaller occupancy reduction is achieved using procedure commit-based releasing, the infrequent checking of the checkpointed bits means that there is less of an overhead and consequently greater dynamic energy savings can be achieved.

In all these combined schemes, the impact on performance is similar to just using issue-based early releasing alone (Section 8.3). In summary, there is a negligible change in IPC for all benchmarks apart from gcc and vortex, which lose 2% and 4% performance, respectively, when using usage NOOPS in the dynamic schemes.



(a) Percentage of registers released early

(b) Occupancy reduction expressed as a percentage of the maximum available

Fig. 13. Combining issue releasing with tagging and procedure commit releasing.

Table IV. Summary of Recommendations

Requirement	Procs	Tagging	Dyn 1&2	Benefit	Dynamic	Static
Minimal hardware	Yes	Yes		70%	8%	26% (24%)
Backwards compatible	Yes		Yes	87%	8%	32% (27%)
Lowest occupancy	Yes	Yes	Yes	94%	8%	33% (29%)

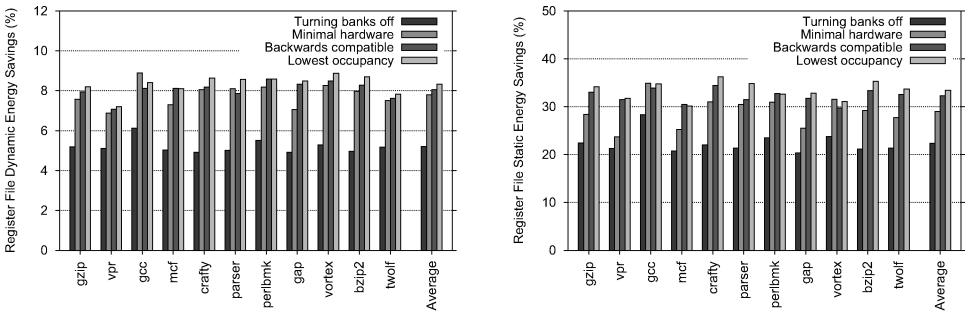
The benefit is the reduction in register file occupancy, dynamic is the dynamic energy savings made in the register file, static is the static energy savings made in the register file with (without) using a super-drowsy circuit for the shadow bitcells.

9.2 Recommendations

Having evaluated our compiler approaches, this section recommends three schemes that could be implemented, depending on limitations of the processor.

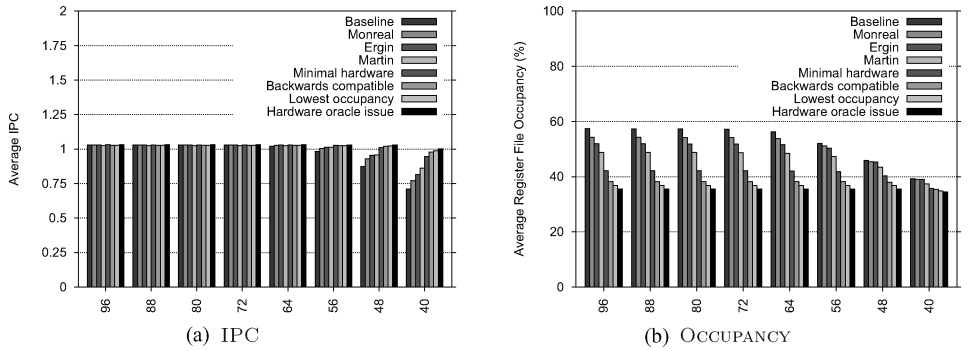
Table IV shows a summary of the recommended schemes. If the ISA impact is to be kept to a minimum, then using dynamic one- and two-use, issue-based releasing combined with commit releasing at procedures is suggested, as this introduces an additional instruction to the ISA but maintains backward compatibility. If the changes to the microarchitecture are to be minimal, then only commit-based releasing should be used. The lowest occupancy is given by the dynamic two-use, issue-based releasing with both commit-based releasing techniques.

Figure 14 shows the normalized dynamic and static energy savings in the register file for the recommended schemes. Also shown are the energy savings that can be achieved on the baseline simply by turning off empty register file banks, with no further optimizations. All compiler approaches achieve 8% dynamic energy savings in the register file (the increasing overheads for the more complex schemes outweigh the gains achieved through more aggressive early releasing). The Minimal hardware approach has the lowest static energy savings of 26% in the register file, shown in Figure 14(b). This is because it only manages to gain 70% of the available benefits. The other schemes manage better with Backward compatible achieving 32% static energy savings from 87%



(a) REGISTER FILE DYNAMIC ENERGY REDUCTION (b) REGISTER FILE STATIC ENERGY REDUCTION

Fig. 14. Normalized dynamic and static register file energy reduction for recommended compiler-directed early releasing schemes.



(a) IPC (b) OCCUPANCY

Fig. 15. Effects of reducing the register file size.

of the occupancy benefits and Lowest occupancy gaining 94% of the benefits, converted to 33% static energy savings in the register file.

10. REGISTER FILE SIZE SENSITIVITY

To verify that the approaches are not specific to a particular register file size, we performed experiments, which decrease the number of registers from 96 (as in the original configuration) to 40 (when only 8 registers are available for holding noncommitted values, the others being used to store the 32 logical registers). We made a comparison between the recommended techniques (Minimal hardware: procedure and tagging commit; Backward compatible: dynamic two use with procedures; and Lowest occupancy: dynamic two use with both commit schemes) and approaches proposed by Monreal et al. [2002], Ergin et al. [2004] and Martin et al. [1997]. We also show the baseline and oracle.

Figure 15(a) shows the IPC for all schemes and Figure 15(b) shows the register file occupancy. As shown, all three of our recommended schemes considerably outperform state-of-the-art approaches across all register file sizes. For large register file sizes, the schemes have better occupancy reduction,

while for small register files, they have superior IPC, approaching the oracle performance in both cases.

As the register file size decreases so does the IPC, although our schemes are always better than the others almost achieving the IPC gains of the oracle. In fact, with a register file size of only 48 entries the IPC of Lowest occupancy is still as good as the baseline with 96 entries. The Monreal, Ergin, and Martin schemes do manage to maintain the same IPC as the baseline, and increase it for small register files, but the compiler techniques presented in this article are consistently better. For example, when there are only 40 registers available then Lowest occupancy increases the IPC from 0.71 to 0.99, an increase of 39%, whereas Martin manages an increase of 21%, Ergin 15%, and Monreal only 8%.

The effects of reducing the register file size on the occupancy are shown in Figure 15(b). With large register file sizes, our schemes considerably reduce the register pressure, far more than Monreal, Ergin, or Martin. In fact, even with a register file size of 40, Lowest occupancy reduces the register pressure by 11% (from 39 to 35), whereas Martin can only reduce it by 5% (to 37). Overall, the recommended schemes are able to maintain higher IPC and reduce register pressure, allowing greater energy savings across all configurations.

11. CONCLUSIONS

This article presented a detailed study of early register releasing with compiler support. We have proposed two types of compiler-directed early releasing, commit-based and issue-based, and shown that together they can reduce register pressure significantly. We have implemented an oracle and found that the best compiler analysis is almost as accurate as this oracle. Our best compiler approach with significant, but realistic changes to the hardware can make savings of 94% of the total. Alternatively, a scheme that maintains backward compatibility with existing binaries can achieve 87% of the maximum occupancy reduction available, or another with minimal hardware changes gains 70% of the benefits. Comparing our techniques to two recently proposed hardware early releasing schemes and one previous compiler-directed approach shows that the register pressure can be significantly improved. Furthermore, our schemes are superior across all register file sizes. In summary, our compiler-directed schemes for early release can approach the limits of that which is possible, out-performing state-of-the-art techniques while relying on less complex hardware.

REFERENCES

ABELLA, J. AND GONZÁLEZ, A. 2003. On reducing register pressure and energy in multiple- banked register files. In *Proceedings of the 21st International Conference on Computer Design (ICCD-21).* IEEE, Los Alamitos, CA.

APPEL, A. W. 2002. *Modern Compiler Implementation in Java*. Cambridge University Press, Cambridge, UK.

BALASUBRAMONIAN, R., DWARKADAS, S., AND ALBONESI, D. H. 2001. Reducing the complexity of the register file in dynamic super-scalar processors. In *Proceedings of the 34th International Symposium on Microarchitecture (MICRO-34).*ACM, New York.

BORCH, E., MANNE, S., EMER, J., AND TUNE, E. 2002. Loose loops sink chips. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture (HPCA-8).* IEEE, Los Alamitos, CA.

BROOKS, D., TIWARI, V., AND MARTONOSI, M. 2000. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th International Symposium on Computer Architecture (ISCA-27)*. ACM, New York.

BURGER, D. AND AUSTIN, T. 1997. The simple-scalar tool set, version 2.0. Tech. rep. TR1342, University of Wisconsin-Madison.

BUTTS, J. A. 2004. Optimizing inter-instruction value communication through degree of use prediction. Ph.D. thesis, University of Wisconsin-Madison.

BUTTS, J. A. AND SOHI, G. S. 2004. Use-based register caching with decoupled indexing. In *Proceedings of the 31st International Symposium on Computer Architecture (ISCA-31)*. ACM, New York.

CANAL, R. AND GONZÁLEZ, A. 2001. Reducing the complexity of the issue logic. In *Proceedings of the 15th International Conference on Super-Computing (ICS-15)*. ACM, New York.

CRUZ, J.-L., GONZÁLEZ, A., VALERO, M., AND TOPHAM, N. P. 2000. Multiple-banked register file architectures. In *Proceedings of the 27th International Symposium on Computer Architecture (ISCA-27)*. ACM, New York.

EMER, J. 2001. Ev8: The post-ultimate alpha. In *Proceedings of the 10th International Conference on Parallel Architectures and Compilation Techniques (PACT'01)*. (Keynote.) ACM, New York.

ERGIN, O., BALKAN, D., GHOSE, K., AND PONOMAREV, D. 2004. Register packing: Exploiting narrow-width operands for reducing register file pressure. In *Proceedings of the 37th International Symposium on Microarchitecture (MICRO-37)*. ACM, New York.

ERGIN, O., BALKAN, D., PONOMAREV, D., AND GHOSE, K. 2004. Increasing processor performance through early register release. In *Proceedings of the 22nd International Conference on Computer Design (ICCD-22)*. IEEE, Los Alamitos, CA.

FRANKLIN, M. AND SOHI, G. S. 1992. Register traffic analysis for streamlining inter-operation communication in fine-grain parallel processors. In *Proceedings of the 25th International Symposium on Microarchitecture (MICRO-25)*. ACM, New York.

GONZÁLEZ, A., GONZÁLEZ, J., AND VALERO, M. 1998. Virtual-physical registers. In *Proceedings of the 4th International Symposium on High Performance Computer Architecture (HPCA-4)*. IEEE, Los Alamitos, CA.

GUNTHER, S. H., BINNS, F., CARMEAN, D. M., AND HALL, J. C. 2001. Managing the impact of increasing microprocessor power consumption. *Intel Tech. J. Q1*.

HU, Z. AND MARTONOSI, M. 2000. Reducing register file power consumption by exploiting value lifetime. In *Proceedings of the Workshop on Complexity Effective Design (WCED) in Conjunction with the 27th International Symposium on Computer Architecture (ISCA-27)*. ACM, New York.

JONES, T. M., O'BOYLE, M. F., ABELLA, J., AND GONZÁLEZ, A. 2005. Software directed issue queue power reduction. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA-11)*. IEEE, Los Alamitos, CA.

JONES, T. M., O'BOYLE, M. F. P., ABELLA, J., GONZÁLEZ, A., AND ERGIN, O. 2005. Compiler directed early register release. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. ACM, New York.

KIM, N. S., FLAUTNER, K., BLAAUW, D., AND MUDGE, T. 2004. Single-VDD and single-VT super-drowsy techniques for low-leakage high-performance instruction caches. In *Proceedings of the International Symposium on Low-Power Electronics and Design (ISLPED)*. ACM, New York.

KIM, N. S. AND MUDGE, T. 2003. The microarchitecture of a low-power register file. In *Proceedings of the International Symposium on Low-Power Electronics and Design (ISLPED)*. ACM, New York.

LIPASTI, M. H., MESTAN, B. R., AND GUNADI, E. 2004. Physical register in lining. In *Proceedings of the 31st International Symposium on Computer Architecture (ISCA-31)*. ACM, New York.

LO, J. L., PAREKH, S. S., EGGERS, S. J., LEVY, H. M., AND TULLSEN, D. M. 1999. Software-directed register deallocation for simultaneous multithreaded processors. *IEEE Trans. Paral. Distrib. Syst. 10*, 9.

MARTIN, M. M., ROTH, A., AND FISCHER, C. N. 1997. Exploiting dead value information. In *Proceedings of the 30th International Symposium on Microarchitecture (MICRO-30)*. ACM, New York.

MARTINEZ, J. F., RENAU, J., HUANG, M. C., PRVULOVIC, M., AND TORRELLAS, J. 2002. Cherry: Checkpointed early resource recycling in out-of-order microprocessors. In *Proceedings of the 35th International Symposium on Microarchitecture (MICRO-35)*. ACM, New York.

MONREAL, T., VIÑALS, V., GONZÁLEZ, A., AND VALERO, M. 2002. Hardware schemes for early register release. In *Proceedings of the International Conference on Parallel Processing (ICPP)*. IEEE, Los Alamitos, CA.

MOUDGILL, M., PINGALI, K., AND VASSILIADIS, S. 1993. Register renaming and dynamic speculation: An alternative approach. In *Proceedings of the 26th International Symposium on Microarchitecture (MICRO-26)*. ACM, New York.

PARK, I., POWELL, M. D., AND VIJAYKUMAR, T. N. 2002. Reducing register ports for higher speed and lower energy. In *Proceedings of the 35th International Symposium on Microarchitecture (MICRO-35)*. ACM, New York.

SAVRANSKY, G., RONEN, R., AND GONZÁLEZ, A. 2004. Lazy retirement: A power aware register management mechanism. In *Proceedings of the Workshop on Complexity Effective Design (WCED) in Conjunction with the 27th International Symposium on Computer Architecture (ISCA-27)*. ACM, New York.

SMITH, M. D. AND HOLLOWAY, G. 2000. The Machine-SUIF documentation set. http://www.eecs. harvard.edu/machsuif/software/software.html.

TARJAN, D., THOZIYOOR, S., AND JOUPPI, N. P. 2006. CACTI 4.0. Tech. rep. HPL-2006-86, HP Laboratories Palo Alto.

TRAN, L., NELSON, N., NGAI, F., DROPSHO, S., AND HUANG, M. 2004. Dynamically reducing pressure on the physical register file through simple register sharing. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*. IEEE, Los Alamitos, CA.

TSENG, J. H. AND ASANOVIĆ, K. 2003. Banked multiported register files for high-frequency superscalar microprocessors. In *Proceedings of the 30th International Symposium on Computer Architecture (ISCA-30)*. ACM, New York.

WALLACE, S. AND BAGHERZADEH, N. 1996. A scalable register file architecture for dynamically scheduled processors. In *Proceedings of the 5th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. ACM, New York.

Received June 2008; revised October 2008; accepted April 2009