Address-Value Decoupling for Early Register Deallocation

Deniz Balkan, Joseph Sharkey, Dmitry Ponomarev
Department of Computer Science
State University of New York at Binghamton
{dbalkan, jsharke, dima}@cs.binghamton.edu

Aneesh Aggarwal
Department of Electrical and Computer Engineering
State University of New York at Binghamton
aneesh@binghamton.edu

Abstract

We propose a series of aggressive register deallocation mechanisms to reduce the register file pressure and increase the parallelism exploited by superscalar microprocessors. Our techniques are based on a key observation that a register value can be temporarily decoupled from the register identifier. Specifically, even if a physical register is deallocated, the value is still available in the register and can be read by the dependent instructions until the register is overwritten. In these situations, we can effectively overlap the consumption of the produced register value and partial processing of the instruction that gets the same register reassigned to it. In this paper, we propose several realizations of the address-value decoupling idea and discuss their implications on the performance. Our most aggressive scheme achieves an average IPC speedup of 14.6% across simulated SPEC 2000 benchmarks.

1. Introduction

In modern superscalar microarchitectures, the access to the register file lies on the critical schedule-to-execute path. As a new physical register needs to be allocated for every instruction with a destination register, larger register files are required to extract more instruction-level parallelism (ILP) out of sequential programs. Additionally, to reduce the amount of data transfers, a single RAM structure is typically used to maintain both committed and speculative register values [13], [16], [34].

Register file access latency increases with the register file size. To achieve higher clock frequencies with large register files, the register file often needs to be pipelined over multiple cycles. A pipelined register file access degrades performance by increasing the branch and load-hit misspeculation penalties [5]. It also complicates the bypass network by increasing the number of stages for which the values are forwarded through the network. The situation is further exacerbated in the SMT processors, where a larger register file is required to satisfy the register requirements of the multiple threads running simultaneously. Using a smaller register file limits the amount of ILP that can be exploited, by stalling instructions in rename due to unavailability of registers. An effective alternative for a larger register file is to use fewer registers (facilitating faster clock), but manage them more efficiently. If the reduction of the register file size is not the goal, then such mechanism will simply increase the ILP that is exploited by providing the illusion of having more registers.

Traditional register allocation and deallocation techniques are very conservative – a new physical register is allocated for the destination of a new instruction at the time of dispatch and this register remains allocated until the next instruction writing to the same architectural register commits. In this paper, we propose a series of novel techniques for early register deallocation, thus making more registers available for rename. Our proposals are based on the key observation that a register value can be temporarily decoupled from the register address. Specifically, even if a register is deallocated, the value stored in that register is still available and can be read by the dependent instructions until the register is overwritten.

We describe several realizations of such Address-Value Decoupling (AVD). In our first implementation of AVD, a physical register is deallocated immediately after the result is written to the register. In our basic scheme, the instruction reacquiring the early deallocated register stalls at the time of dispatch if the prior value stored in that register has not been read by all its consuming instructions. To avoid these dispatch stalls, we introduce an auxiliary structure called Temporary Instruction Buffer (TIB), which temporarily holds instructions that would otherwise be stalled. The instructions are moved from the TIB to the issue queue when the old value stored in the instruction's destination register has been read by all its consumers. The key advantage of moving head-of-the-line instructions into the TIB is that subsequent instructions can execute normally, as long as the conditions imposed on their destination registers allow such execution.

We then take our ideas one step further and explore deallocating a physical register when the next instruction writing to the same architectural register is renamed. We explore several variations of this technique and show that the most aggressive solution achieves 14.6% IPC gains on the average across simulated SPEC benchmarks compared to the baseline machine with traditional register management

mechanisms. Finally, we show that our most aggressive technique favorably compares with some other recently proposed register file optimization schemes.

2. Background and motivations

2.1 Background

The design proposed in this paper relies on the use of Checkpointed Register File (CRF) circuitry that was introduced in [9]. For a circuit schematic and the detailed area, power and timing analysis of CRF we refer the readers to [9], in this subsection we just provide a brief overview of this mechanism.

In CRF, each traditional register file bitcell is backed-up by a shadow cell, which is connected to the main bitcell using pass transistors. When the *Checkpoint* signal rises, the contents of a bitcell are simply copied to the shadow cell. To recover, the contents of the shadow cells are copied back to the main storage when the *Recover* signal rises. As discussed in [9], the resulting bitcell area increases by about 26.5%. Since the area of the other peripheral components of the register file such as sensamps, decoders, word select drivers and prechargers is not impacted by the proposed bitcell modification, the overall increase in the area of the register file is less than 20%. There is a very slight increase in the register file delay due to the longer word select and bit lines. Since no gate capacitance is added to these lines, the increase in the delay is miniscule; it is less than 3% [9]. There is also a similar minimal impact on the delay of the word select line during the normal course of read and write accesses. Consequently, the CRF design represents an efficient way of doubling the register file storage without commensurate increase in the area, access delay, or power dissipation. The only limitation is that half of the register file bitcells (the shadow copies) are not directly accessible through the regular ports.

2.2 Motivations

To motivate the rest of this paper, we now present two key microarchitectural statistics regarding the producer-consumer relationships in a typical superscalar datapath.

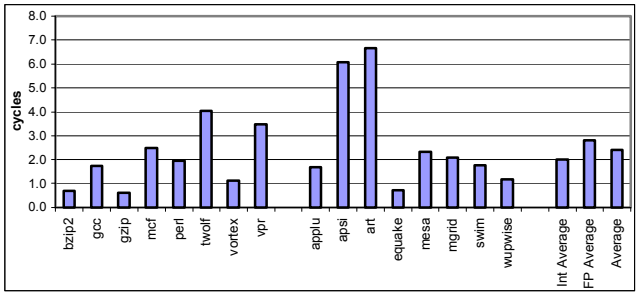


Figure 1. Number of cycles between the writeback of a short-lived value and the issue of its last consumer

First, it has been noticed by several researchers, that most of the register instances in a datapath are short-lived [11], [21], [29]. In [29], a value was defined as ***short-lived*** if the architectural register allocated of an instruction X had been redefined (i.e., a younger instruction that writes to the same architectural register has been renamed) before the value generated by X was written back. We also call the instruction that redefines a register allocated to hold a short-lived value as the ***redefiner***. It was shown in [29] that more than 85% of the generated values are short-lived; our results showed similar percentages.

Another key observation from our experiments is that *all* consumers of a short-lived value are typically issued in a short duration after the short-lived value is written into the register file. Figure 1 shows the average number of cycles between the writeback of the short-lived value and the issuing of the last consumer of that value. On the average across all simulated benchmarks, this duration is only 2.4 cycles, the maximum being slightly less than 7 cycles for *art*. The early register reclamation techniques that we propose in this paper are motivated by these two observations. In the next two sections, we describe the proposed mechanisms.

3. Writeback-time register deallocation

The key idea behind our first technique is to deallocate a register as soon as the following two conditions are true: a) the result has been written into the register and b) the register has been redefined.

To determine whether a register has been redefined or not, we maintain a bit-vector called *Redefined* with one bit for each physical register. When an instruction is renamed, it sets the *Redefined* bit corresponding to the previous mapping of its destination architectural register. The *Redefined* bits are reset when the corresponding physical registers are deallocated, or the redefiners of the physical registers are squashed on a branch misprediction. There is also another bit associated with each physical register, called *Written_back*, which is set one cycle before the writeback to the register takes place and reset when the register is deallocated. A value is identified as short-lived if both *Redefined* and *Written_back* bits are set either at the end of the last execution cycle or at the time of redefiner's renaming. Once a value is identified as short-lived, the register holding the value is immediately deallocated.

To understand the performance benefit of such early deallocation of registers, it is instructive to examine the code example shown in Figure 2. Both the original and the renamed version of a code fragment are shown in this figure.

In Figure 2, the instruction I1 deallocates its destination register P2 after it writes back the produced value into this register. Such an early deallocation is possible because, as we assume in this example, instruction I3 that redefines R1 is already renamed. The instruction I2 consumes the value of P2 *after* it is deallocated, and the instruction I4 is the next instruction that uses physical register P2 as its destination mapping. In the baseline machine without early register deallocation, the renaming of the instruction I4 could be stalled for several cycles due to unavailability of free physical registers - the renaming only resumes when a

committing instruction deallocates a physical register. In contrast, if the early register release proposed in this paper is implemented, then as soon as I1 writes back, its register (P2) can be reassigned to I4, thus avoiding the renaming stalls.

```
Original code  Renamed code
I1: ADD ->R1   ADD ->P2  /* ADD writes to R1 (P2) */
      ……….       ……….
I2: SUB <-R1   SUB <-P2   /* SUB reads R1 (P2) */
      ……….       ……….
I3: XOR ->R1   XOR ->P19 /* XOR writes to R1 (P19)*/

I4: NOR ->R5   NOR ->P2 /* NOR writes to R5 (P2) */
```

Figure 2. Example code sequence

On a branch misprediction or exception, the original contents of an early deallocated register may have to be recovered. For instance, in Figure 2, if I4 overwrites the value of P2 before I3 executes, and then I3 raises an exception, then the precise state of architectural register R1 is defined by the result of I1, which has to be recovered. To address this problem, the contents of an early-deallocated register are saved when the register is overwritten, using the register file design with shadow bitcells, as introduced in [9] and reviewed in Section 2.1. Since each register is only backed by a single shadow copy, a maximum of two allocated instances of the same physical register are allowed at one time. Hence, when an instruction writes back a short-lived value, it should deallocate the register only if it has the only instance of the register. To ensure that no more than two instances of a register are alive at the same time, we maintain two bits – *Early-deallocated Register Reallocated (ERR)* and *Short-lived Register Written-back (SRW)* – for each physical register.

The *ERR* and the *SRW* bits of a register are checked and updated when the register is allocated and written back, and when the redefiner of the register is renamed and committed. These bits can be trivially manipulated to reconstruct a precise state following exceptions or interrupts. The specific details are not central to the ideas proposed in this paper. Therefore, not to deviate from the conceptual discussions of our ideas, the details pertaining to the manipulation of the *ERR* and *SRW* bits, along with the finite state machine for updating these bits, are formally and completely presented in Appendix A.

In the example shown in Figure 2, it may happen that the instruction I4 writes the early deallocated and reallocated physical register P2 before the instruction I2 reads its operands, resulting in an incorrect value for I2. This could transpire because of the delaying of I2 in the issue queue for various reasons. To address this problem, we delay the dispatch of an instruction allocated an early deallocated register until all consumers of the previous register instance have read the value and have begun execution. In the above example, until the instruction I2 reads the register P2, I4 and all the following instructions are not dispatched. To detect this condition, we maintain a dynamic counter of the number of in-flight consumers for each physical register. These counters are incremented as the consumers are renamed, and are decremented as the consumers start execution or the consumers get squashed (because of branch misprediction). The consumer counters are reset to zero when a register is allocated. Similar support has been used by other works [4], [23] and [27]. An instruction can only be dispatched if the consumer counter, corresponding to its destination physical register, is zero. Otherwise, instruction dispatch stalls.

3.1 Priority-based register allocation

In the early register deallocation scheme discussed above, the free registers can be classified into three types: (i) normally released registers (i.e., the ones that have not been early deallocated), (ii) early deallocated free registers with consumer counter equal to zero, and (iii) early deallocated free registers with consumer counter not equal to zero. The number of dispatch stalls (which occur when an instruction that has been assigned an early deallocated register with a non-zero consumer counter reaches the dispatch stage) can be reduced by giving the highest priority to normally released registers during register allocation. It is also conceivable that the second highest priority should be given to the registers with the consumer counter equal to zero and the least priority should be given to the registers, whose consumer counters are not equal to zero. However, our results indicated little difference (when prioritizing between different types of early deallocated registers) in terms of the overall IPCs, and therefore we only make a distinction between two priority classes among the registers in the free list: normally-released registers (which are given higher priority) and early-released registers. To determine the class of a register, we can use the *ERR* and the *SRW* bits. If the *ERR* and the *SRW* bits of a register that is being deallocated are set to "00", then the register is a normally released register, else the register is an early released register. If the bits for a register are set to "00", the bit for that register in the early-released register free list is reset and that in the normally released register free list is set.

4. Reducing the number of stalls: TIB

In Section 3, instruction dispatch stalls if an instruction with an early deallocated register assigned to it reaches the dispatch stage and the pending consumer counter of the register is not zero. Consequently, all subsequent instructions, even when they are independent of the blocked instruction, are also stalled, often unnecessarily. In this section, we propose the use of a small Temporary Instruction Buffer (TIB) to temporarily store (provided that a free entry exists in the TIB) the instructions which were allocated registers with the non-zero consumer counters. Simultaneously, to avoid deadlock, these instructions are also dispatched to the issue queue; however, the corresponding issue queue entries are not marked "valid" immediately. If a

free entry is not available either in the issue queue or within the TIB, then instruction dispatching stalls as before.

When an instruction is dispatched into the TIB, its TIB entry is tagged with the destination register of the instruction. Simultaneously, the destination register of the instruction is marked (by setting a bit for the register). The "marking" is removed when the register is deallocated. Eventually, when the consumer counter associated with this destination register reaches zero, the register identifier is broadcast (using tag lines) across the TIB and the instruction with that particular destination register is moved into the issue queue (the "movement" here simply accounts to validating the corresponding issue queue entry, which was already established at the time of establishing the TIB entry), freeing up its TIB entry. The real key advantage of this technique is that instructions following the one sent to the TIB (even if they depend on it) are still dispatched to the issue queue provided that their register states allow it (their destinations do not have a non-zero consumer counter). Obviously two sets of consumer counters are needed in this case to keep an accurate track of the number of in-flight consumers of each register instance.

5. Rename-time register deallocation

An even more aggressive realization of the AVD philosophy is to deallocate a register when its redefiner is renamed, regardless of whether the value is produced or not. Hardware support for this scheme is similar to that described in Section 3 (and in Appendix A) with some trivial modifications. The consumer counters and the *Redefined* bit-vector remain the same as in Section 3. The *ERR* and the *SRW* bits also remain the same. However, they are not updated at writeback, but only when the redefiner is renamed. The main difference is that instead of one *Written_back* bit per register, this scheme requires two *Written_back* bits per register. These two bits are needed, because with rename-time deallocation scheme there can be two in-flight instructions with the same destination physical register that have both not reached the writeback stage. The branch misprediction handling mechanism now needs to consider the value of both the *Written_back* bits to update the bits and register free list accordingly. We do not present all details here due to the space constraints, but the logic described in Appendix A can be easily extended to support this with two copies of *Written_back* bits.

Several variations of this general rename-time deallocation AVD scheme can be considered.

Variation (a). First, an instruction that acquired an early-released register can be stalled at the time of dispatch if the consumers of the previous instance have not issued.

Variation (b). Second, the TIB-based mechanism can be put in place, similar to that described in Section 4.

Variation (c). Third, an instruction that acquired an early-released register does not have to stall at dispatch even if the consumers of the previous instance have not issued. Instead, this instruction can proceed all the way to the writeback stage, and can re-check the conditions just prior to writeback. If the consumers of the previous instance have already issued (the likelihood of which increases), then the writeback can be performed normally, otherwise the instruction will be re-issued. This scheme requires that the instruction is stored in the issue queue until it can safely writeback, and the mechanisms similar to those used to support replays in cases of load-hit mispredictions [18] can also be used here to replay the instructions. Note that even when an instruction has to be re-issued, its dependents can still execute without any delays provided that they obtain the value off the bypass network. Also in this scheme, the consumer counters are only decremented when the consumers writeback (to avoid storing erroneous information in the counters because of possible replays). Two instructions writing to the same physical register may be simultaneously present in the issue queue in this scheme. An additional bit is used for each issue queue entry to distinguish between these two instances.

Variation (d). Finally, we propose the most aggressive implementation of rename-based deallocation with AVD. If an instruction that acquired an early-released register is allowed to progress into the writeback stage even if the consumers of the previous instance are not issued, as explained in the above variation, then there is a possibility that a normally released physical register will become available by the time this instruction enters the writeback stage. In such a case, it is possible to remap the early-released register to a normally released register. While the instruction that acquired an early-released register re-checks the conditions just prior to writeback, in parallel to that it can also check if a normally released register is available. If the consumers of the previous instance are still not issued, and there is a normally released register available, then the instruction that acquired the early-released register can remap its destination to one of the normally released registers and carry on with the writeback operation. In such a case, this instruction does not need to be re-issued. To update the source tags within the issue queue, an update broadcast of a (old tag, new tag) pair, accompanied by the appropriate control signal, is needed. At first glance, one may assume that such an update broadcast may require the additional set of tag buses. However, since the tag buses are typically grossly underutilized [1], the existing set of wakeup buses is sufficient to accommodate all update broadcasts. Other studies [10] have also shown that such a broadcast-based remapping of registers is not difficult to perform. Of course the corresponding entries in the remap tables should also be updated. If remapping is not possible, then the scheme reverts to the mechanism described in variation (c) above.

6. Related work

Researchers have exploited the inefficiencies in register usage to reduce the number of registers in three major ways. One set of solutions delays the actual allocation of physical registers until the time that the result is written back [12],

[24], [28], [33]. The second set of solutions reduces the number of registers through the use of register sharing [2], [14], [15], [31]. The third set of techniques aim at reducing the register file pressure by using the early deallocation of physical registers [9], [23], [25], [26] and [27]. These techniques are close in spirit to our proposal, and in the subsequent paragraphs we describe them in detail.

In the Cherry scheme [23], a physical register is recycled if both the instruction that produces the physical register and all those that consume it have executed and are free of replay traps and are not subject to branch mispredictions. The scheme of [26] describes two techniques to release registers as soon as the processor knows that there will be no further use of them. Both schemes are less aggressive than what we propose and also do not support precise interrupts. In [9], the authors proposed schemes to release registers at the time of commitment. Again, techniques proposed herein are more aggressive in nature as we release registers at the time of writeback or even renaming.

The scheme of [27] proposes to release a register early if the register value has been produced, all consumers of the value have issued, the register has been redefined, and all branch instructions between the value producing instruction and the refiner have been resolved. Unfortunately, this technique does not support precise exceptions unless the precise state is explicitly checkpointed in regular intervals. The authors of [27] then discuss an alternative scheme to support precise interrupts in a less expensive manner. Note that this scheme is widely used in modern processors, and is exactly the baseline that we assumed for our work.

Alternative register file organizations (mainly using various forms of caching or banking) have also been explored for reducing the access time (which goes up with the number of ports and registers), particularly in wire-delay dominated circuits [4], [5], [7], [8], [17], [32]. In [22], register file usage was optimized using compiler support to exploit dead value information. Techniques exploiting narrow-width values were proposed to reduce the register file pressure [10] , [20] and to reduce the area, access time and energy consumption of the register file [19] and in [2]. The concept of non-blocking register file, where instructions waiting for the long- latency events (such as the caches misses) do not tie up physical registers, was introduced in [30].

7. Simulation methodology

For estimating the performance gains achieved by using the proposed scheme, we used a significantly modified version of the Simplescalar simulator [6] that implements realistic models for a datapath where a unified register file is used. The studied processor configuration is shown in Table 1. We assumed a pipeline with two stages for instruction fetch, two stages for register renaming, two stages dispatch. We used 16 SPEC 2000 benchmarks for our analysis. Benchmarks were compiled using the Simplescalar GCC compiler that generates code in the portable ISA (PISA) format. Reference inputs were used for all the simulated benchmarks. The results from the simulation of the first 1 billion instructions were discarded and the results from the execution of following 200 million instructions were used.

Table 1 Configuration of the Simulated Processor

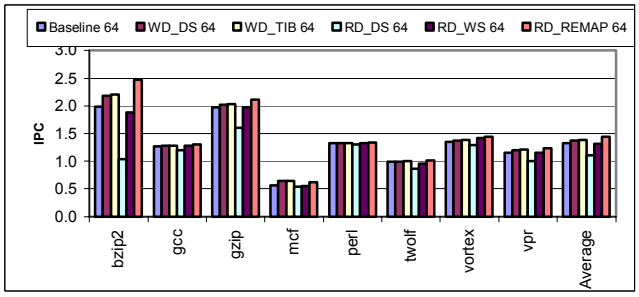
Parameter	***Configuration***
Machine width	4-wide fetch, 4-wide issue, 4 wide commit
Window size	64 entry issue queue, 64 entry load/store queue, 128–entry ROB
Registers	Varied as indicated in the text
Function Units and Latency (total/issue)	4 Int Add (1/1), 1 Int Mult (3/1) / Div (20/19), 2 Load/Store (2/1), 4 FP Add (2), 1FP Mult (4/1) / Div (12/12) / Sqrt (24/24)
L1 I–cache	32 KB, 2–way set–associative, 32 byte line, 2 cycles hit time
L1 D–cache	32 KB, 4–way set–associative, 32 byte line, 2 cycles hit time
L2 Cache unified	512 KB, 4–way set–associative, 128 byte line, 8 cycles hit time
BTB	1024 entry, 4–way set–associative
Branch Predictor	Combined with 1K entry Gshare, 10 bit global history, 4K entry bimodal, 1K entry selector
Memory	128 bit wide, 120 cycles first chunk, 2 cycles interchunk
TLB	64 entry (I), 128 entry (D), fully associative, 30 cycles miss latency

8. Results

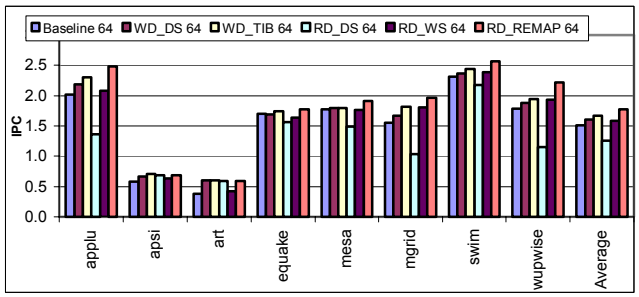
Figures 3 (a) and (b) present the IPC performance for the schemes described in this paper for integer and floating point benchmarks respectively. The results are presented for a setup with 64-entry integer and 64-entry floating point register files. Six bars are presented for each benchmark. The leftmost bar shows the performance of the baseline case. The next bar shows the performance of the scheme discussed in Section 3, which is called WD_DS (Writeback-time Deallocation with Dispatch Stalls). The next bar shows the performance of the scheme from Section 4, where the writeback-time deallocation is augmented with a 4-entry TIB. The three rightmost bars show the IPCs of the rename-time deallocation schemes, with dispatch stalls (RD_DS – variation (a) from Section 5), writeback stalls (RD_WS – variation (c) from Section 5) and finally with remapping (RD_REMAP – variation (d) in Section 5). We do not present results for variation (b) since the performance difference compared to (a) was minimal.

On the average across the benchmarks, WD_DS achieves 4% IPC gain for integer benchmarks and 12.6% for floating point benchmarks with an overall average of 8.3%, WD_TIB achieves 4.9% and 16.7% IPC gain for integer and floating point benchmarks respectively, the average speedup for WD_TIB across all benchmarks is 10.8%. The rename-time deallocation schemes without remapping perform much worse. The basic RD_DS scheme actually loses 13.5% and 6.7% in performance compared to the baseline case for integer and floating point benchmarks respectively, with an

overall average degradation of 10.2%, while the RD_WS scheme performs 0.6% worse than the baseline case for integer benchmarks and gains 6.2% for the floating point benchmarks on the average, across all benchmarks this scheme performs 2.8% better compared to the baseline machine.



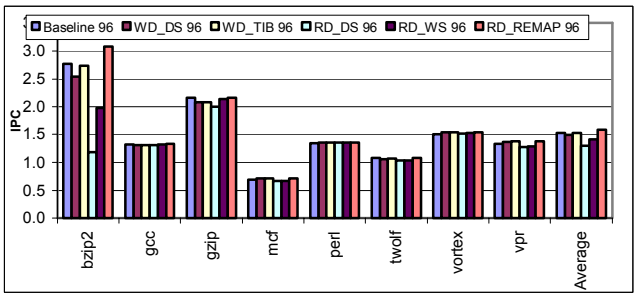
(a)



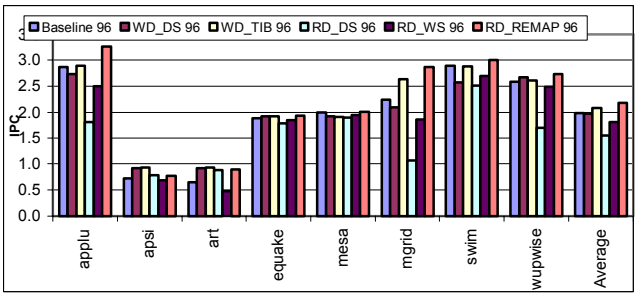
(b)

Figure 3. IPC performance of the proposed schemes for a setup with 64 entry register files - (a) Integer benchmarks, (b) Floating Point benchmarks

The reason why the rename-time deallocation schemes without remapping perform so poorly is due to the excessive number of stalls because of the very aggressive nature of register deallocations. The performance is worse than in the baseline case, because an instruction often acquires a "bad" register and has to stall for a large number of cycles, while in the baseline even if the instruction is stalled due to the lack of registers, chances are that some register will be freed up soon and the instruction dispatching will continue. Similar problems, albeit on a lesser scale, occur with the RD_WS scheme. Such stalls can be avoided if the writeback-time remapping (variation (d) in Section 5) is implemented. The right-most bar, labeled RD_REMAP, on Figure 3 shows the performance improvement when the remapping scheme is used with rename time deallocation. In this case, we see a significant performance improvement: 7.8% for the integer benchmarks, 21.5% for the floating point benchmarks on the average and 14.6% on the average across all benchmarks. The remapping scheme not only makes up for the performance losses seen with the other two rename time allocation techniques but it also outperforms the writeback-time deallocation techniques.



(a)

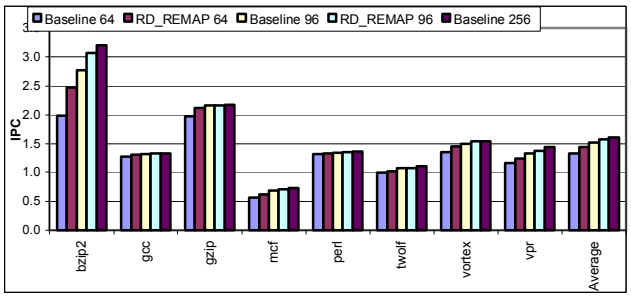


(b)

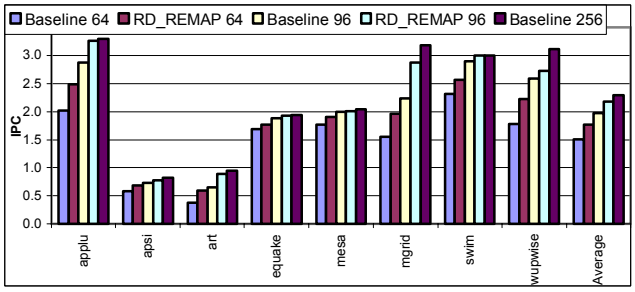
Figure 4 IPC performance of the proposed schemes for a setup with 96 entry register files - (a) Integer benchmarks, (b) Floating Point benchmarks

Figure 4 presents the same performance evaluation for a setup with 96 entry integer and 96 entry floating point register files. We can see that WD_DS scheme exhibits a slight slowdown (smaller than 1%) with respect to the baseline case for the integer benchmarks but it shows a speedup of for the floating point benchmarks with the overall average speedup of 2.6% across all benchmarks. The second writeback time deallocation scheme, namely WD_TIB shows only a slight speedup (0.4%) for the integer benchmark. This scheme shows much higher IPC gains for the floating point benchmarks; 10.9% on the average. Overall, across all benchmarks the speedup of the WD_TIB scheme is 5.7%. The rename time deallocation techniques without remapping again perform much worse than the writeback deallocation schemes. It can also be seen from the figure that the rename time deallocation scheme where remapping is performed achieves an average speedup of 2.7% and 12.6% for integer and floating point benchmarks respectively. The overall IPC gain across all benchmarks for this scheme is 7.5%.

Figure 5 depicts the IPC numbers for the baseline case and the RD_REMAP scheme with 64 and 96 entries register files as well as the baseline case with 256 registers (in our configuration this corresponds to having infinite number of registers). It can be seen from the figure that the RD_REMAP scheme comes within 2% and 5% of the baseline case with infinite number of registers for integer and floating point benchmarks respectively.



(a)



(b)

Figure 5 IPC performance of the baseline machine and the RD_REMAP scheme for several different register file sizes

Finally, Figure 6 compares the RD_REMAP scheme that is proposed in this paper to the Physical Register Inlining scheme of [20], which stores narrow-width results directly in the rename table, and the early register release scheme of [9] (Ergin_ICCD). We implemented all these schemes in our framework and therefore provide the performance comparison on an even footing. As the figure shows, the RD_REMAP scheme performs better than previous techniques on the average for both integer and floating point benchmarks. The RD_REMAP scheme results in a 10% higher speedup than the scheme of Ergin_ICCD on the average. The additional performance gains are due to the more aggressive nature of our schemes. While the scheme of [9] early deallocates registers at the time of instructions commitment, our techniques go much further than that. The Physical Register Inlining outperforms the RD_REMAP scheme proposed in this paper for some benchmarks (*gzip, perl, twolf, vpr, mesa, swim*) due to the high percentage of narrow-width values in these benchmarks, but RD_REMAP scheme performs 6% better on the average.

9. Concluding remarks

It is important to consider techniques for efficient use of physical registers in modern high performance processors, because even though smaller register files facilitate faster clocks, they limit the instruction level parallelism exploited in the processor. In this paper we proposed a series of techniques for early register deallocation based on the idea that a register value can be temporarily decoupled from the register address. Early register deallocation increases the effective register file size. We described several realizations of this idea and showed that our most aggressive design achieves about 14.6% IPC gains on the average across the SPEC benchmarks in a somewhat register-constrained datapath configuration. Our techniques also compare favorably against some previously proposed solutions. For example, the performance gains almost triple compared to a less aggressive scheme of [9]. We also show that our most aggressive technique outperforms the Physical Register Inlining scheme of [20] by 6% on the average. Finally, our techniques applied to 96-entry register files come as close as 3.5% of the performance of a machine with infinite number of physical registers.

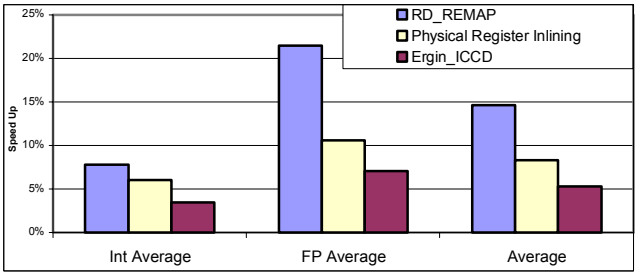


Figure 6. Speed-up comparison of the RD_REMAP scheme to Physical Register Inlining [20] and the scheme of Ergin_ICCD [9]

10. References

[1] Aggarwal, A., Franklin, M., Ergin, O., "Defining Wakeup Width for Efficient Dynamic Scheduling", in Proc. of ICCD, 2004.

[2] Aggarwal, A. Franklin, M. "Energy Efficient Asymmetrically Ported Register Files" in Proc. of ICCD 2003.

[3] Balakrishnan, S., Sohi, G., "Exploiting Value Locality in Physical Register Files", in Proc. MICRO-36, 2003.

[4] Balasubramonian, R., Dwarkadas, S., Albonesi, D., "Reducing the Complexity of the Register File in Dynamic Superscalar Processor", in Proc. of MICRO-34, 2001.

[5] Borch, E., Tune, E., Manne, S., Emer, J., "Loose Loops Sink Chips", in Proc. of HPCA-8, 2002.

[6] Burger, D. and Austin, T. M., "The SimpleScalar tool set: Version 2.0", Tech. Report, Dept. of CS, Univ. of Wisconsin-Madison, June 1997 and documentation for all Simplescalar releases.

[7] Butts J. A., G. Sohi "Use-Based Register Caching with Decoupled Indexing", in Proc. Of ISCA-31 2004

[8] Cruz, J-L. et. al., "Multiple-Banked Register File Architecture", in Proc. ISCA-27, 2000.

[9] Ergin O., et.al., "Increasing Processor Performance through Early Register Release", in Proc. of ICCD, 2004

[10] Ergin O., et.al., "Register Packing: Exploiting Narrow-Width Operands for Reducing Register File Pressure", in Proc. of .MICRO, 2004.

[11] Franklin, M., Sohi, G., "Register Traffic Analysis for Streamlining Inter-Operation Communication in Fine-Grain Parallel Processors", in Proc. of MICRO-25, 1992.

[12] Gonzalez, A., Gonzalez, J., Valero, M., "Virtual-Physical Registers", in Proc. of HPCA-4, 1998.

[13] Hinton, G., et.al., "The Microarchitecture of the Pentium 4 Processor", Intel Technology Journal, Q1, 2001.

[14] Hu, Z. and Martonosi, M., "Reducing Register File Power Consumption by Exploiting Value Lifetime Characteristics", in Workshop on Complexity-Effective Design (WCED), 2000.

[15] Jourdan, S.,et. al. ., “A Novel Renaming Scheme to Exploit Value Temporal Locality through Physical Register Reuse and Unification”, in Proc. of MICRO-31,, 1998.
[16] Kessler, R.E., "The Alpha 21264 Microprocessor", IEEE Micro, 19(2) (March 1999), pp. 24-36.
[17] Kim, N., Mudge, T., "Reducing Register Ports Using Delayed Write-Back Queues and Operand Pre-Fetch", in Proc. of ICS-17, 2003.
[18] Kim. I., Lipasti, M., “Understanding Scheduling Replay Schemes”, in Proceedings of HPCA, 2004.
[19] Kondo M. and Nakamura H. “A Small, Fast and Low-Power Register File by Bit-Partitioning”, in Proc.HPCA-11, 2005.
[20] Lipasti, M., et.al., “Physical Register Inlining”, in ISCA-31, 2004.
[21] Lozano, G. and Gao, G., "Exploiting Short-Lived Variables in Superscalar Processors", in Proc. MICRO-28, 1995.
[22] Martin, M., Roth, A., Fischer, C., “Exploiting Dead Value Information”, in Proc. of MICRO-30, 1997.
[23] Martinez, J., Renau, J., Huang, M., Prvulovich, M., Torrellas, J., "Cherry: Checkpointed Early Resource Recycling in Out-of-order Microprocessors", in Proc. of MICRO-35, 2002.
[24] Monreal, et.al., “Delaying Register Allocation Through Virtual-Physical Registers”, in Proc. of MICRO-32, 1999.
[25] Monreal, T., et.al., “Late Allocation and Early Release of Physical Registers”, IEEE Transactions on Computers, October 2004.
[26] Monreal, T., Vinals, V., Gonzalez, A., Valero, M. “Hardware Schemes for Early Register Release”, in Proc. of ICPP-02, 2002.
[27] Moudgill, et. al., "Register Renaming and Dynamic Speculation: An Alternative Approach", in Proc. of MICRO-26, 1993.
[28] Park, I., Powell, M., Vijaykumar, T., "Reducing Register Ports for Higher Speed and Lower Energy", in Proc. of MICRO-35, 2002.
[29] Ponomarev, D., Kucuk, G., Ergin, O., Ghose, K., "Reducing Datapath Energy Through the Isolation of Short-Lived Operands", in Proc. of PACT-12, 2003.
[30] Srinivasan S., et.al., “Continual Flow Pipelines”, in ASPLOS 2004.
[31] Tran, N., et.al., “Dynamically Reducing Pressure on the Physical Register File through Simple Register Sharing”, in ISPASS-2004.
[32] Tseng, J., Asanovic, K., "Banked Multiported Register Files for High Frequency Superscalar Microprocessors", in Proc. of ISCA-30, 2003.
[33] Wallase, S., Bagherzadeh, N., "A Scalable Register File Architecture for Dynamically Scheduled Processors", in Proc. PACT-5, 1996.
[34] Yeager, K., “The MIPS R10000 Superscalar Microprocessor", IEEE Micro, Vol. 16, No 2, April, 1996

Appendix A: Reconstructing the precise state

The finite state machine (FSM) diagram for updating the *ERR (Early-deallocated Register Reallocated)* and and the *SRW* (*Short-lived Register Written-back)* bits is shown in Figure 7. In Figure 7, the first bit in the circle is the *ERR* bit and the second bit is the *SRW* bit. If no instances of a register are alive, then the bits are “00”.

At register allocation time, the bits are not updated if they are “00” (*i.e.* no instance alive), else they are updated to “10” from “01” (one short-lived early-deallocated instance alive). Note that, when a register is allocated, the *ERR* and the *SRW* bits can only be either “00” or “01”. At writeback, if a register is not yet short-lived (the redefiner has not been renamed), nothing is done. Otherwise, if the bits are “00”, the register is deallocated and the bits are updated to “01”, and if the bits are “10”, they are updated to “11” without deallocating the register (ensuring that only two instances of a register are alive).

When the redefiner is renamed, the state is updated from “00” to “01” and from “10” to “11” if the register has been written back, else nothing is done. When the redefiner of a register commits, the bits can be “00” if the only instance of the register is not short-lived, “01” if the early-deallocated instance of the register has not been reallocated, “10” if the second instance of the register is not short-lived, or “11” if the second instance of the register is short-lived. The bits are updated to “00” from “01” and “10”, and to “01” from “11”, signifying the accurate condition of the register after the commitment of the redefiner. The register is deallocated if the bits are “11”, otherwise the register is not deallocated.

The FSM of Figure 7 significantly simplifies the handling of branch mispredictions and precise interrupts. When an exception occurs, we assume that it is handled when the instruction generating the exception is the oldest instruction in the pipeline. When handling an exception, if the *ERR* and *SRW* bits of a register are “11”, then the original value is restored from the shadow bitcells. If the bits are “01”, then the register is removed from the free list. For all the registers, the *ERR* and the *SRW* bits are modified to “00”, and the *Renamed* bits are reset. These updates ensure no early-deallocated and short-lived registers across exceptions.

When handling branch mispredictions in the traditional case, the ROB entries occupied by the instructions younger than the mispredicted branch are examined to recover the precise register renaming. In parallel, the *ERR* and the *SRW* bits are checked for the register assigned to the instruction and the register the instruction redefines. When an instruction is squashed, the bits for the register it redefines can be in any one of the four states. If the bits are “11”, they are updated to “10”, and if they are “01”, they are updated to “00”. Otherwise, the bits are not modified. In addition, the *Renamed* bit for the register the instruction redefines is *reset*.

When an instruction is squashed, the bits for the register assigned to it can only be “10” or “00”, because either the register is not short-lived, or it is short-lived and the redefiner has been squashed. If the bits are “10”, then it is not known whether the instruction, that was assigned the second instance of the register, has written back its result. To obtain this information, we use the *Written_back* bit of the registers (refer Section 3). If the *Written_back* bit of a register is “1”, then the original value is restored. Irrespective of the *Written_back* bit, the register is deallocated, the *Written_back* and the *Renamed* bits are set (because the earlier instance of the register has been written to and renamed), and the bits are updated to “01”. If the *ERR* and the *SRW* bits are “00”, the register is deallocated and the *Written_back* bit is reset. A register may be deallocated twice if two instructions using the same register are squashed. However, if a bit-vector is maintained to record the free registers, the bit for the register deallocated twice will be set twice, which will not cause any problems. Note that, when squashing instructions, the register operands of the instructions are also used to decrement the consumer counters for the registers.

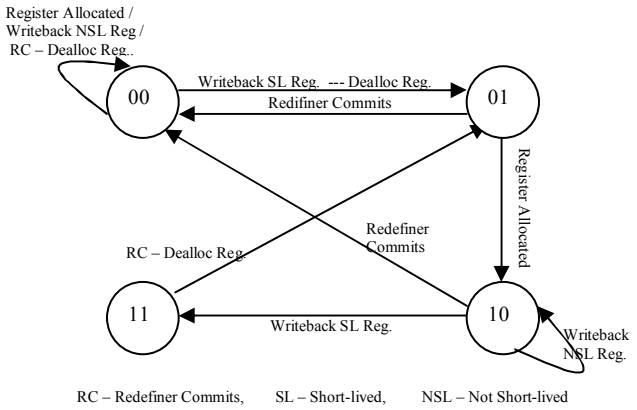


Figure 7. Finite state diagram for the ERR and SRW bits of one register; as the register is allocated and written back, and the redefiner of the register is committed

Handling branch mispredictions in a checkpoint-based recovery mechanism is even easier. When a checkpoint of the rename map table is created, the *ERR*, *SRW*, *Written_back*, and *Renamed* bits for all the registers are also checkpointed. On recovery, all the bits are restored. The *Written_back* bits of instructions that writeback after the checkpoint has been created will still be zero. However, this does not affect the correctness of this scheme.