Increasing Processor Performance Through Early Register Release

Oguz Ergin, Deniz Balkan, Dmitry Ponomarev, Kanad Ghose
Department of Computer Science
State University of New York, Binghamton, NY 13902-6000
e-mail:{*oguz,dbalkan,ghose,dima}@cs.binghamton.edu*

Abstract

Modern superscalar microprocessors need sizable register files to support large number of in-flight instructions for exploiting ILP. An alternative to building large register files is to use smaller number of registers, but manage them more effectively. More efficient management of registers can also result in higher performance if the reduction of the register file size is not the goal.

Traditional register file management mechanisms deallocate a physical register only when the next instruction with the same destination architectural register commits. We propose two complementary techniques for deallocating the register immediately after the instruction producing the register's value commits itself, without waiting for the commitment of the next instruction with the same destination. Our design relies on the use of a checkpointed register file (CRF), where a local shadow copy of each bitcell is used to temporarily save the early deallocated register values should they be needed to recover from branch mispredictions or to reconstruct the precise state after exceptions or interrupts. The proposed techniques outperform the previously proposed schemes for early deallocation of registers. For the register-constrained datapath configurations, our techniques result in up to 35% performance increase with 23.3% increase on the average across SPEC2000 benchmarks.

1. Introduction

Dynamic superscalar processors extract instruction-level parallelism from sequential code by maintaining a large window of instructions and issuing ready instructions for execution, possibly out of program order. Sizable physical register files are mandated in such designs to support large instruction windows, as every in-flight instruction with a destination register is allocated a new physical register. Most recent implementations of superscalar CPUs use unified register files for holding both committed and speculative (non-committed) register values within a single RAM structure. A back-end register mapping table, updated at the time of instruction commitment, is typically used to point to the most recently committed instance of each architectural register. The information stored in this table allows the reconstruction of the precise state on interrupts or exceptions and also assists in the rapid recovery from branch mispredictions.

Traditional allocation and deallocation mechanisms, associated with the unified register files are too conservative – they are designed to support the worst-case scenarios, which rarely occur in practice. A new physical register is allocated for the destination of a new instruction at the time of dispatch and this register remains allocated till the next instruction writing to the same architectural register commits. This guarantees that if an instruction producing a later instance of the architectural register is squashed out of the pipeline, the earlier instance is available and can be resurrected to reconstruct the precise register state. While simple to implement, such a register deallocation mechanism results in a situation, where a lifetime of a physical register significantly exceeds the lifetime of the associated instruction – the register remains allocated well beyond the point of instruction commitment. Consequently, large register files are needed to avoid stalls in instruction dispatching due to the lack of free physical registers. Such large register files result in high access delay and power consumption. In addition, the register files in future wide issue machines also need to be highly-ported, which further exacerbates the situation and also increases the overall design complexity.

An alternative to building large register files is to use smaller number of registers, but manage them more effectively. More efficient management of registers can also result in higher performance if the reduction of the register file size is not the goal. Researchers have addressed the inefficiencies in register usage to reduce the number of registers by using late register allocation [7, 15], early deallocation [10, 11, 12] and register sharing [8, 4, 14]. We defer the detailed discussion of these related efforts to Section 4. The register lifetime analyses in Figure 1 show that the number of cycles between the

result writeback and the register deallocation is 36 cycles on average, which is significantly higher than the number of cycles between the register allocation and the result writeback into this register (16 cycles). Consequently, it is relatively more important to pursue techniques for early register deallocation.

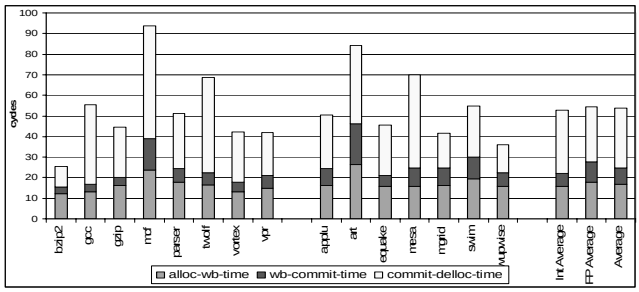

Figure 1 - Register Lifetime

Our first scheme releases a physical register allocated for the destination of an instruction immediately after the instruction commitment, under certain conditions. Specifically, if a destination register allocated for an instruction is renamed before the instruction commits and all potential consumers of the value have started the execution (i.e. obtained the value), then the register can be released and reallocated for future instructions. Since branch mispredictions can result in the need to restore the values of some deallocated registers, the values of the registers that are deallocated immediately after its commitment have to be saved. We will refer to these registers as "early deallocated" registers. To support this capability, we use a locally checkpointed register file (CRF) where each bitcell has a locally connected shadow copy. The value of a register can be saved in the shadow bits of the register in a single parallel step. On branch mispredictons, the saved values can be restored from the checkpoint of the shadow bits, as needed. Our design does not introduce the extra ports to the register file, the communication for saving and restoration of values occurs directly between a bitcell and its shadow. We also introduce a complementary technique that increases the performance by early releasing registers that were not deallocated by the first scheme.

The rest of the paper is organized as follows. We present the motivation for this work, define some terms and also describe the CRF technique in Section 2. Our designs for the early register release are presented in Section 3. Section 4 presents the related work. Section 5 describes our simulation methodology followed by the simulation results in Section 6. Finally, we offer our concluding remarks in Section 7.

2. Motivation, Definitions and Checkpointed Register Files

This work was primarily motivated by the fact that a large percentage of generated register values in a datapath are short-lived. Researchers have used the term "short-lived" in many different contexts [6, 9, 13]. For example, [9] identified a value as short-lived if it is exclusively consumed during its residency in the reorder buffer. (The study of [9] used a P6-style datapath, where the generated results values are first written into the reorder buffer and then later moved to the architectural register file during instruction retirement). In [13], the authors defined the value to be short-lived if the destination architectural register used to hold the value is renamed (redefined) by the time the value-producing instruction reaches the *write-back* stage. In this paper, we call the value short-lived, if the corresponding destination register is renamed by the time the value producing instruction reaches the *commit* stage. At first glance, the difference between the two definitions may seem to be minor, but in reality an instruction can spend many cycles between its write-back and commitment, as write-backs occur out-of-order, but commits are performed strictly in program order. Consequently, a larger percentage of values can be identified as short-lived using the new definition.

Figure 2 shows the percentage of values that are short-lived according to our definition. The results are shown for different physical register file sizes, as detailed in the legend of the figure. As depicted, the percentage of short-lived values increases for larger register files. When the register file size is small, it is often the case that an instruction redefining an architectural register had not yet been dispatched by the time the instruction producing the previous instance of the same register committed. This primarily happens because of the frequent pipeline stalls due to the lack of physical registers.

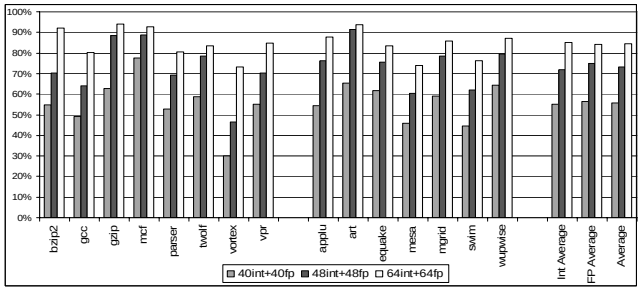

Figure 2 - Percentage of Short-lived Values

The basic idea behind our designs is to deallocate a physical register allocated to hold a short-lived value immediately after the instruction producing the value commits. The only additional condition that we impose is that all consumers of the value must commence execution before deallocation can occur. Notice that the presence of the unresolved branch instructions between the instruction producing the short-lived value and the instruction that redefines the same architectural register (called redefiner in the rest of the paper) is not a condition for early register deallocation. Consequently, the values kept in the deallocated registers may need to be restored as a result of branch mispredictions when some redefiners

are squashed. Therefore, a mechanism is required to make the early deallocated register available for new allocations, but at the same time somehow preserve the value that was kept in this register till the time when this value can be discarded completely (which would happen at the time of the redefiner's commitment, just like in traditional designs)

To accomplish this goal, we use a register file design with an embedded checkpoint. The schematic of such Checkpointed Register File (CRF) bitcell is shown in Figure 3. Here, each traditional register file bitcell is backed-up by a pair of cross-coupled inverters (I4 and I5) which are connected to the main bitcell using pass transistors (T5 and T6). When the *Checkpoint* signal rises, the contents of every bitcell are simply copied to the shadow cells. To recover, the contents of the shadow cells are copied back to the main storage when the *Recover* signal rises. As seen from Figure 3, the additional area required by the shadow cell (the area in the grey box) is virtually independent of the number of register file ports. In fact, the area overhead of the shadow bits becomes relatively smaller as the number of register file ports increase.

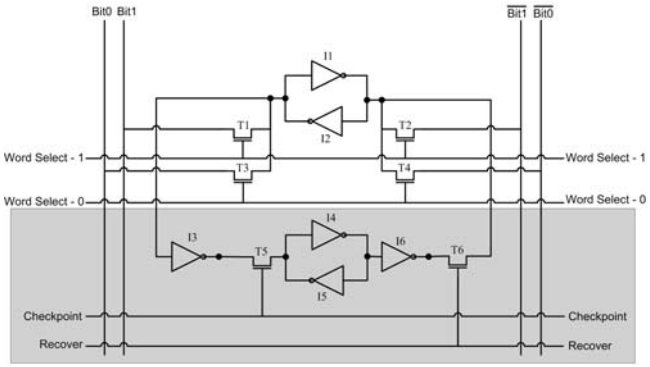

Figure 3 - A Dual-ported CRF bitcell

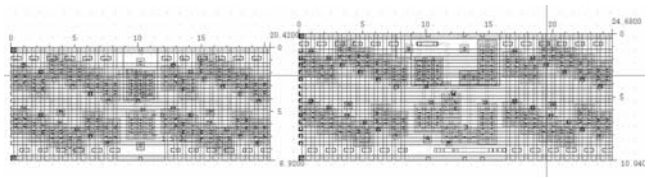

Figure 4 - Layouts of a Register File Bitcell

In our design, the short-lived values whose consumer count is zero are checkpointed at the time of commitment, using the logic shown in Figure 3. At the same time, the main registers holding these values are immediately released for future allocations. Notice that a separate checkpoint and recover signals are needed here for each register, as the saving and the restoration of the register values are performed on an individual basis, as we explain in detail in Section 3.

In our implementation of the circuit from Figure 3, we removed the inverters I3 and I6 to save the layout area. This can be done through appropriate transistor sizing. Figure 4 shows the CMOS layouts of a traditional 6-transistor SRAM bitcell (left portion of Figure 4), and a traditional SRAM bitcell with the embedded shadow bitcell that implements checkpoint, as depicted at Figure 4. For both layouts, we used 12-ported bitcells. As can be measured from the figure, the resulting bitcell area increase is about 26.5%. This area increase is not proportional to the number of register ports, as can be easily seen from Figure 4. 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 0.5% for the layouts that were designed and simulated. There is also a similar minimal impact on the delay of the word select line during the normal course of read and write accesses.

3. Schemes for Early Register Release

In this section, we describe two complementary schemes for early register deallocation that rely on the hardware support described above.

3.1. Scheme 1

Our first scheme releases a physical register allocated for an instruction if the following two conditions are true: (1) the value produced by the instruction is short-lived, and (2) all potential consumers of this value have started execution. The percentage of registers that can be early released in this manner strongly depends on the register file size, as well as on a few other parameters, as we detail in the results section. Note that although the register is released at the time of commitment, the retirement rename table is still updated, just as in traditional designs.

To check for condition (1) above, we maintain a bit vector called *Redefined*, with one bit for each physical register. In the register renaming stage, each value producing instruction sets the *Redefined* bit of the physical register, which was previously mapped to its destination architectural register. Each instruction checks the *Redefined* bit of the physical register assigned to its destination at the time of commitment. This bit is reset when the corresponding physical register is deallocated.

To detect the second condition for early register deallocation, we maintain consumer counters for each physical register to keep track of how many consumers have not yet read the value of the register. The counters are incremented at the time of renaming, and they are decremented when instructions begin execution. The consumer counters are also accordingly adjusted in the course of branch misprediction handling. A register R, allocated to hold a result of instruction I, is deallocated at the time of I's commitment if and only if *Redefined*[R] = 1 and *Consumer_Counter*[R] = 0.

Note that the absence of the unresolved branches between the instruction producing a short-lived value and its redefiner is not a condition for early release (adding this condition would be too restrictive and would involve significant implementation complexity.) We discuss the handling of branch mispredictions later in this section. For now, it is sufficient to understand that reconstruction of the precise register state is possible, since the values of early-deallocated registers are not completely discarded, but instead saved in the shadow bitcells of the CRF.

One potential problem with the proposed scheme is that the deallocation of the same register can occur twice, thus creating inconsistencies in the register free list and leading to erroneous results. Consider a scenario, where three instructions – A, B and C - follow each other in program order, such that A and B write to the same architectural register, (B is a redefiner for A). Assume that a physical register, assigned for the instruction A is early deallocated at the time of A's commitment and then reallocated to the instruction C before B commits. In this case, instruction B, when it eventually commits, will try to deallocate the register initially assigned to A, as it has no knowledge that the register had already been deallocated. The net effect of such actions will be the deallocation of a register, which was assigned to a younger instruction (C in this case) leading to the incorrect program behavior. The root of the problem is that a physical register is deallocated twice – once in the course of early release and once in the course of the regular commitment activity of the redefining instruction.

We avoid this scenario by associating a single bit called *don't_deallocate*, with each physical register. When a register holding a short-lived value is released at the time of commitment, its *don't_deallocate* bit is set. When the redefiner commits and the *don't_dealloate* bit of the register that it normally deallocates (obtained from the commitment rename table) is set, then no deallocation occurs, as that register has already been released early.

We now discuss how the branch mispredictions are handled in this scheme. A branch misprediction can cause the redefining instruction to get squashed. Consequently, the value of the early deallocated register needs to be restored in this case. This mechanism is implemented as follows. At the time of renaming, each instruction saves the old mapping of its destination register within the ROB entry. This information can be easily obtained from the rename table and some implementations of superscalars already perform similar action for branch misprediction recovery. In our scheme, when an instruction is flushed from the pipeline as a result of a misprediction, it checks the *don't_deallocate* bit associated with the previous mapping of its destination register (which, as explained above, is saved within the ROB entry of the instruction in question). The setting of that bit indicates if the previous instance of the destination architectural register was early deallocated. If so, the previous value is restored from the shadow cells and the *don't_deallocate* bit is reset. Also, the physical register whose value was restored is removed from the free list.

The above solution still, however, has a subtle problem. We illustrate this problem below using a short code fragment. Consider the following sequence of instructions, where only the destination register of each instruction is shown for simplicity. Both the original and the renamed codes are included. Note that the particular instruction mnemonics are not important for the purposes of our discussion: they are shown merely for the sake of readability.

```
Original code          Renamed code
  I1: ADD r1             ADD p2
      ……..               ……..
  I2: SUB r1             SUB  p17
      ……..               ……..
  I3: XOR r5             XOR p2
  I4: BRANCH
  I5: NOR r5             NOR p19
```

Figure 5 - The Example Code Sequence

Assume that the value produced by the instruction ADD is identified as short-lived and physical register p2 is released to the free list at the time of ADD's commitment. Simultaneously, the value produced by the ADD is saved in the shadow bitcells of register 2 and *don't_deallocate*[2] bit is set to 1 to prevent the instruction SUB (which is a redefiner for ADD) from duplicate deallocation of register 2.

Further assume that after physical register 2 has been released, it was reallocated to another instruction (XOR in this case). After that, the branch instruction (I4) and XOR's redefiner (NOR) were dispatched. Next, the branch I4 was mispredicted. In the course of branch misprediction handling, as explained above, the instruction NOR checks the value of *don't_deallocate[2]* bit, because physical register 2 is a previous mapping of NOR's destination architectural register (r5). The problem is that this bit was set not by the XOR instruction (as the NOR instruction thinks), but by the ADD instruction. Now, if the NOR instruction blindly uses the value of *don't_deallocate[2]* and restores the previous value of physical register p2 from the shadow bitcells of the CRF, then the value of register p2 produced by the XOR instruction will be overwritten, resulting in the incorrect behavior of the program. The bottom line is that in the situation illustrated in this example the NOR instruction should not restore the previous destination architectural register value, as the previous producer of that register (the XOR) has not yet committed. This problem arises due to the simultaneous presence of two instructions (I2 and I5 in this example), which are unmapping the same physical register (p2 in this example). Notice that traditional register management mechanisms do not suffer from such problem.

One can think of several ways to overcome the aforementioned deficiency. One solution would be to associate the *don't_deallocate* bits with the ROB entries,

rather than with physical registers. While conceivable, this solution can result in a prolongation of the cycle time, as the ROB entries of the two instructions producing consecutive values of the same architectural register would essentially have to cross-reference each other.

Instead, we propose a more elegant mechanism to handle the above phenomenon. A simple 2-bit counter can be associated with each physical register to count the number of its in-flight redefiners. When a newly dispatched instruction redefines an architectural register, the counter associated with the previous mapping is incremented. When an instruction commits, the counter associated with the previous mapping of the destination register (as obtained from the retirement rename table) is decremented. For example, in the scenario considered in Figure 5, the counter associated with register p2 has the value of 2, it was incremented once during dispatch of I2 and once during dispatch of I5. A two-bit counter is sufficient, because the number of in-flight redefiners of the same physical register never exceeds 2 (for example, XOR cannot early release p2 before SUB commits).

When an instruction is flushed from the pipeline, it checks the value of this counter associated with the previous mapping of its destination register, along with the *don't_deallocate* bit. The restoration of the previous register value from the CRF checkpoint occurs only when the *don't_deallocate* bit is set and the value of the counter is 1. For example, in the scenario of Figure 5, the instruction NOR will not restore the value of p2, because the value of the counter associated with p2 is 2.

To summarize, the following actions are performed in various pipeline stages to support our design. Assume that P is the destination physical register of the instruction, R is the destination architectural register and K is the previous mapping of R.

Register Renaming:
(1) The consumer counters corresponding to each of the source operands is incremented.
(2) The *Redefined* bit of physical register K is set.
(3) The *don't_deallocate* bit of K is set to 0.
Commitment:
(1) If *Redefined*[P] = 1 and *Consumer_Counter*[P] = 0, the generated value is saved in the backup cells of the CFR and P is deallocated and the *don't_deallocate* bit of P is set to 1.
(2) If the *don't_deallocate* bit of K is set, physical register K is not deallocated. Otherwise, it is added to the free list.
Issue:
(1) The consumer counters corresponding to the instruction's source operands are decremented.

3.2. Scheme 2

The first condition for early register release in Scheme 1 mandates that the register can only be deallocated at the time of commit when the redefining instruction has been already dispatched into the pipeline. We notice, however, that it is frequently not the case for small register files. In other words, the percentage of short-lived values drops significantly when the size of the register file decreases. We also noticed that in many situations, even though the redefining instruction is not yet renamed, the consumer counter of the register is zero at the time of commitment. This observation motivated our second scheme.

Specifically, we propose to deallocate the registers holding committed values when the redefining instructions enter the pipeline, instead of waiting for the redefining instruction to commit, as is the case with traditional register management mechanisms. The only additional condition that needs to be imposed is that the consumer counter is zero. The same hardware support in the form of the CRF and a few bit-vectors, as described in Sections 2 and 3, can be used. The only additional logic that is needed here is in the form of the *Committed* bit vector, with one bit for each physical register. An instruction sets the Committed bit of its destination physical register when it commits and the register is not early deallocated. The bit is reset when the physical register is released (either early or in a regular manner). Each new instruction undergoing renaming checks the Committed bit associated with old mapping of its destination architectural register. If the bit is set, and the consumer counter of that register is zero, the register can be freed up immediately. While this second technique is of little use for reasonably large register files (where the percentage of short-lived values is large), it provides significant performance improvement for small-sized register files.

4. 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 [7, 15]. As a result, the register pressure is reduced and the effective size of the register file increases. These schemes can be used in conjunction with the techniques proposed in this paper. The drawback of delayed register allocation is in some design complexity, which stems from the need to maintain several levels of register maps.

The second set of solutions reduces the number of registers through the use of register sharing. The technique of [8] relies on the fact that duplicate values are produced with a high frequency within a small segment of the dynamic instruction stream. The additional complexity required to keep track of the duplicate values is quite significant. In [4] and [14], simpler techniques are proposed which only eliminate the duplication of two values in the register file - "0" and "1", yet allow to reap most of the benefits of the scheme proposed in [8].

The third set of techniques aim at reducing the register file pressure by using the early deallocation of physical registers [10, 11, 12]. These techniques are close in spirit to our proposal, and in the subsequent paragraphs we describe them in detail.

In Cherry scheme [10], 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. To reconstruct the precise state in cases of exceptions or interrupts, the scheme relies on periodic register file checkpointing. In section 6, we provide a quantitative comparison of our techniques with the Cherry scheme.

The scheme of [12] describes two techniques to release registers as soon as the processor knows that there will be no use of them. Their first scheme identifies the last user and the next version of a particular physical register. Provided that there are no pending branches waiting for verification between the last user and the next version, as soon as the next version is decoded the corresponding physical register is released if the last user has already committed or an early release is scheduled for the time when the last user commits. As our scheme does not have to wait for the commitment of the last user to release the register but instead releases the register at the time when the value-producing instruction itself commits, the register deallocation in our scheme occurs earlier than in the scheme of [12]. Consequently our performance gains are higher. According to the results presented in [12] their scheme achieves the performance gain of 5% for the integer programs and 9% for floating point programs for the configuration with 40 registers. Our performance gains for the same configuration are 20% and 26% for integer and floating point programs respectively (as presented in Section 6). The second scheme of [12] is an extension of the first scheme, where the condition of not having an unresolved branch between the last user and next version is removed by using a series of queues and additional logic. This scheme does not perform any better than ours as the absence of branches is not a condition in our schemes in the first place, but the register deallocation still occurs earlier in our designs. The performance improvements of our scheme compared to the two schemes of [12] are higher, especially when a smaller register file is used.

The scheme of [11] 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. Just like in the first scheme of [12] the additional condition on the absence of unresolved branches limits the effectiveness of this approach. Unfortunately, this technique does not support precise exceptions, so we did not directly compare our results against the results of [12]. In the same paper, the authors describe a simplified scheme which supports precise exceptions, which serves as a baseline case for our analysis.

Alternative register file organizations (mainly using various forms of caching) 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 [2, 3, 5]. A large number of solutions have also been proposed for reducing the register file energy consumption. Since energy optimization is not the goal of this paper, we do not discuss those schemes.

5. Simulation Methodology

For estimating the performance of our techniques, we used a significantly modified version of the Simplescalar simulator [1]. We implemented separate structures for the reorder buffer, issue queues and physical register file. All aspects of the state restoration on mispredictions were simulated accurately. The studied processor configuration is shown in Table 1. We simulated a subset of SPEC 2000 benchmarks, including both integer and floating point codes. Benchmarks were compiled using the gcc compiler that generates code in the portable ISA (PISA) format. Benchmarks were compiled with –O2 optimizations. 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 the 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, 32 entry load/store queue, 96–entry ROB
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, 60 cycles first chunk, 2 cycles interchunk
TLB	64 entry (I), 128 entry (D), fully associative, 30 cycles miss latency

6. Experimental Results

6.1 Evaluation of Scheme 1

We first evaluate the percentage of register values that can be early released using Scheme 1 and also estimate the number of bits needed to represent consumer counters.

Figure 6 shows the percentage of early deallocated registers for a configuration with 64 integer and 64 floating point registers. The first bar shows the percentage of short-lived values and the subsequent bars depict the percentage of early released registers for the various sizes of consumer counters. We assume that if the counter

overflows, the corresponding register is not early released. As seen from the figure, more than 70% of the registers can be early released on the average, and 2-bit wide consumer counters are sufficient in most cases. For a configuration with 48 registers, about 57% of the register values are early deallocated, and for a configuration with 40 registers the percentage goes down to 36% (results of Figure 1 can be used to understand this phenomenon.)

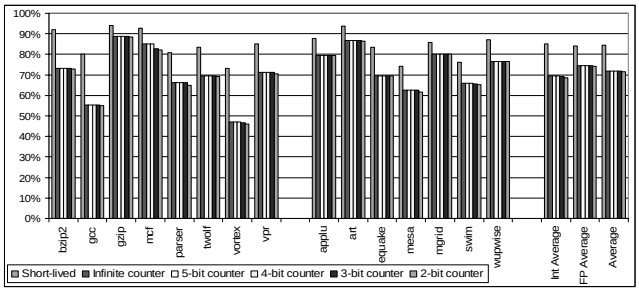

Figure 6 - Percentage of Early Released Registers

Figure 7 depicts the IPC gains achieved by Scheme 1 as compared to the baseline processor. Results are presented for three different configurations of the register files. The leftmost bar shows the speedup for the configuration with 40 integer and 40 floating point physical registers. The next bar shows the configurations with 48 integer and 48 floating point registers and the rightmost bar show the configuration with 64 integer and 64 floating point registers. We understand that having just 40 registers may be too low in the configuration considered, but the results are presented with the purpose of estimating the performance impact of our schemes on a register-constrained datapath.

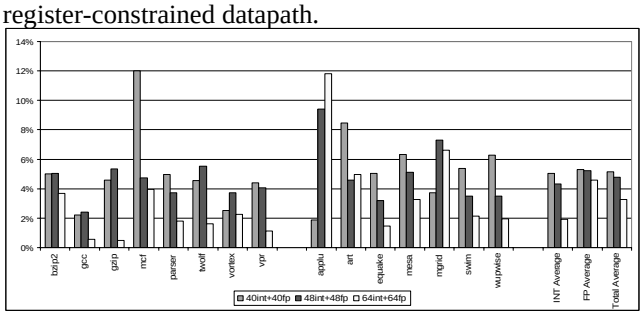

Figure 7 – Speedup of Scheme 1 Compared to the Base Case

The speedups shown in Figure 7 vary considerably among the benchmarks as the register file changes. Some benchmarks (*mcf, mesa, swim, wupwise, equake, parser and vpr*) show higher performance increase with smaller register files as one would expect since the register file is a more significant bottleneck when smaller register files are used. However, other benchmarks (*applu, mgrid, bzip2, gcc, gzip, twolf, mgrid, art, vortex*) show the opposite behavior, at least for some configurations. In the case of *applu* the speedup consistently increases with larger register files. This can be explained by examining the statistics of Figure 1. Although with larger number of registers, the performance is less sensitive to the size of the register file, significantly higher percentage of values can be considered for early deallocation, as the percentage of the short-lived values increases. On the average, the speedups are 5%, 4.8% and 3.2 % for 40, 48 and 64 registers respectively.

6.2 Evaluation of Scheme 2

Figure 8 shows the performance increase achieved by Scheme 2. Results show a consistent picture across all of the benchmarks with *mcf* being the only exception. As seen from the figure, larger performance gains are achieved for the smaller register files. On the average, the performance increase is 12.9%, 6.6% and 1.7% for 40, 48 and 64 registers respectively.

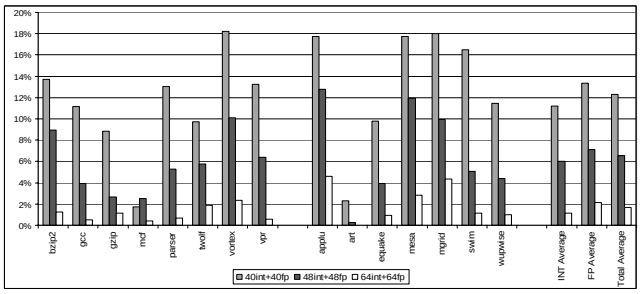

Figure 8 - Speedup of Scheme 2 Compared to the Base Case

To summarize, Scheme 2 is particularly effective for small register files and provides little benefit when a larger number of registers is in use.

6.3 Combined Evaluation of Schemes 1 and 2

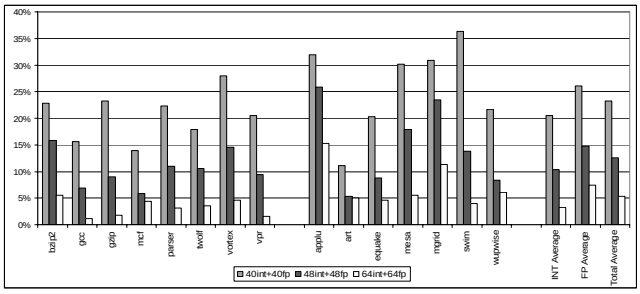

Figure 9 – Speedup of Combined Scheme Compared to the Base Case

Figure 9 shows the speedups obtained with the combined scheme. As shown, when both schemes are combined we can achieve speed-ups of up to 36.4% (for swim, using 40 registers.) The average speedups are 23.3%, 12.6% and 5.3% for 40, 48 and 64 registers respectively. Note that the speedups we observe when both schemes are used together are higher than the sum of the individual speedups of Scheme 1 and Scheme 2 for most of the benchmarks. This is because both schemes have a positive influence on each other. In particular, the early release of physical registers as a result of applying Scheme 2 decreases the number of pipeline stalls. Consequently, more instruction can be renamed, thus redefining some of the registers which wouldn't be

redefined otherwise, effectively increasing the number of short-lived values which can be directly exploited by Scheme 1.

6.4 Comparison with Previous Work

Finally, we compare the results of our schemes against the work of [10], where the Cherry Scheme is described. A qualitative comparison with Cherry as well as with other schemes for early register release is given in section 4. The performance benefits of the Cherry Scheme come from early deallocation of registers as well as from the optimizations performed on the load/store queue. For objective comparison, we only implemented the register optimizations of the Cherry Scheme. The performance results are shown in Figure 10 for various register file sizes.

As shown, our techniques consistently outperform the early register deallocation component of the Cherry Scheme for all simulated register file sizes. For 40 registers, our combined scheme achieves 23.3% speedup on the average whereas the Cherry Scheme only improves the performance by 12.2%, and for 64 registers Cherry achieves a performance gain of 2%, whereas our scheme improves the performance by 5.3%.

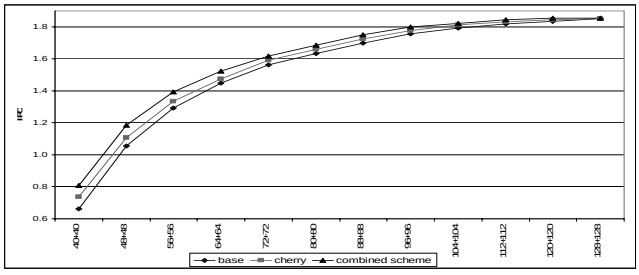

Figure 10 - Comparison of Cherry and Combined Scheme

7. Concluding Remarks

We presented two complementary techniques for early deallocation of physical registers in a superscalar processor. Our first technique releases a physical register immediately after the instruction commitment, without waiting till the next instruction writing to the same destination register commits. Our second technique further improves the performance by deallocating the committed instance of a register when the instruction producing the next instance is renamed. The combination of the two schemes results in the performance gains of 23.3%, 12.6% and 5.3% for 40, 48 and 64 registers respectively on the average across simulated SPEC2000 benchmarks.

8. Acknowledgements

We thank Matt Yourst for his help in developing the simulation environment. This work is supported in part by DARPA through contract number FC 306020020525 under the PAC--C program, the NSF through award no.MIP 9504767 &EIA 9911099, and by IEEC at SUNY--Binghamton.

9. References

[1] 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 (through version 3.0).

[2] Balasubramonian, R., Dwarkadas, S., Albonesi, D., "Reducing the Complexity of the Register File in Dynamic Superscalar Processor", in *Proc. of the Int. Symposium on Microarchitecture (MICRO-34)*, 2001.

[3] Borch, E., Tune, E., Manne, S., Emer, J., "Loose Loops Sink Chips",in *Proc. of Int. Conf. on High Perf. Computer Architecture (HPCA-8)*, 2002.

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

[5] Cruz, J-L. et. al., "Multiple-Banked Register File Architecture", in *Proc. Intl. Symp/ on Comp. Architecture (ISCA-27)*, 2000, pp. 316-325.

[6] Franklin, M., Sohi, G., "Register Traffic Analysis for Streamlining Inter-Operation Communication in Fine-Grain Parallel Processors", in *Proc. of Int. Symp. on Microarchitecture (MICRO-25)*, 1992.

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

[8] Jourdan, S., Ronen, R., Bekerman, M., Shomar, B. and Yoaz, A., "A Novel Renaming Scheme to Exploit Value Temporal Locality through Physical Register Reuse and Unification", in *Proc. of MICRO-31*, 1998.

[9] Lozano, G. and Gao, G., "Exploiting Short-Lived Variables in Superscalar Processors", in *Proc. of MICRO-28*, 1995, pp. 292-302.

[10] 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.

[11] Moudgill, M., Pingali, K., Vassiliadis, S., "Register Renaming and Dynamic Speculation: An Alternative Approach", in *Proc. of MICRO-26*

[12] Monreal, T., Vinals, V., Gonzalez, A., Valero, M. "Hardware Schemes for Early Register Release", in *Proc. of ICPP-02*, 2002.

[13] Ponomarev, D., Kucuk, G., Ergin, O., Ghose, K., "Reducing Datapath Energy Through the Isolation of Short-Lived Operands", in *Proc. of PACT-12*, 2003.

[14] Tran, N., et.al., "Dynamically Reducing Pressure on the Physical Register File through Simple Register Sharing", in *Proc. of Int. Symp. on Performance Analysis of Systems and Software (ISPASS-2004)*, 2004.

[15] Wallase, S., Bagherzadeh, N., "A Scalable Register File Architecture for Dynamically Scheduled Processors", in *Proc. of PACT-5, 1996*.