

Selective Writeback: Improving Processor Performance and Energy-Efficiency

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

Abstract

A significant fraction of the result values in today's superscalar microprocessors are delivered to their consumers via forwarding and are never read out from the destination registers. Such transient values are kept in the register file solely for the purpose of recovering the processor state on interrupts or exceptions. In this paper, we propose a simple technique to identify such transient register values and avoid their writebacks into the register file. Our scheme results in significant performance improvement, as high as 40% for some benchmarks and 12% on the average because the register file is utilized more efficiently. Energy savings of 27% within the register file are also achieved because much fewer writes to the register file are performed.

1. Introduction

In modern processors, speculative state is typically embodied in physical register files, where a physical register allocated for the destination of an instruction is deallocated only when the next instruction writing to the same architectural (logical) register commits. Such a conservative register management guarantees that until all instructions between the two consecutive definitions of the same architectural register commit, the earlier definition is available and can be resurrected should the later definition be squashed as a result of a branch misspeculation, exception or interrupt. Many recent superscalar processors, such as the Intel's Pentium 4 [11], MIPS 10000 [27] and Alpha 21264 [14] implement the register files in this manner. While significantly simplifying the recovery to a precise state, this arrangement increases the register pressure and effectively mandates the use of larger register files if pipeline stalls due to the lack of physical registers are to be avoided.

Large register files with a multi-cycle access time complicate the design of the bypass networks, as multiple bypass stages are required to provide uninterrupted supply of the source operands. Furthermore, the increased number of pipeline stages in the branch resolution loop negatively affects the overall

instruction throughput. Finally, large register files also dissipate more power. The power dissipated in the register file can be anywhere between 10% and 25% of the total chip power [3, 30]. The situation is further exacerbated in the SMT processors, where the pressure on the register file is increased and larger physical register files are needed to support multiple thread contexts.

Several optimizations, primarily targeting the reduction of the register file size through the more efficient use of registers have been recently proposed. Delayed register allocation schemes [9, 26] avoid tying up destination physical register between the time of instruction dispatch and instruction writeback by allocating physical register only at the time of writeback and using separate tags to maintain the data dependencies.

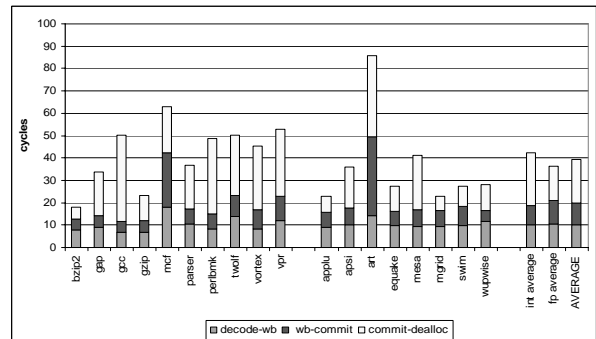


Figure 1 - Register Lifetimes

As the number of cycles between the result writeback and the register deallocation is large (Figure 1), it is important to pursue schemes targeting the aggressive register deallocation policies. Some solutions have been suggested in this direction [17, 18, 19, 31, 35]. In all of these schemes, however, each and every result value is still written into the register file and the validity of the physical register is used as one of the conditions for its early deallocation. In this paper, we propose a more aggressive approach towards the early reclamation of physical registers by avoiding the writebacks of the values that are consumed by all dependent instructions via the bypass network and deallocating the corresponding physical registers immediately. In the

* Oguz Ergin is currently with Intel Labs Barcelona, Spain.

rest of the paper, we refer to this mechanism as *Selective WriteBack (SWB)* and refer to the act of avoiding the writing of a result value into the register file as “dropping” or “discarding” the value.

The rest of the paper is organized as follows. In Section 2, we motivate this work by presenting various microarchitectural level statistics. In Section 3, we formally define transient values, describe the selective instruction writeback mechanism and explain how the precise state is reconstructed in the absence of some recent register instances. Section 4 describes our simulation methodology. We present the simulation results in Section 5, discuss the related work in Section 6 and offer our concluding remarks in Section 7.

2. Motivation and Definitions

It has been noticed by several researchers, that most of the register instances in a datapath are short-lived [8, 16, 20]. Following the work of [20], we define a value to be *short-lived* if the architectural register allocated as a destination of the instruction has been renamed before the value generated by this instruction is written back. We further define the instruction that renames a register allocated to hold a short-lived value as the *renamer*. In our simulations of the SPEC 2000 benchmarks, more than 80% of all generated register values were identified as short-lived.

Figure 2 presents some statistics about the short-lived values. The first bar shows the percentage of generated results that are short-lived across the SPEC 2000 benchmarks. The second bar shows the percentage of values that are short-lived and have no more than one consumer. The third bar shows the percentage of short-lived values with at most one consumer such that the consumer is issued before the short-lived value is written into the register file (while our technique can be extended to “drop” the values with multiple consumers, in this paper we only focus on the values with at most a single consumer, because 95% of the cases are captured and the detection hardware is significantly simplified, as we discuss in Section 3). For these instructions, the required value is supplied through the bypass network and is never read from the register file. Finally, the fourth bar shows the percentage of values captured by the third bar with the additional condition that there are no branch instructions between an instruction producing the short-lived value and its renamer. On the average across all benchmarks, in about 51% of the cases all of the four conditions are satisfied. This implies that more than half of the generated results are neither read from the register file, nor are they required to recover from a branch misprediction.

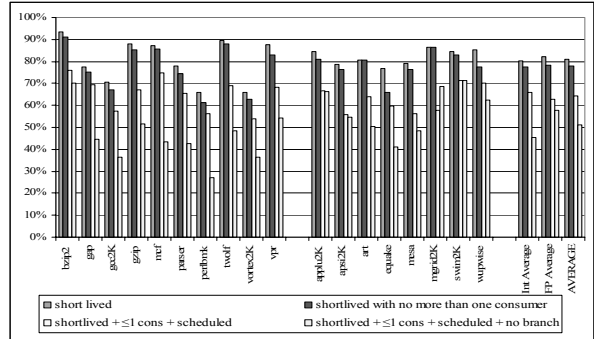


Figure 2 - Various Statistics about Short-Lived Value

We refer to a result that satisfies the above four conditions as a *transient value*. Transient values thus form a subset of the set of short-lived values. We propose to exploit transient values in the following manner. If a value is identified as a transient value, then it is not written into the register file and the corresponding destination physical register is deallocated immediately. Such selective writeback of register values results in two forms of benefits: reduced energy consumption (as fewer power-hungry writes to the register file are performed) and, higher performance (as the early release of the physical register targeted by the dropped value eases the pressure on the register file, effectively creating the illusion of having more physical registers).

3. Implementation of Selective Writeback

3.1 Identifying Short-Lived Values

To identify the short-lived values, we maintain a bit-vector called *Renamed* with one bit for each physical register. When the renamer is dispatched, it sets the *Renamed* bit corresponding to the previous mapping of its destination architectural register. The *Renamed* bits are reset when the corresponding physical registers are deallocated. Every instruction that has a destination register checks the status of the *Renamed* bit corresponding to its destination physical register one cycle before the writeback. If the bit is set, then the value to be produced is identified as short-lived.

3.2 Detecting the Absence of Branches

The second condition for discarding the value is the absence of the branch instructions between the value producing instruction and its renamer. We now describe how this can be easily determined. Each in-flight branch instruction is assigned a unique integer number called a branch tag. We assumed that 16 branch instructions could be in-flight at the same time, thus requiring 4 bits to uniquely identify each branch. Each physical register is tagged by the identity of the preceding branch and one extra bit, called *No_Branches*. The *No_Branches* bit is set to one when the physical register is allocated. When an instruction (the renamer) renames the

architectural register, it compares the branch tag of the previous physical register (say, Rk) corresponding to this destination (as obtained directly from the map table) against its own branch tag. If these branch tags are different, then the *No_Branches* bit of register Rk is set to zero.

3.3 Determining the Number of Consumers

To enforce the condition C3 for discarding a value, we use additional two bits (called *Num_Cons*) for each physical register. These two bits represent the number of consumers of each register. We distinguish three scenarios: zero consumers (*Num_Cons* contains 00), one consumer (01) and more than one consumer (10). The most significant bit in this bit-pair thus determines if the register has more than one pending consumer. When an instruction using physical register X as a source is dispatched, the contents of the *Num_Cons* bit-pair corresponding to register X are updated accordingly.

3.4 Ensuring That the Consumer Issues

To determine if the dependent instruction has issued by the time of the result writeback, we maintain a bit-vector called *Cons_Issued* with one bit for each physical register. The bit is reset when the corresponding physical register is allocated. When an instruction is selected for execution and moves to the stage(s) where it starts the register file access or waits to read the source via bypassing, the same source register address bits that are used for reading or bypassing are used to set the *Cons_Issued* bits associated with both of its source registers. Most contemporary and emerging datapaths have a multi-cycle delay between the tag broadcast for waking up instructions and the actual delivery of the result to permit this timing requirement to be met naturally. At the time of writeback, the value producing instruction checks the *Cons_Issued* bit corresponding to its destination physical register and if the bit is set, the value is dropped. Note that if the value of *Num_Cons* is 00, then the register value can be dropped irrespective of the value stored in the *Cons_Issued* bit, to ensure that transient values with no consumers are not written into the register file.

3.5 Putting it All Together: Avoiding Writebacks

In summary, in order to avoid writing a value into a physical register R, the following conditions have to be satisfied by the time the instruction producing the value of R enters the writeback stage: $Renamed[R] = 1$, $No_Branches[R] = 1$, $Num_Cons[R] = 0X$ and $Cons_Issued[R] = 1$. These conditions can be checked just prior to delivering the value on the forwarding network to waiting instructions and to the register file, as discussed earlier. When a value is dropped, a bit flag

(*No_retirement_RAT_update*) in the ROB entry for the producer is set. In the normal course of committing the ROB entry, if the *No_retirement_RAT_update* flag is set, the retirement RAT is not updated, thus preserving the most recently committed value of the corresponding architectural register. Such selective update of the retirement RAT is needed to correctly recover to a precise state despite the absence of some recent register values, as explained later.

The energy savings and potential performance gains from avoiding writebacks into the register file come at the cost of maintaining and managing a few additional bit-vectors, as described earlier. In our power evaluations, we fully account for the additional dissipations due to these bit vectors as detailed in Section 5.

3.6 Constructing the Checkpoints

We note that the TLB misses are the most dominant of all exceptions. Such exceptions do not present a problem to our scheme as we assume they are either handled in hardware (e.g. IA-32, PowerPC) or they are handled in software using dedicated registers. Yet another alternative would be to use [12] to inline the miss handler code in the ROB.

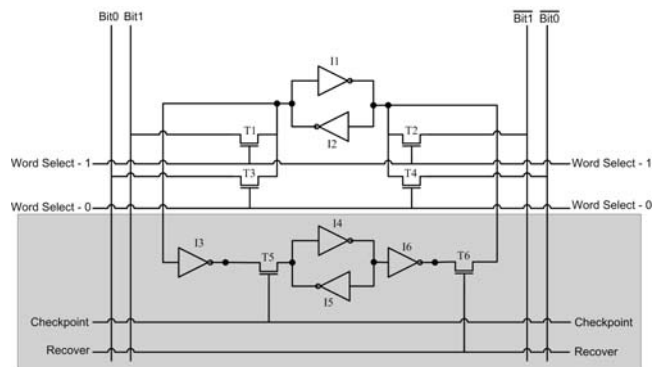


Figure 3 - Modified RF Bitcell

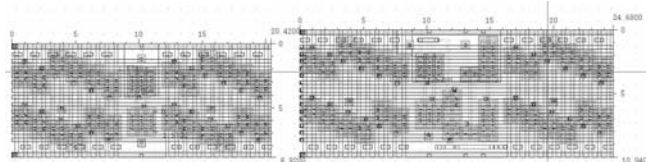


Figure 4 - Layouts of a Register File Bitcell

However, when page faults, other exceptions or interrupts occur, the precise processor state must still be restored despite the absence of some recent register values. This can be achieved by creating periodic checkpoints of the register file in the following manner. Let's suppose that the decision to initiate the creation of a register file checkpoint is made at some arbitrary point in time when the instruction I1 is the youngest

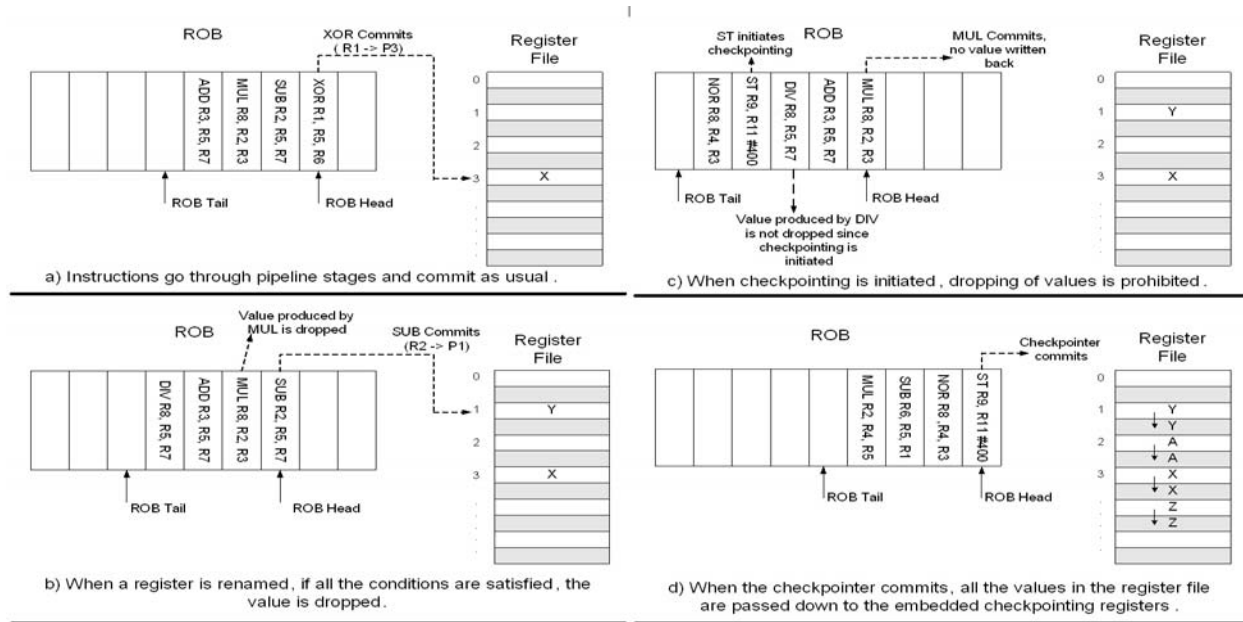


Figure 5 - Example of SWB with Checkpointing

dispatched instruction in the ROB. At this moment, the most recent instance of every architectural register is either the value in the register file or it is defined by some in-flight instruction, whose result has not yet been generated. For example, if the value produced by the instruction writing to architectural register X has been dropped, then there is another in-flight instruction down the stream that writes into the same architectural register. Consequently, if we allow all the instructions between the *ROB_head* (which points to the commitment end of the ROB) and instruction I1 to unconditionally write their results to the register file, then at the time of I1's commitment we will have a precise state of the register file. At this point, the full checkpoint of the register file can be created.

The use of checkpointing in conjunction with early register deallocation is not a new idea. Such checkpointing mechanism has been used, for example, in the Cherry scheme of [17], where the entire set of architectural registers was copied to a separate file that maintained the checkpointed register state. In [17], the process of creating a checkpoint is spread over several cycles and involves explicit data movement from the physical register file to a smaller storage component that implements the checkpointed register file. Of course, the multi-ported register file for maintaining the checkpointed state also needs to be implemented.

In contrast to Cherry, we propose to embed the checkpoint within the physical register file itself, as we proposed in [35]. This design is shown in Figure 3. Here, each 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. The contents of the retirement rename table also have to be checkpointed in the same manner. Since the rename table is a relatively small structure (it is indexed by the addresses of the architectural registers and the width of each entry is the number of bits in the physical register address), the overhead of the RAT checkpointing is minimal.

Figure 4 shows the CMOS layouts of a traditional 6-transistor 12-ported SRAM bitcell (left portion of Figure 4), and a traditional 12-ported SRAM bitcell with the embedded shadow bitcell that implements checkpoint, as depicted at Figure 3. For both layouts, we used 0.18um 6-metal layer TSMC process. 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 3. 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 because of the presence of the shadow cells, the increase in the delay is miniscule; it is less than 3.5% when all components of the RF access are accounted for according to our layout simulations. There is also a similar minimal impact on the word

select power dissipation during the normal course of read and write accesses. For all practical purposes, these overheads can be considered negligible, but we still account for them in our analysis.

3.7 Example of Checkpointing

Figure 5 shows the example that illustrates the checkpointing procedure using a short code fragment. For each instruction, the first register identifier specifies the destination register. Figure 5(a) shows four instructions located in the ROB. Figure 5(b) shows the situation where the oldest instruction (XOR) has been committed and the new instruction (DIV) had been inserted into the ROB and had undergone the process of register renaming before the value of the MUL instruction was produced. The destination register of the DIV instruction (R8) is the same as the destination register of the early instruction MUL. Therefore, the DIV instruction serves as the renamer for the MUL instruction. When the MUL instruction completes the execution, suppose that all four conditions for the early deallocation of its destination register, as described in Section 2, are satisfied. Consequently, the transient value produced by the MUL instruction is dropped, as shown in the Figure 5(b). At about the same time the SUB instruction is committed.

Figure 5(c) shows the situation when an instruction that initiates the checkpointing is dispatched (the STORE instruction). The register file checkpoint will be taken when the STORE instruction commits. Despite the fact that the value produced by MUL has been dropped, there is a younger instruction (DIV) which updates the same architectural register as MUL. After the dispatching of the checkpointing instruction (STORE), the process of selective writeback is disabled, and therefore all instructions prior to STORE will be forced to write their values back into the register file. Thus, the value produced by DIV will still be written into the register file despite the fact that the renamer of DIV (the NOR instruction) may have been dispatched and all other conditions for dropping the value may have been satisfied. This guarantees that at the time of the commitment of the checkpointing instruction the register file will have the most recent committed value for each architectural register and thus, the full register file state can be checkpointed at that instant.

Figure 5(d) shows the instant of the commitment of the checkpointed instruction. At this time, the values of all physical registers (or just the committed values) are stored within the shadow storage of the register file.

3.8 Handling STORE Instructions

To buffer a large number of store instructions between two consecutive checkpoints, we use the approach proposed in [17]. The values are stored within the local

cache hierarchy, but their propagation to the main memory is avoided until it is safe to do so. Each cache line updated in this manner is marked *Volatile*, using one extra bit for each cache line. When a processor needs to rollback to a checkpoint, all cache lines marked *Volatile* are invalidated using gang-invalidate signal. When the processor exits the early recycling mode and the precise state is created, then the *Volatile* bits are cleared. The idea of maintaining some speculative state in the cache hierarchy was also used in [32].

In our design, we handle the memory updates in the same manner as they are handled in [17], with the only difference being that volatile lines are not displaced beyond the L1 cache.

4. Simulation Methodology

For estimating the energy savings and the performance gains achieved by using the SWB scheme, we used a significantly modified version of the SimpleScalar simulator [2] that implements realistic models for a datapath where a unified register file is used. The studied processor configuration is shown in Table 1. 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 100 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

For estimating the energy/power dissipated in the course of accessing the register files, the event counts gleaned from the simulator were used, along with the

energy dissipations, as measured from the actual hand-crafted VLSI layouts using industry-standard Cadence[®] design tools. CMOS layouts for the register files and the bit-vectors in a 0.18 micron 6 metal layer process (TSMC) were used to get an accurate idea of the energy dissipations for each type of transition. For the register file layouts we used Metal 1 layer for local connections, Metal 2 layer for horizontal word select lines, VDD and GND lines, and Metal 3 layer for vertical bit and bit bar lines. In the interests of performance, we used double stage senseamps to insure fast amplification of the voltage difference between bit and bit bar lines. To save register file layout area, two neighboring bitcells in the same column share the same VDD and GND lines

5. Results and Discussions

This section is divided into three subsections. First, we present the results of the SWB scheme with no checkpointing overhead. Here, we assume an ideal scenario where exceptions or interrupts never occur. The results of these simulations show the upper bound on the benefits that can be achieved by using the SWB scheme. Second, we estimated the overhead of checkpointing by simulating the SWB scheme with checkpointing but still without exceptions or interrupts. And third, we simulated the realistic situation where exceptional events were incorporated into our models. The following three subsections describe the results of these experiments respectively.

5.1 SWB without Checkpointing

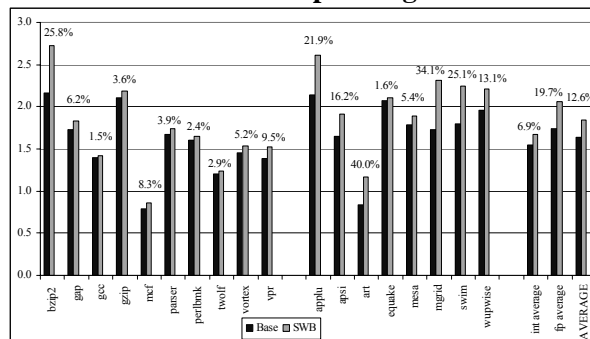


Figure 6 - Performance Impact of the SWB Scheme

In Figure 6, two bars are depicted for each simulated benchmark. The first bar shows the IPCs of the baseline configuration as defined in Section 4. The second bar shows the IPCs of the processor using the SWB scheme. The number displayed on top of each pair of bars shows the performance improvement of the SWB scheme over the baseline machine. For this experiment, we assumed that both integer and floating point register files have 64 entries. As seen from the figure, the performance increase due to the more efficient use of physical registers in the SWB scheme can be as high as 40% (in case of *art*). The average performance

improvement across all simulated benchmarks is 6.9% for the integer benchmarks and 19.7% for the floating point benchmarks and 12.6% overall. One can also see that some of the benchmarks, such as *gcc* and *equake* showed little improvements over the baseline performance. Overall, there is a broad range of performance improvements, as the benchmarks exhibit different register needs.

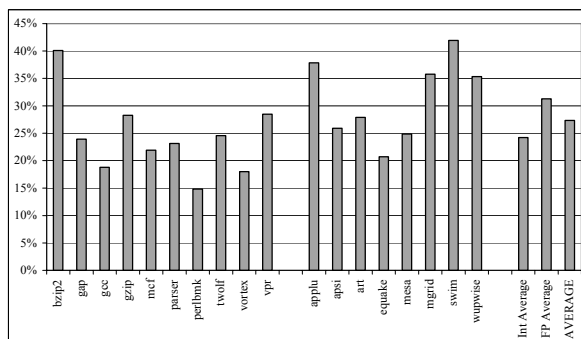


Figure 7 - Energy Reduction within the Register File

Figure 7 presents the overall reduction in the register file energy consumption when the SWB scheme is used. The energy reduction results from not writing a large percentage of the generated results into the register file. Although the total number of reads from the register file is higher than the number of writes, the write energy per access is larger than the read energy per access by about 1.8 times, as detailed in Table 2. The circuit simulations of the actual full custom register file layouts reveal that the energy dissipated in the course of driving the write lines plus the energy dissipated in the course of changing the contents of the SRAM bitcell significantly exceeds the energy dissipation of sense amps and prechargers used during the read accesses to the register file (Table 2). In the interests of low power, we precharged the read bit lines to $V_{dd}/2$. As shown in the figure, more than 27% of the overall register file energy can be saved, with savings across the individual benchmarks ranging from 15% to 42%. Note that the presented energy savings account for the additional dissipations that occur within the auxiliary bit vectors that are needed to implement the SWB scheme.

Table 2. Energy Dissipation Within the Register File Components

Energy (fJ)	Decoder	Word Select	Senseamp	Precharger	Bitcell	Write Driver	Total
Read Energy	724	1242	14592	9184	3456	-	29198
Write Energy	724	1242	-	-	8800	42473	53239

As is evident from Figures 6 and 7, using the SWB scheme results in significant IPC improvements and

non-trivial energy savings, when applied to a processor with fixed number of registers. It is also possible to apply the SWB scheme to use a smaller number of registers without hurting the performance significantly.

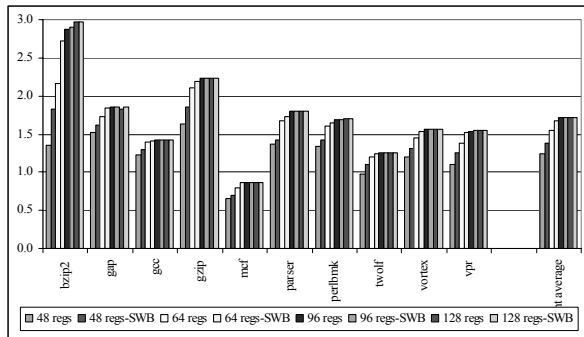


Figure 8 - SWB performance (integer codes)

Figures 8 and 9 show the commit IPCs for a range of register file configurations for integer and floating point benchmarks respectively. For these experiments, we used register files with 48, 64, 96 and 128 registers in both integer and floating point register files. When 128 registers are used, the SWB scheme does not result in any performance improvement. This is due to the fact that 128 registers are sufficient to sustain the performance for the considered processor configuration (where only a 96-entry ROB is used). Since there are almost no stalls due to the lack of physical registers, making the registers available by deallocating them early does not result in any IPC increase, with the exception of a few floating point benchmarks which mostly use double-precision arithmetic.

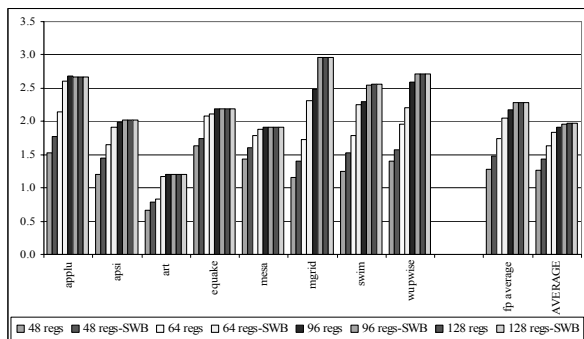


Figure 9 - SWB performance (fp codes)

The most significant IPC improvement is seen when 48 registers are in use. It can also be observed from the figures that the processor with 64-entry register files that implements the SWB comes as close as 5.1% to the performance of a processor with 128 registers in each register file, where the traditional register allocation and deallocation mechanisms are used. To put this number into perspective, one can also see from the figure that if the number of registers in the traditional design is

reduced from 128 to 64, then the average IPC degradation is about 17%. The use of the SWB with the 64-entry register files also results in about 51% register file energy savings compared to the baseline machine with 128 registers. Here, the energy reduction results because the size of the register file is reduced and fewer writes are performed.

5.2 SWB with Checkpointing

We now quantify the checkpointing overhead on the performance and power consumption of the SWB scheme. We performed the experiments with a range of different checkpointing periods. There is a tradeoff between the performance and power overhead of checkpointing and the number of instructions that need to be re-executed in the case of exceptions or interrupts. If we checkpoint too frequently, then a larger fraction of values have to be written to the register file, but fewer instructions have to be re-executed after the exceptions or interrupts. On the other hand, if the interval between two consecutive checkpoints is too large, then the overhead is minimized, but the number of instructions that need to be re-executed after exceptional events can be very high.

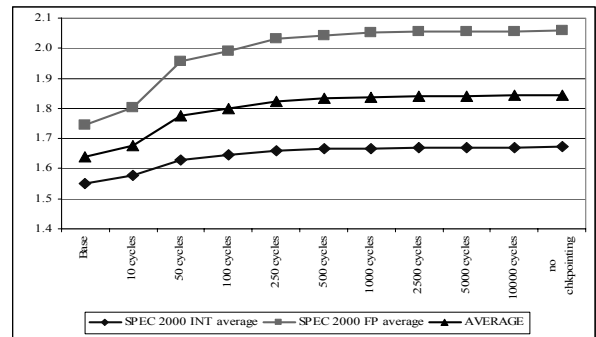


Figure 10 - IPCs of the SWB Scheme with Checkpointing

Figure 10 demonstrates the impact of checkpointing on the commit IPCs of the SWB scheme for the integer, floating point and the overall averages (due to the space limitations, in this and subsequent graphs we do not include the results for the individual benchmarks). The figures present the impact of checkpointing performed every 10, 50, 100, 250, 500, 1000, 2500 and 5000 cycles. As seen from Figure 10, provided that the checkpoints are not created very frequently, the overall performance gain of the SWB scheme with checkpointing is still significant. The important result to take away from this graph is that the IPCs of the SWB scheme where checkpoints are created every 500 cycles are virtually identical to the IPCs of the SWB scheme without checkpointing (i.e. the ideal scenario described in Section 5.1). Specifically, the IPC difference between the ideal case and the case where checkpoints are taken every 500 cycles is only 0.3% on

the average. Because more frequent checkpointing results in fewer instructions being re-executed, we conclude that 500 cycles is the optimal checkpointing period for a given machine configuration and the set of benchmarks.

Figure 11 shows the percentage of the values whose writebacks are avoided by the SWB scheme with checkpointing. As seen from the figure, frequent checkpointing results in fewer values being dropped, as expected. Similar to the results of Figure 9, one can see that when the checkpointing period is 500 cycles or higher, the percentage of dropped values is close to the ideal scenario without checkpointing.

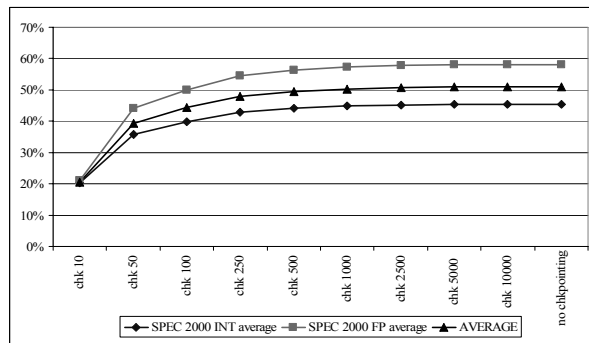


Figure 11 - Percentage of Dropped Values

5.3 SWB with Exceptions

Finally, we simulated the effects of the exceptional events, such as page faults, on the performance of the SWB scheme. These events have important implications on the performance of our design because the rollback to a checkpoint is required when they occur. For the purposes of this study, we assumed that the exceptional events occur at periodic intervals, where one out of every X memory operation results in an exception (a page fault). We performed the simulations across a large number of periodic intervals to gain a sufficient understanding of the potential impact of such exceptional events on the SWB scheme. The collection of the accurate statistics about the actual frequency of page faults and other exceptional events was beyond the limits of this work. We believe that the scope of parameters used in our experiments captures any practical range of the frequency of these events.

For each checkpointing period, we considered several frequencies of the exceptional events, requiring the rollback to a checkpoint. Figure 12 shows the performance impact. The experiments were performed for the checkpointing periods of 100, 250, 500 and 1000 cycles, and for each of these periods we assumed that a page fault occurs every 1000, 10000, 100000 and 1000000 memory instructions. The legend of Figure 11 is to be interpreted as follows: X_Y means that

checkpointing is done every X cycles and page faults occur every Y memory operations.

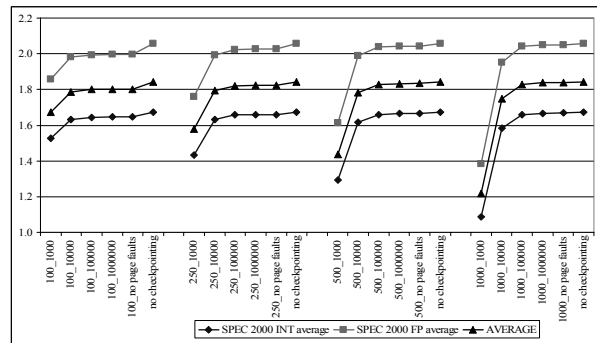


Figure 12 - IPCs of the SWB Scheme with Page Faults

As seen from the figure, unless the page faults occur exceedingly often such as once every 1000 memory operations (which is, of course, an unrealistically high rate), the performance is very close to that of the system with no page faults. For example, if the checkpoints are created every 500 cycles, then the difference in IPC between the system where the page faults occur every 100000 memory instructions and the system where page faults do not occur is only 0.25% on the average across all executed benchmarks.

Sustained page fault rates corresponding to the rising part of the plots in Figure 12 are very unlikely in contemporary systems that are endowed with a large amount of RAM. The system is likely to operate in the flat part of the plots, implying that IPC degradation due to the checkpointing mechanism is negligible.

6. Related Work

Several techniques have been proposed in the recent literature to reduce the power requirements of the register files and improve the efficiency of register file usage. These can be broadly classified into several categories: minimizing the number of registers, reducing the number of register ports, using various register file caching schemes and multi-banked register files.

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 [26, 9]. The drawback of delayed register allocation is in some design complexity, which stems from the need to maintain several levels of register maps. Delayed physical register allocation was also used in [21] to reduce the conflicts over the write ports in a multiple-banked register file.

The second set of techniques aim at reducing the register file pressure by using the early deallocation of

physical registers [17, 18, 19, 31, 35]. These techniques are close in spirit to our proposal, but there are fundamental differences. In all of these works, each and every generated result is still written into the register file and the validity of the register value is one of the conditions for the earlier register deallocation. Also, in some of the above schemes the deallocation of a register is bound either to the instant of instruction commitment [31, 35] or even to the instant of the commitment of the last user [19]. In contrast, we propose a more aggressive register reclamation mechanism by deallocating registers at the time of instruction writeback. As shown in Figure 1, significant additional cycles in the register lifetime can be saved. We also avoid writing the high percentage of the register values into the register file, thus saving significant amounts of energy. Energy reduction was not the goal of the previous proposals for the early deallocation of registers.

The third set of solutions reduces the number of registers through the use of register sharing [13,6,24] or register packing [36].

There is a large body of work that targets the energy reduction in the register files through reducing the number of register ports and register file banking. A two-level register file implementation, along with multiple register banks is described in [3]. In [21], the number of register file read ports is reduced by using the bypass hint. In [15], the peak read port requirements are reduced by prefetching the operands into an operand pre-fetch buffer. A delayed write-back queue is also used in [15] to reduce both the number of read and write ports on the register file. The use of multi-banking incurs considerable implementation complexity and has inherent performance loss due to the bank conflicts. In fact, the complex control structures of the banked schemes are likely to limit the processor cycle time [25]. In [25], an additional pipeline stage is introduced to arbitrate for the ports of multi-banked register files thus removing the port arbitration logic from the critical wakeup-select cycle, but negatively impacting the commit IPCs due to the increased branch misprediction penalties. The common feature of these techniques is that they all encounter a small performance loss due to various reasons. In contrast, our techniques improve energy-efficiency and performance at the same time. Also, all proposed register file banking schemes can be used in conjunction with our technique.

Replicated[14] and distributed [28,29] register files in a clustered organization have been used to reduce the number of ports in each partition and also to reduce delays in the connections in-between a function unit group and its associated register file. 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 [7, 4, 34]. In [4], Borch et.al. proposed the solution to extend the existing forwarding network to increase the number of cycles for which source operands are available without accessing the register file. The idea of caching recently produced values was used in [10], where a cache called the VAB (Value Aging Buffer) was used to hold most recently generated results. In [33], register file usage was optimized using compiler support to exploit dead value information.

7. Concluding Remarks

We introduced the notion of transient values and showed that 51% of all the results produced in a typical superscalar datapath are transient in nature. We also presented a scheme that eliminates the writes of such transient values to the register file. Along with elimination of these writes, we performed early deallocation of these registers to ease the pressure on the register file. The problems for maintaining a precise state that would be otherwise caused by dropping these results are avoided by using a local periodic checkpointing scheme that moves the checkpointed register values concurrently to shadow bitcells. Based on the simulation of SPEC 2000 benchmarks which also takes into account of the overhead of rollbacks which are a consequence of periodic checkpointing, our scheme achieves up to 40% performance improvement in some benchmarks with an average of 12% among all the benchmarks. Power savings in the register file range from 15% to 42%, with an average power saving of 27% across all benchmarks.

We realize these gains by using a simple hardware mechanism to identify transient values. Incorporation of the shadow bitcells into the register file increases the area of the register file by less than 20% but completely eliminates the external traffic for checkpointing.

8. Acknowledgements

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.

9. References

- [1] Akkary, H., Rajwar, R., Srinivasan, S., "Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors", in *Proc. of MICRO-36*, 2003.
- [2] 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).
- [3] Balasubramonian, R., et.al, "Reducing the Complexity of the Register File in Dynamic Superscalar Processor", in *Proc. of MICRO-34*, 2001.
 - [4] Borch, E., Tune, E., Manne, S., Emer, J., "Loose Loops Sink Chips", in *Proc. of HPCA-8*, 2002.
 - [5] Butts, A., Sohi, G., "Characterizing and Predicting Value Degree of Use", in *Proc. MICRO-35*, 2002.
 - [6] Balakrishnan, S., Sohi, G., "Exploiting Value Locality in Physical Register Files", in *Proc. of MICRO-36*, 2003.
 - [7] Cruz, J-L. et.al., "Multiple-Banked Register File Architecture", in *Proc. of ISCA-27*, 2000, pp. 316-325.
 - [8] Franklin, M., Sohi, G., "Register Traffic Analysis for Streamlining Inter-Operation Communication in Fine-Grain Parallel Processors", in *Proc. of MICRO-25*, 1992.
 - [9] Gonzalez, A., Gonzalez, J., Valero, M., "Virtual-Physical Registers", in *Proc. of HPCA-4*, 1998.
 - [10] Hu, Z. and Martonosi, M., "Reducing Register File Power Consumption by Exploiting Value Lifetime Characteristics", in *Workshop on Complexity-Effective Design (WCED)*, 2000.
 - [11] Hinton, G., et.al, "The Microarchitecture of the Pentium 4 Processor", *Intel Technology Journal*, Q1, 2001.
 - [12] Jaleel A. and Jacob B. "In-line interrupt handling for software-managed TLBs." in *Proc. ICCD-19*, pp 62-67. Austin TX, 2001.
 - [13] 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.
 - [14] Kessler, R.E., "The Alpha 21264 Microprocessor", *IEEE Micro*, 19(2) (March 1999), pp. 24-36.
 - [15] Kim, N., Mudge, T., "Reducing Register Ports Using Delayed Write-Back Queues and Operand Pre-Fetch", in *Proc. of ICS-17*, 2003.
 - [16] Lozano, G. and Gao, G., "Exploiting Short-Lived Variables in Superscalar Processors", in *Proc. of MICRO-28*, 1995, pp. 292-302.
 - [17] Martinez, J., et.al, "Cherry: Checkpointed Early Resource Recycling in Out-of-order Microprocessors", in *Proc. of MICRO-35*, 2002.
 - [18] Moudgill, M., et.al, "Register Renaming and Dynamic Speculation: An Alternative Approach", in *Proc. of MICRO-26*, 1993, pp.202-213.
 - [19] Monreal, T., Vinals, V., Gonzalez, A., Valero, M. "Hardware Schemes for Early Register Release", in *Proc. of ICPP-02*, 2002.
 - [20] Ponomarev, D., et.al, "Reducing Datapath Energy Through the Isolation of Short-Lived Operands", in *Proc. of PACT-12*, 2003.
 - [21] Park, I., Powell, M., Vijaykumar, T., "Reducing Register Ports for Higher Speed and Lower Energy", in *Proc. of MICRO-35*, 2002.
 - [22] Sethumadhavan, S., et.al, "Scalable Hardware Memory Disambiguation for High ILP Processors", in *Proc. of MICRO-36*, 2003.
 - [23] Smith, J. and Pleszkun, A., "Implementation of Precise Interrupts in Pipelined Processors", in *Proc. of ISCA-12*, pp.36-44, 1985.
 - [24] Tran, N., et.al., "Dynamically Reducing Pressure on the Physical Register File through Simple Register Sharing", in *Proc. of ISPASS-2004*, 2004.
 - [25] Tseng, J., Asanovic, K., "Banked Multiported Register Files for High Frequency Superscalar Microprocessors", in *Proc. of ISCA-30*, 2003.
 - [26] Wallase, S., Bagherzadeh, N., "A Scalable Register File Architecture for Dynamically Scheduled Processors", in *Proc. of PACT-5*, 1996.
 - [27] Yeager, K., "The MIPS R10000 Superscalar Microprocessor", *IEEE Micro*, Vol. 16, No 2, April, 1996.
 - [28] Canal, R., Parserisa, J.M., Gonzalez, A., "Dynamic Cluster Assignment Mechanisms", in *Proc. of HPCA-6*, 2000, pp.171-182.
 - [29] Farkas, K., et.al, "The Multicluster Architecture: Reducing Cycle Time Through Partitioning", in *Proc. of MICRO-30*, 1997.
 - [30] Azevedo, A., et.al., "Profile-based Dynamic Voltage Scheduling using Program Checkpoints in COPPER Framework", in *Proc. DATE*, 2002.
 - [31] Lipasti, M., et.al., "Physical Register Inlining", in *Proc. of ISCA-31*, 2004.
 - [32] Gopal, S., Vijaykumar, T.N., Smith, J., Sohi, G., "Speculative Versioning Cache", in *Proc. of HPCA-4*, 1998.
 - [33] Martin, M., Roth, A., Fischer, C., "Exploiting Dead Value Information", in *Proc. MICRO-30*, 1997.
 - [34] J. A. Butts, G. Sohi "Use-Based Register Caching with Decoupled Indexing", in *Proc. of ISCA-31*, 2004
 - [35] Ergin O., Balkan D., Ponomarev D., Ghose K. "Increasing Processor Performance through Early Register Release" in *Proc. of ICCD*, 2004.
 - [36] Ergin O. Balkan D., Ghose K. Ponomarev D. "Register Packing: Exploiting Narrow-Width Operands for Reducing Register File Pressure" in *Proc. of MICRO-37*, 2004