Appears in the Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, 2003

Reducing Datapath Energy Through the Isolation of Short–Lived Operands

Dmitry Ponomarev Gurhan Kucuk Oguz Ergin Kanad Ghose

Department of Computer Science
State University of New York, Binghamton, NY 13902–6000
e–mail:{dima, gurhan, oguz, ghose}@cs.binghamton.edu
http://www.cs.binghamton.edu/~lowpower

Abstract

We present a technique for reducing the power dissipation in the course of writebacks and committments in a datapath that uses a dedicated architectural register file (ARF) to hold committed values. Our mechanism capitalizes on the observation that most of the produced register values are short–lived, meaning that the destination registers targeted by these values are renamed by the time the results are written back. Our technique avoids unnecessary writebacks into the result repository (a slot within the Reorder Buffer or a physical register) as well as writes into the ARF by caching (and isolating) short–lived operands within a small dedicated register file. Operands are cached in this manner till they can be safely discarded without jeopardizing the recovery from possible branch mispredictions or reconstruction of the precise state in case of interrupts or exceptions. The power/energy savings are validated using SPICE measurements of actual layouts in a 0.18 micron CMOS process. The energy reduction in the ROB and the ARF is in the range of 20–25% and this is achieved with no increase in the cycle time, little additional complexity and no IPC drop.

1. Introduction

Traditional high–end microprocessor designs were driven mainly by performance considerations until very recently. With a steady increase in the number of transistors in the implementation of these processors along with the use of faster clock frequencies, power and energy considerations have emerged as additional design drivers for current high–end processors. In addition to the problem of coping with the large total power dissipations of a high–end processor, the designer also needs to deal with hot spots – localized areas of very high energy/power dissipations – within the processor die. High operating temperatures, especially within the processor's hot spots, also significantly reduce lifetime and reliability of the integrated circuits because several silicon failure mechanisms, such as electromigration, junction fatigue, and gate dielectric breakdown are exacerbated at high temperatures [25]. It is therefore imperative to operate these devices at safe temperatures to ensure their long life and reliable performance. Efficient packaging and cooling solutions, costing about $1 to $3 per Watt of power dissipated by the processor [24], continue to be used for coping with the increasing power dissipations in contemporary high–end processors, along with a variety of solutions that are at the level of the devices as well as at circuit, microarchitecture and system levels.

Most of today's high–performance microprocessors are implemented using dynamic out–of–order superscalar designs. These machines harvest available instruction–level parallelism (ILP) in sequential programs by exploiting a variety of techniques such as dynamic instruction scheduling, register renaming and speculative execution. The additional complexities introduced at the microarchitectural level for implementing these techniques greatly contribute to the increasing levels of power dissipation. In this paper, we introduce a technology–independent, microarchitecture–level solution for mitigating this increasing complexity without any performance loss and with a marked savings in the power dissipation in some key components of a modern superscalar processor.

In high–end superscalar processors, instructions are scheduled for execution dynamically, possibly out–of–order, as their source operands are produced. Data dependencies are usually handled by register renaming, where destination architectural registers are renamed to physical registers, with a new physical register allocated for every new result targeting an architectural register. Control dependencies are overcome through branch prediction and aggressive speculative execution along the predicted paths. A reorder buffer (ROB) is used to implement a precise state for interrupt handling [19] and also to squash the instructions along the mispredicted paths.

In some implementations, the ROB is also used as the repository of the results, where the ROB slot allocated for an instruction is itself used as the destination physical register. The Intel P6 microarchitecture (used by the Pentium II and III processors) typify such a datapath [10]. A variation of this datapath uses a dedicated physical register, called a rename buffer, to implement the physical result repositories. The IBM Power PC 604 uses this variation of the out–of–order execution datapath [18]. A common feature of these two designs is the presence of a separate register file (ARF) to hold the committed register values, requiring the movement of a result from the ROB (or rename buffer) to the ARF in the course of instruction commitment. In the rest of the paper, we consider the implementation, where physical registers are directly implemented by the ROB slots. The proposed technique is also applicable to a processor with the rename buffers.

Figure 1 depicts a grossly simplified version of a datapath where the ROB slots are used to implement physical registers. This figure only shows the datapath details that are relevant for this study. Forwarding interconnections as well as the connections needed for accessing the load–store queue and the data cache are not shown. The speculative results produced by the execution units (EX) are written into the ROB slots and simultaneously forwarded to dispatched instructions waiting in the Issue Queue (IQ). We assume that each IQ entry holds the actual source operand values in addition to the source tags. To enable back–to–back execution of dependent instructions, the result tags are broadcast before the actual data. If a source operand is available at the time of instruction dispatch, the value of the source register is read out from the most recently established entry of the corresponding architectural register. This entry may be either an ROB slot or the architectural register itself. If the result is not available, appropriate forwarding paths are set up. The result values are committed to the ARF in program order at the time of instruction retirement.

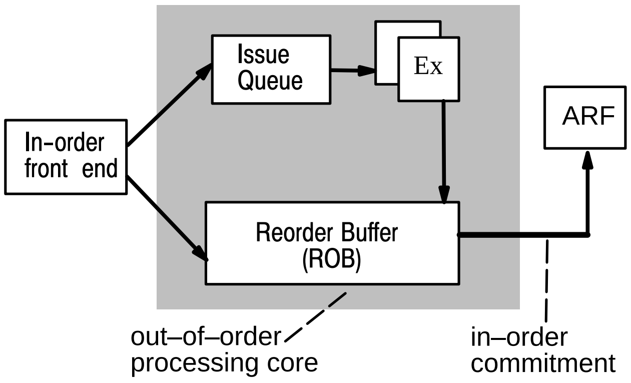


Figure 1. A P6–style superscalar datapath

The typical implementation of a ROB is in the form of a large, multi–ported register file. A significant amount of power is consumed in the course of result writebacks to the ROB (in the form of writes to the register file implementing ROB) and the commitments of these results from the ROB to the ARF (in the form of reads from the ROB and writes to the ARF). Some studies indicate that a ROB serving as a repository of non–committed results contributes a significant percentage to the overall chip power consumption [6]. In this paper, we propose a mechanism for reducing the power of the ROB and the ARF without impacting the performance of the processor. Our technique exploits the notion of short–lived operands – values targeting architectural registers that are renamed by the time the instruction producing the value reaches the writeback stage.

The simulations of the SPEC 2000 benchmarks show that as much as 71% to 97% of the results are short–lived. Note that our definition of the short–lived variables is different from the one used by Lozano and Gao in [13], where they define the variable to be short–lived if the value is consumed by all dependent instructions during its residency in the ROB. Our definition of short–lived variables is thus more restrictive than the one used in [13], especially for large ROBs, typical of current implementations. Our technique avoids unnecessary writebacks into the result repository (a slot within the ROB or a physical register) as well as writes into the ARF from unnecessary commitments by caching (and isolating) short–lived operands within a small dedicated register file. Short–lived operands are held in this dedicated register storage until they can be safely discarded without compromising a precise state reconstruction or a recovery from a possible branch misspeculation.

The rest of the paper is organized as follows. We quantify the percentage of short–lived values and show how these values can be identified in Section 2. The details of our technique for isolating short–lived values are presented in Section 3. Our simulation methodology is described in Section 4 and we present and discuss the simulation results in Section 5. Section 6 reviews the related work and we offer our concluding remarks in Section 7.

2. Identifying short–lived operands

We begin by showing that a large percentage of all generated results are short–lived. We then explain how these short–lived values can be identified and used for energy minimization. The extensions needed in the datapath for the identification of short–lived operands are very simple and are similar to what is used in [14] and [15] for early register deallocation and in [2] for moving the data between the two levels of the hierarchical register file. We maintain a bit vector, *Renamed*, with one bit for each ROB entry. An instruction that renames destination architectural register X (in the rest of the paper we call such an instruction the ***Renamer***) sets the *Renamed* bit of the ROB entry corresponding to the previous mapping of X, if that mapping indicates an ROB slot. If, however, the previous mapping was to the architectural register itself – i.e., to a committed value, no action is needed because the instruction that produced the previous value of X had already committed. These two cases are easily distinguished by maintaining the additional bit within each entry in the RAT. *Renamed* bits are cleared when the corresponding ROB entries are deallocated. At the end of the last execution cycle, each instruction producing a value checks the *Renamed* bit associated with its ROB entry. If the bit is set, then the value to be generated into the entry is identified as short–lived.

Figure 2 depicts the process of identifying the short–lived values. The instruction ADD renames the destination register (R1) previously written by the instruction LOAD. So, the ADD acts as a *Renamer* for the LOAD. Assume that the ROB entries numbered 31 and 33 are assigned to the instructions LOAD and ADD respectively. When the ADD is dispatched, it sets the *Renamed* bit corresponding to the ROB entry 31 (the previous mapping of R1), thus hinting that the value produced by the LOAD could be short–lived. When the LOAD reaches the writeback stage, it examines *Renamed[31]* bit and identifies the value it is about to produce as short–lived. Notice, however, that the indication put by the ADD is just a hint and it does not necessarily imply that the value produced by the LOAD will be identified as short–lived. For example, if the LOAD had already passed the writeback stage by the time the ADD set the value of *Renamed[31]* bit, then the LOAD would have not seen the update performed by the ADD and the value produced by the LOAD would not have been identified as short–lived.

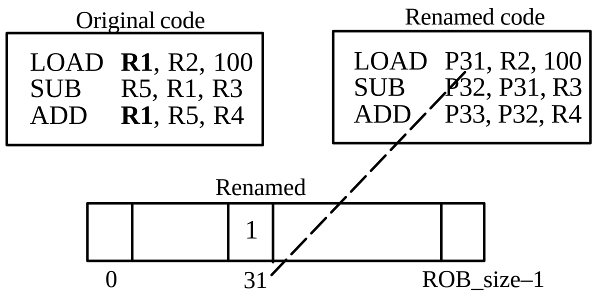


Figure 2. Identifying short–lived values

On the average across the subset of the SPEC 2000 benchmarks, about 87% of all produced values were identified as short–lived in our simulations, ranging from 97% for *bzip2* to 71% for *perl*. Details of our simulation framework are given in Section 4. Figure 3 shows the percentages of short–lived result values, identified as described, for the individual benchmarks. As detailed in Section 4, our simulations were performed for the pipeline with modest number of stages. As the number of pipeline stages between register renaming and writeback will increase in future implementations, the percentage of short–lived results will also increase commensurately.

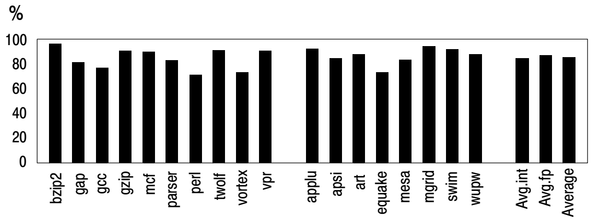


Figure 3. Percentage of short–lived values

While it was reported by other researchers [13] that short–lived values comprise a very large percentage of all generated results, our definition of short–lived operands is more restrictive than what is used in [13] and yet the percentages that we report are very close to the percentages presented in [13].

3. Isolating short–lived operands

As seen in Figure 3, a large percentage of all generated results are short–lived. We now describe how this can be exploited to minimize the energy dissipated during instruction writebacks and commitments.

If a result value is short–lived, why should it be written into the ROB and later committed into the ARF in the first place? All instructions that could potentially consume this value had already been dispatched by the time of the result generation. Therefore, the produced value is forwarded directly to all potential consumers waiting in the issue queue. This avoids the need to read the short–lived values from the physical register file. The only reason why we need to maintain a short–lived operand after the cycle of its generation is to support precise interrupts and to recover from branch mispredictions.

For energy reduction, we propose to isolate the short–lived result values from the rest of the generated results by writing them into a small dedicated register file, where the short–lived values would reside until they can be safely discarded without endangering precise state reconstruction or branch misprediction recovery. The energy savings then result from writing the majority of the results into a small register file for short–lived variables (**SRF**) instead of writing them to the ROB (or a large physical register file) and also not committing these values to the ARF. Figure 4 shows a P6–like datapath augmented with the SRF.

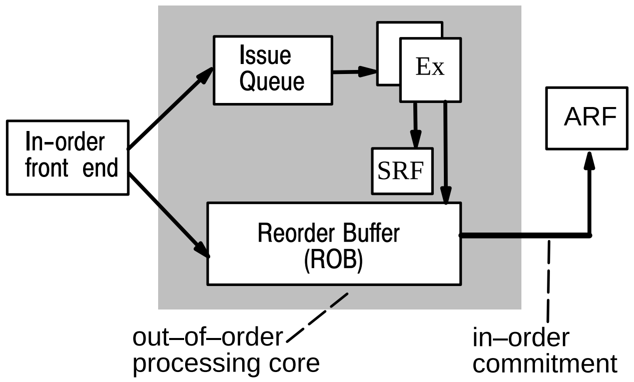


Figure 4. A P6–style superscalar datapath augmented with the SRF

If such a scheme is to be implemented, a number of problems need to be resolved. These are as follows: When and how are the short–lived variables inserted into the SRF and later removed from the SRF to free up the space for the new results? What happens if the SRF is full? How are the branch mispredictions handled? How do we reconstruct a precise state if an exception or an interrupt occurs? The rest of this section provides the solutions to these questions and addresses other considerations of our design.

3.1. Storing operands into the SRF

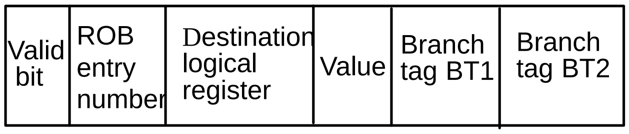
When an instruction producing a value reaches the writeback stage and the result value is identified as short–lived, an entry is established in the SRF when the following two conditions are valid:

(1) A free SRF entry exists, and,

(2) No SRF entry associated with the same ROB slot has already been established. This requirement is necessitated by the fact that the ROB index associated with

an instruction is used as an associative search key to locate the SRF entry for removal (Section 3.2). Unless this condition is imposed, two different SRF entries can be associated with a common ROB slot. This arises from the fact that the SRF entry established for an instruction can exist well after the instruction is committed (to permit the construction of a precise state, Section 3.4).

When both of the above conditions are valid, a SRF entry is set up for the instruction, whose result value is identified as short–lived. The SRF entry is established with the following fields (Figure 5): (a) the produced value, (b) the index of the ROB slot assigned to the instruction, (c) the destination architectural register address, (d) the branch tag of the preceding branch instruction and, (e) the tag of the branch instruction preceding the *Renamer*. The branch tags are needed in the SRF to support fast branch misprediction recovery. We assume that every branch instruction in flight is assigned a unique tag dynamically to allow instructions along its predicted path to be squashed if the branch is mispredicted. In Section 3.3, we describe how the renamer's branch tag is obtained. Finally, the valid bit of the SRF entry is set.



Valid bit	ROB entry number	Destination logical register	Value	Branch tag BT1	Branch tag BT2

Figure 5. The format of a SRF entry

If either one or both of the above conditions for writing the result into the SRF are not satisfied, the result is simply written into the ROB, as usual. A more radical solution would be to limit the number of write ports available for writing the results into the ROB (with the expectation that the majority of the results will end up being written into the SRF) and block the writeback if the ROB write port was not available. This, however, incurs a significant performance loss – in excess of 5% on the average if two ROB write ports are retained, as revealed by our simulations.

The second condition for establishing a SRF entry, as described above, can be enforced by simply maintaining a bit vector, called *Allocated_in_SRF*, with one bit for every ROB entry and checking the *Allocated_in_SRF* bit corresponding to the ROB entry of the instruction producing the value one cycle before the writeback. If the bit is set, the value is not written into the SRF. The *Allocated_in_SRF* bit is set when the instruction allocated to the respective ROB entry writes its value into the SRF and the bit is reset when the instruction in evicted from the SRF.

3.2. Removing entries from the SRF

We now describe how the SRF entries are removed. The key idea is to keep the value written into the SRF alive till the next instruction with the same destination architectural register (the *Renamer*) commits. When the *Renamer* commits, it is guaranteed that the previous instance of its destination architectural register will no longer be needed. The SRF entry holding that previous instance can then be safely discarded. This is analogous to the register deallocation procedure in the datapath with a unified register file for committed and speculative values.

The removal of the values from the SRF is a two–step process. First, at the time of instruction dispatching, the *Renamer* identifies the instruction, whose result value it will attempt to remove from the SRF. Second, at the time of commitment, the *Renamer* uses this information to actually remove the value from the SRF under certain conditions. Removal of a value from the SRF simply amounts to setting the valid bit of the corresponding SRF entry to zero.

When a destination architectural register is renamed and the previous mapping indicates that the register was mapped to an ROB slot, this ROB slot (obtained directly from the RAT) is recorded in the ROB entry of the *Renamer*. An additional ROB field, called *FS* (standing for "Flush SRF"), is used for this purpose. At the time of commitment, the *Renamer* examines the value stored in its *FS* field and if the field is set, it attempts to remove an earlier instance of the same destination architectural register from the SRF by performing an associative search on the SRF keyed with the value stored in the *FS* field. To avoid reading the entire *FS* field for each and every committing instruction, we extend the *FS* field by one bit (*FS_valid*) and set this bit to 1 whenever a value is assigned to the *FS* field. When the ROB entry is deallocated, this bit is reset to zero. The contents of this extra bit enable the readout of the remaining *FS* bits, thus avoiding the extra energy dissipations in the course of committing the instructions whose *FS* field was not set. In such cases, only one extra bit is actually read in addition to the regular commitment activity.

In processors that save the old register mapping information within the ROB to reconstruct a precise state, this old mapping can be directly used for our purposes and no additional field (like *FS*) is needed. If, however, the reconstruction of a precise state is performed using the second register mapping table, which points to the committed register values, then *FS_valid* bit in the ROB is used as described earlier. In any case, we conservatively account for the additional power dissipation due to the incorporation of extra ROB field in our simulations.

The two following conditions must be met for removing an entry from the SRF established by the instruction A when its *Renamer*, say the instruction B, commits:

(1) The *FS* field of the *Renamer's* ROB entry must match the ROB index field of some SRF entry.

(2) The matching SRF entry must not belong to any instruction different from A.

The second condition is necessitated by the fact that the ROB slot (say slot number *i*) originally assigned to A may have been reassigned to another instruction, say C, by the time B was committing. This second condition can be detected by adding a bit vector, *Uncommitted_Write*, with a bit for every ROB slot. Bits in this vector are set and reset as follows:

- When an instruction assigned to the slot numbered *i* within the ROB establishes an entry within the SRF, it sets both the *Uncommitted_Write* [*i*] flag as well as the *Allocated_in_SRF* [*i*] flag.

- The *Uncommitted_Write* [*i*] flag is reset when the associated instruction is committed.

Recall that the *Allocated_in_SRF* [*i*] flag remains set till the associated SRF entry is discarded. This flag by itself does not indicate if the corresponding ROB entry is being reused. Let us revisit the scenario described above involving the instructions A (which was allocated to ROB entry numbered *i*) and its renamer B. When B commits and the associative search in the SRF keyed by the ROB index specified by B's *FS* field indicates a match, the following two cases arise:

Case 1. *Uncommitted_Write* [*i*] is not set. This could happen for the following three reasons:

(1) The ROB slot numbered *i* was not reallocated after its use by A. As A was committed before B, *Uncommitted_Write*[*i*] was reset at the time A was committed.

(2) The instruction reassigned to the ROB slot numbered *i* (instruction C) was not considered as short–lived.

(3) The value produced by instruction C was not written into the SRF because one of the two conditions for writing into the SRF (as described in Section 3.1) was not satisfied – either the associated SRF entry was still in use by A or the SRF was full.

In any of these three scenarios under Case 1, the SRF entry in question clearly belongs to A and it would be safe to discard this entry as B commits.

Case 2. *Uncommitted_Write*[*i*] is set: the implication is that an instruction, say C, following B in program order was reallocated to the ROB slot indexed by *i* and this instruction owns the SRF entry. As B is committing, C must follow B in program order, as the *Uncommitted_Write*[*i*] flag is reset only when the associated instruction commits. In this situation, the SRF entry located by associatively searching the SRF with the index *i* clearly belongs to an instruction other than A and cannot be removed as B commits.

The steps for removing a SRF entry when an instruction commits are thus as follows:

(1) Check if the *FS* field within the ROB entry of the committing instruction is valid by examining the *FS_valid* bit. Perform the following step if the *FS* field is valid. No entry needs to be removed from the SRF if the *FS* field is invalid.

(2) Assume that the value of the valid *FS* field is *i*. If *Uncommitted_Write* [*i*] is not set, then associatively locate the SRF entry whose ROB index field equals *i* and invalidate the entry if one if found.

In our power evaluations, we assume that checking of the *Uncommitted_Write* bit and performing the associative search is done in parallel for performance reasons. It is possible to slightly optimize the accesses from the energy standpoint by enabling the associative search in the SRF only when the *Uncommitted_Write[i]* bit is not set and the *Allocated_in_SRF[i]* bit is set. In practice, however, this has little advantage because in the majority of the cases the associative search would be enabled if the *FS* field contains a valid ROB index.

Figure 6 depicts the process of removing the values from the SRF. We continue with the example that was used

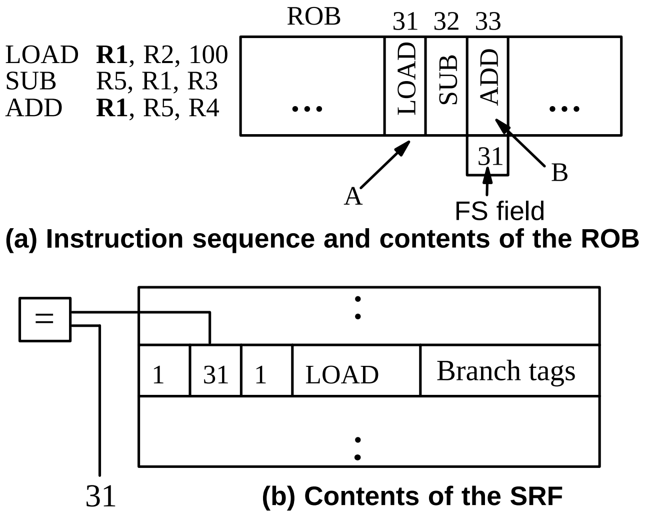


Figure 6. Removing the values from the SRF (B is the renamer for A)

in Section 2. Here, the ADD instruction (instruction B) renames the register R1, whose previous instance is produced by the LOAD instruction (instruction A). Thus, the ADD acts as the *Renamer* for the LOAD. Assume that the LOAD instruction is allocated to the ROB entry 31 and the ADD instruction is allocated to the ROB entry 33. When the ADD is dispatched, it obtains the previous mapping of its destination register (ROB slot 31 in this case) and sets the *FS* field within its own ROB entry to this value. The LOAD instruction, whose result value was identified as short–lived, establishes the SRF entry, as indicated in the picture. When the ADD commits, it uses the value stored in its *FS* field (31 in this case) to perform the associative search within the SRF. Provided that the *Uncommitted_Write[31]* is not set, the entry established in the SRF by the LOAD is removed by the ADD in the course of its commitment.

When an instruction commits, it also uses the value of the *Allocated_in_SRF* bit corresponding to its own ROB entry to decide whether the generated result value has to be written into the ARF. If the bit is not set, the produced result value is in the ROB and it is committed into the ARF in a regular manner. If the *Allocated_in_SRF* bit is set, the value generated by this instruction is short–lived, it resides in the SRF and it does not have to be written into the ARF.

3.3. Handling branch mispredictions

The branch mispredictions can cause an instruction that is designated to remove an SRF entry (the *Renamer*) to get squashed. If no additional action is taken on the branch mispredictions, then some values in the SRF will never be removed. Suppose that in the above example the instruction B is squashed as a result of a branch misprediction (assume that a branch is used in the place of the SUB instruction and this branch is mispredicted). In that case, no instruction will be responsible for invalidating the SRF entry written by the instruction A.

One way to handle this situation is to walk through the instructions on the mispredicted path in the ROB and undo the actions performed by these instructions on the SRF. No additional support is needed on the SRF in this case, but such misprediction handling is slow (because for a W–way machine at most W instructions can be examined in one cycle) and is hardly an option in today's high–performance designs. Most modern processor implementations rely instead on the use of aggressive checkpointing and branch tagging for quick recovery from the branch misspeculations. We now describe how similar mechanism is used with the SRF.

Each entry in the SRF is tagged with the branch tag (*BT1* in Figure 5) of the branch instruction preceding the *Renamer* of the instruction associated with the SRF entry. When such a branch is mispredicted, its tag is broadcast across the SRF and the SRF entries of all instructions whose renamers are on the mispredicted path are removed from the SRF and written to either the ROB slot (if the instruction whose value is removed from the SRF has not committed yet) or to the ARF (if the instruction had already been committed). In the above example, the value produced by the instruction A is removed from the SRF and is written to A's ROB slot (say, slot *i*) if A had not yet committed (*Uncommitted_Write*[*i*] = 1) or to A's destination register if A had committed (*Uncommitted_Write*[*i*] = 0).

The number of entries that can be removed in this manner from the SRF in one cycle depends on the number of read ports available on the SRF. To minimize the misprediction recovery latency, we multiplex the ports available for instruction writebacks to also support the SRF reads during the handling of mispredictions. This results in no performance degradation, as in almost all of the cases the SRF can be rid of the affected entries in one cycle.

We now describe how the branch tag *BT1* of the branch instruction immediately preceding the *Renamer* is obtained at the time of establishing a SRF entry – during the last execution cycle. When the *Renamer* is dispatched and renames a non–committed destination register of an instruction which was allocated to slot number *j* in the ROB, it obtains the branch tag of the preceding branch instruction (*BT1* in this case) and writes this tag into the *j*–th entry of a separate array called *Branch_Tags*, with one entry for each ROB slot. This step is not needed if the *Renamer* renames an already committed value. During the last execution cycle, an instruction whose *Renamed* bit is set, reads the value stored in the *Branch_Tags* array entry indexed by its ROB slot and in the next cycle writes this value into its SRF entry, if one can be established. To avoid the ambiguity in the branch tags, we use a different set of branch tags for branch instructions allocated across two consecutive usages of the ROB. In total, we allow 16 unique branch tags to be used (with 8 unresolved in–flight branches allowed simultaneously), thus requiring 4 bits to uniquely identify each branch.

The second branch tag in a SRF entry, *BT2* in Figure 5, is used to identify the branch that precedes the instruction whose value is inserted into the SRF. If such a branch is mispredicted, the value in the SRF simply has to be removed, without any need to copy it to the ROB or the ARF since the value was generated on the mispredicted path. On the occasion of multiple branch instructions on the mispredicted path, the branch tags of each branch are broadcast across the SRF, starting from the oldest branch. The number of cycles needed to recover from a misprediction can then increase, because only one branch tag can be handled in a cycle. However, it does not degrade the overall performance compared to the baseline machine, because similar mechanisms are used in the baseline implementation to flush the stale instructions from the instruction queue and from the FUs.

An associative lookup for the two branch tags is thus performed within the SRF on every branch misprediction. If both branch tags within an entry match the branch tag being broadcast, then the matching entry is simply invalidated. This corresponds to the situation when an instruction whose value is in the SRF, as well as its *Renamer* are both on the mispredicted path.

In addition to the state of the SRF, the precise state of the three bit–vectors used in our design needs to be restored. We now describe how this can be accomplished. The contents of the three bit–vectors are restored to a precise state by walking through the instructions on the mispredicted path in the ROB and undoing the modifications performed on these bit–vectors. Specifically, the *FS* field of each squashed instruction, if set, indicates the bit position in the *Renamed* bit–vector that was set by the instruction in question. Consequently, to recover the state of the *Renamed* bit–vector following a branch misprediction, it is necessary to reset the values of all such bits back to zero. The contents of the *Allocated_in_SRF* bit are reset for all instructions on the mispredicted path, whose *Uncommitted_Write* bit is set to zero. The *Uncommitted_Write* bit of each squashed instruction is then reset. To summarize, the following steps are performed during the examination of the ROB entry numbered *i* for the instruction on the mispredicted path:

1) Check if the *FS* field of the ROB entry numbered *i* is set. If it is set, then set the value of the *Renamed* bit indexed by the *FS* field to zero.

2) Check if the *Uncommitted_Write[i]* bit is reset (set to zero). If it is, then reset the *Allocated_in_SRF[i]* bit; if it is not, then proceed to the next step.

3) Reset the *Uncommitted_Write[i]* bit.

Steps 1) and 2) above can be performed in parallel. Due to the small access time of the bit–vectors, all three steps can be easily performed in one clock cycle – this was corroborated by our circuit simulations. For a W–way machine, W instructions on the mispredicted path can be examined in such a manner in one cycle. The overall branch misprediction recovery latency is not impacted though, because the actions detailed in the above three steps are performed in parallel with the reconstruction of the state of the register alias table.

3.4. Supporting precise interrupts

Almost no additional support is needed in our scheme for reconstructing a precise state on the occasion of an interrupt or an exception. The values written into the SRF represent part of a precise program state when instructions that produced these values commit. Ideally, the registers targeted by these values are overwritten in a very short period of time, but exceptions or interrupts present complications.

Let us return to the example that we considered earlier in this section and assume that the instruction A wrote its value into the SRF and subsequently committed. In the meantime, the instruction D that was dispatched between A and B caused an exception. To reconstruct the precise register state at this point, the value produced by the instruction A has to be copied from the SRF to the ARF. Notice that after all instructions preceding the faulting instruction commit and the values of all squashed instructions are removed from the SRF, *any architectural register is targeted by at most one value in the SRF.* This is a direct consequence of the second condition for removing a SRF entry. To reconstruct a precise state, it is therefore sufficient to simply copy the contents of all valid entries in the SRF into their corresponding architectural registers, as indicated by the destination architectural address field of each SRF entry. Since a sufficient number of read ports on the SRF is maintained (again, we are multiplexing the ports normally used for writebacks) and because of the small size of the SRF, such movement of short–lived values does not increase the interrupt handling time in any significant fashion.

3.5. Complexity of the solution

Our scheme for avoiding the writes of short–lived operands into the ROB slots/physical registers and for avoiding writes into the ARF during commitments incurs modest additional complexity. Three multi–ported bit–vectors, *Renamed, Allocated_in_SRF* and *Uncommitted_Write*, with one bit for each ROB entry, are used. We also need to maintain a 4–bit wide array *Branch_Tags*, with one entry for each ROB slot. Additionally, the ROB is extended to include one extra field (*FS*), 8–bit wide for the ROBs of 128 entries or less with one of these bits used to identify the validity of this field. (Again, this is not needed in processors that already checkpoint the old register mappings in the ROB). Of course, the register file to implement the SRF is also needed, with W read/write ports. The W ports to the SRF can be multiplexed between the writes of the operand values in the normal course of operations and the (disjoint) reads that are required for restoring the system to a precise state or in the course of handling branch mispredictions. The ROB index field of the SRF needs to be implemented as a CAM to support associative search to perform the invalidation of the SRF entries. To minimize the energy dissipated in the course of such associative lookups, we make use of the energy–efficient dissipate–on–match comparators proposed in [5]. These comparators dissipate energy predominantly on a full match in the comparands. Also, the comparators operate slightly faster than the traditional dissipate–on–mismatch pull–down comparator circuits. As at most one entry in the SRF can match the ROB slot that is broadcast by a committing instruction, the use of dissipate–on–match comparators thus results in significant energy savings. (Despite the obvious advantages of the dissipate–on–match comparators, their use is not the main reason of our energy savings. Even the use of traditional pull–down comparators to perform associative search within the SRF results in the overall energy reduction). Compared to the ROB, the SRF is a lightly–ported structure, as only the ports needed for writebacks are maintained. No operand reads are needed from the SRF, as all short–lived values are forwarded to the data slots within the issue queue at the time of result writebacks.

Associative searching is also used on the branch tag fields in the SRF. However, the power dissipations are less acute in this case, because the branch tag fields are only examined in the course of branch mispredictions which are relatively infrequent. In our power estimations, we accounted for the energy dissipated in the process of accessing the bit vectors and the SRF, including the associative lookups. The energy savings result from writing the majority of the short–lived result values into the *small lightly–ported* SRF. Notice that the SRF design is very simple, because no operand reads are performed from the SRF. The values are neither written into the ROB, nor are they committed into the ARF, saving a significant amount of energy. Our scheme also dissipates energy in the course of setting up a SRF entry and it expends energy to copy selected SRF values to either the ROB or the ARF on the occasion of branch mispredictions, exceptions or interrupts. As seen from the results presented later, despite the additional energy dissipations incurred in our design, we have a non–trivial savings in the overall energy associated with writebacks and commitments and the overall energy reduction.

4. Simulation methodology

For estimating the energy savings achieved by using our techniques, we used a significantly modified version of the Simplescalar simulator [1] to implement realistic models for such datapath components as the ROB (integrating a physical register file), the issue queue, the rename table, the ARF and the SRF. The studied processor configuration is shown in Table 1. We assumed a pipeline with 2 stages for instruction fetch, one stage for register renaming, 2 stages for accessing the register file (ROB) and one stage each for issue, execution, writeback and commitment. (Of course, instructions with multi–cycle execution latencies take more than one cycle to execute).

We randomly selected 10 integer SPEC 2000 benchmarks (*gap*, *gcc*, *gzip*, *parser*, *perlbmk*, *twolf*, *vortex*, *mcf*, *bzip* and *vpr*) and 8 floating point SPEC 2000 benchmarks (*applu*, *art*, *mesa*, *mgrid*, *swim*, *apsi*, *equake* and *wupwise*). 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

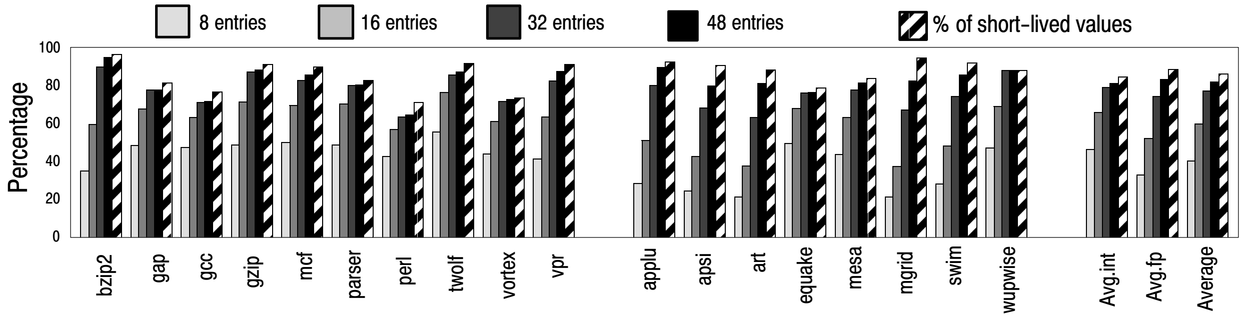


Figure 7. Percentage of the generated results written into the SRF

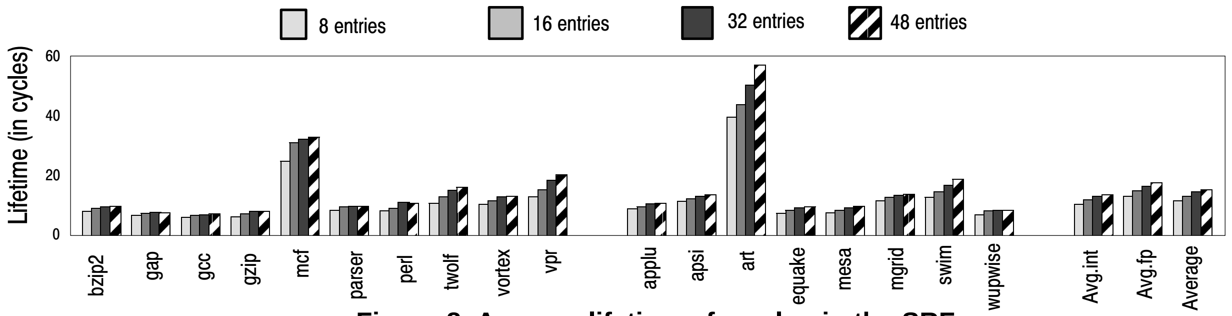


Figure 8. Average lifetime of a value in the SRF

and the results from the execution of the following 100 million instructions were used.

Table 1. Architectural configuration of a simulated processor

Parameter	Configuration
Machine width	4–wide fetch, 4–wide issue, 4–wide commit
Window size	32 entry issue queue, 32 entry load/store queue, 96–entry ROB with the allocation of 2 ROB slots for doubles
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, 100 cycles first chunk, 2 cycles interchunk
TLB	64 entry (I), 128 entry (D), fully associative, 30 cycles miss latency

For estimating the energy/power of the datapath components used in this study, the event counts gleaned from the simulator were used, along with the energy dissipations, as measured from the actual hand–crafted VLSI layouts using SPICE. CMOS layouts for the ROB, the ARF, the SRF 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.

5. Results and discussions

In this section we evaluate our design in terms of the percentage of data movements that are avoided as well as in terms of the actual energy dissipation.

Figure 7 shows the percentage of the generated result values that are written into the SRF for various SRF sizes. For 8–entry SRF, 40% of the produced values are stored in the SRF. Across the individual benchmarks, the smallest percentage is recorded for *art* – 21.6%. For 16, 32, and 48–entry SRF, 60% and 77% and 82% of all generated values are written into the SRF respectively. These percentages show the potential for energy reduction, as the values written into the SRF are not written into the ROB and not committed to the ARF. Finally, for comparison purposes the last bar of Figure 7 shows the percentage of result values identified as short–lived (this is identical to the results presented in Figure 3). The disparity between the percentage of short–lived values and the percentage of values that are written into the SRF is due to the limited SRF size.

Figure 8 shows the average lifetime of a value in the SRF. The lifetime is defined as the number of cycles between the insertion of a value into the SRF and the removal of this value. As expected, the lifetimes are short (about 10 cycles) for the majority of the simulated benchmarks, because the renamers are in close proximity

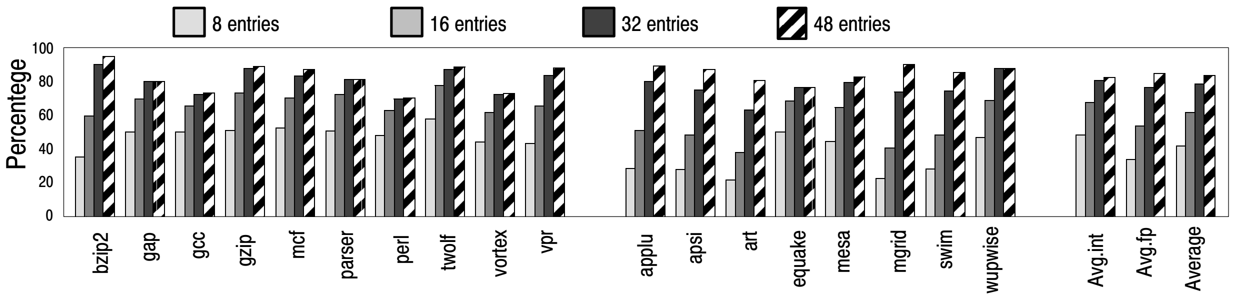


Figure 9. Percentage of result values that do not have to be committed to the ARF

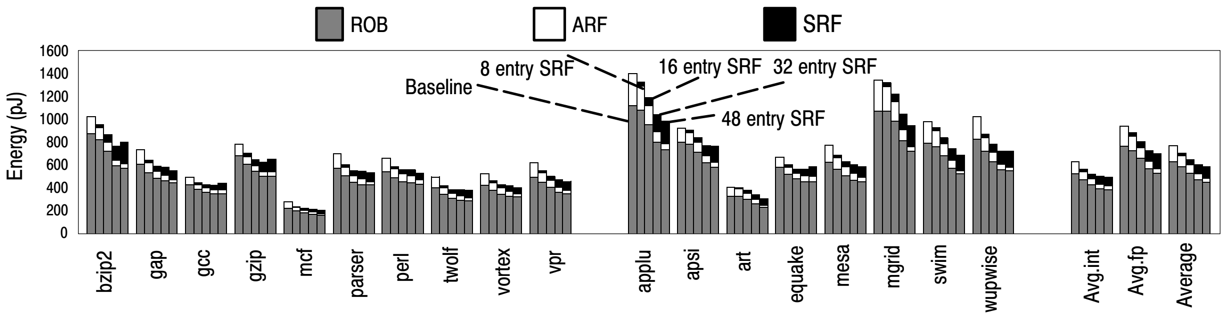


Figure 10. Energy per cycle in the ROB, the ARF, and the SRF

to the instructions whose destination registers they rename. The only benchmark with the relatively high value lifetime in the SRF is *art*. This is the primary reason why the SRF is frequently full for this benchmark and only a small percentage of the short–lived results can be written into the SRF. Notice also that a value lifetime in the SRF does not always correlate with the percentage of values written into the SRF. This is illustrated by *mcf* and *gzip* benchmarks. The reason for the relatively high percentage of values written into the SRF for *mcf* benchmark is the low IPC (0.67 for *mcf* vs. 2.1 for *gzip* vs. 1.57 for the average across all integer benchmarks). Consequently, despite the relatively high average value lifetime in the SRF, the pressure on the SRF is low and in many cases a SRF entry is freed up before a new short–lived value arrives.

Figure 9 shows the percentages of result values that do not have to be committed to the ARF. On the average, 42%, 62%, 79% and 84% of the data moves during commitment are avoided for 8, 16, 32 and 48 entries in the SRF respectively. These results are in close proximity to the statistics presented in Figure 7, as expected. The unintuitive result is that the percentage of the data movements avoided at commitment is slightly higher (by about two percentage points in each case) than the percentage of generated results that are not written into the ROB. This disparity is due to the instructions that produce the results but, due to branch mispredictions, get squashed out of the pipeline before they are committed.

Figure 10 shows the combined energy dissipated per cycle in the ROB, the ARF, the SRF, the array of branch tags and the bit–vectors used in our scheme. The energy of the ARF and the SRF is shown explicitly in the form of subbars, and the energy dissipated while accessing the bit vectors and the array of branch tags is assumed to be a part of the ROB energy. If a larger SRF is used, the energy savings from not writing higher percentage of result values into the ROB and not committing these values to the ARF outweigh the increased dissipations in the course of writes to the larger SRF for the SRF sizes of up to 32 entries. As the SRF size is increased beyond 32 entries, the resulting energy savings depend on the benchmark. For some benchmarks the energy savings are higher with 48–entry SRF as compared to 32–entry SRF. The reason is a higher percentage of data movements that are avoided for these benchmarks. For other benchmarks, the use of a 48–entry SRF results in higher overall energy dissipations, because there is no improvement compared to a 32–entry SRF in terms of the eliminated data movements and larger SRF dissipates some additional energy. Other factors that have significant power impact are the IPCs and the actual percentage of short–lived values written into the SRF. For high–IPC benchmarks with significant percentage of short–lived values, the write traffic to the SRF is high and that results in noticeable elevation of the SRF energy as we move from 32 to 48 entries (*bzip2, gzip*).

The resulting average energy savings within the ROB and the ARF are 9%, 16% and 21% and 23% for the SRF of 8, 16, 32 and 48 entries respectively. Somewhat different behavior of integer and floating point benchmarks can be explained by the fact that two entries in the SRF are allocated for storing a double–precision result and most of the results generated by floating point benchmarks are in the double–precision format. While it was beyond the scope of this paper to perform global power analysis, some rough estimates can be obtained from the numbers presented in [6]. In [6], Folegnani and Gonzalez show the

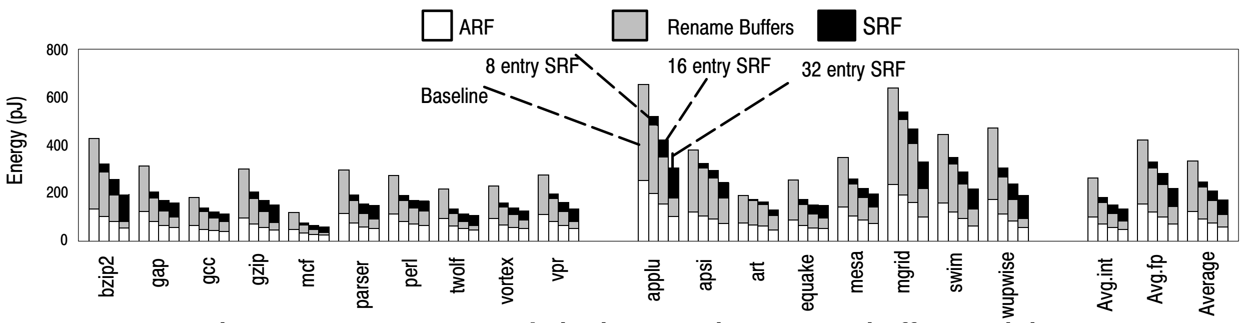


Figure 11. Energy per cycle in the ARF, the Rename buffers and the SRF

power distribution across the major blocks of a P6–like microprocessor. According to their analysis, the ROB represents about 27% of the overall chip power. Assuming that similar power distributions are true for the configurations that we studied, our technique then results in about 5% savings of the overall chip power.

Figure 11 shows the energy dissipation in the system that uses a set of rename buffers – physically separate from the ROB – to implement physical registers. The results presented in Figure 11 are for the processor with a 96–entry ROB and 64 rename buffers. The use of fewer rename buffers than the ROB entries is a consequence of the fact that some instructions, notably branches and stores, do not require any destination register and thus no rename buffer allocation is needed for these instructions. Our technique is directly applicable to this style of datapath if we use the rename buffer address instead of the ROB slot in the *FS* fields. Also, the associative search in the SRF is performed using the rename buffer address as the key. Energy savings in the ARF and the rename buffers are 14%, 21% and 24% if the SRF of 8, 16 and 32 entries is used respectively. Again, the energy dissipated while accessing the bit vectors and the array of branch tags is assumed to be a part of the rename buffers energy. Since the energy dissipated in the rename buffers is roughly half of the energy dissipated in the ROB integrating physical registers, the overall chip energy savings are in the order of 2–3% in this case. Still, as this is achieved with no performance loss, the solution is attractive.

6. Related work

It has been noticed by several researchers, that most of the register instances in a datapath are short–lived. In [7], Franklin and Sohi report the statistics about the useful lifetime of register instances. They conclude that maintaining an auxiliary register storage of as little as 30 registers to buffer the most recently generated results allows to avoid the writes of about 80% of the produced values into the register file, since these values are retrieved by all potential consumers during their residency in the auxiliary register storage. However, no solution was proposed in [7] to identify when it is safe to drop the values from the auxiliary storage. In this paper, we propose exactly such mechanism.

The exploitation of short–lived variables continued with the work of Lozano and Gao [13], who observed that about 90% of the generated result values are short–lived, in the sense that they are exclusively consumed during their residency in the ROB. The authors of [13] then proposed mechanisms to avoid the commitment of such variables to the architectural register file and also avoid register allocation for such variables. Their approach is based on a compiler analysis, where the ROB slots are effectively exposed to the compiler in the form of symbolic registers for storing the short–lived variables. In contrast to the scheme of [13], our approach does not rely on a compiler support.

In a recent study [17], Savransky, Ronen and Gonzalez proposed a pure hardware mechanism to avoid useless commits in the datapath that uses the ROB slots to maintain the non–committed results. Their scheme delays the copy from the ROB to the architectural register file until the ROB slot is reused for a subsequent instruction. In many cases, the register represented by this slot is invalidated by a newly retired instruction before it needs to be copied. Such a scheme avoids about 75% of commits, thus saving energy. The overhead in the scheme of [17] is in the form of the additional mapping table to keep track of the place in which the last non–speculative copy of each architectural register is stored. Our technique and the scheme of [17] represent two different approaches to avoiding the copying of committed register values from the ROB to the ARF. While the comparison of the results presented in [17] with our results is not quite indicative, because different ISAs, benchmarks and processor configurations were used, the percentage of result values that do not need to be committed, as reported in [17], is comparable to the similar percentage achieved by our design. Our scheme also saves the energy dissipated in the course of writebacks, which was not addressed in [17], where each and every generated result value is still written into the ROB. While the same can perhaps be achieved using the multi–banked implementation of the ROB (or the rename buffers), multi–banking incurs considerable implementation complexity and has an 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 [20]. In [20], additional pipeline stage is introduced to arbitrate for the ports of multi–banked register files thus removing the port arbitration from the critical wakeup–select cycle. Our scheme, on the other hand, has virtually no performance loss and no arbitration is needed.

The idea of caching recently produced values was used in [9], where a cache called the VAB (Value Aging Buffer) was used to hold most recently generated results. The VAB scheme has an inherent performance loss, because the accesses to the VAB and the register file are serialized.

Alternative register file organizations (mainly using various forms of caching and partitioning) have also been explored for reducing the access time and energy [2,3,4,11,21,22,23].

In [12], we introduced a scheme for reducing the ROB complexity by eliminating the read ports needed on the ROB for reading the source operands. To compensate for the resulting performance degradation, a small number of associatively–addressed retention latches was used to satisfy most of the requests that would otherwise have to be serviced from the ROB. In [26], the scheme was enhanced by not writing the short–lived operands into retention latches. The scheme of [12] and especially [26] can be used in conjunction with the technique proposed in this paper.

7. Concluding remarks

This paper proposed a technique to effectively remedy some of the principal inefficiencies associated with the datapaths that use separate register file for storing committed register values. Our scheme isolates the short–lived values in a small, dedicated register file avoiding the need to write these values into the ROB (or the rename buffers) and later commit them to the architectural register file. With minimal additional hardware support and with no performance degradation, our technique eliminates close to 80% of unnecessary data movements in the course of writebacks and commitments and results in the energy savings of about 20–25% on the ROB or the rename buffers. For considered processor configurations, this roughly translates to about 5% of the overall chip energy savings for the datapath, where the physical registers are implemented as the ROB slots, and to about 2–3% of the overall energy savings if physical registers are maintained in a separate set of rename buffers.

8. Acknowledgements

This work was 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 34th Int’l. Symposium on Microarchitecture (MICRO–34), 2001.

[3] Borch, E., Tune, E., Manne, S., Emer, J., “Loose Loops Sink Chips”, in Proceedings of Int’l. Conference on High Performance Computer Architecture (HPCA–02), 2002.

[4] Cruz, J–L. et. al., ”Multiple–Banked Register File Architecture”, in Proceedings 27th Int’l. Symposium on Computer Architecture, 2000, pp. 316–325.

[5] Ergin, O., et.al., “A Circuit–Level Implementation of Fast, Energy–Efficient CMOS Comparators for High–Performance Microprocessors”, in Proceedings of ICCD, 2002.

[6] Folegnani, D., Gonzalez, A., “Energy–Effective Issue Logic”, in Proceedings of Int’l. Symp. on Computer Architecture, July 2001.

[7] Franklin, M., Sohi, G., “Register Traffic Analysis for Streamlining Inter–Operation Communication in Fine–Grain Parallel Processors”, in International Symposium on Microarchitecture, 1992.

[8] Gwennap, L., “PA–8000 Combines Complexity and Speed”, Microprocessor Report, vol 8., N 15, 1994.

[9] Hu, Z. and Martonosi, M., “Reducing Register File Power Consumption by Exploiting Value Lifetime Characteristics”, in Workshop on Complexity–Effective Design, 2000.

[10] Intel Corporation, “The Intel Architecture Software Developers Manual”, 1999.

[11] Kessler, R.E., ”The Alpha 21264 Microprocessor”, IEEE Micro, 19(2) (March 1999), pp. 24–36.

[12] Kucuk, G., Ponomarev, D., Ghose, K., “Low–Complexity Reorder Buffer Architecture”, in Proceedings of the International Conference on Supercomputing, 2002, pp. 57–66.

[13] Lozano, G. and Gao, G., “Exploiting Short–Lived Variables in Superscalar Processors”, in Proceedings of Int’l Symposium on Microarchitecture, 1995, pp. 292–302.

[14] Martinez, J., Renau, J., Huang, M., Prvulovich, M., Torrellas, J., “Cherry: Checkpointed Early Resource Recycling in Out–of–order Microprocessors”, in Proceedings of the 35th International Symposium on Microarchitecture, 2002.

[15] Moudgill, M., Pingali, K., Vassiliadis, S., “Register Renaming and Dynamic Speculation: An Alternative Approach”, in International Symposium on Microarchitecture, 1993, pp.202–213.

[16] Slater, M., “AMD’s K5 Designed to Outrun Pentium”, Microprocessor Report, vol.8, N 14, 1994.

[17] Savransky, E., Ronen, R., Gonzalez, A., “Lazy Retirement: A Power Aware Register Management Mechanism”, in Workshop on Complexity–Effective Design, 2002.

[18] Song, S.P., Denman, M., Chang, J., “The PowerPC 604 Microprocessor”, IEEE Micro, 14(5), pp.8–17, October 1994.

[19] Smith, J. and Pleszkun, A., “Implementation of Precise Interrupts in Pipelined Processors”, in Proc. of Int’l. Symposium on Computer Architecture, pp.36–44, 1985.

[20] Tseng, J., Asanovic, K., “Banked Multiported Register Files for High Frequency Superscalar Microprocessors”, in Proc. of ISCA, 2003.

[21] Wallase, S., Bagherzadeh, N., “A Scalable Register File Architecture for Dynamically Scheduled Processors”, in Proceedings of PACT, 1996.

[22] Park, Il., Powell, M., Vijaykumar, T., “Reducing Register Ports for Higher Speed and Lower Energy”, in Proc. of the 35th Int’l. Symp. on Microarchitecture, 2002.

[23] Kim, N., Mudge, T., “Reducing Register Ports Using Delayed Write–Back Queues and Operand Pre–Fetch”, in Proc. of Int’l Conference on Supercomputing, 2003.

[24] Gunther, S., Binns, F., Carmean, D., Hall, J., “Managing the Impact of Increasing Microprocessor Power Consumption”, *Intel Technology Journal,* Q1, 2001.

[25] Small, C., “Shrinking Devices Put a Squeeze on System Packaging”, EDN, Vol, 39, N4, pp. 41–46, Feb.17, 1994.

[26] Kucuk, G. et.al. “Reducing Reorder Buffer Complexity Through Selective Operand Caching”, in Proc. of ISLPED, 2003.