Appears in 2004 International Symposium on Performance Analysis of Systems and Software

Dynamically Reducing Pressure on the Physical Register File through Simple Register Sharing

Liem Tran, Nicholas Nelson, Fung Ngai, Steve Dropsho, and Michael Huang
Dept. of Electrical & Computer Engineering
University of Rochester
{litran, ninelson, ngai, dropsho, michael.huang}@ece.rochester.edu

Abstract

Using register renaming and physical registers, modern microprocessors eliminate false data dependences from reuse of the instruction set defined registers (logical registers). High performance processors that have longer pipelines and a greater capacity to exploit instruction-level parallelism have more instructions in-flight and require more physical registers. Simultaneous multithreading architectures further exacerbate this register pressure.

This paper evaluates two register sharing techniques for reducing register usage. The first technique dynamically combines physical registers having the same value. The second technique combines the demand of several instructions updating the same logical register and share physical register storage among them. While similar techniques have been proposed previously, an important contribution of this paper is to exploit only special cases that provide most of the benefits of more general solutions but at a very low hardware complexity.

Despite the simplicity, our design reduces the required number of physical registers by more than 10% on some applications, and provides almost half of the total benefits of an aggressive (complex) scheme. More importantly, we show the simpler design to reduce register pressure has significant performance effects in a simultaneous multithreaded (SMT) architecture where register availability can be a bottleneck. Our results show an average of 25.7% performance improvement for an SMT architecture with 160 registers or, equivalently, similar performance as an SMT with 200 registers (25% more) but no register sharing.

Keywords: register sharing, register renaming, frequent values, simultaneous multithreading (SMT)

1 Introduction

In pursuit of higher performance through higher clock rates and greater instruction level parallelism (ILP), modern microarchitectures are buffering an ever greater number of instructions in the pipeline. The larger window of in-flight instructions offers the microarchitecture hardware more opportunities to discover independent instructions to issue simultaneously. However, maintaining more instructions requires a corresponding increase in the buffering structures; in particular, a larger physical register file with which to hold the generated results.

On the other hand, the counter forces to arbitrarily sized buffers are effects on cycle time due to non-scalable wire delays [1] and limits on power consumption [7]. Also, smaller buffers can reduce complexity in other regions of the chip; *e.g.*, the number of wires in the issue logic of Alpha 21264 is directly proportional to the number of physical registers [5]. Thus, despite large transistor budgets from shrinking technology dimensions, efficient use of register resources will always be an important design consideration.

A method for improving the use of physical registers to decrease the overall average demand is the technique of register sharing based on value [8]. In this type of sharing, logical registers containing the same value can be mapped to the same physical register, with the other physical register being released early. In [8], a general scheme to detect shared values is outlined for the Intel IA-32 architecture, an instruction set that exhibits considerable register pressure due to the small set of logical registers.

Another method to reduce register pressure is to aggressively reclaim physical registers when their values are no longer needed. Typical register renaming schemes conservatively allocate and release registers with the result that physical register lifetimes are unnecessarily long. Techniques have been proposed to alleviate this issue such as delaying the allocation until it is necessary [6] and early releasing dead registers indicated by compiler analysis [11].

An important contribution of this paper is to look at the practical design issues of these different ways of reducing register pressure, with a special focus on lowering the hardware complexity. We show that there are important special cases which provide many of the benefits but with much less complexity than that required for the general cases.

In particular, we show that optimizing the shared value detection to the special set of values zero and one generates almost *half* the benefits of the more general technique which shares arbitrary values. Restricting sharing to these two values enables a number of optimizations that greatly simplifies the implementation. Additionally, we propose a very simple mechanism that allows multiple versions of the same logical register to share the same physical register. We focus on a special type of instruction that we call single-use self-overwriting instructions. These instructions are quite numerous (about a quarter of all value-producing instructions) and their data dependence guarantees that they will execute in program order; thus, such instructions do not need multiple physical registers to avoid anti- and output-dependences.

Part of our contribution is the detailed design of our simple register sharing scheme that exploits these special cases. We further show that the benefits of register sharing can be significant in a

simultaneous multithreaded architecture where registers are more likely to be a limiting resource. In our simulations, we demonstrate a 25.7% performance improvement on average across a set of integer benchmark mixes.

The rest of this paper is organized as follows: Section 2 presents an overview of the methods to dynamically share physical registers. Our evaluation environment is discussed in Section 3. Results are presented in Section 4. We present a practical implementation of our design in Section 5. Related work is presented in Section 6. We conclude and discuss future work in Section 7.

2 Dynamically Sharing Physical Registers

Register renaming schemes dynamically transform a program into single-assignment form to remove false dependences and thus expose more instruction-level parallelism. However, allocating a physical register for every value-instance can lead to register waste. For example, if multiple physical registers contain the same value then the mapping can be adjusted to use a single copy of the value and the redundant registers can be freed.

Sharing physical registers increases the effective number of physical registers which can improve performance when registers are a scarce resource, such as in simultaneous multithreaded (SMT) processors [18].

In this section, we describe a general method for detecting shared values in the integer register file and reducing the number of physical registers in use. We also describe how, in certain cases, multiple instructions can share the physical register storage.

We show in Section 4 that specializing these techniques provides most of the opportunities for register sharing and greatly simplifies the design. In particular, limiting value-based sharing to just two values, zero and one is a good design trade-off. In Section 5 we present details of a design that is both simple and effective.

2.1 The common value buffer (CVB)

The *common value buffer (CVB)* is a mechanism for detecting arbitrary shared values between registers. The CVB is a fully-associative buffer of the last N generated values (LRU replacement). After instruction execution, results are compared to values in the CVB. Matches (*hits*) are considered to be common values.

Associated with the value in the buffer is the ID of an active physical register having that value. The mapping of the logical register associated with the just executed instruction is modified to point to the physical register from the CVB. This update requires modifying the logical-to-physical register alias table (RAT) and also updating the source fields of the instructions waiting for issue. The implementation requires a counter be associated with each physical register [8] that is incremented each time another register is redirected to use the value. The physical register cannot be released until its count decrements to zero. Since the precise timing of these actions requires details of the pipeline, we defer discussing the specifics until Section 5.

2.2 Trivial computations

Trivial computations [21] are computations in which the result can be known from the operand values without performing the calculation itself, *e.g.*, a logical bit-wise AND with zero. We detect two types of trivial computations. The first, called *Trivial 0*, detects results that can be determined *a priori* to always be zero. The second class, called *Trivial X*, detects computations in which the result can be determined *a priori* to match the value of either operand. The list of trivial computations we detect and their input conditions is given in Table 1.

Table 1. Trivial computations

Operation	Normal	Trivial 0	Trivial X
Add	X+Y	X=Y=0	X or Y=0
Subtract	X-Y	X=Y=0	Y=0
Multiply	X*Y	X or Y=0	X=1 or Y=1
AND	X&Y	X or Y=0	not detected
OR	X | Y	X=Y=0	X or Y=0
XOR	X *xor* Y	X=Y=0	X or Y=0
Logical shift	X<<Y or X>>Y	X=0	Y=0
Arithmetic shift	X<<Y or X>>Y	X=0	Y=0

These computations are detected during decoding and renaming when the physical register mappings of the source operands are read. Upon detection of a trivial computation, the register rename logic maps the destination register to the zero register (for *Trivial 0* computations) or to the same register as the source operand (for *Trivial X* computations).

2.3 Register lifetime reduction

In an R10000-like register renaming scheme, a physical register's lifetime spans from allocation to release, whereas the actual useful time is between the definition of the register and its last use, as shown in Figure 1. For simplicity, the register is allocated at the decode/register-renaming stage. Depending on the time of waiting and the number of cycles of execution, this can be many cycles earlier than the write-back stage, where the register storage is truly necessary. Furthermore, the allocated register is only released conservatively at the commit time of the next instruction that updates the logical register. This standard strategy can lead to much longer lifetime than is necessary. Our analyses show that a register is only needed for an interval about 10-20% of its total lifetime. This suggests that there is potential to increase effective register size by reducing lifetime.

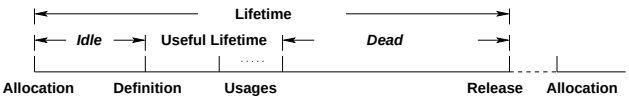


Figure 1. The life-cycle of physical registers.

In this paper, we propose a simple technique that exploits *self-overwriting* instructions to reduce effective register lifetime. We refer to instructions that update one of the source registers as self-overwriting (SO) instructions, *e.g.*, $r1 + r2 \rightarrow r1$. As shown in Figure 2, if the self-overwriting instruction is also *single-use* (*i.e.*, the instruction is the only consumer of the value in the destination register), then the lifetime of each version (in physical registers) of the (logical) destination register does not overlap. This

is enforced by standard data dependence checking hardware that serializes this set of instructions due to their read-after-write dependences. Therefore, instead of allocating multiple physical registers to hold the different versions of the logical register, these versions can conveniently share the same physical storage. Notice that although the instructions can share storage space, distinction among these versions is still necessary to allow correct dependence tracking and value communication. We defer these discussions and other implementation details to Section 5.

2.4 Register sharing designs

There are a number of options in how value-based (register) sharing, trivial computation detection, and lifetime-based sharing can be implemented. Several features to the design are the following:

Common values. The CVB permits register sharing with arbitrary values. A subset of the common value space is the set of highly used values of zero and one. While the CVB can provide more opportunities for sharing, focusing only on the zero-one subset allows only dedicated (hardwired) registers to be shared and eliminates the register use counters.

Trivial computations. Implementing *trivial X* detection requires general support for register sharing. In contrast, implementing *trivial 0* requires redirection to a hardwired register that is not actually part of the pool of physical registers and making detection extremely simple.

Early release stage. For zero/one detection, common values are detected during the execution stage. For general value detection using a CVB, detection occurs in the writeback stage. The earliest registers can be released back to the free pool is one cycle after detection. In Section 5 we discuss why delaying register release until the instruction *commits* reduces implementation complexity.

Bandwidth. Permitting multiple register redirections per cycle minimizes the delay in freeing register resources. However, it requires duplication of logic. On the other hand, buffering sharing requests and limiting redirection to one request per cycle eliminates the need for duplicate logic at the cost of delaying early register release.

Instruction type. When a single-use self-overwriting (SUSO) instruction shares a register with the previous definition and overwrites upon execution, we need the ability to retrieve the overwritten value when handling exceptions. One way to retrieve the value is to reverse the instruction. To be able to do this, we can only allow SUSO instructions with reversible opcodes to share register (see Section 5.4.2). Alternatively, we can rely on a more elaborate checkpointing scheme and allow all types of SUSO instructions.

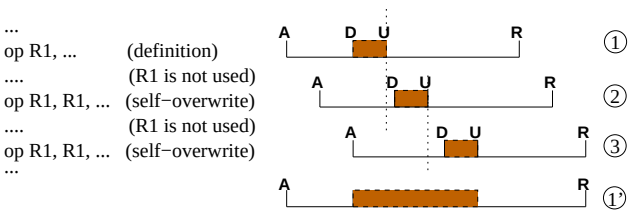


Figure 2. Single-use self-overwriting (SUSO) instructions and the life-cycle of physical registers allocated. A, D, U, and R stands for Allocation, Definition, Usage, and Release. Physical registers 1 to 3 can be substituted with a single register 1' with an extended lifetime.

Chain scope and length. SUSO instructions can form a chain of arbitrary length and it can span across multiple basic blocks. Limiting the chain to be within the same basic block will greatly simplify the design in branch misprediction handling. Also, the length of the chain that is allowed to share a register dictates the number of bits in the wakeup system to differentiate between different instructions that write to the same register. These design considerations will be discussed in more detail in Section 5.4.

From the above features, we construct two design points with which to compare to a base design without register sharing. In all, there are three cases:

- *base*: No register sharing.
- *complex*: The most aggressive design to maximize register sharing and early release of registers. The design includes a CVB, *trivial X*, immediate release, a redirection bandwidth matching the issue bandwidth. Every chain of self-overwriting instructions shares a single physical register, regardless of the length and scope of the chain. We note that, *complex* requires complicated micro-architectural support, especially for handling branch mispredictions and exceptions.
- *simple*: A modest design that trades some reduction in register sharing opportunities for much less implementation complexity. For value-based register sharing: values are restricted to zero/one (there is no CVB); only *trivial 0* is exploited; early-release is delayed until the instruction commits; and register sharing updates are restricted to one per cycle. For register sharing based on lifetime, sharing is limited to only those SUSO instructions within the same basic block of the initial assignment, and only up to three self-overwriting instructions are allowed to share the allocated register. This is the scheme we detail in Section 5.

3 Methodology

We explore the effects of the various register sharing schemes on individual applications to provide insight into behavior with sharing. From this data, we justify the *simple* scheme as providing the best advantage for the cost, and then demonstrate this scheme's effects on performance in the much more register intensive environment of an SMT architecture.

For the exploratory, per-application results we use the Simplescalar simulator version 4.0 (MASE) [10]. The processor is modeled after the MIPS R10000 [20] and has a general pool of 64 physical integer registers (no dedicated architectural registers). For the SMT simulations (the primary results) we use SMTSIM [18]. The SMT processor configuration parameters are given in Table 2.

We have modified both simulators to perform common value detection and register redirection as described in the text. Our focus is on common value reuse in integer benchmarks. The complete list of SPECInt 2000 benchmarks is given in Table 3 and the

Table 2. SMT Architectural Parameters

Threads	4
Fetch/Decode width	16 instructions
Branch predictor	2K gshare
Branch mispred. latency	6 cycles
Branch target buffer	256 entries, 4-way associative
Issue width	11 instructions
Reorder buffer entries/thread	256 entries
Issue queue entries	40 entries Int/Ld/St, 40 entries FP
Physical integer regs	see results
Functional units	8 Int (4 handle loads/stores), 3 FP
TLB	48 entries (I), 128 entries (D)
L1 I-cache	32 KB, 2-way, 64B line
L1 D-cache	32 KB, 2-way, 64B line
L2	256 KB 2-way, 64B line, 15 cycles
Memory latency	120 cycles

six application mixes used in the SMT simulations. In all simulations, we use the reference input set, fast-forward 500 million instructions per thread and then simulate for 500 million cycles.

Table 3. Applications & SMT Mixes

bzip2, gcc, crafty, gap, gzip, mcf parser, perlbmk, twolf, vortex, eon, vpr	
Mix 1: vpr, crafty, gcc, gzip	Mix 4: gzip, gcc, mcf, twolf
Mix 2: bzip2, mcf, twolf, parser	Mix 5: gap, gzip, gcc, mcf
Mix 3: twolf, parser, vpr, crafty	Mix 6: mcf, twolf, parser, gcc

4 Results

We limit the primary study to comparing three register sharing configurations: *base*, *simple*, and *complex* (defined in Section 2). Shown in Figure 3 is the per application performance improvement for the three configurations. More important is the reduction in register pressure using the three schemes shown in Figure 4. In this figure lower bars are better. While average performance improvement across all the individual applications is only 1.3% for *simple* and 4.5% for *complex*, the overall average reduction in number of physical registers in use decreases by 4.4% and 10.6%, respectively. In other words, the simple scheme garners almost half the reduction in register pressure of that of the more complex scheme.

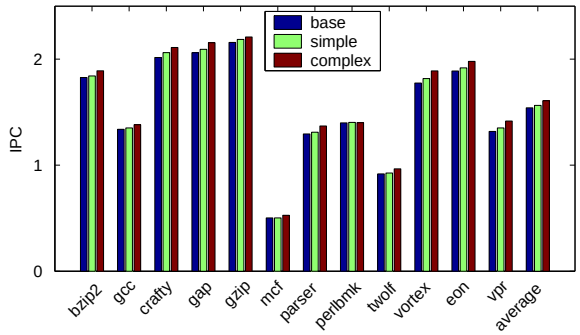


Figure 3. Relative performance

Table 4 and Figures 5 and 6 help explain why the simple scheme is so effective. In Figure 5, we show the hit rate in the CVB averaged across the benchmarks for various buffer sizes buffering values *excluding the special values of zero and one*.

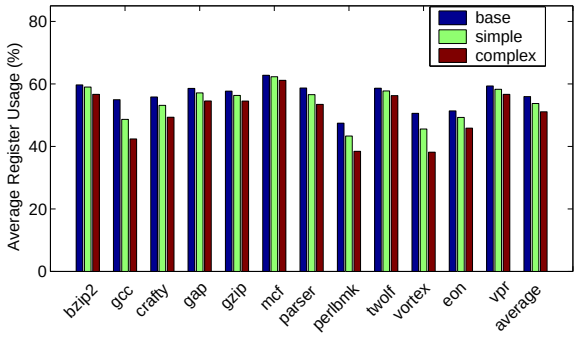


Figure 4. Relative physical register usage

Special care was taken to eliminate NOP-type instructions from being counted. As the buffer size is doubled the hit rate increases linearly. Shown in Figure 6 is the percentage of values that are either zero or one. These two values are invariably the top two most frequently occurring values. On average, 9.6% of the instructions generate one of these special values. The occurrence rate of almost 10% for values of zero and one nearly matches the hit rate of the larger CVB; thus, simply detecting these two values provides much of the benefits. Moreover, for all the applications, the frequency of the third frequent value is negligible and the value itself is application-dependent.

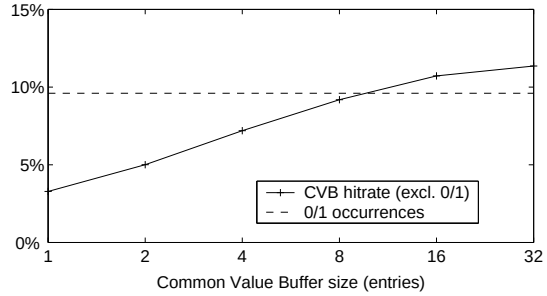


Figure 5. Hit rate of CVB

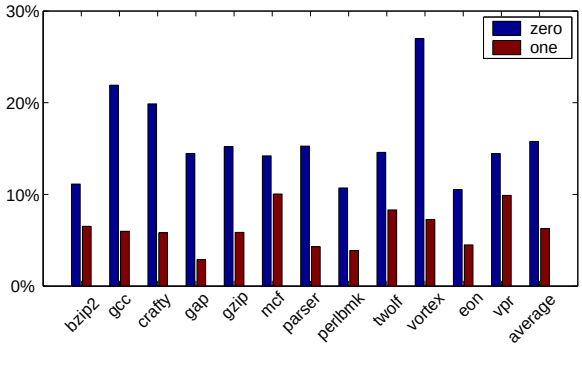


Figure 6. Fraction of values that are zero or one

The distribution of 0/1 detections occurring per cycle is shown in Table 4. Multiple occurrences of zero/one results in a cycle arise in only 4% of the cycles. Thus, buffering sharing requests and limiting their processing to one per cycle incurs little delay on the early release, in general. In contrast to buffering requests,

Table 4. Distribution of 0/1 detection per cycle

Number of values	0	1	2	3+
Percentage of cycles	82.9%	13.1%	3.35%	0.58%

another method would be to arbitrarily ignore all but one sharing request each cycle. We implement the former.

The simplicity of the simple scheme and its demonstrated effectiveness to exploit register sharing suggests the simple design is preferable to the more complex design for actual implementation.

Simultaneous multithreading. Figure 4 showing the reduction in register pressure is the more interesting data than the performance data since the effect on performance from reducing register pressure is highly dependent on whether the registers are a performance limiting resource. In an SMT processor, however, the availability of physical registers is often a limiting factor, much more so than in a single threaded architecture. We explore this effect in Figure 7, showing various aspect of the performance improvement for the SMT processor having the *simple* scheme.

In Figure 7-(a) we show the reduction in register conflict using the *simple* scheme. Register conflict is measured by the number of cycles the decode stage is stalled due to lack of available registers. With 160 physical registers, register conflicts are reduced by 28% to 40%, with an average of 35%. The *simple* scheme has the same or better effect as adding 40 physical registers (25% more). The decrease in register pressure from sharing improves performance by an average of 25.7% (Figure 7-(b)). Even with 200 registers, the scheme can still improve the performance and by an average of 8.8%, and up to 13.2%. In Figure 7-(c), we show the performance improvements for each individual application in the mixes, using the *simple* scheme with 160 registers. We measure the execution of the SMT processor for a fixed number of cycles and calculate the number of instructions finished per-thread with and without the *simple* scheme. Not surprisingly, as shown in the figure, the increase in resource benefits all threads relatively evenly. Finally, in Figure 7-(d) we show the effect of value-based and lifetime-based register sharing in isolation and combined. We can see that while the value-based sharing is more effective, the improvements from both components are additive.

Overall, we have shown that the *simple* scheme is indeed very effective and delivers significant performance improvement in an SMT processor.

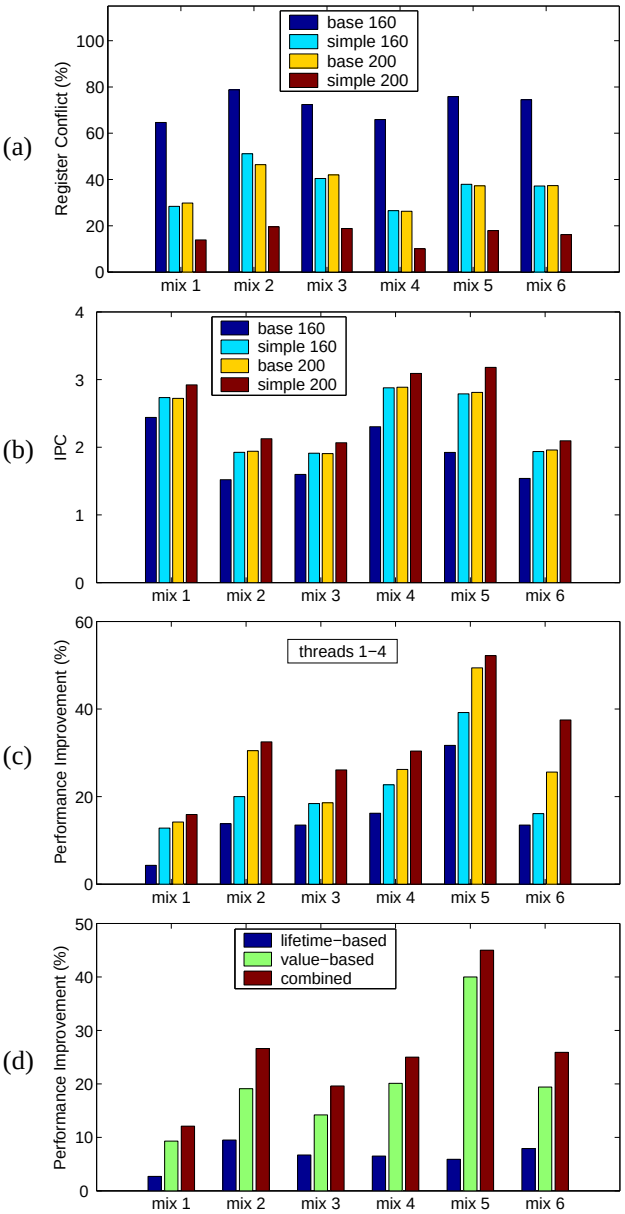


Figure 7. SMT performance analysis

5 A Design for Dynamic Register Sharing

5.1 The baseline system

While dynamic out-of-order microprocessors have various possible implementations, we focus on a straightforward baseline processor core that is largely based on MIPS R10000 [20]. The pipeline of the processor is shown in Figure 8.

Register renaming In this processor, there are no dedicated architectural registers [13, 20, 9]. Instead, physical registers from a larger pool are dynamically assigned to represent the logical registers. In the decode/map stage, the logical register numbers are translated into physical register numbers. This is done through a multi-ported RAM table (RAT). In this rename process, source logical registers are renamed into physical registers by reading its corresponding RAT entry. Each instruction with a destination register will allocate a free physical register in FIFO manner from the *free list*. This newly allocated physical register is written into the RAT, in the entry for the destination (logical) register. The previous value of that entry (the "old" physical register ID) is copied into the instruction's entry in the reorder buffer (ROB). When this instruction commits, the "old" physical register is appended to the free list. To handle branch misprediction, the RAT, together with the read pointer of the free list is checkpointed upon decoding of

Fetch	Dec/Map	Dispatch	EXE	(Mem)	WB	Commit

Figure 8. Pipeline of the baseline processor core.

a branch [20]. The checkpoint is restored when the branch is detected as mispredicted. Since the read pointer of the free list is also restored, physical registers allocated to wrong-path instructions are freed instantly.

Instruction wakeup Each physical register has a dedicated busy bit that is set during allocation to indicate that the producer has not finished execution, and therefore, dependent instructions need to wait in the instruction queue (or issue queue). When an instruction is issued, its destination register is broadcast to wake up dependent instructions and mark the associated operands as *ready*. Instructions with all operands marked *ready* can be issued in the following cycle. To speed up back-to-back data dependent instructions, any operands that use the result of a currently executing instruction will read the value off the bypass path as it is written to the register file.

5.2 Value-based register sharing

5.2.1 Overview

When two physical registers P_a and P_b contain the same value, one of the register (say P_b) can be early-released and re-allocated for other instructions. To ensure future instructions intending to read from P_b will read from P_a, the following needs to be done:

1. Change the RAT entry pointing to P_b to P_a.

2. Any instruction in the issue queue with a P_b in the source operand field needs to change it into P_a.

Apparently these steps would correctly modify the processor state and allow the release of P_b. However, there are three primary complications we list below, $C1 - C3$. For discussion purposes, let us call their corresponding logical registers L_a and L_b, respectively:

C-1 Physical register P_a cannot be released as usual, namely when the next producer of L_a is committed. We have to wait until both L_a and L_b are defined again, and the producer instructions have committed. Reference counting has been proposed to keep track of when a physical register can be freed [8], but the design is complex especially if it permits sharing along speculative paths, as reference counters need to be fixed upon a branch misprediction.

C-2 If the RAT entry of L_b has been overwritten, then one of the in-flight instructions (I_s), will release P_b at the commit time. Only one release, either the early-release or the normal release by I_s can be allowed.

If we allow the early-release, we have to search the ROB, perhaps associatively, and modify the entry of I_s. As the ROB continues to grow in size, and would otherwise need only indexing-based access, this search functionality would have significant impact on the scalability of the ROB.

C-3 Branch mispredictions also present complications. First, if we allow an instruction to early-release its allocated physical register before the instruction is committed, then if the instruction is squashed because of misprediction recovery we cannot free the register again. In our R10000-like register renaming scheme, this presents a major design challenge, since freeing all the registers allocated on the wrong path is done in a single action of restoring the read pointer of the free list [20].

Second, when we early-release a register (say P_b), we need to change not only the current RAT (if it is still mapped) but also any checkpoints where P_b appears. Otherwise when a branch misprediction happens, we may restore a checkpoint (made before the early-release) containing P_b, resulting in an error. The ability to search and selectively change checkpoint entries would introduce significant overhead.

To implement a generic dynamic register sharing, handling these complications would require complicated hardware support and/or a very conservative sharing scheme. We now describe an implementation for the special case of register sharing that is simple and straightforward. Limiting sharing to the values of zero and one allows the following simplifications:

1. Because their values are fixed, special dedicated registers (P_0 and P_1) can be provided without the need of freeing (C-1).
2. Detecting these two common values is almost trivial.
3. Changing the content inside the RAT or the source register field in the issue queue is greatly simplified: only one bit needs to be set (to 0 or 1), while others can be cleared.

Additionally, in the interest of hardware simplicity, we only early-release a physical register when (1) the common value-producing instruction commits, and (2) if the register is still in the RAT. Restriction (1) ensures that a branch misprediction roll back will not free any early-released register (C-3). Restriction (2) avoids complication C-2. Combining (1) and (2), we know that if the register is still in the RAT it will be present in the same entry in *every* checkpoint as well. Thus, we only need the ability to check the primary RAT, not any of the checkpoints. Given that we only exploit values zero and one, changing the checkpoints is quite easy.

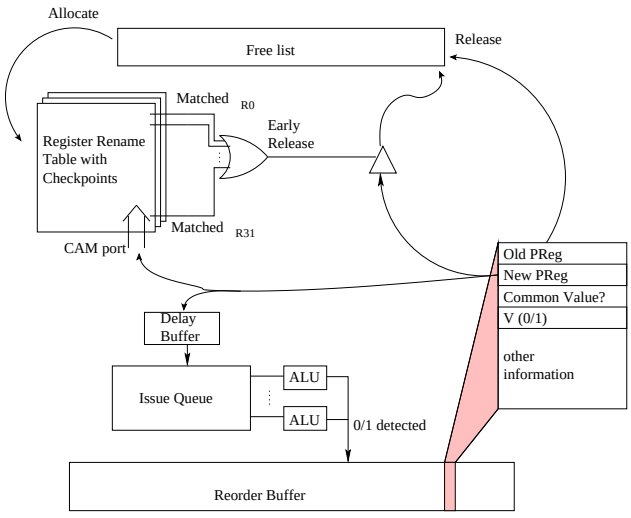


Figure 9. The overview of the microarchitecture for early release of registers containing common values.

Figure 9 shows a simplified overview of the proposed microarchitectural support for register sharing. The outline of the algorithm is as follows (some specific design choices are discussed later):

1. If an ALU operation results in 0 or 1, the instruction is marked in the ROB as a common-value-producing instruction. The specific value is also recorded. This requires only two extra bits in the ROB.
2. When an instruction is committed, the superseded physical register $OldPReg$ is released as usual. If the instruction is marked as a common-value-producing instruction, its allocated physical register $NewPReg$ becomes a candidate for early-release. It is broadcast through a special CAM port (Section 5.2.2) to detect its presence in the RAT. When a match occurs:
   (a) The physical register is released to the free list.
   (b) The matching RAT entry and the corresponding checkpoint entries are all set to the dedicated physical register (either 0 or 1, see Section 5.2.2).
   (c) The physical register number is entered into a 1-cycle delay buffer and used to rename instructions inside the issue queue (Section 5.2.3).

5.2.2 Modified RAT

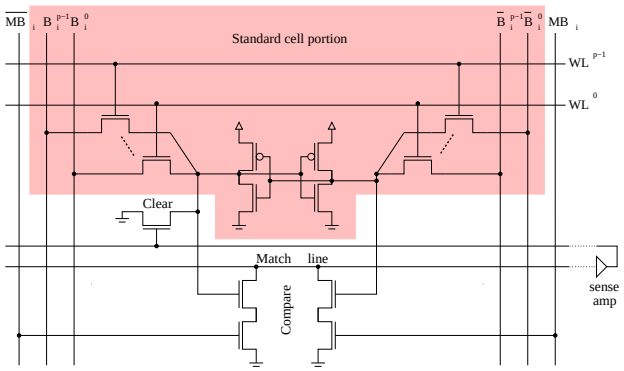
The register alias table needs to be slightly modified to add the functionality mentioned above. Figure 10 shows the diagram for the modified rename table. The base cell design is shaded for a p-ported table. Figure 10-(a) shows the comparators and *clear* transistor for all bits other than the least significant bit (LSB) of the physical register ID. Figure 10-(b) has additional logic (shown in bold lines) that sets the LSB to the common value (V) produced by the instruction. The match line is precharged and sense-amplified to perform a CAM-style parallel search. If the logical register number is readily available, only its corresponding match line needs to be precharged. To avoid race conditions, the CAM port should be accessed in the opposite clock phase as the RAM ports. Since early-release is not time critical, we assume this clock phase is after that of the RAM port access phase.

The added five or six transistors represent an insignificant increase: in a four-way issue pipeline, the map table requires 12 read ports and 4 write ports ($p = 16$) for a total of 36 transistors in the base cell design [16]. This circuit allows a maximum early-release of 1 per cycle. As we have seen in Section 4, only 4% of the cycles have more than one early release candidate. A simple, small buffer can easily accommodate the occasional bursts.

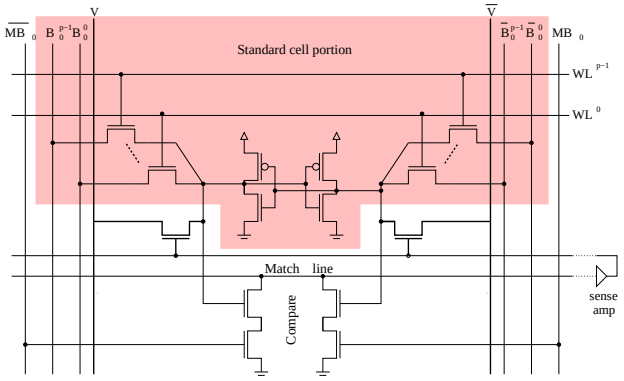
To ensure the early-released physical register ID does not reappear erroneously by way of RAT checkpoint restore (Section 5.2.1), the corresponding mappings in *all* checkpoint copies are likewise set to the same dedicated ID (*P0/P1*) simultaneously. The reason we can change all copies indiscriminately is that the common-value-producing instruction is being committed and any valid checkpoint at the moment should also point to that register.

5.2.3 Modified issue queue

Broadcasting the to-be-released register ID into the issue queue is necessary to ensure all dependent instructions read the correct



(a) Non least signifi cant bits ($B_i,\ i = 1..n-1$)



(b) Least signifi cant bit (B_0)

Figure 10. Diagram of modifi ed RAT cell

source register when issued. This broadcast is very similar to the instruction wake-up broadcast. The difference is, normal wake-up marks the operand as ready and the instruction is ready to issue if all source operands are ready. The broadcast for register early release, however, marks the matching source register as a common value, indicating that during issue, rather than reading from the register file, zero or one should be used.

This logic can be implemented in two ways. In one method, a dedicated broadcast port can be built into the issue queue. When a source operand register ID field matches the content on this special broadcast port, the operand is marked, and the common value recorded. Alternatively, an existing free wake-up broadcast port can be used. In this case, each such port is augmented with two special bits. One bit indicates that the port is used for early-release broadcast, and the other for the specific value.

The reason for the delay buffer in Figure 9 is that, when a candidate early-release physical register is checked in the RAT in cycle n, there may be instructions decoded and mapped in the same cycle that references the candidate register. Recall that these instructions read the RAT earlier than the potential RAT update due to early-release, and thus will not see any change. They will enter the issue queue in cycle $n + 1$. If the broadcast is done in cycle n, these instructions will not be notified. Notice that a 1-cycle delay is sufficient since typically broadcasts happen in a later clock phase than dispatch. This is to ensure proper wake-up. Finally, we note that there is no race condition between the register's release and subsequent reuse even though the register is released one cy-

cle before the broadcast. This is because there are multiple cycles between when a released register can be written to again.

5.2.4 Discussion

By freeing the register at commit time and only if the register is still mapped, our design is much simplified. Compared to immediate early-release, this does miss a few opportunities to free more registers containing the frequent values zero or one. This is indeed a good design tradeoff as exemplified in Figure 11. In this figure, we show the breakdown of physical registers during 200 million cycles of execution of one application mix (mix 5). From bottom up, we show the number of registers holding 0, 1, or some other value. The remaining registers are either not yet written to (unassigned), or not allocated (free). The left and right half of the plot corresponds to the breakdown without and with our sharing scheme respectively. The figure shows that while about 40 registers contain either 0 or 1 in the baseline system, only about 4 registers still hold one of the two special values.

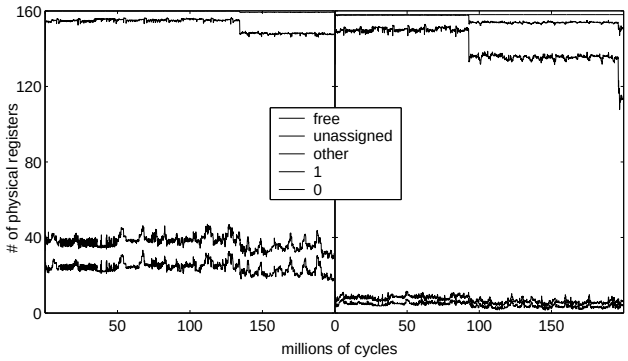


Figure 11. Register usage breakdown

5.3 Zero value trivial computations

Minimal logic is required to detect zero value trivial computations. Some calculations are zero regardless of the input operands (*e.g.*, $X \; xor \; X = 0$). For instances where one or both of the operands must be known to be zero for the computation to be trivial, we limit the detection to cases where the operand registers have been mapped to the zero register already so an explicit read of the register value is unnecessary.

5.4 Lifetime-based register sharing

Given an SO (self-overwriting) instruction, we call the previous dynamic instruction that writes to the same logical register its *assignment instruction*. Notice that, this assignment instruction can be an SO instruction itself. As explained in Section 2, an SUSO (single-use self-overwriting) instruction can share the physical register allocated to the assignment instruction. An SO instruction is an SUSO instruction if no other instructions between the SO and the assignment instruction sources the destination register of the assignment instruction.

5.4.1 Detection, sharing, wakeup, and release

Detection: Detecting SO instructions dynamically is straightforward. Detecting SUSO instructions requires cross-comparing sources and destinations of simultaneously renamed instructions and an extra *reference* bit per logical register in the RAT. The reference bit is cleared when the logical register is written to and set when it is read from. To limit the detection of SUSO instructions to be within the same basic block, we simply set all the reference bits after decoding a conditional branch and making a checkpoint for the RAT (Section 5.4.2). When restoring a checkpoint, we also set all the bits.

Sharing and wakeup: When an SUSO instruction is detected, we can simply reuse the currently mapped physical register (for the destination register), without allocating a new one. However, in our baseline system, the physical register ID also serves the purpose as a tag for instruction wakeup and value communication. Therefore we cannot allow two instructions to have the same destination physical register number. In a system with virtual physical registers [6], this can be solved by using two virtual physical register IDs pointing to the same physical register. In our design, we choose a much simpler scheme: we extend the physical register ID and use the most significant bits to differentiate different versions. In particular, if we extend the ID by n bits, we can allow 2^n instructions to share the same physical register storage.

For example, in a system with 160 physical registers (requiring 8-bit addresses), if we add two most-significant bits, then the IDs 10, 266, 522, and 778 are the four tags associated with physical register 10. A non-SUSO instruction that produces a value will be allocated a tag with the two most-significant bits set to 0. Subsequent SUSO instructions writing to the same destination will increment these two bits, until it reaches 3. The next SUSO instruction (writing to the same destination) will be treated as a non-SUSO instruction and assigned a new register.

When decoding an SUSO instruction, if the destination register is mapped to P_0 or P_1, the special dedicated registers, the SUSO instruction will also be treated as a normal instruction and obtain a new physical register.

Release: When an SUSO instruction shares the physical register with its corresponding assignment instruction, the $OldPReg$ field of the SUSO instruction is set to an invalid value, the same way as a non-value-producing instruction. Thus, when the SUSO instruction is committed, no register is released. The shared register will eventually be released at commit time of the next instruction that updates the same logical register and allocates a new register.

5.4.2 Misprediction and exception handling

Allowing multiple instructions to write to the same physical register presents a challenge to branch misprediction and exception handling. Consider this sequence of events: (1) if the original assignment instruction occurs in a different (earlier) basic block than the associated SUSO instruction, (2) a branch between the SUSO instruction and its assignment instruction is mispredicted, and (3) if the assignment instruction has been committed, then we cannot recover the original assignment value that had been speculatively overwritten. For this reason, we only allow an SUSO instruction to share a register with its assignment instruction if they belong to the same basic block. (If the SUSO instruction falls into the next basic block, it will be treated as a normal instruction and will allocate a new register.) This way, if the SUSO instruction is on the wrong path, so is the assignment instruction.

It is possible that an exception occurs for an instruction between an SUSO instruction and its assignment instruction. If, by the time the exception is handled, the SUSO instruction has already finished execution, then the physical register shared by the SUSO instruction and its assignment instruction no longer contains the assignment instruction's result. After handling the exception, the SUSO instruction will be re-executed leading to an erroneous result.

To solve this problem, we need to reverse the effect of any already-executed SUSO instructions. In a typical exception handling mechanism, to reconstruct the RAT, the oldest valid RAT checkpoint is restored, and the ROB is "walked" in reverse order to unmap the instructions in the oldest basic block [16, 20]. During this process, the only additional effort for us is to reverse any already-executed SUSO instruction: we compute the overwritten operand using the result and the remaining operands, if there are any. (For example, there is no remaining operand for $r1 + r1 \rightarrow r1$, and performing a right-shift on $r1$'s current value recovers the overwritten operand.)

To be able to do this, we need the remaining operand unchanged and a reversible opcode for the instruction. Fortunately, the remaining operand is guaranteed to be in a normal register (not shared) and stay unchanged since it is sourced by the SUSO instruction, and therefore can not be the destination of another SUSO instruction (violates the single-use rule).

To guarantee a reversible operation, we simply do not perform register sharing for a non-reversible SUSO instruction (*e.g.*, load) in the first place. Most ALU instructions and address manipulation are reversible. The reverse operation depends on the exact format of the SUSO instruction and the detail can be found in [17].

An alternative design is to roll back to the beginning of the basic block, re-execute the basic block without sharing registers. To do so, we cannot commit any instruction in a basic block until all instructions in the basic block finish execution without exception. To handle the pathological case where a basic block is larger than the size of ROB, we have to artificially divide a large basic block into smaller ones by inserting a not-taken branch instruction dynamically. This design is not only complicated, but also suboptimal in that instruction commit (and thus resource recycling) can be delayed unnecessarily.

SUSO instructions and adding 2 bits to extend the register ID is a good design point. On average, out of all value-producing instructions, 24.3% are SUSO instructions. Our design captures 80% of these SUSO instructions, or 19.2% of all value-producing instructions.

5.5 Implementing both schemes

When implementing both value-based and lifetime-based register sharing, we have to make sure they work together. In particular, we have to ensure that a shared register is not erroneously released early. In our design, this is not a problem.

Consider a pair of instructions sharing a register: an SUSO instruction and its corresponding assignment instruction. (1) The assignment instruction will not early-release the shared register because of the restriction that the register ID be still mapped in the RAT. Recall that although an SUSO instruction shares the physical register, it still updates the RAT table with a different register ID (incrementing the two most-significant bits). (2) The SUSO instruction can safely early-release the shared register. Because the assignment instruction commits before the SUSO instruction and thus does not need the register anymore.

6 Related Work

One of the closest work to ours is the register renaming by Jourdan *et al.* [8]. As previously discussed, the authors use register sharing to exploit value locality and reduce register pressure in the Intel IA-32 architecture. In this paper, we restrict the range of values and greatly simplify the design. In a concurrent work, Balakrishnan and Sohi also propose to use dedicated registers for values zero and one to reduce register waste on storing these frequent values [2]. However, in [2], implementation and support for branch misprediction handling are not discussed in detail. We analyze tradeoffs and present a simple and very effective design that only reclaims registers at the commit stage. Moreover, we also propose a simplified scheme that reduces the lifetime of physical registers and show how both schemes work together.

In Cherry [12], registers are released early but state must be recovered on exceptions. The design relies on checkpointing and the ROB to rollback to a correct architectural state on exceptions and replay instructions up to the exception in order to recover state from resources released early. In our register sharing, when processing an exception, we only need to reverse the effect of already-executed SUSO instructions in the oldest basic block.

In [6, 19] register allocation is delayed until the value is actually ready to be written. In particular, Gonzalez *et al.* [6] describe a *virtual physical register* design. The virtual register scheme assigns a virtual register ID during decode as a placemarker. An actual physical register is not allocated until the result is ready in the writeback stage. This technique reduces register pressure by not requiring physical registers for many of the in-flight instructions. Our work is complementary to the virtual physical register work. In fact, combining our work with virtual registers would further decrease register pressure and simplify our design. The simplification occurs since assignment to a physical register will not occur until writeback when the value of the result is known and common values can be assigned to the zero/one registers directly, eliminating the necessity of redirection later.

Martin *et al.* [11] detect registers having *dead values* (values no longer needed) and return them to the free pool. The technique is used to reduce the number of saves/restores at procedure calls, but also reduces register pressure by reducing register lifetime. Our lifetime-based register sharing is simpler in that it is purely hardware-based.

Yi and Lilja [21] evaluate the most extensive set of trivial computations in the literature to improve performance. In addition to our *Trivial 0* and *Trivial X* policies, the authors also perform strength reduction (*e.g.*, convert a multiply by 2 to a shift). In contrast, we explore the effect of squashing trivial computations on register pressure.

The frequent value cache (FVC) of Zhang *et al.* [23] attempts to improve the performance of a direct-mapped cache by supple-

menting it with a small additional buffer. Because only frequently used values are stored in the FVC, the data can be encoded to keep the buffer small. The FVC acts as a specialized victim cache optimized to storing only data lines with known frequent values. The CVB is similar, but leverages these common values for a different purpose, sharing physical registers to reduce register pressure.

Finally, there is a body of work that tries to build large register files while limiting the adverse effect on the access timing [3, 4, 15, 22, 14]. Our approach is complementary to these approaches in that we reduce the demand of physical registers through sharing.

7 Conclusion and Future Work

As high performance processors attempt to exploit ever greater parallelism, more instructions are in-flight and, consequently, more physical registers are required. With the register file being a complex multi-ported structure, its capacity has both power and performance implications. In this paper, we present a method to reduce the demand for physical registers by sharing registers that have the same value or non-overlapping useful lifetimes. It is important in any implementation that the benefits justify the cost. We propose a *simple* register sharing scheme and show that it provides almost *half* the benefits of the most aggressive scheme, but with significantly less implementation complexity.

In an SMT architecture, reducing the demand for physical registers using our simple scheme results in a 25.7% performance improvement, which is equivalent as having 25% more registers and no register sharing. In future work, we plan to explore additional opportunities to collapse registers and to combine this scheme with a virtual physical register design, where we expect the late binding of values to physical registers to increase the opportunities for sharing and also further simplify the register sharing implementation.

References

[1] V. Agarwal, M. Hrishikesh, S. Keckler, and D. Burger. Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures. In *International Symposium on Computer Architecture*, pages 248–259, Vancouver, Canada, June 2000.

[2] S. Balakrishnan and G. Sohi. Exploiting Value Locality in Physical Register Files. In *International Symposium on Microarchitecture*, pages 265–276, San Diego, California, December 2003.

[3] R. Balasubramonian, S. Dwarkadas, and D. Albonesi. Reducing the Complexity of the Register File in Dynamic Superscalar Processors. In *International Symposium on Microarchitecture*, pages 237–248, Austin, Texas, December 2001.

[4] J. Cruz, A. González, M. Valero, and N. Topham. Multiple-Banked Register File Architectures. In *International Symposium on Computer Architecture*, pages 316–325, Vancouver, Canada, June 2000.

[5] J. Farrell and T. Fischer. Issue Logic for a 600-Mhz Out-of-Order Execution Microprocessor. *IEEE Journal of Solid-State Circuits*, 33(5):707–712, May 1998.

[6] A. Gonzalez, J. Gonzalez, and M. Valero. Virtual-Physical Registers. In *International Symposium on High-Performance Computer Architecture*, pages 175–184, Las Vegas, Nevada, January–February 1998.

[7] M. Gowan, L. Biro, and D. Jackson. Power Considerations in the Design of the Alpha 21264 Microprocessor. In *Design Automation Conference*, pages 726–731, San Francisco, California, June 1998.

[8] S. Jourdan, R. Ronnen, M. Bekerman, B. Shomar, and A. Yoaz. A Novel Renaming Scheme to Exploit Value Temporal Locality through Physical Register Reuse and Unification. In *International Symposium on Microarchitecture*, pages 216–225, Dallas, Texas, November–December 1998.

[9] R. Kessler. The Alpha 21264 Microprocessor. *IEEE Micro*, 9(2):24–36, March 1999.

[10] E. Larson, S. Chatterjee, and T. Austin. MASE: A Novel Infrastructure for Detailed Microarchitectural Modeling. In *International Symposium on Performance Analysis of Systems and Software*, pages 1–9, Tucson, Arizona, November 2001.

[11] M. Martin, A. Roth, and C. Fischer. Exploiting Dead Value Information. In *International Symposium on Microarchitecture*, pages 125–135, Research Triangle Park, North Carolina, December 1997.

[12] J. Martinez, J. Renau, M. Huang, M. Prvulovic, and J. Torrellas. Cherry: Checkpointed Early Resource Recycling in Out-of-order Microprocessors. In *International Symposium on Microarchitecture*, pages 3–14, Istanbul, Turkey, November 2002.

[13] M. Moudgill, K. Pingali, and S. Vassiliadis. Register Renaming and Dynamic Speculation: An Alternative Approach. In *International Symposium on Microarchitecture*, pages 202–213, Austin, Texas, December 1993.

[14] M. Postiff, D. Greene, S. Raasch, and T. Mudge. Integrating Superscalar Processor Components to Implement Register Caching. In *International Conference on Supercomputing*, pages 348–357, Sorrento, Italy, June 2001.

[15] R. Russell. The Cray-1 Computer System. *Readings in Computer Architecture*, 2000.

[16] D. Sima. Register Renaming Techniques. In V. Oklobzija, editor, *The Computer Engineering Handbook*, chapter 6.2, pages 6.6 – 6.20. CRC Press, 2002.

[17] L. Tran, N. Nelson, F. Ngai, S. Dropsho, and M. Huang. Dynamic Register Sharing for Register Pressure Reduction: Design Issues and Considerations. Technical report, Electrical & Computer Engineering Department, University of Rochester, January 2004.

[18] D. Tullsen, S. Eggers, and H. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *International Symposium on Computer Architecture*, pages 392–403, Santa Margherita Ligure, Italy, June 1995.

[19] S. Wallace and N. Bagherzadeh. A Scalable Register File Architecture for Dynamically Scheduled Processors. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 179–184, Boston, Massachusetts, October 1996.

[20] K. Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, 6(2):28–40, April 1996.

[21] J. Yi and D. Lilja. Improving Processor Performance by Simplifying and Bypassing Trivial Computations. In *International Conference on Computer Design*, pages 462–465, Freiburg, Germany, September 2002.

[22] J. Zalamea, J. Llosa, E. Ayguadé, and M. Valero. Two-level Hierarchical Register File Organization for VLIW Processors. In *International Symposium on Microarchitecture*, pages 137–146, Monterey, California, December 2000.

[23] Y. Zhang, J. Yang, and R. Gupta. Frequent Value Locality and Value-Centric Data Cache Design. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 150–159, Cambridge, Massachusetts, November 2000.