Physical Register Inlining

Mikko H. Lipasti[1], Brian R. Mestan[2], and Erika Gunadi[1]

[1]*Department of Electrical and Computer Engineering*
University of Wisconsin-Madison
mikko@ece.wisc.edu, egunadi@wisc.edu

[2]*IBM Microelectronics*
IBM Corporation - Austin, TX
bmestan@us.ibm.com

Abstract

Physical register access time increases the delay between scheduling and execution in modern out-of-order processors. As the number of physical registers increases, this delay grows, forcing designers to employ register files with multicycle access. This paper advocates more efficient utilization of a fewer number of physical registers in order to reduce the access time of the physical register file. Register values with few significant bits are stored in the rename map using physical register inlining, a scheme analogous to inlining of operand fields in data structures. Specifically, whenever a register value can be expressed with fewer bits than the register map would need to specify a physical register number, the value is stored directly in the map, avoiding the indirection, and saving space in the physical register file. Not surprisingly, we find that a significant portion of all register operands can be stored in the map in this fashion, and describe straightforward microarchitectural extensions that correctly implement physical register inlining. We find that physical register inlining performs well, particularly in processors that are register-constrained.

1. Introduction and Motivation

One of the critical delay paths in a modern superscalar processor is through the physical register file. The relentless demand for higher performance drives designers to deeper pipelines to enable higher frequencies [21, 9], wide superscalar issue to extract maximal instruction-level parallelism, and support for multiple threads [23, 8] to maximize throughput. All three of these objectives exacerbate design of the physical register file [6,24], since the minimum number of physical registers required is determined by the number of threads, and the number of additional registers for inflight instructions should be greater than the product of the pipeline depth (between the rename and commit stages) and issue width. For example, a first-generation out-of-order processor with a short 5-stage pipeline that supported 4-wide issue (e.g. the MIPS R1000 [25]) would need O(4x3) or O(12) physical registers to hold the results of inflight instructions in addition to the base set of physical registers needed to contain committed architected registers (usually 32 for RISC instruction sets but considerably fewer for CISC instruction sets). A future processor with a 30-stage execution pipeline that supports 8-wide issue would need O(240+32) physical registers. Adding support for 4-way-multithreading [23] increases this to O(240+4x32 = 368). In brief, the number of physical registers needed in an out-of-order processor is O(issue-width*pipeline-depth + threads*architected-registers). Furthermore, long-latency instructions such as cache misses exacerbate this problem since they hold physical registers for the entire duration of the cache miss, and also delay the release of registers assigned to all subsequent instructions. Processors that attempt to mask cache miss latencies by continuing to fetch and execute new instructions must provide register file storage for the results of all the intervening instructions, driving demand for hundreds of physical registers.

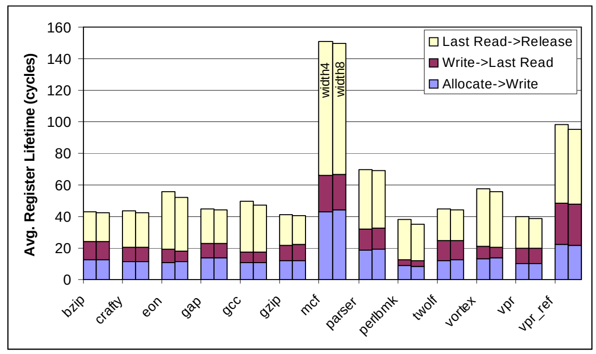

Figure 1. Average Register Lifetime. The stacked bars shows average physical register lifetime broken down into the interval between allocation and register write, register write and last read, and last read and release. The left stacked bar is for 4-wide machine, while the right stacked is for 8-wide machine, both with 64 physical registers. Machine model used is listed at Table 1.

A large register file is detrimental to performance primarily because it increases the delay between the processor's dynamic scheduling stage and its execution stage, since instruction operands have to be retrieved in this interval. Access to a large register file can be pipelined over multiple cycles, thus providing sufficient throughput and not necessarily degrading core frequency, but the additional pipeline stages contribute to the problems caused by speculative scheduling [11, 8], and increase the size of one of the most critical "loose loops" [1] in the processor. Furthermore, pipelined register file access increases the total number of pipeline stages, further increasing the demand for physical

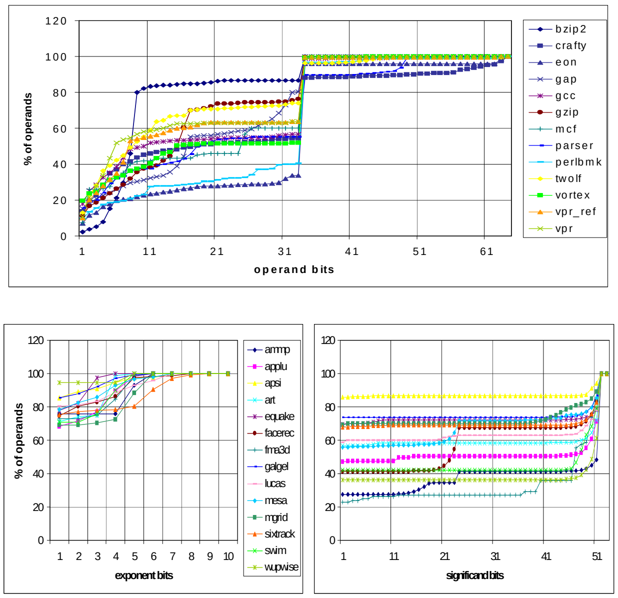

Figure 2. Operand Significance. The top graph shows the dynamic cumulative distribution of the number of bits needed to represent integer operands for the SPEC 2000 integer benchmarks. The bottom graph shows the dynamic cumulative distribution of the number of bits needed to represent floating point operands for SPEC 2000 fp benchmarks (left: exponent bits, right: significand bits). Approximately half of all floating point operands actually contain only zeroes.

registers to hold results of inflight instructions.

Numerous researchers have investigated various solutions to this pressing problem. For example, register caching or hierarchical solutions, as well as multi-banked register files have been proposed and investigated (e.g. [22,26,20,5,1]). Exploiting program semantics to release registers early was studied in [16] and more recently in the context of simultaneous multithreading by [14], while a checkpointing schemes that enable early release of physical registers and other resources were described in [28] and [29]. Differentiated treatment for short-lived variables was studied in [15], while value-based optimizations were described in [10].

Analysis of physical register lifetimes reveals some interesting opportunities for maximizing the utilization of physical registers in an out-of-order processor. Figure 1 shows physical register lifetime for the SPEC2000 integer programs broken down into three phase.

1. *Allocated, but not written.* Conventional superscalar processors allocate registers in program order when instructions are inserted into the out-of-order window. Using this conservative allocation policy avoids deadlock, but reduces effective utilization of the registers.

2. *Written, last read not done.* This phase represents the time during which the register value is live, since the register has been written but the last consumer has not yet read it.

3. *Last read done, but not released.* In this phase the last read has occurred, and the register is dynamically dead. It could be released, but the conventional approach for register deallocation waits until the next write to the same architected register has been committed, guaranteeing that the register is dead.

It can be seen from Figure 1, while phases 1 and 2 are not unimportant, average register lifetime is dominated by phase 3. In fact, prior work in early deallocation has focused on exploiting program semantics or relying on the compiler to communicate register liveness, so that hardware can deallocate dead registers [16, 14]. The main drawbacks of these approaches are that they require compiler analysis to identify dead register values, and instruction set support to communicate liveness to the hardware. While such support is conceptually straightforward, it has failed to materialize in production compilers or instruction sets.

An alternative approach to register renaming reduces the magnitude of the third phase (from last read to release) [27]. Specifically, it avoids waiting for the next write of the architected register to commit by deallocating the register as soon as the register has been unmapped and all reads of the register are complete. This scheme is implemented by adding an unmap flag, a counter, and a complete flag for each physical register. The unmap flag will be set to true as soon as the architected register pointing to that physical register is remapped to another physical register. The complete flag is set to false when the physical register is removed from the free pool, and is set to true when it is written. The counter keeps track of the number of issued instructions that need the value in the physical register but have not yet read it. As soon as 1) the complete flag is true, 2) the unmap flag is true for current and checkpointed copies, and 3) the counter is zero, the register can be freed.

Similarly, this paper advocates an approach for reducing the impact of phase 3 of a register's lifetime. Specifically, we observe that significance compression [2,4] can be exploited to eliminate the cost of storing many operands during phase 3 of their lifetime. Moreover, any operand that can be represented more efficiently with a small number of bits can be stored elsewhere, freeing up space in the register file and effectively eliminating the detrimental register file occupancy caused by this operand. Figure 2 plots the dynamic cumulative distribution of the number of bits needed to represent all integer operands for the SPEC 2000 integer benchmarks and all floating point operands for the SPEC 2000 floating point benchmarks. For example, we see that only 10 bits of storage can represent approximately half (worst case 23% and best case 82%) of all integer register operands. Even though there is no significant amount of narrow values in floating point register operands, about 77% (worst case 68% and best case 94%) of all exponents and about 54% (worst case 23% and best case 86%) of all significands contain only zeroes or ones. This indicates a significant opportunity for freeing physical registers early, once their values have been computed and found to be compressible.

Of course, the compressed values still need to be stored somewhere so subsequent instructions can read them. We

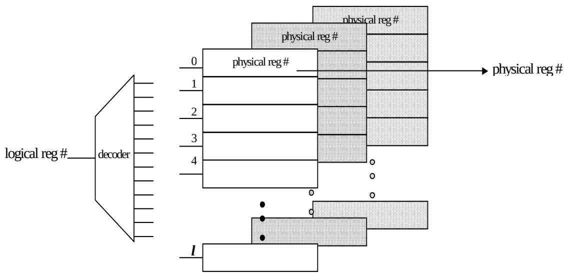

Figure 3. RAM Map Table. In RAM map tables, the total number of entries is equal to the number of *logical* registers. A logical register number is decoded to drive a single word line. The entry selected is a pointer to the physical register location that holds the latest value produced for that logical register. Shadow copies of the physical register numbers for each entry are needed for control speculation and for precise exceptions.

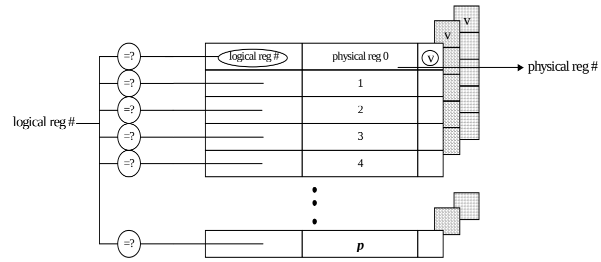

Figure 4. CAM Map Table. In CAM map tables, the total number of entries is equal to the number of *physical* registers implemented in the machine. The logical register number is compared to the logical register number at each entry. If the entry is valid, the corresponding physical register number is output from the map. Shadow copies of the map are maintained by checkpointing the valid bits.

propose a new technique called *physical register inlining*, which stores the narrow values in the processor's rename table or map table instead of in the physical register. The map table is used to store mappings between architected register names and the physical registers that contain the most recent instance of those registers. Since modern processors have a large number of physical registers, the map entries themselves are large enough to represent many operands as immediate values (e.g. a 256-entry physical register file requires 8-bit register identifiers in the map; based on Figure 2, a sizable fraction of integer operands can be stored in less than 8 bits directly in the map table and a significant portion of floating point operands can be stored even using only 1 bit). *Physical register inlining* simply reserves part of the name space provided by the available storage in the map table for immediate values, rather than consuming all of it for pointers into the register file. This is analogous to a common programming optimization: inlining pointer-based data structures whenever the object being pointed to is smaller than the pointer, or whenever doing so improves spatial locality. Conceptually, we are introducing a second addressing mode into the map table: instead of simply a register indirect mode, we also support an immediate mode. We find that such a scheme can dramatically lower register file pressure and can improve performance by approximately 7.3% in a 4-wide machine model, representing 96.8% of the performance of an idealized machine with unlimited physical registers, and approximately 14.7%% in an 8-wide machine model, representing about 83.5% of the performance of an idealized machine with unlimited physical registers.

Further details and microarchitectural implications of the proposed scheme are detailed in Section 2 and Section 3. Section 4 details our machine model, while Section 5 presents detailed performance analysis of the proposed physical register inlining scheme, and compares its performance with the previous work using reference conter and flags to release physical registers early. Section 3.5 also shows that the two techniques are complementary and can be quite easily integrated for further performance benefit, reaping speedup of 2.9%-15.9% (94.9%-99.9% of ideal performance) in a 4-wide configuration, and speedups of 10.0%-26.9% (68.6%-95.3% of ideal performance) in an 8-wide configuration. Finally, we conclude and discuss future work in Section 6.

2. Microarchitectural Issues

Modern out-of-order processors rename the logical register identifiers that specify control and data flow through a program to physical registers that are implemented in the machine. In this section, we review register renaming logic, describe the microarchitecture structures used to implement register renaming, and discuss the management of these structures.

2.1. Register Rename Map Tables

Register renaming logic maps logical register names in a program to physical registers that reside in the machine. The objective of introducing this level of indirection is to remove false name dependences (both Write-After-Write and Write-After-Read) that artificially limit the amount of parallelism in a program, and to offer a mechanism to recover state for control speculation. In order to remove these false dependences in a program and to enable deep speculation, the physical register file is larger than the number of architected registers specified in the instruction set. Therefore, a single logical register name can reference multiple physical register locations, with each reference corresponding to different points in time of the sequential program.

For each logical destination register in an instruction, renaming logic allocates a physical register location and records this mapping so that subsequent logical input registers will correctly reference the physical register holding their latest value. The key structure used by modern processors to keep track of the renamings is a register map table

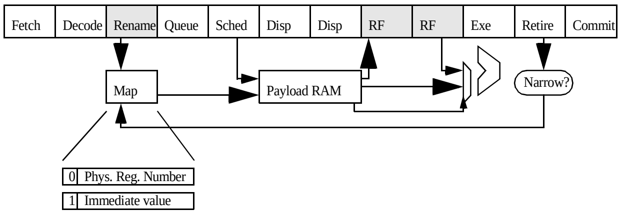

Figure 5. Pipeline Diagram. Map entries read during the rename stage contain either a physical register number or immediate value, determined by the addressing mode bit. The entry is stored in the payload RAM until the instruction is scheduled. Subsequently, the register number (if present) is used to access the physical register file, or the immediate value is fed directly as an operand to the execution stage. Execution result is written back to the register file at retire stage. At commit stage, the old definition of the destination register is freed.

(also referred to as register alias table) [8, 25]. The two most common types of register maps are the RAM and CAM tables [19, 18]. In both types, an associated free list is also used in order to quickly find registers which are free to be allocated to the next output logical register name. The two types of tables are described below and depicted in Figure 3 and Figure 4.

• ***RAM Map.*** In RAM map tables, the total number of entries in the map is equal to the number of *logical* registers (i.e. architected registers). A logical register number is decoded to select a single entry in the table. This entry holds the physical register number associated with this logical register. Mappings are checkpointed by copying the contents of the present table to a shadow table for recovering for precise exceptions and from control flow misspeculation. Therefore, multiple copies of the map table exist corresponding to each branch of control flow (a checkpoint is taken only for each branch in the MIPS R10000 [25]). Notice that the structure of the RAM map looks very similar to a smaller version of a physical register file. Rather than holding a 32- or 64-bit value, a RAM map holds just enough bits to specify the physical register location. Physical register inlining attempts to exploit this storage space naturally available in the map table to hold values which can be represented in this smaller space.

• ***CAM Map.*** In CAM (content-addressable memory) map tables, the number of entries in the map is equal to the total number of *physical* registers. In this type of map, a logical register number is compared against the logical register number stored in each entry of the table, and a valid bit is polled. A matching entry then outputs the corresponding physical register number out of the table. The valid bit is necessary since a single logical register can point to multiple physical registers, but only one mapping is valid at a given time. When a new physical register is allocated, the logical register number is copied into the table and the valid bit in the old mapping is cleared. This valid bit is not needed in RAM tables since the number of entries in the table is precisely the number of logical registers. Therefore, in RAM tables, old mappings are simply overwritten when a physical register is allocated. Checkpointing the table in CAM maps is slightly easier than RAM tables, as only the valid bits need to be copied to backup shadow maps [18]. Since CAM maps encode physical register numbers positionally, they are not amenable to storing narrow values, as advocated in this paper. In a CAM map, only a single physical register number can tag-match for a given logical register. This is fine, since there is never more than one valid logical register that corresponds to any given physical register. However, if the physical register number is used to encode a value, the CAM is limited to storing only a single logical register per unique value. Hence, for example, if the value 0 occurs in 2 logical registers at the same time, only one of those instances can be stored in a CAM map. For this reason, we conclude that physical register inlining is not practical with CAM maps, but only with RAM maps. We argue that CAM maps do not scale well to large numbers of physical registers, and hence are less likely to be used in future machines, so this is not a fundamental problem with our proposed technique.

2.2. Allocating and Freeing Map Table Entries

Conventional processors allocate a new physical register when an instruction is decoded and inserted into the instruction window. At this time, an entry in the map table is entered for the instruction's logical destination register and a physical register is taken off the free list. A physical register is released when a subsequent instruction that has the same logical destination register is committed. At this time, the physical register is put back on the free list, and its entry in the map is made available to the register renaming logic for reuse.

2.3. Checkpoint and Recovery of Register Map

In order to support precise exceptions and recovery from control flow speculation, multiple copies of the register map table exist. Figure 3 and Figure 4 show the amount of storage that is necessary in RAM and CAM map tables to enable recovery to a previous known state. The existence of these shadow maps is important to note, since the proposed physical register inlining mechanism must be aware of these maps and manage them properly. We discuss this management in the following sections.

3. Physical Register Inlining

In this section, we outline our proposed scheme, identify some possible problems with it, and detail our solutions to these problems. As discussed, physical register inlining exploits storage available in the map table to store an immediate value, whenever the register value can be represented as a narrow operand, or a register identifier, when the value is either unknown (i.e. yet to be computed) or known to contain too many significant bits to fit into the map entry. Figure 5 illustrates our pipeline, and shows how the map is accessed during the rename stage to determine the current

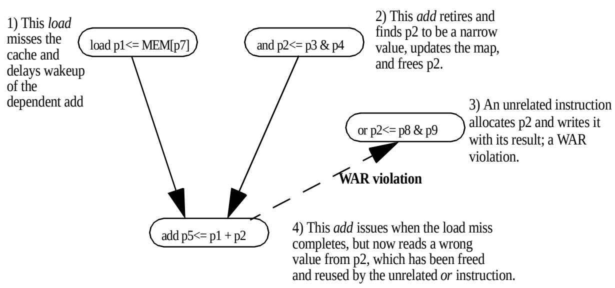

Figure 6. Example of WAR Violation. Early deallocation of physical registers can cause dependent instructions that have already read the map entry to read a physical register that has already been reallocated to an unrelated instructions**.**

mapping for an architected register, how that map entry is stored in the reorder buffer's payload RAM[1] until the instruction is scheduled, and finally, how the map entry is used to either access the physical register file, or is itself used to provide the immediate operand to the instruction in the execution stage. After execution, the result of each instruction is checked to see if it can be represented in the map, and the corresponding map entry is updated if that is the case.

Supporting such a scheme requires changes to several areas of the processor. The following sections outline changes needed to the processor's dataflow, the map table itself as well as the processor's control logic.

3.1. Modifications to the Data Flow

First of all, the execution stage must allow immediate operands that are read from the payload RAM to be delivered to the ALU inputs. Since both integer ALUs and address generation units usually already support immediate operands for one of two operands, the only real change required is to add symmetric support, since either the left or the right operand may now be delivered from the payload RAM instead of the register file. Sign extension hardware is added between payload RAM and the ALU input in order to avoid increasing the size of payload RAM. Unless the operand delay paths are unbalanced due to physical design constraints, this should not affect the critical delay through the execution stage. The second change to the dataflow consists of significance checking logic that verifies whether or not all n high-order bits of a computed results are either 1 or 0. While this logic function has relatively high fan-in, it is not speed-critical, and could easily be pipelined, if necessary. We assume this logic operates in the retire stage, in time to identify the narrow operand and write it to the map in the following stage. The final change requires adding a narrow datapath back to the map table, so that the narrow register value can be written into the map.

1. Payload RAM refers to the subset of the reorder buffer state that is accessed after an instruction is scheduled for execution but before it actually executes, and includes fields like the opcode specifier and source and destination registers, which form the payload for the execution pipeline.

3.2. Modifications to the Map Table

A minor modification is required to the free-list management portion of the map table to allow registers to be deallocated and placed on the free list from the retire stage as well as the commit stage. The free-list manager must also be tolerant of duplicate deallocations of the same physical register. Duplicate deallocations can happen when a physical register is found to be narrow and deallocated at retire stage by its producer. However, before it is found to be narrow, the next writer to the same architectural register is renamed. Once this second instruction commits, it will release the previous physical register since it still does not know that the previous physical register has been released. In this case, the free-list manager must have a scheme that allows the physical register to be placed on the free list only once for every time it is allocated, otherwise the free list will contain duplicate entries of the same physical register. We do not anticipate that either of these changes would be particularly challenging.

A more significant modification to the map table is that the map entries themselves need to be writable from the retire stage in addition to the rename stage. As described in Section 2, conventional map entries are only updated when new instructions are renamed, since the mappings for their destination registers need to be recorded appropriately. In our scheme, an additional write port is needed to update an entry once the instruction has retired.

This problem is compounded by the fact that the map is repeatedly checkpointed to allow efficient and correct recovery from mispredicted branches. Since each checkpoint now contains copies of the original map entry, a late update requires each copy to be updated as well. While this

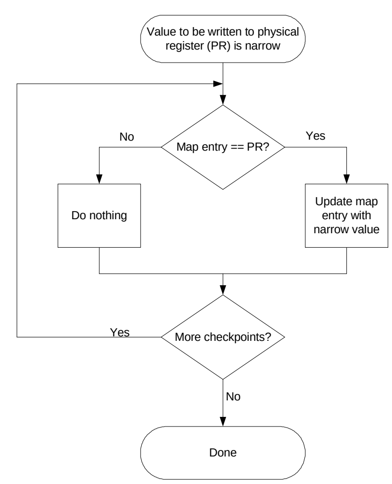

Figure 7. Narrow Value Update in the Map Table. In order to avoid WAW violations, the narrow value is copied to the map entry only if the map entry has not been remapped to other physical registers.

appears difficult, in fact only the current map needs to be updated immediately, to guarantee that subsequently renamed instructions see the effects of the write. The checkpointed copies can be updated lazily, since they are only needed on a branch misprediction recovery. Some independent control logic to check and copy the narrow value written to the current map can be added to each of the checkpointed copy, with a write on the second write port as a trigger. Although feasible, modifications necessary to the map table to enable this late update are not trivial. Trading complexity for performance, we propose a second solution to the shadow mapping update by using a reference counting scheme similar to [29]. Each map checkpoint increments the reference count of each physical register that it points to. As map checkpoints are retired, these reference counts are decremented. A physical register cannot be deallocated until its reference count is zero, effectively avoiding any problems with stale pointers in the checkpointed maps.

3.3. Changes to Control Logic

The astute reader has already noticed that there is a potential write-after-read violation inherent in the late update discussed in the preceding sections. Specifically, any consumer instruction that reads the map table entry for one of its source operands and copies it to its entry in the payload RAM will fail to see the late update performed by the producer instruction during its retire stage. Since only the map is changed, the consumer instruction will continue to expect its operand to be delivered from the physical register file, and not from an immediate value in the payload RAM. While most dependent instructions are issued in the shadow of the producer instruction, and usually pick up their operands from the processor's bypass network, there can be cases where the consumer is delayed due to structural hazards (i.e. failing to be selected) or other unresolved data dependences. Figure 6 shows an example scenario in which one input operand to an add instruction is delayed due to a cache miss, while the second input is found to be narrow, the map entry for it is updated, and the physical register is deallocated. An unrelated *or* instruction allocates this physical register and overwrites it with its own result. Once the dependent add instruction issues following completion of the cache miss, it reads this new unrelated value from the register file. We have found that this scenario is generally rare, but can occur for up to 1% of instructions for some benchmarks and machine models.

The problem we face here is identical to the one faced by a dynamic program optimizer that attempts to relocate or inline data objects to improve locality. Since there may be multiple pointers to an object, all pointers must be located and updated to point to the new location. This problem is similar in complexity to garbage collection in languages without explicit heap management, as all possible pointer locations must be checked for stale pointers before storage can be reclaimed. Operating systems employ page tables and virtual memory to solve a similar problem, but must also deal with stale entries in translation lookaside buffers (TLBs) whenever a virtual memory mapping is updated. In our case, we must find stale register identifiers in both the map checkpoints (discussed in Section 3.2) as well as all payload RAM entries that contain copies of the relevant map entry.The payload RAM can contain stale pointers to input operands, as well as pointers to deallocation candidates (i.e. the previous instance of the instruction's output register); both entries must be found to solve the problems outlined in the preceding sections.

One can envision various solutions to this problem, ranging from a simple detection mechanism that replays violating instructions through the map so they pick up the correct register specifier, all the way to fully-associative search and update of payload RAM entries (hence now a CAM) to avoid any penalty. While a detect-replay mechanism looks

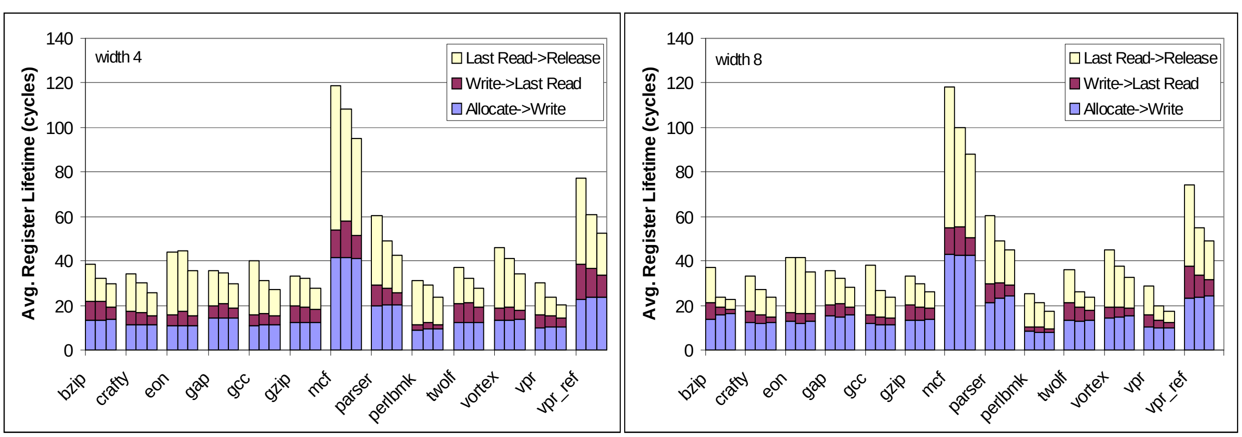

Figure 8. Reduction in Register Lifetime. This graph shows average physical register lifetime for the baseline 4- and 8-wide machine models (see Section 4) in the left stacked bars, with physical register inlining (refcount+ckptcount) enabled in the middle stacked bars, and with both physical register inlining (refcount+ckptcount) and early release option enabled in the right bars**.**

simple enough to implement, a WAR violation will trigger instructions replay all the way up to the previous branch. We think that this is too costly and we do not investigate this option any further.

In the performance evaluation in Section 5, we model two schemes that bound the performance effect of such approaches. The first approach is a reference counting mechanism that is incremented every time a consumer instruction references a physical register mapping during renaming. The counter will be decremented once each consumer instruction is done reading the register. A non-zero counter means that there are still some active instructions waiting to read the specific register. Hence, it will not be freed until the counter becomes zero. The second approach is a fully associative search of payload RAM entries to avoid any penalty. We are aware that this approach is not realistic since the time it takes to do the search and update may not be trivial. However, we still include this approach in our performance comparison as an ideal case of physical register inlining scheme. We find that for some benchmarks, an efficient approach is necessary for high performance, since scenarios such as the one shown in Figure 6 are not infrequent.

Recovering from branch mispredictions and exceptions requires careful updating of the reference counters. In order to recover the correct state of the counters quickly after a misprediction or an exception, the counters are also saved in the map checkpoints.

3.4. Rename Table Hazards

Since the rename table needs to be writeable from the retire stage in addition to the rename stage as pointed out in Section 3.2, there are two possible hazards that can occur at the rename table: RAW hazards and WAW hazards. RAW hazards occur when consumers have renamed their source operand to a register pointer at the rename stage, but before they actually read the value, the producer changes the pointer to the actual data and frees the physical register at retire stage. This hazard corresponds to the register file WAR violation discussed in Section 3.3 and can be treated using the same schemes: reference counting or instantenous search-update of the payload RAM. Since the narrow value is also written to the physical register file as well as to the rename table in retire stage, reference counting will make sure that the physical register does not get freed and overwritten by other instructions before active consumers read it. That way, consumers will still get the correct value even though register pointer in the rename table entries are changed to the narrow value.

The second hazard, a WAW, can happen when a narrow value operand is written to a map entry that is already remapped to other registers. In order to avoid this violation, control logic must ensure that the physical register of the narrow value to be written is the same as the physical register in the map entry. For the lazy checkpoint update scheme, this checking, as described in Figure 7,has to be done for all copies of the map table.

3.5. Integrating PRI With Early Release

Realizing that there are still some operands value that are not narrow, more register lifetime can be reduced by combining physical register inlining scheme with previous work to release registers early [27]. Since reference counting is already used in our scheme to avoid WAR violations and map table checkpoint update, it will not be too challenging to add complete flags and unmap flags to our scheme. We think these two schemes complement each other and more register lifetime can be reduced as shown in Figure 8.

4. Machine Model and Benchmarks

Our execution-driven simulator used in this study is

Table 1: Machine Configurations

	4-wide	8-wide
Out-of-order Execution	4-wide fetch/issue/commit, 512 ROB, 256 LSQ, 32-entry scheduler, 64 physical register, 64 floating point register, speculative scheduling, selective recovery for latency mispredictions, fetch stops at first taken branch in a cycle	8-wide fetch/issue/commit, 512 ROB, 256 LSQ, 512-entry scheduler, 64 physical register, 64 floating point register, speculative scheduling, selective recovery for latency mispredictions, fetch stops at first taken branch in a cycle
Branch Prediction	Combined bimodal (4k entry) / gshare (4k entry) with a selector (4k), 16 RAS, 1k-entry 4-way BTB, at least 11 cycles taken for misprediction recovery	
Memory System (latency)	32KB 2-way 32B line IL1 (2), 32KB 4-way 16B line DL1 (2), 512KB 4-way 64B line unified L2 (12), main memory (150)	
Physical register inlining	Integer -- all values with 7 or fewer significant bits are stored in the map table Floating Point -- all values that are all zeroes or ones are stored in the map table	Integer -- all values with 10 or fewer significant bits are stored in the map table Floating Point -- all values that are all zeroes or ones are stored in the map table
Physical register inlining WAR recovery policy	PRI-refcount: WAR conditions are avoided completely by keeping track if there are still some reference to the physical register to be freed PRI-ideal: WAR conditions are avoided completely; there is no penalty	

derived from the *SimpleScalar* / *Alpha* 3.0 tool set [3], a suite of functional and timing simulation tools for the Alpha AXP ISA. Specifically, we extended *sim-outorder* to perform speculative scheduling with selective recovery, and to model a finite physical register file and scheduler. In this pipeline, instructions are scheduled in the scheduling stage assuming instructions have fixed execution latency and any latency changes (e.g. cache misses) cause all dependent instructions to be re-scheduled. We modeled a 12-stage out-of-order pipeline with 4- and 8-instruction machine width. The pipeline structure is illustrated in Figure 5. The detailed configurations of each machine model are shown in Table 1.

The SPEC2000 integer benchmark suite and SPEC2000 floating point benchmark suite are used for all results presented in this paper. All benchmarks were compiled with the DEC C compiler under the OSF/1 V4.0 operating system using -O4 optimization. Table 2 shows the benchmarks, input sets, the number of instructions committed, and IPC on 4 and 8-wide base machines. The large reduced input sets from [12] were used for all integer benchmarks except for *crafty*, *eon* and *gap*. These three benchmarks were simulated with the reference input sets since the reduced inputs were not available. We also simulated the vpr benchmark with both the reference input set and the reduced input set, since we observed that the two differed substantially. For all floating point benchmarks, reference input sets are used.

We used two machine models in our simulations; a 4-wide machine with a limited scheduler size (32 entries) to represent current-generation machines, and an 8-wide machine with a large scheduler (512 entries) to represent future machines. Both models assume 64 physical registers for each integer and floating point, 512-entry reorder buffers, aggressive branch predictors, and realistic cache sizes and memory latencies. A large reorder buffer and load/store queue eliminate window-size effects, instead allowing to focus our study on register file effects. We choose 64 physical registers because a larger register file has little sensitivity on SPEC2000 integer benchmarks as shown in Figure 9. All simulations are fast-forwarded for 400M instructions

Table 2: Benchmark Programs Simulated

Integer	Base IPC (4-wide)	Base IPC (8-wide)	Floating Point	Base IPC (4-wide)	Base IPC (8-wide)
bzip2	1.62	1.67	ammp	0.06	0.06
crafty	1.35	1.40	applu	2.05	2.20
eon	1.81	2.11	apsi	1.37	1.50
gap	1.55	1.59	art	0.37	0.38
gcc	1.16	1.23	equake	2.28	2.38
gzip	1.51	1.54	facerec	1.35	1.41
mcf	0.36	0.37	fma3d	1.91	1.94
parser	0.98	1.00	galgel	0.65	0.66
perlbmk	1.15	1.21	lucas	2.29	2.43
twolf	1.17	1.22	mesa	1.97	2.08
vortex	1.40	1.52	mgrid	1.54	1.59
vpr	1.36	1.42	sixtrack	1.38	1.44
vpr_ref	0.63	0.64	swim	1.86	1.99
			wupwise	1.83	1.86

and run for 100M instructions. The baseline IPC for both models is shown in Table 2. For the 4-wide model, we assumed that the map table could store up to 7 bits per operand; this is not unrealistic, given current-generation physical register file sizes. For the 8-wide model, we increased this to 10 bits, since Figure 2 shows that this can capture significant additional opportunity, and a slight increase in the map table entry size seems reasonable to us to capture this benefit. The number of bits used to store narrow operands in map table here does not exactly match the number of bits needed to store the physical register pointer because we think that a slight increase in the map table entry size seems reasonable.

5. Results and Analysis

We collected performance results for base machine models with 4-wide issue and 8-wide issue using a physical register file size of 64 for each integer and floating point. In the 8-wide case we also use a scheduler with effectively infinite entries (i.e. 512, which matches the ROB size) in order to remove stalls not caused by the window size or number of

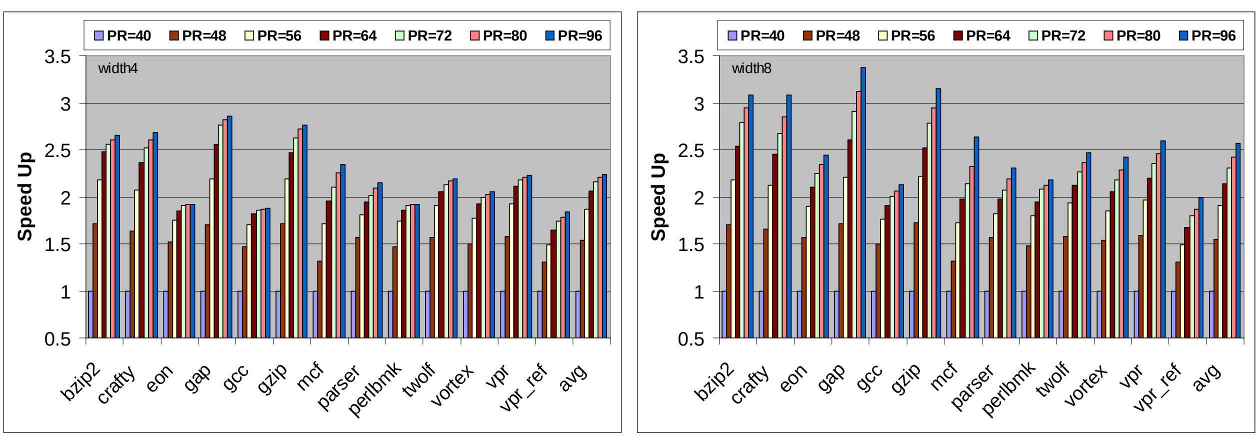

Figure 9. Register File Sensitivity Study. Data is normalized to PR=40. Machine configuration for this simulation is described in Table 1.

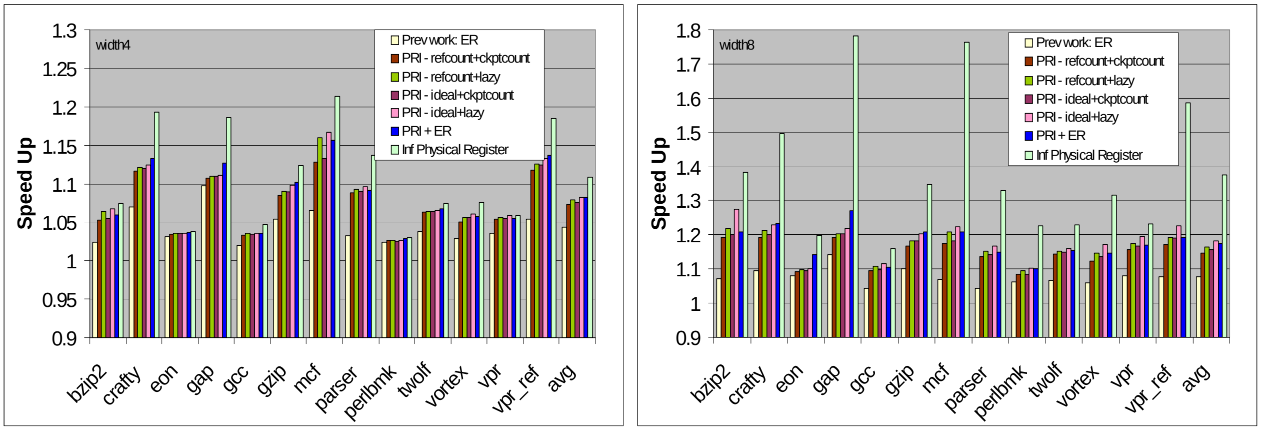

Figure 10. PRI Speed-Up for Integer Benchmarks.

physical registers. Viable proposals for building large schedulers exist (e.g. [13]), hence we assume some mechanism that enables large schedulers to be in place for this configuration. Our intent in using these two different machine configurations is to analyze PRI in both a conservative and aggressive machine with various degrees of register file pressure.

In both the conservative and aggressive machine models, we evaluate the three proposed schemes from Section 3: previous work in early release using counter and flags (ER), physical register inlining (PRI), and physical register inlining combined with early register release (PRI+ER). For each PRI case, we show results for reference counting mechanism to avoid WAR violations and results for instant payload RAM updates (ideal) as an ideal PRI scheme to study the benefit from a more aggressive mechanism. For each of WAR scheme, we show results for lazy checkpoint update mechanism and checkpoint reference counting. For PRI+ER scheme, we use reference counting and checkpoint counting scheme.

As discussed in Section 4, the 4-wide machine configuration uses an 8-bit rename identifier in the map table, allowing for 7 bits of value after subtracting the mode bit. Therefore, values that can represented in 7 bits or less use the immediate addressing mode described in Section 3. For the 8-wide machine configuration, a rename identifier of 11 bits is used, allowing for 10-bit values to be represented. For the floating point registers, we only inline registers that contain zero or one.

5.1. Integer PRI Results

The speed-ups in IPC for PRI across the SPEC2000 integer benchmarks are shown in Figure 10. On average, PRI outperforms the baseline machine by 7.3% and 14.8% for the 4-wide and 8-wide configurations, when using reference counting to avoid WAR violations and stale checkpoint entries. PRI with reference counting and the lazy update scheme results in a slightly higher speedup, 7.9% and 16.4% for 4-wide and 8-wide machine respectively. It shows that the checkpoint counting scheme does not trade too much performance for the complexity that it avoids. With ideal payload RAM updates, the speed up on average increases slightly to 7.6% and 15.6% for checkpoint counting and

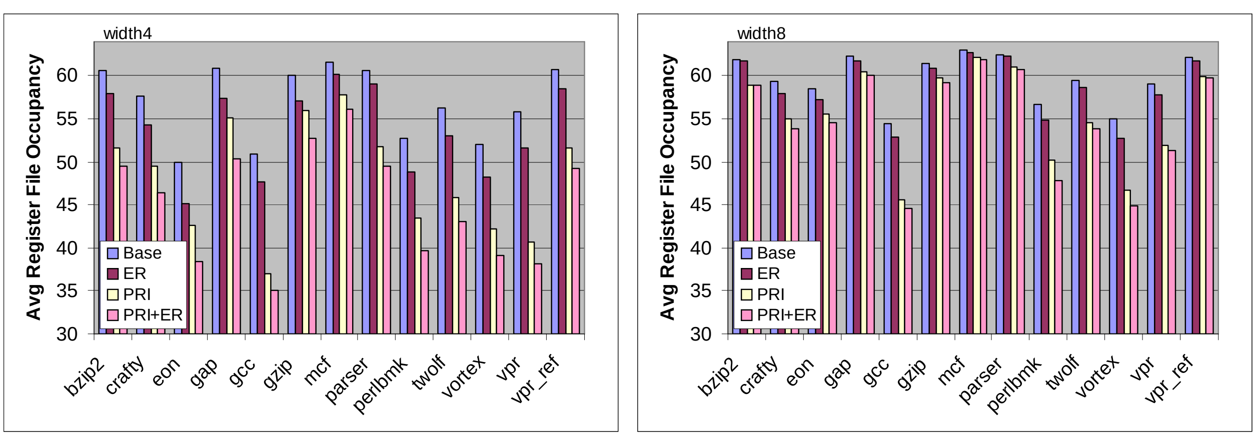

Figure 11. PRF Occupancy for Integer Benchmark.

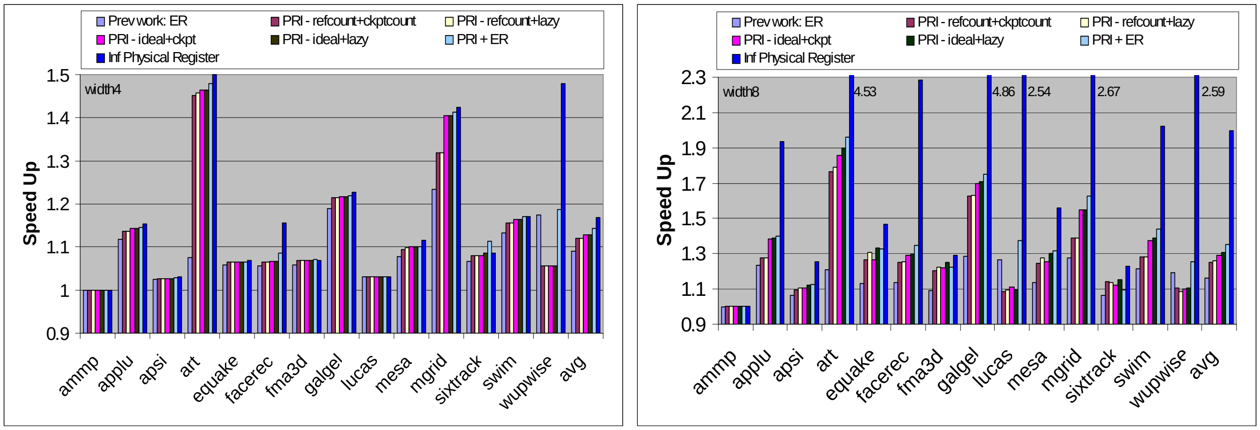

Figure 12. PRI Speed-Up for Floating Point Benchmark.

8.2% and 18.3% for lazy update, indicating that the reference counting scheme used performs close enough to the ideal case of physical register inlining. The benchmarks experiencing the largest performance improvement for the 4-wide machine configuration are *crafty* (10.8% with reference counting, 12.0% in the ideal case), *mcf* (12.8%, 13.3%), and *vpr_ref* (11.8%, 12.5%). For the 8-wide machine configuration, *bzip2* (19.2%, 20.0%) and *crafty* (21.5%, 20.1%) show the largest speed-ups.

Also, on average, physical register inlining with reference counting and checkpoint counting performs substantially better than the previous work on early register release: about 3.7% and 9.2% better for the 4-wide and 8-wide machines. Combining physical register inlining and early release, speedup of 8.3% and 17.5% over the baseline machine is gained. This shows that the schemes complement each other in reducing register file pressure. Performance improvement of 11.0% and 39% in infinite physical register case shows the upper limit of performance on a machine with no register file pressure.

Figure 11 shows the average register file occupancy for the base machine mode, early register release, physical register inlining (reference counting and checkpoint counting), and physical register inlining combined with early register release scheme. Due to increased register pressure, the reduction in average occupancy is not as dramatic in the 8-wide machine as it is in the 4-wide machine.

5.2. Floating Point PRI Results

Figure 12 shows the speed-ups in IPC for PRI across SPEC2000fp. On average, PRI with reference counting and checkpoint counting out-performs the baseline machine by 12.0% and 25.2% for the 4-wide and 8-wide, with 64 integer physical registers and 64 floating point physical registers. The performance increases slightly for PRI with reference counting and lazy update to 12.1% and 26.0%. With the ideal scheme and no checkpoint counting, the average speedup increases to 12.8% and 28.9%, representing the upper limit of physical register inlining performance. In addition, the combination of PRI with early release increases the speed-up to 14.3% and 35.3% for 4-wide and 8-wide machine.

However, as can be seen from Figure 10, the performance improvement varies widely from benchmark to benchmark due to the variation in resources needed. One interesting case is the ammp benchmark. As can be seen from the graph, there is no improvement across any scheme for the 4-wide machine and almost no improvement in 8-wide machine. In this case, it is clear that physical registers are not a performance bottleneck as there is also no improvement when the pressure on the register file is eliminated in the infinite physical register case.

Many of the other benchmarks do not show much improvement in the 4-wide machine. This happens because there are other limitations in the 4-wide machine, such as the number of issue queue (32 entries). However, when the issue queue limit is removed, it is clearly seen that limited physical registers are a major bottleneck, even after our optimizations for reducing register lifetime have been applied. The performance of 8-wide machine increases significantly when register file pressure is eliminated in the infinite physical register case, motivating even more sophisticated techniques for reducing register file pressure.

5.3. Results Summary

In summary, we find that PRI can lead to substantial performance improvement for both SPEC2000 integer benchmarks and SPEC2000 floating point benchmarks. The additional performance available without reference counting leads us to conclude that the extra hardware needed to directly update stale rename pointers in the payload RAM may be beneficial. This is particularly true for the more aggressive 8-wide machine model, where delaying register release longer due to a non-zero reference counter can cause the machine to slow down compared to the ideal case.

6. Conclusions and Future Work

This paper proposes a new microarchitectural technique for reducing pressure on the physical register file. Specifically, we advocate physical register inlining, where register values with few significant bits are stored directly in the map table, and the physical registers allocated to them are released. This approach reduces register lifetime dramatically, and improves performance by up to 12.8% (average 7.3%) for a 4-wide machine model and up to 19.2% (average 14.8%) for an 8-wide machine model. We also show that physical register inlining complements the previous technique of early register release, and that further speedup can be obtained when both techniques are applied in concert.

In future work, we plan to investigate the interaction of physical register inlining with delayed register allocation, as described in the virtual-physical register work [7]. We are also interested in the interaction of PRI with software-based techniques for deallocating dead registers. In particular, the presence of PRI enables a binary-compatible mechanism for the compiler to communicate the fact that a register is dead to the hardware. The compiler can simply insert a load-immediate of a narrow value to any register it deems dead, and the hardware is able to free the corresponding physical register by storing the narrow value in the map. We plan to investigate this opportunity in the future.

7. Acknowledgments

This work was made possible through generous equipment donations and financial support from IBM and Intel, and NSF grants CCR-0073440, CCR-0083126, EIA-0103670 and CCR-0133437.

References

[1] E. Borch, E. Tune, S. Manne, and J. Emer. Loose loops sink chips. In *International Symposium on High-Performance Computer Architecture*, February 2002.

[2] D. Brooks and M. Martonosi. Dynamically exploiting narrow width operands to improve processor power and performance. In *International Symposium on High-Performance Computer Architecture*, pages 13–22, 1999.

[3] D. C. Burger and T. M. Austin. The simplescalar tool set and version 2.0. Technical Report CS-TR-1342, Computer Sciences Dept., University of Wisconsin-Madison, 1997.

[4] R. Canal, A. Gonzalez, and J. E. Smith. Very low power pipelines using significance compression. In *International Symposium on Microarchitecture*, 2000.

[5] J.L. Cruz, A. Gonzalez, M. Valero, and N. T. Topham. Multiple-banked register file architectures. In *International Symposium on Computer Architecture*, 2000.

[6] K. I. Farkas, N. P. Jouppi, and P. Chow. Register file design considerations in dynamically scheduled processors. In *International Symposium on High-Performance Computer Architecture*, pages 40–51, 1996.

[7] A. Gonzalez, J. Gonzalez, and M. Valero. Virtual-physical registers. In *International Symposium on High-Performance Computer Architecture*, pages 175–184, 1998.

[8] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the pentium 4 processor, 2001.

[9] M.S. Hrishikesh, N. P. Jouppi, K. I. Farkas, D. Burger, S. W. Keckler, and P. Shivakumar. The optimal useful logic depth per pipeline stage is 6-8 fo4. In *International Symposium on Computer Architecture*, 2002.

[10] S. Jourdan, R. Ronen, 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*, 1998.

[11] I. Kim and M.H. Lipasti. Implementing optimizations at decode time. In *International Symposium on Computer Architecture*, June 2002.

[12] A.J KleinOsowski, J. Flynn, N. Meares, and D.J. Lilja. Adapting the SPEC2000 benchmark suite for simulation-based computer architecture research. In *Workshop on Workload Characterization*, 2000.

[13] A.R. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg. A large, fast instruction window for tolerating cache misses. In *International Symposium on Computer Architecture*, pages 59–70, 2002.

[14] J. L. Lo, S. S. Parekh, S. J. Eggers, H. M. Levy, and D. M. Tullsen. Software-directed register deallocation for simultaneous multithreaded processors. *IEEE Transactions on Parallel and Distributed Systems*, 10(9), 1999.

[15] L. A. Lozano and G. R. Gao. Exploiting short-lived variables in superscalar processors. In *International Symposium on Microarchitecture*, pages 292–302, 1995.

[16] M. M. Martin, A. Roth, and C. N. Fischer. Exploiting dead value information. In *International Symposium on Microarchitecture*, pages 125–135, 1997.

[17] T. Monreal, A. Gonzalez, M. Valero, J. Gonzalez, and V. Vinals. Delaying physical register allocation through virtual-physical registers. In *International Symposium on Microarchitecture*, 1999.

[18] S. Palacharla, N. P. Jouppi, and J. E. Smith. Quantifying the complexity of superscalar processors. Technical Report CS-TR-1996-1328, Computer Sciences Dept., University of Wisconsin-Madison, 1996.

[19] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *International Symposium on Computer Architecture*, pages 206–218, 1997.

[20] M. Postiff, D. Greene, S. Raasch, and T. N. Mudge. Integrating superscalar processor components to implement register caching. In *International Conference on Supercomputing*, pages 348–357, 2001.

[21] E. Sprangle and D. Carmean. Increasing processor performance by implementing deeper pipelines. In *International Symposium on Computer Architecture*, 2002.

[22] J. A. Swenson and Y. N. Patt. Hierarchical registers for scientific computers. In *International Conference on Supercomputing*, pages 346–353, 1988.

[23] D. M. Tullsen. *Simultaneous multithreading*. Thesis (ph.d.), University of Washington, Seattle, WA, USA, 1996.

[24] S. Wallace and N. Bagherzadeh. A scalable register file architecture for dynamically scheduled processors. In *International Conference on Parallel Architectures and Compilation Techniques*, October 1996.

[25] K. C. Yeager. Mips R10000 superscalar microprocessor. *IEEE Micro*, 1996.

[26] J. Zalamea, J. Llosa, E. Ayguade, and M. Valero. Two-level hierarchical register file organization for VLIW processors. In *International Symposium on Microarchitecture*, 2000.

[27] M. Moudgill, K. Pingali, and S.Vassiliadis. Register renaming and Dynamic Speculation: an Alternative Approach. In *Proceedings of the 26th International Symposium on Microarchitecture*, December 1993.

[28] 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*, Novermber 2002.

[29] H. Akkary, R. Rajwar, S.T. Srinivasan. Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors. In *Proceedings of the 36th International Symposium on Microarchitecture*, 2003.