IEEE Transactions on Parallel and Distributed Systems

Software-Directed Register Deallocation for Simultaneous Multithreaded Processors

Jack L. Lo[‡], Sujay S. Parekh, Susan J. Eggers, Henry M. Levy, and Dean M. Tullsen*

[‡]Transmeta Corp.
3940 Freedom Circle
Santa Clara, CA 95054

Dept. of Computer Science and Engineering
Box 352350
University of Washington
Seattle, WA 98195-2350

*Dept. of Computer Science and Engineering
University of California, San Diego
9500 Gilman Drive
La Jolla, CA 92093-0114

Abstract

This paper proposes and evaluates software techniques that increase register file utilization for simultaneous multithreading (SMT) processors. SMT processors require large register files to hold multiple thread contexts that can issue instructions out of order every cycle. By supporting better inter-thread sharing and management of physical registers, an SMT processor can reduce the number of registers required and can improve performance for a given register file size.

Our techniques specifically target register deallocation. While out-of-order processors with register renaming are effective at knowing when a new physical register must be allocated, they have limited knowledge of when physical registers can be deallocated. We propose architectural extensions that permit the compiler and operating system to (1) free registers immediately upon their last use, and (2) free registers allocated to idle thread contexts. Our results, based on detailed instruction-level simulations of an SMT processor, show that these techniques can increase performance significantly for register-intensive, multithreaded programs.

Keywords: multithreaded architecture, simultaneous multithreading, register file, architecture

1 Introduction

Simultaneous multithreading (SMT) is a high-performance microarchitecture that substantially improves processor performance by executing instructions from multiple threads every cycle. By dynamically sharing processor resources among the executing threads, SMT increases functional unit utilization, boosting instruction throughput and program speedups on a variety of workloads that include commercial databases and scientific applications in both multiprogrammed and parallel environments [5][10][28][31].

Previous research has examined several aspects of the SMT design, including instruction fetch mechanisms [25], cache organization [14], multiple-path execution [28], and synchronization hardware [26]. This paper focuses on another hardware feature that impacts SMT's cost-effectiveness: the organization and utilization of its register file. SMT raises a difficult tradeoff for register file design: while a large register file is required to meet the architectural and renaming register needs of the multiple thread contexts, smaller register files provide faster access times. Therefore, an SMT processor needs to use its register resources efficiently in order to optimize both die area and performance.

In this paper, we propose and evaluate software techniques that increase register utilization, permitting a smaller, faster register file, while still supporting multiple threads. Our techniques involve coordination between the operating system, the compiler, and the low-level register renaming hardware to provide more effective register use for both single-threaded and multithreaded programs. The result is improved performance for a given number of hardware contexts and the ability to handle more contexts with a given number of registers. For example, our experiments indicate that an 8-context SMT processor with 8 eight extra registers for register renaming (i.e. 264 total registers), managed with the techniques we present, can attain performance comparable to a processor with 100 extra physical registers for renaming.

Our techniques support the effective *sharing* of register contexts in an SMT processor, using register renaming to permit multiple threads to share a single global register file. In this way, one thread with high register pressure can benefit when other threads have low register demands. Unfortunately, existing register renaming techniques cannot fully exploit the potential of a shared register file. In particular, while existing renaming hardware is effective at allocating physical registers, it has only limited ability to identify register *deallocation* points; therefore it frees registers conservatively, possibly underutilizing them.

We have designed software support to expedite the deallocation of two types of dead registers: (1) registers allocated to idle hardware contexts, and (2) registers in active contexts whose last use has already retired. In the first case, when a thread terminates execution on a multithreaded architecture, its hardware context becomes idle if no threads are waiting to run. Although the registers allocated to the terminated thread are dead, they are not freed, because hardware register deallocation only occurs when registers in a new, active thread are mapped. This causes a potentially-shared SMT register file to behave like a partitioned collection of per-thread registers. We propose a privileged instruction that deallocates idle SMT contexts earlier than is done on current multithreaded machines. Our experiments show that by notifying the hardware of OS scheduling decisions, performance with a register file of size 264 is boosted by more than 3 times when 2 or 4 threads are running.

To address the second type of dead registers, those in active threads, we investigate five mechanisms that allow the compiler to communicate last-use information to the processor, so that the renaming hardware can deallocate registers more aggressively. Without this information, the hardware must conservatively deallocate registers only after they are redefined. Simulation results indicate that these mechanisms can reduce register deallocation inefficiencies; in particular, on small register files, the best of the schemes attains speedups of up to 2.5 for some applications, and 1.6 on average. All the register deallocation schemes could benefit any out-of-order processor, not just SMT.

The remainder of this paper is organized as follows. Section 2 briefly summarizes the SMT architecture and register renaming inefficiencies. Our experimental methodology is described in Section 3. Section 4 describes the OS and compiler support that we use to improve register usage. We discuss related work in Section 5 and offer concluding remarks in Section 6.

2 Simultaneous Multithreading

Our SMT processor model is similar to that used in previous SMT studies: an eight-wide, out-of-order processor with hardware contexts for eight threads. On every cycle four instructions are fetched from each of two threads. The fetch unit favors high throughput threads, selecting the two that have the fewest instructions waiting to be executed. After fetching, instructions are decoded, their registers are renamed, and they are inserted into either the integer or floating point instruction queues. When their operands become available, instructions (from any thread) are issued to the functional units for execution. Finally, instructions are retired in per-thread order.

Most components of an SMT processor are an integral part of any dynamically-scheduled, wide-issue superscalar. Instruction scheduling is an important case in point: instructions are issued after their operands have been calculated or loaded from memory, without regard to thread; the register renaming hardware eliminates inter-thread register name conflicts by mapping thread-specific architectural registers onto the processor's physical registers.

The major additions to a conventional superscalar are the instruction fetch unit mentioned above and several per-thread mechanisms, such as program counters, return stacks, retirement and trap logic, and identifiers in the TLB and branch target buffer. The register file contains register state for all resident threads, and consequently the pipeline contains two additional stages for accessing it (one each for reading and writing). The additional stages cause little degradation in single-thread performance, because we assume a fully bypassed datapath.

2.1 Register Renaming and the Register Deallocation Problem

Register renaming eliminates false (output and anti-) dependences that exist in compiled code. It does so by assigning a different physical register to the same logical register being defined in a later instruction. This allows the later (in code sequence) instruction to proceed without violating any false dependence with an earlier instruction (that reads or writes the same logical register). These false dependences are introduced when the compiler's register allocator assigns an arbitrary number of compiler-generated pseudo-registers to the limited number of architectural registers in the instruction set architecture. Each instruction that defines (writes) a new register value is assigned a unique physical register. Subsequent reads of that logical register will try to read that physical register, unless there has been an intervening write to the logical register (and a new physical register assigned). Thus, a write and a read of a register (in teh processor at the same time) will only share the same physical register if they have a true dependence, and two writes will never share the same physical register.

SMT assumes a register mapping scheme similar to that used in the DEC 21264 [8] and MIPS R10000 [32]. The register renaming hardware is responsible for three primary functions (illustrated in Figure 1): (1) physical register allocation, (2) register operand renaming, and (3) register deallocation. Physical register allocation occurs after instruction decoding. When an instruction defines an architectural register (i.e., the instruction writes a destination register), a mapping is created from the architectural register to an available physical register and is entered into a mapping table. If no physical registers are available, instruction fetching stalls. To rename a source register operand to the physical register number, the renaming hardware locates its architectural-to-physical mapping in the mapping table and aliases it to its physical register number. Register deallocation works in conjunction with instruction retirement. An active list keeps track of all uncommitted instructions in the pipeline in per-thread, program order. As instructions retire, the physical registers that are mapped to architectural registers they *re*define (again, a destination register) are deallocated and become available for reallocation.

Renaming hardware handles physical register allocation and renaming rather effectively, but fails to manage deallocation efficiently. A register is dead and could be deallocated once its last use commits. The hardware, however, cannot identify the last uses of registers, because it has no knowledge of register lifetimes. Consequently, hardware can only safely deallocate a physical register when it commits another instruction that redefines its associated architectural register, as shown in Figure 1.

```
1 ldl   r20,addr1(r22)
2 ldl   r21,addr2(r23)
3 addl  r20,r21,r12
           ...

n ldl   r20,addr4(r29)
```

Figure 1: This example illustrates the inability of the renaming hardware to efficiently deallocate the physical register for r20. (The destination registers are italicized). Instruction 1 defines r20, creating a mapping to a physical register, say P1. Instruction 3 is the last use of r20. However, P1 cannot be freed until r20 is redefined in instruction n. In the meantime, several instructions and potentially a large number of cycles can pass between the last use of P1 (r20) and its deallocation.

2.2 Physical Register Organization and the Register Deallocation Problem

In fine-grained multithreaded architectures like the Tera [1], each hardware context includes a register file for its thread, and a thread only accesses registers from its own context, as shown in Figure 2a.[1] In contrast, in an SMT processor, a single register file holds the register context for all threads and thus is shared among all contexts (Figure 2b). We call this organization *Fully-*

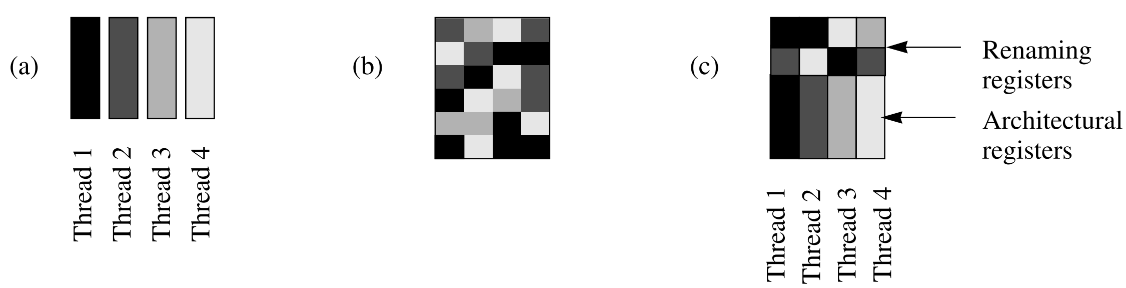

Figure 2: Logical register file configurations: (a) is a Tera-style, partitioned register file; (b) is an SMT register file in which all threads share a common pool of physical registers; (c) is an SMT register file, given current register deallocation schemes: each hardware context has dedicated physical registers for the ISA-defined architectural registers and only the renaming registers are shared across all contexts.

1. Note that we are discussing different *logical* organizations for the register file. How the file is physically structured is a separate issue.

Shared. Since SMT's register file is structured as a single pool of physical registers, SMT's register renaming hardware is essentially an extension of the register mapping scheme to multiple contexts. Threads name architectural registers from their own context, and the renaming hardware maps these thread-private, architectural registers to the pool of thread-independent physical registers. Register renaming thus provides a transparent mechanism for sharing the register pool.

Although an SMT processor is best utilized when all hardware contexts are busy, some contexts may occasionally be idle. To maximize performance, no physical registers should be allocated to idle contexts; instead, all physical registers should be shared by the active threads. However, with existing register deallocation schemes, when a thread terminates, its architectural registers remain allocated in the processor until they are redefined by a new thread executing in the context. Consequently, the Fully-Shared organization behaves more like a partitioned file, as shown in Figure 2c. (We call this organization *Partially-Shared*.) Most ISAs have 32 architectural registers; consequently, thirty-two physical registers must be dedicated to each context in a Partially-Shared scheme. So, for example, on an eight-context SMT with 352 registers, only 96 (352-8*32) physical registers are available for sharing among the active threads.

3 Methodology for the Experiments

We have defined several register file management techniques that were devised to compensate for the hardware's conservative register deallocation, and evaluated them using instruction-level simulation of applications from the SPEC95-FP [23] and SPLASH-2 [30] benchmark suites (Table 1). The SUIF compiler [9] automatically parallelized the SPEC benchmarks into multithreaded C code; the SPLASH-2 programs were explicitly parallelized by the programmer. All programs were compiled with the Multiflow trace-scheduling compiler [15] into DEC Alpha object files. The object files were then linked with our versions of the ANL [2]

	Application	Data Set	Instructions Simulated
SPEC 95 FP	applu	33x33x33 array, 2 iterations	271.9 M
	hydro2d	2 iterations	473.5 M
	mgrid	64x64x64 grid, 1 iteration	3.193 B
	su2cor	16x16x16x16, vector length 4096, 2 iterations	5.356 B
	swim	512x512 grid, 10 iterations	419.1 M
	tomcatv	513x513 array, 5 iterations	189.1 M
	turb3d	64x64x64 array, 1 timestep	1.941 B
SPLASH 2	fft	64K data points	32.0 M
	LU	512x512 matrix	431.2 M
	radix	256K keys, radix 1024, 524288 max key value	5.8 M
	water-nsquared	512 molecules, 3 timesteps	869.9 M
	water-spatial	512 molecules, 3 timesteps	783.5 M

Table 1: Benchmarks used in this study. For the SPEC95 applications, our data sets are the same size as the SPEC reference set, but we have reduced the number of iterations because of the length of simulation time.

and SUIF runtime libraries to create executables.

Our SMT simulator processes unmodified Alpha executables and uses emulation-based, instruction-level simulation to model in detail the processor pipelines, hardware support for out-of-order execution, and the entire memory hierarchy, including the TLBs (128 entries each for instruction and data TLBs), cache behavior, and bank and bus contention. The memory hierarchy in our processor consists of two levels of cache, with sizes, latencies, and bandwidth characteristics, as shown in Table 2. Because register file management is affected by memory latencies,[2] we experimented with two different memory hierarchies. The larger memory configuration represents a probable SMT memory hierarchy for machines in production a few years in the future. The smaller configuration serves two purposes: (1) it models today's memory hierarchies, as well as those of tomorrow's low-cost processors, such as multimedia co-processors, and (2) it provides a more appropriate ratio between data set and cache size, modeling programs with larger data sets or data sets with less data locality than those in our benchmarks [21].

	L1 I-cache	**L1 D-cache**	**L2 cache**
Size (bytes)	128 K / 32 K	128 K / 32 K	16 M / 2 M
Associativity	two-way	two-way	direct- mapped
Line size (bytes)	64	64	64
Banks	4	4	1
Accesses/cycle	2	2	1/2
Cache fill time (cycles)	2	2	4
Latency to next level	10	10	68

Table 2: Configuration and latency parameters of the SMT cache hierarchies used in this study.

We also examined a variety of register file sizes, ranging between 264 and 352, to gauge the sensitivity of the register file management techniques to register size. With more than 352 registers, other processor resources, such as the instruction queues, become performance bottlenecks. At the low end, at least 256 registers are required to hold the architectural registers for all eight contexts,[3] and we provide an additional 8 renaming registers for a total of 264. Smaller register files are attractive for several reasons. First, they have a shorter access time; this advantage could be used either to decrease the cycle time (if register file access is on the critical path) or to eliminate the extra stages we allow for register reading and writing. Second, they take up less area. Register files in current processors occupy a negligible portion (roughly 1%) of the chip area, but a large, multi-ported SMT register file could raise that to around 10%, an area allocation that might not be acceptable [18][29]. Third, smaller register files consume less power.

For branch prediction, we used a McFarling-style hybrid predictor with a 256-entry, 4-way set-associative branch target buffer, and a hybrid predictor (8k entries) that selects between a global history predictor (13 history bits) and a local predictor (a 2k-entry local history table that indexes into a 4k-entry, 2-bit local prediction table) [18].

Because of the length of the simulations, we limited our detailed simulation results to the parallel computation portion of the applications (the norm for simulating parallel applications). For the initialization phases of the applications, we used a fast simulation mode that warmed the

2. Smaller caches increase miss rates, and because more latencies have to be hidden, register pressure increases. The opposite is true for larger caches.

3. in the absence of mechanisms to avoid or detect and recover from deadlock.

caches, and then turned on the detailed simulation mode once the main computation phases were reached.

4 Techniques for improving register file management

Despite its flexible organization, an SMT register file will be underutilized, because renaming hardware fails to deallocate dead registers promptly. In this section, we describe communication mechanisms that allow the operating system and the compiler to assist the renaming hardware with register deallocation, by identifying dead registers that belong to both idle and active contexts.

4.1 Operating system support for dead-register deallocation

As explained in Section 2.2, when an executing thread terminates, the thread's physical registers remain allocated. Consequently, active threads cannot access these registers, causing a fully-shared register file (Fully-Shared) to behave like one in which most of the registers are partitioned by context (Partially-Shared).

After a thread terminates, the operating system decides what to schedule on the newly-available hardware context. There are three options, each of which has a different implication for register deallocation:

1. *Idle contexts*: If there are no new threads to run, the context will be idle. The terminated thread's physical registers could be deallocated, so that they become available to active threads.
2. *Switching to a new thread*: Physical registers for a new thread's architectural registers are normally allocated when it begins execution. A more efficient scheme would free the terminated threads's physical registers, allocating physical registers to the new thread on demand. Unallocated physical registers would then be available to other contexts.
3. *Switching to a swapped-out thread*: Context switch code loads the register state of the new thread. As these load instructions retire, physical registers used by the terminated thread are deallocated.

All three scenarios present an opportunity to deallocate a terminated thread's physical registers early. We propose a privileged, *context deallocation instruction* that triggers physical register deallocation for a thread. The operating system scheduler would execute the context deallocation instruction in the context of the terminated thread. In response, the renaming hardware would free the terminating thread's physical registers when the context deallocation instruction retires.

Three tasks must be performed to handle the context deallocation instruction: creating a new map table, invalidating the context's register mappings, and returning the registers to the free list. When a context deallocation instruction enters the pipeline, the current map table is saved and a new map table with no valid entries is created; the saved map table identifies the physical registers that should be deallocated, while the new table will hold subsequent register mappings. Once the context deallocation instruction retires, the saved map is traversed, and all mapped physical registers are returned to the free list. Finally, all entries in the saved map are invalidated. If the context deallocation instruction is executed on a wrong-path and consequently gets squashed, both the new and saved map tables are thrown away.

Much of the hardware required for these three tasks already exists in out-of-order processors with register mapping to handle branch prediction. When a branch enters the pipeline, a copy of the map table is created; when the branch is resolved, one of the map tables is invalidated, depending on whether the prediction was correct. If instructions must be squashed, the renaming hardware traverses the active list (or some other structure that identifies physical registers) to determine which physical registers should be returned to the free list. Although the context deallocation instruction adds a small amount of logic to existing renaming hardware, it allows the SMT register file to behave as a true Fully-Shared register file, instead of a Partially-Shared register file by deallocating registers more promptly.

Experimental results

To evaluate the performance of the Fully-Shared register organization, we varied the number of active threads and register set sizes, and compared it to the Partially-Shared organization with identical configurations. We modeled an OS scheduler that frees all physical registers for terminated threads, by making all physical registers available when a parallel application began execution.

The results of this comparison are shown in Figure 3. With the Partially-Shared register

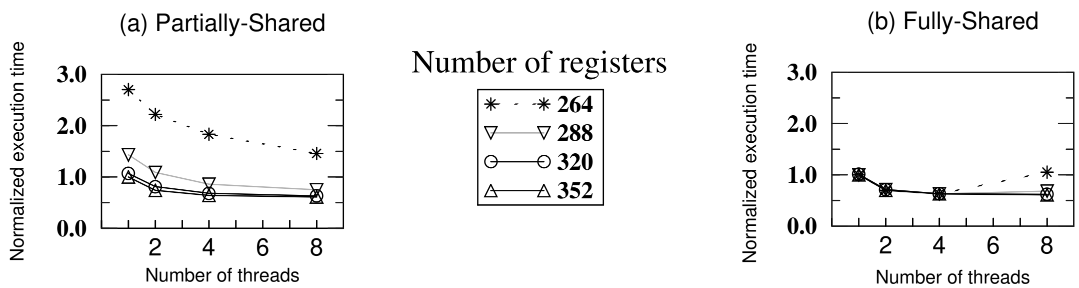

Figure 3: Execution time for Fully-Shared and Partially-Shared with the larger memory hierarchy. Each register file organization is normalized to its 352-register, 1-thread execution. Results with the smaller SMT memory hierarchy had identical results.

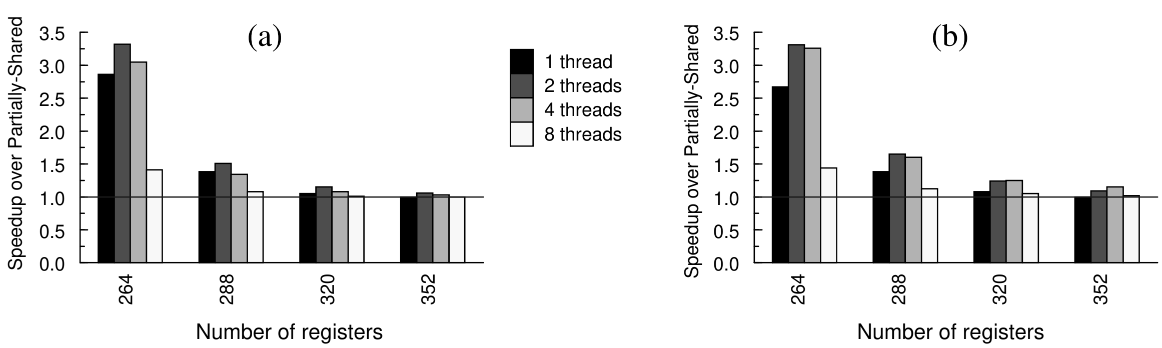

Figure 4: Fully-Shared speedups over Partially-Shared for the larger (a) and smaller (b) memory hierarchies, at different register file sizes.

organization (Figure 3a), only renaming registers are shared among threads. Execution time therefore was higher for smaller register files, as more threads competed for the renaming registers. The Fully-Shared organization, shown in Figure 3b, was less sensitive to both parameters. In fact, the smaller register files had the same performance as larger ones when few threads were executing, because registers were not tied up by idle contexts. Except for the smallest configuration, Fully-Shared performance was stable with varying numbers of threads, because the parallelism provided by additional threads overcame the increased competition for registers; only the 264-register file had a performance sweet spot.

The speedups in Figure 4 show that the Fully-Shared organization equals or surpasses Partially-Shared for all register file sizes and numbers of threads. Fully-Shared provides the greatest benefits when it has more registers to share (several idle contexts) and Partially-Shared has fewer (small register files). For example, with 320 registers and 4 idle contexts (4 threads), Fully-Shared outperformed Partially-Shared by 8%, averaged over all applications. With only 288 or 264 registers, Fully-Shared's advantage grew to 34% and 205%, and with 6 idle contexts (and 320 registers) to 15%. Taking both factors into account (288/264 registers, 6 idle contexts), Fully-Shared outperformed Partially-Shared by 51%/232%. Only when all contexts were active were Fully-Shared and Partially-Shared comparable; in this case, since the architectural state for all threads is resident in both schemes, they have the same number of shared renaming registers available to them.

The Fully-Shared register organization has a slightly larger performance edge with smaller cache hierarchies, because hiding the longer memory latencies requires more in-flight instructions and therefore more registers in use. This suggests that efficient register management is particularly important on memory-intensive workloads or applications with relatively poor data locality.

In summary, the results illustrate that partitioning a multithreaded register file restricts its ability to expose parallelism. Operating system support for deallocating registers in idle contexts, which enables the register file to be fully shared across all threads, both improves performance, and makes it less dependent on the size of the register file and the number of active threads.

4.2 Compiler support for dead-register allocation

As described in section 2.1, hardware register deallocation is inefficient, because the hardware has knowledge only of a register's redefinition, not its last use. Although the compiler can identify the last use of a register, it currently has no means for communicating this information to the hardware.

In this section, we describe and evaluate several mechanisms that allow the compiler to convey register last-use information to the hardware, and show that they improve register utilization on SMT processors with an Fully-Shared register file organization. The proposed mechanisms are either new instructions or fields in existing instructions that direct the renaming hardware to free physical registers.

First, however, we examine three factors that motivate the need for improved register deallocation: (1) how often physical registers are unavailable, (2) how many registers are dead each cycle, and (3) how many cycles pass between a register's last use and its redefinition, which we call the *dead-register distance*. Register unavailability is the percentage of total execution cycles in which the processor runs out of physical registers (causing fetch stalls); it is a measure of the severity of the problem caused by current hardware register-deallocation mechanisms. The average number of dead registers each cycle indicates how many physical registers could be reused, and thus the potential for a compiler-based solution. Dead-register distance measures the average number of cycles between the completion of an instruction that last uses a register and that register's deallocation; it is a rough estimate of the likely performance gain of a solution.

The data in Table 3 indicate that, while the projected SMT design (352 registers in a Fully-Shared register organization) is sufficient for most applications, smaller register files introduce

a) base

```
ldl   r20,addr1(r22)
ldl   r21,addr2(r23)
addl  r20,r21, r24
addl  r21,0x1, r21
      ...
stl   r12,addr3(r21)
ldl   r20,addr4(r29)
```

b) Free Register

```
ldl   r20,addr1(r22)
freg  r22
ldl   r21,addr2(r23)
freg  r23
addl  r20,r21, r24
freg  r20
addl  r21,0x1, r21
      ...
stl   r24,addr3(r21)
freg  r24,r21
ldl   r20,addr4(r259)
freg  r25
```

(c) Free Mask

```
ldl   r20,addr1(r22)
ldl   r21,addr2(r23)
addl  r20,r21, r24
addl  r21,0x1, r21
      ...
stl   r12,addr3(r21)
ldl   r20,addr4(r25)
      ; mask for reg 24
lda   r25,0x1000(r31)
      ; mask for regs 20,21,22,23,25
ldah  r25,0x20f0(r24)
      ;free int regs identified
      ;by mask
fmask r25
```

Figure 5: These code fragments illustrate the register freeing mechanisms: a) is the original code fragment; b) shows the Free Register instructions necessary to free registers r12 to r25, c) is the Free Mask instructions necessary to free the same registers

Number of registers	integer								FP							
	applu	hydro2d	swim	tomcatv	fft	LU	radix	water-n	applu	hydro2d	swim	tomcatv	fft	LU	radix	water-n
Large Cache Hierarchy																
352	3.1	0.1	0.0	0.0	3.8	0.0	0.4	0.0	**27.7**	*6.3*	0.0	**18.0**	0.0	0.0	0.0	0.1
320	3.1	0.7	0.1	0.0	**16.1**	0.1	**36.9**	0.1	**40.2**	**17.7**	0.1	**20.3**	0.0	0.0	0.0	0.6
288	1.9	1.6	0.4	0.0	**21.5**	0.3	**75.9**	0.3	**64.0**	**521**	0.5	**58.4**	0.5	0.0	0.0	**14.1**
264	1.5	2.2	**21.9**	0.0	**58.2**	2.2	**91.1**	0.6	**87.8**	**83.0**	0.6	**90.1**	8.1	0.2	0.0	**73.9**
Small Cache Hierarchy																
352	3.4	1.0	0.4	0.0	8.4	1.3	**12.0**	0.0	**32.3**	**34.4**	5.3	**45.7**	0.5	1.7	0.0	0.1
320	3.5	1.5	1.5	0.0	**16.2**	3.3	**49.1**	0.2	**44.6**	**48.0**	8.6	**60.2**	1.4	3.5	0.0	0.6
288	2.6	2.1	5.3	0.0	**25.2**	**10.5**	**83.5**	0.3	**67.5**	**70.1**	**13.2**	**79.8**	5.3	5.5	0.0	**17.0**
264	2.2	2.5	**44.2**	0.2	**64.9**	**22.7**	**93.4**	0.7	**88.1**	**87.2**	2.7	**94.0**	8.5	6.7	0.0	**74.7**

Table 3: Frequency (percentage of total execution cycles) that no registers were available when executing 8 threads. Bold entries (frequencies over 10%) represent severe stalling due to insufficient registers.

bottlenecks, often severe, on many applications. (Register pressure was particularly high for integer registers in `fft` and `radix`, and for floating-point registers in `applu`, `hydro2d`, `tomcatv`, and `water-n`.) Applications also ran out of registers more frequently with smaller cache hierarchies. A closer examination reveals that in all cases where stalling due to insufficient registers was a problem (bold entries in Table 3), a huge number of registers were dead (shown in Table 4). Table 5 shows that if these dead registers had been freed, they could have been reallocated many instructions/cycles earlier. All this suggests that, if registers were managed more efficiently, performance could be recouped and even a 264-register Fully-Shared register organization might be sufficient.

Five compiler-based solutions

Using dataflow analysis, the compiler can identify the last use of a register value. In this section, we evaluate five alternatives for communicating last-use information to the renaming

Number of registers	integer								FP							
	applu	hydro2d	swim	tomcatv	fft	LU	radix	water-n	applu	hydro2d	swim	tomcatv	fft	LU	radix	water-n
Large Cache Hierarchy																
352	74	69	29	52	135	110	312	168	**120**	153	73	**136**	68	42	48	199
320	74	68	29	53	**134**	110	**311**	168	**120**	**153**	73	**136**	68	42	36	199
288	75	64	29	52	**132**	110	**301**	168	**117**	**152**	73	**135**	67	42	48	**199**
264	77	57	**29**	51	**140**	110	**211**	168	**111**	**150**	73	**133**	71	42	48	**198**
Small Cache Hierarchy																
352	74	62	30	54	132	147	**265**	168	**121**	**158**	77	**141**	68	88	24	199
320	74	61	30	55	**132**	147	**258**	167	**121**	**157**	76	**141**	68	81	21	216
288	74	58	30	55	**131**	**110**	**231**	168	**118**	**155**	**76**	**139**	69	50	44	**199**
264	76	54	**32**	54	**137**	**110**	**173**	168	**113**	**152**	76	**136**	72	50	48	**198**

Table 4: Average number of dead registers per cycle when executing 8 threads. Bold entries are those where no registers were available more than 10% of execution cycles.

Number of registers	applu	hydro2d	swim	tomcatv	fft	LU	radix	water-n	average
int instrs	57.6	59.1	32.3	67.2	**30.7**	**56.9**	**27**	32.7	47.2
int cycles	214.6	155.4	27.8	225.7	**89.9**	**85.6**	**80**	215.4	125.5
FP instrs	**18.4**	**30.9**	11.7	**22.6**	20.4	7.1		**32.7**	18.5
FP cycles	**97.1**	**157.4**	28.4	**120.0**	65.7	22.4		**133.7**	81.8

Table 5: Dead register distance for 264 registers and the smaller cache hierarchy. The data indicate that registers are frequently not deallocated until many cycles after their last use has retired. Figures for other register sizes were similar. Bold entries are those where no registers were available more than 10% of execution cycles.

hardware:

1. **Free Register Bit** communicates last-use information to the hardware via dedicated instruction bits, with the dual benefits of immediately identifying last uses and requiring no instruction overhead. Although it is unlikely to be implemented, because most instruction sets do not have two unused bits, it can serve as an upper bound on performance improvements that can be attained with the compiler's static last-use information. To simulate Free Register Bit, we modified the Multiflow compiler to generate a table, indexed by the PC, that contains flags indicating whether either of an instruction's register operands were last uses. On each simulated instruction, the simulator performs a lookup in this table to determine whether register deallocation should occur when the instruction is retired.
2. **Free Register** is a more realistic implementation of Free Register Bit. Rather than specifying last uses in the instruction itself, it uses a separate instruction to specify one or two registers to be freed. Our compiler generates a Free Register instruction (an unused opcode in the Alpha ISA) immediately after any instruction containing a last register use (if the register is not also redefined by the same instruction). Like Free Register Bit, it frees registers as soon as possible, but with an additional cost in dynamic instruction overhead.
3. **Free Mask** is an instruction that can free up to 32 registers, and is used to deallocate dead registers over a large sequence of code, such as a basic block or a set of basic blocks. For our experiments, we inserted a Free Mask instruction at the end of each Multiflow trace. Rather than identifying dead registers in operand specifiers, the compiler generates a bit mask. In our particular implementation, the Free Mask instruction uses the lower 32-bits of a register as a mask to indicate which registers can be deallocated. The mask is generated and loaded

into the register using a pair of `lda` and `ldah` instructions, each of which has a 16-bit immediate field. (The example in Figure 5 compares Free Register with Free Mask for a code fragment that frees integer registers 20 through 25.) Free Mask sacrifices the promptness of Free Register's deallocation for a reduction in instruction overhead.

4. **Free Opcode** is motivated by our observation that 10 opcodes were responsible for 70% of the dynamic instructions with last use bits set, indicating that most of the benefit of Free Register Bit could be obtained by providing special versions of those opcodes**.** In addition to executing their normal operation, the new instructions also specify that either the first, second, or both operands are last uses. In this paper, we use the 15 opcodes listed in Table 6, obtained by profiling Free Register Bit instruction frequencies on `applu`, `hydro2d` and `tomcatv`.[4] Retrofitting these 15 instructions into an existing ISA should be feasible; for example, all can be added to the DEC Alpha ISA, without negatively impacting instruction decoding.
5. **Free Opcode/Mask** augments Free Opcode by generating a Free Mask instruction at the end of each trace. This hybrid scheme addresses register last uses in instructions that are not covered by our particular choice of instructions for Free Opcode.

Integer		FP	
Opcode	Operand	Opcode	Operand
addl	1	addt	1
subl	1	subt	1
mull	1	mult	1, 2
stl	2	stt	1, both
beq	1	fcmov	1, both
lda	1		
ldl	1		

Table 6: The opcodes used in Free Opcode. Note that for `mult`, `stt`, and `fcmov`, two new versions of each must be added. The versions specify whether the first, second, or both operands are last uses.

Current renaming hardware provides mechanisms for register deallocation (i.e., returning physical registers to the free register list) and can perform many deallocations each cycle. For example, the Alpha 21264 deallocates up to 13 registers each cycle to handle multiple instruction retirement or squashing. All five proposed register deallocation techniques use a similar mechanism. Free Mask is slightly more complex, because it can specify up to 32 registers; in this case deallocation could take multiple cycles if necessary. (In our experiments, however, only 7.2 registers, on average, were freed by each mask.) Deallocating a register before it is redefined means that exception recovery or branch misprediction recovery may find some registers in an undefined state; however, this will only be the case for registers that are guaranteed not to be read again. This optimization is safe.

The five register deallocation schemes are compared in Figure 6, which charts their speedup

4. We experimented with between 10 and 22 Free Opcode instructions. The additional opcodes after the top 15 tended to occur frequently in only one or two applications, and using them brought limited additional benefits (the exceptions were `swim` and `radix`).

versus no explicit register deallocation. The Free Register Bit bars show that register deallocation

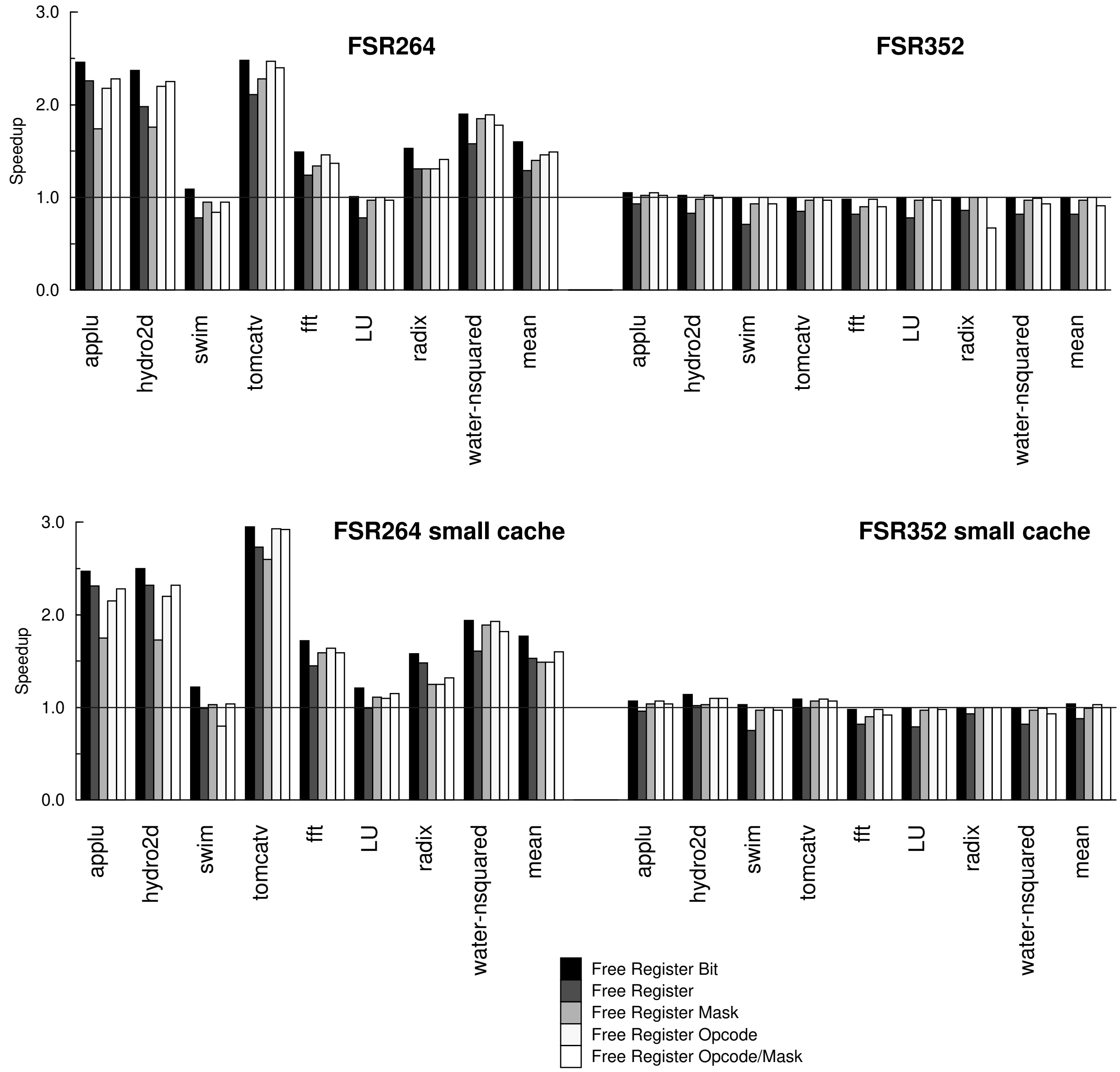

Figure 6: A comparison of register deallocation alternatives. Each bar is the speedup over no deallocation with 8 threads.

can (potentially) improve performance significantly for small register files (77% on average, but ranging as high as 195%). The Free Register Bit results highlight the most attractive outcome of register deallocation: by improving register utilization, an SMT processor with small register files can achieve large register file performance, as shown in Figure 7. The significance of this becomes apparent in the context of conventional register file design. Single-threaded, out-of-order processors often double their registers to support greater degrees of parallelism (e.g., the R10000 has 64 physical registers, the 21264 has 80). With multiple register contexts, an SMT processor need not double its architectural registers if they are effectively shared. Our results show that an 8-context SMT with an Fully-Shared register file (i.e., support for deallocating registers in idle contexts) needs only 96 additional registers to alleviate physical register

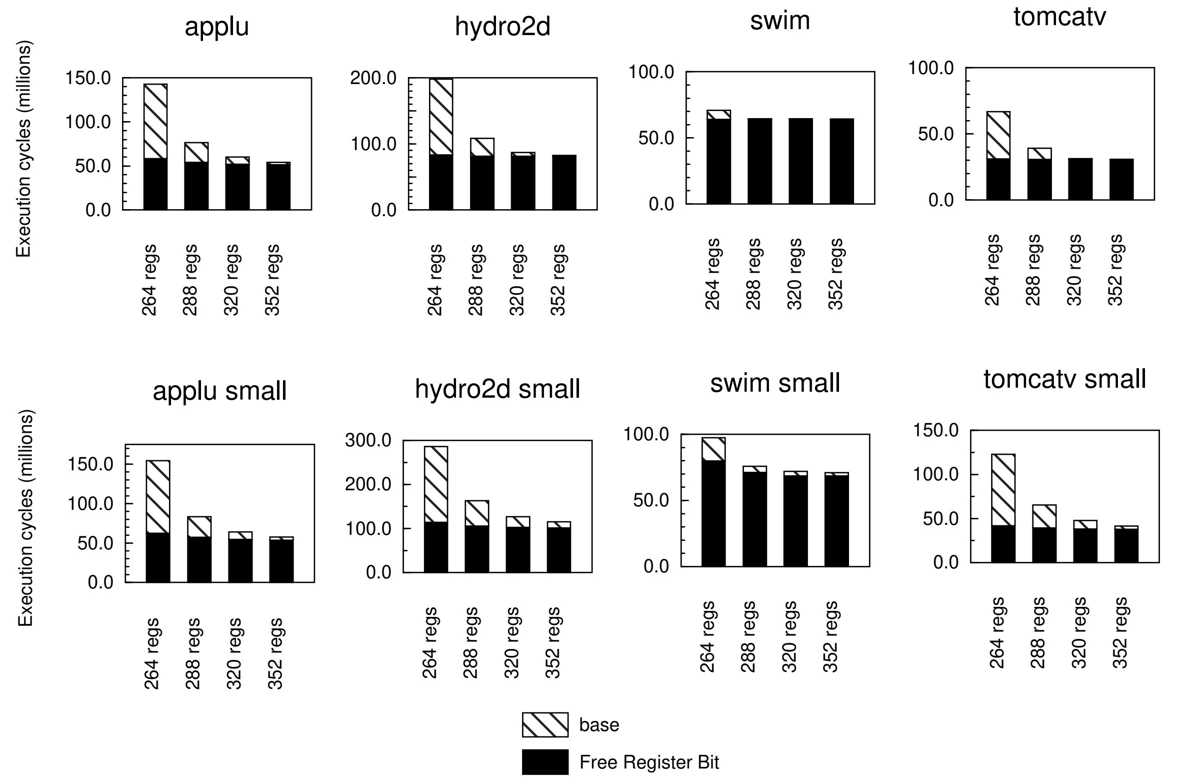

Figure 7: Comparison of execution time for the Fully-Shared register file organization with and without Free Register Bit for a range of file sizes, on both the larger and smaller cache sizes. The height of the solid black bar represents the execution time when Free Register Bit is used. The total height of the bar corresponds to the execution time when no deallocation is performed. The relatively flat height of the black bars indicates that with Free Register Bit, smaller register files can achieve the performance of larger register files.

pressure, lowering the renaming register cost to 27% of the ISA-defined registers. Compiler-directed register deallocation for active contexts drops the overhead even further, to only 8 registers or 3% of the architectural register state.

The Free Register and Free Mask results highlight the trade-off between these two alternative schemes. Free Register is more effective at reducing the number of dead registers, because it deallocates them more promptly, at their last uses. When registers are a severe bottleneck, as in `applu`, `hydro2d`, `tomcatv`, and `radix` with small register files, Free Register outperforms Free Register Mask. Free Register Mask, on the other hand, incurs less instruction overhead; therefore it is preferable with larger register files and applications with low register usage.

Free Opcode and its variant, Free Opcode/Mask,[5] are the schemes of choice. They strike a balance between Free Register and Free Mask by promptly deallocating registers, while avoiding instruction overhead. When registers were at a premium, Free Opcode(/Mask) achieved or exceeded the performance of Free Register; with the larger register file and for applications with low register usage, Free Mask performance was

5. We profiled a very small sample of programs to determine the best selection of opcodes for Free Opcode, and used Free Opcode/Mask to provide more flexibility in opcode choice. The speedups of the two schemes are very close, and which has the performance edge varies across the applications for 264 registers. Looking at a different or larger set of programs to determine the hot opcodes might tip the performance balance for these cases. (For example, by adding 6 single-precision floating point Free Opcodes to the single-precision swim, Free Opcode exceeded both Free Register and Free Mask.) Therefore we discuss the results for Free Opcode and Free Opcode/Mask together.

attained or surpassed.

For most programs (all register set sizes and both cache hierarchies) Free Opcode(/Mask) met or came close to the optimal performance of Free Register Bit. (For example, it was within 4% on average for 264 registers, and 10% for 352, on the small cache hierarchy.) With further tuning of opcode selection and the use of other hybrid schemes (perhaps judiciously combining Free Opcode, Free Mask, and Free Register), we expect that the gap between it and Free Register Bit will be narrowed even further, and that we will achieve the upper bound of compiler-directed register deallocation performance. Although the addition of the Free Mask to Free Opcode is clearly desireable, there were certain results that indicated it needn't always be used, as Free Opcode actually outpeformed Free Opcode/Mask due to the overhead of the mask instructions.

In summary, by providing the hardware with explicit information about register lifetimes, compiler-directed register deallocation can significantly improve performance on small SMT register files, so that they become a viable alternative even with renaming register-intensive applications. Although particularly well-suited for SMT, register deallocation should benefit any out-of-order processor with explicit register renaming.

5 Related work

Several researchers have investigated register file issues similar to those discussed in this paper. Large register files are a concern for both multithreaded architectures and processors with register windows. Waldspurger and Weihl [27] proposed compiler and runtime support for managing multiple register sets in a register file. The compiler tries to identify an optimal number of registers for each thread, and generates code using that number. The runtime system then tries to dynamically pack the register sets from all active threads into the register file. Nuth and Dally's [19] named state register file caches register values by dynamically mapping active registers to a small, fast set of registers, while backing the full register name space in memory.

To reduce the required chip area in processors with register windows, Sun designed 3-D register files [24]. Because only one register window can be active at any time, the density of the register file can be increased by overlaying multiple register cells so that they share wires.

Several papers have investigated register lifetimes and other register issues. Farkas, et al. [6] compared the register file requirements for precise and imprecise interrupts and their effects on the number of registers needed to support parallelism in an out-of-order processor. They also characterized the lifetime of register values, by identifying the number of live register values present in various stages of the renaming process.

Martin, et al. [17] communicate dead-register information through a mechanism very similar to our free-register instruction (the E-DVI instruction). The focus of that paper is to use the information to dynamically eliminate unnecessary load and store instructions, particularly around procedure calls, and achieve 0.4% to 4.8% improvements as a result. They do note its advantages in physical register-size reduction, even for the single-thread architectures they consider. We consider more effective mechanisms for communicating live ranges, and examine both single-thread and multithreaded scenarios.

Franklin and Sohi [7] and Lozano and Gao [16] found that logical-register values have short lifetimes, and often do not need to be committed to the register file. Both proposed compiler support to identify last uses and architectural mechanisms to allow the hardware to ignore writes to reduce register file traffic and the number of write ports, but neither applied these concepts to register deallocation. Pleszkun and Sohi [20] proposed a mechanism for exposing the reorder buffer to the compiler, so that it could generate better schedules and provide speculative execution. Sprangle and Patt [22] proposed a statically-defined tag ISA that exposes register renaming to the compiler and relies on basic blocks as the atomic units of work. Part of the register file is used for storing basic block effects, and the rest handles values that are live across basic block boundaries.

Wallace and Bagherzadeh [29] move renaming-register allocation late in the pipeline, at register-write, to reduce pressure on the register file. Janssen and Corporaal [11], Capitanio, et al. [3], Llosa, et al. [13], Multiflow [4], and Kiyohara, et al. [12] also investigated techniques for handling large register files, including partitioning, limited connectivity, replication, and the use of new opcodes to address an extended register file.

6 Conclusions

Simultaneous multithreading has the potential to significantly increase processor utilization on wide-issue, out-of-order processors, by permitting multiple threads to issue instructions to the processor's functional units within a single cycle. As a consequence, SMT requires a large register file to support the multiple thread contexts. This raises a difficult design tradeoff, because large register files can consume die area and impact performance.

This paper has introduced new software-directed techniques that increase utilization of the registers in an SMT. Fundamental to these techniques is the global sharing of registers among threads, both for architectural register and renaming register needs. By introducing new instructions or additional fields in the ISA, we allow the operating system and compiler to signal physical register deallocation to the processor, thereby greatly decreasing register waste. The result is more effective register use, permitting either a reduction in register file size or an increase in performance for a given file size.

We have introduced explicit software-directed deallocation in two situations. First, when a context becomes idle, the operating system can indicate that the idle context's physical registers can be deallocated. This permits those registers to serve the renaming needs of other executing threads. Our results show that such notification can significantly boost performance for the remaining threads, e.g., a register file with 264 registers demonstrates performance equivalent to a 352-register file when only 4 threads are executing. Second, by allowing the compiler to signal the last use of a register, the processor need not wait for a redefinition of that register in order to reuse it. We proposed several mechanisms for signalling last register use, and showed that on small register files, average speedups of 1.6 can be obtained by using the most efficient of these mechanisms. While our results are shown in the context of an SMT processor, these mechanisms would be appropriate for any processor using register renaming for out-of-order instruction issue.

References

[1] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera computer system. In *International Conference on Supercomputing*, pages 1–6, June 1990.

[2] J. Boyle, R. Butler, T. Diaz, B. Glickfeld, E. Lusk, R. Overbeek, J. Patterson, and R. Stevens. *Portable Programs for Parallel Processors*. Holt, Rinehart, and Winston, Inc., 1987.

[3] A. Capitanio, N. Dutt, and A. Nicolau. Partitioned register files for VLIWs: A preliminary analysis of tradeoffs. In *25th International Symposium on Microarchitecture*, pages 292–300, December 1992.

[4] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. Papworth, and P. Rodman. A VLIW architecture for a trace scheduling compiler. *IEEE Transactions on Computers*, C-37(8):967–979, August 1988.

[5] S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm, and D. M. Tullsen. Simultaneous multithreading: A platform for next-generation processors. In *IEEE Micro*, pages 12–19, October 1997.

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

[7] M. Franklin and G. Sohi. Register traffic analysis for streamlining inter-operation communication in fine-grained parallel processors. In *25th International Symposium on Microarchitecture*, pages 236–245, December 1992.

[8] L. Gwennap. Digital 21264 sets new standard. *Microprocessor Report*, pages 11–16, October 1996.

[9] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, 29(12):84–89, December 1996.

[10] H. Hirata, K. Kimura, S. Nagamine, Y. Mochizuki, A. Nishimura, Y. Nakase, and T. Nishizawa. An elementary processor architecture with simultaneous instruction issuing from multiple threads. In *19th Annual International Symposium on Computer Architecture*, pages 136–145, May 1992.

[11] J. Janssen and H. Corporaal. Partitioned register files for TTAs. In *28th International Symposium on Microarchitecture*, pages 303–312, December 1995.

[12] T. Kiyohara, S. Mahlke, W. Chen, R. Bringmann, R. Hank, S. Anik, and W. Hwu. Register connection: A new approach to adding registers into instruction set architectures. In *20th Annual International Symposium on Computer Architecture*, pages 247–256, May 1993.

[13] J. Llosa, M. Valero, and E. Ayguade. Non-consistent dual register files to reduce register pressure. In *First Annual International Symposium on High-Performance Computer Architecture*, pages 22–31, January 1995.

[14] J. L. Lo, S. J. Eggers, J. S. Emer, H. M. Levy, R. L. Stamm, and D. M. Tullsen. Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading. *ACM Transactions on Computer and Systems*, 15(3):322–354, August 1997.

[15] P. G. Lowney, S. M. Freudenberger, T. J. Karzes, W. Lichtenstein, R. P. Nix, J. S. O'Donnell, and J. C. Ruttenberg. The Multiflow trace scheduling compiler. *Journal of Supercomputing*, 7(1/2):51–142, May 1993.

[16] C. L. Lozano and G. Gao. Exploiting short-lived variables in superscalar processors. In *28th International Symposium on Microarchitecture*, pages 292–302, December 1995.

[17] M. Martin, A. Roth, and C. Fischer. Exploiting dead value information. In *30th International Symposium on Microarchitecture*, pages 125–135, December 1997.

[18] S. McFarling. Combining branch predictors. Technical Report TN-36, DEC-WRL, June 1993.

[19] P. R. Nuth and W. J. Dally. The named-state register file: Implementation and performance. In *First Annual International Symposium on High-Performance Computer Architecture*, pages 4–13, January 1995.

[20] A. R. Pleszkun and G. S. Sohi. The performance potential of multiple functional unit processors. In *15th Annual International Symposium on Computer Architecture*, pages 37–44, May 1988.

[21] J. P. Singh, J. L. Hennessy, and A. Gupta. Scaling parallel programs for multiprocessors: Methodology and examples. *IEEE Computer*, 27(7):42–50, July 1993.

[22] E. Sprangle and Y. Patt. Facilitating superscalar processing via a combined static/dynamic register renaming scheme. In *27th International Symposium on Microarchitecture*, pages 143–147, December 1994.

[23] The Standard Performance Evaluation Council. *SPEC CPU '95 Technical Manual*. August 1995.

[24] M. Tremblay, B. Joy, and K. Shin. A three dimensional register file for superscalar processors. In *28th Hawaii International Conference on System Sciences*, pages 191–201, January 1995.

[25] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *23rd Annual International Symposium on Computer Architecture*, pages 191–202, May 1996.

[26] D. M. Tullsen, J. L. Lo, S. J. Eggers, and H. M. Levy. Supporting fine-grained synchronization on a simultaneous multithreading processor. In *Fifth Annual International Symposium on High-Performance Computer Architecture*, January 1999.

[27] C. A. Waldspurger and W. E. Weihl. Register relocation: Flexible contexts for multithreading. In *20th Annu-*

al International Symposium on Computer Architecture, pages 120–129, May 1993.

[28] S. Wallace, B. Calder, and D. M. Tullsen. Threaded multiple path execution. In *25th Annual International Symposium on Computer Architecture*, pages 238–249, June 1998.

[29] S. Wallace and N. Bagherzadeh. A Scalable Register File Architecture for Dynamically Scheduled Processors. In *Parallel Architectures and Compilation Techniques*, pages 179-184, October 1996.

[30] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *22nd Annual International Symposium on Computer Architecture*, pages 24–36, June 1995.

[31] W. Yamamoto, M. J. Serrano, A. R. Talcott, R. C. Wood, and M. Nemirovsky. Performance estimation of multistreamed, superscalar processors. In *27th Hawaii International Conference on System Sciences*, pages I:195–204, January 1994.

[32] K. C. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, pages 28–40, April 1996.