The Named-State Register File: Implementation and Performance

Peter R. Nuth and William J. Dally
MIT Artificial Intelligence Laboratory
545 Technology Square, NE43-617
Cambridge, MA 02139
Tel: (617) 253-8572
Fax: (617) 253-5060
Email: {nuth,billd}@ai.mit.edu

Abstract

Context switches are slow in conventional processors because the entire processor state must be saved and restored, even if much of the state is not used before the next context switch. This paper introduces the *Named-State Register File*, a fine-grain associative register file. The NSF uses hardware and software techniques to efficiently manage registers among sequential or parallel procedure activations. The NSF holds more live data per register than conventional register files, and requires much less spill and reload traffic to switch between concurrent contexts. The NSF speeds execution of some sequential and parallel programs by 9% to 17% over alternative register file organizations. The NSF has access time comparable to a conventional register file and only adds 5% to the area of a typical processor chip.

Keywords: multithreaded, processor, register, context switch.

NOTE: This is a draft copy of a paper that has been submitted for publication.
Please do not reference or redistribute without the consent of the authors.

The Named-State Register File: Implementation and Performance

1. Introduction

Most sequential and parallel applications execute as a data-dependant chain of procedure activations. Each activation requires a small amount of run-time state for local variables. While some of this local state may reside in memory, the rest occupies the processor's register file. The register file is a critical resource in modern processors [10]. Operating on data in registers rather than memory speeds access to that data, and allows one instruction to access several operands [26,7].

There have been many proposals for hardware and software mechanisms to manage the register file and to efficiently switch between activations [4,34]. These techniques work well when the activation sequence is known, but behave poorly if the order of activations is unpredictable [14]. Dynamic parallel programs [6,12,30], in which a processor may switch between many concurrent activations, or *threads*, run particularly inefficiently on conventional processors. To switch between threads, a conventional processor must spill a thread's *context* from registers to memory, then load a new context. This may take hundreds of cycles [11]. If context switches are frequent and unpredictable, a large fraction of execution time is spent saving and restoring registers.

1.1 The Named-State Register File

This paper introduces the *Named-State Register File*, a register file organization that permits fast switching among many concurrent activations while making efficient use of register space. It does this without sacrificing sequential thread performance, and can often run sequential programs more efficiently than conventional register files.

The NSF has a slightly longer access time than conventional register files, but not enough to affect a processor's cycle time. While the NSF requires more chip area per bit than conventional register files, that storage is used more effectively, leading to significant performance improvements over alternative register files.

This paper describes the Named-State Register File, evaluates the cost of its implementation and its benefits for context switching. It presents the results of architectural simulations of large sequential and parallel applications to evaluate the effect of the NSF on register usage, register reload traffic, and execution time.

1.2 Advantages of the NSF

The Named-State Register File uses a combination of hardware and software to dynamically map a large register name space into a small, fast register file. In effect, it acts as a cache for the register name space. It has several advantages for running sequential and parallel applications:

- The NSF has low access latency, and high bandwidth.
- Instructions refer to registers in the NSF using short compiled register offsets, and may access several register operands in a single instruction.
- The NSF can use traditional compiler analysis [5] to allocate registers in sequential code, and to manage registers across code blocks [30, 34].
- The NSF expands the size of the register name space, without increasing the size of the register file or of the instruction format.
- The register name space is separate from the virtual address space, and mapping between the two is under program control.
- The NSF uses an associative decoder, small register lines, and hardware support for register spill and reload to dynamically manage registers from many concurrent contexts.
- The NSF uses registers more effectively than conventional files, and requires less register traffic to support a large number of concurrent active contexts.

2. Motivation

Compile-time or link-time inter-procedural register allocation works well for many sequential programs [34]. But it is less effective for programming models that support recursion, dynamic linking, or run-time dispatching [30,12]. For these programs, register file hardware can often allocate the registers more efficiently across procedure calls [9,4].

Inter-procedural register allocation is especially difficult for parallel programming models that dynamically spawn parallel procedure invocations, or *threads*[1]. Those programs may run across hundreds of processors of a parallel computer. Since parallel threads are spawned dynamically in this model, and synchronization between threads may be data dependent, the order in which threads are executed on a single processor cannot be determined in advance. A compiler may be able to schedule the execution order of a local group of threads [14], but in general will not be able to determine a total ordering of threads across all processors.

1. As opposed to models of parallelism in which the number of tasks is fixed at compile time [8].

In addition, a processor of parallel computer may often switch between concurrent threads in order to mask communication and synchronization latencies. Most parallel applications frequently pass data among processors. Fine grain programs send messages every 75 to 100 instructions [12], each of which may require a round trip latency of more than 100 instruction cycles [3]. Threads also often synchronize with other threads to exchange data. A thread may only run 20 to 80 instructions [6] between synchronization points, and may wait an unbounded amount of time at any synchronization point [20]. Stalling for every remote access or synchronization point would waste a large fraction of the processor's performance.

An alternative to idling a processor on communication and synchronization points is to quickly switch to another thread and continue running. (See Figure 1). The less time spent context switching, the greater a processor's utilization [2].

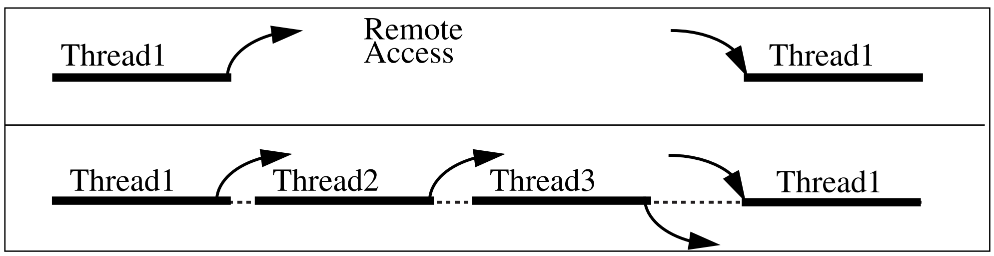

FIGURE 1. Advantage of fast context switching.
A processor idling on remote accesses or synchronization points (top), compared with rapid context switching between threads (bottom).

3. Multithreaded Processors

Multithreaded processors [27,32,8] reduce context switch time by holding the state of several threads in the processor's high speed memory. Typically, a multithreaded processor divides its local registers among several concurrent threads. This allows the processor to quickly switch among those threads, although switching outside of that small set is no faster than on a conventional processor.

Multithreaded processors may interleave successive instructions from different threads on a cycle-by-cycle basis [27,16,24,19], or as blocks of instructions [8,3]. Although the techniques introduced in this paper are applicable to both forms of multithreading, this discussion will concentrate on block multithreading.

3.1 Segmented Register Files

Figure 2 describes a typical implementation of a multithreaded processor [27, 16,3,28]. This processor partitions a large register set into a few register *frames*, each of which

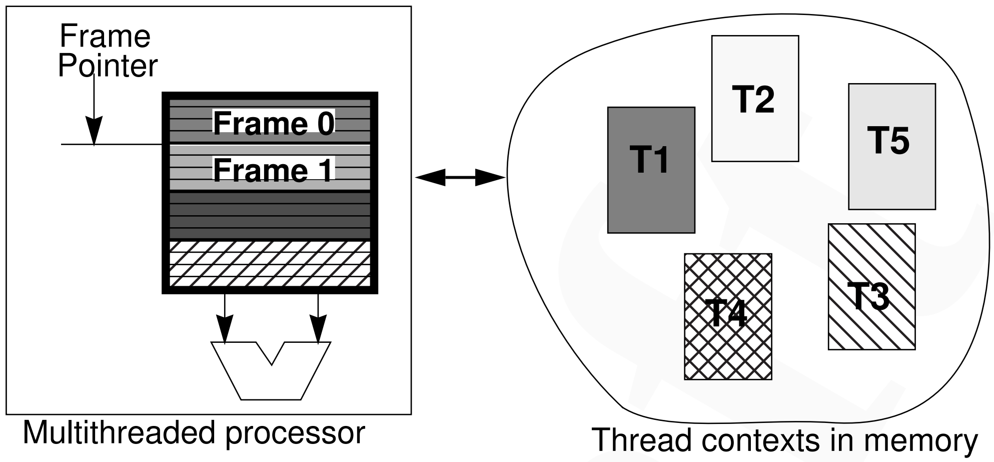

FIGURE 2. A multithreaded processor using a segmented register file.
The register file is segmented into equal sized frames, one for each concurrent thread. The processor spills and restores thread contexts from register frames into main memory.

holds the registers of a different thread. A frame pointer selects the current active frame. Instructions from the current thread refer to registers using short offsets from the frame pointer.

Switching between the resident threads is very fast, since it only requires setting the frame pointer. However, in a parallel computer with long communication and synchronization delays, often none of these resident threads will be able to make progress. To switch to a non-resident thread, the processor must spill the contents of a register frame out to memory, and load the registers of a new thread in its place.

This static partitioning of the register file is also an inefficient use of processor resources. Some threads may not use all the registers in a frame. Also, if the processor switches contexts frequently, it may not access all the registers in a context before it must spill them out to memory again. In both cases, the processor wastes memory bandwidth loading and storing unused registers.

Dividing the register file into large, fixed sized frames also wastes space in the register file. At any time, some fraction of each register frame holds live variables, and the remainder is not used. This wastes a large fraction of the register file, which is the most precious memory in the machine. A more efficient scheme would hold only live data in the register file.

The problem with a segmented register file organization is that it binds a set of variable names (for a thread) to an entire block of registers (a frame). A more efficient organization would bind variable names to registers at a finer granularity.

4. The Named-State Register File

The Named-State Register File (NSF) is an alternative register file organization. It is not divided into large frames for each thread. Instead, the NSF is a *fully-associative* structure with very small lines. A thread's registers may be distributed anywhere in the register array, not necessarily in one continuous block. An active thread may have any number of its registers resident in the array.

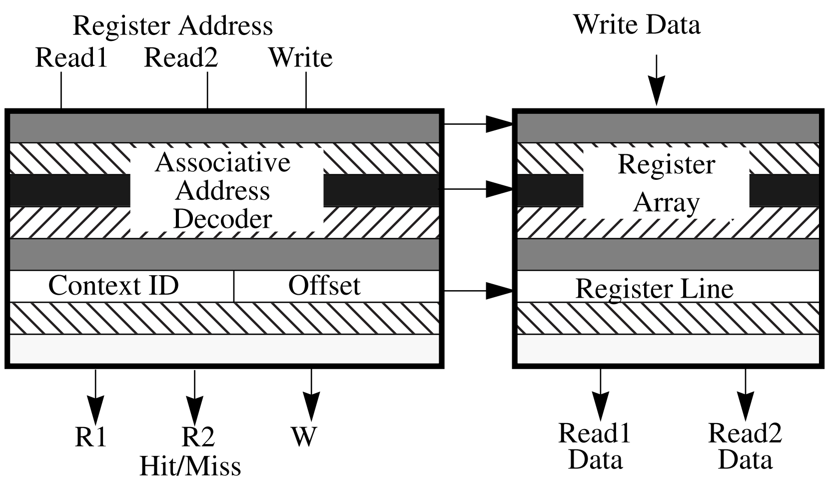

FIGURE 3. Structure of the Named-State Register File.
The NSF holds registers from a number of resident contexts. The processor spills and restores individual registers to main memory as needed by the active threads.

The NSF uses hardware and software mechanisms to dynamically allocate the register set among the active threads. The NSF does not explicitly spill and reload contexts after a thread switch. Registers are loaded on demand by the new thread. Registers are only spilled out of the NSF as needed to clear space in the register file.

The NSF allows a processor to interleave many more threads than segmented files, since there can be as many resident threads as register lines. The NSF keeps more active data resident than segmented files, since it is not coarsely fragmented among threads. It spills and reloads far fewer registers than segmented files, since it only loads registers as they are needed.

4.1 Structure of the NSF

Figure 3 outlines the structure of the Named-State Register File. The NSF is composed of two components: the register array itself, and a fully-associative address decoder. The NSF is multi-ported, as are conventional register files, to allow simultaneous read and write operations. Figure 3 shows a three ported register file.

A conventional register file is a non-associative, indexed memory, in which a register address is a line number in the register array. Once a variable has been written to a location in the register file, it does not move until the context is swapped out.

The Named-State Register File, on the other hand, is fully-associative, since a register address may be assigned to any line of the register file. During the lifetime of a context, a register variable may occupy a number of different locations within the register array. The unit of associativity of the NSF is a single *line*. Each line is allocated or deallocated as a unit from the NSF. Depending on the design, an NSF line may consist of a single register, or a small set of consecutive registers. Typical register organizations may have line sizes between one and four registers wide.

The NSF uses an associative address decoder to achieve this flexibility. Each line of the address decoder contains a *content addressable memory* (CAM) wide enough to hold a register address. The NSF binds a register name to a line in the register file by programming that line of the address decoder. Subsequent register reads and writes compare an operand address against the address programmed into each line of the decoder. [1] describes the structure and implementation of the Named-State Register file in more detail.

4.2 Operation of the NSF

As in any general register architecture, instructions refer to registers in the NSF using a short register *offset*. This identifies the register within the current procedure or thread activation. However, instead of using a frame pointer to identify the current context, the processor tags each context with a *Context ID*. This is a short integer that uniquely identifies the current context from among those resident in the register file.

A register address in the NSF is the concatenation of its Context ID and offset. The current instruction specifies the register offset, and a processor status word supplies the current CID. Each Context ID defines a separate set of register names. The width of the offset field determines the size of the register set (typically 32 registers). The NSF avoids any restrictions on how Context IDs are used by different programming models. [1] describes some issues related to the management of Context IDs.

The first write to a new register also writes its address into the associative decoder, allocating that register in the array. The NSF can explicitly deallocate a single register after it is no longer needed, or can deallocate all registers associated with a particular context. This frees the lines to be used by a new set of register variables.

The NSF holds a fixed number of registers. After a register write operation has allocated the last available register line in the register file, the NSF must spill a line out of the register file and into memory. The NSF could pick this victim to spill based on a number of different strategies. This study simulates a *least recently used* (LRU) strategy.

If an instruction attempts to read a register that has already been spilled out of the NSF, that operation will *miss* on that register. The NSF signals a miss to the processor pipeline, stalling the instruction that issued the read. The register file then reloads that register from memory. Depending on the organization of the NSF, it may reload only the register that missed, or the entire line containing that register. Although this strategy may cause several instructions to stall during the lifetime of a context, it ensures that the NSF never loads registers that are not needed. Better utilization of the NSF register file more than compensates for the additional misses on register fetches.

Writes may also miss in the register file. Depending on the NSF design, a write miss may cause a line to be reloaded into the file (fetch on write), or may simply allocate a line for that register in the file (write-allocate).

Context switching is very fast with the NSF, since no registers must be saved or restored. The NSF does not explicitly spill a context out of the register file after a switch. The processor simply issues instructions from the new context. These instructions may miss in the register file and reload registers as needed.

Although register allocation and deallocation in the NSF use explicit addressing modes, spilling and reloading are implicit. The instruction stream creates and destroys contexts and local variables. The NSF hardware manages register spilling and reloading in response to run-time events. In particular, there are no instructions to spill a register or a context from the register file.

4.3 NSF and memory hierarchy

Figure 4 illustrates how the Named-State Register File fits into a processor's memory hierarchy. In most modern computers, programs refer to data stored in memory using *virtual memory* addresses. A data or instruction cache transparently captures frequently used data from this virtual address space. The cache is not the primary home for this data, but must ensure that data is always saved out to memory to avoid inconsistency. In a similar manner, physical memory stores portions of that virtual address space under control of the operating system.

A conventional register file defines a *register name space* separate from that of main memory. Registers are addressed by register number, not using a virtual memory address. Since the register set is separate from the rest of memory, a compiler may efficiently manage this space [5]. A program typically spills and reloads variables from the register set into stack or heap frames in main memory. A compiler may use local knowledge about variable usage to optimize this movement [30]. Register variables can be allocated and destroyed as needed by the program.

In the Named-State Register File, the name space now consists of a <Context ID: Offset> pair. The Context IDs significantly increase the size of the register name space. Since the

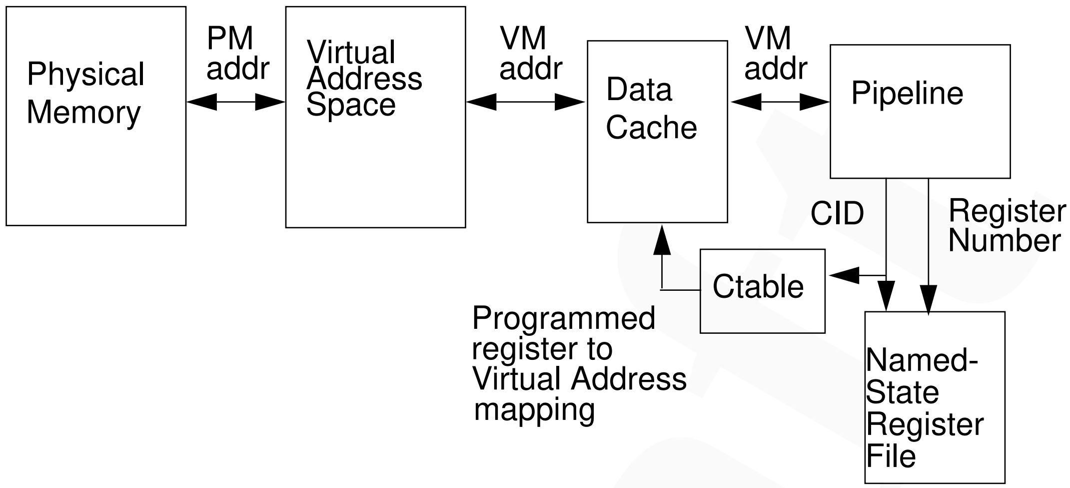

FIGURE 4. The Named-State Register File and memory hierarchy.
The NSF addresses registers using a <Context ID: Offset> pair. This defines a large register name space for the NSF. The Ctable is a short indexed table to translate Context IDs to virtual addresses.

NSF is an associative structure, it can hold any registers from this large address space in a small, efficient memory.

The NSF can use the same compiler techniques as conventional register files to effectively manage the register name space. A program may explicitly copy registers to and from the virtual memory space (the backing store) as do conventional register files. But the NSF provides additional hardware to help manage the register file under very dynamic programming models, where compiler management is less effective [14].

Figure 4 shows how the NSF hardware maps registers into the virtual memory space to support spills and reloads. The block labelled *Ctable* is a short table indexed by Context ID that returns the virtual address of a context. This allows the NSF to spill registers directly into the data cache. A user program or thread scheduler may use any strategy for mapping register contexts to structures in memory, simply by writing the translation into the Ctable.

The NSF provides a mechanism to handle multiple activations, but does not enforce any particular strategy. Since Context IDs are neither virtual addresses, nor global thread identifiers, they can be assigned to contexts in any way needed by the programming model. For instance, a compiler for a sequential program may allocate a new CID for each procedure invocation. A parallel language might allocate a new context for every thread activation. A programming model may even allocate two Context IDs to a single procedure or thread activation. [1] discusses issues in managing the register name space and Context IDs.

5. Related Work

Keppel [17] and Hidaka [11] propose running multiple concurrent threads in the register windows of a Sparc [31] processor by modifying window trap handlers. The Sparcle chip [3] adds trap hardware and tuned trap handlers to a Sparc chip. Arvind [23] and Agarwal [29] propose register file organizations that either pre-load contexts before a task switch, or spill contexts in the background. Each of these approaches uses a segmented register file as described in Section 3.1, and has the same disadvantages. The large, fixed partitioning leads to poor utilization of the register file, and spilling frames on context switches generates high register spill and reload traffic.

Waldspurger [33] proposes modifications to a processor pipeline, and compiler and runtime software to share a register file among different threads. A compiler must determine the optimum frame size for a thread, and runtime software attempts to dynamically pack these different frame sizes into the register file. In contrast, the NSF allows a more dynamic binding of registers to contexts, so that an active thread can use a larger proportion of the register file.

The C-machine [4] is a register-less architecture that stores the top of stack in a multi-ported stack buffer on chip. Russell and Shaw [25] propose a stack as a register set, using pointers to index into the buffer. These structures might improve performance on sequential code, but are very slow to context switch because of the implicit FIFO ordering.

Huguet and Lang [13], Miller and Quammen [21], and Kiyohara [18] have each proposed register file designs that use indirection to add additional register blocks to a basic register set. These designs are expensive to implement and may slow down sequential execution.

6. Implementation

This section describes an implementation of the Named-State Register File and compares its access time and chip area to conventional register files. Figure 5 shows a photograph of a prototype NSF chip, built in $2\mu m$ CMOS technology. The chip was built as a proof of concept for the NSF logic, and to validate area and speed estimates of different NSF organizations. [1] describes the NSF implementation in more detail.

6.1 Performance comparison

Figure 6 shows the results of Spice [22] simulations of the Named-State Register File and conventional register files. The NSF required slightly more time to decode addresses, since it had to compare more bits than a two-level decoder for a conventional register file. It also took more time to combine Context ID and Offset address match signals and drive a word line into the register array.

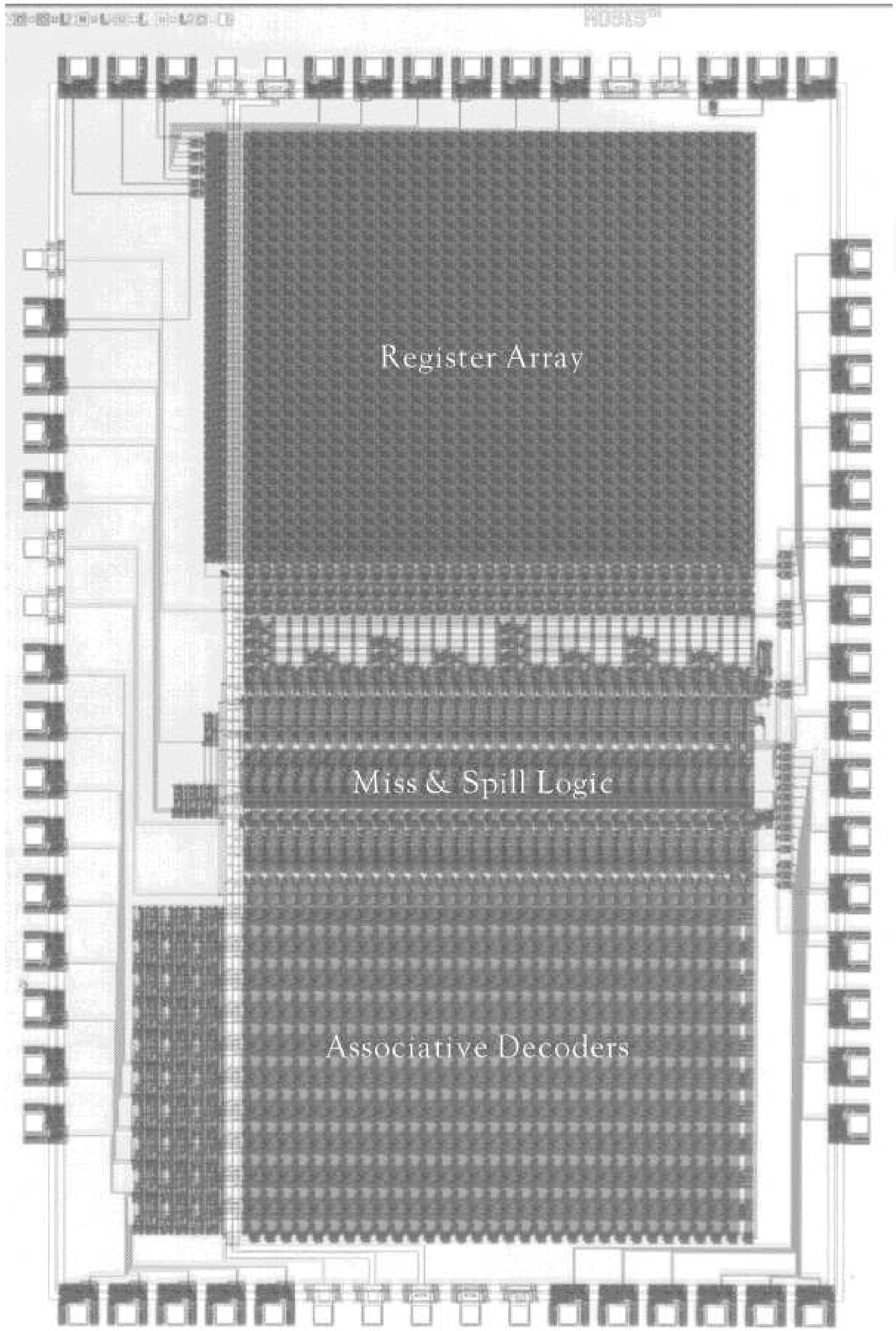

FIGURE 5. A prototype Named-State Register File.
This prototype chip includes a 32 bit by 32 line register array, a 10 bit wide fully-associative decoder, and logic to handle misses, spills and reloads. The register file has two read ports and a single write port.

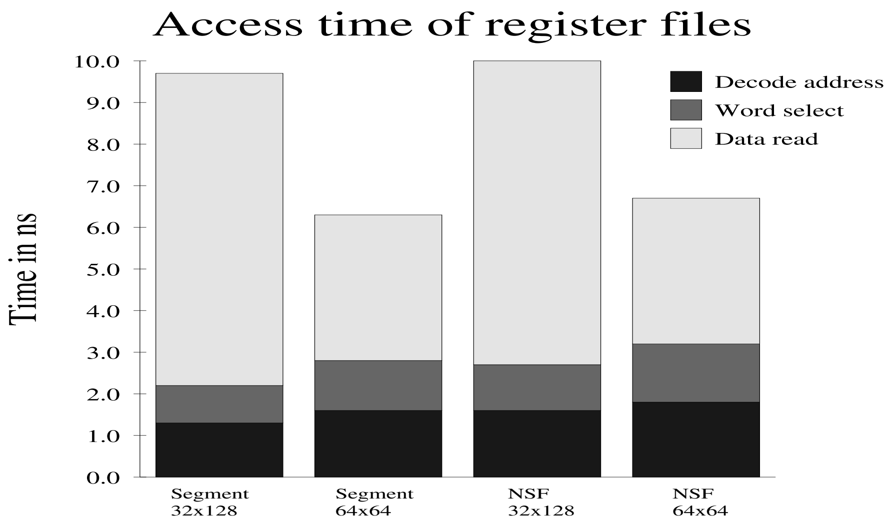

FIGURE 6. Access times of segmented and Named-State register files. Files are organized as 128 lines of 32 bits each, and 64 lines of 64 bits each. Each file was simulated by Spice in *1.2µm* CMOS process.

For both register file sizes, the time required to access the Named-State Register File was only 5% or 6% greater than for a conventional register file. Since register files are rarely in a processor's critical path [10], this should have no effect on the processor's cycle time.

6.2 Area comparison

Figure 7 illustrates the relative area of the Named-State and segmented register files in a *1.2µm* CMOS process. In this technology, a 128 row by 32 bit wide NSF is 54% larger than the equivalent segmented register file. An NSF that holds 64 rows of two registers each requires 30% more area than the equivalent segmented register file. Since most register files consume less than 10% of a processor chip area [10], the NSF should only increase processor area by 5%.

As ports are added to the register file, the area of an NSF decreases relative to segmented register files. Figure 8 estimates the relative area of segmented register files and the NSF, each with two write ports and four read ports. A 128 row by 32 bit wide Named-State register file is only 28% larger than the equivalent segmented register file. A 64 by 64 bit wide NSF is only 16% larger than the equivalent segmented register file. The area of a multiported register cell increases as the square of the number of ports. Decoder width increases in proportion to the number of ports, while miss and spill logic remains constant.

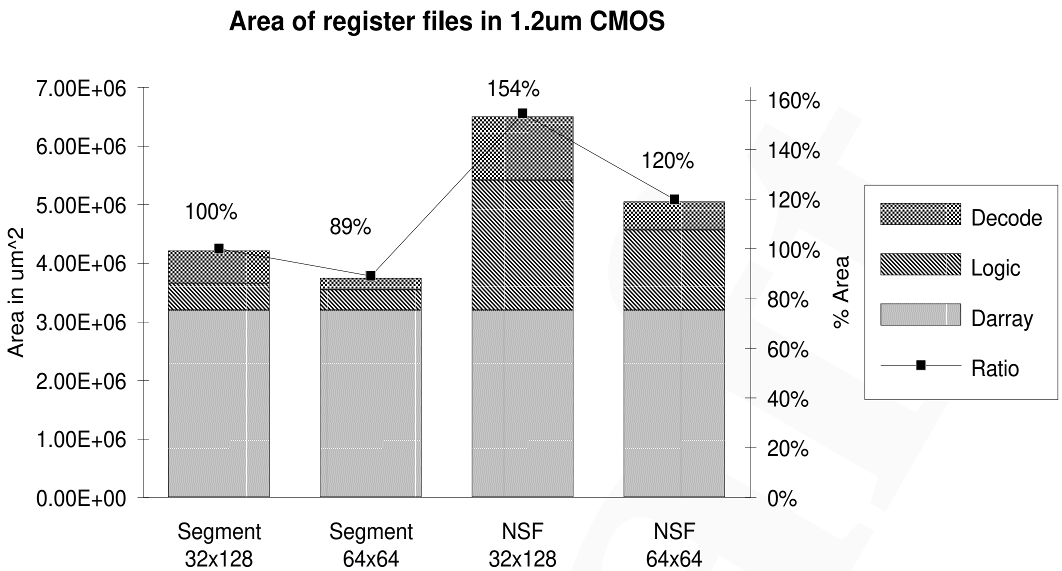

FIGURE 7. Relative area of segmented and Named-State register files in 1.2um CMOS. Area is shown for register file decoder, word line and valid bit logic, and data array. All register files have one write and two read ports.

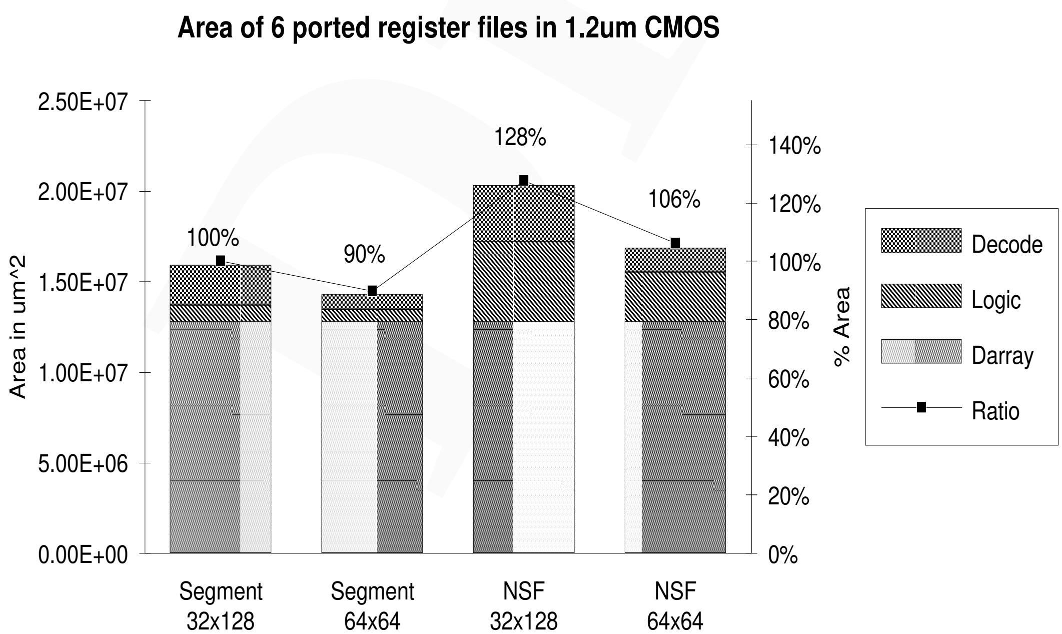

FIGURE 8. Area of six ported segmented and Named-State register files in 1.2um CMOS. Area is shown for register file decoder, word line and valid bit logic, and data array. These register files have two write and four read ports.

7. Simulation Results

A flexible register file simulator was written to evaluate the performance of the NSF on sequential and parallel applications. The simulator measured register utilization, miss rates, and spill and reload traffic for the applications listed in Table 1. The next few sections summarize these results for different register file organizations. Section 8. computes the effect of register traffic on application performance.

The sequential programs were cross-compiled from Sparc [31] assembly code. The simulator allocated a context of 20 registers for each sequential procedure activation. The parallel programs were translated from TAM [6] dataflow code. The simulator allocated a 32 register context for each thread activation. [1] describes the simulation strategy in more detail.

Benchmark	Type	Source code lines	Static instructions	Instructions executed	Avg. instr. per context switch
GateSim	Sequential	51,032	76,009	487,779,328	39
RTLSim	Sequential	30,748	46,000	54,055,907	63
ZipFile	Sequential	11,148	12,400	1,898,553	53
AS	Parallel	52	1,096	265,158	18,940
DTW	Parallel	104	2,213	2,927,701	421
Gamteb	Parallel	653	10,721	1,386,805	16
Paraffins	Parallel	175	5,016	464,770	76
Quicksort	Parallel	40	1,137	104,284	20
Wavefront	Parallel	109	1,425	2,202,186	8,280

TABLE 1. Characteristics of benchmark programs used in this chapter.
Lines of *C* or *Id* source code in each program, static instructions in the translated program, instructions executed by the simulator, and instructions executed between context switches.

7.1 Performance by application

This section compares register utilization and reload traffic for all applications, running on equivalent sized segmented and Named-State register files. The segmented file is divided into 4 equal frames, while the NSF is organized with one register per line. Each register file contains 80 registers for sequential programs and 128 registers for parallel programs.

7.1.1 Register file utilization by application

Figure 9 shows the average fraction of *active* registers in the NSF and segmented register files. It also shows the maximum number of registers that are ever active. The NSF makes better use of register area by holding more active data than the equivalent segmented file.

On average, the NSF holds active data in 70% to 80% of its registers. This is 2 to 3 times more than an equivalent segmented file for sequential programs, and 1.3 to 1.5 times more for parallel programs.

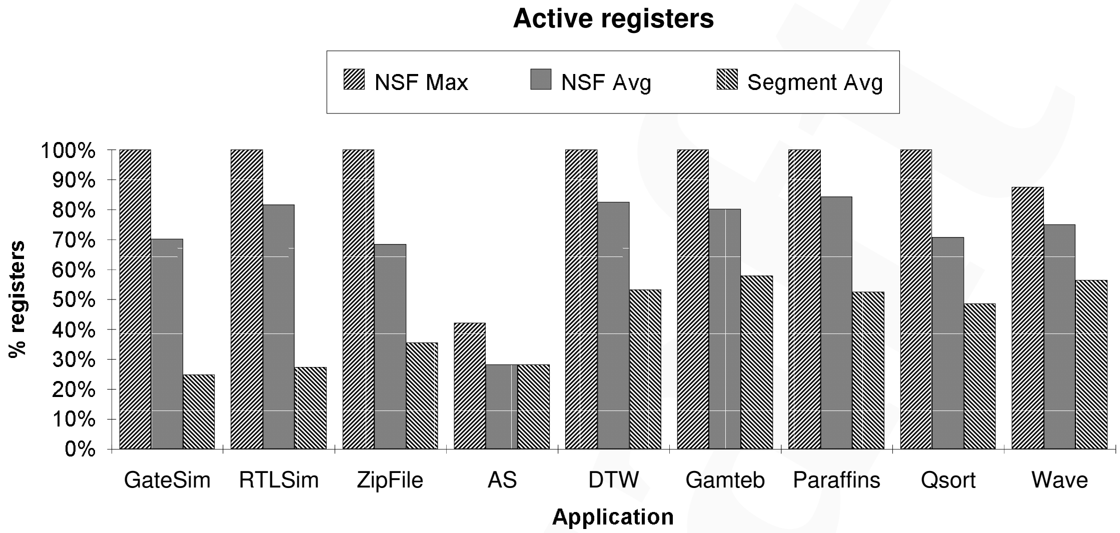

FIGURE 9. Percentage of NSF and segmented registers that contain active data. Shown are maximum and average registers accessed in the NSF, and average accessed in a segmented file. Each register file contains 80 registers for sequential simulations, or 128 registers for parallel simulations.

The difference between sequential and parallel applications is largely due to differences in compilation. The sequential compiler uses a register allocator to efficiently re-use registers. Each procedure has an average of 8-10 active registers. This results in many empty registers and poor utilization of a segmented register file. The parallel code translator simply folds hundreds of thread local variables into a context's registers, without regard to variable lifetime. This inflates the number of active registers to an average of 18-22 per parallel context, and may not accurately count register load and store traffic.

In addition, some simple parallel programs such as *AS* and *Wavefront* spawn very few parallel threads. These applications do not fill either register file with active registers.

7.1.2 Register reload traffic by application

The NSF spills and reloads dramatically fewer registers than a segmented register file. Figure 10 shows the number of registers reloaded by NSF and segmented files for each of the benchmarks. Also shown is the number of registers containing valid data reloaded by the segmented file. Every miss in the NSF reloads a single register, while each miss in the segmented file reloads an entire frame.

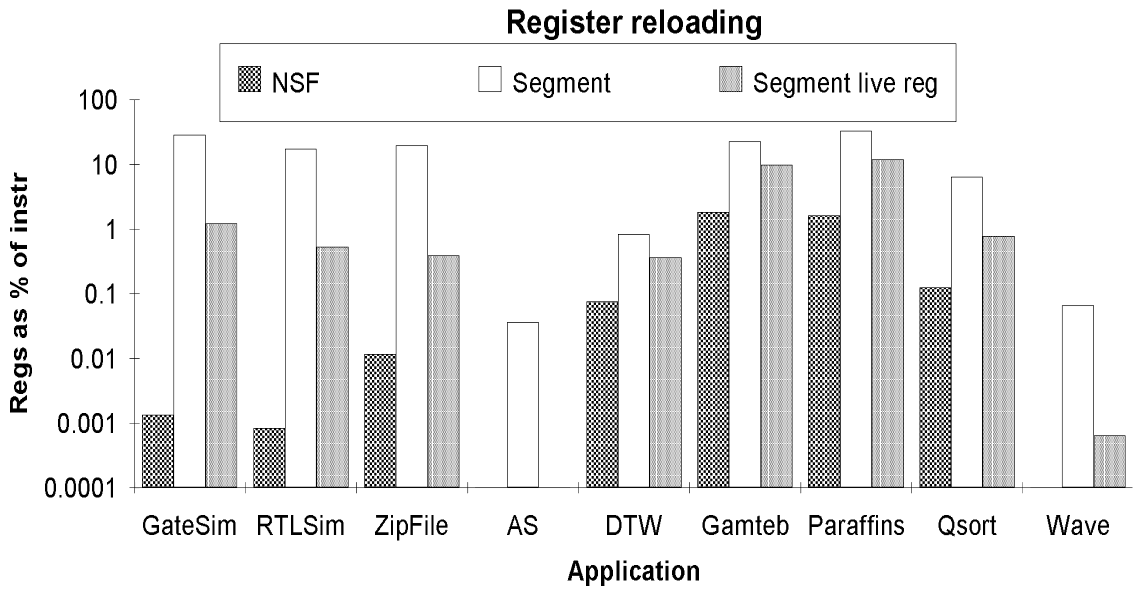

FIGURE 10. Registers reloaded as a percentage of instructions executed.
Also registers containing live data that are reloaded by segmented register file. Each register file contains 80 registers for sequential simulations, or 128 registers for parallel simulations.

For sequential applications, the segmented register file reloads 1,000 to 10,000 times as many registers as the NSF. A segmented file must reload a frame of 20 registers every 100 instructions. Even if the segmented file only reloaded registers that contained valid data, it would still reload 100 to 1,000 times more registers than a NSF. For most parallel applications, the NSF reloads 10 to 40 times fewer registers than a segmented file. If the segmented file only reloaded valid registers, it would still load 6 to 7 times as many registers as the NSF.

7.2 Performance vs. register file size

This section shows register utilization and reload traffic as a function of register file size for two representative applications: *Gatesim* and *Gamteb*. Each segmented register file is divided into frames of 20 registers for sequential code or 32 registers for parallel programs. It is compared to a Named-State Register File with the same number of registers, organized in single word lines.

7.2.1 Resident contexts vs. register file size

An NSF may hold more than twice as many resident contexts as an equivalent segmented register file. While an N frame segmented file holds at most N contexts, an NSF holds as many active contexts as can share the registers in the file. Figure 11 shows the average number of contexts resident in NSF and segmented register files as a function of register file size.

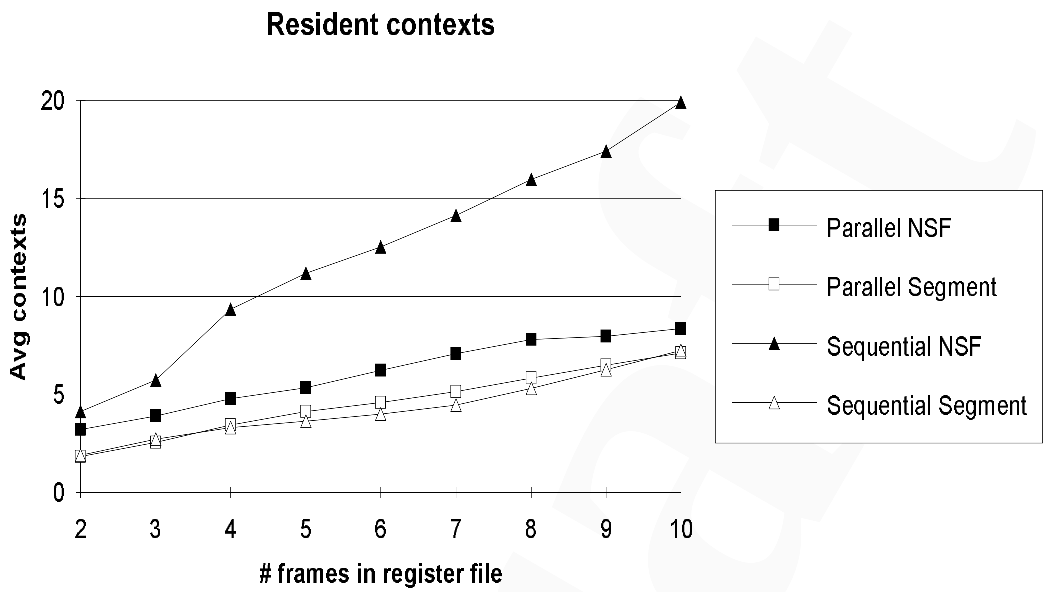

FIGURE 11. Average contexts resident in various sizes of segmented and NSF register files. Size is shown in context sized frames of 20 registers for sequential programs, 32 registers for parallel code.

Since both register files reload registers or contexts on demand, they fill on deep calls but empty on returns. The *N* frame segmented register files hold an average of *0.7N* resident contexts for both sequential and parallel code. An equivalent NSF holds an average of *0.8N* contexts for parallel code, and more than *2N* contexts for sequential code. The difference is due to poor register allocation and many active registers for parallel threads, as discussed in Section 7.1.1.

7.2.2 Register reload traffic vs. register file size

A Named-State Register File spills and reloads fewer registers than much larger segmented register files. On sequential code, the smallest NSF requires an order of magnitude fewer register reloads than any practical size of segmented register file.

As shown by Figure 12, typical segmented files reload a register every 30 instructions for sequential code. In contrast, a moderate sized NSF can hold the entire call chain of a large sequential program with almost no register spilling and reloading. A typical NSF reloads 10^{-4} as many registers as an equivalent sized segmented register file.

Parallel programs require more traffic to support more active registers per context, reloading a register every 8 instructions on an average segmented file. An NSF typically

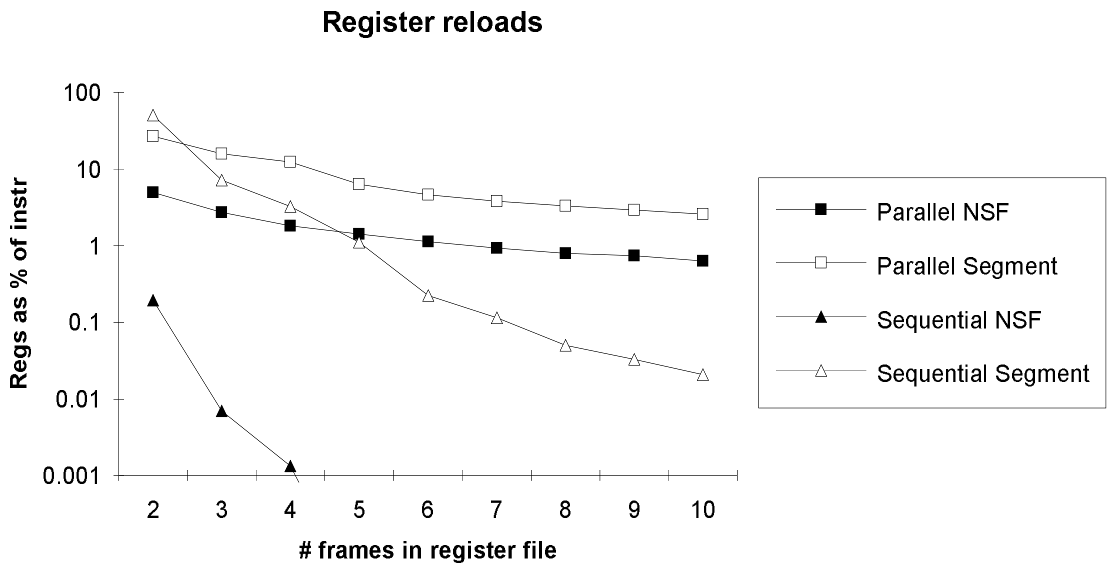

FIGURE 12. Registers reloaded as a percentage of instructions executed on different sizes of NSF and segmented register files.

reloads a register every 50 instructions. Overall, an NSF reloads 5 to 6 times fewer registers than a comparable segmented register file, and fewer registers than a segmented file that is twice as large.

7.3 Register reload traffic vs. line size

Two factors contribute to the high performance of the Named-State Register File.

- An associative decoder and small register lines that allow fine grain binding of variables to registers.
- A valid bit for each register that allows register replacement within a line.

This section demonstrates that fully-associative, fine-grain addressing of registers is more important than the ability to spill and reload individual registers[1]. Figure 13 shows the effect of line size on register reload traffic for different register file organizations. The figure compares three strategies for handling register misses. The simplest reloads the entire missing line, whether or not all of the registers contain data. Another tracks which registers contain valid data, and only spills and reloads those registers from a line. The final strategy tags each register with a valid bit, and only reloads a single register into a line on a miss.

1. The optimum block size for register spilling and reloading to the NSF also depends on the data cache latency and bandwidth.

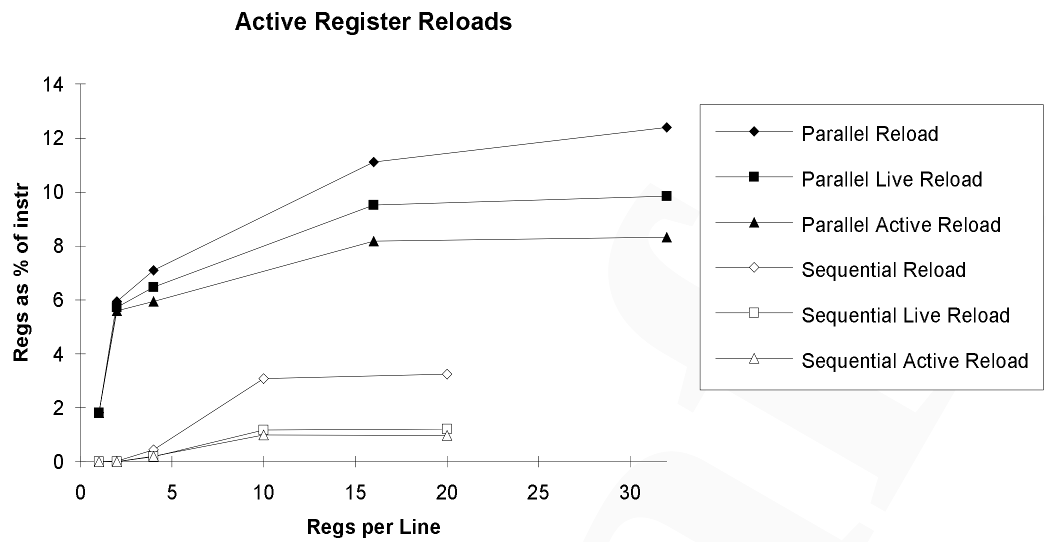

FIGURE 13. Registers reloaded as a percentage of instructions.
Three curves are shown for each application:
A. Reloaded lines * registers/line. Counts both empty registers and those containing valid data.
B. Live register reloads. Counts only registers containing valid data.
C. Active reloads. Counts registers that will be accessed while the line is resident.
Shown as a function of line size. Each file holds 80 registers for sequential simulations, 128 for parallel code.

An NSF with single word lines and valid bits is much more efficient than a segmented file with valid bits alone. A segmented file with large frames can reduce spill and reload traffic by 35% for parallel programs or by 65% for sequential code by tagging each register with valid bits. However, an NSF with single word lines reloads only 25% as many registers as a tagged segmented file on parallel code, and 1000 times less registers on sequential code. Since valid bit logic consumes a significant fraction of the NSF chip area, it is more efficient to build an NSF with small lines and fully associative decoders.

In addition, an NSF with single word lines reloads only 10% as many registers as an NSF with double word lines on sequential code, or 30% as many on parallel code. This justifies the additional cost of single word lines described in Section 6.2.

8. Application Performance

Figure 14 estimates the net effect of different register file organizations on processor performance by counting the cycles executed by each instruction in the program, and estimating the cycles required for each register spill and reload[1]. Three different sets of cycle

1. The instruction and memory access times were taken from a Sparc2 processor emulator [15].

counts are shown: timing for the NSF; for a segmented file with hardware assist for spills and reloads; and for a segmented file that spills and reloads using software trap routines.

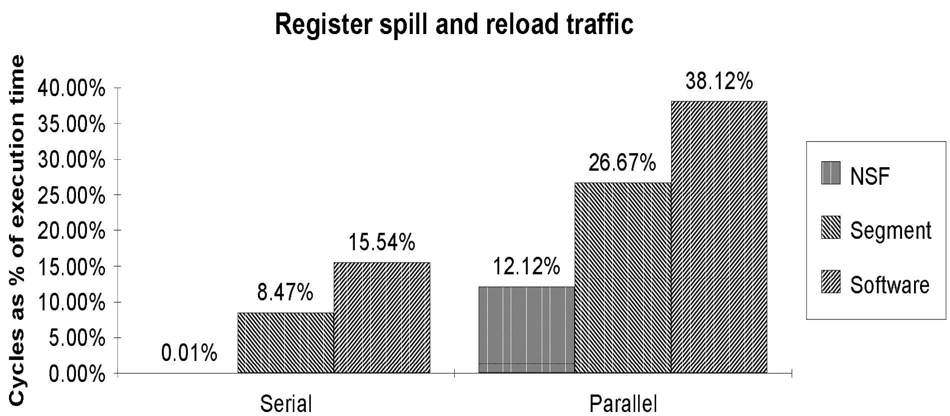

FIGURE 14. Register spill and reload overhead as a percentage of program execution time. Overhead shown for NSF, segmented file with hardware assisted spilling and reloads, and segmented file with software traps for spilling and reloads. All files hold 128 registers.

The NSF completely eliminates register spill and reload overhead on sequential programs, which for a hardware assisted segmented file accounts for 8% of execution time. The difference is almost as dramatic for parallel programs, cutting overhead from 28% for the segmented file to 12% for the NSF.

9. Conclusion

The Named-State Register File enables fast switching among parallel and sequential procedure activations while making efficient use of register space. The NSF uses hardware and software mechanisms to dynamically allocate the register set among active threads. The NSF allows a processor to interleave many more threads than segmented files. The NSF keeps more active data resident than segmented files, since it is not coarsely fragmented among threads. It spills and reloads far fewer registers than segmented files, since it only loads registers as they are needed.

- The NSF holds more active data than a conventional register file with the same number of registers. For the large sequential and parallel applications tested, the NSF holds 30% to 200% more active data than an equivalent register file.
- The NSF holds more concurrent active contexts than conventional files of the same size. The NSF holds twice as many procedure call frames as a conventional file for sequential programs, and holds 20% more contexts for parallel applications.

- The NSF is able to support more resident contexts with less register spill and reload traffic. The NSF can hold the entire call chain of a large sequential application, spilling registers at 10^{-4} the rate of a conventional file. On parallel applications, the NSF reloads 10% as many registers as a conventional file.
- The NSF speeds execution of sequential applications by 9% to 18%, and parallel applications by 17% to 35%, by eliminating register spills and reloads.
- The NSF's access time is only 5% greater than conventional register file designs. This should have no effect on processor cycle time.
- The NSF requires 16% to 50% more chip area to build than a conventional file. This requires only 1% to 5% of a typical processor's chip area.

The simulations in this study indicate that the Named-State Register File may significantly increase the performance of both sequential and parallel applications at very little cost in chip area or complexity.

References

[1] Ph.D. thesis.

[2] Anant Agarwal. Performance tradeoffs in multithreaded processors. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):525–539, September 1992.

[3] Anant Agarwal et al. Sparcle: An evolutionary processor design for large-scale multiprocessors. *IEEE Micro*, June 1993.

[4] A. D. Berenbaum, D. R. Ditzel, and H. R. McLellan. Architectural innovations in the CRISP microprocessor. In *CompCon '87 Proceedings*, pages 91–95. IEEE, January 1987.

[5] G. J. Chaitin et al. Register allocation via graph coloring. *Computer Languages*, 6(47-57):130, December 1982.

[6] David E. Culler et al. Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 164–175. ACM, April 1991.

[7] James R. Goodman and Wei-Chung Hsu. On the use of registers vs. cache to minimize memory traffic. In *13th Annual Symposium on Computer Architecture*, pages 375–383. IEEE, June 1986.

[8] Anoop Gupta and Wolf-Dietrich Weber. Exploring the benefits of multiple hardware contexts in a multiprocessor architecture: Preliminary results. In *Proceedings of 16th Annual Symposium on Computer Architecture*, pages 273–280. IEEE, May 1989.

[9] D. Halbert and P. Kessler. Windows of overlapping register frames. In *CS 292R Final Reports*, pages 82–100. University of California at Berkeley, 1980.

[10] John L. Hennessy. VLSI processor architecture. *IEEE Transactions on Computers*, C-33(12), December 1984.

[11] Yasuo Hidaka, Hanpei Koike, and Hidehiko Tanaka. Multiple threads in cyclic register windows. In *International Symposium on Computer Architecture*, pages 131–142. IEEE, May 1993.

[12] Waldemar Horwat, Andrew Chien, and William J. Dally. Experience with CST: Programming and implementation. In *Proceedings of the ACM SIGPLAN 89 Conference on Programming Language Design and Implementation*, pages 101–109, 1989.

[13] Miquel Huguet and Tomas Lang. Architectural support for reduced register saving/restoring in single-window register files. *ACM Transactions on Computer Systems*, 9(1):66–97, February 1991.

[14] Robert Iannucci. Toward a dataflow/von Neumann hybrid architecture. In *International Symposium on Computer Architecture*, pages 131–140. IEEE, 1988.

[15] Gordon Irlam. *Spa - A SPARC performance analysis package*. gordoni@cs.adelaide.edu.au, Wynn Vale, 5127, Australia, 1.0 edition, October 1991.

[16] Robert H. Halstead Jr. and Tetsuya Fujita. MASA: a multithreaded processor architecture for parallel symbolic computing. In *15th Annual Symposium on Computer Architecture*, pages 443–451. IEEE Computer Society, May 1988.

[17] David Keppel. Register windows and user-space threads on the Sparc. Technical Report 91-08-01, University of Washington, Seattle, WA, August 1991.

[18] Tokuzo Kiyohara et al. Register Connection: A new approach to adding registers into instruction set architectures. In *International Symposium on Computer Architecture*, pages 247–256. IEE, May 1993.

[19] James Laudon, Anoop Gupta, and Mark Horowitz. Architectural and implementation tradeoffs in the design of multiple-context processors. Technical Report CSL-TR-92-523, Stanford University, May 1992.

[20] Beng-Hong Lim and Anant Agarwal. Waiting algorithms for synchronization in large-scale multiprocessors. VLSI Memo 91-632, MIT Lab for Computer Science, Cambridge, MA, February 1992.

[21] D. R. Miller and D. J. Quammen. Exploiting large register sets. *Microprocessors and Microsystems*, 14(6):333–340, July/August 1990.

[22] L. W. Nagel. SPICE2: A computer program to simulate semiconductor circuits. Technical Report ERL-M520, University of California at Berkeley, May 1975.

[23] Rishiur S. Nikhil and Arvind. Can dataflow subsume von Neumann computing? In *International Symposium on Computer Architecture*, pages 262–272. ACM, June 1989.

[24] Gregory M. Papadopoulos and David E. Culler. Monsoon: an explicit token-store architecture. In *The 17th Annual International Symposium on Computer Architecture*, pages 82–91. IEEE, 1990.

[25] Gordon Russell and Paul Shaw. A stack-based register set. University of Strathclyde, Glasgow, May 1993.

[26] Richard L. Sites. How to use 1000 registers. In *Caltech Conference on VLSI*, pages 527–532. Caltech Computer Science Dept., 1979.

[27] Burton J. Smith. Architecture and applications of the HEP multiprocessor computer system. In *SPIE Vol. 298 Real-Time Signal Processing IV*, pages 241–248. Denelcor, Inc., Aurora, Col., 1981.

[28] Burton J. Smith et al. The Tera computer system. In *International Symposium on Computer Architecture*, pages 1–6. ACM, September 1990.

[29] V. Soundararajan. Dribble-Back registers: A technique for latency tolerance in multiprocessors. BS Thesis MIT EECS, June 1992.

[30] Peter Steenkiste. Lisp on a reduced-instruction-set processor: Characterization and optimization. Technical Report CSL-TR-87-324, Stanford University, March 1987.

[31] Sun Microsystems. *The SPARC Architectural Manual*, v8 #800-1399-09 edition, August 1989.

[32] J. E. Thornton. *Design of a Computer: The Control Data 6600*. Scott, Foresman & Co., Glenview, IL, 1970.

[33] Carl A. Waldspurger and William E. Weihl. Register Relocation: Flexible contexts for multithreading. In *International Symposium on Computer Architecture*, pages 120–129. IEEE, May 1993.

[34] David W. Wall. Global register allocation at link time. In *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*, 1986.