The Named-State Register File: Implementation and Performance

Peter R. Nuth
Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94304
nuth@hpl.hp.com

William J. Dally
MIT Artificial Intelligence Laboratory
545 Technology Square
Cambridge, MA 02139
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.

1. Introduction

Both sequential and parallel applications run as a chain of procedure activations. The local variables of each activation compete for space in a processor's register file. Good compilers can efficiently allocate registers between procedures for many sequential programs [31]. But interprocedural register allocation is less effective for dynamic parallel programs [4,9,11], in which a processor may switch between many concurrent activations, or *threads*. 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 [7]. 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 efficiently using register space. It does this without sacrificing sequential thread performance, and can often run sequential programs more efficiently than conventional register files.

The NSF 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. The NSF has a slightly longer access time, and requires more chip area per bit than conventional register files. However, 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.

2. Multithreaded Processors

Fine grain parallel programs are often composed of concurrent light-weight threads in a common address space. These programs may send messages between processors every 75 to 100 instructions [9], each of which requires a round trip latency of more than 200 instruction cycles [1]. Parallel threads also synchronize with other threads every 20 to 80 instructions [4], and may wait an unbounded amount of time at any synchronization point [18]. Rather than stalling for each remote access or synchronization point, a *multithreaded processor* switches to another parallel thread.

Multithreaded processors [24,13,1,25] reduce context switch time by holding the state of several threads in the processor's registers. Figure 1 describes a typical multithreaded register file, divided into a few register *frames*, each of which holds the registers of a different thread. This

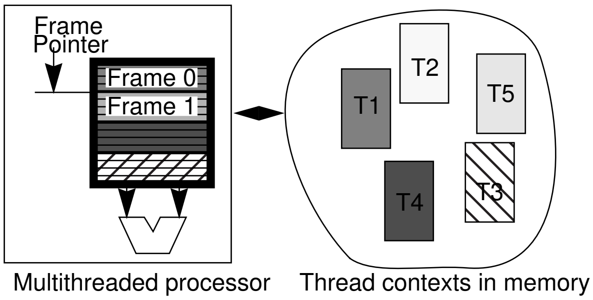


FIGURE 1. A multithreaded processor using a segmented register file. The register file is segmented into equal sized frames, one for each concurrent thread. The processor spills and restores thread contexts from register frames into main memory.

segmented register file can quickly switch between only a few active contexts.

Dividing the register file into large, fixed sized frames wastes space in the register file. Some threads will not need all the registers in a frame. Also, if the processor switches contexts frequently, it will 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.

Multithreaded processors may interleave instructions from different threads on a cycle-by-cycle basis [14,24,13,22,17], or as blocks of instructions [5,1]. Although the techniques introduced in this paper are applicable to both forms of multithreading, this discussion will concentrate on block multithreading.

This paper does not analyze the full costs and benefits of multithreaded processors, but only introduces a more efficient register file for such a processor. Other research has studied how instruction issue logic and caches are shared among the threads of a multithreaded processor [17,29].

3. 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.

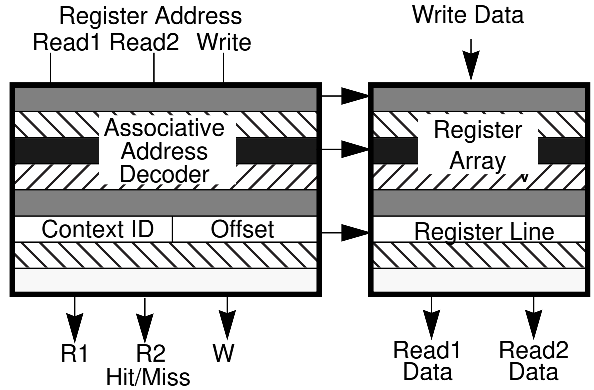


FIGURE 2. Structure of the Named-State Register File. The NSF holds registers from a number of resident contexts. The processor spills and restores individual registers to main memory as needed by the active threads.

The NSF could be used simply as a large conventional register file. A program could allocate and deallocate register variables and explicitly save and restore registers to memory on a context switch. But NSF hardware can instead spill and restore registers on demand, as needed by the program. After a context switch, the processor simply issues instructions from the new thread. If any register operands for the new thread are not resident, the NSF reloads them on demand.

The NSF allows a multithreaded 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.

3.1 Structure of the NSF

The NSF is multi-ported, as are conventional register files. Figure 2 shows a three ported register file. Each line of the NSF contains a *content addressable memory* (CAM) that holds a register address, and a register line of typically one to four registers. For each register read or write, the NSF matches against all the decoder lines in parallel, and selects the appropriate register cell. On the first write to a new register, the NSF allocates a free line to that register by writing its address into the associative decoder. The NSF always maintains a free line to ensure that writes complete.

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 stalls the processor pipeline and reloads that register from memory. Depending on its design, an NSF may reload only the register that missed, or the entire line containing that register. Write misses either reload a line or simply allocate a new line for a register. This strategy causes several instructions to stall during the lifetime of a context, but ensures that the NSF only loads registers that are needed. Better utilization of the NSF register file [and reduced demands on memory bandwidth] more than compensates for the additional misses on register fetches.

A register address in the NSF consists of a *Context ID* and an *offset*. The current instruction specifies the register offset, and a processor status word supplies the current Context ID. A context may be the state of a sequential procedure activation, a parallel thread, or some code block. The NSF places no restrictions on how Context IDs are used by different programming models. Context ID management is discussed in more detail in [21].

By matching on a full register address, or on just the Context ID, the NSF can explicitly *deallocate* a register after it is no longer needed, or can deallocate all registers in a context[1]. This frees the lines to be used by a new set of register variables. After allocating the last available register line in the register file, the NSF must spill a line out of the register file and into memory. To select a victim, the NSF uses a *pseudo least recently used* (LRU) strategy [8].

3.2 NSF and memory hierarchy

Figure 3 shows how the NSF 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. Both read and write operand CIDs are translated through the Ctable in the pipeline stage before operand fetch, in case either operand misses in the NSF. 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 Named-State register file is not a data cache. The NSF is fast, multi-ported and close to the pipeline.

1. Deallocation requires changes to a processor's ISA. Our implementation uses a single instruction to deallocate the current context, and a single bit in each instruction to read, then deallocate register operand 1.

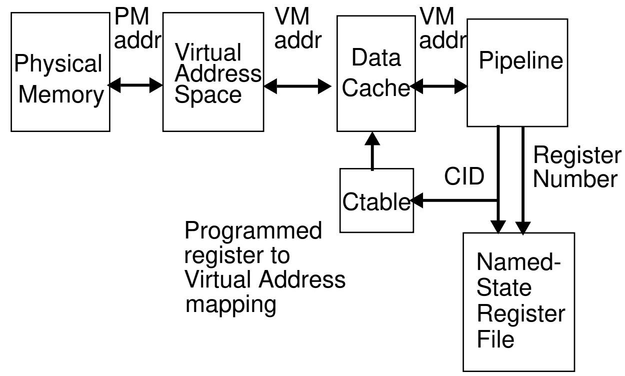


FIGURE 3. The Named-State Register File and memory hierarchy. The NSF addresses registers using a <Context ID: Offset> pair. The Ctable is a short table indexed by CID that returns a virtual address.

Programs refer to registers using short indices. But more importantly, the NSF defines a register name space separate and much smaller than the virtual address space. A compiler can efficiently manage that register space. Programs allocate, read, write and deallocate registers in the NSF as with conventional register files. But the NSF frees applications from explicitly spilling and restoring registers. Instead, the program sets Ctable entries to point to stack or context frames in memory, and the NSF manages the registers implicitly.

4. Related Work

Keppel [15] and Hidaka [7] propose running multiple concurrent threads in the register windows of a Sparc [28] processor by modifying window trap handlers. The Sparcle chip [1] adds trap hardware and tuned trap handlers to a Sparc chip. Arvind [20] and Agarwal [26] 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 2., 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 [30] 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. 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 [2] is a register-less architecture that stores the top of stack in a multiported stack buffer on chip. Russell and Shaw [23] 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 [10], Miller and Quammen [19], and Kiyohara [16] 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.

5. Implementation

Figure 4 shows a simple CAM cell used for the NSF address decoder. It is a register cell connected to three pre-charged NAND address comparators. When a free line is allocated to a new register, signal *alloc* loads its address into the CAM cell. Subsequent register read and writes that match this address pull down on the appropriate *mch#* line. The register word line is enabled if both CID and offset fields match the operand address.

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. The structure and implementation of the Named-State Register File is described in more detail in [21].

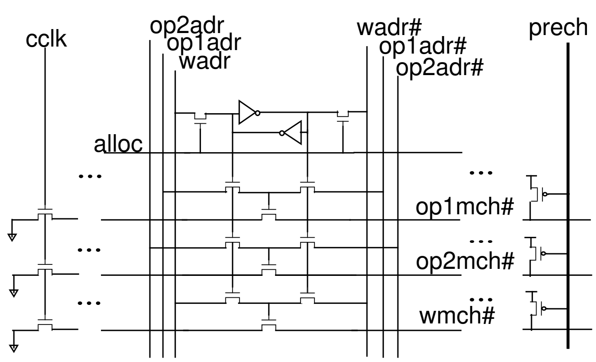


FIGURE 4. One bit of a programmable address decoder.

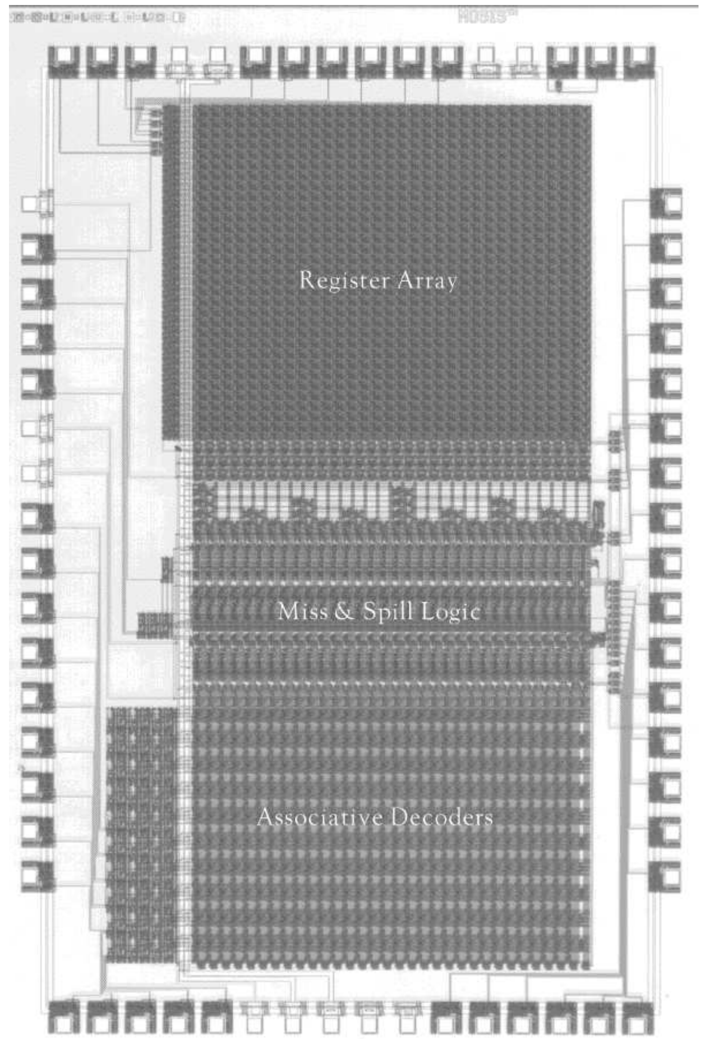


FIGURE 5. A prototype Named-State Register File. This prototype chip includes a 32 bit by 32 line register array, a 10 bit wide fully-associative decoder, and logic to handle misses, spills and reloads. The register file has two read ports and a single write port.

5.1 Performance comparison

Figure 6 shows circuit simulations of the Named-State Register File and a conventional register file. Both used pre-charged, NAND pull-down decoders. 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.

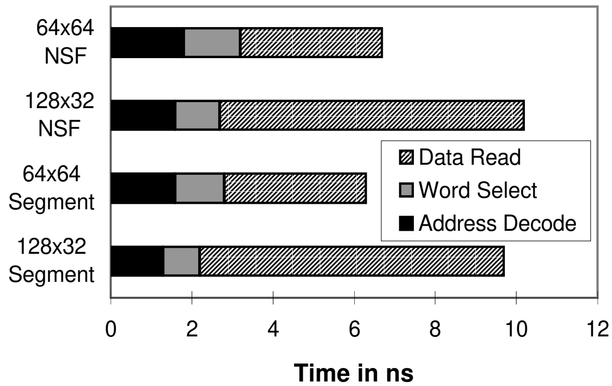


FIGURE 6. Access times of segmented and Named-State register files. Files are organized as 128 lines of 32 bits each, and 64 lines of 64 bits each. Each file was simulated by Spice in *1.2*μ*m* CMOS process.

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 not always on a processor's critical path [6], this might not effect the processor's cycle time.

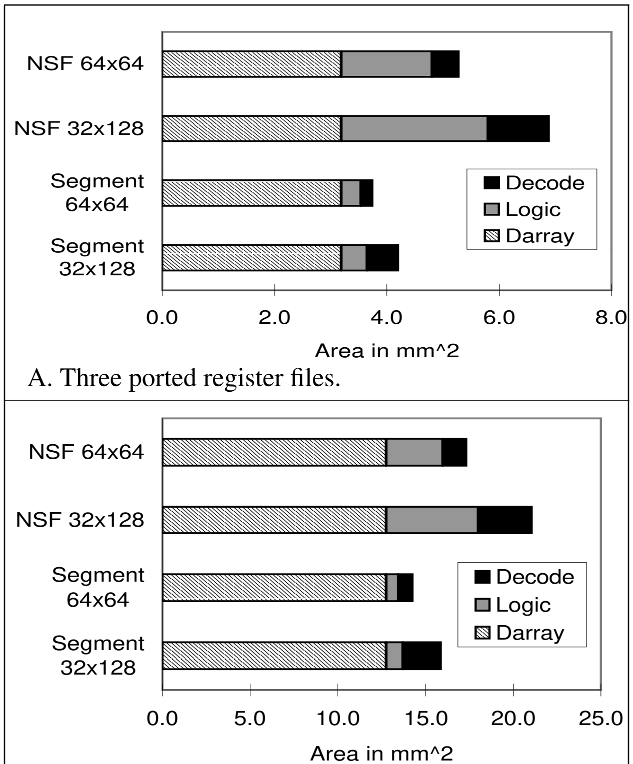
5.2 Area comparison

Figure 7(A) illustrates the relative area of three ported Named-State and segmented register files in a *1.2*μ*m* 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 [6], the NSF would increase processor area by less than 5%.

As ports are added to the register file, the area of an NSF decreases relative to segmented register files. Figure 7(B) compares six-ported register files. 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 decoders and register cells increases with the number of ports, while miss and spill logic remains constant.

6. Simulation Results

A flexible register file simulator was written to evaluate the performance of the NSF on sequential and fine-grain parallel applications. The simulator measured register



A. Three ported register files.

B. Six ported register files.

FIGURE 7. Area of segmented and Named-State register files in 1.2um CMOS. Area is shown for register address decoder, word line and valid bit logic, and data array.
A. Register files with one write and two read ports.
B. Register files with two write and four read ports.

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 7. summarizes the effect of register load and spill traffic on application performance.

The sequential programs were cross-compiled from Sparc [28] assembly code. The simulator allocated a context of 20 registers for each sequential procedure activation. The parallel programs were translated from TAM [4] dataflow code. The simulator allocated a 32 register context for each thread activation. Neither program deallocated individual registers at the end of their lifetime. Instead, the code deallocated entire contexts on procedure and thread return. Such register lifetime analysis would significantly improve NSF performance. The simulation strategy is described in more detail in [21].

Program	Source instructions	Executed instructions	Instrs/ context
Sequential			
GateSim	76,009	487,779,328	39
RTLSim	46,000	54,055,907	63
ZipFile	12,400	1,898,553	53
Parallel			
AS	1,096	265,158	18,940
DTW	2,213	2,927,701	421
Gamteb	10,721	1,386,805	16
Paraffins	5,016	464,770	76
Quicksort	1,137	104,284	20
Wavefront	1,425	2,202,186	8,280

TABLE 1. Characteristics of benchmark programs used in this chapter. Count of instructions in each program, instructions executed by the simulator, and instructions executed between context switches.

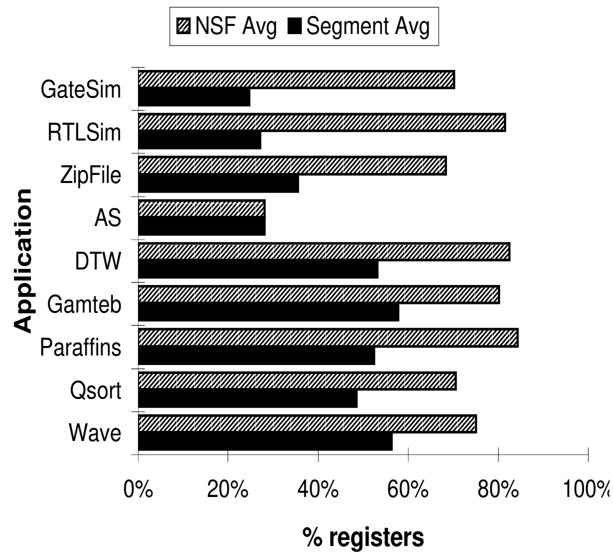


FIGURE 8. Average percentage of NSF and segmented register files that contain active data. Each register file contains 80 registers for sequential simulations, or 128 registers for parallel simulations.

6.1 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.

6.1.1 Register file utilization by application

Figure 8 shows the average fraction of register in use by the NSF and segmented register files. 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.Some of the difference between sequential and parallel applications is due to compilation. The sequential compiler uses a register allocator to efficiently re-use registers. Each procedure has an average of 8-10 active registers. It is far more difficult to allocate registers for fine-grain parallel code [4]. The Id translator simply folds all of a thread's local variables into a context's registers, without regard to variable lifetime. This increases the number of active registers to an average of 18-22 per parallel context. With a better register allocator, the NSF would perform better relative to a segmented file.

6.1.2 Register reload traffic by application

The NSF spills and reloads dramatically fewer registers than a segmented register file. Figure 9 shows the number of registers reloaded by NSF and segmented files for each of the benchmarks. Every miss in the NSF reloads a single register, while each miss in the segmented file reloads an entire frame. Also shown are the registers containing live data reloaded by the segmented file. An efficient register management scheme would only reload live registers.

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. For most parallel applications, the NSF reloads 10 to 40 times fewer registers than a segmented file. 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 sequential programs, and 6 to 7 times more for parallel code. The NSF wins both only reloading registers as needed, and by efficiently packing those registers in the file.

6.2 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

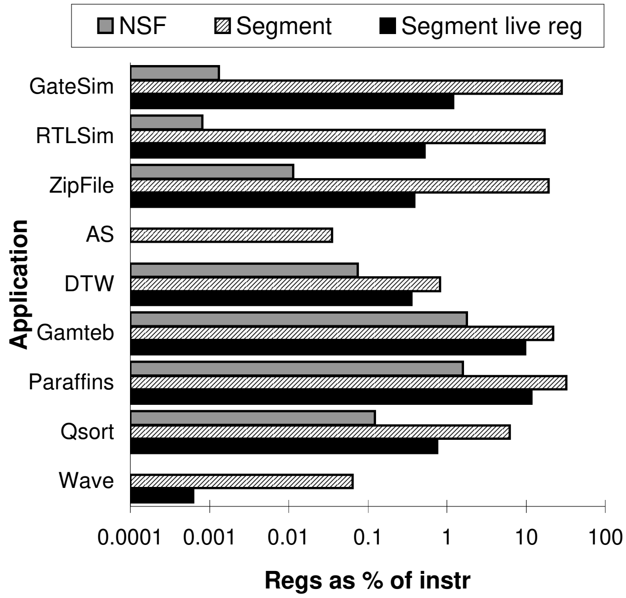


FIGURE 9. Registers reloaded as a percentage of instructions executed for NSF, and for oblivious and intelligent segmented files. Each register file contains 80 registers for sequential simulations, or 128 registers for parallel simulations.

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.

6.2.1 Resident contexts vs. register file size

An NSF holds 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 10 shows the average number of contexts resident in NSF and segmented register files as a function of register file size.

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 6.1.1.

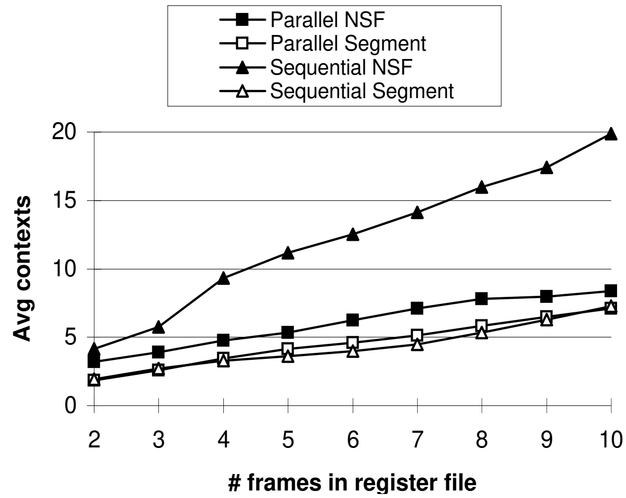


FIGURE 10. Average contexts resident in various sizes of segmented and NSF register files.
Size is shown in context sized frames of 20 registers for sequential programs, 32 registers for parallel code.

6.2.2 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.

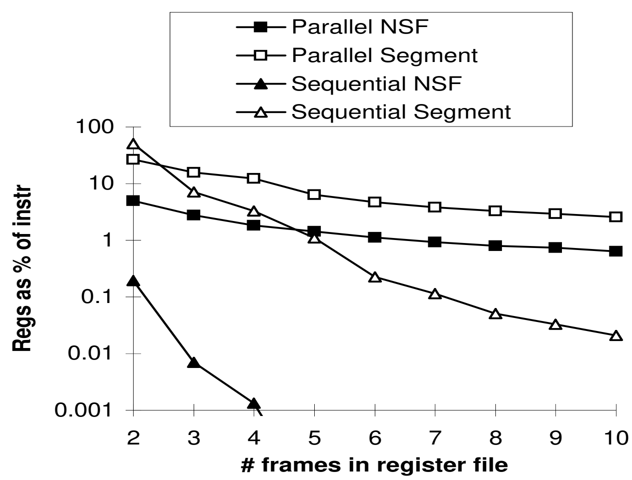


FIGURE 11. Registers reloaded as a percentage of instructions executed on different sizes of NSF and segmented register files.

As shown by Figure 11, 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 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.

6.3 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 12 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.

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 [21], it is more efficient to build an NSF with small lines and fully associative decoders.

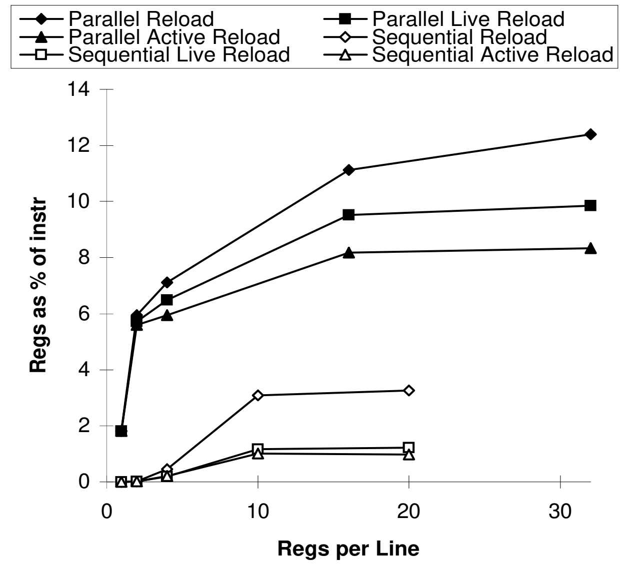


FIGURE 12. Registers reloaded as a percentage of instructions. Three curves are shown for each application:
A. All registers: Both empty and live registers.
B. Live registers: Containing valid data.
C. Active registers: Which will be accessed before line is swapped out again.
Shown as a function of line size. Each file holds 80 registers for sequential simulations, 128 for parallel code.

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 5.2.

7. Application Performance

Figure 13 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[2]. Three different sets of cycle 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.

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.

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

2. The instruction and memory access times were taken from a Sparc2 processor emulator [12].

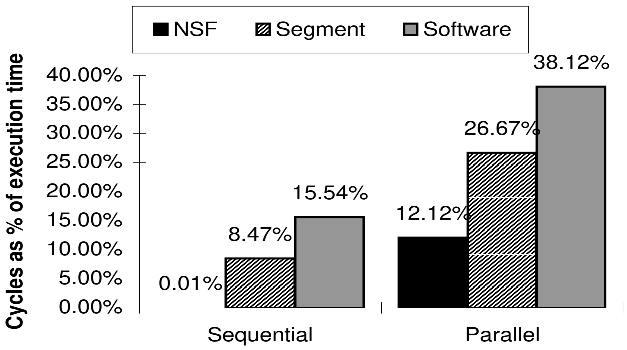


FIGURE 13. Register spill and reload overhead as a percentage of program execution time.
Overhead shown for NSF, segmented file with hardware assisted spilling and reloads, and segmented file with software traps for spilling and reloads. All files hold 128 registers.

The difference is almost as dramatic for parallel programs, cutting overhead from 28% for the segmented file to 12% for the NSF.

8. Conclusions

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.

Acknowledgments

This work was supported in part by the Defense Advanced Research Projects Agency under ARPA orders 6370 and 8272 and monitored by the Office of Naval Research under contract N00014-88-K-0738 and by the Air Force Electronic Systems Division under contract F19628-92-C-0045. Additional support provided by a National Science Foundation Presidential Young Investigator Award with matching funds from General Electric Corporation, IBM Corporation and AT&T.

David Harris designed circuits and layout, and built the NSF prototype chip. Theodore Tonchev wrote the first version of the TL0 translator. Professors Tom Knight and Greg Papadopoulos helped oversee this research. Stuart Fiske, Mike Noakes, and Steve Keckler proof-read early drafts of this work.

References

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

[2] 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.

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

[4] 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.

[5] 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.

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

[7] 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.

[8] Kirk Holden and Steve McMahan. Integrated memory management for the MC68030. In *International Conference on Computer Design*, pages 586–589. IEEE, October 1987.

[9] 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.

[10] 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.

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

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

[13] 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.

[14] Stephen W. Keckler and William J. Dally. Processor coupling: Integrating compile time and runtime scheduling for parallelism. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 202–213, Queensland, Australia, May 1992. ACM.

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

[16] 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.

[17] 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.

[18] 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.

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

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

[21] Peter R. Nuth. The Named-State Register File. AI-TR 1459, MIT Artificial Intelligence Laboratory, Cambridge, MA, August 1993. Ph.D. Thesis.

[22] 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.

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

[24] 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.

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

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

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

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

[29] R. Thekkath and S.J. Eggers. Impact of sharing-based thread placement on multithreaded architectures. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 176–186. IEEE, April 1994.

[30] 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.

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