CONTENTS

CONTRIBUTORS

Peter R. Nuth
William J. Dally
Artificial Intelligence Lab,
Massachusetts Institute of Technology

1

NAMED STATE AND EFFICIENT CONTEXT SWITCHING

Peter R. Nuth and William J. Dally

Artificial Intelligence Lab,
Massachusetts Institute of Technology,
Cambridge, Massachusetts 02139

ABSTRACT

Context switches are slow in conventional processors because the entire processor state must be saved and restored, even if much of the restored state is not used before the next context switch. This unnecessary data movement is required because of the coarse granularity of binding between names and registers. In this paper we introduce the *Named-State Register File* (NSF), which binds variable names to individual registers. This allows context switches to be very inexpensive, since registers are only loaded and saved out as needed. Analysis shows that the Named-State Register File uses registers more efficiently than a multithreaded register file, and supports more tasks without spilling to memory. Circuit simulations indicate that the access time of a NSF is only 6% greater than a conventional register file. The NSF only requires 25% more VLSI chip area to implement than a conventional register file.

1 INTRODUCTION

This paper proposes a mechanism to support fast context switching in a processor without compromising single thread performance.

Fast context switching is one way of addressing the "Two Fundamental Problems" of multicomputer systems described by Arvind and Iannucci [1]. The first is communication latency for messages across a multicomputer network. Communication latency has not kept pace with decreasing processor cycle times. Even very low-latency networks [8] have round trip message latencies of $\approx 1\mu s$. This is equivalent to 50 instructions of a 50 MHz processor. Context switching allows us to mask this time with useful processing.

The second problem is synchronization, since highly parallel programs may consist of short tasks that frequently synchronize with each other. The average run length of such a task may be 20-50 instructions [6]. Rather than stalling at every synchronization point, context switching allows the processor to run another task.

Most conventional processors are unsuitable for nodes of a multicomputer system, because they require too much time to switch between concurrent tasks. A context switch on a conventional processor takes 50-500 instructions, because processor state must be saved and restored and the addressing environment must be switched.

Saving and restoring the entire process state on a context switch is wasteful since only a few (typically 2 or 3) words of this state are required by the next instruction of the new process. If context switches are frequent, as in some types of multicomputers [7, 6], much of the restored state may not be used before it is saved for the next context switch. This unnecessary data movement is required because the granularity of binding between names and registers is very coarse. A large block of physical registers is allocated to a single process. To switch processes, the value in each register is saved and the registers are reloaded with values bound to names in the new process.

Multithreaded processors [2, 3, 12, 14, 4] have been proposed as an alternative. In the following sections, we discuss how one might build such a multithreaded processor. We introduce the *Named-State Register File*, a fully-associative register file, as an alternative to the segmented register file of a conventional multithreaded processor. We show how a fine grain binding between names and registers in the NSF improves its efficiency in supporting many concurrent tasks. We show how it can be built in conventional VLSI technology. Finally, we analyze its access time and the chip area required to implement it, and show that it is competitive with conventional register file designs.

This report introduces research described more fully in [9]. That document explains the implementation of Named-State register files, and describes their performance running sequential and parallel applications.

This proposal has been influenced by the work of Halstead [7], Iannucci [6], and Sites [11]. In related work, Omondi has proposed using a cache in a multithreaded memory-accumulator machine to resolve data dependencies [10]. Waldspurger [15] proposed a register file that can be shared among different sized contexts by run-time software. And Agarwal [13] has suggested techniques for background loading and spilling of registers.

2 MULTITHREADED PROCESSORS

A conventional multithreaded processor [12, 4] partitions a single large register file among a small set of *resident* tasks. Figure 1 shows such a segmented register file. Each task is given a distinct frame of the register file. When the processor switches contexts, it moves its frame pointer to another frame. Instructions refer to registers as simple offsets from the frame pointer.

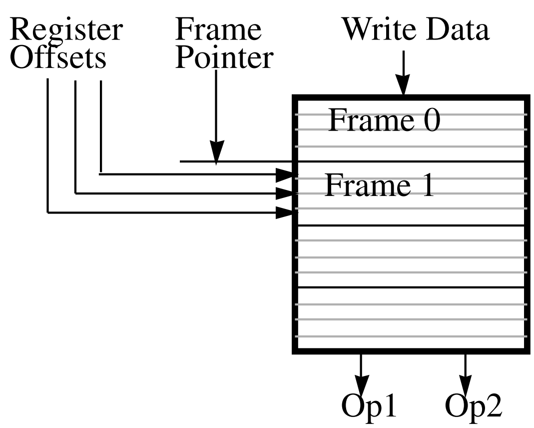

Figure 1 A segmented register file for a multithreaded processor.

Note that the processor can quickly switch among this small set of concurrent tasks. But switching outside of this set is very slow, since the processor must save and restore an entire frame's worth of registers. As multicomputers become larger, and communication latencies increase, processors must compensate by supporting more active tasks. A small fixed limit on the number of concurrent tasks reduces the processor performance.

Also note that this static partitioning is an inefficient use of processor registers. Not all of the thread's registers will be live at any given point. Since each task is allocated the same fixed sized frame, many registers in the register file will contain dead or unused variables. This is a serious problem, since the register file, as the memory nearest to the processor's ALU, is the most precious realestate in the machine. A more efficient scheme would hold only current, live data in the register file.

The fundamental problem with this segmented register file organization is the same as with conventional register files. The processor is binding a set of variable names (for a task) to an entire block of registers (a frame). A more efficient organization would bind variable names to registers at a finer granularity. This is done in the *Named-State Register File*.

3 THE NAMED-STATE REGISTER FILE

The Named-State Register File (NSF) is an alternative register file organization. It is a true register file, but instead of being segmented into a small number of frames, it is fully-associative. The registers belonging to some task may be distributed anywhere in the register array, not necessarily in one continuous block. In this way, the NSF binds variable names to individual registers. A value in a register must be saved only when the register is needed for another purpose, and only those values that are immediately needed and no longer resident must be loaded after a context switch.

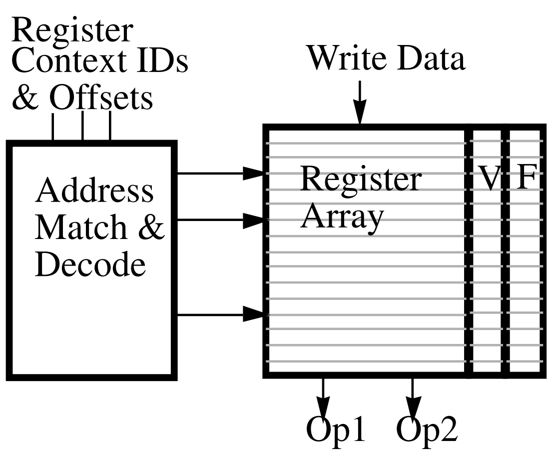

Figure 2 The structure of the Named-State Register File.

Figure 2 shows the organization of the Named-State Register File. The NSF retains the advantages [11] of a conventional register file:

- It has two read ports and a single write port, to support conventional 3 operand instructions.
- Access time for NSF registers is comparable to conventional register files.
- The NSF uses short register *offsets* to refer to registers.

Several features distinguish the Named-State Register File from conventional register files. In order to support fully-associative lookup, each line in the NSF is tagged with the *Context ID* (CID) and *offset* of the variable occupying that location. The processor appends the current Context ID to each offset to present a full address to the register file.

Each line in the NSF also contains two flag bits. The *valid* (V) bit indicates whether a line is in use. If set, it means that the line has been bound to a variable with a particular CID and offset. Unlike conventional cache valid bits, this bit is set and cleared under software control. The first write to a context register typically allocates a line for that register in the Named-State Register File, and stores a value in that location. Similarly, an instruction may deallocate a register on the last read of that register. This explicitly frees a line so that it can be allocated to another variable. Otherwise, a special instruction may deallocate all lines associated with a particular context in the register file.

The *full* (F) bit is used to synchronize through a NSF word. An instruction may set or clear the full bit on some valid word to synchronize with another task. Full bits can be used for semaphores or to support producer-consumer parallelism.

The NSF does not explicitly flush any data on a context switch. The instructions of the new thread read and write registers as usual. If the thread reads a register that is not present in the register file, that register value is loaded from memory. If necessary, a line of the NSF is flushed out to memory to make room for it. In this way, register variables are loaded into and flushed out of the register file as needed. The register file will tend to only contain live data from all concurrent contexts.

Despite being an associative structure, the Named-State Register File is unlike a conventional memory cache in several respects. First, as noted above, the allocation and deallocation of register names is under software control. Secondly, memory access patterns in a conventional cache range across the entire virtual address space. A single thread could generate a bad access pattern that would cause considerable interference in a small memory cache. In contrast, the Named-State Register File is addressed using relatively short Context IDs and offsets. Several threads worth of registers are easily accommodated in the Named-State Register File. Since the NSF is fully-associative, interference only occurs when switching between a large number of concurrent threads. We believe that in such a situation, the Named-State Register File will still outperform a segmented register file.

3.1 NSF and Memory

Figure 3 shows how the NSF would be integrated into a processor's memory hierarchy. The NSF loads register data from, and spills data to a first-level memory cache. In order to map registers into the memory space of the machine, each live context in the system is associated with a segment in memory. The context's registers are spilled into the first N locations of its segment. The memory system uses the simple lookup table *CTable* to translate a Context ID to the address its segment. Since the CID field is short (say 5 bits), this is a very fast lookup table.

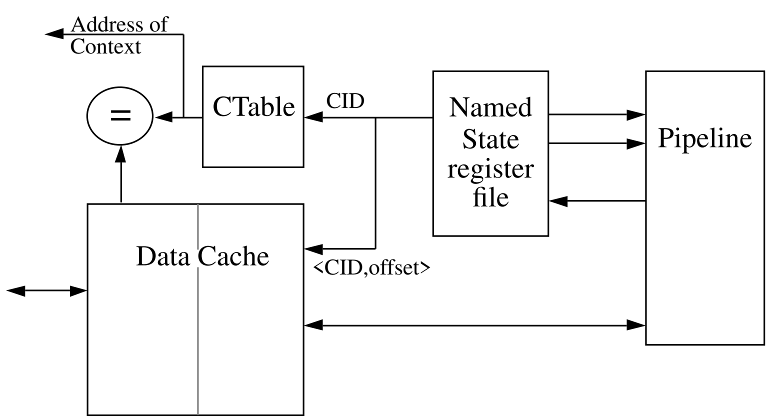

Figure 3 The Named-State Register File and memory hierarchy.

The NSF flushes data out of the register file to ensure that there is always at least one free line. This ensures that register allocation and writes always complete, and makes victim selection easier. The NSF uses a pseudo-LRU strategy [5] to pick a victim to flush out of the register file.

4 REGISTER UTILIZATION

Intuitively, a fine-grain binding of names to locations in the Named-State Register File would seem to require fewer loads and stores than a segmented register file to support the same number of active threads. The simple analysis in this section bears this out. For more detailed simulations of fine-grained parallel applications on the Named-State Register File, see [9].

Consider a processor with N words of fast register storage [1]. Assume that the number of live registers per task is an exponentially distributed random variable with mean μ. To support i tasks on a segmented register file, the registers are divided into i frames of N/i words each. With this arrangement a spill occurs in a single frame with a probability of $e^{-N/i\mu}$. So the probability of a spill in any frame is given by:

$$P(\mathrm{MTspill}) = 1 - \left(1 - e^{-N/i\mu}\right)^i . \tag{1}$$

The probability density function for the total number of live registers is obtained by convolving the density functions of the tasks together giving an Erlang distribution:

$$f_{\mathrm{liveregs}}(x) = \frac{(x/\mu)^{i-1}}{\mu(k-1)!} e^{-x/\mu} \tag{2}$$

In a Named-State Register File, a spill occurs only when the total number of live registers exceeds N. The probability of such a spill is given by integrating (2) from N to ∞ giving:

$$P(\mathrm{CCspill}) = e^{-N/\mu} \sum_{r=0}^{i-1} \frac{(N/\mu)^{i-1-r}}{(i-1-r)!} . \tag{3}$$

Figure 4 compares the spill probabilities of a segmented register file and a Named-State Register File as the number of tasks, i, is varied from one to 32

[1] In this analysis we do not consider name spilling which occurs when the number of names that can be bound to fast register storage is exhausted. Nor do we consider the impact of different replacement strategies in the NSF.

with the number of registers, N, fixed at 128, and the average number of live registers per task, μ, fixed at 8.

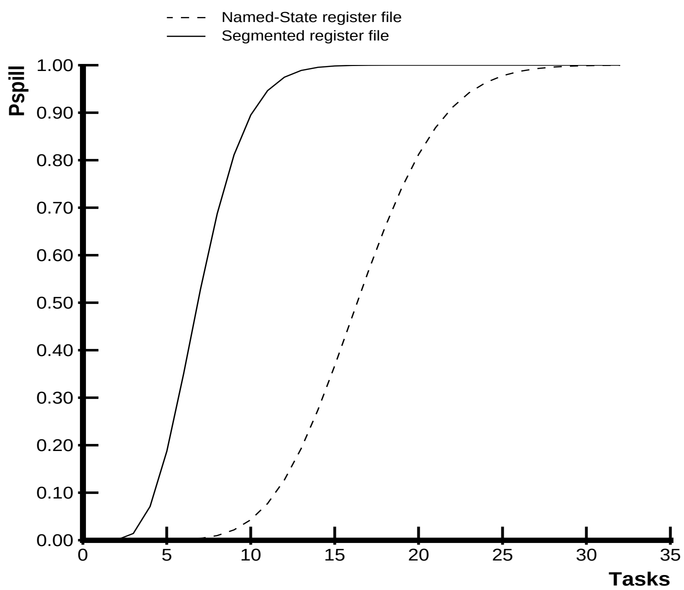

Figure 4 Probability of register spilling for a segmented register file and a Named-State register file. Total registers: $N = 128$, average live registers per task: $\mu = 8$

The figure shows that the NSF has a significantly lower probability of spills than a conventional segmented register file. With eight active tasks, half of the register storage holds live variables. At this 50% utilization level the NSF has only a 1% probability of spilling while the segmented register file has a 69% spill probability. Alternatively, the NSF could attain this spill probability while accommodating eighteen active tasks, many more than a conventional register file.

5 IMPLEMENTATION

The primary difference between a Named-State Register File and a conventional register file is in the address decoders used to drive word lines. Refer to Figure 5. While a conventional register file uses decoders hard-wired to different address bits, each row of the NSF decoder is programmable. If the data in latch *CDLATCH* matches the corresponding bit of an operand address (for example *op1adr*), the cell pulls down on the precharged *op1match#* line. The address is partitioned into offset and context ID fields. If both offset and context fields match, a word line *Op1Sel* is enabled to read a word of the register file.

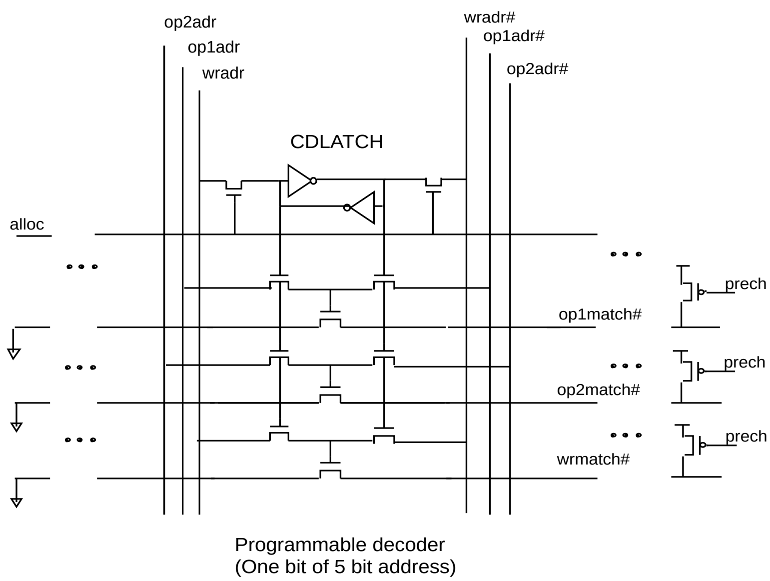

Figure 5 The Named-State Register File address decoder.

A similar precharged NAND compares the write address and operand2 address buses against *CDLATCH*. The latch is written from bit lines *wradr* and *wradr#* when the line is bound to a new variable. The valid bit is set on register allocation, to reserve this register for later reads or writes. A register may be deallocated by clearing the valid bit on read. We use a simple LRU strategy to pick which entry of the NSF to replace [5]. By matching on Context ID and offset separately, the NSF can also deallocate all registers owned by a particular context.

6 PERFORMANCE

Circuit simulation has shown that the additional latency required for this fully associative register file is a small percentage of the register access time. Figure 6

Type	Organization Height	Organization Width	Decode Address	Word Select	Data Read	Total Access	Ratio
Segmented	128	32	1.3ns	0.9ns	7.5ns	9.7ns	1
NSF	128	32	1.6ns	1.1ns	7.5ns	10.2ns	1.05
Segmented	64	64	1.6ns	1.2ns	3.5ns	6.3ns	1
NSF	64	64	1.8ns	1.4ns	3.5ns	6.7ns	1.06

Figure 6 Access time of register files in $1\mu m$ CMOS.

Type	Data Array Height	Data Array Width	Address Lines	Match Logic	Address Register	Total Width	Ratio
Segmented	128	32	48	12	—	$536\mu m$	1
NSF	128	32	50	30	10	$671\mu m$	1.25
Segmented	64	64	36	9	—	$922\mu m$	1
NSF	64	64	45	27	9	$1062\mu m$	1.15

Figure 7 Estimated register file width in $1\mu m$ CMOS.

shows the access time required for a segmented register file and a Named-State Register File of similar organizations.

The register files were implemented in a double level metal, $1\mu m$ CMOS process. Precharging and predecoding organizations were used for each design. For each proposal, we show the total register access time divided into address decode, word select, and data read times.

The first example is of a 128 line by 32 bit wide register file. The segmented register file is addressed with a 2 bit Context ID and 5 bit offset. The NSF is addressed with a 5 bit CID and a 5 bit offset. The second example shows a 64 word by 64 bit wide register file. The segmented register file is addressed with a 2 bit CID and 4 bit offset, the NSF with a 5 bit Context ID and a 4 bit offset. For both organizations, the access time of the NSF was only 6% greater than that of the segmented register file.

Figure 7 compares the area required for a NSF register file with a segmented register file. These area estimates are for a double metal, $1\mu m$ CMOS process, with a metal-1 pitch of $2.5\mu m$, and SRAM cell size of $13\mu m$ by $17\mu m$. We assume that rows of the address decoder are pitch-matched with those of the register data array. For each register file organization, we show how many metal lines run through the address decoder, the width in bits of the decoder match logic and of the NSF address register. While the NSF requires a larger address decoder than a segmented register file, that increases the total width of the register file by less than 25%.

For more detailed discussion of the cost and performance of a Named-State Register File, and the description of a prototype chip, refer to [9].

7 CONCLUSION

The Named-State Register File reduces the penalty of context switches by binding variable names to individual registers. The NSF is able to support more concurrent tasks with less register spilling than a conventional register file of the same size. This fully-associative register file can be built without significantly increasing the latency of register accesses. The NSF is only a moderately larger than conventional register files.

Acknowledgements

The research described in this paper was supported in part by the Defense Advanced Research Projects Agency under contracts N00014-88K-0738 and N00014-87K-0825 and by a National Science Foundation Presidential Young Investigator Award, grant MIP-8657531, with matching funds from General Electric Corporation, IBM Corporation and AT&T.

REFERENCES

[1] Arvind and Robert A. Iannucci. Two Fundamental Issues in Multiprocessing. Technical report, Massachusetts Institute of Technology Laboratory for Computer Science, Cambridge, Massachusetts, May 1987.

[2] R.W. Cook and M.J. Flynn. System Design of a Dynamic Microprocessor. *IEEE Transactions on Computers*, C-19(3), March 1970.

[3] William J. Dally et al. Architecture of a Message-Driven Processor. In *Proceedings of the 14th International Symposium on Computer Architecture*, pages 189–205. IEEE, June 1987.

[4] 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.

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

[6] Robert A. Iannucci. A Dataflow/von Neumann Hybrid Architecture. Technical Report TR-418, Massachusetts Institute of Technology Laboratory for Computer Science, May 1988.

[7] 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.

[8] Peter R. Nuth. Router Protocol. MIT Concurrent VLSI Architecture Group Internal Memo 23, 1990.

[9] Peter R. Nuth. The Named State Register File. Technical Report, MIT Artificial Intelligence Lab, (to appear).

[10] Amos R. Omondi. Design of a high performance instruction pipeline. *Computer Systems Science and Engineering*, 6(1):13–29, January 1991.

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

[12] 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.

[13] V. Soundararajan. *Dribble-Back Registers: A Technique for Latency Tolerance in Multiprocessors.* Bachelor's thesis, MIT, 1992.

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

[15] Carl A. Waldspurger and William E. Weihl. Register Relocation: Flexible Contexts for Multithreading. In *Proceedings of the International Conference on Computer Architecture*. IEEE, May 1993.

Index