A Mechanism for Efficient Context Switching

Peter R. Nuth William J. Dally

Artificial Intelligence Lab,
Massachusetts Institute of Technology
Cambridge, MA 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 *Context Cache*, 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 Context Cache holds more live data than a multithreaded register file, and supports more tasks without spilling to memory. Circuit simulations show that the access time of a Context Cache is only 7% greater than a conventional register file of the same size.

1 Introduction

In conventional processors, context switches are slow, taking from 50-500 instructions, because processor state must be saved and restored and the addressing environment must be switched. Multithreaded processors [1, 2, 10, 11, 3] can switch between a small set of processes very quickly but must save and restore state to switch to a process not in this set.

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 [6, 5], 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 are 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.

1.1 The Context Cache

In this paper we introduce the *Context Cache* which replaces a conventional register file. The Context Cache 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 on a context switch. The Context Cache is a component of the *Named State Processor*, which is described in detail in [7]. The NSP can switch contexts in a single cycle, without having to reload registers or bubble its pipeline.

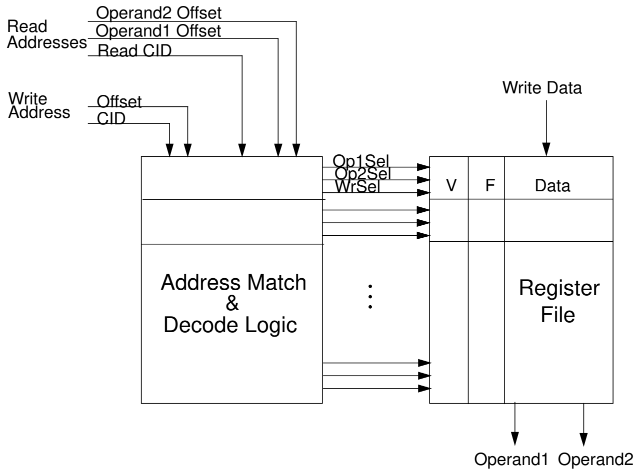

Figure 1: The structure of a Context Cache.

Figure 1 shows the organization of a context cache. The context cache retains the advantages of a register file: fast multi-port access and short register names as addresses [9] while providing a finer granularity of binding. Each location in the context cache is tagged with the *Context ID* and *offset* of the variable occupying that location. Most instructions refer to registers by offset only. The *CID* field defaults to the current context. The context cache has two read ports and a single write port, to support conventional 3 address instructions.

The *valid* (V) bit indicates whether a line is use. The *full* (F) bit is used to synchronize through a Ccache location. In the NSP, pipeline data dependencies are enforced by clearing the full bit of an instruction's destination register. In this case, the full bits are analogous to a scoreboard except that the processor context switches rather than stalls when the issue condition for an instruction is not met.

Omondi has also proposed using a cache in a multithreaded memory-accumulator machine to resolve data dependencies and rename variable accesses [8].

1.2 Multithreaded Processors

A multithreaded processor [10, 3] partitions a single large register file among a small set of *resident* processes. This organization results in inefficient use of registers and imposes a high overhead on switching to a process not in the resident set. With large register sets, sized to minimize spilling during peak usage, on average less than 50% of the registers are live at a given point in time. The coarse binding between registers and names prevents unused registers from being assigned to other processes making poor use of the fastest level of the memory hierarchy. It also makes switching to a context not in the resident set a costly operation. The entire register set of a process must be saved and another restored to switch out of the set. The fine-grain binding of names to registers in a named-state processor overcomes both of these limitations. Unused registers are free to be reassigned to other processes, and only a few registers need be loaded to switch to any process.

2 Register Utilization

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 conventional multithreaded processor, 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:

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

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:

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

In a named state processor 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:

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

Figure 2 compares the spill probabilities of a conventional multithreaded processor and a Named State Processor as the number of tasks, i, is varied from one to 32 with the number of registers, N, fixed at 128, and the average number of live registers per task, μ, fixed at 8.

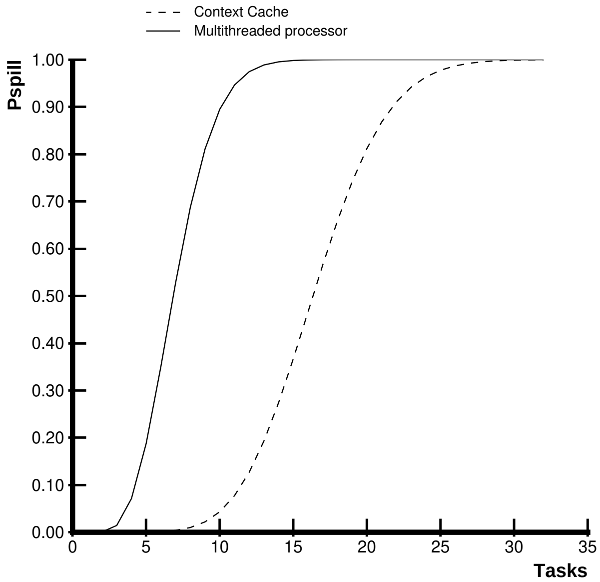

Figure 2: Probability of register spilling for a multithreaded processor and a Named State Processor. $N = 128$, $\mu = 8$

[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 context cache.

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

3 Implementation

The primary difference between a context cache and a conventional register file is in the address decoders used to drive word lines. Refer to Figure 3. While a conventional register file uses decoders hard-wired to different address bits, each row of the Ccache decoder is programmable. If the data in latch *CDLATCH* matches the corresponding bit of an operand address *op1adr*, the cell pulls down on the precharged *op1match#* line. If both offset and context address match, the word line *Op1Sel* is enabled to read the register file.

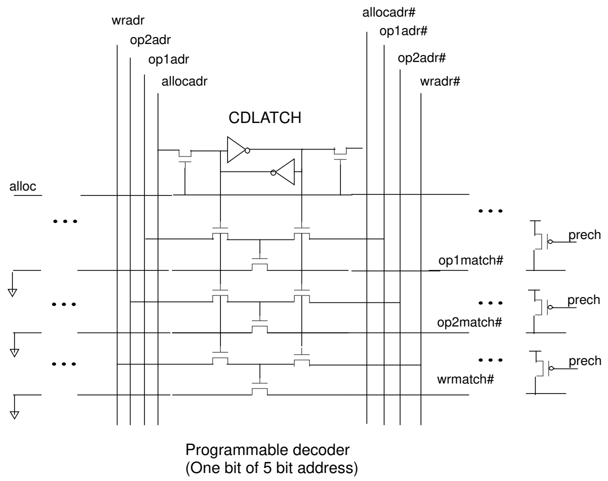

Figure 3: The Context Cache address decoder.

A similar precharged NAND compares the write address and operand2 address buses against *CDLATCH*. The latch is written from bit lines *allocadr* when the line is bound to a new variable. The valid bit is set on register allocation, to reserve this register for a later write. A register may be deallocated by clearing the valid bit on read. We use a simple LRU strategy to pick which entry of the Ccache to replace on allocate [4]. By matching on Context ID and offset separately, the Ccache can also deallocate all registers owned by a particular context.

Circuit simulation has shown that the additional latency required for this fully associative register file is a small percentage of the register access time. We simulated a 128 line by 1 word register file decoder in $1\mu m$ CMOS process. We addressed the Ccache with a 5 bit Context ID and a 5 bit offset. We compared it to a conventional register file, addressed with a 2 bit CID and 5 bit offset. The Ccache decode took $0.8ns$ longer to drive word lines on a read cycle. This is less than 7% of the $12ns$ register file access time.

The chip area required for a Ccache decoder depends on the width of the address buses that it must match against. This area can be reduced for a given size register file by increasing the cache line size. This may decrease the utilization of the Ccache, since several registers would then be swapped in and out of the cache at a time. We have written a functional simulator for the context cache to evaluate the effect that these cache parameters have on the efficiency of the register file. We are simulating the Ccache under a variety of workloads, including both sequential and parallel programs.

4 Conclusion

The Context Cache reduces the penalty of context switches by binding variable names to individual registers. Analysis has shown that the Cache is able to support more concurrent tasks with less register spilling than a conventional register file of the same size. We have shown that this fully-associative register file can be implemented without significantly increasing the latency of register accesses. We are continuing to simulate the Context Cache under a variety of workloads to determine the best balance of cost and efficiency.

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

[2] 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, Computer Society Press, June 1987.

[3] 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. Multithreaded architecture.

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

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

[6] 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.

[7] Peter R. Nuth. Parallel Processor Architecture: A Thesis Proposal. MIT VLSI Memo, 1990.

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

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

[10] 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.

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