

Another direction for future research on this topic involves investigating heuristics for determining when to dribble. Also, how effective are D-registers when there is less parallelism than assumed here? In addition, this analysis assumed all threads were created equal, but another topic of study might be a more intelligent choice of which threads to load.

## References

- [1] Anant Agarwal, Beng-Hong Lim, David Kranz, and John Kubiawicz. APRIL: A Processor Architecture for Multiprocessing. In *Proceedings 17th Annual International Symposium on Computer Architecture*, June 1990.
- [2] Wolf-Dietrich Weber and Anoop Gupta. Exploring the Benefits of Multiple Hardware Contexts in a Multiprocessor Architecture: Preliminary Results. In *Proceedings 16th Annual International Symposium on Computer Architecture*, June 1989.
- [3] Peter R. Nuth and William J. Dally. A Mechanism for Efficient Context Switching. In *ACM SIGARCH and IEEE Workshop on Multithreaded Computers*, Supercomputing '91, November 1991
- [4] Burton J. Smith. Architecture and Applications of the HEP Multiprocessor Computer System. *SPIE*, 298:241–248, 1981.
- [5] Jaswinder Pal Singh, Wolf-Dietrich Weber and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *SIGARCH Computer Architecture News*, Spring 1992
- [6] Kiyoshi Kurihara. *Performance Evaluation of Large-Scale Multiprocessors*. Technical Report, S.M. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, September 1990.
- [7] Beng-Hong Lim. *Waiting Algorithms for Synchronization in Large-Scale Multiprocessors*. Technical Report, S.M. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, January 1991.
- [8] Richard L. Sites. How to Use 1000 Registers. In *Proceedings, 1st Caltech Conf. VLSI*, California Institute of Technology. January 1979
- [9] Stephen A. Ward and Robert H. Halstead, Jr. *Computation Structures*. Cambridge, MA: MIT Press. 1990
- [10] Herbert H. J. Hum and Guang R. Gao. Efficient Support of Concurrent Threads in a Hybrid Dataflow/von Neumann Architecture. In, *Third IEEE Symposium on Parallel and Distributed Computing*, 1991
- [11] Anant Agarwal. Performance Tradeoffs in Multithreaded Processors. In, *IEEE Transactions on Parallel and Distributed Systems*, October, 1992.
- [12] Leonard Kleinrock. *Queueing Systems Volume 1: Theory*. New York, NY: John Wiley and Sons. 1975

## 7 Conclusions

Dribbling registers afford increased performance over other register file designs when synchronization run lengths are comparable to context load/unload latency and cache misses are not overly frequent. In particular, D-registers provide higher utilization than multiple register sets under average load/store conditions and all run lengths because of the load/unload latency reduction its polling behavior guarantees. The Context Cache outperforms D-registers at extremely short synchronization run lengths, but its approach is specialized toward this range; for longer run lengths, D-registers provide better processor utilization.

With D-registers, increasing the number of hardware contexts beyond three does not significantly improve performance. As little as three hardware contexts provides performance close to that expected of an infinitely large register file. In this limit a simple queuing model can accurately predict processor utilization. D-registers also alleviate cache miss latencies by increasing the probability that on a fault, there will be a runnable resident context available to which the stalled thread can switch. By reducing waits caused because no contexts are available for switching, D-registers allow fast context switching.

There are some disadvantages to D-registers when compared to multiple register sets and the Context Cache. First, it requires an instruction cache so that the bus to memory is not used on every cycle. Second, its effectiveness relies on the number of load/store operations in each process. With a large number of load/stores, fewer cycles are free for dribbling and the gains D-registers provide are decreased. Third, it requires more complicated control circuitry and a more complicated RAM cell design (to accommodate the extra ports) than multiple register sets. As a result of the extra ports and the extra capacitance that word lines would have to drive, D-register reads and writes will be slower than reads and writes in a multiple register set. D-registers are less complicated to implement than the Context Cache, however, requiring less complicated decode logic (owing to the fully-associative nature of the Context Cache). In addition, process identifier tags required by each register in the Context Cache necessitate extra tag bits.

During the early stages of the project, a VLSI implementation of the register file was undertaken to determine the feasibility of the design. This project provided separate dribble read and dribble write ports (at the onset of the project, 2 ports were to be used rather than 1). The project was a good demonstration as all vital parts of the register file were implemented, including an array of the 5-ported cell and crucial pieces of the control circuitry, namely the register sequencing logic, in order to perform dribbling. The sequencing logic can be implemented very easily. By using parallel shift registers, one can sequence through the word lines of the read ports and write ports without requiring any counters or decode logic for the individual registers. SPICE simulations indicated that D-registers are about 3-5% slower than conventional SRAM read/writes.

Our experiments indicate that the dribbling register mechanism merits further consideration in multiprocessor design. More extensive simulation studies on a multiprocessor simulator would be useful before an implementation project. Another major area for further work includes a careful study of whether the same idea can be applied to multiple-instruction-issue processors. It is likely that idle cache cycles will exist in such processors as well when the data caches are designed to support the maximum load/store bandwidth required by the datapaths.

contexts	synchronization run length	$\rho$ (D-reg)	$\rho$ (MRS)	$\rho$ (CCache)
1	30	.23	.23	.38
2	30	.28	.26	.38
3	30	.30	.28	.38
4	30	.30	.30	.38
5	30	.30	.33	.38
6	30	.32	.35	.39
1	40	.27	.27	.43
2	40	.34	.30	.43
3	40	.37	.33	.43
4	40	.38	.36	.43
5	40	.39	.39	.43
6	40	.41	.42	.43
1	50	.30	.30	.47
2	50	.38	.33	.47
3	50	.43	.37	.47
4	50	.46	.41	.48
5	50	.47	.44	.48
6	50	.50	.47	.48
1	60	.32	.32	.51
2	60	.42	.36	.51
3	60	.48	.40	.51
4	60	.52	.44	.51
5	60	.55	.48	.51
6	60	.58	.52	.51
1	75	.35	.35	.56
2	75	.46	.39	.56
3	75	.56	.44	.56
4	75	.62	.49	.56
5	75	.67	.53	.56
6	75	.71	.58	.56

Table 3: Comparison of D-reg, MRS, and Context Cache, load/store percentage = 27%, cache hit ratio = 91%, 30 threads

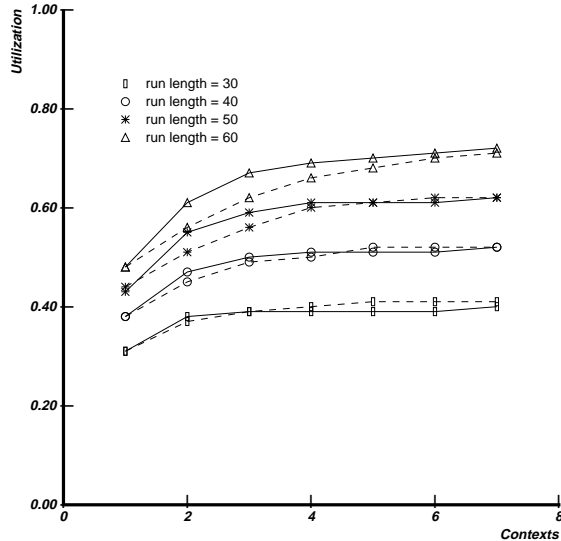


Figure 8: Queueing model vs. simulation, including load/store instructions. Dashed lines are model predictions, solid lines are simulation results

#### Context Cache.

However, at moderately small synchronization run lengths (greater than 50 cycles), and average load/store conditions[5], D-registers outperform the Context Cache. At synchronization run lengths slightly larger than the dribble overhead, D-registers provide much higher processor utilization than the Context Cache. At such run lengths, there is higher probability that more registers will require access. In such instances, the Context Cache loads more and more registers, and each process will have to remove more state from the Context Cache in order to fit its information within the register file. Because there are fewer spurious loads with larger run lengths than with smaller run lengths (i.e., when the run lengths are large, there is a high chance that all registers will be accessed, so that D-registers and MRS do not load as many registers that go unused), the Context Cache does not cut much from the average loading overhead of the multiple register set design. In fact, because each instruction has 3 register accesses, the Context Cache suffers a potential 6-cycle penalty per instruction until the working set of registers is stored in the register file. D-registers, on the other hand, still cut a great deal from the typical multiple register set overhead, and as a result provide high utilization even when run lengths are comparable to dribble time. Even when load/stores are relatively frequent (27% of total instructions executed), D-registers outperform the Context Cache at run lengths slightly higher than the dribble overhead.

Finally, when synchronization run lengths are very large, (say, greater than a few hundred cycles), processor utilization will be close to unity, and all three schemes will perform roughly the same.

Another important result of these simulations that confirm our analytical findings is that the added performance that D-registers afford does not increase significantly with more than three hardware contexts.

contexts	synchronization run length	$\rho$ (D-reg)	$\rho$ (MRS)	$\rho$ (CCache)
1	30	.25	.25	.38
2	30	.31	.27	.38
3	30	.32	.29	.38
4	30	.33	.31	.38
5	30	.33	.34	.38
6	30	.34	.36	.38
1	40	.29	.29	.43
2	40	.37	.32	.43
3	40	.41	.35	.43
4	40	.42	.38	.43
5	40	.43	.41	.43
6	40	.44	.43	.43
1	50	.33	.33	.48
2	50	.42	.36	.48
3	50	.48	.39	.48
4	50	.51	.43	.48
5	50	.52	.46	.48
6	50	.55	.49	.48
1	60	.35 (B)	.35 (A)	.51
2	60	.46	.39	.51
3	60	.54	.43	.51
4	60	.59	.47	.51
5	60	.61	.50	.51
6	60	.64	.54 (C)	.52
1	75	.38	.38	.56
2	75	.52	.43	.56
3	75	.61	.47	.56
4	75	.68	.52	.56
5	75	.72	.56	.56
6	75	.77	.60	.56

Table 2: Comparison of D-reg, MRS, and Context Cache, load/store percentage = 20%, cache hit ratio = 91%, 30 threads

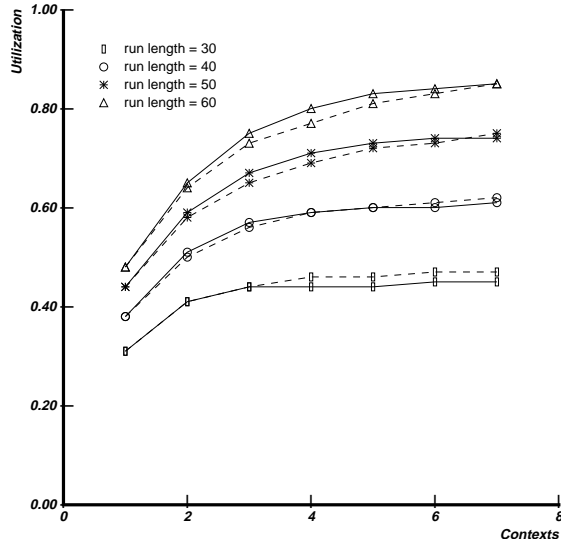


Figure 7: Queuing model vs. simulation, ignoring load/store instructions. Dashed lines are model predictions, solid lines are simulation results

cache misses would lower the utilizations a few percent, but the comparison of the model to the experimental data indicates that the model still predicts trends accurately.

## 6.2 Analysis of Results

Tables 2 and 3 display the data obtained from the sets of simulations we ran in which cache misses are enabled.  $\rho$  is processor utilization. The performance of traditional processors is indicated by the rows corresponding to contexts=1. D-registers appear to outperform multiple register sets over all ranges of grain sizes. In addition, it performs well even when load/store instructions are relatively frequent, indicating that dribbling works at reducing loading latencies even when it is not done under optimal conditions. The tables reinforce both the benefits of multiple register sets for cache misses and the benefits of dribbling for synchronization faults.

For example, consider table 2 and the entry marked A. We see that increasing the number of contexts from 1 to 6 increases utilization by almost 20%. Now consider the entry marked B. Multiple register sets with dribbling increases performance by almost 30% over the single-context operation. Entry C shows the additional benefits of dribbling over non-dribbling, as the utilization of multiple register sets without dribbling is 10% less than multiple register sets with dribbling.

At extremely small synchronization run lengths, D-registers perform slightly worse than the context cache. We can explain this behavior by noting that the context cache loads specific context information only upon demand. When synchronization run lengths are short, there is less chance for every register in a context to be accessed, and as a result, loading all registers (as the dribbler does) causes extra register loads. The Context Cache does not incur this overhead, and the elimination of excess register loads results in better fine-grain performance. Because fewer registers are touched per synchronization run length, more contexts can be resident in the

If the registers are already present, such loading is unnecessary. Synchronization faults disable processes for the appropriate wait time, and disallow any registers used by that process to be loaded or referenced until the wait time is satisfied.

On load/store operations that result in cache misses, the process is tagged as stalled until the satisfaction of the cache miss. Cache hits result in instruction executions, and do not otherwise affect performance.

The Context Cache uses an LRU replacement policy. Registers belonging to stalled contexts are considered first. Registers in the current context will never be removed.

The default cache miss latency is 40 cycles. The time required for a register load is one cycle because upon detection that a value is absent, it can be loaded directly from the cache into the ALU. If a register load causes the displacement of a register already resident in the Context Cache, the load/unload time is two cycles: one cycle corresponding to the time required to unload the old register, and one cycle corresponding to the time required to load in the new register.

## 5.4 Experiments

The register numbers in the register trace are chosen from 1-32 (assuming 32 registers/context) with a uniform probability. The parameters for the workload model are average numbers taken from real parallel applications [5, 7]. The workload model provides trace lengths for each of the threads. The simulator runs until one thread completes and then terminates the simulation.

The metric of performance is processor utilization. This is measured as the total number of useful cycles divided by the total number of cycles executed. A useful cycle consists of any cycle on which an instruction is executed. A wasted cycle is any other kind of cycle, including cycles spent waiting for remote memory accesses and synchronization faults, cycles spent loading/unloading contexts (not including cycles overlapped with useful instructions by the dribbler), or context switch cycles.

## 6 Results

This section validates the model presented in Section 4. In addition, we present processor utilization figures for each register file under equivalent workload parameters for various synchronization run lengths and load/store rates, and draw conclusions about the performance of D-registers by comparison with the other two register file architectures.

### 6.1 Validation

Figure 7 shows the agreement of the finite-register frame model (ignoring load/store instructions) with simulation. We see that the model is accurate to within 10%. The simulation is run with load/store instructions disabled since the model ignores load/stores. We see in Figure 8 that with moderate values for the percentage of load/stores (in this case, percentage of load/stores = 30), and neglecting cache misses, the complete finite-register frame model (including load/stores) is able to give good estimation of performance, and predicts the trends accurately. Including

least once before the wait time can be satisfied.

The following sections identify aspects of the simulations specific to a given register design.

## 5.1 Multiple Register Sets Simulator

In the multiple register sets (MRS) simulator, whenever a synchronization fault occurs, a context switch is attempted. A context load is initiated only if no resident contexts are runnable. If there are no runnable threads in the system, the scheduler idles until a process becomes ready to run.

On load/store instructions that result in cache misses, the scheduler attempts a context switch. If there are no free contexts available, the scheduler idles until one is ready. A context load is not performed in this case because cache miss latencies are generally shorter than load/unload times.

The default context switch overhead is 1 cycle. Although this value is smaller than usual best-case context switch time in multiprocessors (which is usually 4-5 cycles due to a pipeline flush), as long as context switch time is less than fault latencies, this number changes utilizations only slightly. It was chosen this small in order to de-emphasize the context switch overhead relative to the dribbling and loading overhead. The default average cache miss latency is 40 cycles. The time for a context load is 64 cycles. Physically, this implies that we save out 32 registers and load in 32 registers with one cycle used for each transaction.

## 5.2 Dribbling Registers Simulator

In the dribbling registers simulator, when the first synchronization fault occurs, the dribbler initiates a context load. After each dribble completes, the dribbler polls the resident contexts for any contexts stalled on synchronization faults.

On load/store instructions all dribbling is suspended for the duration of the instruction to allow the load/store to complete. A context switch is attempted on cache misses, and if no contexts are available then dribbling is suspended until the cache miss is serviced. If a ready context is available, dribbling is suspended until the context switch is complete. It is then assumed that the cache miss only disables the dribbler when the data returns (i.e., when the wait is satisfied). On a cache hit, dribbling is suspended for 1 cycle. As with multiple register sets, contexts that are stalled on cache misses are not unloaded, and the dribbler ignores such contexts when polling for contexts to remove from the register file. Other parameters are identical to those used for multiple register sets.

## 5.3 Context Cache simulator

The Context Cache simulator operates differently from each of the other simulators because there is no distinction between a context load and a context switch in terms of simulated behavior. The registers in the register file are not partitioned into separate contexts. Rather, as registers are used, they are loaded into the register file and the register is tagged with a process and register offset identification. A context switch involves updating the PC and context pointer. Registers that need to be loaded will get loaded automatically by missing in the context cache.

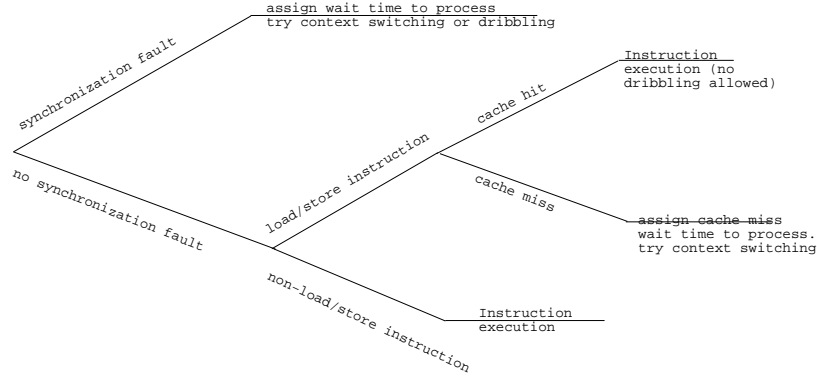


Figure 6: Decision tree for useful cycles.

The frequency of load/store instructions is also modeled as a Bernoulli process. On any instruction execution cycle that did not result in a synchronization fault, a separate coin flip is performed to determine whether that instruction is a load or a store. If it is not a load/store, sequencing proceeds through the trace. If it is a load/store, however, a final coin flip determines whether the data access results in a cache hit or miss. On a cache hit, an instruction is still executed, but on a cache miss, a wait time (exponentially distributed) is assigned to the process that missed. As with synchronization faults, this process cannot be reactivated and run until the wait time has completed. The decision tree for each cycle is displayed in Figure 6.

Load/store operations are assumed to be single-cycle (excluding cache misses), and it is also assumed that there are no cache misses when loading the register file with a context. This assumption is reasonable when contexts are loaded and unloaded multiple times. Context state stored in contiguous locations of memory is also likely to have good spatial locality. Also, the cache miss rate is assumed to be insensitive to the number of outstanding threads (which may occur, for example, if all threads are cooperating on the same problem).

Our architectural assumptions (like single-cycle load/stores and no cache misses on loading and unloading) are made to simplify the implementation of the simulators. These assumptions reflect an important aspect of this paper: it is not the precise values that we obtain through our simulations that necessarily matter. What is important is the relationships between the various factors that influence the behavior of the simulated designs, and the relative performances of the designs. By making the same base architectural assumptions for each type of register file we hope to obtain useful insights without having completely hardware-accurate functionality. For example, because the same number of threads is maintained in all our experiments, the total number of distinct data blocks placed in the cache for each of the three register file designs is the same.

The simulators require that a finite number of threads and a finite wait time be specified. However, the analyses presented within this paper assume there are always runnable threads and that synchronization wait times are infinite. With short wait times, there is a potential for threads that stall to be quickly ready again, thus eliminating, for example, the need for unloading. We wish to force the need for loads in order to evaluate D-registers against the Context Cache and multiple register sets under worst-case conditions. To force loading, wait times are made long enough so that all of the resident threads will have each been scheduled at

parameter	default value (cycles)
# of runnable threads	30
# of hardware contexts	3 (= 96 Ccache Registers)
# of registers/context	32
time between synchronization faults	50
time between load/store cache misses	50
percent of load/store instructions	26%
cache hit rate on load/store operations	91%
wait time for synchronization satisfaction	900
cache miss latency	40
register accesses/thread	65000

Table 1: Register file simulator parameters.

We solve for  $\rho$  and obtain a new utilization equation,

$$\rho = \frac{\lambda}{\mu + \lambda\alpha}. \quad (9)$$

It is easy to see that if all instructions reference memory, that is, if  $\alpha = 1$ , the processor utilization expression becomes identical to that for a single-context processor (see Equation 4).

## 5 Experimental Framework

We wrote functional simulators for several designs including D-registers, multiple register sets, and the Context Cache. The base architecture assumed is a three-ported register file in which 2 register reads and 1 register write can occur on each cycle. The simulators were trace-driven by *register traces* generated using workload models. A register trace is a list of register accesses on a cycle-by-cycle basis. An example of a register trace segment is the following:

```

1. reg1 reg2 reg3
2. reg3 reg4 reg1
3. reg4 reg2 reg1
...

```

The register traces are synthetically produced based on actual workload parameters, for ease of collection and use. These traces are then fed into the simulators. Table 1 lists the parameters for each of the simulators and their default parameter values.

Synchronization events are generated by the simulator so that thread run lengths are exponentially distributed. In order to simulate exponential distributions, a Bernoulli test (i.e., a coin flip with an appropriately biased probability of success or failure) is performed on each instruction sequencing cycle to see whether a synchronization fault is to occur. In the limit of a large number of trials, a Bernoulli process closely approximates an exponential distribution. When a synchronization fault occurs, the process that faulted is assigned a wait time (see table 1). This process cannot be reactivated and run until the wait time has elapsed.

with infinite hardware contexts and the utilization with a single-context:

$$\frac{\rho_\infty}{\rho_1} = \frac{\frac{\lambda}{\mu}}{\frac{\lambda}{\lambda+\mu}} = 1 + \frac{\lambda}{\mu}. \quad (5)$$

To maximize the gain we want to maximize  $\frac{\lambda}{\mu}$ . Given that  $\lambda \leq \mu$ , we realize that  $\frac{\lambda}{\mu}$  reaches a maximum of 1, so that

$$\max \frac{\rho_\infty}{\rho_1} = 2. \quad (6)$$

Thus, we see that the maximum improvement multiple-context dribbling registers can provide over a single-context processor when  $\lambda \leq \mu$  is a factor of two, and that the relative gain is maximum when the time between synchronization faults is equal to the latency of unloading and loading a context. This is why we do not want to use the dribbling mechanism in isolation for both cache misses and synchronization faults. Additional improvement in utilization can be earned by round-robin context switching on cache misses.

We note that the maximum value of two is a result of another important constraint imposed on dribbling registers. Our analysis assumed that the wait time for satisfaction of synchronization faults has a sufficiently high latency that it is always advantageous to force a faulted processor-resident thread to be dribbled-out in favor of a runnable, non-resident thread. If we relax this constraint by assuming that the wait time of a faulted thread might be satisfied before the thread must be completely dribbled-out, then dribbling registers can provide even more improvement in overall processor performance by switching to a processor-resident thread (see Figure 4: if thread 2 faults but its wait time is satisfied before thread 3 faults, then when thread 3 faults the processor can context switch to thread 2 and keep the dribbling of thread 1 in the background).

### 4.3 Including the Effect of Load/Store Instructions

Load/store instructions compete with the dribbler in the use of the memory (or cache) bus. We can make a simple adjustment to the above model to approximate the behavior with load/stores. Disregarding load/stores that result in local cache misses and that require remote memory accesses (we assume these happen infrequently enough for their effects to be ignored), we can think of single-cycle load/stores as simply extending the latency of a dribble. The number of useful cycles, however, does not change from the previous model, so the frequency of synchronization faults ( $\mu$ ) remains the same as before.

The increased dribble service time,  $(1/\lambda)$ , is related to the frequency of load/store cycles. Denote the fraction of instructions executed that are load/stores by  $\alpha$ . The probability that an instruction is executed on any given cycle is actually the utilization of the system,  $\rho$ . Thus, the fraction of cycles that are loads or stores is  $\rho\alpha$ , and the corresponding fraction of cycles that are not loads or stores is  $1 - \rho\alpha$ . The dribble rate in the presence of load/store operations is diminished by this factor. Denoting the effective dribble rate in the presence of loads and stores as  $\lambda_{eff}$ , we get

$$\lambda_{eff} = \lambda(1 - \rho\alpha). \quad (7)$$

Thus, for infinite contexts,

$$\rho = \frac{\lambda_{eff}}{\mu} \quad (8)$$

Let the arrival rate be  $\lambda$ , which is the probability that a dribble completes on a given cycle, and a frame is added. In other words,  $1/\lambda$  is the time between completions of dribbling a register frame. For example,  $1/\lambda$  is 64 cycles for 32 registers (assuming 1 cycle to unload state from a register and 1 cycle to load state into a register).

The utilization,  $\rho$ , of the queueing server is given by [12]

$$\rho = \frac{\lambda}{\mu}. \quad (1)$$

where,  $\rho$  has a maximum of 1. That is, if the rate of synchronization faults is less than the rate of dribbling, then there will be no idle time in the steady state. This formula applies when  $\lambda \leq \mu$ , when the register file can be arbitrarily large, i.e. when it can hold as many frames as the dribbler can pitch into it. The intuition is that if the dribbler finishes faster than the average time between synchronization faults, there will always be enough frames among which to switch.

As an example, suppose that on average, synchronization faults happen every 100 cycles ( $\mu = 0.01$ ), and that the average time to dribble a register frame is 500 cycles ( $\lambda = 0.002$ ). Then processor utilization is

$$\rho = \frac{\lambda}{\mu} = \frac{0.002}{0.01} = 0.2. \quad (2)$$

## 4.2 Finite Register Frames

A more realistic model for a finite number of register frames uses a queueing formula for bounded length queues. If  $K$  is the number of hardware contexts in the register file, we obtain from [12]

$$\rho = 1 - \frac{1 - \frac{\lambda}{\mu}}{1 - \left(\frac{\lambda}{\mu}\right)^{K+1}} \quad (3)$$

This model assumes that the dribbler is continually running. If  $1/\mu$  is larger than  $1/\lambda$ , then when the dribble completes there will not necessarily be a stalled context to remove. This can result in slightly optimistic utilization values.

In a single-context processor, context loads performed on synchronization faults cannot be overlapped with useful work. The corresponding utilization (obtained by substituting  $K = 1$  in the above formula) is,

$$\rho = \frac{\lambda}{\lambda + \mu}. \quad (4)$$

This model gives us valuable insight into the behavior of dribbling registers. When there are three or more contexts, the processor utilization is insensitive to the number of contexts. Instead, it depends on the ratio of average synchronization run length to average dribble duration (see the plots of utilization versus number of contexts in Figure 7 in Section 6.1). In addition, we see that with D-registers, 3 hardware contexts are enough to achieve performance close to that of infinite hardware contexts.

It is also instructive to compute the maximum gain afforded by the dribbling register file over a traditional single-context register file. The gain is indicated by the ratio of the utilization

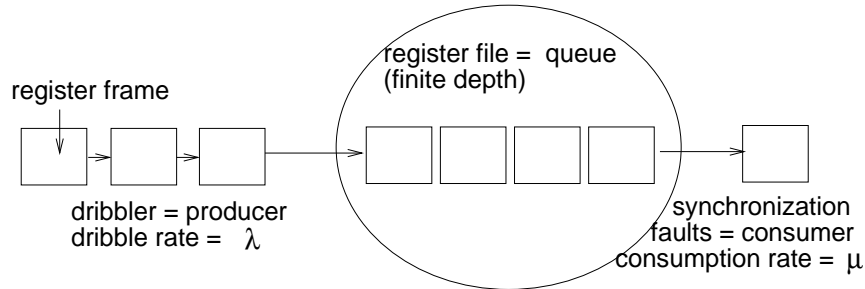


Figure 5: D-registers as a queuing server.

prediction is much less accurate and more complicated to implement: Sites suggests restoring a dribbled-out subroutine after the subroutine that was dribbled-in over it returns to the calling subroutine. This scheme can fail if a subroutine returns to its caller, but then is called again, so that the subroutine that was dribbled-in from memory must be overwritten again. In addition, it is not clear that a fixed number of register frames can provide universally effective performance for subroutines.

## 4 Mathematical Analysis of Dribbling Registers

To gain insight into factors affecting the performance of D-registers, this section develops a simple model for the utilization of a processor employing D-registers. We will then see how well this model matches the experimental data in section 5.

The development of the model proceeds in three steps: First, we derive an expression for processor utilization assuming no load/store operations occur and infinite hardware contexts exist. Second, we extend the model to include a finite number of hardware contexts. Finally, we include the effects of load/store operations. In all cases, we assume the wait time for the satisfaction of a synchronization fault is infinite, and that there are an infinite number of available threads.

### 4.1 No Load/Stores with Infinite Register Frames

In the absence of load/store operations, we can model the register file as a queuing server in which register frames (i.e., hardware contexts) are added to the register file at some rate by a dribbler, and register frames are consumed at some other rate by an execution process that causes synchronization faults. Figure 5 illustrates the situation. A register frame is added into the queuing server the moment the dribble of a frame completes. Similarly, a register frame is considered to be consumed the moment a synchronization fault occurs. The utilization of the processor is the fraction of the time the register file is busy, and is simply the utilization of the queuing server.

Let the service rate or consumption rate be  $\mu$ , which is simply the probability of a synchronization fault on a given useful cycle. (We say useful cycle because we do not have synchronization faults when, for example, servicing a cache miss). In other words,  $1/\mu$  is the average synchronization run length.

context and the loading of another runnable thread from memory. All of this loading/unloading occurs as instructions are executed out of another context. From this time on, whenever the dribbler finishes the loading of a context, it polls all of the resident contexts to see if any are stalled due to synchronization. If any are stalled, and if there are any runnable threads that can be loaded in, another dribble is initiated. If all contexts in memory are stalled, then no instructions can be executed until the dribbling is complete, unless a context's synchronization wait is satisfied during the dribble. In this case a context switch to the ready context occurs and the dribble continues (this dribbling is essentially transparent from the point of view of the processor). Note that the dribbler will always be operating if there are any stalled contexts, so that the wait for a dribble to finish (i.e., when all processor resident threads have faulted) will never be as large as the wait for an unassisted load/unload. Notice in Figure 4 that the dribbler's polling behavior requires it to dribble if a stalled context is resident and a runnable thread exists in memory.

The most obvious gain of this polling is that we reduce the cost of waiting for a context to load in the rare case that everything has stalled. Second, by keeping the register file full of ready contexts, we increase the chance that on a cache miss there will be a context to switch to and the latency of the cache miss will be tolerated (i.e., the cache miss will be unnoticeable since we will be running useful work while waiting for the miss to be satisfied). Third, we increase the chance that there will be a ready context to switch to on a synchronization fault. In fact, we can think of D-registers as a software-managed cache of process contexts in which dribbling is tantamount to prefetching on free cycles in order to avoid load/unload latencies.

### 3.3 Dribble-Back Registers

D-registers are inspired by Sites' dribble-back registers [8]. Sites outlines a method for dealing with saving and restoring state on subroutine calls. On subroutine calls, registers that are in use by the calling routine are generally needed for the subroutine, so the contents of these registers must be saved to memory to allow the subroutine to use those registers. The contents of these registers must then be restored when control flow returns to the calling routine. By adding multiple register sets, the calling function can use a set of registers separate from those that the subroutine requires, thus obviating the need for saving and restoring state.

D-registers are similar to Sites' dribble-back registers in that each provides a mechanism for latency tolerance, albeit for different purposes. In this paper we are concerned with latencies due to save/restores required because of synchronizations in multithreaded systems, whereas Sites' is concerned with latencies due to subroutine register reuse.

There is also a similarity in control system needs between Sites' idea and our idea. Our scheme invokes a polling mechanism to keep the register file full of useful work. Sites' work involves copying the contents of each register frame to memory, and making sure that any subroutine has a register frame it can write to, avoiding explicitly waiting for the completion of a save/restore; clearly this scheme could benefit by the same polling mechanism we employ in the dribbler.

One major difference between dribble-back registers and dribbling registers is that a small number of contexts can suffice to provide virtually infinite context performance. With context operations, the dribbler can more easily predict which contexts to switch to and which to unload. In fact, the dribbler can be fed information directly by the scheduler. Subroutine return

to tolerate the latency of cache misses, as in multiple register sets. Synchronization faults, on the other hand, typically suffer much higher latencies and require the faulted context to be unloaded in favor of a runnable thread that must be loaded into the register file. The dribbling register file design allows the loading and unloading of contexts to be overlapped with computation.

Lim’s [7] report on measurements of the relative latencies of cache misses and synchronization faults indicates over an order of magnitude difference: cache misses run about 40 cycles, while synchronization faults take well over 500 cycles to be satisfied. Our results show that D-registers can perform well under these types of conditions.

Given the encouraging performance of dribbling registers, it is natural to ask whether a single mechanism – namely, loading and unloading of contexts using the dribbling mechanism – can suffice to tolerate the latencies of both cache miss and synchronization faults. As demonstrated by Equation 6 in Section 4.3, the performance advantage of dribbling to overlap context load latencies in terms of improved processor utilization has an upper bound of two. Furthermore, as shown by Equation 5, the performance improvement afforded by D-registers is small when the time between loads and unloads is much smaller than the load/unload latency. Thus, to obtain higher performance, dribbling cannot be the sole mechanism employed, rather it must be used in conjunction with round-robin context switching.

Our performance comparisons assume that enough parallelism exists so that there are always threads to run. Our goal is to demonstrate that when sufficient parallelism exists, we can build a processor that can switch between multiple threads efficiently. Our analysis also ignores the impact of multithreading on cache hit rate. This assumption is reasonable in this study for three reasons. First, threads in parallel processing environments share significant portions of their code and data sets. Second, if the combined working set sizes of the threads is not significantly greater than the cache size, cache performance is not adversely impacted [11], and third, because all our experiments compare the various synchronization hiding mechanisms using the same number of threads, cache effects are expected to be the same in all cases.

### 3.2 Operation of D-Registers

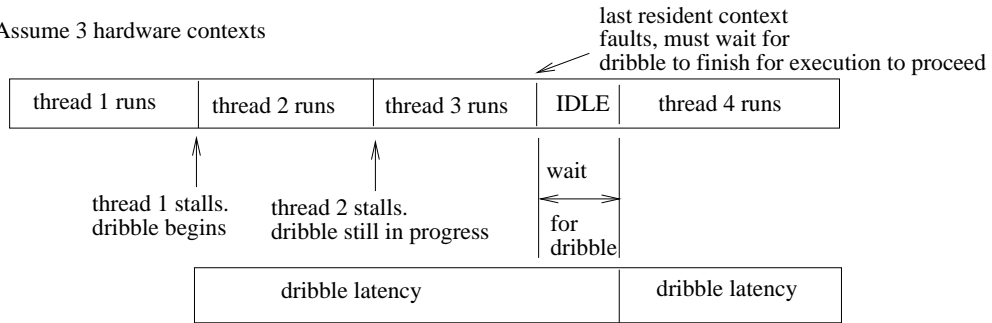
The processor state machine (henceforth termed the dribbler) must initiate two types of actions: context switches between processor-resident threads on cache misses, and dribbling unloads/loads of contexts on synchronization faults.

A simple round-robin strategy for context-switching on cache misses among processor-resident threads will be assumed in the rest of the paper. If a context switch is not possible, the processor waits out the latency of the miss. This protocol is used because cache miss latencies are much smaller than synchronization fault latencies. We would not choose to unload a thread faulted on a cache miss because the act of unloading the thread is potentially more costly than waiting out the cache miss (this is rarely true for a synchronization fault, as loading is generally on the order of 100-200 cycles [7]), and usually switching to and running other processes provides enough time for the remote request to be serviced and for that context to become runnable again.

The selection of contexts for unloading and loading works as follows. The dribbler uses a polling mechanism to attempt to keep the register file full of runnable threads. When the first synchronization fault occurs, the dribbler begins its operation, initiating the unloading of that

### D-registers

Assume 3 hardware contexts



### Multiple Register Sets, no dribbling

Assume 3 hardware contexts

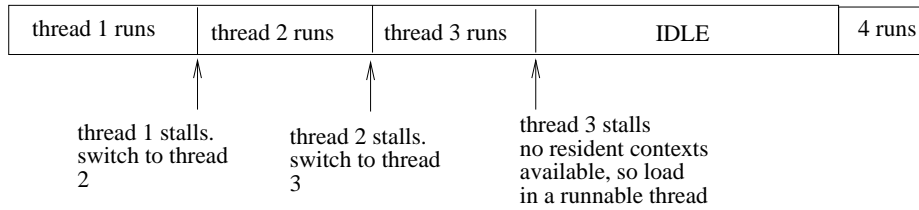


Figure 4: Comparing load/unload latency in D-registers versus multiple register sets. D-registers reduce the effective latency of loading and unloading even when thread run lengths are short.

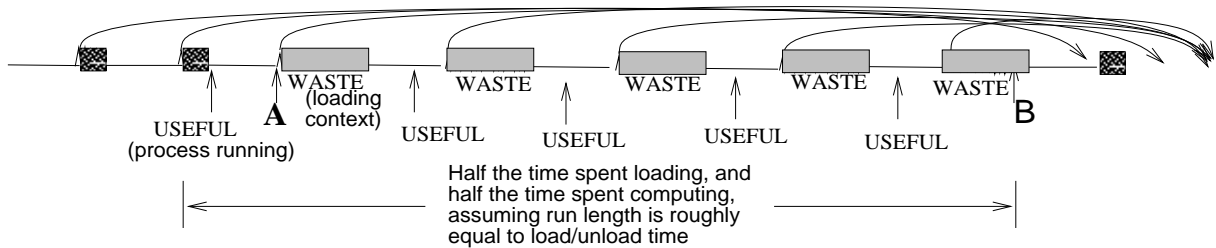


Figure 3: Cost of context loading/unloading in multithreaded designs.

pointer. All state loading/unloading is done at the granularity of a single register upon demand by instructions. The Context Cache facilitates context switching and hides latency by potentially having more runnable threads resident than in the multiple (fixed) register set register file. On cache misses and on synchronization faults, the processor context switches.

### 3 Dribbling Registers

This section presents a description of dribbling registers. First, the architectural differences between dribbling and normal multiple register set designs are discussed. Following that is a discussion of the datapaths and the control system required by D-registers.

#### 3.1 Multiple Register Sets versus Dribbling

One shortcoming of multiple register sets is that the context load/unload latency can be expensive when switching between a large number of threads is necessary. Thus, even though loading reduces waste due to synchronization idle waiting, it incurs its own cost and limits the processor efficiency, as Figure 3 illustrates. D-registers are designed to mitigate the cost of loading/unloading in multithreaded architectures.

Dribbling registers try to reduce context load/unload latency by prefetching context information from the data cache on free cache cycles. (We assume a separate instruction cache exists.) Load/stores interrupt the dribbling process as they must also access the cache, but studies have shown that the number of non-load/store instructions and hence free cache cycles can approach 75%[5] in RISC processors. Concurrency of program execution and context loading/unloading is achieved by providing an additional read/write port to the register file.

The number of free data cache cycles is critical to the operation of D-registers: if too many instructions prohibit dribbling, then the dribbles rarely complete before the synchronization run lengths expire, and the gain D-registers afford is minimal. If somehow all resident threads fault before the dribbler has replenished one of the contexts, then no useful work can occur until the dribbler has finished the context load. We shall refer to this scenario as a *forced load*. In multiple register sets with short run lengths, this case is common, and when this happens the processor is idle for the full context load interval. With D-registers, however, the processor is idle only for the amount of the dribble that could not be finished during the synchronization run lengths, as displayed in Figure 4.

The basic idea behind the dribbling registers scheme is to use round-robin context switching

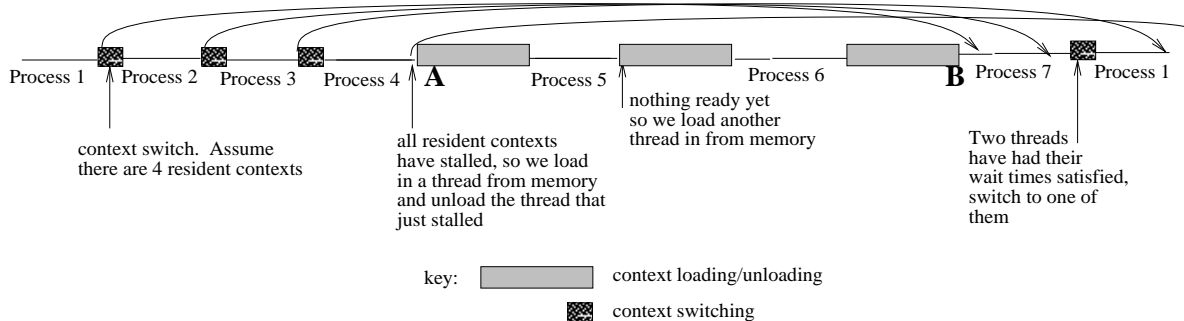


Figure 2: Multiple contexts alleviating long fault latencies by unloading stalled threads and loading runnable threads.

idle cycles until the data is fetched. There is no explicit latency tolerance mechanism that can handle both synchronization faults and cache misses.

## 2.2 Multiple Register Sets

In a multiple register set design, the register file holds a small number of hardware contexts. A context switch involves incrementing a context pointer and updating the program counter. The processor uses a two-level strategy for tolerating latencies. On cache misses, the processor context switches to tolerate the cache-miss latency. On synchronization faults, the processor switches to tolerate the synchronization latency. If all resident contexts fault due to synchronization, one of the threads is unloaded from the register file and a runnable thread is loaded in from memory. MIT's Alewife machine employs this strategy[1].

The Alewife system uses two-phase locking with a multiple register set processor. Because the processor spin-waits for a while before blocking the frequency of unloads can be reduced by the fraction of time that the synchronization is satisfied within the spin period. Lim's measurements[7] suggest that two-phase spin-then-block and always-block perform about the same in most situations because the long synchronization latencies in large-scale multiprocessors virtually always force the two-phase algorithm to block. The only exception to this observation (that is, when the two-phase did better) is when the number of processes in the system is *exactly matched* to the number of processors in the system. In the matched case, it does not make sense to unload because the extra parallelism isn't there[7].

Figure 2 illustrates how multithreading overlaps fault latencies with useful work. Notice that from point A to point B, no resident contexts are ready. However, by loading in runnable threads and unloading faulted threads, 2 more synchronization run lengths are squeezed into this otherwise wasted time.

## 2.3 Context Cache

In the Context Cache [3, 10] design, the register file does not store a complete context; instead, it stores the registers involved in the computations that occur or have occurred (treating the register file as a register cache). The pipeline allows instructions from different threads concurrently, complicating control circuitry but simplifying context switching to modifying a context

the load and unload cycles with useful work by allowing the processor to context switch to another processor-resident thread. In contrast, typical multiple-context processor designs [1] stall during loads and unloads. Because virtually infinite threads can be stored in memory, long synchronization latencies can be tolerated using this approach.

## 1.1 Contributions of this Paper

This paper proposes a register file design called dribbling registers that allows fast context switching between a few processor-resident threads to hide cache-miss latencies, but overlaps the switching latency with useful computations for a virtually infinite number of threads to hide the much longer synchronization latencies. This overlapping is accomplished by adding an extra read-write port to the register file and loading/unloading (“dribbling”) context state on free cache cycles over the extra port. Dribbling registers are inspired by Sites’ dribble-back registers for subroutine returns[8]. The differences between dribbling registers and dribble-back registers are discussed in Section 3.3.

We develop an analytical model for understanding the behavior and performance of this mechanism, as well as the results of a preliminary VLSI implementation to understand the required data paths and control mechanisms. We also describe a simulation system for register files and corresponding simulations which compare the performance of D-registers with other designs and serve to validate the model. Initial studies confirm that with dribbling registers, a few hardware contexts are sufficient to achieve processor utilizations of about 70%, even in the presence of frequent, high latency synchronization faults.

## 1.2 Organization of this Paper

The rest of this paper is organized as follows. Section 2 reviews existing register file designs. Section 3 presents a more detailed description of the architecture of dribbling registers. Section 4 describes an analytical model for the D-registers, and Section 5 describes our simulation methodology. Section 6 discusses the results of our analysis, and Section 7 presents our conclusions and outlines directions for future work.

# 2 Background

Let us first review other kinds of register file designs. These include (1) single register sets, where only one context is stored in the register file at any given time, (2) multiple register sets, where multiple hardware contexts enable support of many threads, and (3) context caches, where the register file is treated as a cache of register state.

## 2.1 Single Register Set

In a single register set design, the register file holds one hardware context. A switch of control from one thread of computation to another (e.g., due to a synchronization fault) requires the unloading of the currently executing thread and the loading of a new thread. Cache miss latencies are endured, because their latencies are short compared to context-switch times, forcing

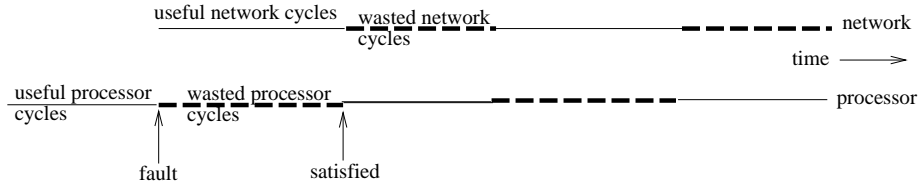


Figure 1: Processor and network idling in a single-thread system.

in parallel. The processor only operates on one thread at a time. By maintaining several processes in parallel and making process switch overhead small, multithreading allows the processor to perform useful work during remote memory accesses or on synchronization faults, rather than simply waste cycles waiting for the requests to be satisfied. Increasing the processor utilization also increases network and memory utilization: an idle processor does not make memory requests through the network.

Multithreading is simply a design that pipelines processor, network, and memory. Other ways of achieving this goal include prefetching and weak ordering. However, these do not help tolerate synchronization latencies.

Dribbling registers complement multithreading as a mechanism for latency tolerance in the presence of synchronization faults. To facilitate the discussion of D-registers, let us first clarify our terminology.

We define a *context switch* as the transfer of processor control from a processor-resident thread to another processor-resident thread in a multithreaded processor. No thread state needs to be saved into memory because the thread state remains in the register file and is not flushed; in fact, for a context switch, all that is necessary is to bump a context pointer to point to the new set of registers.

We define *loading* a thread as the action of installing the state of a thread into a hardware context on a processor, and *unloading* a thread as the complementary action of saving the processor-resident state of a thread into memory. The processor's register file contains the hardware contexts, and memory (or cache) is the storage place for the unloaded threads. Unloaded threads are maintained in memory via a task queue.

In this paper we define *synchronization run length* as the time between failed synchronization requests, and a *cache run length* as the time between cache misses.

The D-register design employs a split-level strategy for dealing with run-time latencies introduced by cache misses and synchronization faults:

- On cache misses, as the request for the data is issued into the network, the processor *context switches* among the processor-resident threads in a round-robin fashion. Because cache miss latencies are relatively short compared to the total cache run lengths of the processor-resident threads, by the time control returns to the thread that missed in the cache, the data is usually in that processor's cache and that thread can continue computation.
- However, when a thread faults due to a synchronization condition, which is a high latency operation, the faulted thread is *unloaded* into memory and a thread that is not processor-resident is *loaded* to occupy its place in the register file. The D-register design overlaps

# Dribbling Registers: A Mechanism for Reducing Context Switch Latency in Large-Scale Multiprocessors

Vijayaraghavan Soundararajan and Anant Agarwal  
Laboratory for Computer Science  
Massachusetts Institute of Technology  
Cambridge, MA 02139

November 6, 1992

## Abstract

As parallel machines grow in scale and complexity, latency tolerance of synchronization faults and remote memory accesses becomes increasingly important. One method for tolerating latency is multithreading, in which the processor rapidly context switches between a few threads on cache misses and synchronization faults. While a few threads (say, 4) are adequate [2] to completely overlap all the latency when the latencies being tolerated are short compared to the total run lengths of all the processor-resident threads, many more threads are needed if this condition is not met, which is often the case for synchronization latencies.

This paper proposes dribbling registers (D-registers) as a mechanism for fast switching between a large number of threads, and compares its performance against other methods including software context switches, multiple register sets, and context caches. The idea behind D-registers is to implement a few threads (say, 2) in hardware, but attempt to provide a larger supply of threads to switch among on synchronization faults by continually loading and unloading contexts on free cache cycles over an extra register file read-write port. Although SPICE analysis on a preliminary VLSI implementation indicates D-registers to be 3-5% slower due to the extra port, they result in higher processor utilizations than the other methods for typical workload parameters.

## 1 Introduction

As multiprocessors grow in size and in complexity, latency tolerance becomes an increasingly important issue. Two sources of high latency operations can degrade performance. First, cache misses that must be satisfied by remote memory modules suffer a communication latency. Second, failed synchronization operations incur a synchronization latency. If the processor is forced to idle during such high latency operations, as Figure 1 illustrates, machine resources including processor cycles and the network bandwidth are wasted. Tolerating these latencies is crucial to increasing machine utilization.

Multithreading facilitates latency tolerance by overlapping communication costs with useful computation, minimizing idle processor and network cycles[1, 2, 4]. Multithreaded multiprocessors use parallelism to hide latency by supporting multiple threads of computation in each processor. These threads may come from different programs or from the same program running