

# **ARCHITECTURAL AND IMPLEMENTATION TRADEOFFS IN THE DESIGN OF MULTIPLE-CONTEXT PROCESSORS**

**James Laudon  
Anoop Gupta  
Mark Horowitz**

**Technical Report: CSL-TR-92-523**

**May, 1992**

This research has been supported by DARPA contract N00039-91-C-0138.  
Authors also acknowledge support from IBM for James Laudon and an NSF  
Presidential Young Investigator Award for Anoop Gupta

# ARCHITECTURAL AND IMPLEMENTATION TRADEOFFS IN THE DESIGN OF MULTIPLE-CONTEXT PROCESSORS

James Laudon, Anoop Gupta, and Mark Horowitz

Technical Report: CSL-TR-92-523

May 1992

Computer Systems Laboratory  
Departments of Electrical Engineering and Computer Science  
Stanford University  
Stanford, California 943054055

## Abstract

Multiple-context processors have been proposed as an architectural technique to mitigate the effects of large memory latency in multiprocessors. We examine two schemes for implementing multiple-context processors. The first scheme switches between contexts only on a cache miss, while the other interleaves the contexts on a cycle-by-cycle basis. Both schemes provide the capability for a single context to fully utilize the pipeline. We show that cycle-by-cycle interleaving of contexts provides a performance advantage over switching contexts only at a cache miss. This advantage results from the context interleaving hiding pipeline dependencies and reducing the context switch cost. In addition, we show that while the implementation of the interleaved scheme is more complex, the complexity is not overwhelming. As pipelines get deeper and operate at lower percentages of peak performance, the performance advantage of the interleaved scheme is likely to justify its additional complexity.

**Key Words and Phrases:** multiple-context processors, multithreading, latency hiding, multiprocessors, pipelining

Copyright © 1992

by

James Laudon, Anoop Gupta, and Mark Horowitz

# Architectural and Implementation Tradeoffs in the Design of Multiple-Context Processors

James Laudon, Anoop Gupta, and Mark Horowitz  
Computer Systems Laboratory  
Stanford University, CA 94305

## Abstract

Multiplecontext processors have been proposed as an architectural technique to mitigate the effects of large memory latency in multiprocessors. In this paper, we examine two schemes for implementing multiple-context processors. The first scheme switches between contexts only on a cache miss, while the other interleaves the contexts on a cycle-by-cycle basis. Both schemes provide the capability for a single context to fully utilize the pipeline. We show that cycle-by-cycle interleaving of contexts provides a performance advantage over switching contexts only at a cache miss. This advantage results from the context interleaving hiding pipeline dependencies and reducing the context switch cost. In addition, we show that while the implementation of the interleaved scheme is more complex, the complexity is not overwhelming. As pipelines get deeper and operate at lower percentages of peak performance, the performance advantage of the interleaved scheme is likely to justify its additional complexity.

## 1 Introduction

The latency for memory references is a serious problem for high-performance, large-scale, shared-memory multiprocessors. The combination of large physical distances between processors and memory and the fast cycle time of processors leads to latencies ranging from many tens to hundreds of processor cycles [14, 17, 213]. For these multiprocessors to be effective on a wide range of applications, some mechanism is needed to avoid or hide this memory latency.

There are several ways to avoid this large memory latency, including caching of shared data, and restructuring of the application to maintain as much data locality as possible. While these schemes improve the amount of computation a processor performs before requiring a long-latency memory access, often applications still end up spending a large portion of their time waiting on memory references. To deal with this remaining latency, several latency tolerating schemes have been proposed, including relaxed memory consistency models, prefetching, and multiple-context processors. Recent studies [7, 12, 19] have shown that multiple-context processors are a promising way to address the problem; this paper focuses on the multiple-context solution.

There have traditionally been two approaches for building multiple-context processors. The first approach, which has been referred to as a *blocked* scheme [4], shares the processor between a number of contexts. A context utilizes all of the processor resources until it reaches a long latency operation, such as a cache miss, at which it is considered blocked. It then gives up the processor to another context, which hopefully has useful work to perform. The second approach, which we refer to as an *interleaved* scheme, switches between multiple contexts on a cycle-by-cycle basis. A context issues a single instruction, then all other active contexts issue their instructions in turn, after which the first context can issue its next instruction.

The blocked scheme has been explored primarily in the context of multiprocessors with coherent caches [2, 12, 25]. For these machines, the point at which a processor is considered to be blocked is generally at a cache or local memory miss. For example, the APRIL [2, 12] processor switches contexts when a memory

request cannot be satisfied by the cache or local memory. In addition, APRIL provides the ability to force a context switch on a failed synchronization attempt. Using these two events as switch criteria, APRIL has the ability to tolerate the latency of both memory requests and synchronization.

Interleaved processors have been proposed to solve two problems: stalls due to both pipeline dependencies and memory latency. Two representative examples are the Denelcor HEP [23] and Halstead’s MASA [8]. In the **HEP** multiprocessor, interleaved contexts were used to remove almost all pipeline stalls. A context could only issue an instruction every 8 cycles, which corresponded to the pipeline depth. Since an instruction could never encounter a pipeline dependency, no hardware or compiler resources had to be devoted to resolving pipeline hazards.<sup>1</sup> In addition, HEP provided the ability to hide the latency of a memory request by removing an instruction stream from the issue queue while the memory was being accessed. The MASA architecture similarly masks both pipeline dependencies and memory latencies by only issuing a context’s instruction after its previous instruction has completed.

While the interleaved scheme removes stalls due to pipeline dependencies, the performance of the interleaved processor suffers when there are not enough processes available to keep the pipeline full. This can be a severe drawback when the application does not provide enough parallelism. In addition, the performance of a single thread can never be better than one instruction every pipeline depth cycles. So even if there are enough processes to keep the pipeline full, it is not possible to run a single thread at near-peak processor performance. To address these limitations, we add *full pipeline interlocks* to the traditional interleaved schemes. Instructions are still only issued from a ready context; however, instructions from a single context can be issued every cycle, allowing a single process to achieve the same performance on the multiple-context processor as on a single-context processor. Since this **modified** interleaved scheme has advantages over the traditional interleaved scheme, we will only consider the modified interleaved scheme further, referring to it as simply the interleaved scheme.

In this paper, we compare the performance of the interleaved and blocked schemes. For our comparison, we use the SPLASH [22] application suite, and simulate two pipelines that roughly model those of the MIPS R3000 [10] and R4000 [16]. We show that the relative advantage of the interleaved scheme is greater for the deeper, **R4000-like** pipeline. The advantage of the interleaved scheme also improves with increasing number of contexts per processor. The performance advantage of the interleaved scheme is promising; for the SPLASH applications, the interleaved scheme provides a 21% increase in processor utilization over the blocked scheme when running on the deeper pipeline with four contexts per processor.

In addition to comparing the performance of the two schemes, we also discuss the implementation. We show that the interleaved scheme imposes more constraints on the macro-architecture of the processor. These constraints arise primarily because the processor issues a variable number of instructions between two instructions from the same context, which limits the amount of decision making which can be performed in the compiler. In addition to constraining the macro-architecture, the variable instruction issue complicates the implementation. However, the additional complexity is not overwhelming, and for processors which noticeably suffer from non-ideal pipeline utilization, the performance advantage of the interleaved scheme may warrant the additional complexity.

Before comparing the performance of the two schemes, we describe our methodology in Section 2. Then, in Section 3, we present the results of the performance comparison. We also examine the relationship between an application’s behavior and the performance gains experienced by the interleaved scheme. To address the other half of the cost/performance equation, we address implementation issues for the two schemes in Section 4. We then discuss related work in Section 5, and finally present conclusions in Section 6.

---

<sup>1</sup>An exception to this is the divide unit, which could not sustain an issue rate of one divide per cycle.

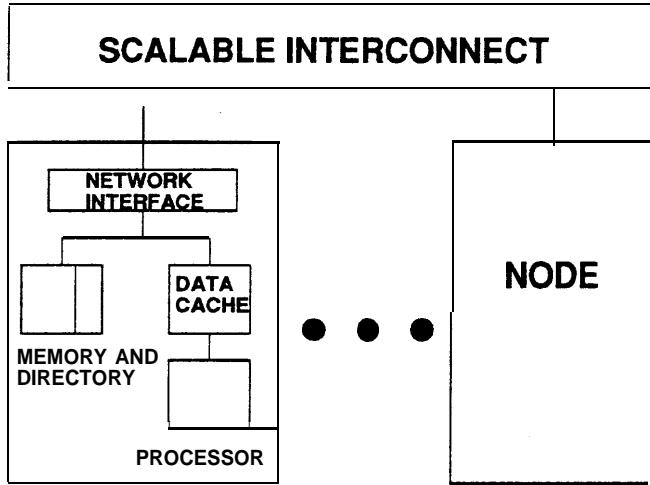


Figure 1: Base Architecture.

## 2 Evaluation Methodology

To describe our methodology, we start by describing the base architectures used in the evaluation. We then describe the simulation environment, and finally discuss the benchmark applications used in the study.

## 2.1 Base Architecture

Figure 1 shows the base multiprocessor architecture used in this study. The multiprocessor consists of a number of nodes connected together through a high-bandwidth, low-latency interconnect. Each node consists of a processor, cache, local memory, and a network interface. The caches are kept coherent using a directory-based cache coherence protocol, such as that used in the Stanford DASH multiprocessor [14].

We use two different processor architectures for our study. The pipelines of the processors are shown in Figure 2; we refer to them as *base* and *deeper*. The base pipeline is representative of commercially available RISC processors. The pipeline phases are: IF, instruction fetch; RF, instruction decode and register fetch; EX, execution; DF, data fetch; and WB, result writeback. The second pipeline is representative of more deeply pipelined RISC processors just becoming available (ours is based on the MIPS R4000 pipeline [16]). It is similar to the base pipeline, except the IF cycle is split into two cycles, IF1 and IF2, and the DF cycle is split into three cycles, DF1, DF2, and TC, where TC represents the cache tag check. The specific instruction set used is the MIPS I instruction set [10]. This instruction set is representative of load/store RISC architectures.

## 2.2 Simulation Environment

The simulation environment consists of a detailed architecture simulator, which is tightly coupled to the Tango [6] reference generator. Tango executes multiple processes on a uniprocessor, interleaving the processes to simulate a multiprocessor. Tango augments the parallel program at the assembly code level to add this interleaving. Our memory system and pipeline simulator is linked with the augmented parallel program to provide memory and instruction latencies to Tango.

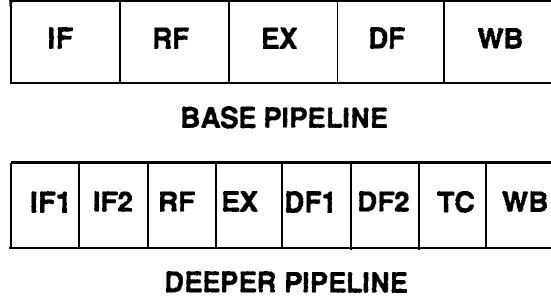


Figure 2: Structure of the Two Pipelines.

### 2.2.1 Pipeline Simulation

The simulator handles three pipeline dependencies, which account for almost all of the pipeline stalls for the applications we will study. The first dependency occurs between a load instruction and a subsequent instruction that uses the result of the load instruction. If the load data is not available for the subsequent instruction, a *hazard* has occurred. The *load delay* is the number of instructions following a load instruction that will cause a hazard. The second dependency results from operations which take more than a single cycle to complete the execution (**EX**) phase. For the pipelines we are considering, floating-point operations and integer multiply and divide are the major operations taking more than a single *EX* cycle. For these operations, a *result hazard* occurs when an instruction uses the result of an instruction before it is available. In addition, the functional units may only be able to accept a new instruction every  $I$  cycles, where  $I$  is the issue rate of the functional unit. If another instruction wishes to use the functional unit before  $I$  cycles have elapsed, an *issue hazard* results. The final pipeline dependency occurs when a control-transfer instruction, such as a branch is encountered. The address of the instruction following the branch may not be known until several cycles after the branch is issued. A common solution used by RISC processors is the *delayed branch*, where a certain number of instructions sequentially following the branch (in the branch *delay slots*), are executed regardless of the branch outcome.

The pipeline dependency parameters for our two processors are given in Table 1. For the branch delays, the delay slots are nonsquashing (the delay slot instruction executes regardless of the branch outcome). While the deeper pipeline has a single architectural delay slot, the pipeline is unable to generate a branch target until three cycles after the branch. Thus for the deeper pipeline, all taken branches will have an additional two cycle penalty. There are separate floating-point units for addition, multiplication, and division, and these units can operate in parallel. In addition, there is a single multiply/divide functional unit, which can operate in parallel with the floating-point units. The integer multiply/divide unit and the floating-point units are not pipelined, so the issue latency and the result latency are equivalent. Operation on single-precision floating-point is slightly faster than double-precision, and the single-precision numbers are shown in parenthesis in the table.

Each pipeline dependency typically stalls the processor for a much shorter duration than a remote memory access. However, pipeline stalls are much more frequent than remote memory accesses. Since the total number of pipeline stalls depends heavily on how well the compiler can hide these dependencies, it is important that the code be optimized with respect to the pipelines that are being simulated. In the **MIPS** compiling environment, pipeline scheduling is done by the assembler. We insure that the instrumented code is properly scheduled by replacing the standard **MIPS** assembler with a **parameterizable** assembler developed at Stanford.

### 2.2.2 Multiple Context Simulation

The architecture simulator handles the multiplexing of multiple instruction streams on the multiple-context processor. For the blocked scheme, the simulator switches contexts on read misses. The cost of a context switch

Table 1: Pipeline Simulation Parameters.

	Base Pipeline	Deeper Pipeline
Load Delay	1 pclock	2 pclocks
Branch Delay	1 pclock	1 pclock
Integer Multiply	12 pclocks	12 pclocks
Integer Divide	35 pclocks	76 pclocks
Floating-point Add	2 (2) pclocks	4 (4) pclocks
Floating-point Multiply	5 (4) pclocks	8 (7) pclocks
Floating-point Divide	19 (12) pclocks	36 (23) pclocks

for this scheme is equivalent to the pipeline depth (5 cycles for the base pipeline, 8 for the deeper pipeline), since we assume the instructions already in the pipeline must be squashed. For the interleaved scheme, the simulator cycles through the available contexts in a round-robin fashion. When a context encounters a read miss, the context is marked unavailable until the read data returns. While the cost to switch between contexts for the interleaved scheme is zero, there is an indirect cost associated with a context encountering a read miss. This cost is equal to the number of instructions from this context that are already in the pipeline, as they must be squashed. We will refer to this cost as the context switch cost for the interleaved scheme.

The simulator handles synchronization events in a manner analogous to memory operations. A processor context switches (and marks a context unavailable) when blocking on a synchronization operation. When the synchronization event successfully completes, the context is marked available. The lone exception to simulating synchronization operations in this manner occurs for applications which perform explicit spinning on shared memory locations. Spin-waiting poses an interesting problem for multiple contexts. For the blocked scheme, deadlock is possible if one context repeatedly spins on an event, never missing in the cache, while the context which is to produce the event is inactive on the same processor. To solve this problem for our simulations, we add an explicit context switch instruction to each spin loop iteration. This is a reasonable solution, given the programmer knows where the spinning is occurring. However, for cases where the programmer may not catch all the spinning in an application, the processor should have some form of watchdog timer that forces a context switch after a certain number of instructions by the same context.

In contrast, the interleaved scheme does not suffer from this potential deadlock situation. Even though the context spinning may be on the same processor as the context that will produce the event, both contexts are interleaved, providing forward progress for the event-producing context. However, the spinning context does take processor cycles that would be better used by the contexts which are not spinning. This cycle stealing by a spinning context from the event-producing context can increase the time spent spinning by a program. For our simulations, we fix this problem by giving priority to instructions from non-spinning contexts. To enable a fair comparison, for the blocked scheme we also give priority to switching to a context which is not spinning.

### 2.2.3 Memory System Simulation

The memory simulator consists of a detailed data cache model connected to a simplified network and memory system.<sup>2</sup> The cache is lockup-free [11, 20, 24], with the simulator modeling all the internal states of the cache. The parameters for the cache are given in Table 2. We have assumed that a write buffer and release consistency [5] are being used, resulting in the cost associated with writes being very low [7]. We model this by using a cache with a write-through policy for stores, and assume writes complete in the network immediately. Therefore the processor will never stall on a write. While connecting a write-through cache directly to the network is not a reasonable design decision for an actual multiprocessor implementation, it simplifies the

<sup>2</sup>Instruction references are not sent to the cache simulator and are implicitly assumed to hit in the cache.

Table 2: Cache Parameters.

<b>Size</b>	<b>64 Kbytes</b>
<b>Associativity</b>	<b>Direct Mapped</b>
<b>Line Size</b>	<b>32 bytes</b>
<b>write Policy</b>	<b>Write-through, no allocation</b>
<b>Pending Read Limit</b>	<b>1 per context</b>
<b>write Occupancy</b>	<b>1 pclock</b>
<b>Read Occupancy</b>	<b>1 pclock</b>
<b>Invalidate Occupancy</b>	<b>2 pclocks</b>
<b>Cache Fill Occupancy</b>	<b>4 pclocks</b>

Table 3: Memory Latencies.

<b>Hit in Primary Cache</b>	<b>1 pclock</b>
<b>Reply from Local Node</b>	<b>25-45 pclocks (uniform distribution)</b>
<b>Reply from Remote Node</b>	<b>75-135 pclocks (uniform distribution)</b>

simulator and reasonably models the low cost associated with writes under our assumptions. Read operations have the latencies shown in Table 3. We model cache contention, which is added to these base latencies. While cache contention is modeled, the network and memories are contentionless. This should not greatly affect our results, as the network and memory contention should roughly increase the memory latency equally for both the blocked and interleaved schemes. Simplifying the network and memory system allows us to simulate larger problems, while still providing a sufficient model of the memory system behavior.

### 2.3 Applications

To evaluate the two architectural alternatives, we use the SPLASH suite, a set of parallel scientific and engineering applications written in either C or FORTRAN. The applications use the Argonne National Laboratory macro package [15] for synchronization and sharing. We briefly describe the six SPLASH applications below; a more detailed discussion of the applications can be found in the SPLASH technical report [22].

Water is an example of an N-body application. It evaluates forces and potentials in a system of water molecules in the liquid state. Our Water simulations were for 288 molecules. We simulated Water for two time steps, while only recording the results for the second time step. Since Water does almost identical work on each time step, gathering statistics from a single time step suffices. The second step is recorded to avoid undesirable cache effects caused by the initialization of data by a single processor.

Ocean studies the role of eddy and boundary currents in influencing large-scale ocean movements. The program sets up a set of spatial partial differential equations. These equations are transformed into difference equations which are solved on two-dimensional fixed-size grids, statically partitioned between the processors. We simulated Ocean on a 98x98 grid over six time steps. Again, the first time step is not recorded to avoid initialization effects.

**MP3D** is a three-dimensional particle-based simulator for rarefied air flow. The overall computation consists of evaluating the positions and velocities of the particles over several time steps, with barriers between time steps. For our simulations, we simulated **MP3D** for four time steps, with 50,000 particles, and a **14x24x7** space array. Again, the first time step is not recorded.

**LocusRoute** is a VLSI standard cell router. It attempts to minimize circuit area by routing wires through

regions (routing cells) which have few other wires running through them. It does this by calculating a cost function for each route being considered for a wire, and selects the lowest cost route. The cost function is based on the number of wires already routed through the routing cells selected by the route. We routed *Primary1 grin*, a circuit consisting of 1266 wires and 481x18 routing cell array, for two iterations using geographical wire partitioning. We recorded statistics from the spawning of the processes, stopping just before the final route quality computation.

PTHOR is a parallel, distributed-time, event-driven, circuit simulator. We simulated the *risc* circuit for 5,000 time ticks, with a 200 tick clock cycle. The *risc* circuit consists of 5,060 elements, where an element corresponds to a basic functional block, such as an AND gate or latch. We started recording statistics after the input files have been read and the simulator initialized, stopping just before the final simulation results were printed.

Cholesky performs a parallel Cholesky factorization of a sparse positive definite matrix. We simulated the factorization of the matrix *BCSSTK14*, which is a 1806x1806 matrix, with 30,824 non-zero elements. We recorded statistics only for the parallel numeric factorization.

The SPLASH suite contains complete applications, not kernels. Four of the applications (Ocean, Water, **MP3D**, and Cholesky) are scientific computations and two (**LocusRoute** and PTHOR) are from the area of computer-aided design. While no application suite is ideal for all architectural evaluations, the SPLASH suite does cover a range of problem domains and algorithms, and constitutes a reasonable workload for a shared-memory multiprocessor.

## 3 Performance Results

While the previous section described the base architecture, we still have a number of degrees of freedom in the design of a multiple context machine, such as the number of contexts supported in hardware. We start this section by discussing the hardware configurations we simulated. We then examine the results of the performance evaluation, using three of the SPLASH applications to illustrate the major trends. Using these applications, we also examine the relationship of the application’s behavior and the performance gains experienced by the interleaved scheme. Finally, we summarize the section.

### 3.1 Simulation Configurations

One of the important multiple-context parameters is the number of contexts supported in the **hardware**. For our simulations, we vary this number of contexts between one and eight. As we will see from the simulation results, eight contexts is usually enough to cover almost all the memory latency.

For all the simulations, a fixed number of application *processes are used*. We used 16 processes for our simulations. Previous studies [7, 12,251] have used a fixed number of *processors in* evaluating the effectiveness of multiple contexts. With a fixed number of processors, the number of application processes increases as the number of contexts (processes) per processor is increased. Since the number of application processes is increasing, lack of application parallelism and varying process workloads affect the results. Using a fixed number of processes does not have these problems, as the amount of work done by each context is roughly the same across the different numbers of contexts per processor. However, using a fixed number of processes affects the run-time behavior by reducing the amount of interprocessor communication needed by the program, since processes which would have previously needed to communicate are now sharing a cache. While both methods have their advantages and disadvantages, we are more interested in avoiding the effects of limited application parallelism, and consequently simulate a fixed number of processes.

We present the simulation results of each application in two formats. To examine general trends, we show a set of four graphs, presenting the processor utilization breakdown across the two pipelines and two multiple-context schemes. In these graphs, the total time spent by the processor executing the program is broken into several categories. These categories are (a) BUSY, time spent executing instructions, (b) SPIN, time spent spin-waiting on an event, (c) PIPESTALL, stalls due to load hazards, branch delay slots without useful work, and functional unit result and issue hazards, (d) SWITCH, context switch cost, (e) CACHESTALL, contention for the lockup-free cache, and (f) IDLE, unoverlapped memory latency. To allow for a more detailed examination of the results, we also provide a table containing the processor utilization breakdown and application miss rate. In the tables, we have broken the PIPESTALL category of the graphs into its three components.

### 3.2 Simulation Results

To help illustrate the major trends, we have selected three of the SPLASH applications: **MP3D**, Water, and PTHOR. These three were selected as a representative sample of the SPLASH applications we simulated. The results for the other applications are given in Appendix A. Looking at the processor utilization graphs for our three selected applications in Figures 3-5, we observe several general trends.

First, the addition of multiple contexts is quite effective in increasing the processor utilization. For example, for **MP3D** running on the deeper pipeline, the processor utilization improves from under 25% to over 60% for the eight-context blocked scheme, while improving to nearly 80% for the interleaved scheme. **MP3D** operates on a large number of particles each time step, performing a relatively short computation on each particle. Since the particle data is too large to fit in the cache, there is little reuse between time steps, resulting in the high miss rate of 12.0%. This high miss rate explains the low single-context processor utilization for **MP3D**. Water, on the other hand, has very good cache behavior (a 0.7% miss rate) and consequently has a much higher single-context utilization (approximately 45%) for the deeper pipeline. Eight-context processor utilization improves to 55% for the blocked scheme and to over 70% for the interleaved scheme. For PTHOR, the utilization increases from 16% to over 45% for the blocked scheme and to nearly 65% for the interleaved scheme. While the increases in processor utilization vary from application to application, all applications benefit from the addition of multiple contexts,

Looking next at the differences in processor utilization between the two pipelines, we observe that, as expected, for the deeper pipeline the applications spend a greater percentage of their time in context switches and pipeline stalls than for the base pipeline. The percentage of time spent in pipeline stalls for the deeper pipeline can be fairly substantial. Using the more detailed numbers in Tables 4-6, we observe that PTHOR spends 0.66 cycles in pipeline stalls for every busy cycle, while for Water the ratio is 0.72 cycles stalled for every busy cycle. **MP3D** has the lowest ratio of cycles stalled to cycles busy, 0.38. For Water and **MP3D**, scientific codes, (floating-point) functional unit hazards are the major contributor to the pipeline stall time. PTHOR, a CAD application, does not contain significant amounts of floating-point computation, and most of its pipeline stall is due to load hazards. For all the SPLASH scientific applications, floating-point hazards are the major contributor to stalls, while for the engineering applications, load or control-transfer hazards are the major contributor.

Finally, we turn to the differences between the two multiple-context schemes. From the graphs, we observe that the interleaved scheme is indeed benefiting from reduced pipeline stalls and lowered context switch costs, although the benefits are being slightly offset by increased cache contention. We first examine in more detail the difference in context switch overhead between the two schemes. For the interleaved scheme, the percentage of total time spent context switching decreases as the number of contexts per processor increases. There are three factors at play in determining the percentage of time the interleaved scheme spends context switching. First, the total number of context switches depends directly on the read miss rate. For all the

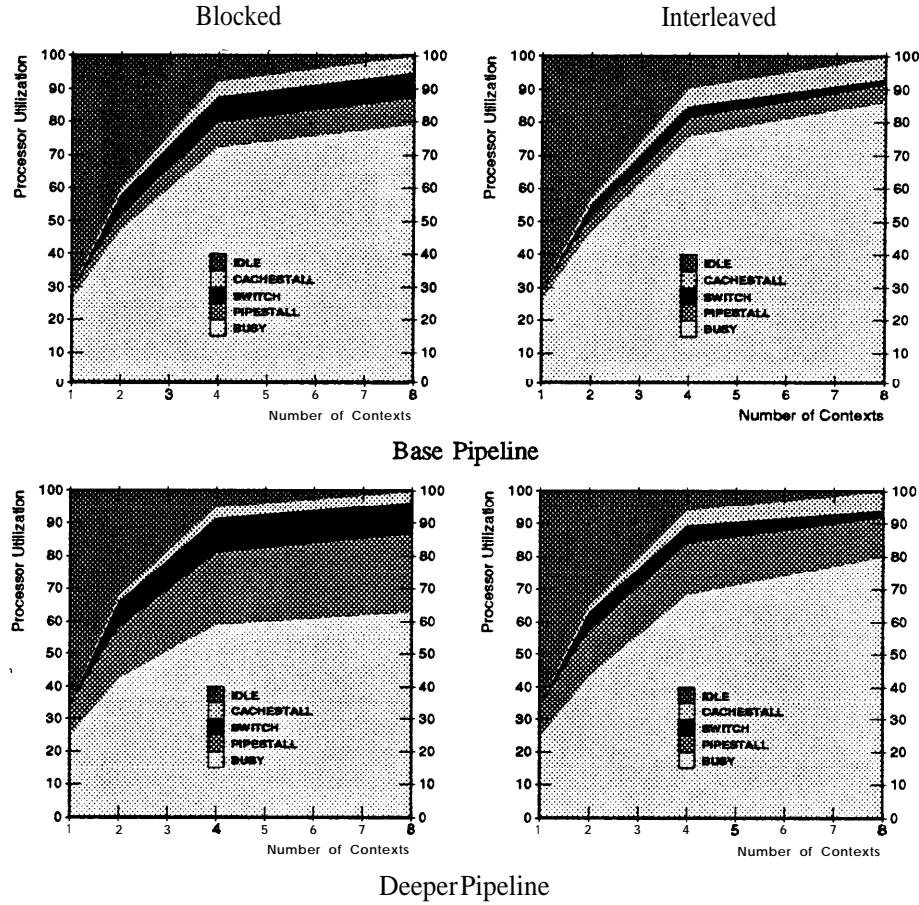


Figure 3: Processor Utilization — MP3D.

Table 4: Detailed MP3D Results.

	Base Pipeline							
	Single Context		Two Contexts		Four Contexts		Eight Contexts	
	Block	Inter	Block	Inter	Block	Inter	Block	Inter
Idle	70.8	70.8	40.5	43.3	8.0	9.8	0.4	0.3
Cache Contention	0.5	0.5	1.9	1.9	45	5.3	4.8	6.7
Switch Overhead	0.0	0.0	5.7	3.7	8.1	3.8	7.7	1.8
Load Stall	0.0	0.0	0.1	0.1	0.1	0.0	0.1	0.0
Branch Stall	1.1	1.1	1.9	1.9	2.9	3.1	3.2	3.5
Execution Stall	1.7	1.7	2.7	2.3	4.2	24	4.7	1.6
Busy	25.8	25.8	47.2	46.9	72.2	75.6	79.1	86.1
<b>Read Miss Rate</b>	<b>120</b>	<b>12.0</b>	<b>11.6</b>	<b>11.6</b>	<b>10.7</b>	<b>10.7</b>	<b>9.3</b>	<b>9.2</b>
Deeper Pipeline								
	Single Context		Two Contexts		Four Contexts		Eight Contexts	
	Block	Inter	Block	Inter	Block	Inter	Block	Inter
	65.6	65.6	32.4	35.3	5.4	6.2	0.5	0.3
Idle	0.5	0.5	1.6	1.7	3.2	4.4	3.2	5.8
Cache Contention	0.0	0.0	8.1	6.1	10.5	5.5	9.8	2.3
Switch Overhead	1.2	1.2	2.0	1.4	2.7	0.8	2.9	0.0
Load Stall	2.8	2.8	4.8	3.9	6.7	3.8	7.1	3.3
Branch Stall	5.6	5.6	9.0	8.4	12.6	11.1	13.6	8.4
Execution Stall	24.3	24.3	42.1	43.2	58.8	68.1	62.8	79.9
<b>Read Miss Rate</b>	<b>12.0</b>	<b>12.0</b>	<b>11.6</b>	<b>11.6</b>	<b>10.7</b>	<b>10.7</b>	<b>9.3</b>	<b>9.2</b>

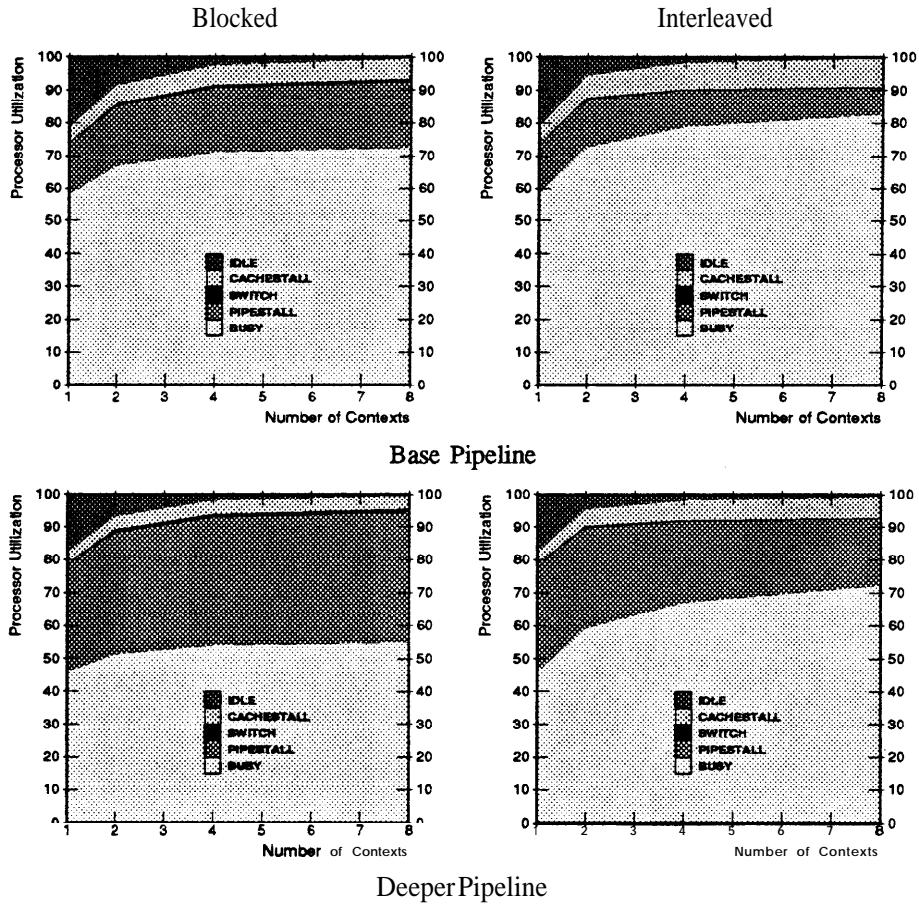


Figure 4: Processor Utilization — Water.

Table 5: Detailed Water Results.

		Base Pipeline							
		Single Context		Two Contexts		Four Contexts		Eight Contexts	
		Block	Inter	Block	Inter	Block	Inter	Block	Inter
<b>Idle</b>		21.5	21.5	8.6	5.6	2.6	1.9	0.6	0.4
<b>Cache Contention</b>		4.6	4.6	55	6.8	5.9	8.0	6.1	8.6
<b>Switch Overhead</b>		0.0	0.0	1.0	0.5	1.0	0.3	1.0	0.3
<b>Load stall</b>		1.1	1.1	1.3	0.4	1.4	0.0	1.4	0.0
<b>Branch Stall</b>		2.8	2.8	3.2	35	3.4	3.8	3.5	4.0
<b>Execution Stall</b>		12.0	120	13.7	10.5	14.6	7.2	15.0	4.1
<b>Busy</b>		57.9	57.9	66.7	72.6	71.0	78.8	72.4	82.6
<b>Read Miss Rate</b>		0.7	0.7	0.8	0.7	0.7	0.7	0.7	0.8
<hr/>									
		Deeper Pipeline							
		Single Context		Two Contexts		Four Contexts		Eight Contexts	
		Block	Inter	Block	Inter	Block	Inter	Block	Inter
<b>Idle</b>		17.9	17.9	7.0	4.8	2.1	2.1	0.4	0.8
<b>Cache Contention</b>		3.4	3.4	3.9	5.0	4.1	6.0	4.2	6.6
<b>Switch Overhead</b>		0.0	0.0	1.2	0.7	1.1	0.4	1.2	0.2
<b>Load Stall</b>		25	25	2.8	1.8	3.0	0.3	3.0	0.0
<b>Branch Stall</b>		8.0	8.0	8.9	6.9	9.4	5.0	95	4.8
<b>Execution Stall</b>		22.5	225	25.0	21.9	26.4	19.3	26.9	15.4
<b>Busy</b>		45.7	45.7	51.2	58.9	53.9	66.9	54.8	72.1
<b>Read Mii Rate</b>		0.7	0.7	0.8	0.7	0.7	0.7	0.7	0.8

applications the read miss rate stays relatively constant across the different number of contexts, and has a minor effect on the context switch overhead. Second, as the processor spends less time idle, the relative percentage of time spent in the other categories increases. Finally, as more contexts are added, the cost of each context switch decreases. It is this third factor which dominates for the interleaved scheme, causing the context switch time to decrease as **more** contexts **are** added. For the blocked scheme, the percentage of time spent in context switching is **influenced** only by the miss rate and decreasing idle time. Since the miss rate is relatively constant, the decreasing idle time dominates, causing the relative percentage of time spent context switching to increase as more contexts **are** added.

To examine the difference in pipeline **stall** time between the two schemes, we need to look at the more detailed numbers in Tables 4-6. From these tables, we can see that the load, branch, and functional unit issue and result stalls tend to decrease for the interleaved scheme with increasing numbers of contexts. In contrast, for the blocked scheme the amount of time spent in pipeline stalls remains the same relative to the processor busy time. For example, for Water running on the deeper pipeline, regardless of the number of contexts, there is 0.49 cycles of functional unit issue and result stalls for each busy cycle. Examining these three pipeline effects in more detail, we see that the interleaved scheme does very well at reducing the load stalls to near zero as the number of contexts is increased. However, while the branch and functional unit stalls are reduced, they do not get removed completely. For branch stalls, this is because the penalty associated with an unfilled branch delay slot cannot be hidden by the interleaved scheme. For functional unit stalls, the reduction is limited by separate contexts encountering an issue hazard while contending for the same functional unit. The frequency and duration of these **conflicts** depends on the application characteristics and the context interleaving.

As a final point, we notice that the performance of the interleaved scheme is less dependent on the underlying pipeline than the blocked scheme. As more contexts are added, the processor utilization difference between the two pipelines narrows. This is an important advantage of the interleaved scheme, as the trend towards exploiting intra-stream, instruction-level parallelism is leading towards pipelines which operate less efficiently relative to their peak rate.

### 3.3 Results Summary

As we have seen, the interleaved does indeed benefit from reduced pipeline delays and context switch costs. The **performance** advantage of the interleaved scheme varies depending on the application, and can be substantial for certain applications. Table 7 summarizes the results for all the SPLASH applications. The benefits of the interleaved scheme are greater for deeper pipelines, with their higher pipeline dependency costs. These results show the performance advantage of the interleaved scheme to be promising; what is now needed is an investigation into its implementation cost. To address the cost, we need to look in more detail at the implementation issues involved in building multiple-context processors using the two schemes.

## 4 Implementation Issues

The complexity associated with building any multiple-context processor is manifested primarily in three requirements. The first requirement is that the memory system be able to handle multiple outstanding memory operations, implying the need for a lockup-free cache, [11]. In addition, **resources** shared between contexts, such as the instruction cache and buffers, the data cache, and the TLB may need to be optimized to handle the multiple working sets. While design of lockup-free caches and optimization of shared resources is beyond the scope of this paper, they are important issues, and need to be addressed in the design of a multiple-context processor. The second requirement is that the user-visible state be replicated to have a unique copy per context. This user-visible state includes the register **file(s)**, processor status word program counter, and any other state

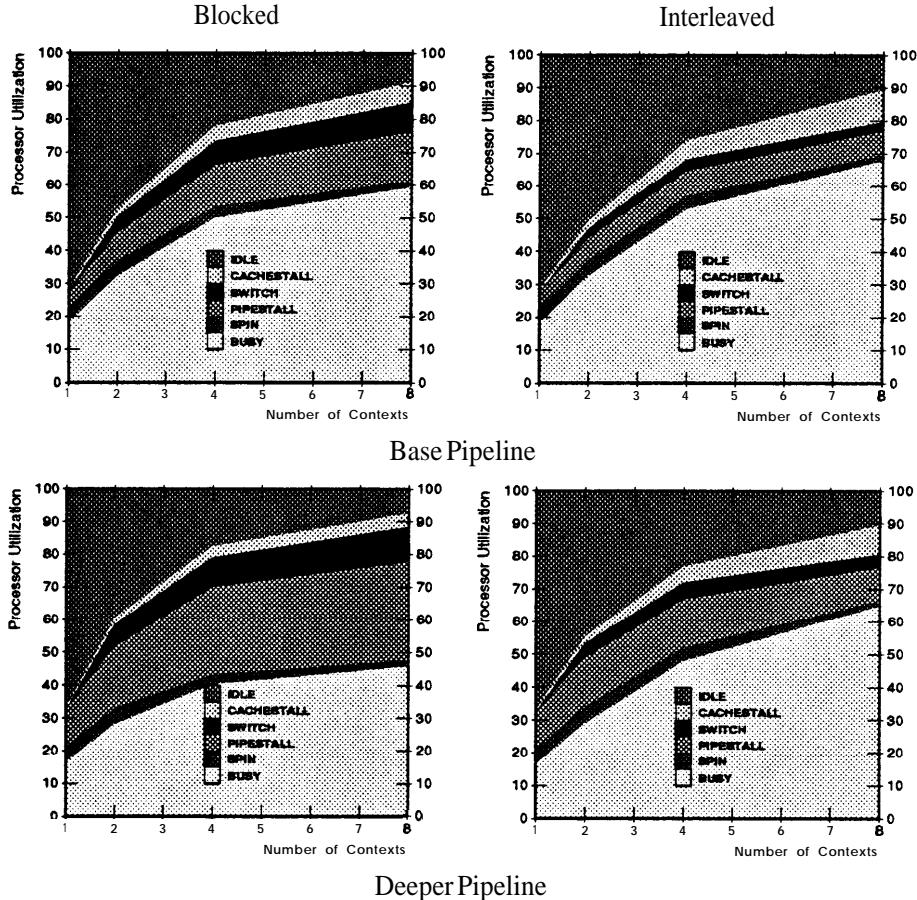


Figure 5: Processor Utilization — PTHOR.

Table 6: Detailed PTHOR Results.

		Base Pipeline							
		Single Context		Two Contexts		Four Contexts		Eight Contexts	
		Block	Inter	Block	Inter	Block	Inter	Block	Inter
Idle		71.6	71.6	47.6	50.7	220	26.0	8.8	10.7
Cache Contention		1.0	1.0	2.2	2.4	4.6	5.8	6.2	9.7
Switch Overhead		0.0	0.0	5.1	2.9	7.4	3.7	8.9	29
Load Stall		2.7	2.7	4.7	3.1	7.3	2.2	8.7	0.7
Branch Stall		1.8	1.8	3.2	3.2	4.9	5.2	5.7	6.5
Execution Stall		0.2	0.2	0.3	0.3	0.5	0.3	0.6	0.1
Spin		4.6	4.6	4.7	4.9	3.4	4.1	1.9	1.7
Busy		18.1	18.1	32.1	32.5	49.8	52.8	59.2	67.7
Read Mii Rate		7.2	7.2	6.9	7.3	6.8	7.5	7.2	8.4
Deeper Pipeline									
		Single Context		Two Contexts		Four Contexts		Eight Contexts	
		Block	Inter	Block	Inter	Block	Inter	Block	Inter
Idle		66.9	66.9	39.8	44.7	17.6	23.1	7.5	10.5
Cache Contention		0.8	0.8	1.8	2.0	3.4	4.9	4.2	8.8
Switch Overhead		0.0	0.0	6.9	4.5	9.2	5.3	10.8	4.2
Load Stall		6.3	6.3	10.5	7.8	15.1	6.3	17.1	1.9
Branch Stall		4.5	4.5	7.2	6.2	10.3	7.4	11.7	7.6
Execution Stall		0.5	0.5	0.8	0.8	1.2	1.1	1.4	0.6
Spin		4.7	4.7	5.0	5.2	3.0	4.2	1.7	1.6
Busy		16.3	16.3	27.9	28.8	40.1	47.8	45.7	64.7
Read Miss Rate		7.2	7.2	6.9	7.3	6.8	7.5	7.3	8.3

Table 7: Processor Utilization Improvement of Interleaved Scheme.

Number of Contexts	Two	Four	Eight
<b>Base Pipeline</b>			
Water	<b>8.9%</b>	<b>10.9%</b>	<b>14.1%</b>
<b>MP3D</b>	-0.7%	<b>4.8%</b>	<b>8.9%</b>
<b>ocean</b>	-2.2%	<b>1.0%</b>	7.5%
<b>LocusRoute</b>	<b>10.6%</b>	<b>4.8%</b>	<b>2.8%</b>
Cholesky	<b>16.7%</b>	<b>23.7%</b>	<b>10.2%</b>
PTHOR	1.0%	<b>5.9%</b>	<b>14.3%</b>
Geometric mean	5.5%	<b>8.3%</b>	<b>9.6%</b>
<b>Deeper Pipeline</b>			
Water	<b>15.2%</b>	<b>24.2%</b>	31.7%
<b>MP3D</b>	<b>2.7%</b>	<b>15.8%</b>	27.2%
<b>ocean</b>	<b>2.7%</b>	<b>14.6%</b>	<b>22.8%</b>
<b>LocusRoute</b>	<b>13.7%</b>	<b>24.1%</b>	<b>33.2%</b>
Cholesky	<b>15.0%</b>	<b>28.7%</b>	<b>29.8%</b>
PTHOR	<b>3.2%</b>	<b>19.1%</b>	<b>41.4%</b>
Geometric mean	<b>8.6%</b>	<b>21.0%</b>	<b>30.9%</b>

that must be saved during a context switch on a single-context processor. The third requirement is that additional control logic and its associated state needs to be added to schedule the multiple contexts on the processor and insure correct operation of the pipeline for all exceptional conditions.

In this section, instead of focusing on general issues in building multiple-context processors, we focus primarily on the issues that will allow us to differentiate between the two multiple-context schemes. Since some of the different complexities reflect back on the macro-architecture, we first explore the macro-architectural issues and then turn to micro-architectural issues.

#### 4.1 Macro-architectural Considerations

In modern instruction set architectures, many of the pipeline dependencies that must be obeyed are made visible to the compiler. This is done in order to allow the compiler to schedule the code around these hardware dependencies. Since the compiler has more global knowledge about the program, it can often make scheduling decisions that are not practical to make in hardware at run-time. This has a simplifying effect on the hardware, as the only place where hardware is needed to resolve dependencies is for those cases where the compiler does not have enough knowledge to perform an adequate job.

Since the blocked scheme has only a single context active at any given point, the decision to resolve pipeline dependencies in the compiler or hardware can be made on the basis of cost/performance analyses very similar to the well-understood single-context processor tradeoffs. However, for the interleaved scheme, the instruction issue of a single context relative to the processor instruction stream cannot be determined statically. This greatly **impairs** the ability of the compiler to make scheduling decisions, forcing it to make conservative assumptions. For intra-context pipeline dependencies, the compiler must assume that no intervening instructions will resolve the dependency. Inter-stream dependencies cannot be resolved by the compiler, and require dynamic hardware resolution. Using examples, we explore the constraints imposed by the interleaved scheme on the macro-architecture. We first examine load and branch hazards, which show the effects of the interleaved scheme on intra-context pipeline dependencies. Then we examine the floating-point hazards, showing the effects on inter-context dependencies.

As our first example, we look at the tradeoff between the compiler removing load **hazards** by inserting

*nop* instructions into the code and the hardware providing interlocks to stall the pipeline to remove load hazards. For this example, the use of dynamic hardware resolution has a larger performance advantage for the interleaved scheme than the blocked scheme. For the blocked scheme, using the compiler to resolve load hazards results in a simpler design by removing the need for hardware to detect the load hazard. On the other hand, using hardware interlocks removes the need for the extra *nop* instructions, resulting in a smaller code space and its associated benefits. For the interleaved scheme, we can still use the compiler to insert *nops* to resolve the load hazards, however, providing hardware interlocks has an additional performance benefit over using the compiler. With an interleaved multiple-context processor, instruction(s) from other contexts may be in the pipeline between the two dependent instructions, and may provide enough delay that the hazard will never occur. The compiler is unable to exploit this potential hazard removal, as it does not know how many instructions from other contexts will lie between the two dependent instructions, and must conservatively assume that no instructions will separate them.

As an example where the cost portion of the cost/performance tradeoff equation changes, we examine the tradeoffs involved in reducing the penalty associated with control-transfer instructions. Many RISC processors utilize *delayed branches* to reduce the branch penalty. Assuming that *unfilled* branch delay slots execute regardless of the interleaving, there appear to be no performance differences for delayed branches between the blocked and interleaved processors. However, when the implementation issues involved in providing delayed branches for an interleaved processor are examined, it becomes apparent that there is a larger hardware cost than for the blocked processor (we explore this greater hardware cost in the micro-architecture section). This extra complexity for implementing delayed branches for the interleaved schemes reduces the cost differential between implementing delayed branches and *more* aggressive branch handling schemes such as using a branch target buffer [ 13].

As our **final** example, we explore a case where the compiler is simply unable to resolve pipeline hazards for the interleaved scheme. For this example, we examine the floating-point functional units. More specifically, let us assume the processor supports a set of floating-point functional units which are capable of operating in parallel, but with different latencies. For the blocked processor, the standard single-context processor tradeoffs between the compiler and hardware exist for scheduling the overlapping of operations, preventing result and issue hazards, and scheduling register file write port usage. However, for the interleaved scheme, the compiler does not have control over the inter-context interleaving, and can no longer guarantee that these hazards will not occur. For example, while the compiler will be able to prevent a single stream's instructions from violating the issue limits of a floating-point functional unit, the compiler cannot control two streams trying to issue operations too quickly to the same functional unit. Also, if a single write port to the floating-point register **file** exists, the compiler will be unable to schedule the write port between contexts, as the instructions from different contexts executing in different functional units may complete at the same time. To solve these problems, the interleaved scheme requires some hardware to perform dynamic hazard resolution.

While these examples only explore a small number of the tradeoffs involved in defining the **macro**-architecture, they point out the major differences between the interleaved and blocked schemes. The blocked scheme does not greatly alter the macro-architectural tradeoffs from those of the single-context architecture. On the other hand, the interleaved scheme favors making macro-architectural decisions which rely on dynamic pipeline-dependency resolution.

## 4.2 Micro-architectural Considerations

We have just seen that the two multiple-context schemes impose different constraints on the macro-architecture. We now show that this is also true for the micro-architecture. As mentioned before, the primary **micro**-architectural issues for a multiple-context processor involve the replication of the per-context state, and the scheduling and control of the multiple contexts on a single pipeline. We address each of these issues in turn.

#### 4.2.1 State Replication

Each context (or process) has a **fair** amount of state associated with it. Some of this state is kept in hardware, such as the floating-point and integer register sets, the program counter, and the processor status word. The rest of the state, such as the process control block, page tables, and the stack and heap are kept in main memory or on disk. When a context yields the processor, any of this **state** which has the potential to be lost by the time the context is restarted must be saved. Certain process state may be replicated for performance reasons, such as the TLB containing copies of the page table entries. For this redundant state, the decision to save state across a context switch depends on the probability of the state being lost, the performance cost of losing the state, and the hardware and performance cost of saving the state.

Since multiple-context processors experience frequent context switches, state-saving solutions which incur high overhead (such as writing the processor state to main memory) are not feasible, as the cost of the context switch will greatly overshadow any performance gains due to the addition of multiple contexts. Therefore, the portion of the state residing in hardware will need to be replicated on the processor in order to get a reasonable context switch cost.

While the interleaved and blocked schemes **require** the same amount of state to be replicated, the cycle-by-cycle switching of the interleaved scheme places more constraints on the hardware required for state replication. This results in two implementation implications. The first involves state selection control. Since state tends to be distributed throughout the processor, the interleaved scheme may require a selection decision to be made at each state location in order to be able to switch between contexts on a cycle-by-cycle basis. For example, a process identifier may be needed for the TLB lookup, and the controller selecting the proper process identifier may need to be separate from the controller selecting the next instruction to issue. In contrast, the blocked scheme has the option of keeping this decision local (e.g. making the decision solely at the controller which selects the next instruction), and broadcasting the result to the distributed state during the context switch pipeline flush. The second implication is for the amount of hardware needed to replicate the state. With the interleaved scheme, all the state must be simultaneously active, since there can be any interleaving of contexts within the pipeline. However, for the blocked scheme only a single set of state needs to be active, since at any given point there is only a single context executing on the machine. Since there may be extra functionality, such as a large drive capacity, associated with active state, requiring all sets of state to be active may require this extra functionality be replicated.

A good example of the difference in replication constraints is in the register file implementation. For both schemes, the register cells must be replicated to provide a cell per context. Since adding multiple contexts places no additional requirements on the number of reads or writes per cycle the register file needs to support, the portion of the register file devoted to the read and write ports does not need to be replicated. Thus, the bit and **word** lines, sense amplifiers, multiple port drivers, and a portion of the register addressing can be shared by all the contexts. For the interleaved scheme, cells **from** any context can be active during a cycle. In order to be able to address any memory cell, we need a new register specifier that consists of the context identifier and the original register specifier. This adds a delay to the register file fetch time due to the increased time to decode the larger register specifier. Looking at the blocked scheme, an optimization can be made to remove this extra delay. If we physically transfer data from the context cells to a shared master cell at each context switch, the register file fetch time can remain the same, since the register **file** only needs to be addressed by the register specifier. In addition to the register file, there are many other cases, such as the TLB process identifier mentioned earlier, where the single active context of the blocked scheme allows increased implementation flexibility.

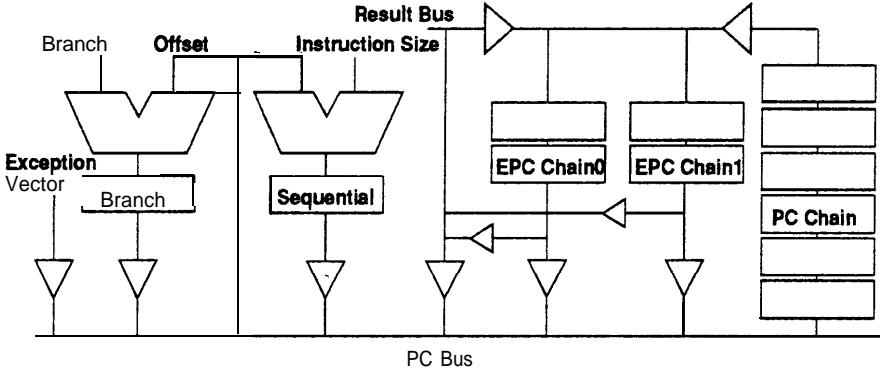


Figure 6: Blocked Processor PC Unit.

#### 4.2.2 Context Schedule and Control

Obviously the hardware required to identify the **current** context and determine the next context differs for the two multiple-context schemes. For the blocked scheme, where a single context is active, a single context identifier (CID) register can keep track of the currently executing context. When a context blocks, the CID register is loaded with the new context identifier. However, for the interleaved scheme, a single CID register is no longer enough to decide which context is performing an action, since instructions from different contexts can be at various stages in the pipeline. Instead, each instruction is effectively tagged with a CID. This CID flows down the program counter (PC) chain along with the PC value. At any given point in the pipeline, the appropriate CID determines the set of state to be used,

In addition to the control of the state machines, the multiple-context processor needs to be able to issue instructions from the multiple contexts to the pipeline, and handle pipeline exception conditions. To explore this further, we describe possible designs for the PC units of the two multiple-context schemes. To make the discussion of the differences required by the two schemes more concrete, we assume the deeper pipeline. The issues for the base pipeline are similar.

**Blocked Scheme** One possible PC unit for the blocked scheme supporting two contexts is shown in Figure 6. The rectangles in the diagram represent registers; all registers have input enable capability. The register enable and tristate control is not shown. This PC unit is very similar to that of a single-context processor, with the only difference being a modification to the exception PC (EPC) chain in order to support the multiple contexts. This **modification** adds an EPC chain per context, which doubles as both the exception PC chain and the context restart chain. Note, when we refer to the EPC chain, we are referring to the EPC chain of the current context unless otherwise specified. We explain the PC Unit operation by first describing its operation, and then discuss how the PC unit handles exceptions and context switches.

On any given cycle, one of several sources drives the PC bus. The possible PC sources are: (a) old PC value plus the instruction size (normal sequential flow), (b) old PC value plus branch offset (branch), (c) result bus (jump and jump indirect), (d) exception vector, or (e) EPC chain (restore from an exception or context switch). The exception vector and EPC chain provide the ability to take and recover from exceptions. In general, the depth of the EPC chain must be the number of branch delay slots in the machine, plus one. The reason this depth is required is that the excepting instruction could be in the delay slot of a branch. The PCs of the following instructions will not be strictly sequential for this case, so it is necessary to save the PCs up to and including the first non-sequential instruction.

During normal operation, the EPC chain of the active context is continuously being loaded with PC

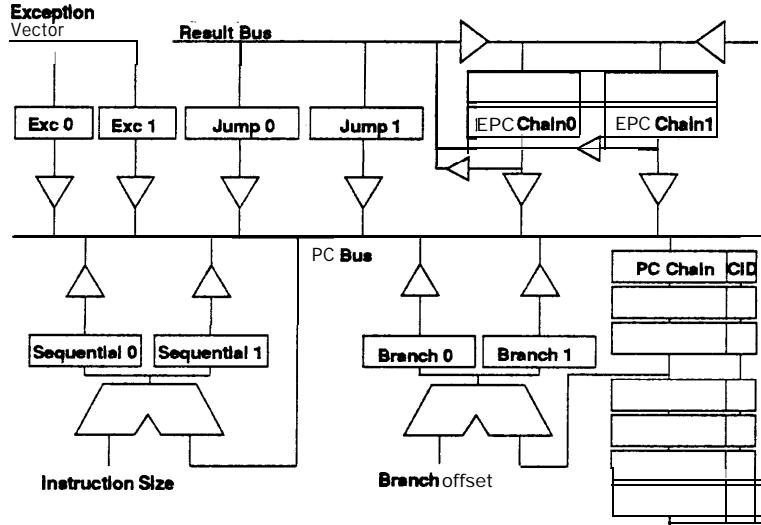


Figure 7: Interleaved Processor PC Unit,

values as instructions complete. When an exception occurs, the EPC chain of the active context contains the PC values of the excepting and subsequent instructions. All instructions in the pipeline are then squashed, and control is transferred to the exception vector. Since there is a single EPC chain for this context, the exception handler must save the EPC chain before enabling further exceptions. To return from the exception, the EPC chain is shifted out onto the PC bus as the new PC values, and once the chain is empty, the normal flow of instructions can proceed.

A data cache read miss, which forces a context switch, can be treated similarly to a hardware-handled exception. The context switch is delayed until the normal exception point, at which time the EPC chain for the blocked context is loaded as if an exception had occurred. The instructions in the pipeline **are** squashed and the next context to run is selected. The EPC chain of the new context is shifted out, and the new context starts executing **from** the instruction at which it had previously blocked.

Sharing the same EPC chain for both context switches and exceptions causes a problem when the exception handler takes a data cache miss. Under the above scheme, the EPC chain values would be destroyed by the context switch. Solutions to this problem are to either have an additional EPC chain for handling exceptions, or to disable context switches upon entry into an exception handler until the EPC chain can be saved (much like **interrupts** are disabled when taking an exception). Under most circumstances, the latter approach appears to be better, due to its lower hardware cost. While the latter approach requires a privileged instruction in the architecture for enabling/disabling context switches, this instruction can be useful in other situations, such as to avoid performing context switches in application critical sections.

**Interleaved Scheme** Figure 7 shows a possible implementation for an interleaved PC unit supporting two contexts. The interleaving of multiple contexts in the pipeline presents two complications in the design of the PC unit. The first complication is relatively minor. Since the new PC value becomes available a specific number of processor instructions after issue, this new PC value must be kept until the context becomes active again. For example, for our pipeline, the branch result always becomes available at the end of the EX cycle. Since the context may not be able to immediately drive its new PC value onto the bus at the point when the branch target becomes available, holding registers must be provided until the context is selected to drive the PC Bus. The same is true for the sequential instruction flow and other control transfers — the only difference is the exact

point in the pipeline where the new PC value becomes available.

The second, more serious complication arises because the number of instructions issued by the processor between two instructions from the same context is variable. This complicates both the branch and exception handling. Our macro-architecture specifies a delayed branch. A context may not have completed issuing all of the delay slot instructions by the time the branch is resolved. Therefore, a state machine needs to compute at the branch resolution point how many instructions from the delay slots have already been issued, and then force the remaining delay slot instructions to be issued before the branch occurs. While the hardware to do this is relatively **straightforward**, it is additional hardware that is not necessary for the blocked multiple-context processor.

Similarly, the value on the PC bus cannot be assumed to be **from** the same context as an instruction in the pipeline. This affects the computation of the branch target, since the branch offset cannot simply be added to the value on the PC bus — this value is either from another context or from the same context, but a variable number of instructions from the branch. To solve this problem, the branch offset is added to the PC value of the branch (which is taken from the PC chain).

Problems also arise for exceptions, as the PC values from the excepting context may be scattered throughout the pipeline, and even worse, there may not be enough PC values to completely fill the EPC chain. To gather the scattered PC values in the pipeline, instructions from the excepting context can be put into the EPC chain as they fall off the end of the pipeline (with instructions from the excepting context marked to not update any state). Note that this still allows instructions from the other contexts to continue; only the excepting context needs to be affected. As mentioned before, the length of the EPC chain is the minimum number of PC values that allow the processor to be restarted after an exception, therefore, not having generated enough PC values from the context will prevent proper **return** from the exception. To solve this problem, PC values can continue to be generated from the excepting context, until enough exist to fill the EPC chain. At this point, the exception vector can be forced onto the PC Bus during the context's next cycle, starting the exception handler.

Since the interleaved scheme uses the EPC chain to also save the PC values on a context switch, we would expect the **same** problems to occur for a context switch as for an exception. This is the case for the scattering of PC values within the pipeline, however, for the case where there are not enough PC values to fill the EPC chain, it is not necessary to continue to generate more PC values from this context. Since the context will be suspended, the information on the instruction sequence will remain in this context's branch, jump, and sequential PC registers and PC control state machine. When the context is made ready again, the saved portion of the PC chain is shifted out, then the PC control state machine for this context is unfrozen and the proper PC value can be driven from the branch, jump, or sequential register. This does require that the context switch control logic keep track of how many instructions it must shift out of the EPC chain.

In summary, the interleaved scheme places greater constraints on the architectural decisions than the blocked scheme. This is due to the nondeterministic instruction issue of a context with respect to the processor instruction stream. In particular, this nondeterminism removes much of the compiler's ability to effectively schedule code for the hardware's pipeline dependencies. As a result, for the interleaved scheme the tradeoff is tilted towards the use of more hardware-intensive dynamic dependency resolution. It may be that these potentially higher-performing dynamic options would have already been chosen due to other considerations, lowering the incremental cost of implementing the interleaved scheme. Indeed, the trend in processor architectures is towards the use of dynamic dependency resolution. For example, the DEC Alpha architecture [3] has eliminated branch and load delay slots, and has single-cycle issue for its floating-point units. However, these higher performance options do have their cost, not only in terms of hardware, but also in increased design and verification time.

## 5 Related Work

The modification of the traditional interleaved processors proposed is similar to two recent proposals, which employ interleaved processors to increase the functional unit utilization of superscalar processors. Prasad and Wu [18] propose a VLIW architecture, where instructions from several contexts are combined at run time to increase the utilization of the functional units. The paper uses an extremely shallow pipeline (3 stages), and does not address pipeline dependencies. Daddis and Tomg[9] present a superscalar architecture in which instructions from several contexts are used to **fill** a common instruction window. Instructions from this window are available for issue when they have no dependencies. This scheme can hide pipeline dependencies by issuing instructions from the same or a different context. Memory latency is also hidden by issuing instructions which do not depend on result of the memory operation. Both of these proposals do not contain enough implementation details to completely evaluate their impact on processor complexity. However, there appears to be a fair amount of added complexity to simultaneously issue instructions from multiple contexts. The alternative solution of only simultaneously issuing instructions from a single context, while interleaving contexts on a cycle-by-cycle basis may provide comparable **performance** at a lower cost. We are planning to investigate these issues in the near future.

Recently, **Farrens** and Pleszkun used both the interleaved and blocked schemes to help improve the issue rate of the Cray- 1S. They compared the schemes and showed the blocked scheme to have superior performance[4]. There are two reasons why they arrived at a different conclusion than us. First, their interleaved scheme switches between contexts regardless of the availability of instructions for the context, resulting in cycles being lost switching to a context which has no useful work. Second, they assumed no context switch overhead for both of the schemes. This second assumption is reasonable for their base architecture, where instruction decoding and availability decision is done **before the** issue of the instructions. However, in general for the blocked scheme, some time is needed to decode an instruction and determine whether a context switch should be performed, and this time results in a non-zero switch cost.

## 6 Conclusions

In this paper, we have proposed an interleaved scheme which appears attractive for implementing multiple contexts. The interleaved scheme has a performance advantage over the blocked alternative. While the blocked scheme hides memory latency, the interleaved scheme can also reduce the effects of pipeline interlocks and lower the context switch cost in addition to hiding memory latency. We have presented simulation results for two pipelines which show that these advantages of the interleaved scheme exist, with the differences between the two schemes becoming larger as the pipeline depth increases. However, performance alone does not determine the usefulness of an architectural feature. In order to determine the usefulness of interleaved contexts, the other side of the cost/performance tradeoff must also be explored. We examined the implementation complexity, concluding that the interleaved scheme is more complex, due to the nondeterministic issue of instructions **from** a single context relative to the processor instruction stream. This **nondeterminism** limits the amount of pipeline scheduling which can be done by the compiler. However, this complexity is not overwhelming, and particularly for processors which suffer from pipeline **inefficiencies**, the performance of the interleaved scheme may justify its additional complexity.

## 7 Acknowledgments

This research was supported by DARPA contract **N00039-91-C-0138**. In addition, James **Laudon** is supported by IBM and Anoop Gupta is partly supported by a NSF Presidential Young Investigator Award.

We thank **Mike** Smith for providing the parameterizable assembler and scheduler used in this study, and for explaining its intricacies. We also thank Steve Goldschmidt for his quick response in adding extra features to Tango.

## References

- [1] Arrant Agarwal, David Chaiken, Godfrey D'Souza, Kirk Johnson, David Kranz, John Kubiatowicz, Kiyoshi Kurihara, Beng-Hong Lim, Gino Maa, Dan Nussbaum, Mike Parkin, and Donald Yeung. The MIT Alewife machine: A large-scale distributed-memory multiprocessor. In *Workshop on Multithreaded Computers, Supercomputing '91*, November 1991.
- [2] Anant Agarwal, Beng-Hong Lim, David Kranz, and John Kubiatowicz. **APRIL**: A processor architecture for multiprocessing. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 104-114, May 1990.
- [3] Digital Equipment Corporation. *Alpha Architecture Handbook – Preliminary Edition*. 1992.
- [4] Matthew K. Farrens and Andrew R. Pleskun. Strategies for achieving improved processor throughput. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 362-369, May 1991.
- [5] Kourosh Gharachorloo, Dan Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15-26, May 1990.
- [6] Stephen R. Goldschmidt and Helen Davis. Tango introduction and tutorial. Technical Report CSL-TR-90-410, Stanford University, 1990.
- [7] Anoop Gupta, John Hennessy, Kourosh Gharachorloo, Todd Mowry, and Wolf-Dietrich Weber. Comparative evaluation of latency reducing and tolerating techniques. In *Proceeding of the 18th Annual International Symposium on Computer Architecture*, pages 254-263, May 1991.
- [8] Robert H. Halstead, Jr. and Tetsuya Fujita. MASA: A multithreaded processor architecture for parallel symbolic computing. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 443–451, June 1988.
- [9] George E. Daddis Jr. and H. C. Tomg. The concurrent execution of multiple instruction streams on superscalar processors. In *Proceedings of the 1991 International Conference on Parallel Processing*, pages I: 76-83, 1991.
- [10] Gerry Kane. *MIPS RISC Architecture*. Prentice-Hall, 1988.
- [11] D. Kroft. Lockup-free instruction **fetch/prefetch** cache organization. In *Proceedings of the 8th Annual Symposium on Computer Architecture*, pages 81-87, 1981.
- [12] Kiyoshi Kurihara, David Chaiken, and Arrant Agarwal. Latency tolerance through multithreading in large-scale multiprocessors. In *Proceedings of the International Symposium on Shared Memory Multiprocessing*, pages 91-101, April 1991.
- [13] J. K. F. Lee and A. J. Smith. Branch prediction strategies and branch target buffer design. *IEEE Computer*, 17:6–22, 1984.

- [14] Dan Lenoski, James **Laudon**, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The **directory-based** cache coherence protocol for the DASH multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 148-159, May 1990.
- [15] Ewing **Lusk**, Ross Overbeek, et al. *Portable Programs for Parallel Processors*. Holt, Rinehart and Winston, Inc., 1987.
- [16] MIPS Computer Systems, Inc. *MIPS R4000 Microprocessor User's Manual*. 1991.
- [17] G. F. **Pfister**, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. **McAuliffe**, E. A. Melton, V. A. Norton, and J. Weiss. The IBM research parallel processor prototype (RP3): Introduction and architecture. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 764-771, 1985.
- [18] R. Guru Prasadh and Chuan lin Wu. A benchmark evaluation of a multi-threaded RISC processor architecture. In *Proceedings of the 1991 International Conference on Parallel Processing*, pages I: 84-91, 1991.
- [19] Rafael H. Saavedra-Barrera, David E. Culler, and Thorsten von **Eicken**. Analysis of multithreaded architectures for parallel computing. In *Proceedings of the 2nd Annual Symposium on Parallel Algorithms and Architecture*, July 1990.
- [20] C. Scheurich and M. Dubois. Lockup-free caches in high-performance multiprocessors. *Journal of Parallel and Distributed Computing*, pages 25-36, January 1991.
- [21] G. E. Schmidt. The **Butterfly** parallel processor. In *Proceedings of the Second International Conference on Supercomputing*, pages 362-365, 1987.
- [22] Jaswinder Pal Singh, **Wolf-Dietrich** Weber, and Anoop Gupta. SPLASH: Stanford parallel applications for shared-memory. Technical Report CSL-TR-91-469, Stanford University, 1991.
- [23] Burton J. Smith. Architecture and applications of the HEP multiprocessor computer system. *SPIE*, 298:241-248, 1981.
- [24] Per Stenstrom, **Fredrick** Dahlgren, and Lars Lundberg. A lockup-free multiprocessor cache design. In *Proceedings of the 1991 International Conference on Parallel Processing*, pages I: 246-250, 1991.
- [25] **Wolf-Dietrich** Weber and Anoop Gupta. Exploring the benefits of multiple hardware contexts in a multiprocessor architecture: Preliminary results. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 273-280, June 1989.

## A Additional Results

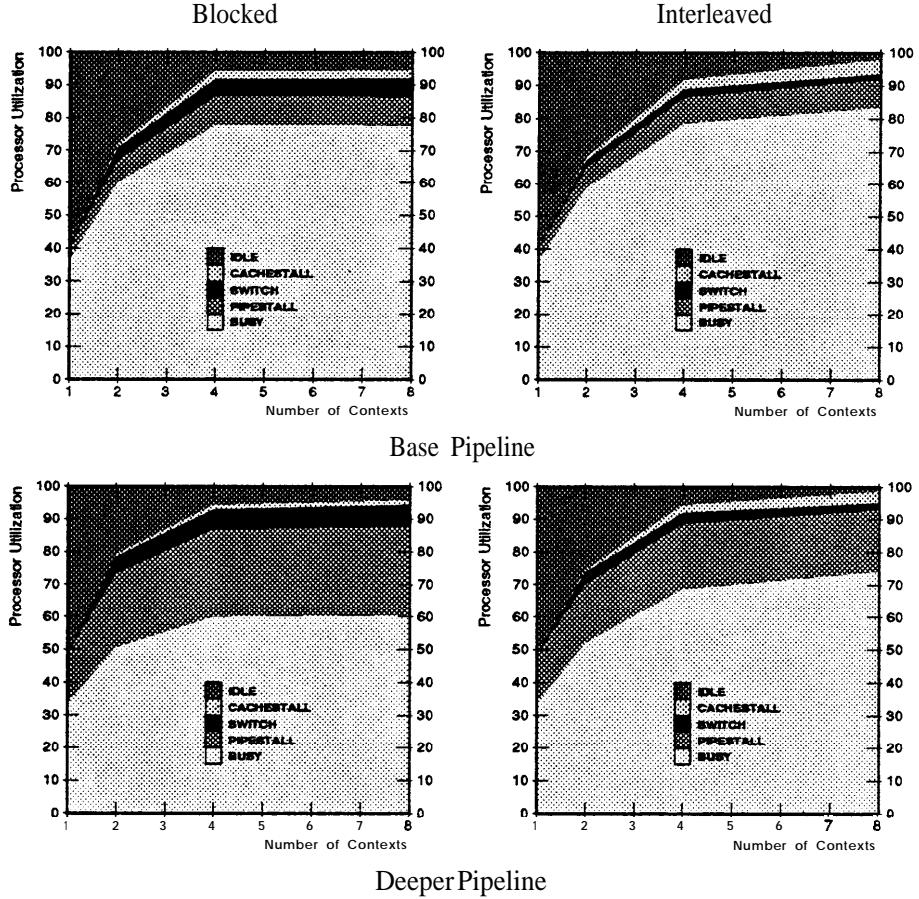


Figure 8: Processor Utilization — Ocean.

Table 8: Detailed Ocean Results.

	Base Pipeline								
	Single Context		Two Contexts						
	Block	Inter	Block	Inter					
Idle	59.9	59.9	28.1	31.7	5.7	8.3	5.6	2.1	
Cache Contention	0.2	0.2	1.1	0.9	25	2.9	2.5	4.5	
Switch Overhead	0.0	0.0	4.0	2.4	5.4	25	5.7	1.7	
Load stall	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
Branch Stall	1.7	1.7	2.8	2.7	3.6	3.6	3.6	3.8	
Execution Stall	2.4	2.4	3.9	35	5.1	4.3	5.1	4.5	
Busy	35.8	35.8	60.1	58.7	77.7	78.4	77.6	83.4	
Read Mii Rate	4.8	4.8	4.9	4.9	5.1	5.4	5.4	6.3	

	Deeper Pipeline							
	Single Context		Two Contexts		Four Contexts		Eight Contexts	
	Block	Inter	Block	Inter	Block	Inter	Block	Inter
Idle	51.7	51.7	21.0	25.6	5.8	5.7	4.3	1.4
Cache Contention	0.2	0.2	0.6	0.7	1.3	2.3	1.4	3.6
Switch Overhead	0.0	0.0	5.4	3.8	6.2	3.5	6.9	2.1
Load Stall	0.1	0.1	0.2	0.1	0.2	0.0	0.2	0.0
Branch Stall	4.9	4.9	7.3	4.3	8.7	4.1	8.7	35
Execution Stall	10.1	10.1	15.2	13.8	18.0	15.7	18.1	15.3
Busy	33.1	33.1	50.4	51.8	59.8	68.5	60.3	74.1
Read Miss Rate	4.7	4.7	4.9	4.9	4.7	5.4	5.3	6.3

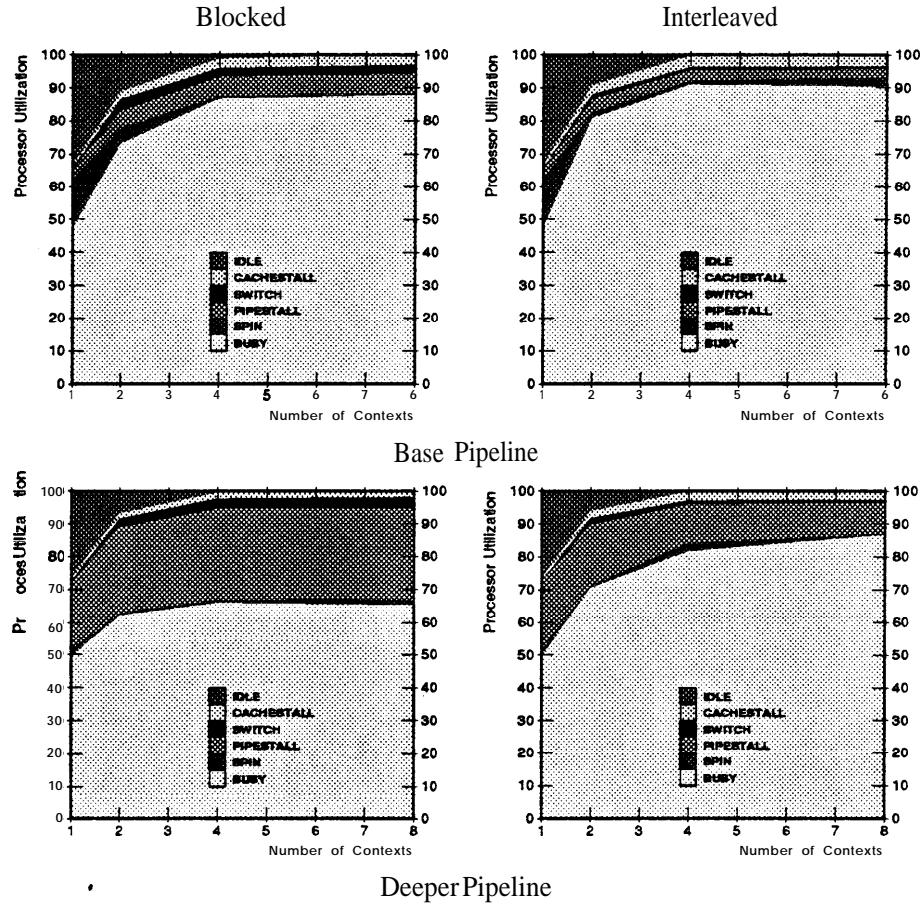


Figure 9: Processor Utilization — LocusRoute.

Table 9: Detailed Locus Results.

Base Pipeline								
	Single Context		'Two Contexts'		Four Contexts		Eight Contexts	
	Block	Inter	Block	Inter	Block	Inter	Block	Inter
Idle	33.3	33.3	11.2	9.3	1.1	0.3	0.1	0.0
Cache Contention	1.2	1.2	2.2	2.8	3.0	3.6	3.0	3.7
Switch Overhead	0.0	0.0	3.6	1.4	2.6	0.9	2.7	0.6
Load stall	1.0	1.0	15	0.6	1.8	0.1	1.9	0.0
Branch Stall	1.0	1.0	1.6	1.7	1.9	2.0	1.9	1.9
Execution Stall	1.3	1.3	1.9	1.7	2.3	1.2	2.4	0.7
Spin	15.3	15.3	4.9	1.7	0.5	1.0	0.2	2.7
Busy	46.9	46.9	73.0	80.7	86.7	90.9	87.9	90.3
Read Miss Rate	3.2	3.2	2.8	2.9	2.6	3.0	2.8	3.3
Deeper Pipeline								
	Single Context		Two Contexts		Four Contexts		Eight Contexts	
	Block	Inter	Block	Inter	Block	Inter	Block	Inter
Idle	26.5	26.5	6.9	6.2	0.5	0.3	0.0	0.0
Cache Contention	1.1	1.1	1.6	2.1	1.9	2.7	1.9	2.8
Switch Overhead	0.0	0.0	2.7	2.0	3.1	1.2	3.6	0.7
Load stall	3.1	3.1	3.9	2.6	4.1	0.5	4.1	0.0
Branch Stall	15.7	15.7	19.6	13.3	20.8	8.9	20.6	8.5
Execution Stall	2.2	2.2	2.7	2.8	2.9	2.3	2.9	1.2
Spin	2.2	2.2	0.5	0.6	0.9	2.5	1.8	0.0
Busy	49.3	49.3	62.0	70.5	65.7	81.6	65.1	86.7
Read Miss Rate	2.8	2.8	25	3.0	25	3.0	2.9	2.8

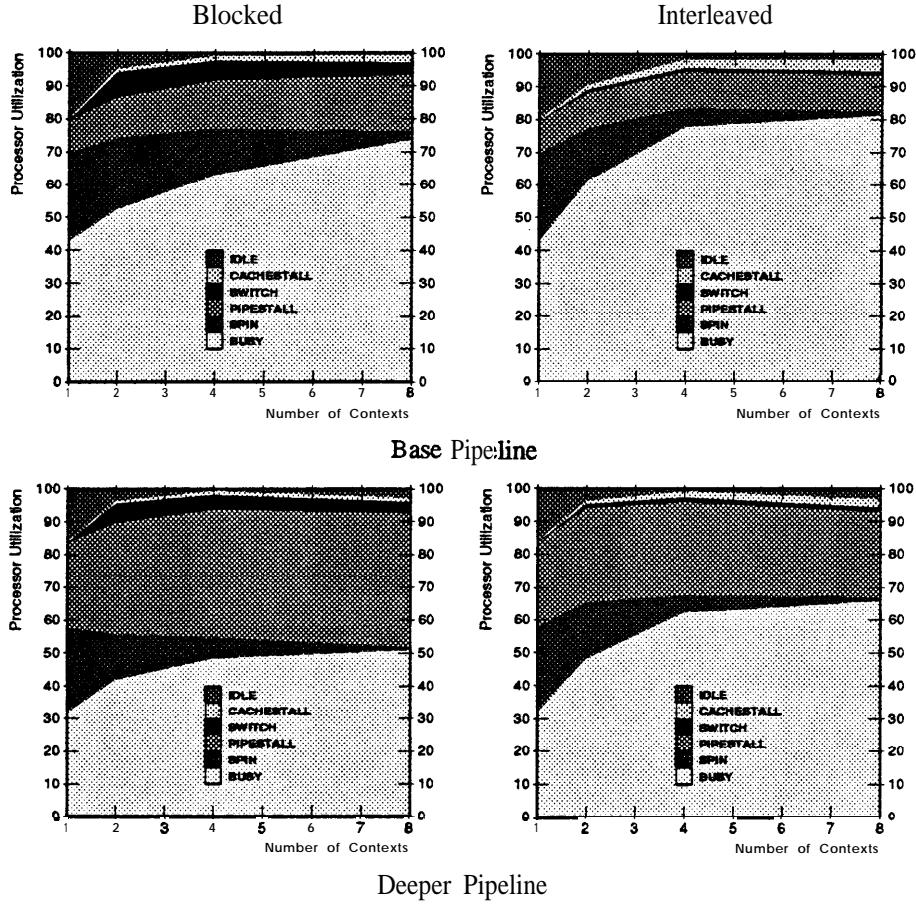


Figure 10: Processor Utilization—Cholesky.

Table 10: Detailed Cholesky Results.

		Base Pipeline						
	Single Context		Two Contexts		Four Contexts		Eight Contexts	
	Block	Inter	Block	Inter	Block	Inter	Block	Inter
Idle	19.8	19.8	4.9	95	1.0	1.7	0.8	1.7
Cache Contention	0.8	0.8	1.1	15	15	2.7	2.2	3.7
Switch Overhead	0.0	0.0	7.6	1.0	6.0	1.1	3.8	1.1
Load Stall	0.5	0.5	0.6	0.3	0.8	0.1	0.9	0.0
Branch Stall	1.2	1.2	15	1.8	1.8	2.3	2.2	2.4
Execution Stall	8.2	8.2	10.0	8.6	12.0	8.7	13.9	8.6
spin	27.1	27.1	21.9	16.2	14.1	5.7	2.6	1.3
Busy	42.4	42.4	52.3	61.0	62.8	77.7	73.6	81.1
<b>Read Miss Rate</b>	1.1	1.1	1.3	15	1.7	2.0	2.3	2.8
Deeper Pipeline								
	Single Context		Two contexts		Four Contexts		Eight Contexts	
	Block	Inter	Block	Inter	Block	Inter	Block	Inter
Idle	16.0	16.0	3.8	3.9	0.8	0.9	2.9	3.3
Cache Contention	0.5	0.5	0.8	1.0	1.1	1.9	1.3	2.6
Switch Overhead	0.0	0.0	5.9	1.2	4.6	15	35	1.4
Load Stall	1.6	1.6	2.1	1.4	2.4	0.8	2.6	0.4
Branch Stall	4.3	4.3	5.7	45	65	3.7	6.9	2.9
Execution Staiii	20.0	20.0	26.2	225	29.9	23.6	31.4	22.3
Spin	<b>25.9</b>	<b>25.9</b>	13.8	17.5	6.4	55	0.8	1.2
Busy	31.6	31.6	41.7	47.9	48.2	621	S0.8	65.9
<b>Read Miss Rate</b>	1.1	1.1	1.2	1.4	1.8	2.0	2.2	2.6