Measurement-Based Characterization of Global Memory and Network Contention, Operating System and Parallelization Overheads: Case Study on a Shared-Memory Multiprocessor *

Chitra Natarajan Sanjay Sharma Ravishankar K. Iyer

Center for Reliable and High Performance Computing
Univeristy of Illinois at Urbana-Champaign
1308 W. Main Street, Urbana, IL 61801

Abstract

This study presents a characterization of (1) the global memory and interconnection network contention overhead, (2) the operating system overheads, and (3) the runtime system parallelization overheads for the Cedar shared-memory multiprocessor. The measurements were obtained using five representative compute-intensive, scientific, loop parallel applications from the Perfect Benchmark Suite. The overheads were measured for a range of Cedar configurations from 1 processor to the full 4-cluster/32-processor configuration, thus characterizing the effect of this scaling on the overheads. For the full 4-cluster Cedar, the operating system overhead was found to constitute 5-21% of the total completion time of an application. The parallelization overhead accounts for 10-25% of the application completion time, and the overhead due to global memory and network contention contributes 8-21% of the application completion time.

1 Introduction

The complexities due to network contention, synchronization, cache coherency, communication and parallel work distribution introduce various overheads in a parallel processing system. A primary goal of a parallel processing system is to maximize performance and minimize overhead so as to speedup the execution time of an application. An important step towards realizing this goal is to measure, quantify, characterize, and understand the major overheads involved.

There are several studies that characterize or predict application performance in a dedicated environment [1-4]. A methodology for characterizing the behavior of shared-memory hierarchies for multiprocessors using a set of parameterized kernels was introduced by Gallivan [1]. Fatoohi [2] attempted to understand application performance by investigating vector performance in several machines. A methodology for predicting the performance of a full application using measured, representative kernels of code was proposed by Koss [3]. Saavedra-Barrera attempted to predict application performance using a more detailed characterization of individual code sections [4]. Several other studies have looked at OS and memory performance, and system overheads [5-8]. Lenoski et al., [5] presented the hardware overhead of the directory-based cache coherence in the DASH prototype and characterized the effectiveness of coherent caches. Torrellas [6] used a hardware monitor to study the caching and synchronization performance of the IRIX OS running on a 4-processor SGI POWER Worstation. Clark [7] used a performance monitor to study memory performance of a VAX 11/780 and a histogram hardware monitor to measure VAX 8800 performance [8]. An earlier study [9] on Cedar used computation kernels to study the performance difference between using the global memory with and without prefetches and caching. The study also presented Cedar performance results for Perfect Benchmarks for different levels of compiler and hand optimizations.

This paper provides a measurement-based characterization of the overhead due to global memory and interconnection network contention, the operating system and the runtime system parallelization overheads of the Cedar shared-memory multiprocessor in a dedicated, single user setting. To our knowledge, there is no other study that provides a similar detailed characterization of performance overheads from the runtime system, operating system and hardware perspectives.

The paper is organized as follows. Section 2 describes the experimental environment – the Cedar architecture, the Xylem OS, the Cedar Fortran runtime library, and the applications used in the study. Section 3 presents a high level view of the performance and overheads. Section 4 describes the runtime library and operating system instrumentation, and the measurement facility. The operating system overheads, the runtime system parallelization overheads, and the global memory and network contention overheads are quantified in sections 5, 6, and 7 respectively. Section 8 provides the conclusion.

*Acknowledgement: This research was supported by NASA under Grant No. NAG-1-613, in cooperation with the Illinois Computer Laboratory for Aerospace Systems and Software (ICLASS), and by the NSF under Grant No. NSF-MIP-89-20891.

2 Experimental Environment

Cedar Architecture In this section we summarize the key features of the Cedar system. For more detailed descriptions see [9,10]. The Cedar system currently consists of 4 clusters of processors connected through an interconnection network to a shared global memory (64 MB). Each cluster is a modified Alliant FX/8, with 8 computational elements (CEs), 64 MB of local memory, a 4-way interleaved shared data cache and a number of peripheral I/O processors (IPs). Each cluster also has a concurrency control bus which enables fast cluster-level parallel loop distribution, and fast synchronization of processors within a cluster. The CEs are pipelined vector processors. The global memory consists of 32 independent modules and is double-word (8 bytes) interleaved and aligned. A Global Interface connects each CE to the interconnection network, which in turn is connected to the shared global memory. The network is a two-stage shuffle-exchange network consisting of 8x8 crossbar switches. There is one network for the path going from the CEs to the global memory and another network for the path coming back from the global memory to the CEs.

Cedar Xylem OS The Cedar operating system, Xylem, manages the Cedar hardware resources. Xylem is an extension of Unix, modified to exploit the hierarchical nature of the Cedar architecture. The Cedar system provides three levels of parallelism. Vector instructions and intracluster parallelism are supported by the Cedar hardware, while intercluster parallelism is managed by Xylem along with runtime libraries [11]. The primary structure added to Unix, the *Xylem process*, is made up of one or more *cluster tasks* which can share portions of their address space. Xylem also provides multitasking and virtual memory management of the Cedar memory system, system calls for creating, starting, and stopping tasks, and calls for inter-task synchronization.

Cedar Fortran Runtime Library Cedar Fortran provides two constructs – the hierarchical SDOALL/CDOALL construct and the flat XDOALL construct – to exploit loop level parallelism. They allow the individual iterations of a loop to be spread across all 32 processors. Dependencies between iterations of the loop are not allowed. Both the constructs are implemented through helper tasks. The Cedar Fortran runtime library creates a helper task on each cluster other than the master cluster (where the program itself is started) with the help of the OS. Each cluster then independently handles the scheduling of its tasks. Within each cluster, all 8 CEs are gang scheduled.

In the hierarchical construct, the CDOALL loop (cluster loop) is nested in an SDOALL loop (spread loop). The iterations of the outer loop (SDOALL) are self-scheduled one at a time to each helper task and the main task. The inner loop (CDOALL) is then spread across the 8 processors of the cluster. When a helper task is scheduled to run on its cluster, it begins spin-waiting for work. When the main task of an application encounters an SDOALL, it posts the same in the shared global memory. When this is seen by a helper task of that application, it joins in the execution of the loop. After each SDOALL loop, the main task spin waits at a barrier for all the helpers which entered the loop to detach themselves. This is to ensure that all helper tasks are finished with their work before the main task executes the code after the loop.

In the XDOALL construct, a single CE on the master cluster enters the XDOALL, activating one CE on each helper task. These 4 *lead* CEs then enter the user's code causing all CEs on each cluster to become active. Each CE then independently calls the next-iteration function to get an iteration of the loop body. They continue to compete for iterations of the loop, until there are no more, then the CEs of each cluster synchronize using the high speed synchronization bus available in each cluster. When all CEs of a single cluster have synchronized, one of them continues on into the runtime library. Here, the CE from the main task spin waits at a barrier for all the helpers which entered the loop to detach themselves. Then the main task continues on.

In addition to the above two constructs, Cedar Fortran also provides DOACROSS loops to make it possible to serialize regions within a parallel loop.

Applications Used in the Study High performance multiprocessor systems such as the Cedar supercomputer are mainly intended for large scale scientific computing. In this study, five representative compute-intensive, scientific applications from the Perfect Benchmark Suite [12] were used to characterize the global memory and network contention, operating system and parallelization overheads of the Cedar multiprocessor. The codes for the five applications (FLO52, ARC2D, MDG, OCEAN, and ADM) used are the best that can be generated by a parallelizing compiler, as described in [13]. These applications use algorithms such as sparse linear system solvers, rapid elliptic solvers and ordinary differential equation solvers which are representative of the basic set of algorithms used in various computational disciplines. All of these applications predominantly consist of loops which can be concurrently executed. The application FLO52 only uses the hierarchical SDOALL/CDOALL construct; ADM uses only the flat XDOALL construct; the other applications use both SDOALL/CDOALL and XDOALL constructs to exploit loop level parallelism. The applications also have a few main cluster-only loops (CDOALL or CDOACROSS without an outer spread loop), i.e., these loops only spread the iterations among the processors in the cluster executing the main task and not across clusters.

3 High Level View of Performance & Overheads

Table 1 summarizes the completion times (CTs) of the target applications when they are executed on different Cedar configurations. [1] The measurements were made in a dedicated, single user setting with only the target application and the OS executing on the system.

[1†] All the 4 processors for the 4-processor configuration are from the same cluster.

3.1 Speedups and Concurrency

Table 1 presents the speedups obtained when multiple processors are used over that on a 1-processor configuration. The average concurrency/processor utilization is also presented. The average concurrency represents the average number of active processors at any given time during the program execution. The average concurrency values were obtained using a software monitor, *statfx*. This monitor measures the concurrency on each cluster; for the multi-cluster Cedar configurations, the values provided in the table are the sum of the concurrency values on the different clusters.

The key results from Table 1 are:

(1) MDG obtains nearly linear speedups as more number of processors are utilized. This is because of the high degree of parallelism (reflected by the high average concurrency/processor utilization values) available in MDG. OCEAN shows near linear speedups upto 8 processors, but beyond 8 processors the speedup becomes sub-linear due to decreasing level of available concurrency. The other applications FLO52, ARC2D, and ADM achieve sub-linear speedups with scaling due to poor concurrency.

(2) We also see that the speedups achieved are lower than the average concurrency values (the average number of active processors during the program execution). This indicates that part of the active processors' processing time is spent on different overhead activities in the multiprocessor systems. This study attempts to qunatify some of these overheads.

Program		1 proc	4 † proc	8 proc	16 proc	32 proc
	CT (s)	613	214	145	96	73
FLO52	Speedup	-	2.86	4.23	6.39	8.40
	Concurr	-	3.49	6.11	9.66	14.82
	CT (s)	2139	593	342	203	142
ARC2D	Speedup	-	3.61	6.25	10.54	15.06
	Concurr	-	3.70	6.82	12.28	20.56
	CT (s)	4935	1260	663	346	202
MDG	Speedup	-	3.89	7.44	14.26	24.43
	Concurr	-	3.92	7.60	15.14	28.82
	CT (s)	2726	711	381	230	175
OCEAN	Speedup	-	3.83	7.16	11.85	15.58
	Concurr	-	3.86	7.53	12.98	17.27
	CT (s)	707	208	121	83	80
ADM	Speedup	-	3.40	5.84	8.52	8.84
	Concurr	-	3.46	6.06	9.42	13.56

Table 1: CTs, Speedups and Average Concurrency

3.2 Overview of the Overheads

The Cedar multiprocessor system has various hardware and software support levels (Figure 1). The applications execute on top of a runtime system, which itself runs on top of the operating system. The operating system manages the underlying hardware resources. Each of these layers introduce various overheads which are also indicated in Figure 1.

In this study, we characterize the operating system overheads, the runtime system parallelization overheads, and the global memory and network contention overhead. To determine the contribution of each of the above overheads to the total completion time of an application, we breakdown the application completion time in finer details (Figure 2) in the later sections. The use of a shared coherent cache in Cedar circumvents the false sharing and cache coherency problems. However, there would still be capacity and conflict cache misses. The overhead due to these cache misses and the other overheads determined by the underlying hardware - the overhead due to TLB misses, and cluster-level *cdoall* parallel loop synchronization overheads - are not characterized in this study.

It must be noted here that, in this study, we are attempting to measure and characterize the overhead due to the *contention* in the global memory and interconnection network, introduced as a result of more than one processor issuing requests to the global memory in a multiple processor configuration. In our study, the different Cedar configurations have the same minimum global memory access latency, since they use the same interconnection network and memory. The fact that all configurations have the same minimum memory access latency actually enables us to isolate the contention factor. In reality, however, a machine with smaller number of processors might be expected to come with a smaller interconnection network and memory, and hence with a lower minimum memory access latency.

4 OS & RTL Instrumentation and Measurement Facility

The Cedar Xylem OS was instrumented to obtain a detailed characterization of the various OS activities. Events were inserted to record entries and exits from various routines such as: (a) routines to acquire/release locks, (b) context switching routine, (c) resource scheduling routines, (d) system call routines, (e) system trap routines, and (f) interrupt service routines.

Parallelization introduces overheads such as the time spent setting up parallel loop parameters, the time spent picking up loop iterations to execute, the time spent by helper tasks busy-waiting for work, and the time spent by the main task spin waiting at the loop barriers. To determine the parallelization overheads the Cedar Fortran runtime library was instrumented to record a variety of events such as: (a) the main task encountering an *s(x)doall* loop, (b) the helper task joing in the execution of an *s(x)doall* loop, (c) entry and exit from pick next iteration routine, (d) start and end of an *s(x)doall* iteration execution, (e) entry and exit from the *s(x)doall_finish_barrier* for the main task, and (f) entry and exit from the wait-for-work routine for the helper tasks.

The above instrumentation causes events to be posted to special hardware performance trigger points.

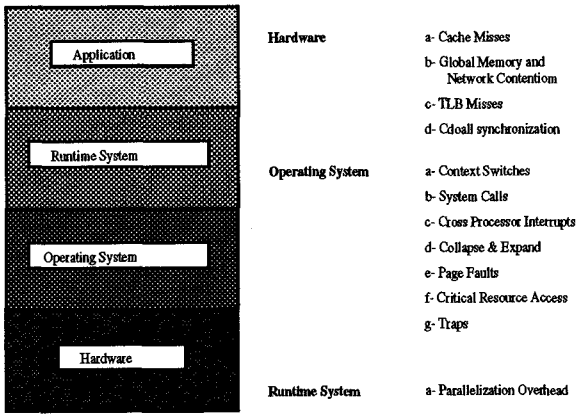


Figure 1: Overview of the Overheads

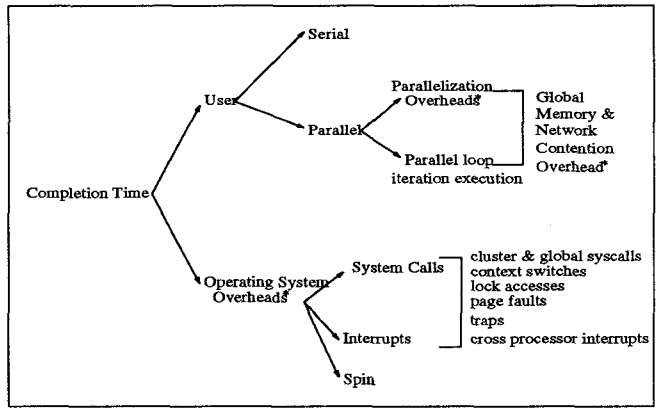


Figure 2: Detailed Breakdown of Completion Time

These performance trigger points are monitored by an external non-intrusive hardware performance monitor, *cedarhpm*, developed at UICSRD [14] and the event traces are collected and stored in the hardware performance monitor trace buffers. The trace buffers are off-loaded to a remote Sun Workstation at the end of the program execution for analysis. For each event, *cedarhpm* records the event *id*, the timestamp and the *id* of the processor on which the event occurred. The timestamp resolution is 50 nanoseconds. The recording of each event is as cheap as a single *move* assembly level instruction, and thus causes negligible overhead. In addition to recording the instrumented events for the RTL and OS, the Cedar Xylem context switching identifier instrumentation was turned on to record the context switching between the application task and system task. This enabled us to accurately determine the durations for which the target application was actually running on the clusters.

5 Operating System Overheads

To characterize the operating system overheads, the total completion time is broken into its individual components – user/CPU, system, interrupt, and spin times (Figure 3-(a)). This breakdown was obtained using a software measurement facility "Q" which monitors the utilization of each cluster. **User time** is the time spent by the cluster executing the user code of the application. The user time includes the actual busy time, stall times due to global memory accesses or cache refills, the time spent spinning on user-level synchronization locks or waiting at the barriers. **System time** is the time spent on general system work such as cluster and global system calls, context switches, and critical resource accesses. **Interrupt time** consists of the time spent servicing software interrupts and cross processor interrupts. **Spin time** is the kernel lock spin overhead, characterized by the time spent waiting for the release of shared global memory(shared by all the Cedar CEs) or private cluster (shared by the individual cluster resources: IP and single cluster CEs) memory locks.

Figures 3-(a) to 3-(f) illustrate the completion time breakdown on the different Cedar configurations for the applications FLO52, MDG, ARC2D, OCEAN, and ADM. The breakdown is provided for the main task of the application. Similar breakdowns were obtained for the helper tasks when multiple clusters are employed. The important results from the figures are:

(1) The operating system overheads become an increasingly more important component of the application completion time, as larger number of processors are used. While the operating system overheads form 3-4% of the application completion time on a 1-processor configuration, their contribution is 5-21% of the completion time on the 32-processor configuration.

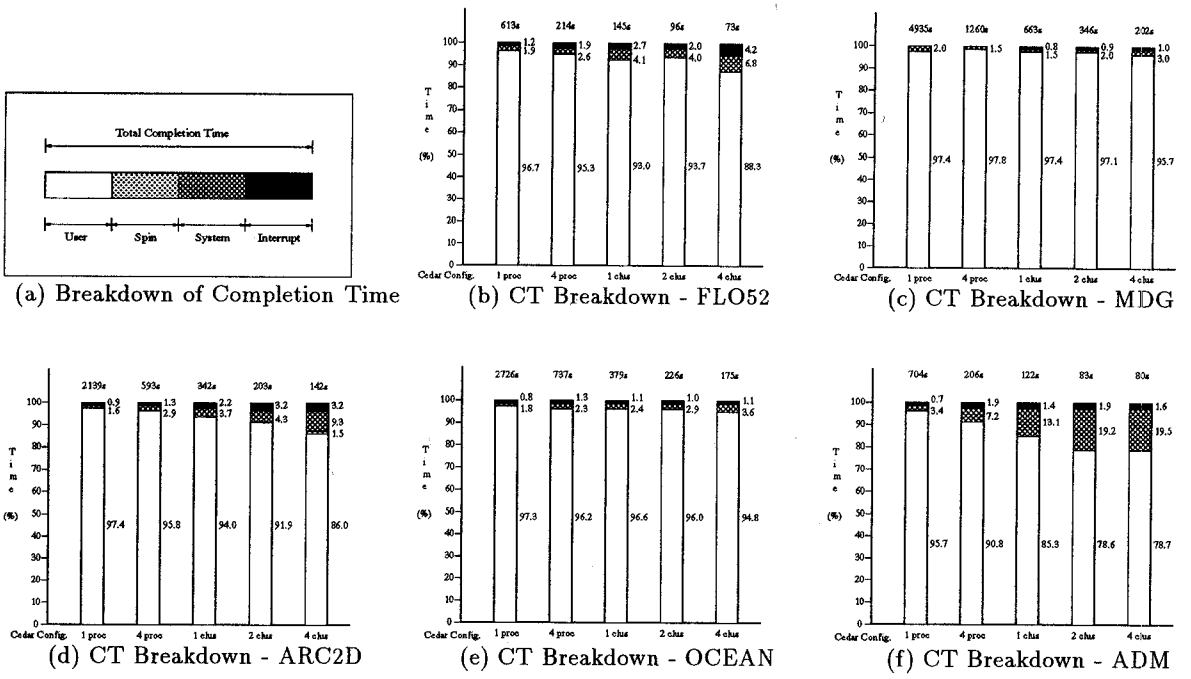
(2) The system time is the largest component (3-19% of the completion time) of the total operating system overhead followed by the time spent in handling interrupts (2-4% of the completion time). Kernel lock contention is negligible (kernel lock spin time is $< 1\%$ of the completion time) for the execution of all the applications on the different Cedar configurations.

5.1 Detailed Characterization of OS overheads for the 4-Cluster Cedar

Having looked at the high level view of operating system overheads for the various configurations, a detailed characterization of the OS overheads for the 4-cluster Cedar is presented in this section for three applications - FLO52, MDG and ARC2D.

Table 2 presents the overheads due to various operating system activities such as servicing cross processor interrupts (**cpi**), context switching (**ctx**), handling concurrent and sequential page faults (**pg flt (c)/(s)**), accessing (cluster and global) critical sections/resources (**Cr Sect (clus)/(glbl)**), servicing cluster and global system calls (**clus/glbl syscall**), and servicing asynchronous system traps (**ast**), in terms of their contribution to the application completion time.

From Table 2 we observe that individually the various OS activities constitute only a small amount of the total completion time. But, put together they are 5-21% of the application completion time. We see that cross processor interrupts, context switching, page faults and accessing (cluster) critical sec-



(a) Breakdown of Completion Time (b) CT Breakdown - FLO52 (c) CT Breakdown - MDG

(d) CT Breakdown - ARC2D (e) CT Breakdown - OCEAN (f) CT Breakdown - ADM

Figure 3: Completion Time Breakdown on Different Cedar Configurations

tions/resources account for more than 90% of the total operating system overhead. If the OS overheads are to be reduced, each of the above components would have to be made more efficient.

Overhead Category	FLO52 (s)	FLO52 %	ARC2D (s)	ARC2D %	MDG (s)	MDG %
cpi	3.48	4.70	5.62	3.95	2.42	1.18
ctx	1.68	2.30	2.91	2.04	3.72	1.84
pg flt (c)	2.22	3.04	3.73	2.62	1.54	0.76
pg flt (s)	1.64	2.25	2.20	1.54	0.48	0.23
Cr Sect (clus)	1.17	1.60	3.43	2.77	2.42	1.18
Cr Sect (glbl)	0.23	0.33	1.18	0.83	0.80	0.39
clus syscall	0.26	0.35	0.84	0.59	0.48	0.28
glbl syscall	0.04	0.05	0.05	0.04	0.03	0.01
ast	0.03	0.04	0.18	0.13	0.05	0.02

Table 2: Detailed Characterization of OS overheads

Context switching takes place in a dedicated system, when the application task blocks for I/O or when the OS server must perform some bookeeping. The measurements show that this context switching activity can account for as much as 2% of the completion time. We could reduce some of the context switching overhead by having the context switching routine cooperate with the runtime library. For example, if it is known that a context switch is occuring when the task is spin waiting for work (if a helper task) or spin waiting at the barrier (if a main task) some of the (inactive) register saves/restores could be avoided.

Cross processor interrupts are issued during concurrent page faults, explicit resource scheduling requests, system calls and context switching requests to obtain a single CE execution thread. We see that servicing cross processor interrupts consume as much as 4% of the application execution time. Although the special intra-cluster bus in Cedar leads to fast synchronization of the CEs to get a single execution thread, the saving/restoring of registers and other miscellaneous accounting calculations that are required to be performed by each CE prior to the synchronization leads to the rather large amount of time being spent on cross processor interrupt servicing.

Concurrent page faults are caused by two or more CEs simultaneously attempting to access a page which had not been accessed previously. Concurrent page faults are more expensive than sequential page faults, and are seen to account for as much as 3% of the completion time. Sophisticated compilation techniques are needed to reduce the concurrent page faults to sequential page faults, thus reducing the paging overhead.

Accessing (cluster) critical sections contributes 1-3% of the application completion time. These critical sections/resources are protected by cluster memory locks. As seen earlier, lock contention is not a problem. Therefore, to reduce the time spent

in the cluster critical sections we could determine the most frequently accessed critical sections and inline the lock acquistion and release code for the locks protecting such sections.

Having analysed the OS overheads, we now characterize the runtime system parallelization overheads.

6 Parallelization Overheads

To characterize the parallelization overheads we breakdown the user/CPU time of the target application tasks as shown in Figure 4. This breakdown was obtained from the program event traces recorded by *cedarhpm*, using the events instrumented in the Cedar Fortran runtime library. [2] The Figures 5 to 9 present the user time breakdown as percentages of total execution time for the five applications studied. For the 1-processor, 4-processor/1-cluster, 8-processor/1-cluster configurations the breakdown is provided for the single (main) task. For the 2- and 4-cluster Cedar the breakdown is provided for the main and the helper tasks. The figures also provide the actual total user time at the top of each bar.

The quantities below the horizontal line on each bar represent the percentage of total execution time spent executing *s(x)doall* loop iterations for both the main and the helper tasks, and the time spent executing serial code and main cluster-only loops for the main task. The quantities above the horizontal line characterize the parallelization overheads. For both the main task and the helper tasks the parallelization overheads includes the time spent setting up loop parameters, time spent picking up *s(x)doall* iterations and time spent determining that no more iterations are left. In addition, for the main task the parallelization overheads include the time spent spin-waiting at the *s(x)doall_finish_barrier*, while for the helper tasks the overheads include the time spent busy-waiting for parallel loop work.

From the figures we see that as more number of processors are utilized the parallelization overhead increases rapidly, especially when multiple clusters are used. For example, for the application FLO52 the parallelization overhead accounts for just 1.5% of the completion time on a 4-processor Cedar and 2% of the execution time for the 8-processor configuration. When 2 clusters (16 processors) are used however, the parallelization overhead consumes 9% of the execution time for the main task and 26% of the completion time for the helper task. On the full 4-cluster Cedar, the parallelization overheads account for 18% of the completion time for the main task and 39% of the completion time for the helper tasks. The figures also show that the two major components of the parallelization overheads for the main task are the *s(x)doall* barrier wait times on the multicluster configurations and the *xdoall* loop distribution overhead. For the helper tasks the two main constituents of the parallelization overheads are the *xdoall* loop distribution overhead and the time spent waiting for parallel loop work.

Barrier wait times: A major cause for the jump in the parallelization overhead when going beyond one cluster is because multiple cluster configurations require *barrier synchronization* of the cluster tasks after every parallel loop execution that involves more than one cluster. These synchronizations degrade performance for problems that do not have sufficiently large loop granularity, as is the case with the Perfect Benchmarks' data set. The measurements show that the barrier wait times account for 2-7% of the completion time on a 2-cluster configuration, and increases to as much as 7-16% of the completion time for the 4-cluster Cedar. It is well known that barrier synchronization time increases with the number of tasks involved in the synchronization. Hence, this number can be expected to increase as more clusters are added to the system.

An interesting question to ask at this point is: *was clustering a good idea?* We believe that clustering has helped. With loop-based parallelism if the system had 32 independent processors rather than 4 clusters, every loop barrier could require barrier synchronization of 32 tasks rather than just 4 tasks. This in itself would result in very large barrier synchronization times. Moreover, with just the simple busy-waiting barrier synchronization mechanism currently used in the system, this would create a hot spot and could severely degrade performance for all traffic in the multistage interconnection network [15]. Special mechanisms such as hardware message combining in the interconnection network or software combining tree approach [16] would be needed to reduce the hot spot effect. What clustering has achieved is to localize the synchronization of processors within a cluster, and then have just one processor from each cluster issue requests to the global memory for global synchronization, thus eliminating a considerable amount of network traffic and contention.

Nevertheless, barrier wait times form a substantial amount of the total overhead. Hence, it might be worth the effort to try eliminate some of the barriers. For example, we could identify and merge several parallel loops in a row that do not have dependencies among them. We could also try eliminating dependencies among parallel loops in a row, if possible, by further detailed analysis of algorithms involved and rewriting of the code, thus transforming a series of multicluster barriers into a single multicluster barrier. Indeed such efforts have been undertaken and along with other manual optimizations have resulted in 2-fold performance improvement for FLO52 [9].

Xdoall loop distribution overhead: Another major contributor to the increasing parallelization overhead with scaling is the time spent in setting up loop parameters, picking up iterations and determining that no more iterations are left, for the *xdoall* parallel loop execution. This *xdoall* overhead increases from under 1% of the completion time on a 4-processor configuration to over 10% of the completion time on a 4-cluster/32-processor Cedar. On the other hand for the *sdoall/cdoall* loops these activities account for less

[2] To determine the time spent in main cluster-only loops the application code was instrumented to post events to the hardware performance trigger points before and after such loops. Since the occurences of these loops were rare the perturbation was still minimal.

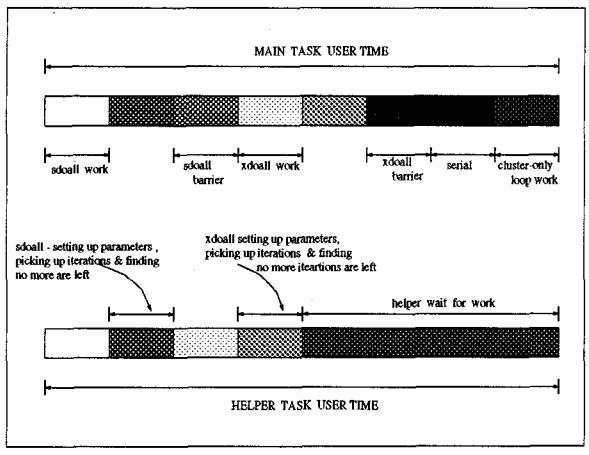


Figure 4: Breakdown of User Time

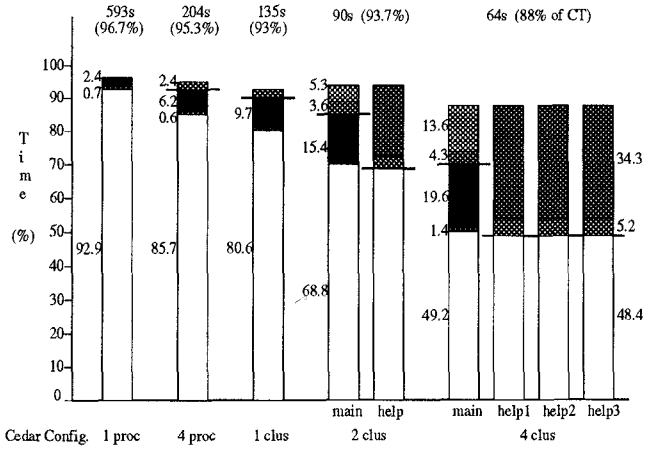


Figure 5: User Time Breakdown for FLO52

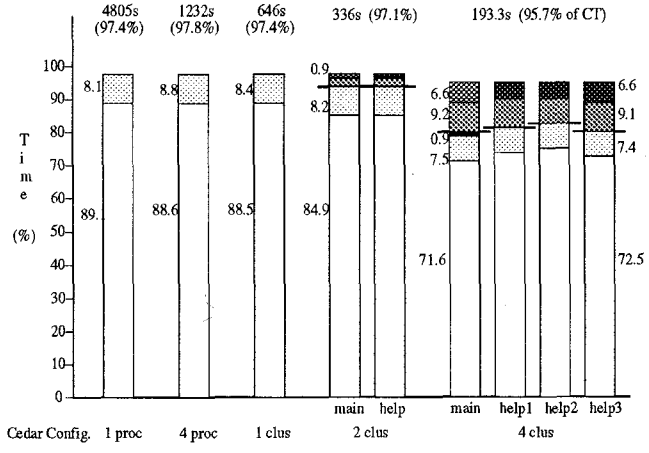


Figure 6: User Time Breakdown for MDG

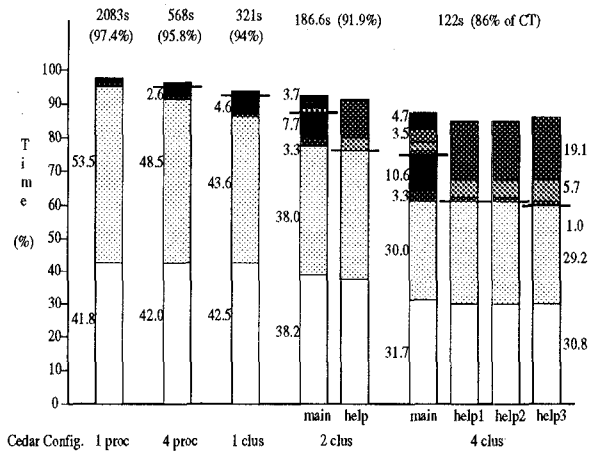


Figure 7: User Time Breakdown for ARC2D

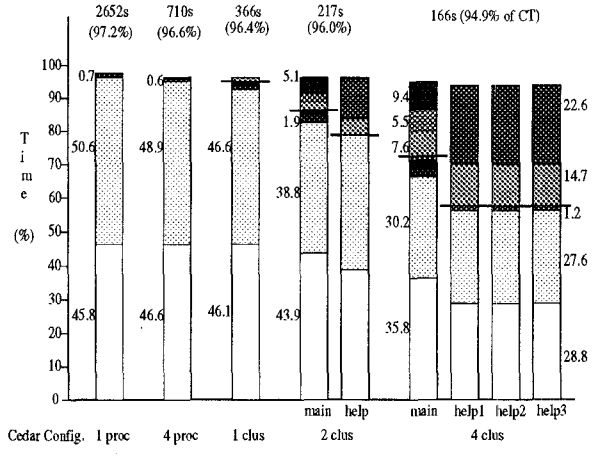


Figure 8: User Time Breakdown for OCEAN

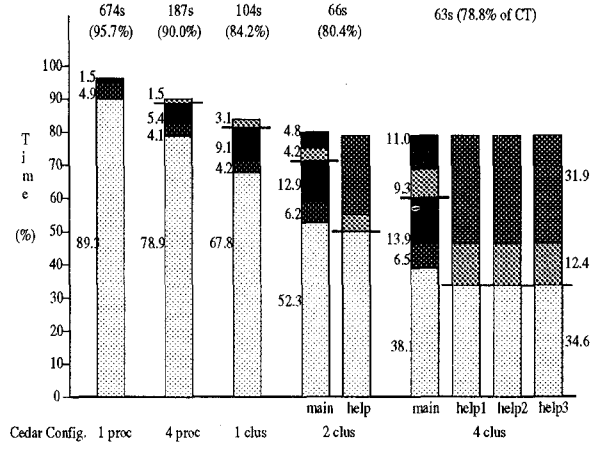


Figure 9: User Time Breakdown for ADM

than 1% of the completion time. This is because, while the *xdoalls* exploit the clustering hardware by using the concurrency control bus for barrier synchronization, for the loop iteration distribution on the other hand, each processor in the Cedar configuration individually and independently issue test and set requests to the critical section locks such as the lock protecting the loop iteration index. This leads to global memory and network contention, and hence, to larger amount of time being spent on picking up loop iterations and in determining that no more iterations are left. But, with *sdoall/cdoalls* only 1 processor from each participating cluster issues requests to the global memory for the critical section locks, for the *sdoall* loop distribution. Therefore, there is little global memory and network contention and little overhead. The inner *cdoall* loop distribution is done using the fast cluster concurrency control bus and creates no network traffic.

Again, we observe the benefit of clustering in reducing the network traffic and contention. We learnt that the *xdoalls* were often used for convenience, since it is easier to convert a loop into an *xdoall* than to strip-mine it into the hierarchical *sdoall/cdoall* nest. Considering that the *xdoall* loop distribution overhead is as much as 10% of the completion on the 4-cluster Cedar it might be worth the effort to try exploit the hierarchical *sdoall/cdoall* construct.

Helper wait times: The measurements show that a large amount of time is wasted by the helper tasks as *helper_wait* times. For example, for the application FLO52 *helper_wait* times constitute as much as 22% of the completion time on the 2-cluster configuration and 34% of the completion time on the 4-cluster configuration. This is because, when the main task is executing serial code or main cluster-only loops, the helper tasks have no work and merely spin-wait looking for parallel loop work. Also, when the main task is spin waiting at the barrier, any helper task that has already reached the barrier spin waits looking for the next parallel loop work. Hence, the *helper_wait* times correspond to the serial code and main cluster-only loop execution time, and the barrier wait times of the main task.

To summarize, we see from the above analysis that the parallelization overheads account for a substantial amount of the total execution time, especially on multiple cluster configurations. The parallelization overheads account for 6-10% of the completion time for the main task and 17-29% for the helper task on the 2-cluster Cedar, and consume as much as 10-25% of the completion time for the main task and 15-44% of the completion time for the helpers on a 4-cluster Cedar.[3] The two major components of the parallelization overheads are barrier synchronization and parallel loop distribution for the loop construct that views Cedar as a machine of independent processors (rather than as clusters of processors) during the loop iteration distribution phase.

[3] The overheads are higher for the helper tasks since the overheads include the time spent by these tasks spin-waiting for parallel loop work, when the application is executing serial code or main cluster-only loops.

7 Global Memory and Network Contention Overhead

In this section, we estimate the contribution of the global memory and interconnection network contention overhead to the overall application completion time in a multiprocessor Cedar configuration. During serial code execution, only one processor from each cluster is active. The processor on the main cluster executes the serial code while the processor from a helper task spin-waits, checking the *sdoall_activity_lock* in the global memory every few cycles for parallel loop work. Hence, the contention will be negligible. System work also entails little network activity. On the other hand, during parallel loop execution (main cluster-only loops or cross-cluster *sdoall/cdoall* and *xdoall* loops) many processors are active. Moreover, most parallel loop execution is such that different processors work on different sections of a vector. Therefore, there could be multiple vector requests issued to the global memory from different processors at the same time leading to substantial global memory and network activity, and hence contention. We now present a methodology to estimate this contention overhead.

Average Parallel Loop Concurrency: To estimate the global memory and network contention overhead we need to know the average parallel loop concurrency, i.e., the average number of processors involved in the parallel loop execution, and hence, contributing to the contention. This can be determined from *pf*, the fraction of the total completion time spent on parallel loop execution on each cluster,[4] and *avg_concurr*, the average concurrency on each cluster over the execution of the entire program. For the main cluster task, *pf* includes the fraction of the total completion time spent on main cluster-only loops. The concurrency during non-parallel work such as serial code execution, picking up iterations for the *sdoall* loops, spin-waiting at the barrier, and busy-waiting for work, is 1 on each cluster. Therefore, the average parallel loop concurrency, *par_concurr*, on each cluster can be determined from the following equation:

$$(1 - pf) + (pf * par\_concurr) = avg\_concurr.$$

The average parallel loop concurrency values on the multiprocessor Cedar configurations are provided in Table 3 for the five applications studied. For the multi-cluster Cedar configurations, the table provides the average parallel loop concurrency values for each task/cluster.

Estimating Contention Overhead: The program execution on the 1-processor configuration would have no network contention. However, all requests may not be satisfied with the minimum memory access latency since the global memory takes 4 processor clock cycles to process a request. Therefore, for example, if the processor issues two requests in successive

[4] For the *xdoall* loops, the iteration pick up is a parallel activity, and hence is included in the parallel fraction. Therefore, there is an overlap between the global memory and network contention overhead and the parallelization overheads for the applications employing the *xdoall* construct.

Config	Task	FLO52	ARC2D	MDG	OCEAN	ADM
4 proc	Main	3.88	3.94	3.96	3.92	3.96
8 proc	Main	7.28	7.64	7.79	7.88	7.93
16 proc	Main	7.01	7.63	7.88	7.42	7.55
	helper	5.93	7.45	7.84	7.62	7.45
32 proc	Main	6.85	7.62	7.98	5.74	5.89
	helper1	6.51	7.15	7.89	5.59	5.94
	helper2	6.34	7.16	7.92	5.61	5.91
	helper3	6.25	7.18	7.95	5.58	5.83

Table 3: Average Parallel Loop Concurrency

clock cycles to the same memory module the second one would be delayed. Nevertheless, the time taken to execute the parallel loop code of an application on the 1-processor configuration would give the minimum possible *total processing time* for the loop execution, for the interconnection network and the global memory organization of the system. Therefore, to estimate the overhead due to the *contention* in the interconnection network and global memory, introduced as a result of more than one processor issuing requests to the global memory in a multiprocessor configuration, we use the time taken to execute the parallel loop code on the 1-processor configuration as the *ideal total processing time* for the parallel loop execution.

The ideal parallel loop execution time (T_{p_ideal}) on single-cluster multiple-processor configurations (i.e., ≤ 8 processors) can be estimated as shown below:

$$T_{p\_ideal} = \frac{(T1_{mc} + T1_{sx})}{par\_concurr}$$

where *par_concurr* is the average parallel loop concurrency on the single-cluster, $T1_{mc}$ is the time taken to execute the **main** cluster-only parallel loops on the 1-processor configuration, and $T1_{sx}$ is the time taken to execute the **s(x)***doall* parallel loops on the 1-processor configuration. For multicluster configurations, the main cluster-only loops are executed only by the processors in the cluster executing the main task, where as the *s(x)doall* loops are spread across all clusters. Taking this into account, the ideal parallel loop execution time on multicluster configurations can be estimated as follows:

$$T_{p\_ideal} = \frac{(T1_{mc})}{par\_concurr\_main} + \frac{(T1_{sx})}{par\_concurr\_total}$$

where *par_concurr_main* is the average parallel loop concurrency on the main cluster and *par_concurr_total* is the sum of the the average parallel loop concurrency on all the clusters in the configuration.

The actual time taken for parallel loop execution, T_{p_actual} can be easily obtained from the program event traces collected by *cedarhpm*. Therefore, the percentage of the completion time of an application attributable to the global memory and network contention overhead, Ov_{cont}, can be estimated as shown below:

$$Ov_{cont} = \frac{(T_{p\_actual} - T_{p\_ideal})}{CT} * 100$$

The actual time spent on parallel loop execution T_{p_actual}, the ideal parallel loop execution time T_{p_ideal}, and the percentage of the application completion time attributable to the global memory and network contention overhead Ov_{cont}, are provided in Table 4 for the different Cedar configurations.

Program		1 proc	4 proc	8 proc	16 proc	32 proc
FLO52	T_{p_actual} (s)	574	185	118	68	37
	T_{p_ideal} (s)	-	148	79	45	22
	Ov_{cont} (%)	-	17	27	24	21
ARC2D	T_{p_actual} (s)	2067	545	300	160	94
	T_{p_ideal} (s)	-	525	270	139	74
	Ov_{cont} (%)	-	3.4	8.8	10.3	14.1
MDG	T_{p_actual} (s)	4800	1228	643	330	178
	T_{p_ideal} (s)	-	1212	616	305	151
	Ov_{cont} (%)	-	1.3	4.1	7.2	13.4
OCEAN	T_{p_actual} (s)	2647	701	360	195	133
	T_{p_ideal} (s)	-	675	336	177	120
	Ov_{cont} (%)	-	3.5	6.3	8.0	7.4
ADM	T_{p_actual} (s)	663	171	89	51	43
	T_{p_ideal} (s)	-	167	84	46	33
	Ov_{cont} (%)	-	1.9	4.1	5.9	12.5

Table 4: GM and Network Contention Overhead

From the table we see that the contention overhead forms a substantial portion of the application completion time on the multiprocessor configurations. For the application FLO52 the global memory and network contention overhead accounts for as much as 17-27% of the completion time on the multiprocessor systems. For the other applications the contention overhead is lower, but increases with the number of processors in the system and is over 10% on the full 32-processor Cedar system.

To summarize, we have used a simple (although indirect) methodology to capture the overall effect of the global memory and network contention overhead, i.e., its contribution to the application completion time.

8 Summary and Conclusions

In this paper, a real-measurement based methodology has been used to characterize the operating system overheads, runtime system parallelization overheads and the overhead due to global memory and interconnection network contention for the Cedar shared-

memory multiprocessor. The overheads were measured for a range of Cedar configurations from 1 processor to the full 4-cluster/32-processor configuration, thus characterizing the effect of scaling the number of processors on the overheads.

For the 4-cluster Cedar, the operating system overhead constitutes 5-21% of the total completion time of an application. Kernel lock contention is found to be negligible (kernel lock spin-wait time is $< 1\%$ of the completion time). Context switching, servicing cross processor interrupts, handling concurrent and sequential page faults, and accessing (cluster) critical sections/resources are found to be the major components of the operating system overhead.

On the 4-cluster Cedar, the parallelization overhead accounts for 10-25% of the completion time for the main cluster task and 15-44% of the completion time for the helper tasks. Exploiting the clustering hardware is found to be helpful in reducing the global memory and network traffic, and hence the overhead. The hierarchical parallel loop construct that exploits clustering during the parallel work distribution phase incurs little overhead. But, the parallel loop distribution overhead is as high as 6-10% of the application completion time for the flat parallel loop construct that views Cedar as a machine of independent processors (rather than as clusters of processors) during the loop iteration distribution phase. The other major component of the parallelization overheads is multicluster barrier synchronization which contributes as much as 7-16% of the completion time.

A simple methodology is used to quantify the contribution of the global memory and network contention overhead to the application completion time. For the 4-cluster Cedar, the contention overhead is found to account for 8-21% of the application completion time.

Thus, we see that the various overheads contribute as much as 30-50% of the completion time for the different applications. Although the numbers are specific to the system studied, by characterizing the overheads from runtime system, operating system and hardware perspectives in one place, this study brings home the importance of the overhead activities in multiprocessor systems.

Acknowledgements

The authors would like to thank the CSRD staff for their generous help. In particular, we thank Prof. Pen Yew and Tom Murphy for their help with *cedarhpm*, Prof. Perry Emrath for discussions on the Xylem Operating System, Prof. Rudolf Eigenmann for clarifications on the Perfect Benchmark programs, and Jay Hoeflinger for discussions on the runtime library and the XDOALL construct.

References

[1] K. Gallivan, et al., "Experimentally characterizing the behavior of multiprocessor memory systems," *Proc. of Conf. on the Measurement and Modeling of Computer Systems (SIGMETRICS)*, vol. 18, no. 1, 1989.

[2] R. Fatoohi, "Vector performance analysis of the NEC SX-2," *Proc. of ACM Int'l. Conf. on Supercomputing*, pp. 389–400, June 1990.

[3] P. F. Koss,"Application performance on supercomputers," *Tech. Rpt. 847, CSRD, Univ. of Illinois*, Jan. 1989.

[4] R. H. Saavedra-Barrera,"Machine characterization and benchmark performance prediction," *Tech. Rpt. UCB/CSD 88/437, Univ. of California at Berkeley*, June 1988.

[5] D. Lenoski, et al., "The DASH prototype: logic overhead and performance," *IEEE Trans. on Parallel and Distributed Systems*, vol. 4, no. 1, Jan 1993.

[6] J. Torrellas, A. Gupta, and J. Hennessy, "Characterizing the caching and synchronization performance of a multiprocessor operating system," *Architectural Support for Programming Languages and Operating Systems*, pp. 162–174, 1992.

[7] D. Clark, "Cache performance of the Vax-11/780," *ACM Trans. on Computer Systems*, vol. 1, no. 1, pp. 24–37, 1983.

[8] D. Clark, P. Bannon, and J. Keller,"Measuring Vax-8800 performance with a histogram hardware monitor," *Proc. 15th Annual Int'l. Symposium on Computer Architecture*, Honolulu, Hawaii, pp. 176–185, 1988.

[9] D. Kuck, et al., "The Cedar system and an initial performance study," *Proc. 20th Annual Int'l. Symp. on Comp. Arch.*, pp. 213–223, May 1993.

[10] J. Konicek, et al., "The organization of the Cedar system," *Proc. Int'l. Conf. on Parallel Processing*, pp. 49–56, 1991.

[11] P. Emrath, "An operating system for the Cedar multiprocessor," *IEEE Software*, pp. 30–37, July 1985.

[12] M. Berry, et al., "The Perfect Club Benchmarks: effective performance evaluation of supercomputers," *Intl. Journal of Supercomputing Applications*, vol. 3, no. 3, pp. 5–40, 1989.

[13] R. Eigenmann, et al., "The Cedar Fortran Project," *Tech. Rpt. 1262, CSRD, Univ. of Illinois*, 1992.

[14] J. B.Andrews, "A hardware tracing facility for multiprocessing supercomputer," *Tech. Rpt. 1009, CSRD, Univ. of Illinois*, May 1990.

[15] G. F. Pfister and V. A. Norton, "Hot spot contention and combining in multistage interconnection networks," *IEEE Trans. on Computers*, vol. 34, no. 10, pp. 943–948, Oct. 1985.

[16] P. Yew, N. Tzeng, and D. H. Lawrie, "Distributing hot-spot addressing in large-scale multiprocessors," *IEEE Trans. on Computers*, vol. 36, no. 4, pp. 388-395, Apr. 1987.