Instrumentation for Future Parallel Computing Systems

Edited by

Margaret Simmons
Rebecca Koskela
Ingrid Bucher

ACM Press
New York, New York

Addison-Wesley Publishing Company
The Advanced Book Program
Redwood City, California • Menlo Park, California • Reading, Massachusetts
New York • Don Mills, Ontario • Wokingham, United Kingdom • Amsterdam
Bonn • Sydney • Singapore • Tokyo • Madrid • San Juan

1

Multiprocessor Instrumentation: Approaches for Cedar

Allen D. Malony[1]

1.1 Introduction

Parallel systems pose a unique challenge to performance measurement and instrumentation. The complexity of these systems manifests itself as an increase in performance complexity as well as programming complexity. The complex interaction of the many architectural, hardware, and software features of these systems results in a significantly larger space of possible performance behavior and potential performance bottlenecks. Programming parallel systems requires that users understand the performance characteristics of the machines and be able to modify their programs and algorithms accordingly. The instrumentation problem, therefore, is to develop tools to aid the user in investigating performance problems and in determining the most effective way of exploiting the high performance capabilities of parallel systems.

This paper gives observations on the parallel system instrumentation problem in the context of the Cedar multiprocessor. The Cedar system integrates several architectural, hardware, and software concepts for parallel operation.

[1]This work was supported in part by NSF Grant Numbers NSF MIP–8410110 and NSF DCR 84–06916, DOE Grant Number DOE DE–FG02–85ER25001, the Air Force Office of Scientific Research Grant Number AFOSR–F49620–86–C–0136, and a donation from IBM.

The combination makes Cedar a particularly interesting machine for investigating instrumentation issues and developing prototype tools. The different needs for performance evaluation on the Cedar machine define the instrumentation requirements. The implementation of instrumentation tools, however, involves tradeoffs in design, resolution, and accuracy, and must be weighed against the payoff in better performance evaluation. This discussion of instrumentation tools targeted for Cedar considers these tradeoffs.

The following presentation is somewhat historical in that it describes the tools in the order in which they were developed. It is important to understand that these tools are targeted to an actual machine and therefore might lack certain instrumentation sophistication possible in less restrictive environments, as in the case of simulation. Nevertheless, to develop good instrumentation techniques for generating detailed performance data of parallel system execution, it is instructive to study prototype tools designed within the constraints placed by real parallel machines.

One of the goals of the performance evaluation activities of the Cedar project is to build a prototype of a *performance instrumented computer* [1,2] (see Figure 1.1). The concept is one of a standard computer system that contains additional performance measurement hardware and software. The additional hardware allows easy access to performance-critical information in the machine (e.g., cache misses, memory conflicts); it allows measurements of these points to be easily made in response to various triggers, and it allows easy storage of selected results. The additional software allows a user to specify which measurements to make; it allows the user to observe performance results and store these results in a database of performance information.

1.2 The Cedar System

The Cedar system of the University of Illinois is characterized by the hierarchical organization of both its computational capabilities and memory system [3,4]. It consists of multiple clusters, each of which is a multivector processor comprising eight computational elements (CEs) (see Figure 1.2). Parallelism can be exploited at three levels. Within each CE, operations on vectors can be done in vector mode. Each cluster is a tightly coupled multiprocessor that can exploit fine grained parallelism through loop-level concurrency. Finally, multiple clusters can be used for medium and large grain parallelism, as well as extended forms of fine grain parallelism. Presently, each cluster is a modified Alliant FX/8.

The memory organization is hierarchical as well, with communication increasing in cost at each level. At the lowest level, each CE has a set of private scalar and vector registers. The next two levels, a cache and a cluster memory, are shared by the CEs within the same cluster. Finally, all clusters have access

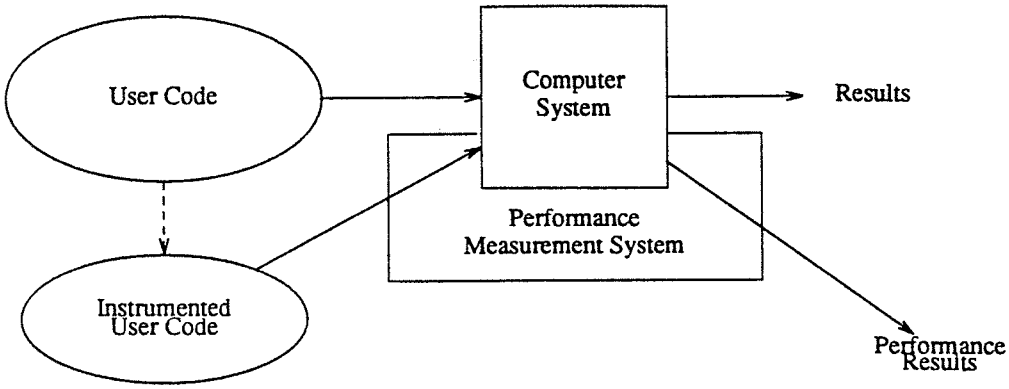


FIGURE 1.1
Performance instrumented computer.

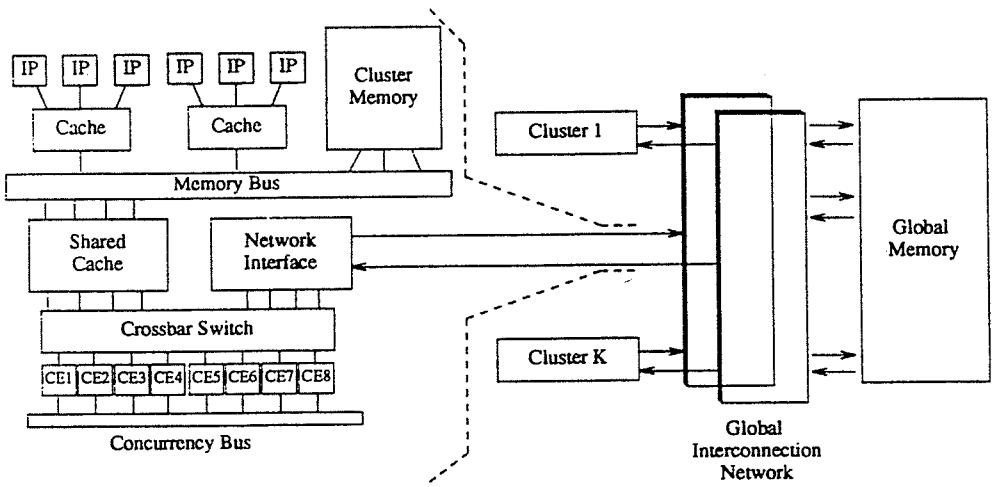


FIGURE 1.2
The Cedar system organization.

to a large global memory. This global memory is accessed through two unidirectional switching networks, one for downloading data from memory, the other for uploading. These switching networks are 2-stage omega networks built from 8×8 crossbar switches.

The Cedar operating system, Xylem, is a modification of Alliant's Concentrix operating system extended for multitasking and virtual memory management of the Cedar memory hierarchy [5,6]. A Xylem process consists of one or more *cluster tasks*. Multiple cluster tasks execute asynchronously across the Cedar system. Xylem provides system calls for starting and stopping tasks and waiting for tasks to finish. System calls are also provided for coarse-grained intertask synchronization. In addition to multitasking, Xylem supports multiprogramming whereby multiple processes can be executing simultaneously. The Xylem virtual memory system provides convenient access to the Cedar physical memory hierarchy.

Fortran is the focus of language and compiler development for Cedar. Cedar Fortran is derived from Alliant FX/Fortran with extensions for memory allocation, concurrency control, multitasking, and synchronization [7]. New data type specification statements reflect the Xylem memory access and locality structure. Vector concurrency is available through array section notation, conditional vector statements, and vector reduction functions. DOALL and DOACROSS constructs specify parallel execution of loop iterations on processors within a single cluster task or spread across multiple cluster tasks. Multitasking routines provide an interface between Cedar Fortan and Xylem for task creation and control. A set of synchronization functions allows access to the Cedar hardware synchronization primitives. Cray-style synchronization operations are also provided. Multitasking and synchronization routines are implemented as part of a Cedar Fortran run-time library [8]. Compiler optimizations for vectorization, parallelization, and memory allocation are also being developed for the Cedar machine.

Due to the complexity of Cedar, a sophisticated performance analysis system of integrated hardware and software tools to collect, present, and analyze performance data is imperative if high performance is to be achieved across a wide range of scientific applications. Such a system is being designed and implemented and an overview of its capabilities is discussed in [9]. The following sections give details of the instrumentation tools developed thus far.

1.3 Cluster Concurrency Instrumentation

The first Cedar performance instrumentation tool implemented was a tool to measure the average number of processors physically active during a computation on an Alliant FX/8 cluster [10-12]. To better characterize a parallel program's execution on the FX/8 it was important to determine the degree of

physical parallelism actually being achieved. A basic performance efficiency metric, *concurrency efficiency* or CEFF, can be derived from measuring the amount of time i processors are active, where $i = 1, n$ and n is the total number of processors available. CEFF indicates the percentage of available physical parallelism being used by the program. It is an interesting performance metric in that a bound on the maximum speedup possible on a cluster for the measured program run can be obtained.

1.3.1 Concurrent Operation on an FX/8 Cluster

A program that has been compiled for concurrent operation on the FX/8 will be allocated the entire computational complex. All processors are thus available for use by the program during its execution. [1] As the program advances through periods of sequential and concurrent operation, the number of active processors will change. Measurements of how the program's execution time was spent in the different levels of physical parallelism must be made to calculate the CEFF metric.

1.3.2 CEFF Analysis

If T_i is the amount of time a program spends executing with i processors active, where $i = 1, n$ and n is the total number of processors, concurrency efficiency is defined as

$$CEFF = \left(\frac{\sum_{i=1}^{i=n} i * T_i}{n * T} \right) * 100\%$$

$$where\ T = \sum_{i=1}^{i=n} T_i$$

Given the concurrency timing information, T_i, it is simple to derive *concurrency utilization* results, CU_i, as the percentage of time i processors are active:

$$CU_i = \frac{T_i}{T} * 100\%$$

The CEFF metric indicates the average percentage of the available parallel processing resource used by the program. The CU values give a breakdown of execution time spent in each concurrent execution state. [2] The *average concur-*

[1] The same is true for a Xylem task that runs concurrently on a cluster.

[2] A concurrent state is defined for each possible number of active processors. Concurrent state i is the state where only i processors are active.

rency, defined as $CAVG = n * CEFF$, gives the average number of processors active as well as an upper bound on program speedup possible for this run. It is an upper bound because active processors may not be directly contributing to the overall program progress; such is the case with synchronization operations. Whereas a low CEFF value implies a low level of concurrent processor activity and, therefore, a poor speedup performance, a high CEFF value is only an indication of high processor concurrency and does not necessarily reflect good parallel performance. CEFF and CU values must be considered with other performance metrics to determine the degree of *effective parallelism* being achieved.

1.3.3 CEFF Implementation

Ideally, changes in the number of active processors should be detected to measure the time spent in the different concurrent states. However, detecting changes in concurrent state is difficult on the FX/8 because it occurs at the instruction level. One alternative considered for the Cedar system was to build a special hardware monitor to look at processor activity signals. This had several drawbacks including signal accessibility, design complexity, and the ability to filter the timing data on a per process basis. The second alternative was to instrument the object code to monitor the instructions that resulted in concurrency state changes and to keep software time measurements. Although exact timing could be maintained by this approach, it was intrusive and its implementation required a more complete tracing facility (see Section 1.4).

A desirable implementation would be easy to design and build, would minimally affect program operation, but would give reasonably accurate CEFF statistics. The approach taken was based on a sampling technique commonly used for profiling. Concentrix was modified to implement the CEFF measurements.

When concurrency efficiency measurements are enabled, the program is interrupted every 10 msec and the state of each processor in the computational complex is sampled. It is possible to determine if a CE is inactive when the program is interrupted by comparing the CE's program counter to a known *idle* value. The total number of active CE's, i, is determined with each interrupt and a counter associated with each concurrent state, N_i, is incremented. The N_is are set to zero at the beginning of the program.

At the end of the program's execution, the time spent in concurrent state i, T_i, is calculated by multiplying the ith concurrency count value, N_i, by 10 msec. The CEFF and CU values can then be easily computed as shown in Section 1.3.2.

From an implementation standpoint, maintaining the information necessary to calculate concurrency efficiency is simple and cheap. Only eight concurrency state counters are needed for the FX/8. These counters can easily be placed in the user's process structure along with the other timing and profiling information.

Determining the active processors and incrementing the appropriate N_i counter is the only real-time processing required.

1.3.4 CEFF Results

CEFF results can be produced for a program's entire execution or for user-selected program sections. The results produced include T_i, CU_i, T, $CAVG$, and $CEFF$. An example of the output is shown in Table 1.1.

There are several things to consider when interpreting the CEFF results. The timing measurements assume that the current process was executing throughout the last 10-msec time interval. Because of the 10-msec sampling procedure, the concurrent state timing data is only a statistical approximation to the actual concurrency timing information. Furthermore, determining CE cluster utilization from a single measurement made every 10 msec is prone to errors because the number of CEs used by the program can change many times during a 10-msec time interval.

It is important also to remember that the concurrency efficiency results only represent measurements of physical processor activity. No analysis is made of what the processors are actually doing when they are active. Thus, the CEFF results should not necessarily be interpreted as effective parallelism.

CEFF results can be used with other measurements to better characterize program performance. For instance, speedups from 1 processor to n processors can help to clarify the effective parallelism. Suppose a program running on eight processors, as opposed to one, achieves a speedup $S = 6$ and a CEFF value of 80% ($CAVG = 6.4$). Although only 80% of the processors are utilized on average, almost all of the average processor concurrency is being used effectively. In this case, the user might conclude that the ability to keep more processors active is the problem. However, $S = 2$ for a program with $CEFF = 80\%$ indicates a low effective parallelism, likely due to synchronization overhead or a large sequential component.

The CU measurements are interesting because they give a histogram of concurrent activity. The CU_i values where $i < n$ are important because they represent periods of reduced parallelism when processors are actually idle. The value CU_1 is most important since it is the percentage of time the program is executing sequentially. The CU_1 value can be plugged directly into Amdahl's equation to get the projected maximum program speedup for p processors. [3] For the results produced by CEFF above:

[3]Amdahl's equation is defined as $\lim_{p\to\infty} S_p = \frac{1}{1-F_p}$ where F_p is the fraction of time all p processors are active. We assume that the percentages of all concurrent activity are summed to get F_p. Thus, the calculated asymptotic speedup is actually optimistic.

TABLE 1.1
CEFF Results

Number of active CEs	*Seconds*	*Concurrency (%)*
1	3.37	28.25%
2	0.39	3.27%
3	0.30	2.51%
4	0.51	4.28%
5	0.75	6.29%
6	1.02	8.55%
7	1.49	12.49%
8	4.10	34.37%
avg. active CEs	total seconds	efficiency
5.05	1.93	63.07%

$$\lim_{p\to\infty} S_p = \frac{1}{\frac{CU_1}{100\%}} = \frac{1}{.2825} = 3.54$$

Although $CAVG = 5.05$, the asymptotic speedup is limited by the significant sequential component.

1.4 Timing Instrumentation

After the implementation of the CEFF utility, it was clear that a more extensive overall process timing utility was required for Cedar. There are two main issues in measuring process execution time on a high-performance multiprocessing system: the time measurement accuracy and the timing of concurrent operation. The need for more accurate time measurements increases as system performance increases because of the finer time granularity of events of interest. Measurement techniques based on periodic sampling of the system state are insufficient for a high-performance multiprocessing environment. Strategies to directly measure execution time using a high-resolution real-time clock provide more accurate and detailed execution time measurements. Moreover, tasks can be in various states of execution requiring a more detailed breakdown of execution time measurements [13,14].

A timing utility, called HRTIME, has been implemented for the Cedar system [15]. The goal of the HRTIME utility is to provide high-resolution, detailed timing measurements of parallel operation of multitasked Xylem pro-

cesses. HRTIME gives a complete timing account of both execution and non-execution task states. In addition, HRTIME provides individual processor timing measurements to give a detailed account of the time spent in various states of sequential and concurrent execution.

1.4.1 HRTIME Motivation

HRTIME addresses several shortcomings of the standard UNIX approach to process timing found on most multiprocessor systems and exemplified by Concentrix. When HRTIME was implemented, Concentrix was using the UNIX sampled execution time technique. More accurate timing can be gained through measured execution time. Briefly, measured execution time is based on determining the elapsed time, spent in a particular process state by recording the value of a high-resolution real-time clock when the state begins and when it ends. The difference between the two recorded time values is the elapsed time which is added to a total execution time kept for that process state. Measured execution time forms the basis for HRTIME and recently has been incorporated by Alliant into Concentrix.

However, Concentrix only records a single USER and SYSTEM time value for each process; for concurrent operation, HRTIME should keep separate time values for each processor used by a program. Additionally, the operating system (OS) instrumentation required to measure execution time allowed the execution state types to be extended as well as nonexecution states to be monitored. These features have been included in the HRTIME design.

Finally, Cedar requires a timing facility that maintains time measurements for a multitasked Xylem process. In particular, there needs to be a way to record a single global process time and to time various levels of task concurrency. HRTIME provides these mechanisms as well.

1.4.2 HRTIME Design

HRTIME is based on measured task timing with all times measured at 10-μsec resolution [13,14]. The Concentrix USER and SYSTEM process states are extended to four task execution states, and the measured times spent in these states are kept on a per-processor basis. Nonexecution states are also defined to track the time a task spends ready, blocked, or idle.

Execution States. Generally, task execution states can be partitioned according to the type of code being executed. Four execution states are defined by HRTIME: USER, SYSTEM, OVERHEAD, and KERNEL. The USER state is active when user code is being executed. Processing of system calls occurs in the SYSTEM state. Interrupt processing that can be attributed to the current

task falls into the OVERHEAD state; this includes interrupts for page faults and general exceptions, such as a floating point exception. Interrupts not directly associated with the current task are processed in the KERNEL state; cross-processor interrupts, device interrupts and timer interrupts are considered part of the KERNEL state. Actually, the KERNEL state is not a task execution state at all, but a state in which the operating system itself is executing. For this reason, only the USER, SYSTEM and OVERHEAD states are timed for a task.

In a multiprocessor system such as Cedar, it is possible for a task to be executing sequentially on one processor or concurrently on several processors. To further audit execution time, HRTIME keeps time measurements on a processing resource basis. In Cedar, a sequential task executes on an interactive processor (IP), a detached CE, and/or one CE of a cluster. HRTIME maintains a USER, SYSTEM, and OVERHEAD timer for each of these sequential processing resources as part of the overall task time measurements.

All concurrent tasks execute on the computational complex of the Alliant FX/8. [4] However, it is possible for processors participating in a concurrent computation to be in different states of execution; e.g., CE 0 is in USER mode while CE 3 is in SYSTEM mode and CE 6 is in OVERHEAD mode, and so on (see Table 1.2).

For this reason, the execution states should really be monitored at the processor level. For concurrent tasks, HRTIME measures execution times per processor per state. [5] USER, SYSTEM, and OVERHEAD timers are therefore defined for each of the eight CEs that can participate in a cluster task's execution.

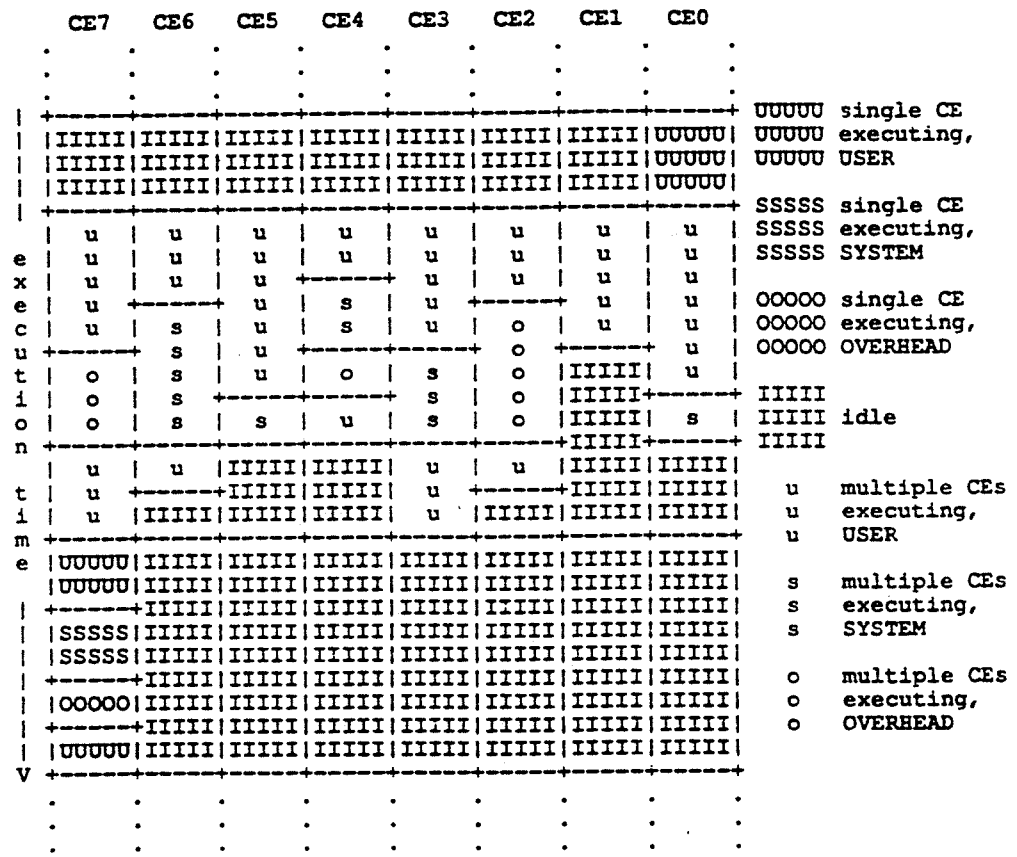
Although the execution state timers defined above give detailed timing information, a complicated calculation must be made to determine total elapsed time, especially in the case of a concurrent task. For this purpose, a VIRTUAL execution state is defined; when any processing resource is executing in USER, SYSTEM, or OVERHEAD state, the task is in a VIRTUAL execution state and a VIRTUAL time value is being updated.

As mentioned before, Xylem supports multitasking of a process for parallel execution across multiple Cedar clusters. Like the VIRTUAL timer for a Xylem task, a Xylem process will also have a process virtual timer, P_VIRTUAL. The motivation for a Xylem process virtual timer is to determine total elapsed execution time for an entire process. The update mechanism is the same as for the task virtual timer except that it is based on when tasks are executing. When any Xylem task is executing in USER, SYSTEM, or OVERHEAD state, the Xylem process is in P_VIRTUAL execution state and a P_VIRTUAL time value is being updated.

[4] A task is said to be concurrent if it requires more than one CE during its execution.

[5] This is not done by Concentrix. Whenever Concentrix detects any processor to be in SYSTEM state, the whole process is considered to be in SYSTEM state. USER time accumulates only when all processors are in USER state.

TABLE 1.2
Concurrent Execution States



Nonexecution States. In addition to the execution states, three nonexecution task states are recognized by HRTIME: READY, BLOCKED, and IDLE. When a task is ready to execute, but not currently running, it is in the READY state. Similarly, a task is in the BLOCKED state when it is blocked from execution. The IDLE state occurs when a task is waiting for some work to do. Because the task is not executing, only one timer is needed for each of these non-execution states.

1.4.3 HRTIME Use

The HRTIME utility is enabled for all processes. Xylem maintains the time measurements as part of each process's state. The state timers are stored as 64-bit integer values indicating the number of 10-μsec time units measured. The timer data structures are allocated as part of a larger process measurement structure in the process's read-only address space. This allocation allows reference to any of the timers directly from the user's program.

The HRTIME utility allows a user to make timing measurements for selected sections of a program as well as for the entire program [15]. The general procedure for making time measurements of a section of a program task is shown below:

1. Read HRTIME measurements for the current task
2. Execute program section
3. Read a second HRTIME measurement sample
4. Calculate the time differences between samples

The time required to execute the program section is the difference between the two HRTIME samples taken before and after the program section. If the time samples are saved, a time-sample trace can be kept during program execution and a post processor used to calculate the desired incremental time values.

In some cases, the user will want to time a program as a whole. The *hrtime* command will time a program and produce HRTIME results for all program tasks. An example of the HRTIME output for one task of a multitasked parallel program running under Xylem is shown in Table 1.3.

1.4.4 Interpreting HRTIME Measurements

HRTIME provides significantly more detailed execution timing information than the standard Concentrix USER and SYSTEM times. Nonexecution times are also generated. Interpreting the time measurements, however, requires some understanding of the program's operation.

TABLE 1.3
HRTIME Results

```
PROCESS/TASK TIME
process virtual :      5.89677
task virtual    :      5.89677
not ready       :      0.18924
ready           :      1.64093
idle            :      0.10224
-------------------------------------------------------
IP, DETACHED CE, and CLUSTER TIME
                        User        System      Overhead
ip              :       0.00000     0.00000     0.00000
dce             :       0.00105     0.05764     0.01262
cluster         :       0.00000     0.00000     0.00000
-------------------------------------------------------
CE TIME
                        User        System      Overhead
ce [0]          :       5.73250     0.00092     0.00737
ce [1]          :       5.73529     0.00000     0.00335
ce [2]          :       5.73540     0.00000     0.00393
ce [3]          :       5.73318     0.00416     0.00863
ce [4]          :       5.73358     0.00000     0.00436
ce [5]          :       5.73354     0.00000     0.00264
ce [6]          :       5.73285     0.00000     0.00288
ce [7]          :       5.71049     0.02005     0.00373
```

One aspect of HRTIME that might be curious is the definition of USER, SYSTEM, and OVERHEAD states. Some system processing actually occurs on behalf of the user and, therefore, should be measured separately from the overhead of general OS operations. Separate measurements let the user know how much overhead processing the program really experiences during its execution as well as how much system support the program requires. By monitoring the OVERHEAD times, as well as the USER and SYSTEM times, the user can get a sense of how vulnerable the program's performance is to the overhead processing.

Nonexecution time measurements might be regarded as superfluous. However, these times can reflect interesting task operation behavior. For instance, the READY time is a good indication of the amount of waiting a task experiences in the scheduling queue. The BLOCKED time is more versatile in that it encompasses all dependent waiting time encountered by the task. This time not only includes blocking due to I/O operations but also represents waiting due to intertask synchronization. IDLE time is particularly interesting for Xylem tasks because it can be used to compute a task utilization metric indicating the percentage of time a task executes some portion of the user's program.

For the most part, the execution time measurements are well defined. The complication comes when trying to work back from the measurements to what the program is actually doing. For sequential, single-task programs, the HRTIME measurements are easy to understand. In this case, the breakdown across the different sequential processing resources is interesting because it shows how the process was scheduled during its execution.

The HRTIME measurements for concurrent tasks are more difficult to understand. The goal of the CE execution time measurements is to give some indication of CE resource usage. Ideally, a single global state space is defined where each point describes a different combination of the CE execution states. Time spent in each global state can then be measured. However, the implementation of this measurement model is impractical because of the complex instrumentation needed to detect global state changes.

Using the individual CE measurements, it is difficult to determine the amount of time CE 0 is in USER state when CE 1 is in SYSTEM state, and so on with other CE state combinations. However, it is unclear whether such time measurements have much value. Because the computational complex is assigned to a task as a single resource, it is more important how the individual CEs themselves are utilized. The HRTIME measurements show this as a breakdown between execution states for each CE.

Finally, the Xylem process virtual time, in conjunction with the individual task timings, can be used to give a general impression of the level of parallel task operation. A possible addition to the current HRTIME utility would be the breakdown of the P_VIRTUAL time into values reflecting different levels of simultaneous task execution.

1.5 Profiling Instrumentation

Timing alone is insufficient for characterizing the behavior of parallel programs. Although the HRTIME utility is able to describe how a program spends its time in different execution states on different processing resources, it is unable to correlate the timing data with the code being executed. The approach that has been taken in most multiprocessor systems is to adapt sample-based profiling tools designed for sequential programs. This approach was initially attempted for Cedar but was abandoned because of its fundamental limitations for parallel program performance characterization and its implementation complexity [16,17]. The following describes the conclusions from this investigation that are believed to be generally applicable to other parallel systems.

1.5.1 Standard Sequential Profiling

The goal of program profiling is to provide an accurate characterization of a program's execution behavior and performance. Such information will help the user evaluate alternative implementations and guide program optimization. Two measurements are commonly defined for the profiling of sequential programs: (1) counting the number of times routines are executed and (2) timing the execution of routines. Focusing on routines is reasonable for sequential program profiling. Because only one routine can be executing at any time, the characterization of a routine's execution in terms of call counts and execution times is a direct measure of its individual performance and its relative importance to the overall computation.

The standard profiling tools of the UNIX operating system are *prof* and *gprof* [18]. Two types of profiling output are produced by these tools. The *flat* profile shows all routines called during program execution with the count of the number of times they were called and their *direct* execution time. [6] The *call graph* profile lists each routine, together with information about its parent routines and children routines. The flat profile results are augmented with *cumulative* time for the routine, the number of calls to each descendant, the time inherited from each of its descendants, and the fraction of total routine time represented by the descendant's times. [7] Similar results are shown for the parents of the routine.

Timing in *prof* and *gprof* is based on sampled execution time. When profiling is enabled, a histogram of the location of the profiled program's program

[6] The direct execution time for a routine is amount of time spent executing the statements of the routine.

[7] The cumulative routine time is the elapsed time from routine entry to exit.

counter is updated at the end of each interval timer interrupt. [8] Routine execution times are determined from a distribution of program counter samples within the histogram. To determine a routine's direct execution time, the PC histogram counts for that routine are summed and multiplied by the interval timer period. Obviously, such timing measurements are subject to statistical sampling errors and have the potential for giving misleading results.

To determine cumulative execution times, the arc call counts are used to calculate the amount of time that should be propagated from descendants to ancestors in the dynamic call graph. Having determined a descendant's cumulative execution time, each ancestor is propagated a time equal to the fraction of total calls to the descendant made by the ancestor times the descendent's cumulative time. The necessary, but possibly incorrect, assumption is that each call to a routine takes the same amount of time. This assumption, coupled with the statistical approximation of direct execution time, can produce invalid timing measurements.

1.5.2 Parallel Profiling

The goal of traditional profiling tools is to optimize the performance of a program by streamlining routines that are major consumers of execution time. Using the routine call counts and execution times, iterative techniques can be applied to integrate excessively called routines or to streamline routines that are execution time bottlenecks. However, parallel program profiling calls for an extension to the common profiling approaches to include measurements of the dynamic interaction between concurrent execution threads. Unfortunately, there are certain fundamental problems that limit such an approach.

The proposed profiling strategy for Cedar was to extend sequential profiling techniques to gather additional information about parallel program activity. In particular, because parallel program execution implies the potential for more than one routine to be executing concurrently, the standard profiling measurements were to be enhanced to include information about the parallelism present when a routine was executing [16,17].

The first conclusion reached is that sampling is totally inappropriate for generating profile timing information for parallel programs. The reasons for this are the same as for sample-based process timing. Furthermore, the assumptions made about achieving statistical accuracy and propagating time back up the calling tree to determine cumulative execution times are invalid for sample-based parallel program profiling. The main reason for this is the inability of sampling to capture changes in parallel execution state that directly affect how

[8] The interval time interrupt usually occurs every $\frac{1}{60}$th of a second. On the Alliant FX/8, it occurs every 10 msec.

time intervals should be classified. Parallel profiling approaches must instead be based on measuring time intervals between successive routine entry/exit events and events reflecting changes in concurrency state.

Unfortunately, many additional execution events are required to capture even basic concurrent activity. These events represent an enumeration of the possible concurrent states that might occur during a program's execution. More importantly, the events do not necessarily occur at routine entry and exit. Thus, a significant amount of additional code instrumentation would be required.

Other timing problems were also identified. In sequential profiling, it is always clear how the current time should be accounted. This is not the case in parallel profiling. Parallel execution necessarily implies that one or more routines are active simultaneously. Several issues arise with respect to accounting execution time to routines in different parallel execution cases:

1. When a routine is executing concurrently with itself, how are direct time and cumulative time for that routine measured?
2. When a routine, B, is called concurrently from the same calling routine, A, how does B's parallel execution time get accounted in A's timing values?
3. When a caller and callee routine are executing concurrently, how are direct and cumulative times being accumulated for the caller and the callee?
4. When there is a concurrent execution overlap of two callee routines (different or the same) from the same caller, how is the overlap time accounted for in the caller's time values?

The root of these issues lies in the definition of execution time. If execution time is to mean *elapsed* time, execution time for a routine A accumulates whenever A is executing, sequentially or concurrently. If, however, execution time means *CPU* time, the time spent on different concurrent execution threads must be accounted for in routine A's execution times. Elapsed execution time measurements are necessary for calculating speedup. On the other hand, CPU time accounts for the amount of computing resources used by the program and is necessary for utilization calculations. Both time values are needed for profiling parallel programs.

The most important conclusion reached concerns profiling as a basis for parallel program optimization. The goal of sequential profiling is to find the routines that take the most time and optimize them. Doing so will directly improve the overall performance of the sequential program. This profile-based optimization strategy is faulty for parallel programs as shown by the simple example in Figure 1.3 for a parallel program running on two processors. The Main routine forks a thread that executes routine A on processor P1 for 10 seconds. The other fork of Main calls routines B, C, D, and E, in that order. Each of these routines executes concurrently with A for 2.5 seconds. The forks then join and the program ends.

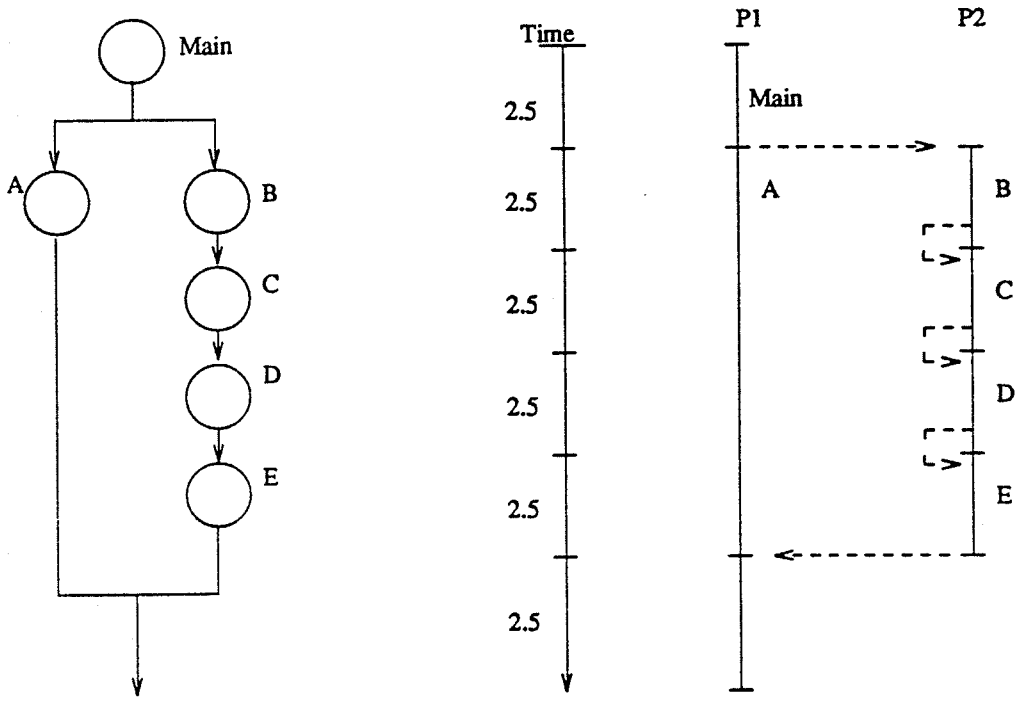


FIGURE 1.3
Parallel program example.

The standard optimization strategy based on profiling results would be to optimize A. However, there are several situations in which this strategy would be incorrect. For instance, suppose there existed dependencies between the routines as shown by the dashed lines. Clearly, A is not the bottleneck since B, C, D, and E have an execution flow dependency. Optimizing A would have little effect on run time of the program. Instead, the programmer should concentrate on improving the performance of the other routines.

The problem is that the profiling data does not make the needed optimization obvious. In order to optimize parallel programs, it is necessary to observe the dynamic interaction of the multiple threads of execution. A parallel profiling approach must describe and measure all the possible parallel execution interaction that might occur as individual events. The instrumentation required to do this would be overly complex to implement.

The instrumentation complexity required for parallel profiling does not pay off in better characterization and optimization. Profiling only summarizes parallel execution information in terms of event counts and times. Unfortunately, it is the dynamic execution information that is required for effective parallel program performance evaluation. Statistical summaries are interesting for some characterization purposes, but parallel program analysis requires the ability to observe and analyze time-ordered concurrent events. A different instrumentation approach is required to achieve this result.

1.6 Tracing Instrumentation

The standard profiling instrumentation approach proved too limited in its ability to measure and characterize parallel execution. Instead, what was needed was a general instrumentation approach simple enough to be efficient but robust enough to capture execution data that could describe complex parallel program behavior. Program event tracing was implemented as an instrumentation technique for performance evaluation of parallel programs written for Cedar [19]. The versatility of tracing comes from the ability to combine low-level primitive event traces to produce information about more complex higher-level events. In the case of parallel program analysis, this is a necessary requirement because of the difficulty of monitoring complex parallel execution states at run time.

1.6.1 The Tracing Approach

The execution of a program can generally be described as a time-ordered sequence of events. The events can be defined to be any logical or physical consequence of program execution. The goal of program performance evaluation is to capture information about these events in meaningful ways that can be used to guide performance optimization.

Three operations can be identified in this process: (1) event detection, (2) event measurement, and (3) event analysis. For an event to be observed, its occurrence must first be detected. The complexity of event detection depends on the scope of the event. For instance, if the event is the entry to a routine, monitoring the routine for an entry event is sufficient. If, however, the event is defined to be a certain number of processors being active, the detection mechanism must continually be testing all processors simultaneously. The scope is broader and, thus, the detection is more difficult.

Event measurement records information about event occurrence and event analysis uses the data to derive various performance results. If event analysis is done at the same time as event detection, all measurement information that

describes an event must be known at a central point. Not only does the detection have to be centralized in this case, but so does the measurement. This requirements makes it difficult to define very complex events about program execution. Unfortunately, the scope of many parallel program events is broad. That is, interesting parallel program events tend to be defined with respect to some global parallel execution state, such as the level of parallelism.

The profiling instrumentation discussed in the previous section suffered from a requirement to perform event detection, measurement, and analysis at the same time during program execution. Each event had to be completely described in the profiling instrumentation, including the complex events regarding global parallel execution state. This severly limited the range of events that could be realistically profiled.

The functions of event detection, measurement, and analysis must be separated if practical performance tools are to be realized. Furthermore, complex events must be defined hierarchically as combinations of more primitive events which are monitored during program execution.

Event tracing is an instrumentation technique whereby data about an event are saved in a buffer whenever the event occurs during program execution. Part of the data is a timestamp indicating the actual time the event took place. The subtle power of timestamped event tracing is that all information required for analysis is saved. The analysis function, therefore, can take place independently of the event detection. Furthermore, detection and measurement of complex events can often be derived from the data saved for the low-level events. Thus, only primitive event measurement is necessary during execution.

The benefit of tracing as an instrumentation technique for parallel program performance evaluation is that very detailed information about a program's execution can be recorded in a trace from which complex queries about performance behavior are answered. The tracing operations are simple to implement and the instrumentation efficiency issues are localized to the management of the trace buffers. Indeed, the functional partitioning allowed by tracing can be seen in the highly modular and parallel approaches to its implementation.

1.6.2 CTRACE – A Tracing Facility for Cedar

CTRACE is a tracing utility developed for the performance evaluation of the parallel programs written for Cedar [19,20]. CTRACE has three components as shown in Figure 1.4. The event specification component defines the events that will be traced during program execution. The measurement component is responsible for enabling program and operating system instrumentation that will monitor the events of interest and generate the program trace. The CTRACE analysis component processes the trace data to produce various performance results. Together, the three components of the CTRACE utility form an integrated performance evaluation environment.

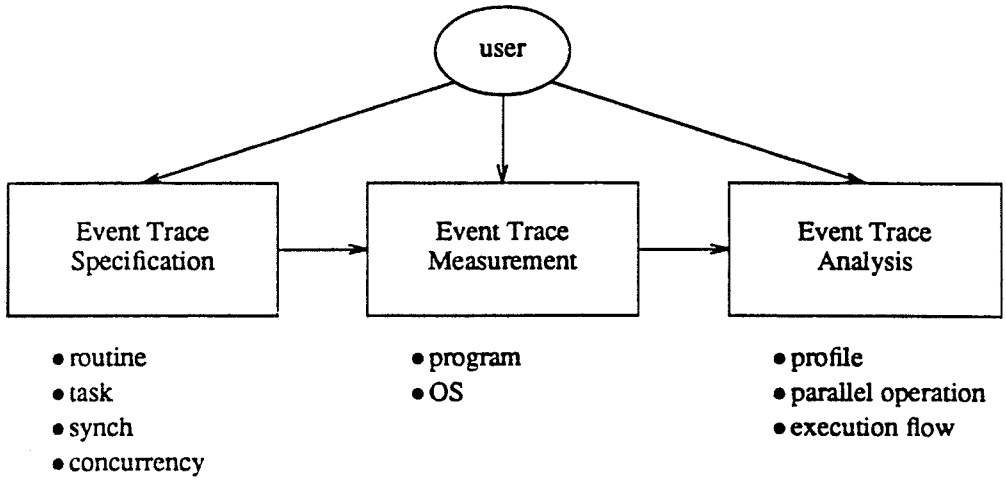


FIGURE 1.4
The CTRACE environment.

Event Specification. Software event specification in CTRACE serves two functions. First, it informs the measurement phase about the nature of the event: what the event is, when it occurs, and what data are needed to describe the event in the trace. Second, the specification serves as the basis for event trace interpretation in the analysis phase by providing information about how the various events are associated.

Two general types of events can be specified: standard and user-defined events. Standard events are defined with respect to common actions in the execution environment and are always available to the user. The set of standard events in the current version of CTRACE is shown below:

```
Program
     routine entry, exit
     basic block entry, exit

Cedar Fortran: Cluster Loop Parallelism
     CDOACROSS entry, exit
     CDOALL entry, exit

Cedar Fortran: Spread Loop Parallelism
     SDOACROSS entry, exit
     SDOALL entry, exit
```

```
Cedar Fortran: CDOACROSS Synchronization
     advance synchronization
     wait synchronization entry, exit

Cedar Fortran: Simple Synchronization
     fetch and op

Cedar Fortran: Cray-XMP Synchronization
     lock on, off
     event post, clear
     event wait entry, exit

Xylem: Multitasking
     create task, delete task
     queue task, resume task
     start task, stop task
     suspend task
     wait task entry, exit

Xylem: Synchronization
     set lock
     clear lock
     wait lock entry, exit
     dawdle entry, exit

Xylem: Task States
     running
     ready
     block
     idle
```

If the programmer desires an event not included in the standard set, the event must be described by the user. CTRACE currently provides for three general types of user-defined software events: MARK, ENTRY, and EXIT. MARK events simply indicate the occurrence of an event during execution. ENTRY and EXIT events indicate the entry into a block of computation and the associated exit, respectively. An ENTRY event must always be paired with an EXIT event.

In general, software event tracing allows any data that the user wants to associate with an event to be recorded in the trace. These data must be defined in the specification phase. The standard event data are predefined. A minimal amount of data will be automatically recorded by CTRACE. These data include an event identifier and a high-resolution global timestamp.

Event Trace Measurement. Currently, CTRACE performs all tracing in software. The approach taken is to provide a separate trace buffer per processor per task. This supports efficient concurrent tracing operations because there is no trace buffer contention.

The approach of separate trace buffers requires that event timestamps be generated from a global clock. The high-resolution timestamps for CTRACE are obtained from a 10-μsec real-time clock maintained by each Alliant FX/8 cluster and directly readable by each CE. The time value is stored as a 64-bit quantity. Because Cedar has one central hardware clock source, all individual FX/8 real-time clocks are synchronized. Thus, global time is distributed across the Cedar machine.

One artifact of using a real-time clock for timestamps is that periods when a task is not running must be identified in order to make execution time measurements. The beauty of tracing is that context switch events can be traced as well as other program events [21]. This makes it possible to have low-overhead timestamp generations as well as accurate execution time measurements. The context switch events are also useful for showing task scheduling behavior.

Support for event measurement takes several forms. A library of tracing routines is provided for initializing the tracing facility, recording events in trace buffers, and writing events to a trace output file. Trace-instrumented standard libraries are also available and can be compiled with a program. These include trace-instrumented run-time libraries for Cedar Fortran. Compiler preprocessor support is provided to insert instrumentation for events defined with respect to Cedar Fortran language statements. Event instrumentation can be enabled either through compile-line arguments or compiler directives. Finally, events external to the program, such as context switch events, require instrumentation in the operating system for their measurement.

Event Trace Analysis. The idea of event trace analysis is simple. Using the event specification and measurement information, the program traces are scanned to determine certain performance results. Two general event trace analysis tools are provided in CTRACE. The *execution profile analysis* tool produces statistical summary results of the program's execution. The statistics are similar to, but more extended than, the common sequential profiling results produced by *prof* and *gprof*. The *execution flow analysis* tool allows the programmer to observe the time-sequenced flow of events as they occurred during program execution. This tool is able to isolate certain periods of execution to identify particular characteristics of program behavior.

In general, the program traces should be viewed as a database of time-related information from which queries can be made regarding parallel program execution. A highly interactive analysis environment can be imagined that interfaces with the user through some query language and displays responses in various statistical and graphical representations.

Execution Profile Analysis. The execution profile analysis tools generate statistical information regarding program execution from the program traces. For

instance, given only standard routine and task event data, all of the following common profiling measurements can be produced:

- routine call counts
- descendant routine call counts
- direct execution time
- cumulative execution time
- average cumulative time per call
- descendant cumulative execution time

In addition to the above measurements, the following concurrency statistics can be generated without the need for defining additional events:

- sequential and concurrent routine call counts
- sequential and concurrent routine execution time
- number of tasks created
- average task execution time
- execution time histogram of task concurrency
- average task concurrency

Data from the other events only increase the database from which execution profile statistics can be drawn. Of particular interest is the execution time of parallel loops and synchronization operations. The following represents the types of statistics provided:

- CDOACROSS and CDOALL execution time
- SDOACROSS and SDOALL execution time
- task wait synchronization counts and times
- event wait synchronization counts and times
- lock wait synchronization counts and times

1.7 Execution Flow Analysis

The ability to observe the program events in a time-ordered sequence of occurrence differentiates tracing from profiling tools. Statistical summaries give a global picture of program execution but lack historical perspective. Execution flow analysis provides the programmer with a window into the program traces at various levels of detail. The concept of *replaying* the program's execution with respect to the traced events forms the basis of execution flow analysis tools.

In general, execution flow analysis is used as a means to explore the program's execution for evidence of good, bad, or strange behavior. Sometimes the programmer just wants to see general characteristics, such as the sequence of routine execution. At other times, the programmer will use execution flow analysis in combination with highly specialized event traces for "search and destroy" missions to pinpoint some anomalous behavior or dissect a poorly performing section of code. The execution flow analysis toolset provides an environment for the programmer to intelligently search and analyze the program trace database.

A basic set of execution flow analysis functions is currently provided in CTRACE. The function of moving around in the program trace and displaying events is called *event trace browsing*. In addition to enabling forward and backward movement in program event history, event trace browsing provides different ways of searching through the event trace. Textual and graphical presentation capabilities also exist for showing the events that occur in certain regions of the trace.

One such graphical representation is the *dynamic call graph* display. The display shows the active calling arcs of the static *subroutine interconnection graph* with the nodes being drawn dynamically as the routines are encountered in the program trace. Figure 1.5 gives an example of the state of a task's execution on an FX/8 Cedar cluster in the form of a dynamic call graph. The path through the static call graph is shown for each execution thread with the leaf node representing the currently executing routine. The global dynamic call graph of the task shows a merge of the individual calling branches with all currently active routines drawn as square nodes.

The key feature of the event trace browser is that it is interactive. It takes the event specification and the program trace and provides a front-end for general inquiries about program execution. Basic searching and event presentation are be handled by the browser. More sophisticated analysis is the responsibility of execution flow generalization.

The basic idea behind execution flow generalization is to provide the programmer with a way of observing higher-level execution behavior not represented directly by some traced event. Execution flow generalization builds high-level events from combinations of traced events. As an example, task concurrency events reflect the number of active tasks during a program's execution. Each level of task concurrency represents a separate event. Although the occurrence of events of this type is difficult to detect at run time, it is easy to derive from analysis of the individual task traces. From the task state event data, the beginning and ending times for the high-level task concurrency events can be determined. A task concurrency event "trace" can be generated from this analysis and a graph of task concurrency produced. An example task activity graph and the accompanying task concurrency graph are shown in Figure 1.6.

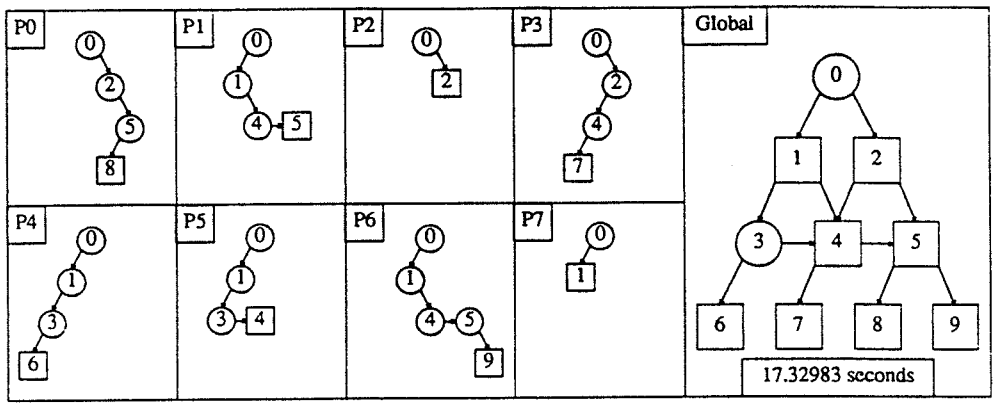


FIGURE 1.5
Dynamic call graph.

Notice the period of time during which none of the program tasks are active. In multiprogrammed multiprocessing systems such as Cedar, this can occur because all tasks within the system are sharing processing resources. Although this is a simple example of execution flow generalization, it illustrates the basic idea upon which more complicated generalizers can be designed.

As the range and details of events increase, so does the complexity of trace analysis. Additional tools will be developed that allow the user to more easily browse through the trace at various levels of detail and query the analysis system about program behavior. These tools will require new techniques for interpreting the trace data as well as reducing the data into meaningful representations for presentation to the user.

1.8 Hardware Instrumentation

In addition to software instrumentation, several hardware instrumentation approaches are being pursued for Cedar. Hardware measurements focus on the physical events taking place within various Cedar machine components. These measurements include Alliant FX/8 cluster, global interface, global network, and global memory measurements (see Table 1.4) [22]. A flexible hardware perfor-

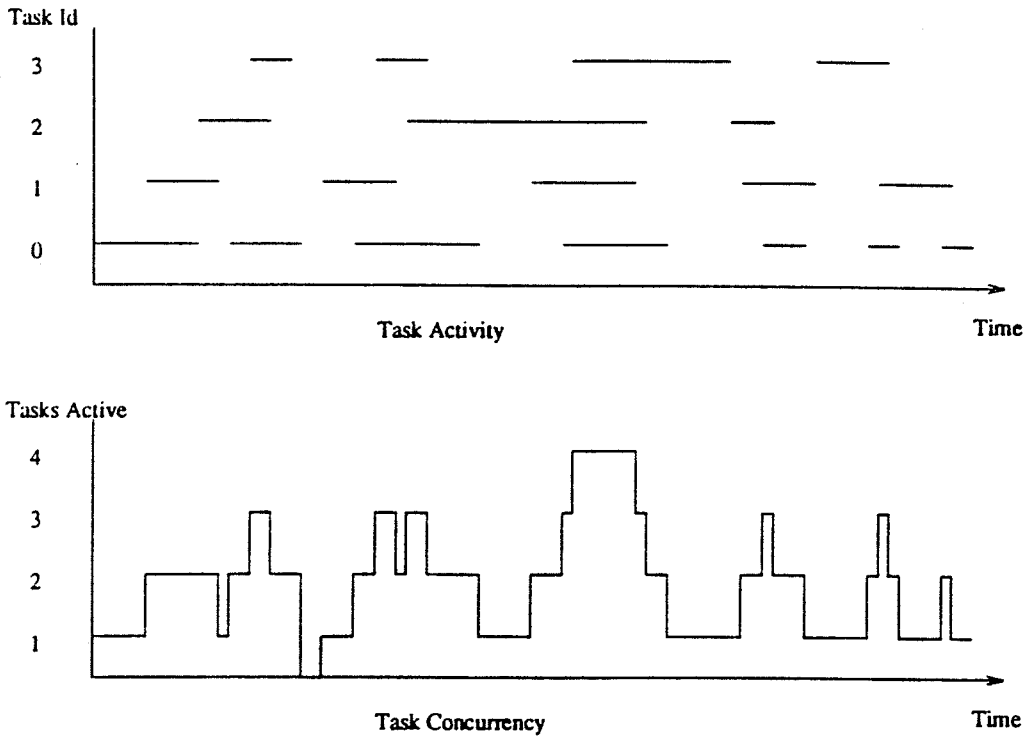


FIGURE 1.6
Task concurrency graphs.

TABLE 1.4
Hardware Measurements

Alliant FX/8	*Global*	*Global*	*Global*
Cluster	*Interface*	*Network*	*Memory*
instructions	reference type	input port utilization	reference type
vector operation	prefetch operation	network contention	reference distribution
concurrent operation	data buffer usage	bandwidth utilization	utilization
cache operations	interface delay	network delay	reference delay
memory references	data transfer rate	output contention	memory contention

mance monitoring system is being developed that will make these measurements for the Cedar machine [9].

Hardware instrumentation can also be used to make certain software instrumentation more efficient. A hardware trace buffering facility is being developed that will provide support for CTRACE in the form of automatic event time-stamping, trace buffer management, and a fast path for generating events.

1.8.1 Hardware Performance Monitor

The integral hardware support for performance monitoring in the Cedar system is represented in Figure 1.7. The key elements are buffered on-board test points, a data acquisition system, and multibus-based control.

The Cedar system is observable by way of a series of performance-measurement test points provided on each circuit board. These test points provide information such as microcode instruction and address on the CE boards, control states and important counters on the network interface boards, and other signals specific to the global network and global memory boards. The signals provide important information on the activities of each board.

The signals are monitored by a general purpose high-resolution data acquisition system, called the black box. A black box card cage can hold up to 32 modules. Several different types of modules have been developed including signal conditioning, counting, timing, and data-logging. The hardware monitoring system is generally configurable to the type of measurement experiment desired. For example, one module can be configured as an interval timer and another as a counter to count the number of floating-point operations. Using this configuration, MFLOPS (million floating-point operations per second) can be easily obtained.

The black boxes are connected to a controller card resident in one of the multibus backplanes used by the IPs in a Cedar cluster. Each controller can

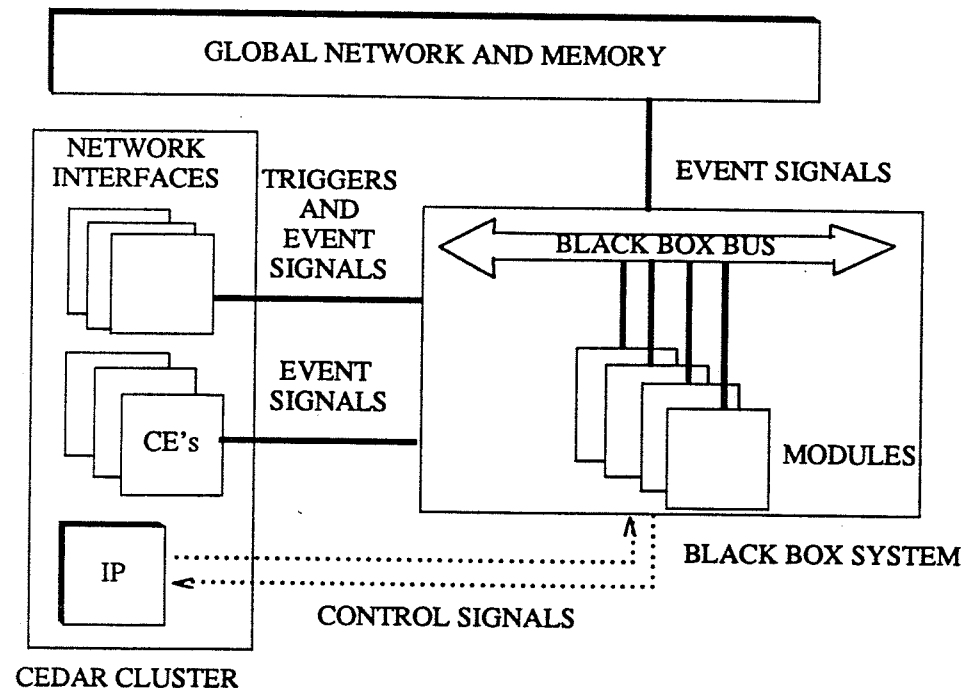


FIGURE 1.7
Cedar hardware monitoring system.

handle as many as 16 black box systems. Multiple controllers can be installed in one multibus, for centralized control by one IP, or in multiple multibuses, for distributed control by several IPs within one cluster or across multiple clusters.

The IP directly controls each black box and can read the contents of the black box registers and memory to obtain the collected data. Since an IP is also responsible for providing disk I/O to the cluster, it is possible for run-time performance data to be logged to disk as they are collected. Real-time analysis by the IP critically depends on the volume of data received and the amount of processing to be performed.

One advantage of using such a hardware monitoring system is that it is highly modular. Each module in a black box system has an identical standard interface that makes the system easily expandable. Special types of performance modules can be designed as needed, and as long as their interfaces stay the same, all of these modules can be controlled by the same mechanism. This allows different types of modules to work together to collect different kinds of data simultaneously. It is often necessary to correlate data collected in different parts of a system within the same time frame. The ability to mix different types of modules together to collect different types of data simultaneously makes the post collection analysis much easier and more accurate.

Within a cluster, under the proper conditions, the performance monitoring can be tightly correlated with events produced by a given task (software), or by a given computational element (hardware). This software and hardware resolution degrades as measurements are made farther away from the cluster, e.g., in the global memory, where it is very difficult to know which CE instruction required the memory access.

The software resolution (i.e., the extent to which performance parameters measured throughout the system can be correlated with specific tasks) is highly dependent on the number and type of explicit and implicit triggers produced by the task that can be detected by the black boxes. Explicit triggers can be generated by special instructions in a task specifically for the purpose of starting, stopping, or signaling the black boxes. The global interface for each CE was designed such that certain instructions issued by the CE would generate special external trigger signals. The trigger instructions can be embedded in a user's program or in the operating system to precisely start and stop hardware measurements in the software. Implicit triggers are sets of circumstances that can be identified as characteristic of the task under analysis (such as certain "unique" sequences of operations that can be correlated with the execution of the task). Recognition of implicit triggers depends either on an accurate program flow model or on a detailed understanding of the task under investigation.

1.8.2 Hardware Trace Buffering System

A hardware trace buffering utility is being developed for Cedar that will support CTRACE program event tracing by providing the means for low-overhead event generation and trace buffer storage. The hardware tracer consists of an interface to the hardware performance monitor, a trace buffer memory, a timestamp clock, and an interface to disk. Each CE in the Cedar system will have its own tracer module.

The hardware performance monitor will provide access to the software trigger signals that will indicate the occurrence of an event. Event data indicated by the triggers will be captured by the monitor and passed to the tracer. The tracer will timestamp the event and store it in a trace buffer specific to the CE producing the event. The tracer is designed to accept events as fast as the CE can produce them.

In addition to providing fast paths for writing event data, the tracer relieves CTRACE of the need to manage trace buffers in software. The automatic time-stamping of events also reduces the cost of generating software events. Hopefully, the tracer will help reduce the impact that trace instrumentation has on parallel program execution. It should also improve the resolution at which events can be observed.

1.8.3 Real-Time Performance Analysis

The hardware trace buffering utility together with the hardware performance monitor will provide the basis in the future for a real-time performance analysis system. The researcher's intent is to design a system that processes in real time the performance data produced by Xylem and parallel programs running on Cedar and shows system and program performance through various forms of graphical performance displays. It is hoped that the system will provide immediate feedback of current performance as well as summary information of past performance that will be useful for tuning the overall performance of Cedar.

1.9 Conclusion

Instrumentation for parallel systems must offer the user ways of observing parallel operation at various levels of detail. However, the hardware and software constraints imposed by real systems make it difficult to implement instrumentation mechanisms that do not somehow perturb the parallel behavior. The challenge is to design instrumentation techniques that integrate well with parallel architectures and parallel execution environments. In the future, these techniques

should be included as part of the overall parallel system design and be provided as standard components on all parallel machines.

The instrumentation tools developed for the Cedar multiprocessor are prototypes used to explore various tradeoffs in design and implementation. The set of tools is continually being improved as more instrumentation is being placed in the hardware and software to gather additional data about the system. Many of the basic instrumentation approaches, however, are considered general enough in scope to serve as a performance instrumentation framework for other machines.

References

1. D. J. Kuck, A. H. Sameh, A Supercomputing Performance Evaluation Plan, *Proceedings 1987 Supercomputing Conference*, Greece, June, 1987.
2. D. J. Kuck, D. H. Lawrie, W. Jalby, P. Yew, A. Malony, and A. Sameh, Methodology for Performance Evaluation for High Performance Computer Systems, CSRD Report No. 725, Center for Supercomputing Res. & Dev., Univ. of Illinois at Urbana-Champaign, Dec. 1987.
3. D. Gajski, D. L. Kuck, D. Lawrie, and A. Sameh, Cedar – A Large Scale Multiprocessor, *Proceedings 1983 International Conference on Parallel Processing*, Belaire, MI, 1983.
4. D. J. Kuck, E. S. Davidson, D. H. Lawrie, and A. H. Sameh, Parallel Supercomputing Today and the Cedar Approach, *Science*, Vol. 231, Feb. 28, 1986, pp. 967–974.
5. P. Emrath, An Operating System for the Cedar Multiprocessor, *IEEE Software*, Vol. 2, No. 4, 1985, pp. 30–37.
6. R.E. McGrath and P. Emrath, Using Memory in the Cedar System, *1987 International Conference on Supercomputing*, 1987.
7. M. Guzzi, Cedar Fortran Reference Manual, CSRD Report No. 601, Center for Supercomputing Res. and Dev., Univ. of Illinois at Urbana-Champaign, 1987.
8. M. Guzzi, Multitasking Runtime Systems for the Cedar Multiprocessor, CSRD Report No. 604, Center for Supercomputing Res. and Dev., Univ. of Illinois at Urbana-Champaign, 1986.
9. K. Gallivan, W. Jalby, A. Malony, and P. Yew, Performance Analysis on the Cedar System, to appear as a chapter in *Performance Evaluation of Supercomputers*, J.L. Martin, Ed., North-Holland, 1988.
10. A. D. Malony, Cedar Performance Evaluation Tools: A Status Report, CSRD Report No. 582, Center for Supercomputing Res. and Dev., Univ. of Illinois at Urbana-Champaign, July 1986.

11. A. D. Malony, Proposal for Concurrency Efficiency Measurements, Internal memo, Center for Supercomputing Res. and Dev., Univ. of Illinois at Urbana-Champaign, August 1986.

12. A. D. Malony, Concurrency Efficiency User's Manual, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. and Dev., CSRD Report No. 675, June 1987.

13. A. D. Malony, Virtual High-Resolution Process Timing, CSRD Report No. 616, Center for Supercomputing Res. and Dev., Univ. of Illinois at Urbana-Champaign, Oct. 1986.

14. R. Barton, P. Emrath, D. Lawrie, A. Malony, and R. McGrath, New Approaches to Measuring Process Execution Time in the Cedar Multiprocessor System, CSRD Report No. 744, Center for Supercomputing Res. and Dev., Univ. of Illinois at Urbana-Champaign, Jan. 1987.

15. A. D. Malony, High Resolution Process Timing User's Manual, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. and Dev., CSRD Report No. 676, June 1987.

16. A. D. Malony, Ideas on Profiling Parallel Programs, Internal memo, Center for Supercomputing Res. and Dev., Univ. of Illinois at Urbana-Champaign, August 1986.

17. A. D. Malony, Program Profiling in Cedar, CSRD Report No. 654, Center for Supercomputing Res. and Dev., Univ. of Illinois at Urbana-Champaign, March 1987.

18. S. L. Graham, P. B. Kessler, and M. K. McKusik, An Execution Profiler for Modular Programs, *Software – Practice and Experience*, Vol. 13. pp. 671–85, 1983.

19. A. D. Malony, Program Tracing in Cedar, CSRD Report No. 660, Center for Supercomputing Res. and Dev., Univ. of Illinois at Urbana-Champaign, April 1987.

20. A. D. Malony, CTRACE User's Manual, CSRD Report No. 710, Center for Supercomputing Res. and Dev., Univ. of Illinois at Urbana-Champaign, Nov. 1987.

21. A. D. Malony, Process Context Switch Tracing, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., CSRD Report No. 688, Oct. 1987.

22. A. D. Malony, Cedar Performance Measurements, CSRD Report No. 579, Center for Supercomputing Res. and Dev., Univ. of Illinois at Urbana-Champaign, June 1986.