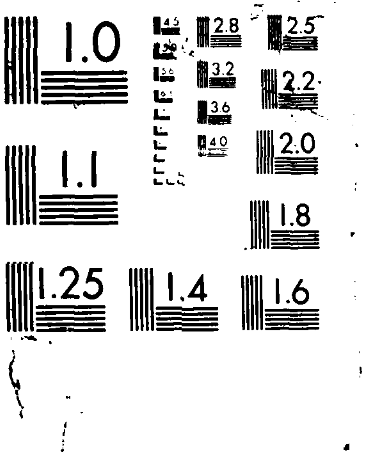
AD-A190 003   PROGRAM PROFILING IN CEDAR(U) ILLINOIS UNIV AT URBANA   1/1
CENTER FOR SUPERCOMPUTING RESEARCH AND DEVELOPMENT
A MALONY 17 MAR 87 CSRD-654 AFOSR-TR-87-1986
UNCLASSIFIED   F49620-86-C-0136   F/G 12/5   NL


1.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

unclassified

DTIC FILE COPY 2

AD-A190 883

REPORT DOCUMENTATION PAGE

1b. RESTRICTIVE MARKINGS

2b. DECLASSIFICATION / DOWNGRADING SCHEDULE

3. DISTRIBUTION / AVAILABILITY OF REPORT
Approved for public release;
distribution unlimited.

4. PERFORMING ORGANIZATION REPORT NUMBER(S)
CSRD Report No. 654

5. MONITORING ORGANIZATION REPORT NUMBER(S)
AFOSR-TR- 87-1986

6a. NAME OF PERFORMING ORGANIZATION
The Board of Trustees of the
University of Illinois

6b. OFFICE SYMBOL (If applicable)

7a. NAME OF MONITORING ORGANIZATION
AFOSR/NM

6c. ADDRESS (City, State, and ZIP Code)
506 S. Wright St.
Urbana, IL 61801

7b. ADDRESS (City, State, and ZIP Code)
AFOSR/NM
Bldg 410
Bolling AFB DC 20332-6448

8a. NAME OF FUNDING/SPONSORING ORGANIZATION
AFOSR

8b. OFFICE SYMBOL (If applicable)
NM

9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
~~AFOSR~~ F49620-86-C-0136

8c. ADDRESS (City, State, and ZIP Code)
AFOSR/NM
Bldg 410
Bolling AFB DC 20332-6448

10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.
61102F	2304		

11. TITLE (Include Security Classification)
Program Profiling in Cedar

12. PERSONAL AUTHOR(S)
Allen D. Malony

13a. TYPE OF REPORT
Internal Report

13b. TIME COVERED
FROM ________ TO ________

14. DATE OF REPORT (Year, Month, Day)

15. PAGE COUNT

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This paper presents an analysis of parallel program profiling for the Cedar multiprocessor and a preliminary functional specification of a parallel program profiling tool, cprof. The standard UNIX profiling tools serve as a beginning design basis for cprof and their basic functionality is discussed in Section 2. Section 3 describes the problems with parallel program profiling. In particular, profiling in the Cedar program execution environment is analyzed. The initial version of cprof supports routine counting and timing functionality. The proposed cprof implementation is presented in detail in Section 4. Profiling operations other than routine counting and timing are interesting for parallel programs. Section 5 briefly describes possible extensions to cprof.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT
□ UNCLASSIFIED/UNLIMITED □ SAME AS RPT. □ DTIC USERS

21. ABSTRACT SECURITY CLASSIFICATION

22a. NAME OF RESPONSIBLE INDIVIDUAL
Maj. John P. Thomas

22b. TELEPHONE (Include Area Code)

22c. OFFICE SYMBOL
NM

83 APR edition may be used until exhausted.
All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

CSRD 654

AFOSR-TR- 87-1986

Program Profiling in Cedar†

Allen Malony
Center for Supercomputing
Research and Development

1. Introduction

The goal of program profiling is to provide an accurate characterization of a program's behavior and performance. Program profiling can have several different meanings depending on the measurements of interest and the programmi^g environment. Generally, profiling measurements are concerned with collecting information regarding the dynamic execution behavior of the program. The purpose is to help the user evaluate alternative implementations and to guide program optimizations. Two categories of profiling are commonly defined: counting the number of times a statement or routine is executed, and timing the execution of statements or routines. For sequentially executing programs, profiling defined in this manner is adequate and implementations are not particularly complex. However, a parallel programming environment calls for an extension to the common profiling approaches to include measurements of dynamic "interaction" between the concurrent execution threads. Consequently, parallel program profiling is considerably more complex to implement.

This paper presents an analysis of parallel program profiling for the Cedar multiprocessor and a preliminary functional specification of a parallel program profiling tool, *cprof*. The standard UNIX® profiling tools serve as a beginning design basis for *cprof* and their basic functionality is discussed in Section 2. Section 3 describes the problems with parallel program profiling. In particular, profiling in the Cedar program execution environment is analysed. The initial version of *cprof* supports routine counting and timing functionality. The proposed *cprof* implementation is presented in detail in Section 4. Profiling operations other than routine counting and timing are interesting for parallel programs. Section 5 briefly describes

† This work was supported in part by the National Science Foundation unser Grant Nos. US NSF DCR84-10110 and US NSF DCR84-06916, the U.S. Department of Energy under Grant No. US DOE DE-FG02-85ER250001, the IBM Donation, and the Center for Supercomputer Research and Development.

possible extensions to *cprof*.

2. UNIX Profiling

Profiling in the UNIX operating system performs two basic functions: counting the number of times a routine is called, and determining routine execution time. The standard profiling tools, *prof* and *gprof*, are based on sampled execution time measurements. Profiling tools running under ELXSI UNIX use time measurement instead of sampling. The following reviews standard UNIX profiling functionality in the context of these two implementations. It is important to realize that these profiling tools are intended for use with sequentially executing programs.

2.1. Standard Profiling Tools

prof and *gprof* are the standard profiling tools of the UNIX operating system [GKMc82,83] [UNIX84]. Two types of presentation output are produced by these tools from the profiling measurements. The *flat* profile shows all routines called during program execution, with the count of the number of times they were called and the *direct* execution time[1]. 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 the descendants' times represent[2]. Similar results are shown for the parents of the routine.

The profiling measurements needed for routine call counting are easily implemented in standard profiling; a call to a monitoring routine exists in the prologue of each profiled routine. The monitoring routine determines the called routine by the return address and increments the call counter for that routine. In addition to counting the number of times a routine is called, the number of times each arc in the dynamic call graph is traversed is also recorded. Knowing the identity of the called routine, the monitor-

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

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

ing finds the return address of the routine, identifying the calling routine. A call count associated with the calling arc is incremented.

Timing in *prof* and *gprof* is based on sampled execution time. When profiling is enabled, UNIX updates a histogram of the location of the profiled program's program counter at the end of each interval timer interrupt (usually 1/60th of a second). Routine execution times are determined from a distribution of these program counter samples within the program. To determine the amount of time spent in a particular routine at the end of program execution, i.e. direct execution time, the PC histogram counts for that routine are summed and multiplied by the interval timer period. Obviously, direct execution times are subject to statistical sampling errors and can give misleading results.

To determine "cumulative" execution times (direct execution time plus time spent in descendent routines), 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 that same amount of time. This assumption, coupled with the statistical approximation of direct execution time, can produce invalid timing measurements.

2.2. Profiling under ELXSI UNIX

Clearly, the statistical sampling method of measuring execution times systems limits the accuracy of the standard UNIX profiling tools. This is especially true for high-performance systems whose cycle time is several orders of magnitude smaller than the sampling period. A more reliable method, such as the one employed by a profiling tool under ELXSI UNIX [Carr84], measures the execution time of a routine by measuring elapsed time from routine entry to routine exit. Normally, such a time measurement would be complicated by the time-slicing of a program, but the ELXSI system maintains a 64-bit per-process timer through hardware and microcode, that is initialised at process start-up and is incremented every 25 nanoseconds when the process is executing. The ELXSI profiling provides functionality equivalent to *prof* and *gprof* but with measured execution times.

The important implementation point to notice is that execution times can be measured exactly and assumptions about the time spent in a routine call are unnecessary. There is no need to keep a PC histogram and cumulative execution time can be measured without counting arc traversals or generating dynamic call graphs. The ELXSI profiling output includes the standard flat profile results plus the cumulative execution times, the percentage cumulative time, and the cumulative time per call. However, ancestor and descendant information is not presented, although it seems trivial to add this to the ELXSI profiling tool.

2.3. Functionality Summary

A summary of standard UNIX profiling functionality, as represented by *prof*, *gprof* and the ELXSI profiler, is given below.

Measurements

- routine call counts
  normally implemented by a prologue to each profiled routine that increments the call count for that routine, plus records a traversal of the calling arc in the dynamic call graph

- sampled execution time
  statistical approximation of routine execution time based on program counter sampling during program execution – not recommended because of potential for statistical timing inaccuracies

- measured execution time
  direct measurement of routine execution time by recording elapsed time from routine entry to exit – desired approach for reliable execution time measurements

Profiling Results

- routine call count
  the number of times a routine was called during program execution

- descendant call count
  the number of times a routine call a descendant routine

- direct execution time
  the time for which a routine is directly accountable – the time spent executing only the statements in the routine

- percentage direct execution time
  the direct execution time expressed as a percentage of the total program execution time

- cumulative execution time
  the direct execution time plus the time spent in each descendant routine call

- percentage cumulative execution time
  the cumulative execution time expressed as a percentage of the total program execution time

- average cumulative time per call
  the cumulative execution time divided by the routine call count

- descendant cumulative execution time
  portion of cumulative time spent in each descendant routine

- percentage descendant cumulative execution time
  descendant cumulative execution time expressed as a percentage of the total routine cumulative execution time

3. Parallel Profiling in Cedar

The traditional profiling techniques , represented by the UNIX profiling mechanisms presented earlier, are clearly defined and are sufficient for sequentially executing programs. However, profiling of parallel programs is not as well understood, and standard methods are inadequate for the goals of program characterization and optimization. This section analyses the parallel profiling problem in the Cedar system. The analysis focuses on how the standard profiling functionality conforms to the Cedar parallel execution environment. Although parallel profiling undoubtedly has a broader definition, several important problems appear in this restricted examination. After understanding the sources of these problems, an initial strategy for profiling in Cedar is proposed. The implementation details of this approach are presented in Section 4. The discussion of extensions to the initial *cprof* definition and implementation is given in Section 5.

3.1. The Basic Parallel Profiling Problem

The goal of traditional profiling tools is to optimize the performance of a program by improving routines that implement an *abstraction* [GKMc82]. Using the routine characterization of call counts and execution times, iterative techniques can be applied to integrate excessively called routines or to improve routines that are execution time bottlenecks. Certainly, parallel program profiling has this same optimization goal, but the techniques for improving performance are not as simple and require a more complete characterization of dynamic program behavior. Intuitively, one would expect this to be the case as the number of possible states of a parallel program are significantly more than those for a sequential program.

Assuming the *object* of profiling is the routine, *how many?* is answered in sequential profiling by routine call counts, and *how long?* by routine execution time (direct and cumulative). Little additional information is needed to provide an effective characterization for optimization analysis, save a complete routine execution trace. However, parallel program execution raises the question of *with whom?* that cannot be answered by the standard profiling measurements. Parallel program optimization depends on a useful characterization of this issue.

3.2. The Cedar Program Execution Environment

Before analyzing the problems of realizing standard profiling functionality in Cedar, we first review the Cedar parallel execution environment. Two main types of parallelism are available in Cedar: multitasking and loop-level parallelism. A program can be decomposed into several cluster tasks which communicate through shared memory. Each cluster task is a separately scheduled entity that executes on a Cedar cluster. Each cluster contains multiple CEs providing an opportunity for parallelism within each cluster task. The cluster CEs operate simultaneously during the execution of parallel loops within a task; iterations of the parallel loop are concurrently assigned to the CEs. Multitasking is provided by the Xylem operating system at the request of the user, whereas loop parallelism occurs automatically at the instruction-level.

When a task executes on a cluster, a fixed number of cluster CEs will always be available to the task. Thus, parallel execution within a task is synchronous, in the sense that the computing resources are fixed[3]. Conversely, parallel execution between tasks is asynchronous and dependent on the load in the system. In

[3] In this case, concurrent execution is synonymous with parallel execution. The terms "concurrent" and "parallel" are used interchangeably in the paper in this context.

the discussion that follows, only the profiling of individual tasks is considered. That is, each task is treated as a separate entity to be profiled. This will help simplify the profiling analysis. We will return to the question of profiling between tasks in Section 5.

3.3. Standard Profiling in Cedar

Several important parallel profiling problems can be observed by examining standard profiling functionality in the Cedar parallel execution context. First, the object of standard profiling is discussed. An analysis of call counting and execution timing is then presented.

3.3.1. Routine Profiling

As mentioned before, the object of standard profiling is the routine. For sequentially executing programs, this is a reasonable focus; only one routine can be executing at any time and the routine is a logical division of computation. Moreover, the characterization of a routine's execution in terms of call counts and execution times is a direct measure of its performance and its relative importance to the overall computation.

However, parallel program execution implies the potential for more than one routine to be executing concurrently. Although routine call counts and time measurements are still important, especially when there is a sizable sequential component, the dynamic interactions that affect parallelism are of equal interest. The problem is that the standard profiling measurements do not include any information about the parallelism present when a routine is executing. The question here is whether new parallel profiling measurements are closely tied with the "routine". Clearly, to identify where in the program an optimization should be applied, there must be some reference point for the profiling measurements. Consequently, considering the routine as providing a reference framework for profiling measurements might make more sense for parallel execution environments.

This issue of the "object" of parallel profiling should be given additional thought. For our purposes in this document, the routine will still be considered as the object of profiling. However, we should be mindful of this viewpoint when interpreting the profiling results for optimization purposes.

3.3.2. Routine Call Counting

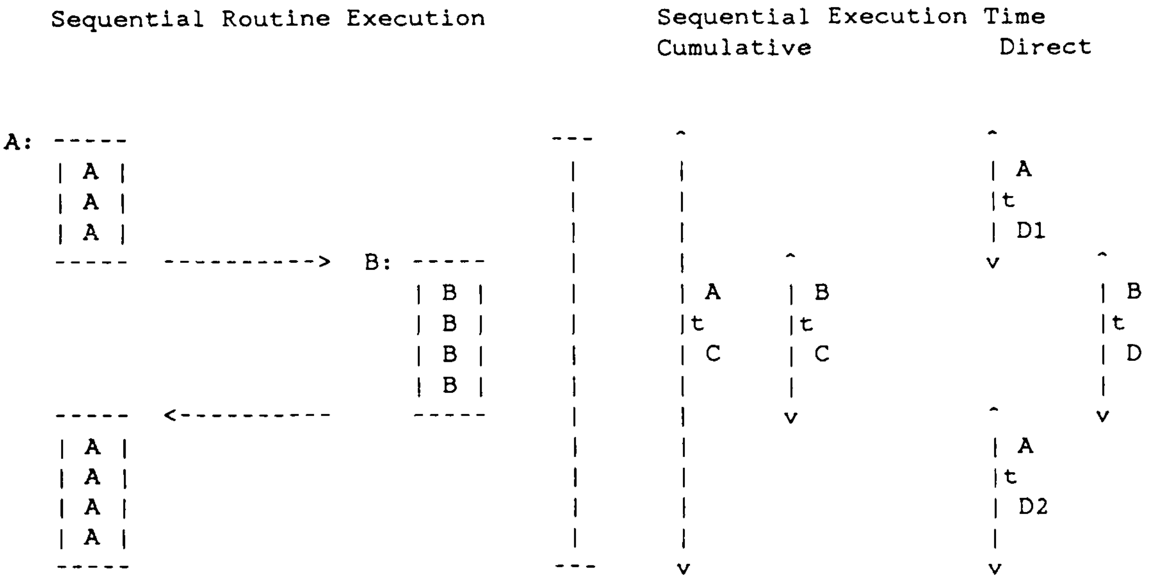
Routine call counting is a simple measurement that is virtually unaffected by parallel execution of the routine. It can be easily implemented by a call to a counting routine at routine entry. Because a routine can be called by two or more CEs simultaneously, the routine call count must be locked when it is incremented. Descendant routine call counts are also simple to obtain.

Although routine call counting poses no real profiling problems in a parallel execution environment, there is the question of its incompleteness as a single counting measure. Clearly, there are other interesting things to count during parallel execution. For example, routine call counting can be extended to the number of sequential calls to the routine and the number of concurrent calls. The number of concurrent calls could be further divided between calls to the routine when other CEs are also executing and concurrent calls to the same routine.

3.3.3. Routing Execution Times

Routine timing is complicated by parallel execution. In sequential execution timing, there are clearly differentiated time boundaries between direct time for a routine and the time for its descendants. This is shown graphically in Figure 1. At any time, direct time (indicated by a D subscript in the figure) is being measured for only one routine. This makes it possible to compute the descendant time and the cumulative time (indicated by a C subscript) directly from the direct time measurements. In fact, this is exactly the basis for the approach taken by the ELXSI profiling tool. One can also derive direct execution times from cumulative time measurements. The different timing relationships are shown in the figure.

The problem that parallel execution poses for profiling is the potential for several routines to be active simultaneously. If there is a call to various routines within a parallel loop, any combination of these routines could be active on the multiple execution streams. This is depicted in Figure 2 for two processors. The different situations that can occur are labeled by numbers in the figure. Each case is described



$$t_D^A = t_{D1}^A + t_{D2}^A$$

$$t_C^A = t_D^A + t_C^B$$

$$t_D^A = t_C^A - t_C^B$$

Figure 1. Sequential Routine Execution and Timing

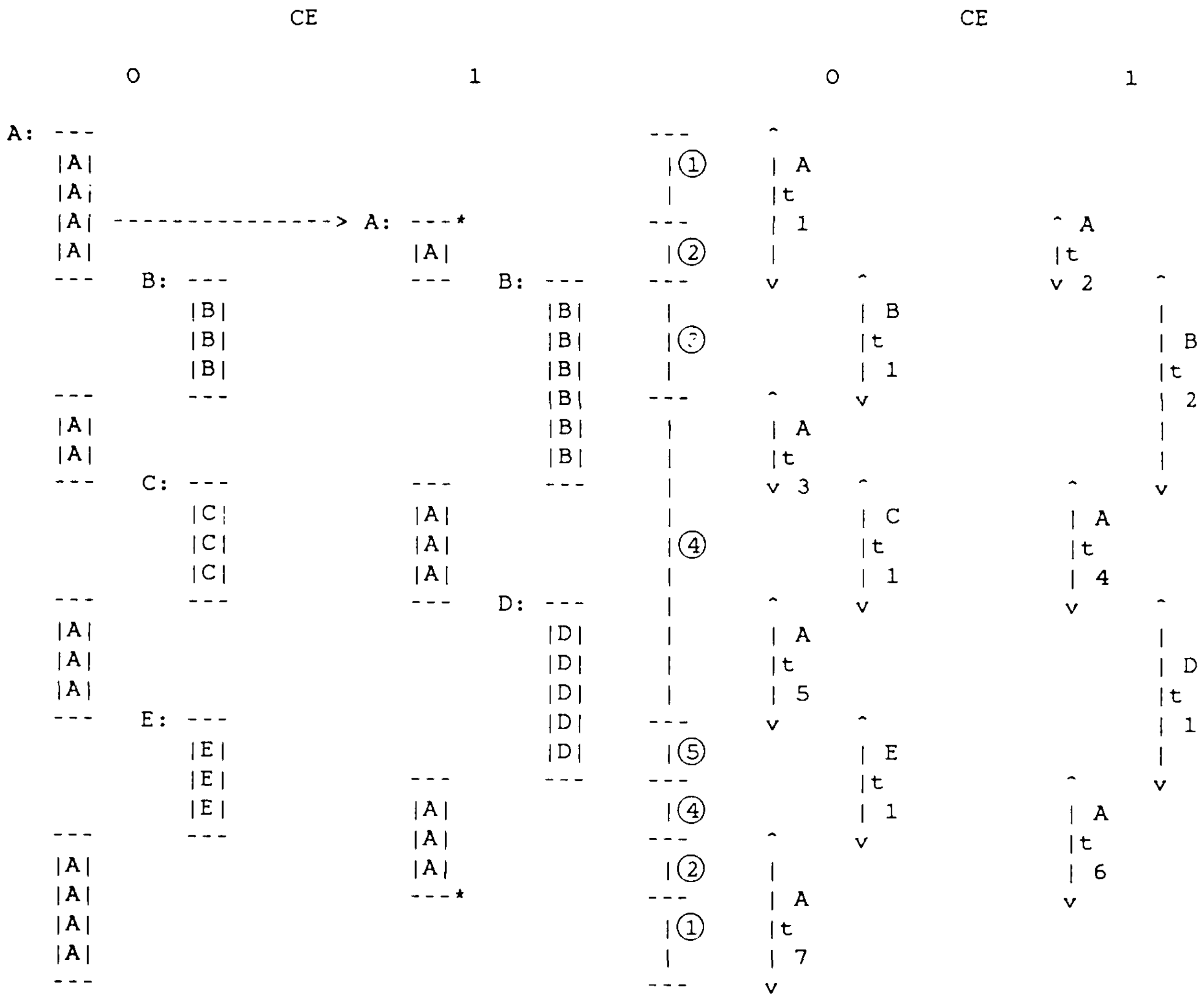
separately below.

There is no concurrency present in region ①. Since routine A is the only routine executing in this region, the time should be accounted to its direct and cumulative execution time.

However, at the beginning of region ②, routine A has just entered a parallel loop and processor 1 begins executing iterations of the parallel loop. During region ②, A is executing concurrently with itself. Herein lies the first timing problem:

> **Timing Problem 1:** When a routine is executing concurrently with itself, how are direct time and cumulative time for that routine measured?

The root of this problem lies in the definition of execution time. If execution time is to mean *elapsed* time,



$$t_D^A = t_3^A + t_4^A + t_5^A + ?$$

$$t_C^A = t_D^A + ?$$

$$t_D^B = ?$$

$$t_D^C = t_1^C \quad t_D^D = t_1^D \quad t_B^D = t_1^B$$

Figure 2. Parallel Routine Execution and Timing

execution time for 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. Under the elapsed execution time definition, the time associated with region ② would be added to A's direct and cumulative times. Under the cpu execution time definition, the time for region ② would be doubled because there are two processors operating.

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. However, depending on the situation, one or the other time measurements may be difficult to obtain. The point marked by * at the start of region ② is a case where parallel operation occurs at the instruction–level in a parallel loop and is unobserved by the program or the operating system. This makes measurement of cpu time in region ② virtually impossible.

It is not until the start of region ③ that profiling can observe the program executing concurrently. Region ③ also has a routine executing concurrently with itself. In this case, each execution thread represents a separate invocation of routine B. This is different from region ②. Elapsed time and cpu time measurements are easier to obtain in this case, but there is still the problem of determining direct and cumulative time. Also, it is unclear what time value should be propagated back to B's calling routine A to add into A's cumulative time. This represents the second timing problem:

Timing Problem 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?

Throughout region ④, A is executing concurrently with one of its called routines. For instance, A completely overlaps its execution with the called routine C. The problem here has to do with how time is recorded – as direct time for A, direct time for C, or both. If the time is direct time for both caller and callee, adding C's cumulative time to A's cumulative will over–estimate A's cumulative time since A's direct time is also added. On the other hand, the callee's cumulative time should be accounted for in the caller's time values in some way.

Timing Problem 3: When a caller and callee routine are concurrently executing, how are direct and cumulative times being accumulated for the caller and the callee?

Region ⑤ shows the case of two different callee routines executing simultaneously. It is reasonable that each routine should be accumulating direct and cumulative time. The problem occurs when these times are reported back to the caller. Because there is no timing information regarding the overlap of parallel execution, it is unclear how to incorporate the callee's times into the caller's time values. This represents the fourth timing problem.

> **Timing Problem 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?

Other problems potentially exist having to do with routines executing on different execution threads at different levels within a calling tree. The example above only shows a two-deep call nesting from a single caller. It is easy to think of a case where a routine is executing concurrently with itself, but where each instantiation was called by a different routine. Again there are the problems of direct time vs. cumulative time, and elapsed time vs. cpu time.

3.4. A Profiling Strategy for Cedar

Although the simplicity of standard profiling functionality is desired, clearly there are problems with applying the approaches to a parallel execution environment. However, counting and timing are fundamental measurements that will certainly be a part of any parallel profiling tool. The proposed profiling strategy for Cedar is to define mechanisms for gathering as much potentially useful information as possible from the execution of the program. In this way, some of the problems encountered earlier can be alleviated by saving profile data for post-analysis and filtering. Although the initial profiling tool, *cprof*, will still be routine-based, hopefully, it will provide a flexible framework upon which to design useful profiling extensions. The following briefly discusses the time measurement capabilities in Cedar. Then four routine profiling situations, for which the initial version of Cedar profiling will be designed, are described.

3.4.1. Timing Capabilities

Before defining what profiling data will be kept by *cprof*, it is important to understand what can and cannot be measured in the Cedar environment. Timing measurements are of particular interest.

The *ctime* high-resolution timing facility for Cedar keeps process and task timing measurements [BELM87]. Execution times are kept for each computing resource on which a task can execute. In particular, execution times for the cluster are broken down individually for each CE. During periods of parallel operation, execution time can be measured separately on each CE. A more detailed discussion of the *ctime* facility is given in [Malo87].

Due to the nature of parallelism within a task, profiling is unable to determine when the task goes from sequential to concurrent operation and vice versa. This situation is indicated by * in Figure 2. Hence, the amount of time a routine spends in sequential or concurrent operation cannot be accurately determined. If we consider a single routine with some parallel loops containing no routine calls, only the elapsed time of the routine can be measured; cpu time, which depends on the level of parallelism, cannot be measured[4].

Execution time measurements are easily made at the beginning and ending of routines by inserting calls to a monitoring routine. In addition, the state of the cluster can be determined at these times; in particular, whether the cluster is executing in sequential or concurrent mode. Thus, although it is difficult to detect the change of state from sequential to concurrent, it is possible to determine the state upon entry to a routine. Different execution times can be calculated for a routine based on cluster state when the routine is called. For instance, in Figure 2, if concurrent execution is observed upon entering routine C, the elapsed time for C's execution can be accounted to the time C is executing when the cluster is operating concurrently.

[4] In this simple case, the elapsed time is just direct execution time.

3.4.2. Routine Profiling Cases

In Cedar, there are four different cases where profiling of a routine might be different. These cases are distinguished by the state of the task at routine entry and the concurrency in the calling routine at the time it calls some descendent routine. Again, in this analysis, we are restricting ourselves only to the profiling of individual tasks[5].

The situation being considered is that of a routine A calling a routine B which calls a routine C. From A's perspective, A is the "caller" routine and B is the "callee" routine. From B's perspective, B is a "callee" of some "calling" routine, in this case A, and a "caller" to any routines it might call, for example C. For our discussion, the triple identity of a routine as calling, caller and callee is resolved by defining these terms with respect to a point of reference. When the reference point is a particular routine, that routine is labeled the *caller* routine, the routine which called the caller is the *calling* routine, and all routines that the caller calls are *callee* routines. In our above example, if the referent point is routine B, A is the calling routine, B is the caller, C is the callee. In the discussion that follows, it is important to understand how calling, caller and callee routines are identified from the reference routine.

Two parameters are defined to describe the four profiling cases. Suppose our reference routine is B; i.e., B is the caller. When B is entered, the task is either in a sequential or concurrent state. This state is called the *caller entry* state of B. It is important to emphasize that the *caller entry* value depends on the state of the task.

The second parameter depends on the execution state of a routine when it calls another routine. Because the only source of concurrency in a task is a parallel loop, a task can be executing only one parallel loop concurrently at any time. This implies that only one routine can be executing a parallel loop concurrently at any time. If B calls C while executing one of its parallel loops concurrently, the parameter *callee call* is given the value of concurrent; otherwise, *callee call* is sequential. Notice, if B's caller entry is

[5] The profiling of multiple tasks together is more difficult to imagine. It is briefly touched upon in Section 5. Notice that this includes loop spreading which is also not allowed in this analysis.

concurrent, B cannot execute concurrently. In this case, all of B's callee calls will be sequential[6].

The caller entry and the callee call parameters are defined in reference to the caller. The caller looks at the state of the task when it is entered to determine its caller entry. However, the caller assigns the callee call depending on the its sequential/concurrent state, not the state of the task[7]. An additional parameter, *call arc*, is defined for identifying calling arcs, although it is not needed for specifying the four profiling cases. It takes on a value equal to the state of the task when the callee is called; i.e., the callee's caller entry. For example, if the caller–callee arc is B–C, call arc is equal to C's caller entry.

The four profiling cases to consider are:

caller entry	*callee call*	*comments*
sequential	*sequential*	1) concurrency is possible in caller before the call and after the return, but not at the time of the call 2) concurrency is possible in the callee 3) there is no simultaneous caller–callee execution
sequential	*concurrent*	1) the caller routine is concurrent when the call is made; i.e. the callee is called from a concurrently executing parallel loop in the caller 2) there is potential for simultaneous caller–callee execution 3) the callee must execute sequentially; since the caller is already executing concurrently, the callee cannot
concurrent	*sequential*	1) since the caller's entry state is concurrent, implying the caller must execute sequentially 2) since the caller is executing sequentially, all callee calls are sequential[8]
concurrent	*concurrent*	1) this case is invalid due to the fact that only one routine can execute a parallel loop concurrently at any time

The three valid cases are shown graphically for four processors in Figure 3. Notice that both the sequential–sequential and sequential–concurrent cases can occur in the same instantiation of a caller

[6] This may be a source of confusion. Remember that the callee call value is based on the concurrency within the caller routine.

[7] As a test of understanding, notice that the callee call value of B's call to C will not necessarily equal C's caller entry value from B.

[8] This case might be a source of confusion. Keep in mind that when talking about the callee call, we are interested in whether the calling routine is executing concurrently, not whether the task is necessarily concurrent.

routine.

S–S: Sequential – Sequential

The profiling strategy for the S–S case is the least complicated. The elapsed time taken by the callee call can be computed from time samples taken before and after the call. Because the caller is not executing simultaneously with the callee, this time can be subtracted 100% from the elapsed time of the caller to give a better approximation of the caller's direct time. Instead of performing this operation during execution, the sequential callee call execution time is instead accumulated separately from the caller's elapsed "sequential caller entry" execution time[9].

Elapsed execution time for the sequential call arcs are kept. The type of call arc, sequential or concurrent, depends on the state of the task when the callee call is made.

The number of sequential caller entries, sequential calls and traversals of each sequential call arc are counted. These counts will be kept separately from the associated concurrent counts.

S–C: Sequential – Concurrent

Because the caller entry is sequential, its exit will also be sequential. Thus, an elapsed time measurement for the caller is obtained as before. However, the concurrent calls raise the possibility for simultaneously executing routines: caller and callee, or callee and callee. Each concurrent call will take place on some CE, and each CE can only execute one concurrent call at a time. The elapsed execution time for a concurrent call is accumulated in the totals for the CE that made the call. Thus, the caller has a separate concurrent call time total for each CE. The execution time for concurrent call arcs is just a single value; it does not need to be kept on a per CE basis.

As in the S–S case, the count of sequential caller entries is kept. Additionally, per CE concurrent callee calls are maintained. Concurrent traversals of call arcs are also counted.

[9] When the caller's entry is sequential, its elapsed execution time is saved as "sequential caller entry" execution time. This is to distinguish it from the C–S case. Notice, again, that the caller can execute concurrently at other times in the routine. However, it is not doing so at the time the sequential call is made.

```
Sequential-Sequential, Sequential-Concurrent
--------------------------------------------

                         caller                 callee

                       0  1  2  3             0  1  2  3
sequential
 caller   --------> A: -
 entry                 |                _ sequential callee call
                       |               /
           caller      *-----------------> B: -   calee
           sequential                         |   sequential
           execution                          |   execution
                       *<-----------------    -
                       |
                       |
                       |
                       ----------         _ concurrent callee call
           caller      |  |  |  |        /
           concurrent  |  |  *--|--------------> C: -   callee
           execution   |  |     |                   |   sequential
                       |  |     |                   |   execution
                       |  |  *--|---------------    -
                       |  |  |  |
                       ----------
                          |
                          |
                          -

Concurrent-Sequential
---------------------

                         caller                 callee

                       0  1  2  3             0  1  2  3
concurrent
 caller   -----------------> A: -
 entry                          |                _ sequential callee call
                                |               /
           caller               *-----------------> B: -   callee
           sequential                                  |   sequential
           execution                                   |   execution
                                *<-----------------    -
                                |
                                |
                                -
```

Figure 8. Routine Profiling Situations in Cedar

C–S: Concurrent – Sequential

The profiling strategy here is similar to the S–S case. However, because the caller entry is concurrent, the caller must execute sequentially. A particular CE executes the caller routine plus all its called callee routines. Therefore, caller elapsed execution time measurement is relative to a particular CE, and is saved as "concurrent call entry" execution time for that CE. Likewise, the sequential call execution time is based on the particular CE and is saved as sequential callee time for the CE. As before, the concurrent call arcs have a single value.

In this case, per CE concurrent caller entries are updated for the caller. Likewise, a per CE count is designated for the sequential callee calls. As before, only a single count is kept for concurrent call arcs.

3.4.3. Profiling Data

Based on the above discussion of the different profiling cases, the profiling data kept for each routine and caller–callee arc are shown below.

Routines

Counts

sequential caller entry count
per CE (8) concurrent caller entry counts
sequential callee calls
per CE (8) sequential callee calls (C–S case)
per CE (8) concurrent callee calls (S–C case)

Execution Time

sequential caller entry execution time
per CE (8) concurrent caller entry execution time
sequential callee call execution time
per CE (8) sequential callee call execution time (C–S case)
per CE (8) concurrent callee call execution time (S–C case)

Caller–Callee Arcs

Counts

sequential call arc count
concurrent call arc count

Execution Time

sequential call arc execution time
concurrent call arc execution time

4. Implementation

The initial *cprof* implementation will support the Cedar profiling strategy discussed in the previous section. Although gathering as much useful information as possible might make the implementation less efficient, doing so will help point out inadequacies in the profiling methodology. The goal of the *cprof* implementation design, therefore, is to provide a flexible profiling framework for easy modification and extension.

This section outlines the *cprof* implementation design. First, the overall design approach is discussed. The source-level mechanism for profiling implementation, the strategy of profiling data structure allocation, and the system support requirements are described. The procedures for counting and timing are then presented[10]. The section concludes with proposals for handling particular implementation issues.

It is intended that *cprof* be provided as a compile-time option of Cedar Fortran. Because only source-level program modifications are allowed in Cedar Fortran, the *cprof* implementation will be supported in the Cedar Fortran preprocessor. Some subset of *cprof* will also be provided to the user for manual use.

4.1. Design Approach

The general *cprof* design approach consists of three parts: defining a mechanism whereby profiling operations at routine entry and exit are possible, understanding the allocation and use of profiling data structures, and identifying necessary profiling support operations. The details of data structure definition, counting operations, and timing operations fill out this general design framework.

[10] All data structures and profiling procedures presented in this document are written in a pseudo-C syntax

4.1.1. The Facade Routine

The general profiling mechanism will be to replace the textual routine call in the program with a call to a "facade" routine for the profiled routine. The facade routine will contain a profiling PROLOGUE and EPILOGUE separated by a call to the profiled routine. For instance, if routine A is to be profiled, all calls to A are replaced by calls to A's profiling facade A'. A' performs some profiling operations and then calls A. When A returns to A', A' performs some more profiling operations before returning to the calling routine. This is shown graphically in Figure 4.

The facade routine should actually be thought of as being defined for a caller routine. However, some profiling operations in the caller's facade will be on behalf of the calling routine. Likewise, callee profiling variables defined in the caller's facade will be updated by callee facade routines. With this understanding in mind, Figure 5 shows the structure of a caller facade routine that should be referred to for the remainder of this section.

The most important aspect of the facade routine-base profiling mechanism is that it can be implemented at the source-level by a compiler preprocessor. In fact, this is a necessary requirement for Cedar Fortran because of the inability to modify the assembly code.

```
     Original Code                                  Profiled Code
     -------------                                  -------------
caller           callee               caller             callee's facade        callee

  ~                                     ~
  ~                                     ~

call A   ----> A:  ~                  call A'   ----> A':  <PROLOGUE>
                   ~                                         call A     ----> A:  ~
                   ~                                                              ~
  ~     <----    return                                                           ~
  ~                                                        <EPILOGUE> <----     return
                                        ~       <----        return
                                        ~
```

Figure 4. Facade Routine Mechanism

Profiling will also include routines for initialization and exit. Profiling initialization is concerned with allocating appropriate data structures, initializing routine and call arc profiling variables, and setting profiling parameters. The normal exit routine will be replaced for program with profiling enabled by an exit profiling routine. This routine is responsible for saving profiling data in a profile output file.

```
FACADE(<original routine parameters>, <address of calling routine's profiling variables - calling^ >)
Prologue:
    temporary caller profiling variable declarations:
        <caller counting variables>
        <caller timing variables>
    prologue operations:
        <determine caller_entry state>
        <determine callee_call state>
        <determine executing CE - xCE>
        <prologue counting operations>
        <prologue timing operations>
Routine Call:
    call ROUTINE(<original routine parameters>, <address of caller's profiling variables - caller^ >)
Epilogue:
    epilogue operations:
        <prologue counting operations>
        <epilogue timing operations>
        <lock calling routines's temporary profiling variables>
        <update calling routines's temporary profiling variables>
        <unlock calling routines's temporary profiling variables>
        <lock caller's global profile variables>
        <update caller's global profile counts>
        <update caller's global profile times>
        <unlock caller's global profile variables>
```

Figure 5. Facade Routine Structure

4.1.2. Facade Routine Parameters

Given that the facade routine is the only profiling link between two routines, it must be able to update both caller and calling profiling variables. Updating the caller count and time variables is no problem since the facade routine knows the identity of the caller[11]. However, unless information regarding the identity of the calling routine is passed to the caller's facade routine, the calling routine's profiling variables cannot be updated. These variables include the calling routine's callee counts and execution times.

The simplest technique is to pass an ID of the calling routine that uniquely identifies it in the program. However, only the address of the calling routine's profiling variables are required for them to be updated. For this reason, and others described below, the address of the caller's profiling variables is passed to the callee's facade. It turns out that to update the calling–caller call arc profiling variables, the calling routine's ID is still required. Thus, the calling ID should be accessible to the caller's facade, probably as part of the calling profiling variables.

Because the original routine is called indirectly through the facade routine, the actual parameters must be transferred to the routine call. The facade will be called with the original parameters and these parameters are used to call the routine.

4.1.3. Profiling Data Structure Allocation

Knowing that the current *cprof* implementation is routine–based, we might allocate profiling variables globally for the routines and call arcs at the beginning of the program. There are two basic problems with this approach. The first is performance related. Because *cprof* intends to profile parallel programs, it is possible that a routine's profiling variables can be simultaneously accessed by two or more processors. Thus, mutual exclusive access to the profiling variables must be implemented to guarantee consistency. Mutual exclusive access is required for a routine's profiling variables during its facade PROLOGUE and EPILOGUE, and during every callee's facade PROLOGUE and EPILOGUE. Obviously, there is the potential for significant mutual exclusion ovcrhead. Some of the overhead can be reduced by allocating

[11] Remember, we defined the facade routine with respect to the caller.

profiling variables associated with callee routines locally and updating the associated global variables once during the caller's facade EPILOGUE.

The second problem with only global profiling variables is that temporary values are required to keep some dynamic profiling values, such as execution time. Allocating and sharing these temporary variables globally will significantly restrict the degree of parallel profiling in cases where multiple instantiations of certain routines are executing.

The propose solution to the profiling variable allocation problem consists of two parts. Profiling values for routines and call arcs representing the current profiled state will be allocated globally. Figure 6 shows the routine and call arc profiling data structures. However, temporary profiling variables are allocated in the facade routine, when a routine is called, to keep dynamic profiling information. When the routine finishes, its facade EPILOGUE will update the global profiled state of the routine using the temporary profiling data.

```
struct routine_profile
{       int     s_caller;          /* sequential caller entry count */
        int     c_caller[8];       /* concurrent caller entry counts */
        int     s_callee;          /* sequential callee call count */
        int     cs_callee[8];      /* C-S sequential callee call counts */
        int     c_callee[8];       /* concurrent callee call counts */
        TIME    s_caller_t;        /* sequential caller time */
        TIME    c_caller_t[8];     /* concurrent caller times */
        TIME    s_callee_t;        /* sequential callee time */
        TIME    cs_callee_t[8];    /* C-S sequential callee times */
        TIME    c_callee_t[8];     /* concurrent callee times */
};

struct call_arc_profile
{       int     s_callarc;         /* sequential call arc count */
        int     c_callarc;         /* concurrent call arc count */
        TIME    s_callarc_t;       /* sequential call arc time */
        TIME    c_callarc_t;       /* concurrent call arc time */
}
```

Figure 6. Routine and Call Arc Profiling Data Structures

This strategy has ramifications on passing of necessary information to the caller's facade routine in order to update the calling routine's profiling variables. The idea is that the address of the necessary calling temporary profiling variables will be passed to the caller's facade. Notice that the address of the temporary profiling variables declared in the facade routine must be passed in the actual routine call.

4.1.4. Support Operations

During profiling operation, it is necessary to determine state of the task at caller entry and the state of the caller when a callee routine is called. It is possible to quickly determine the state of the task by interrogating the state of the CEs. This will be implemented by an assembly language routine which is called at the beginning of the PROLOGUE. Calling this routine is all that is needed to determine the caller entry state. To determine the callee call state of the caller as a callee of the calling routine, the caller's facade routine must know the calling routine's caller entry state. From the calling routine's caller entry and the current state of the task, i.e. the caller entry state of the caller, the facade routine can determine the callee call state. If the calling routine's caller entry is sequential, the ta^sk state determines whether the callee call is sequential or concurrent. If the calling routine's caller entry is concurrent, the call to the caller routine must be sequential[12].

As described earlier, the *ctime* high-resolution timing facility will be used by *cprof* for timing operations. System routines are provided for determining the current time values for all the CEs. Only the user and system time values are used in the profiling timing operations.

4.2. Counting Operations

The counting events discussed earlier are straightforward to implement. The operations for caller entry counting and callee call count are described below. To avoid extraneous locking of the global profile count variables, all all global updates are done in the EPILOGUE.

[12] This use of terms is definitely a source of confusion. It helps to separate the identity of a routine, in the calling-caller-callee relationship described earlier, from the meaning of the caller entry and callee call states. For example, in the above description, the calling routine has its own caller entry state. The caller routine also has a caller entry state. It uses the caller entry state of its calling routine to determine its callee call state.

4.2.1. Caller Entry Counts

The defined profiling counts for caller entry are sequential caller entry and per CE concurrent caller entry. These counts are incremented only once depending on caller entry. Thus, the only operation that needs to be performed is an increment update of the appropriate global caller entry count. The caller entry count update operations are part of the caller's facade EPILOGUE and are shown below[13]:

Global Caller Entry Count Update

```
<lock caller's global profile variables>
...
if (caller entry == SEQUENTIAL)
                    <caller global>.s_caller++;
else
                    <caller global>.c_caller[xCE];
...
<unlock caller's global profile variables>
```

4.2.2. Callee Call Counts

The callee call counts are sequential callee call, per CE sequential callee call, and per CE concurrent callee call. Because the caller routine might call several callee routines, caller's callee call counts might be incremented more than once per caller entry. For this reason, temporary callee counts are declared and initialized in the PROLOGUE of the caller's facade as shown below.

Temporary Callee Call Counts Declaration and Initialization

```
int                 s_callee, cs_callee[8], c_callee[8];
LOCK                callee_lock;

s_callee = 0;
for (i=0; i<8; i++)
{                   cs_callee[i] = 0;
                    c_callee[i] = 0;
}
```

[13] <caller global> represents a pointer to the global profiling data structure for the caller routine.

Notice that the temporary callee call counts defined in a caller routine's facade are actually incremented in the facade PROLOGUE of the callee routines. Consequently, the caller routine's facade is responsible for incrementing the callee call counts for its calling routine. The required operations to update the calling routine's temporary callee call counts are[14]:

Temporary Callee Call Count Profiling Operations

```
<lock calling->callee_lock>
...
if (calling->caller_entry == SEQUENTIAL)
                if (caller_entry == SEQUENTIAL)
                        calling->s_callee++;
                else
                        calling->cs_callee[xCE]++;
else
                calling->c_callee[xCE]++;
...
<unlock calling->callee_lock>
```

Because it is possible to have concurrent routine calls, simultaneous access to the calling routines temporary callee counts can occur. Thus, a lock must be associated with these temporary variables, as shown and used above.

Although incrementing the caller's callee call counts takes place in the callee's facade, the update of the caller's global callee call counts occurs in the EPILOGUE of the caller's facade. Notice that in the case of sequential caller entry, both sequential and concurrent callee calls are possible and, therefore, both must be updated.

The update operation of the global callee count variables is shown below:

Global Callee Call Count Update

```
<lock caller's global profile variables>
...
if (caller_entry == SEQUENTIAL)
{               <caller global>.s_callee += s_callee;
```

[14] Keep in mind that the shown operations are described in reference to the caller's facade for updating its calling routine's temporary *callee* call counts. These operations also take place in the callee routines called by the caller for updating the caller's callee call counts.

```
                for (i=0; i<8; i++)
                        <caller global>.c_callee[i] += c_callee[i];
}
else
                for (i=0; i<8; i++)
                        <caller global>.cs_callee[i] += cs_callee[i];
 ...
<unlock caller's global profile variables>
```

4.2.3. Call Arc Counts

The calling–caller call arc counts are similar to the caller entry counts in that they are incremented only once per caller entry depending on the type of calling arc. No temporary profiling call arc count variables are required. The increment operation takes place as part of the global call arc update in the EPILOGUE of the caller's facade[15]. The identity of the calling routine must be known to correctly identify the call arc. The caller entry indicates whether the sequential or concurrent call arc count should be incremented. The call arc count update is shown below[16]:

Global Call Arc Count Update

```
<lock call arc's global profile variables>
 ...
if (caller_entry == SEQUENTIAL)
                <call arc global>.s_callarc++;
else
                <call arc global>.c_callarc[xCE];
 ...
<unlock call arc's global profile variables>
```

4.3. Timing

As discussed in the previous section, *cprof* uses the *ctime* high–resolution timing facility implemented in Concentrix. Only the user and system times for each CE are used to compute execution times. Timing operations exist both in the PROLOGUE and EPILOGUE of the facade routine. In the PROLOGUE, the caller's beginning time is marked. In the EPILOGUE, the time is again marked and the difference is added

[15] Notice that we are here considering the calling–caller call arc. Caller–callee call arc updates are done in the callee facade.

[16] <call arc global> represents a pointer to the global profiling data structure for the calling–caller call arc.

to the caller routine's global caller time depending on the caller entry.

Because the caller is a callee of its calling routine, the caller's facade EPILOGUE must also update the temporary callee time variables of the calling routine. Which variable is updated depends on the caller's callee call state. The caller and callee timing operations are described below.

4.3.1. Caller Timing

For each call to a routine, the elapsed execution time of the routine will be measured. As with caller entry counts, no temporary caller time variable is needed because the time difference between the beginning and ending marks will be added directly to the global caller time values in the update section of the EPILOGUE. However, temporary variables are needed to hold the beginning and ending time marks.

As described above, the current time is marked as the beginning time in the routine's facade PROLOGUE. The current time is again marked in the EPILOGUE to indicate the ending time. Marking the current time is easily done by a call to the appropriate *ctime* routine. However, computing the difference between the beginning and ending times to derive the elapsed execution time depends on the caller entry.

Let us take the easy case first. If the caller entry is concurrent, the caller routine and all of its descendant routines execute on a single CE; in fact, the same CE[17]. The caller routine is entered and exited on the same CE and all elapsed time is accounted to that particular CE. Thus, the elapsed time can be computed by subtracting the caller beginning time of that CE from its ending time. The elapsed time is added to the caller's global concurrent caller times for that CE.

If, however, the caller entry is sequential, it is not guaranteed that the entry CE is the same as the exit CE. This can occur by the execution of parallel loops in the caller routine or in any of its descendant routines. In this case, the elapsed time is calculated by taking the maximum difference between the beginning and ending times for each CE[18]. This maximum elapsed time is added to the caller's global sequential

[17] Notice that parallel loops must execute sequentially because the caller entry is concurrent.

[18] Because of the way the *ctime* facility is implemented, idling CEs during sequential operation are charged user time. Therefore, one might assume that it would not matter which CE was chosen for computing elapsed time. However, because overhead and kernel time are also kept for each CE, the individual CE time differences can be different.

caller time.

The temporary declarations and operations for caller timing are shown below:

Caller Timing Operations

```
PROLOGUE:
        TIME            begin_mark, end_mark, elapsed;

        get_current_time(begin_mark);

EPILOGUE:
        get_current_time(end_mark);
        if (caller_entry == SEQUENTIAL)
        {               <determine CE with maximum time difference>
                        <save this difference in elapsed time variable>
        }
        else
        {               <calculate time difference for CE xCE>
                        <save this difference in elapsed time variable>
        }

        <lock caller's global profile variables>
        ...
        if (caller_entry == SEQUENTIAL)
                        <caller global>.s_caller_t += elapsed;
        else
                        <caller global>.c_caller_t[xCE] += elapsed;
        ...
        <unlock caller's global profile variables>
```

4.3.2. Callee Timing

Callee timing is really not much more complicated than callee counting. When a routine is called, it is both a caller to any routines it might call, as well as, a callee to its calling routine. We are interested here in its role as a callee. In this respect, the routine's facade is responsible for updating the callee profiling variables of its calling routine[19]. It turns out that the elapsed time value computed above for the routine's caller timing is exactly the time value required for updating the callee time of its calling routine. The difference, however, is that the choice of which callee time to update depends on the caller's callee entry, as was the case in the callee counts.

[19] It is also responsible for the call arc timing to be described later.

Temporary callee time variables must be declared in a routine's facade because several callee routines might be executed. Because of the possibility of concurrent routine execution, the callee lock, defined above for callee call counts, must be used to guaranteed mutual exclusion when the caller's facade performs the callee time update of its calling routine.

Thus, assuming that the elapsed caller time has been computed in a caller's facade EPILOGUE, as shown above, the callee time update operations for the calling routine are:

Temporary Callee Time Profiling Operations

```
<lock calling->callee_lock>
...
if (calling->caller_entry == SEQUENTIAL)
                    if (caller_entry == SEQUENTIAL)
                              calling->s_callee_t = elapsed;
                    else
                              calling->cs_callee_t[xCE] = elapsed;
else
                    calling->c_callee_t[xCE] = elapsed;
...
<unlock calling->callee_lock>
```

The update operation of the caller's global callee time variables is shown below:

Global Callee Time Update Operations

```
<lock caller's global profile variables>
...
if (caller_entry == SEQUENTIAL)
{                   <caller global>.s_callee_t += s_callee_t;
                    for (i=0; i<8; i++)
                              <caller global>.c_callee_t[i] += c_callee_t[i];
}
else
                    for (i=0; i<8; i++)
                              <caller global>.cs_callee_t[i] += cs_callee_t[i];
...
<unlock caller's global profile variables>
```

4.3.3. Call Arc Timing

The call arc times are a measure of execution time spent between the calls and returns along call arcs. Obviously, the time for a particular call arc traversal is the elapsed execution time of the called routine. The call arc time is accumulated in the global call arc time for the calling–caller pair. This occurs as part of the call arc update in the EPILOGUE of the caller's facade. The identity of the calling routine must be known to correctly identify the call arc. The caller entry indicates whether the sequential or concurrent call arc time should be incremented. The call arc time update operations are shown below assuming the elapsed time has already been computed:

Global Call Arc Time Update Operations

```
<lock call arc's global profile variables>
...
if (caller entry == SEQUENTIAL)
                <call arc global>.s_callarc_t += elapsed;
else
                <call arc global>.c_callarc_t[xCE] += elapsed;
...
<unlock call arc's global profile variables>
```

4.4. Implementation Issues

The profiling operations described above give the general implementation strategy for the initial version of *cprof.* However, particular implementation issues must be addressed before an implementation can be realised. Two are discussed below, although, in actuality, there are many such issues.

4.4.1. Timing, Recursion, and Concurrency

When determining the elapsed time for a routine, any computation between the entry and exit of the routine is included in the elapsed time value. At exit, the elapsed time is added to the overall execution time of the routine. However, if the routine is called recursively, directly or indirectly, the exit from the recursive call will result in the elapsed time of the recursive call being added to the routine's overall execution time. When the outermost call to the routine finally exits, it too with update the overall execution time with its elapsed time which includes the elapsed time of the recursive call. Hence, the elapsed time of the recursive call is counted twice in the routine's overall execution time.

To avoid this problem, the ELXSI profilers keeps a count for each routine of the number of times the routine has been entered but not exited; i.e. the recursive call level. If a routine is called recursively, the profiler notices that the entry count is >1 and avoids updating the overall execution time at recursive exit. However, this is the only operation that is not executed. The elapsed execution time of the recursive call must still be reported back to the caller.

In *cprof*, the recursion timing problem is exacerbated. If the task is executing sequentially when a routine is called recursively, the technique described above for timing recursive calls will work fine. The complication occurs when different execution threads call the same routine. In some cases, the calls are treated as concurrent calls to the same routine. In other cases, the calls should be regarded as recursive calls. Identifying when a call to a routine during concurrent operation is a recursive call or a concurrent call is at the heart of the problem.

A concurrent call to the same routine is one where two or more execution threads, i.e. two or more CEs, call the same routine and are simultaneously entered in that routine. Generally, the timing of concurrent calls to the same routine slips nicely into the existing *cprof* profiling operations. Each separate call computes its own elapsed execution time and updates the overall routine execution time value for the CE executing the call. If, however, a particular CE calls the routine recursively, this recursive call should be recognized even though concurrent calls may exist. As before, the execution time for the recursive call should not be added into the overall execution time for the routine for this CE.

The problem is how to recognize a concurrent call from a recursive call in the routine's facade so that the correct overall execution time update can occur. The proposal is to extended the standard solution to maintain a separate entry count for each possible execution thread entry; e.g., given 8 CEs, 8 entry counts are defined for a routine – one for each CE. At entry to the routine's facade, the CE is identified and the entry count for that CE is incremented. Before updating the routine's overall execution time, the facade checks the appropriate CE entry count to determine it this was a recursive call. If not, the overall execution time can be updated.

It is desired to have the procedure described above work for both sequential and concurrent operation. However, there are few minor complications that must be resolved. First, it is possible for a routine to be entered sequentially on one CE and exited on another. This is due to concurrent operation within the routine or one of its descendants. If the routine is called recursively, the facade entry increments the entry count of the entering CE but the facade exit decrements the entry count of the exiting CE. Notice that this problem cannot occur when the routine is entered concurrently, though a similar problem does occur. If the routine had been previously entered sequentially, when the routine is recursively entered during concurrent operation, it might not be executing on the same CE as the one that entered it sequentially.

The different recursive call situations that can occur are shown graphically in Figure 7 and 8 The first case obviously works with our proposed strategy. The recursive call facade entry will be recognized and the update bypassed at recursive call facade exit. The second case has noticeable problems because of the recursive call exits on a different CE. It the proposed strategy is applied literally, just before the recursive call exit, the entry count is 2 for CE 0 and 0 for CE 1. When the facade interrogates the entry counts at exit, it sees an invalid value for CE 1.

The solution for the above problem is the same as the solution to case three. Here a recursive call enters on a different CE. The entry counts at this point are 1 for CE 1 and 1 for CE 2. Thus, the facade observes no recursive entry. To solve these two problems, the following policy is adopted. The facade of a routine called sequentially, i.e. the task is sequential, will increment all CE entry counts at entry and decrement all CE entry counts at exit. Notice that this solution is still consistent with the first case.

The fifth case, Figure 8, poses no problem for the recursion detection strategy. This is similar to the normal case one because only the execution trace of one CE is being considered for recursion. Thus, each separate CE execution thread is deciding whether routine calls along that thread are recursive.

The fourth case is the worst of all. A routine is called recursively along different execution threads. Notice that if the calls were not concurrent, as shown here, the case would degenerate into case three. However, the simultaneous recursive execution suggests a different state of operation that maybed requires

```
                           CE                     CE
                       0  1  2  3             0  1  2  3
non-recursive
sequential -------> A: -
call                   |                _ recursive sequential call
                       |               /
                       *-----------------> A: -           ① normal case
                                              |
                       *<----------------     -
                       |                _ recursive sequential call
                       |               /
                       *-----------------> A: -
                                              |           ② recursive
                                              ----------    call exits on
                                              |  |  |  |    different CE"
                                              ----------
                                                 |
                          *<---------------      -
                          |
                       ----------             _ recursive concurrent call
                       |  |  |  |            /
                       |  |  *--|--------------> A: -      ③ recursive
                       |  |     |                   |        call enters on
                       |  |  *--|---------------    -        different CE
                       |  |  |  |
                       ----------
                             |
                             |
                       ----------
                       |  |  |  |
                       |  |  *--|---------->     A: -      ④ concurrent
                       *--|-----|---------> A: -    |        recursive
                       |  |  *--|----------    |    -        calls on
                       |  |  |  |              |             different CE's
                       *--|--|--|----------    -
                       |  |  |  |
                       ----------
                       |
                       -
```

Figure 7. Recursive Call Situations – Non-Recursive Sequential Entry

```
                           CE                      CE
                        0  1  2  3             0  1  2  3
non-recursive
concurrent -----------------> A: -
call                             |                _ recursive concurrent call
                                 |               /
                                 *----------------> A: -   ⑤ similar to
                                                       |      normal
                                 *<-----------------   -      situation
                                 |
                                 -
```

Figure 8. Recursive Call Situation – Non–Recursive Concurrent Entry

a different type of timing analysis. However, the proposal for this situation is to equate it with the third case, essentially disregarding the concurrent overlap of execution. Otherwise, yet another set of variables to time this particular case would be required.

4.4.2. Selected Profiling

Currently, the profiling implementation approach using the facade routine requires every routine that calls a profiled routine to itself be profiled. Although the approach does allow a user to ignore low–level routines in the profiling analysis, the ancestral profiling restriction can be cumbersome. The basic problem is one of transferring profiling information from a profiled ancestor to a profiled descendant through the arguments lists of routines along the call arcs. These routines must necessarily be modified to perform the appropriate parameter passing operations.

Future *cprof implementations might discover a better mechanism for capturing entry and exit from* profile routines, other than the facade approach. The current restriction to this effort is the fact that the compiler can only make source level transformation. However, other schemes are currently being investigated.

5. Extensions

This section briefly describes extensions of the initial version of *cprof*. As stated earlier, parallel program profiling offers a much larger domain of potentially interesting analysis than sequential profiling. A particularly powerful profiling mechanism is tracing. Several different types of measurements can be obtained in this manner. After discussing the capabilities of tracing, extensions particular to inter-task profiling in Cedar are described. The implementation design of the profiling extensions outlined in this section will be specified in future reports.

5.1. Tracing

The basic idea behind tracing is that at certain points during program execution, various values are written to a buffer, possibly time-stamped, so time-ordered program behavior can be observed after execution has completed. Anything can be designated to be a part of the trace output. This makes tracing a powerful mechanism because virtually any other type of profiling scheme can be realized through tracing. For example, all *cprof* profiling values described in the preceding sections can be reproduced from a sufficiently detailed trace. The problem with tracing is its appetite for storage; traces can become very large very fast. Appropriate uses of tracing identify particular parameters of interest that can be recorded efficiently in a reasonably-sized trace buffer. Several possible tracing tools for Cedar are discussed below.

A particularly interesting idea, especially for parallel programs, is the generation of a dynamic routine call graph showing the order in which routines are executed, the times routines are entered and exited, and the routine execution activity on the different processors. Essentially, the trace consists of time-stamped routine entry and exit events showing the name of the routine and the CE executing the routine. Obviously, some of the information gathered in the profiling operation described earlier would appear in the trace. Instead of writing the profiling information to the global and temporary profiling variables, however, the data is written to a trace buffer[20]. From the trace output, the dynamic routine call graph is

[20] The trace buffer is necessarily an internally defined array of storage. Clearly, an external trace file would significantly intrude on the program's operation.

created.

Clearly, there are situations where the dynamic execution trace described above will take too much storage. There are several ways to remedy this situation. First, not all routines need to be trace; the user can selectively choose which one to monitor. Second, some of the trace data may be unnecessary. For instance, to just observe routine call sequence or concurrent routine execution overlap, the time-stamp can be removed. Another efficiency measure might be to encode some of the trace data or use difference time-stamping.

Traces do not have to be routine-based. As indicated earlier, concurrent program execution is sometimes inadequately measured by routine call counts and execution times. Part of this has to do with the limitations of detecting concurrent operation in the Cedar machine. However, if this could be done, tracing might provide a nice interface for observing concurrency during the program's lifetime. Only those points where changes in the concurrency level occurs would need to be traced.

A possible implementation approach for tracing is to provide a general tracing mechanism that inserts trace code at various places in the program. The user specifies the type of trace data to be collected and where tracing should occur. Initial versions of a tracing tool will support particular types of tracing like dynamic routine call or concurrency tracing.

5.2. Inter-Task Profiling

The Cedar machine supports multitasking of parallel programs. All of the preceding profiling discussion has been directed towards intra-task profiling. The reason for this was partly to remove unnecessary complexity from the explanation of the profiling operations. There are also certain fundamental measurement issues that do not lend themselves to the implementation employed for intra-task routine profiling. The following briefly analyses the general problem of inter-task profiling. Some ideas for profiling tools to capture interesting inter-task behavior are then described. It is assumed that the reader has knowledge of how multitasking works in the Cedar system.

There are two basic problems with inter-task profiling. The first is timing between tasks. Although an elapsed program time, called process time, is maintained by the *ctime* timing facility, it cannot be used to time events within a task. If a task bases all of its timing operations on the process time, it cannot separate the time spent actually running on a cluster from the time spent waiting to run. This is due to the fact that tasks are independently scheduled in the Cedar system. Hence, all profiling timing operations described earlier must be done with the task times.

The second problem concerns observing concurrency across tasks during program execution. Because tasks can execute on physically separate clusters, determining if two or more tasks are concurrently executing requires some form of global communication and monitoring. Unfortunately, this feature cannot be provided just through program modification; support from the operating system is required.

The above problems imply a basic inability to extend the low-level intra-task profiling approach across tasks. However, there are several interesting measurements that cannot be derived from the low-level data that can be obtained with higher-level inter-task profiling operations. Mostly, the events of interest regard task creation, task execution, task synchronization, and other task related functions. The following considers a general type of inter-task profile and adds to it in a couple of ways to show variations on the same idea.

Suppose we want to produce an program task execution graph similar to dynamic routine call graph described above. Generally, the task execution graph shows events related to task execution. These task events are recorded in a program trace and include the type of event, the time of the event, and other related information. Because the program execution trace must be ordered with respect to a single program time, the process time should be used for time-stamping all events[21].

Most task events are identified by the task operations provided by the Cedar Fortran run-time library and the Xylem operating system. For instance, an event could be defined for the Cedar Fortran function *ctskstart* that includes the time the new task is created and the number of processors requested.

[21] Notice that there is no problem using process time in this manner; only a time-stamp is being recorded, not an elapsed time value.

A *ctskwait* event might record the time the wait began and the task being waited on. Another type of *ctskwait* event could record the time the when the suspended task continues. The Xylem multitasking system calls, *create_ctask*, *delete_ctask*, *queue_ctask*, *resume_ctask*, *start_ctask*, *stop_ctask*, *suspend_task*, and *wait_task*, also map to obvious task events. A program task execution graph showing the occurrence of these events during program execution is useful for identifying the number of active tasks and waiting relationships.

Defining events for Cedar synchronization operations further increases the capabilities of the inter-task profiler. Identifying synchronization in the program execution graph is important because it shows dependencies within the program that can limit parallelism. The events should not only indicate the type of synchronization and the time it occurred, but also the task identity, the synchronization variable, and the outcome.

Because of the independent scheduling of tasks, the process time time-stamps recorded by the program will be affected by the task scheduling. A task can be context-switched between two time recordings of the process time. Looking only at the two time-stamps, the amount of time the task actually spent executing between these two events cannot be determined. If, however, the process times when the task is context-switched and when the task resumes execution are recorded in the program trace, the interval execution time can be determined. Although this time can also be calculated using only the individual task's times, what cannot be observed is the task's execution relationship with the other tasks. The problem is that task context-switch and task resume times must be recorded by the operating system.

Assuming the operating system is able to provide the necessary task timing functions, several interesting inter-task profiling analyses can be obtained for the program execution trace. For instance, the task concurrency present during the program's lifetime can be observed. This information may be able to shed some light on reasons for the programs' achieved performance. The trace can also show the effects of synchronization on task waiting and suspension. Again, certain problem areas might be more readily identified.

A particularly interesting analysis, tentatively called *program flow trace analysis*, uses the task synchronization and execution information to construct a "minimum-time" program execution graph. The analysis attempts to remove any task scheduling effects due to Cedar multiprogramming, making it appear as if the program had the entire machine to itself. The resulting program execution graph is a maximally compressed version of the original trace such that all inter-task dependencies, based on the synchronization event trace information, remained ordered in time. Thus, the execution flow trace graph shows a minimum-time in the sense that extraneous delays caused by tasks being context-switch, for reasons other than task synchronization, are removed. Obviously, the analysis gives only an approximation of execution on a single-user machine. However, it will be definitely be useful for estimating poor performance due to system load and inter-task dependencies.

References

[BELM87] R. Barton, P. Emrath, D. Lawrie, A. Malony, R. McGrath. *New Approaches to Measuring Process Execution Time in the Cedar Multiprocessor System.* **CSRD Report #744**, Jan. 1987.

[Carr86] D.A. Carrington. *Profiling Under ELXSI UNIX.* **Software – Practice and Experience**, Vol. 16, No. 9, pp. 865-73, Sept. 1986.

[GKMc82] S.L. Graham, P.B. Kessler, M.K. McKusik. *gprof: a Call Graph Execution Profiler.* Proc. SIGPLAN '82 Symp. on Compiler Construction, **SIGPLAN Notices**, Vol. 17, No. 6, pp. 120-26, June 1982.

[GKMc83] S.L. Graham, P.B. Kessler, M.K. McKusik. *An Execution Profiler for Modular Programs.* **Software – Practice and Experience**, Vol. 13. pp. 671-85, 1983.

[Malo86] A.D. Malony. *Virtual High Resolution Process Timing.* **CSRD Report #616**, Oct. 1986.

[Malo87] A.D. Malony. *The CTIME User's Manual*, Feb. 1986.

[UNIX84] UNIX User's Manual. 4.2 Berkeley Software Distribution, Univ. of California, Berkeley, March 1984.

END
DATE
FILMED
5-88
DTIC