Journal of Parallel and Distributed Computing **61**, 115–136 (2001)
doi:10.1006/jpdc.2000.1667, available online at http://www.idealibrary.com on IDEAL®

Performance Optimization Using Extended Critical Path Analysis in Multithreaded Programs on Multiprocessors

Magnus Broberg, Lars Lundberg, and Håkan Grahn

Department of Software Engineering and Computer Science, Blekinge Institute of Technology, Soft Center, S-372 25 Ronneby, Sweden
E-mail: Magnus.Broberg@ipd.hk-r.se, Lars.Lundberg@ipd.hk-r.se, Hakan.Grahn@ipd.hk-r.se

Received November 22, 1999; revised May 24, 2000; accepted May 25, 2000

Efficient performance tuning of parallel programs is often hard. Optimization is often done when the program is written as a last effort to increase the performance. With sequential programs each (executed) code segment will affect the completion time. In the case of a parallel program executed on a multiprocessor this is not always true, due to dependencies between the different threads. Thus, certain code segments of the execution may not affect the completion time of the program. Optimization of such code segments will not increase the performance. In this paper we present an approach to optimize performance by finding the extended critical path of the multithreaded program. The extended critical path analysis is a generalization of the critical path analysis in the sense that it also deals with more threads than processors. We have implemented the extended critical path analysis in a performance optimization tool. The tool allows the user to determine the extended critical path of a multithreaded application written for the Solaris operating system for any number of processors based on execution on a single processor workstation.

Key Words: Multithreading; multiprocessor; critical path analysis; performance optimization; performance analysis.

1. INTRODUCTION

Parallel processing is an important way to increase the performance of computationally demanding applications, and applications developed for multiprocessors are likely to have high performance requirements. However, it is not always easy to write parallel applications for multiprocessors. The application developers need support in order to write parallel applications with high performance.

A number of commercial multiprocessor platforms have emerged. The developer must make sure that the application runs efficiently on different numbers of processors, since it is the customer that decides the size of the multiprocessor based on

0743-7315/01 $35.00

the actual performance requirements and the price/performance ratio. In some cases, the developers want the multithreaded application to scale-up beyond the number of processors available in a multiprocessor today in order to meet future needs. Thus, there is no single target environment and the development environment may not be the same as the target environment. Often the development environment is the (single processor) workstation on the developer's desk.

Just like the code in sequential applications, the code in parallel applications can be optimized in order to reduce the completion time of the application and thus increase the performance. The optimization is done by reducing the execution time for certain code segments, preferably the code segments that have the largest impact on the completion time of the application. In the case of a sequential application, all executed code segments contribute to the completion time. When executing a parallel application on a multiprocessor all code segments may not contribute to the completion time. Therefore, it is hard for the developer to know which code segments will actually reduce the completion time of the parallel application, thus making it hard to prioritize the optimization efforts.

Consider the program in Fig. 1. We assume that all functions (a–d) take an equal amount of time to execute and that signaling takes no (or negligible) time. When executing the program on an SMP (symmetric multiprocessor) with three processors (or more) the execution will look like Fig. 1. Imagine that function `a( )` is optimized with a small value ε; then the completion time will be 2ε shorter. Further, imagine that function `b( )` is optimized with a small value ε, then there will be no reduction in the completion time at all. When optimizing functions `c( )` or `d( )` with a small value ε the reduction will be equal to ε. Thus, optimizing function `a( )` will have the largest impact. However, when executing the program on one processor, function `b( )` will be the function with the largest impact (3ε).

The functions execute for the same amount of time in this example. Neither different execution length of the different functions nor different execution length for different invocations of the same function (due to, e.g., different input parameters) are limitations in our technique (see Section 3.2).

We define the *extended critical path* as all the executed code segments of a program that, when reduced with a small ε, will reduce the completion time on

```
Thread 1:                Thread 2:                Thread 3:
Execute a()              Execute b()              Wait for event X
Signal event X           Wait for event Y         Execute c()
Execute b()              Execute d()              Signal event Y
Wait for event Z         Signal event Z           Execute b()
Execute a()              End                      End
End
```

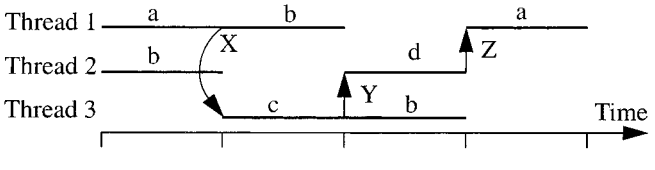

FIG. 1. A simple program with three threads and the execution on a multiprocessor with three processors.

a given number of processors. Consequently, all code segments of a program executed on a single processor will be considered as part of the extended critical path. This is because if we reduce *any* part of a program on a single processor it will result in a shorter completion time.

Ordinary profiling tools, such as Quantify [13], only consider the (extended) critical path in the case of one processor. They will indicate that the program in Fig. 1 spends most of the time in function `b( )` and suggest that function for optimization. This is shown in Fig. 2 where the functions `a( )` to `d( )` are found at the top; the other functions shown are only for internal use in the Solaris operating system. The tools and methods described in [3, 8, 18] have critical path analysis and will show that `a( )` is the most beneficial to optimize. However, those tools and methods assume that there is only one thread (or process) scheduled on each processor.

Our approach with the extended critical path generalizes the two extreme cases above. Not only in the case of one processor for *all* threads, or one processor for *each* thread, but *for any number of processors regardless of the number of threads*. Thus, with our approach it is also possible to analyze the application in Fig. 1 for the case of two processors. The techniques presented here do not require access to a multiprocessor. The approach used is to trace the behavior of a parallel application on a single processor workstation. Using these recordings, information about the performance of the parallel application on a multiprocessor with an arbitrary number of processors is obtained by simulation. As a proof of concept, a tool called visualization and predication of parallel program behavior (VPPB) has been modified to show that the methods are applicable in a real world programming environment.

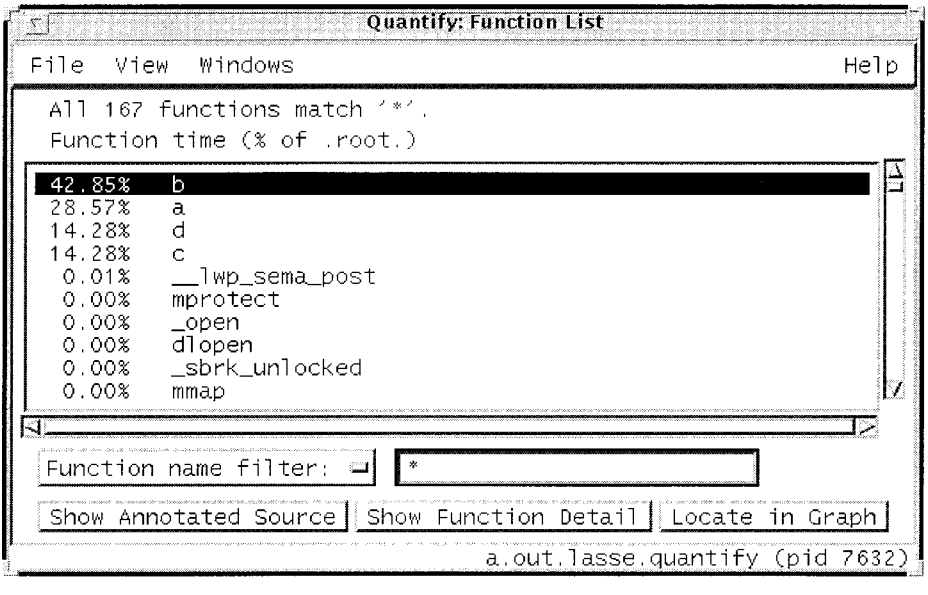

FIG. 2. Quantify's list of the functions to optimize.

The paper is structured as follows: Section 2 describes the algorithm for calculating the extended critical path. In Section 3 we present a working tool with the extended critical path analysis implemented and show by three practical examples in Section 4 the need for such a tool. Discussion is found in Section 5, and related work is found in Section 6. In Section 7 future work is discussed. The conclusion is found in Section 8.

2. THE EXTENDED CRITICAL PATH ALGORITHM

In this section we describe how the extended critical path algorithm works. First, we describe the algorithm for identifying the extended critical path in the special case with one thread per processor. Then, we describe how to calculate the time spent by each function executed in the extended critical path. Finally, we show how to calculate the extended critical path for multithreaded programs with more runnable threads at the same time than there are processors.

2.1. Finding the Extended Critical Path with One Thread per Processor

We start to look at the extended critical path algorithm under the condition that the program is executed on a sufficient number of processors and thus no threads have to share a processor at any time. This is the same assumption used in [3, 8, 18]. We assume that explicitly synchronized events happen at the exact same time, such as one thread releasing another thread. This means that a thread starts to execute at exactly the same moment as the other thread releases it. This assumption is realistic, does not change the algorithm in principal, and is kept here in order to simplify the presentation. A *segment* is the execution for a thread between two synchronizations, in this case we also regard the beginning and the end of the thread as a synchronization. The algorithm is found in Fig. 3 in pseudo-code.

The algorithm starts at the end of the execution and follows the last thread, called *i*; it continues backward until the thread *i* was blocked for some reason. Then the algorithm finds the event by thread *j* that released thread *i*. The algorithm then continues to follow the releasing thread *j* backward until it also was blocked. Thus, in that manner the algorithm continues until the start of the program. All code segments of the program that the algorithm goes through are part of the extended critical path.

```
segment = find_last_executing_segment();
stop = find_first_executing_segment();

mark_for_critical_path(segment);
while(segment != stop) {
  previous_segment = find_previous_segment_for_the_same_thread(segment);
  if(start_time_for(segment) == end_time_for(previous_segment)) /* Was blocked? */
    segment = previous_segment; /* Not blocked */
  else
    segment = find_segment_with_end_time_equal_to_start_time_of(segment); /* Blocked */
  mark_for_critcal_path(segment);
}
```

FIG. 3. Pseudo-code for finding the critical path with unlimited number of processors.

To illustrate the algorithm we use the example program with three threads in Fig. 4. The execution times for the different functions are illustrated in the lower part of Fig. 4, where also the execution of the program on three processors is illustrated.

Following the algorithm described above we start at the end of the execution, i.e., thread 3 at time 11. Tracing thread 3 backward makes us go through functions `f( )`, `a( )`, `h( )`, and finally `g( )`. At time 4 to 5 the thread was not executing; i.e., it was blocked. The thread (3) was waiting for event *Z*. Thread 1 releases thread 3 at time 5; thus we continue the algorithm with thread 1. Thread 1 executes through functions `b( )` and `a( )` without being blocked until the start of the program at time 0.

Thus the extended critical path for the program is thread 1 executing functions `a( )` and `b( )` and thread 3 executing functions `g( )`, `h( )`, `a( )`, and `f( )`. It is easily verified that optimizing any *other* part of the program will not affect the completion time when using three processors.

2.2. Calculating Function Execution Times Contributing to the Extended Critical Path

In the previous section, all the synchronizations were done between the function calls. This is not always the case, since synchronizations could appear inside the functions as well. This complicates the calculation of the time spent in certain functions during the extended critical path. This calculation is done after the extended critical path analysis; thus we only concentrate our effort on those parts that are in the extended critical path. We will calculate three values *per function*:

```
Thread 1:
Execute a()
Signal event X
Execute b()
Signal event Z
Execute c()
Wait for event X
Execute e()
Execute i()
End
```

```
Thread 2:
Wait for event X
Execute c()
Signal event Y
Execute g()
Wait for signal Y
Execute d()
End
```

```
Thread 3:
Wait for event Y
Execute d()
Wait for event Z
Execute g()
Execute h()
Signal event X
Execute a()
Signal event Y
Execute f()
End
```

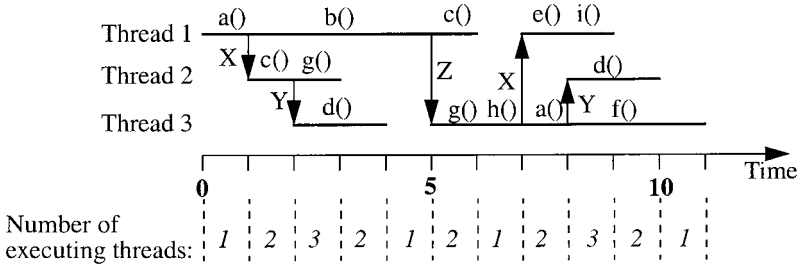

FIG. 4. The optimal execution of the program with three threads. The values in *italic* indicate the number of executing threads.

- The number of calls. This is the number of times the function was called during the extended critical path.
- The total execution time (TET). This time is the accumulated time, in the extended critical path, the program was executing in the function.
- The total execution time including all descendant functions (TETID). This time is the accumulated execution time for the function and its descendants during the extended critical path.

The first value is determined by calculating the number of function entrances along the segments in the extended critical path. The other two values are calculated by re-creating the function call stack. For each segment in the extended critical path, the function on the top of the stack will increase its TET with the length of the segment. The other functions on the stack will increase its TETID with the length of the segment. The re-creation of the function call stack is needed since the function entrance may not be in the extended critical path, but parts of the execution in the function may be.

2.3. Finding the Extended Critical Path with CPU Constraints

In Section 2.1 we defined the algorithm for finding the extended critical path when an unlimited number of processors are available. Unfortunately, this is not always the case in real life. The work done in [3, 8, 18] does not address that. Thus, we need to adjust the algorithm to fit whenever there are more runnable threads than there are processors. In that case, some threads at some times must be multiplexed by the scheduler on one processor.

We keep the example in Fig. 4, but look at what happens if we only have two processors available. The number of executing threads is indicated for each time unit as the figures in *italic*. The interesting parts are when the number of threads is larger than the number of processors. In the example this is time 2 to 3 and time 8 to 9 where three threads are executing on the two available processors. One way of modeling the multiplexing is to consider the processors to run more slowly as the number of threads increases. The processors run at only 2/3 speed for time 2 to 3 and time 8 to 9; i.e., we assume that the scheduling is ideal with infinitely small time slices and no scheduling overhead. This assumption has previously been used in some performance prediction tools with accurate result when compared with real scheduling [7].

In Fig. 5 pseudo-code for finding the extended critical path is found. Before explaining the algorithm we define two different kinds of segments:

- A *segment* is the execution for a thread between two synchronizations, in this case we also regard the beginning and the end of the thread as a synchronization, using one processor per thread. In Fig. 4 each "execute" statement is a segment.
- A *time segment* is the time period when no threads synchronize or are started or finished in the optimal execution. In Fig. 4 each time unit is a time

```
calculate_execution_time() {
  exec_time = 0;
  for_each_time_segment {
    exec_time = exec_time + length_of_time_segment * max(number_of_threads/P, 1);
  }
  return exec_time;
}
calculate_extended_critical_path() {
  original_execution_time = calculate_execution_time();
  for_each_segment {
    decrease_segment_execution_time_with(epsilon);
    set_segment_weight((original_execution_time - calculate_execution_time())/epsilon);
    increase_segment_execution_time_with(epsilon); /* to restore the execution time */
  }
}
```

FIG. 5. Pseudo-code for finding the extended critical path with limited number of processors. P is the (limited) number of processors available.

segment. Consequently, since we assume that no events happen exactly at the same time, the number of time segments is equal to the total number of segments for each thread.

The first function (`calculate_execution_time`) in Fig. 5 calculates the time required to execute the application. First we divide the execution into time segments. The time required for executing such a time segment depends on the number of currently executing threads. If the number of executing threads during the time segment is less than or equal to the number of processors available then the execution time is the same as the time segment length. When there are more threads than processors, the time required to execute that time segment will be longer. How much longer is given by the number of threads divided by the number of processors. This is then multiplied with the time segment length in order to get the executing time of the time segment.

The second part of Fig. 5 (`calculate_extended_critical_path`) describes how to identify the segments that are in the extended critical path. First the total execution time required is calculated. Then, for each segment, the execution time is calculated if that segment is shorted by a small value ε, less than the shortest time segment above. In Fig. 4, this corresponds to less than one time unit. If the execution time is not affected by this change, the segment is not part of the extended critical path. In other cases the segment is part of the extended critical path. By dividing the difference of the execution times with the selected ε we get a weight of how much impact the segment has on the extended critical path.

Following the algorithm in Fig. 5 the extended critical path for the application in Fig. 4 when executed on two processors will be: thread 1, `a( )`, `b( )`, `e( )`, and `i( )`; thread 2, `g( )`; and thread 3, `g( )`, `h( )`, `a( )`, and `f( )`. The path `a( )` and `b( )` on thread 1 and `g( )`, `h( )`, `a( )`, and `f( )` on thread 3 is the critical path for an unlimited number of processors as discussed in Section 2.1. The execution time is found in Fig. 6a and is 12 time units. If we, e.g., select function `g( )` on thread 2, the algorithm will shorten the execution time for `g( )` on thread 2 with a small value, e.g., one half time unit. Then the execution will look like Fig. 6b. The time segment where `g( )` is included is now from time 2 to 2.5 and the following time segment is from time 2.5 to 4. The resulting execution time is now 11.75 and

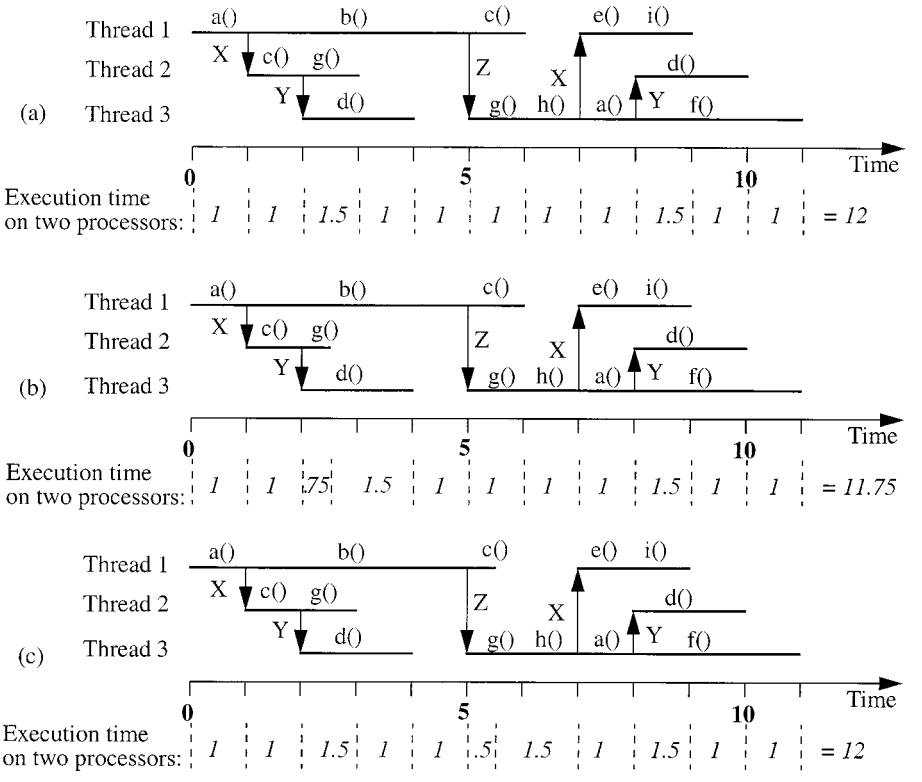

FIG. 6. A program with its three threads. Each time segment is indicated with dotted lines. The values in *italic* indicate the time required to execute that time segment.

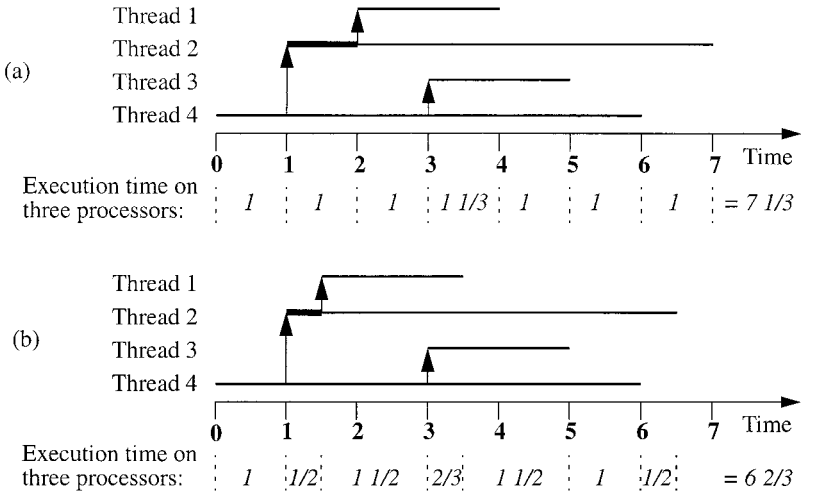

FIG. 7. An example showing that the weight for a segment may be more than one. (a) The nonoptimized execution. (b) The thicker line optimized with half a time unit. In this example the weight for the thicker line is 1.333.

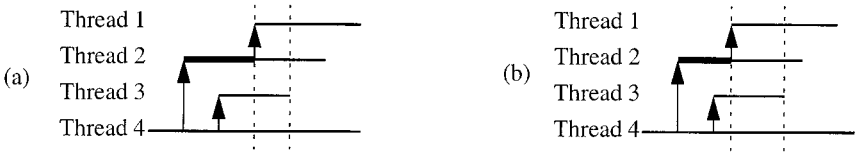

FIG. 8. An example of how optimization can extend the total execution time for an application.

this is less than in Fig. 6a. Thus, function `g( )` on thread 2 is part of the extended critical path and the weight is $0.5 = (12 - 11.75)/0.5$.

A segment, e.g., function `c( )`, on thread 1 is not part of the extended critical path as shown in Fig. 6c since the execution time is the same as in Fig. 6a.

Now, we have found the extended critical path and can calculate how much a function spent in the extended critical path as discussed in Section 2.2, but with the difference that the function times are multiplied with the weight for the segment.

The weight of a segment can be more than one, as shown in Fig. 7. When optimizing the thicker segment with, say, half a time unit and executing on three processors two things happen. First, the period of time when there are four processors is reduced; thus, total execution time is gained. Second, the last thread executing (thread 2) decreases by half a time unit. The total effect is 2/3 time units by optimizing half a time unit. The weight will then be $1.333 = 0.667/0.5$. Note that when analyzing the same example on two processors the same segment will have a weight of 1.0; i.e., the weight of the segment *increases* when going from two processors to three.

Another aspect of the extended critical path algorithm described is that the algorithm also identifies which segments have a *negative* impact on the total execution time when optimized. Consider the example in Fig. 8a. When this application is executed on three processors an optimization of the thicker segment will increase the total execution time. This is because optimizing the thicker line will shift thread 1 and the time segment when there are four threads competing for the processors is even longer as shown in Fig. 8b. Our algorithm will give the segment a negative weight.

3. IMPLEMENTATION OF THE EXTENDED CRITICAL PATH ALGORITHM

The extended critical path analysis has been implemented in a tool called VPPB [1, 2]. The extended critical path analysis increases the capacity of the tool in a natural way.

3.1. Short Description of the VPPB Tool and the Environment

VPPB is a tool for performance optimization of multithreaded programs written in C/C++ on the Solaris 2.X operating system. This is an environment common in both academia and industry. The VPPB tool combines two things: visualization of a multithreaded program's behavior, and prediction of how the program will execute on a multiprocessor.

The tool traces the execution of a multithreaded program on a single processor workstation. The tracing is performed by wrapping the thread library in Solaris 2.X and recording all the calls made by the program. The recorded information includes data about when, by which thread, with what parameters, etc., the call was issued. The recorded information is saved to file upon program exit. Thus, the behavior of the program is traced. Additional information about the preemptive scheduling of the LWPs (lightweight processes) [16] is also collected with a Solaris system command called `prex` described in [1]. The next step is to simulate the recorded information. The Simulator mimics a multiprocessor with any number of processors, the Solaris scheduling model, and different configurations of the threads [2].

The simulated execution is displayed graphically with two graphs. The first graph shows the amount of parallelism over time, as well as the number of runnable threads, i.e., the number of threads ready to execute but with no processor available. The second graph shows the execution as a Gant diagram based on the threads, with all synchronization (semaphores, mutexes, etc.) as symbols through the execution. It is possible to get more information about a single event (such as a signal on a semaphore at a given time) including the source code line where the call was made. With this information the developer can easily detect and identify performance bottlenecks.

3.2. Introducing Extended Critical Path Analysis

The introduction of extended critical path analysis requires that all the function entrances and exits must be traced. The insertion of function probes is done as a stage in the compilation phase of the program. Previously, there was no need for any recompilation or special compilation. The compilation is now done in three stages for each source code file. The first stage is to compile the source code into assembler code. The assembler code is then parsed in the next stage and for each function entrance–exit probes are inserted to record the events. We are then able to probe each invocation of each function. We use the same kind of recording probes, TNF (trace normal form), as previously used in the tool [1]. To simplify the implementation of the parser it assumes that the compiler uses the default optimization. It made the implementation easier because a function will then have one single exit point and the function calls will look the same even if the function is a leaf. A more sophisticated parser could, however, deal with those issues. Also, more extensive optimization may inline functions instead of making explicit calls; this cannot be dealt with (this is, however, a common limitation in profilers). The third step is then to compile the modified assembler code into object code. This stage is performed by the ordinary C/C++ compiler. The probes keep track of the function call depth for each thread. The developer may set a function call depth limit where the function calls are no longer recorded in order to avoid the recursive algorithm generating large amounts of recorded data. In the work by Hollingsworth in [3] the collected data were limited by only considering a few functions to be traced. The functions were selected by hand by the user. However, the technique by Hollingsworth will not automatically avoid recursive function calls generating large amounts of collected data.

The Simulator is extended with the algorithms discussed in Section 2. All the different synchronization primitives in the Solaris 2.X thread library are handled, including semaphores, mutexes, read–write locks, and condition variables as well as the creation and joining of threads.

The Simulator is used to obtain the execution of the program with one thread per processor. After the (simulated) execution is obtained, the extended critical path algorithm can be applied with any number of processors as the argument, as discussed in Section 2.3, and the result is displayed on a function level as discussed in Section 2.2 and by making the lines thicker in the Gant diagram for the segments in the extended critical path.

4. THREE PRACTICAL EXAMPLES

4.1. Parallel Quick Sort

In this example we have used a parallel version of the quick sort algorithm. We sorted 2,000,000 random integers. The algorithm used is shown in pseudo-code in Fig. 9. As can be seen, new threads will be created as long as the number of items to be sorted is greater than 100,000.

The structure of the program can be illustrated as a tree, where the function `RecursiveQuickSort` represents the leaves in the tree and the function `ParallelQuickSort` the other nodes. Since the tree is unbalanced there will always be one leaf that is the last to finish execution. This is illustrated in Fig. 10. Most of the work is performed in the leaves. With one processor per thread only the path to the latest leaf will be considered and in this path the most time will be spent in the function `ParallelQuickSort`. The path is shown in Fig. 10 with the dashed line.

```
int data[2000000];

ParallelQuickSort(int left, int right) {
/* Sort the integers to the left and right of the selected pivot.
Calculate start and end index for the next calculations to left and right. */
   if(right - left > 100000) {
      thread_create(ParallelQuickSort, leftStart, leftEnd);
      thread_create(ParallelQuickSort, rightStart, rightEnd);
   }
   else {
      RecursiveQuickSort(leftStart, leftEnd);
      RecursiveQuickSort(rightStart, rightEnd);
   }
}

RecursiveQuickSort(int left, int right) {
/* Sort the integers to the left and right of the selected pivot.
Calculate start and end index for the next calculations to left and right. */
      RecursiveQuickSort(leftStart, leftEnd);
      RecursiveQuickSort(rightStart, rightEnd);
}
```

FIG. 9. Pseudo-code for the ParallelQuickSort algorithm.

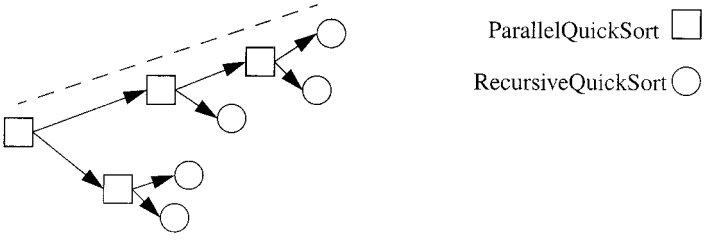

FIG. 10. An illustration of the tree. The length of the edges illustrates how much data are to be sorted by the following node.

When the extended critical path analysis is applied, it yields that with a few processors (five or less) the recursive function will dominate the extended critical path. However, with six processors or more our analysis shows that the parallel function dominates. Thus, depending on the number of processors in the target machine, the most suitable function for optimization will be different. This is shown in Fig. 11.

We optimized the functions with approximately 20%. A verification of the quick sort program was made on a Sun Enterprise 4000 with eight processors. The result is found in Fig. 12 and shows that the sort program behaves as predicted in Fig. 11; i.e., the crossover occurs between five and six processors.

The execution overhead for the instrumentation in this example is less than 2%. This overhead includes all instrumentation needed to do the simulations as well, not only the data needed for extended critical path analysis.

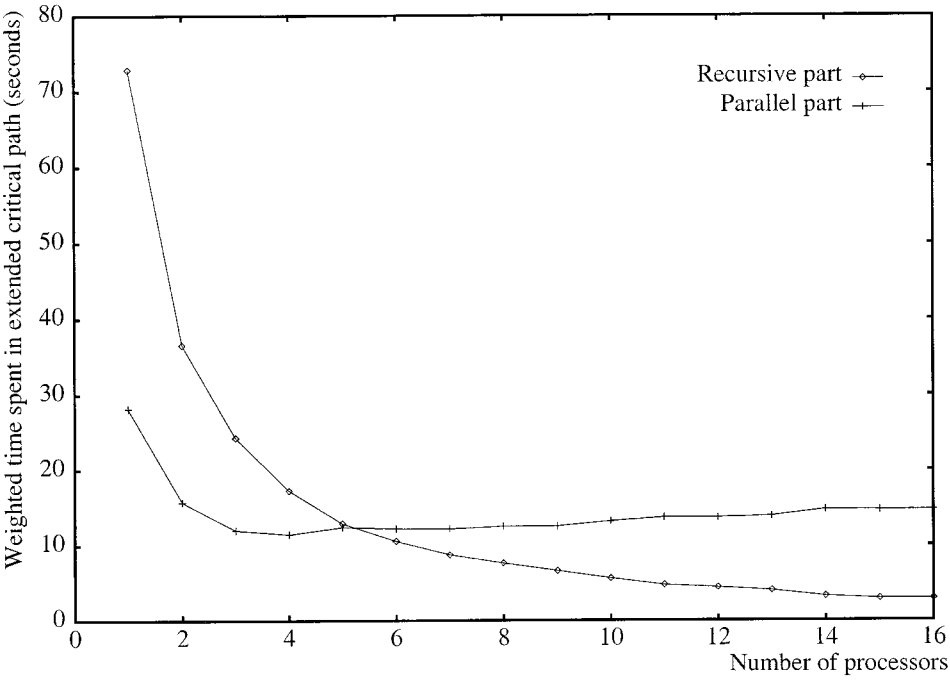

FIG. 11. The time for the recursive sort function and the parallel sort function spent in the extended critical path on different numbers of processors.

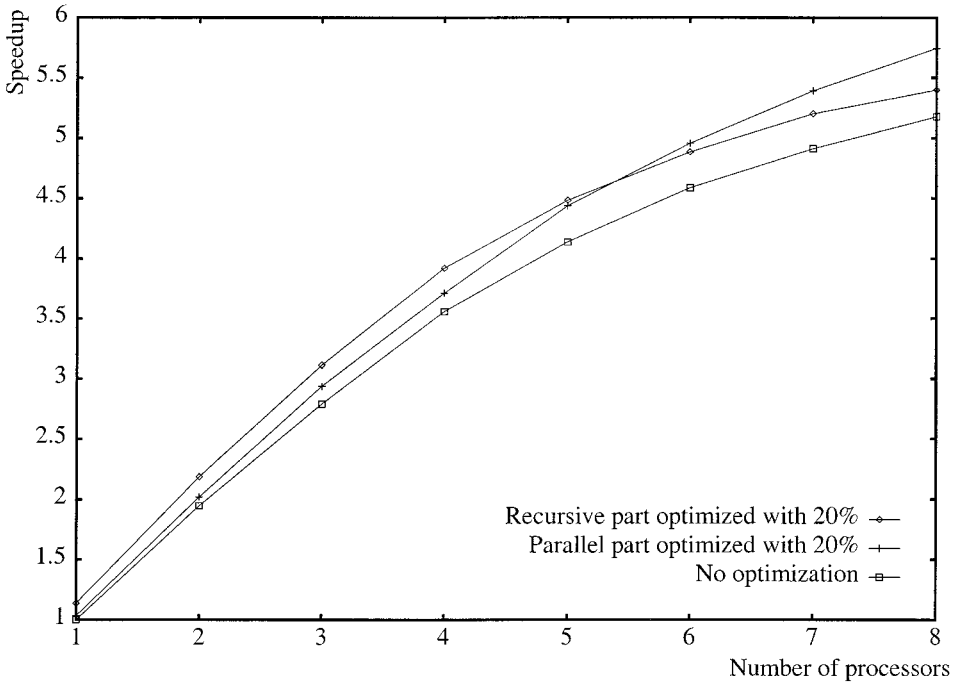

FIG. 12. Validation of the predictions on a multiprocessor with eight processors.

4.2. Finding the n Largest $h(x)$ for N Numbers x_i

The second application used in this study is a general search problem. The problem is to find the n largest $h(x)$, for N numbers x_i $(1 \leqslant i \leqslant N)$ where $n \ll N$. If the function $h(x)$ has local maximas and is computationally expensive then searching

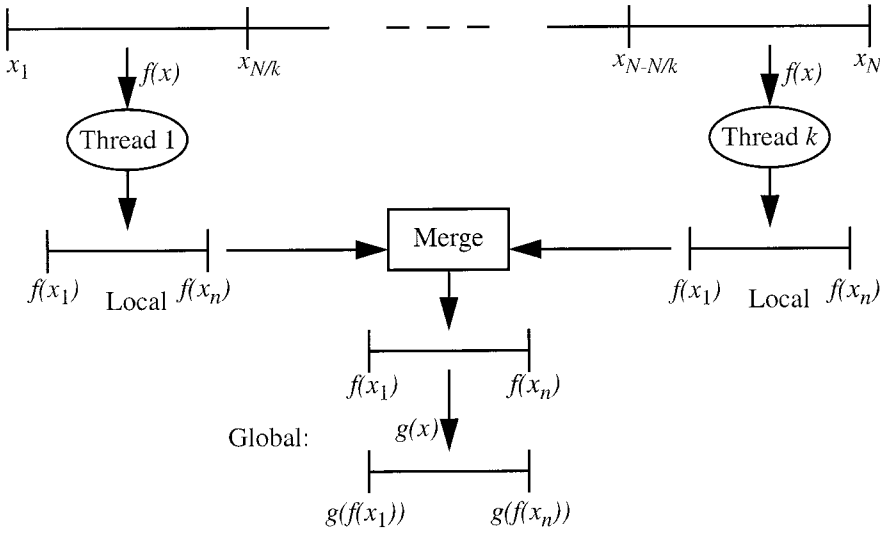

FIG. 13. Schematic picture of the search principle.

$$N = 10000000$$
$$n = 5000$$
$$k = 8$$
$$h(x) = \sum_{n=0}^{671} e^{\sqrt{\left|x + 2x^2\right| + n}}, \text{ which gives us}$$
$$f(x) = \left|x + 2x^2\right| \text{ and } g(x) = \sum_{n=0}^{671} e^{\sqrt{|x+n|}}$$

FIG. 14. The parameters and functions used for the study.

through each $h(x_i)$ would be computationally expensive. If it is possible to identify two functions $f(x)$ and $g(x)$ such that $h(x) = g(f(x))$ where $f(x)$ is computationally inexpensive and $g(x)$ is monotone, then the following solution would be applicable. Assume we have k threads. Apply $f(x)$ on each x_i with a subset of N/k elements for each thread and find the n largest $f(x_i)$ for each thread. Merge the n largest $f(x_i)$ for all threads into one vector containing the n largest $f(x_i)$ for the entire set $x_i (1 \leqslant i \leqslant N)$. Then $g(x)$ will only have to be applied to n numbers at the end. In Fig. 13 a principal sketch is found.

When applying this algorithm with the parameters found in Fig. 14 we get the execution times for $f(x)$ and $g(x)$ in the extended critical path as found in Fig. 15. As can be seen it is in this case most beneficial to optimize $f(x)$ for up to four processors, whereas $g(x)$ is most beneficial to optimize for more than four processors.

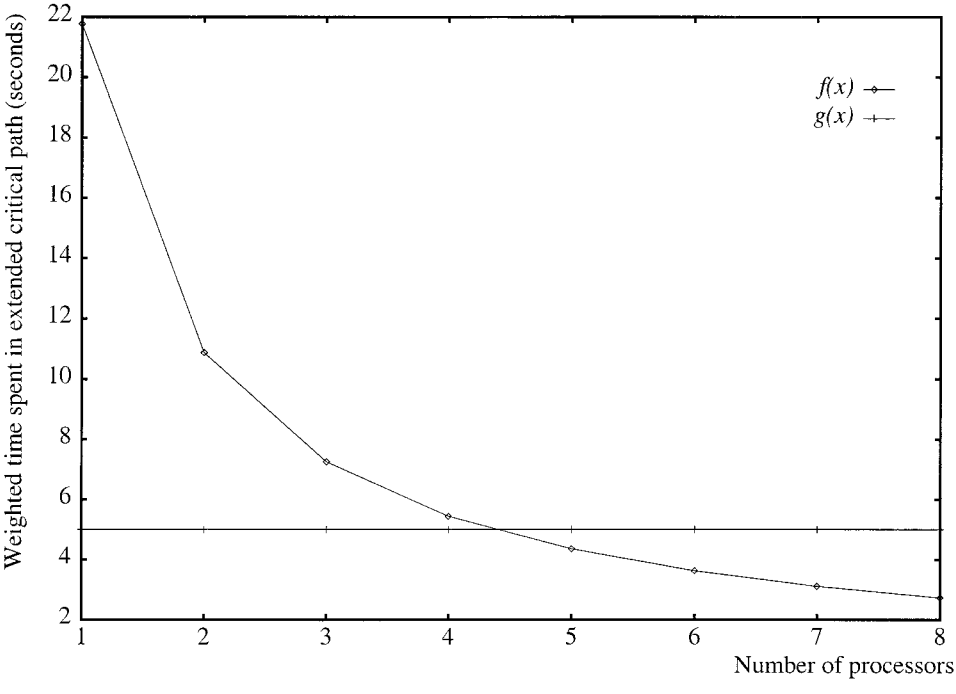

FIG. 15. The time $f(x)$ and $g(x)$ spent in the extended critical path on different numbers of processors.

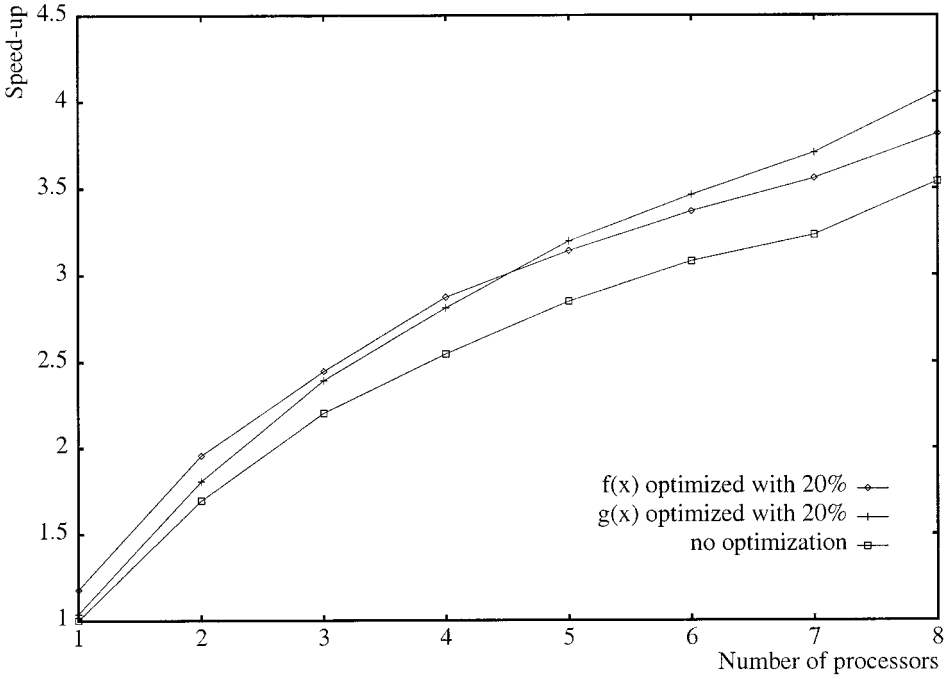

FIG. 16. Validation of the predictions on a multiprocessor with eight processors.

This is verified on an eight-way multiprocessor, as shown in Fig. 16 where $f(x)$ has better speedup than $g(x)$ for four processors or less. The situation is the opposite for five processors or more.

4.3. Billing Gateway

The third application used in this study is based on a commercial telecommunication application built by Ericsson, called BGw (billing gateway) [4]. We used a skeleton version of the BGw for our validation because the real BGw uses a lot of I/O, which is currently not supported by the extended critical path. The original BGw consists of about 100,000 lines of C++ code.

A principal sketch over the BGw (skeleton) is found in Fig. 17. The BGw (skeleton) works as a kind of filter. The Readers get the information to be filtered. As soon as all data are received the information is stored on disk, the disk is synchronized, i.e., all data are physically written to disk, and the receiver is ready for the next chunk of data. The Sorter reads the file created by the Reader, sorts and converts the data, and feeds two Writers. The Writers then read the data and send it further in the system. The skeleton had no I/O and consists of two Readers, two Sorters, and four Writers as shown in Fig. 17. Each Reader got three packets of information.

If the extended critical path analysis (shown in Fig. 18) is applied, it is shown that with a few processors (one or two) the Writer will dominate the extended

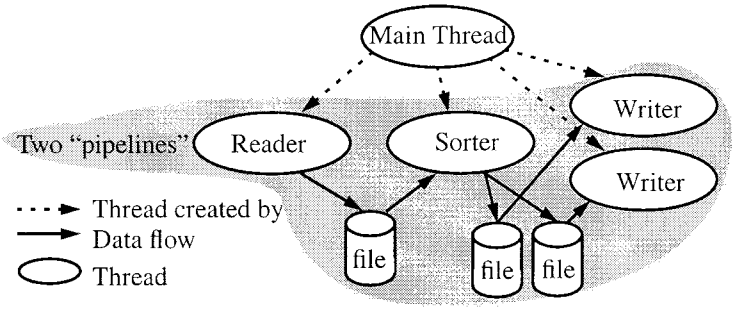

FIG. 17. The organization of the BGw skeleton. In the application used the data were *not* stored on file.

critical path. However, with more than two processors our analysis shows that the Sorter dominates. Ordinary profiling tools, such as Quantify, would indicate Writer as the most beneficial to optimize. Tools that do critical path analysis for an unlimited number of processors would identify Sorter as the most beneficial to optimize. The hump for the Sorter at three to four processors in Fig. 18 is due to the effect of segments with a weight greater than one as previously shown in Fig. 7.

Experimental results on an eight-processor Sun Enterprise 4000 show that Writer is the most beneficial to optimize on one or two processors and Sorter is more beneficial to optimize on more than two processors. This is shown in Fig. 19 where Writer, Sorter, and Reader have been optimized by approximately 20% each. The

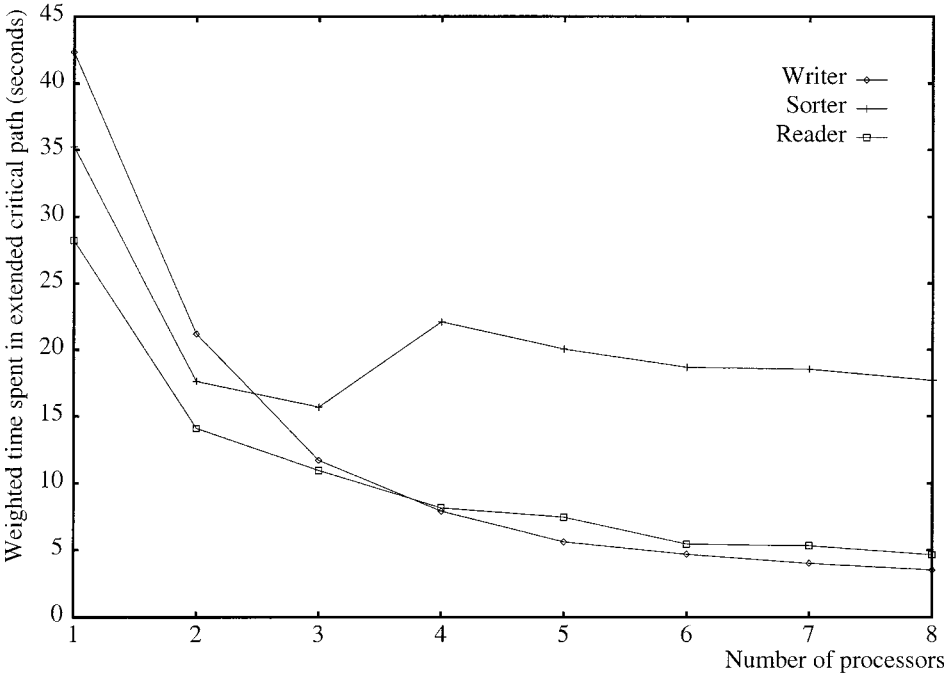

FIG. 18. The time the three function spent in the extended critical path on different number of processors.

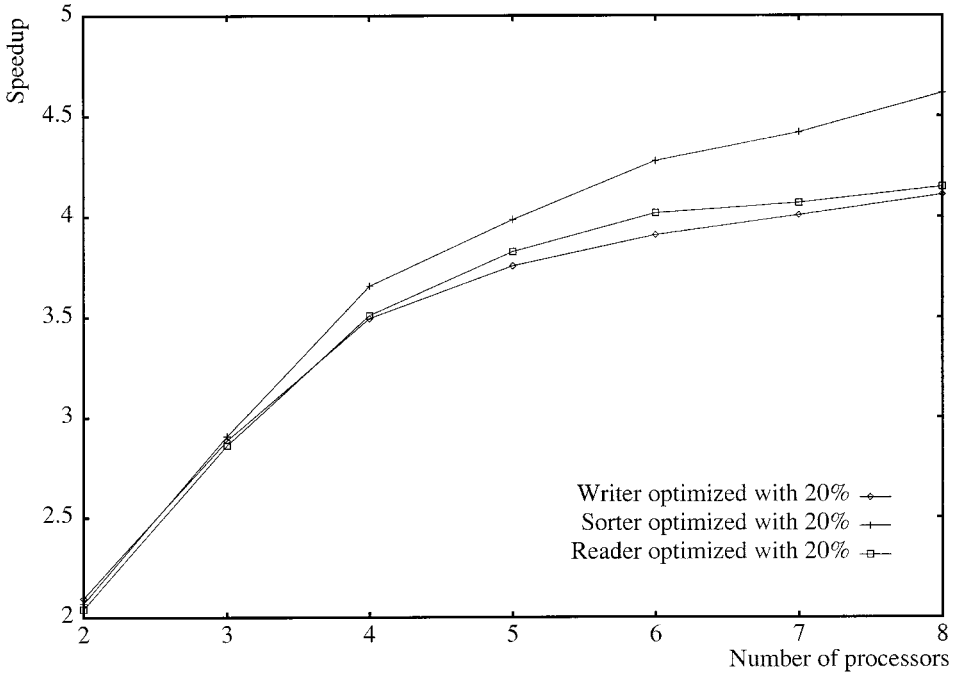

FIG. 19. Execution time for Writer, Sorter, and Reader when optimized with approximately 20% each, normalized to a nonoptimized execution time. The nonoptimized execution and the case with one processor are omitted for readability.

breakpoint when Sorter is more beneficial to optimize than Writer occurs between two and three processors as predicted. We can also see that the crossing point between Reader and Writer is between three and four processors as predicted in Fig. 18.

5. DISCUSSION

The examples in Section 4 show that the function dominating the extended critical path may change when a different number of processors is used for the same application. The extended critical path analysis described here assumes that the scheduling is ideal with infinitely small time slices and no scheduling overhead. Even if this assumption is quite close to the real world [7], it still differs from a real execution. Although not implemented, the tool (VPPB) could be extended with the capability to simulate the execution with one or several functions optimized to any degree. The result will be a simulation that will mimic the scheduling of the Solaris operating system with the optimized application. The complexity of the extended critical path algorithm is manageable, the three examples used for verification in Section 4 took at most 0.2 s to analyze on an ordinary workstation (300 MHz Sun Ultra10).

The extended critical path analysis is developed for CPU intensive applications and I/O has not been addressed. The tool (VPPB) is capable of recording and simulating I/O behavior. A simple approach for adding I/O capability for the extended critical path analysis is to approximate the time the thread waits for the I/O operation with a sleep for the same period of time. The result is then that the thread does not use the processor while waiting for an I/O operation and the extended critical path continues through the I/O operation and continues on the same thread. With this approach, no change in the extended critical path analysis is needed. However, I/O could also be used to (implicitly or explicitly) synchronize between threads. If one thread reads from a file that is empty the thread will be blocked until another thread writes something to the same file. The effect is then a synchronization. By recording the corresponding file descriptor for each I/O operation these dependencies could be solved and I/O will behave more or less as any other lock. The key difference between ordinary locks and the I/O synchronization is, however, that it is hard for the Simulator to know if the file is empty or not at the beginning of execution of the program.

The extended critical path analysis has been addressed for SMPs in this paper. The technique can be modified for message passing applications on distributed memory machines as well. The algorithm must then be modified to manage that a message will take some time to reach its destination. The recording will be local on each node and after execution is done the recorded material will be merged together for all nodes in order to apply the extended critical path algorithm. Message passing applications often use one process–thread per processor and thus the work by Hollingsworth [3] will be appropriate. However, there is an ongoing trend in the message passing community to include thread primitives in the message passing libraries. Then, the situation may occur that there are more threads on one single node than the number of processors on that node, and the technique by Hollingsworth will not be appropriate any longer. Our technique will manage that. For NUMA machines (nonuniform memory architecture) the extended critical path algorithm does not need any modification at all. However, the simulated execution that the algorithm makes use of must be made according to the memory delays for the specific machine. When considering distributed applications, the problem is the same as for a message passing application, but it may be harder to keep a synchronized clock for the distributed machines. A synchronized clock is needed when merging the local log files before applying the extended critical path algorithm.

The extended critical path algorithm can be applied to languages other than C/C++ with Solaris threads. For instance, Java could be used as well. Java includes primitives for critical regions and monitors. Critical regions and monitors can be implemented with the Solaris thread primitives mutex and condition variables. Thus, it is feasible to use the extended critical path analysis for Java as well. It is worth noting that the specification of the Java virtual machine, described in [6] by Lindholm and Yellin, does not say if the Java threads can actually make use of more than one processor. Threads that cannot make use of more than one processor are called green threads, whereas threads that can use more than one processor are called red threads (or native threads). It is the Java virtual machine that decides if a Java thread will be green or red; this cannot be decided in the programming language.

6. RELATED WORK

The critical path analysis is a well-known technique [3, 8, 18]. However, all these approaches originate from the message passing area where the common case is one thread (process) per processor. The works in [3, 8, 18] are based on a PAG (program activity graph) where the critical path is defined as "the longest, time-weighted sequence of events from start of the program to its termination." This means that analysis is done with no limitation on the number of processors available and shows only the points at 16 processors in Fig. 11. However, it is not unusual to have several threads competing for a lesser number of processors in a multithreaded application. In [18] this issue is addressed briefly by using a CPU-based timer when measuring the length of the edges in the PAG. Even if a process is scheduled off the process, the time will be counted to the corresponding edge in the PAG. Thus, when doing the critical path analysis on the PAG the scheduling will be included. The drawback to this solution is that the critical path will not include the process that executed instead. An example with two processes, where process 1 calculates for 12 time units and process 2 acts as a watchdog executing five times for 1 time unit every second time unit, demonstrates this. The example is illustrated in Fig. 20 when executing on a single processor. Given the solution by Yang and Miller in [18] process 1 will have the total length of 17 and thus will be the longest path and be called the critical path. However, optimizing process 2 will also yield a reduced completion time in this example. The critical path by Yang and Miller in [18] will not identify that. Our extended critical path algorithm handles situations like the one in Fig. 20.

In [8], interest is also focused on the second longest path, etc., in the application. This is because if the critical path is optimized to the extent that it will no longer be the longest path, then the second longest path will be the critical path. This technique will not be appropriate for a multithreaded application with more threads executable than processors since the optimization of one function may affect how the application will be scheduled by the operating system.

Other optimization tools for multiprocessors only identify the critical path on one processor. The Solaris program `tha` [15] is designed to work as `prof` [14] with the difference that the information collected is on a per thread basis. `Tha` collects `prof` information per thread and yields correct information per thread. However, there is no information about the execution flow and dependencies between the threads. Simply adding all threads' accumulated execution times for a given function would yield the same information as in Quantify [13]. While `tha` supports better information, it also has some major drawbacks described in [15]; e.g., it is not possible to use the standard C++ I/O primitives.

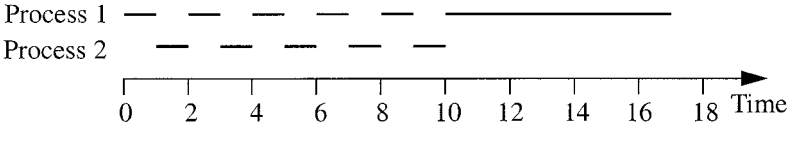

FIG. 20. Two processes executing on one processor.

Another way of presenting the performance problems of multithreaded programs is based on contention, as in Tmon [5]. The contention is based on locks and context switching overhead. Tmon use two single processor workstations, one to execute the multithreaded program and the other to gather the recorded data. The data are analyzed, to some extent, in real time. This setup requires a fast interconnection between the single processors. The applicability of Tmon on multiprocessors is not addressed in [5].

There are other tools that support simulation of a parallel program behavior and then visualize the result [9–12, 17]. However, they do not support (extended) critical path analysis.

7. FUTURE WORK

The extended critical path is shown in the VPPB tool as thicker lines in the Gant diagram. The information about different functions is simply a list of metrics per function. The use of a function call graph found in many conventional profiling tools, e.g., Quantify [13], is not directly applicable, since the extended critical path may move from one thread to another due to synchronizations. The synchronizations may be placed deep down in the functions and thus the extended critical path jumps from a function called deep down in one thread to another function deep down in another thread. The simple call graph will not cope with this kind of jump. An issue for future work is, therefore, to improve the visualization and representation of the extended critical path for the developer.

The simple approach for including I/O in the extended critical path analysis discussed in Section 5 could also be addressed as future work. However, full support for I/O will require further investigations and could be, as such, a future development of the extended critical path analysis.

Another thing that would be appropriate for future work is to identify by how much a function can be optimized before another function will become the function that the most time is spent in. On a single processor it is possible to optimize a function until the total execution time for that function is less than some other function. It is not that simple on a multiprocessor executing more threads than processors, since optimizing one function may alter the scheduling order and another segment may become a part of the extended critical path. Also along those lines, support for elaborating with different optimizations of functions directly in the tool would be interesting. Then it would be possible to allow the developer to specify how much a certain function can be optimized (based on an estimation) and rerun the *simulation* with the "pseudo-optimized" function. After that, the extended critical path analysis can be applied on the rerun simulation.

8. CONCLUSION

Traditional performance optimization is done when the program is written. The main goal is to increase the performance of the application. The existence of a number of commercial profiling tools [13–15] shows the importance of the

optimization task. Multiprocessors are used for the same reason, i.e., to increase the performance. Performance optimization for programs designed for multiprocessors is at least as important as optimization of sequential programs, because high performance requirements are often a major reason for using multiprocessors in the first place.

In the case of a multithreaded program executing on a multiprocessor it is not certain that all executed code segments will add to the completion time. A simple example is a watchdog, which in the single processor case will be a part of the completion time. However, on a multiprocessor, the watchdog may execute on its own processor. The watchdog will then not affect the completion time of the program.

Traditional profilers, such as Quantify [13] and `tha` [15], give misleading information about where in the code to concentrate the optimization efforts. In some cases Quantify will actually give the worst possible indications. The reason these kind of tools give misleading information is that they assume that all executed code segments contribute to the completion time.

To implement efficient performance optimization we must concentrate the efforts on the critical path. The critical path as it is addressed in [3, 8, 18] will not fit our need when more threads are able to execute than there are processors available. Our extended critical path analysis is dependent on the synchronization behavior in the multithreaded program and the number of processors. In this paper an algorithm to find the extended critical path has been presented. The algorithm manages not only an ideal situation when there are as many threads as processors, but also when there are fewer processors than threads.

The possibility for the developer to connect the extended critical path to those functions that are part of the extended critical path is important. We have presented an algorithm that does so on a multiprocessor. The algorithm also shows the amount of time spent in the functions during the extended critical path.

These methods have been implemented in a performance optimization tool called VPPB [1, 2]. The tool pinpoints what parts of the code to optimize without the need of a multiprocessor. The current platform for the tool is the Solaris 2.X operating system; thus, the tool is applicable to a large number of industrial applications.

The usefulness of the method has been demonstrated by using the tool on three parallel programs. The correctness of the predictions was verified on an eight-processor SMP.

ACKNOWLEDGMENT

This work was supported in part by ARTES and the Swedish Foundation for Strategic Research.

REFERENCES

1. M. Broberg, L. Lundberg, and H. Grahn, Visualization and performance prediction of multithreaded Solaris programs by tracing kernel threads, *in* "Proc. 13th International Parallel Processing Symposium," pp. 407–413, IEEE Computer Society, Los Alamitos, CA, 1999.

2. M. Broberg, L. Lundberg, and H. Grahn, VPPB—A visualization and performance prediction tool for multithreaded Solaris programs, *in* "Proc. 12th International Parallel Processing Symposium," pp. 770–776, IEEE Computer Society, Los Alamitos, CA, 1998.
3. J. Hollingsworth, Critical path profiling of message passing and shared-memory programs, *IEEE Trans. Parallel Distrib. Systems* **9**, 10 (October 1998), 1029–1040.
4. D. Häggander and L. Lundberg, Optimizing dynamic memory management in a multithreaded application executing on a multiprocessor, *in* "Proc 1998 Int'l Conf. on Parallel Processing," pp. 262–269, IEEE Computer Society, Los Alamitos, CA, 1998.
5. M. Ji, E. Felten, and K. Li, Performance measurements for multithreaded programs, *Perform. Eval. Rev.* **26**, 1 (June 1998), 161–170.
6. T. Lindholm and F. Yellin, "The Java[tm] Virtual Machine Specification," second ed., Addison–Wesley, Reading, MA, 1999.
7. L. Lundberg and M. Roos, Predicting the speedup of multithreaded Solaris programs, *in* "Proc. 4th International Conference on High-Performance Computing," pp. 386–1392, IEEE Computer Society, Los Alamitos, CA, 1997.
8. B. Miller, M. Clark, J. Hollingsworth, S. Kiersead, S.-S. Lim, and T. Torzewski, IPS-2: The second Generation of a Parallel Program Measurement System, *IEEE Trans. Parallel Distrib. Systems* **1**, 2 (April 1990), 206–217.
9. E. Papaefstathiou, D. Kerbyson, G. Nudd, and T. Atherton, An overview of the CHIP^{3}S performance prediction toolset for parallel systems, *in* "Proc. 8th ISCA International Conference on Parallel and Distributed Computing Systems," pp. 527–533, 1995.
10. V. Pillet, J. Laboarta, T. Cortes, and S. Girona, "PARAVER: A Tool to visualize and Analyse Parallel Code," CEPBA/UPC Report RR-95/03, University of Politencia, Catalonia, 1995.
11. S. Sarukkai and D. Gannon, SIEVE: A performance debugging environment for parallel programs, *J. Parallel Distrib. Comput.* **18**, 2 (June 1993), 147–168.
12. Z. Segall and L. Rudolph, PIE: A programming and instrumentation environment for parallel processing, *IEEE Software* **2**, 6 (November 1985), 22–37.
13. Rational, Quantify version 4.2, http://www.rational.com/products/quantify.
14. Sun Man Pages, prof, Sun Microsystems Inc., 1993.
15. Sun Man Pages, tha, Sun Microsystems Inc., 1996.
16. SunSoft, "Solaris Multithreaded Programming Guide," Prentice–Hall, Englewood Cliffs, NJ, 1995.
17. S. Toledo, PERFSIM: A tool for automatic performance analysis of data-parallel Fortran programs, *in* "Proc. 5th Symposium on the Frontiers of Massively Parallel Computation," pp. 396–405, IEEE Computer Society, Los Alamitos, CA, 1994.
18. C.-Q. Yang and B. Miller, Critical path analysis for the execution of parallel and distributed programs, *in* "Proc. 8th International Conference on Distributed Computing Systems," pp. 366–375, IEEE Computer Society Press, Washington, DC, 1988.