Research Report 9/99

Performance Optimization using Critical Path Analysis in Multithreaded Programs on Multiprocessors

by

Magnus Broberg, Lars Lundberg, Håkan Grahn

Department of
Software Engineering and Computer Science
University of Karlskrona/Ronneby
S-372 25 Ronneby
Sweden

ISSN 1103-1581
ISRN HK/R-RES—99/9—SE

Performance Optimization using Critical Path Analysis in Multithreaded Programs on Multiprocessors

by Magnus Broberg, Lars Lundberg, Håkan Grahn

ISSN 1103-1581
ISRN HK/R-RES—99/9—SE

Printed by Psilander Grafiska, Karlskrona 1999

Performance Optimization using Critical Path Analysis in Multithreaded Programs on Multiprocessors

Magnus Broberg, Lars Lundberg, and Håkan Grahn

Department of Software Engineering and Computer Science
University of Karlskrona/Ronneby
Soft Center, S-372 25 Ronneby, Sweden
{Magnus.Broberg, Lars.Lundberg, Hakan.Grahn}@ipd.hk-r.se

Abstract

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 total execution time of the program. Thus, any code segment that is optimized in a sequential program will decrease the execution time. In the case of a parallel program executed on a multiprocessor this is not always true. This is due to dependencies between the different threads. As a result, certain code segments of the execution may not affect the total execution time of the program. Thus, optimization of such code segments will not increase the performance. In this paper we present a new approach to perform the optimization phase. Our approach finds the critical path of the multithreaded program and the optimization is only done on those specific code segments of the program. We have implemented the critical path analysis in a performance optimization tool.

1. Introduction

Optimization of sequential programs is a well known technique to increase performance. Traditionally this optimization is performed when the program is already written. Today, several tools, often referred to as profilers, support the developers in finding the code segments of the program where the most time is spent. These code segments are then the target for the optimization efforts., i.e., the programmer rewrites these code segments in order to minimize the execution time. This procedure has been used successfully for a long time.

The main reason for using multiprocessors is to achieve higher performance. As a part of reaching this higher performance, traditional optimization is performed on programs designed for multiprocessors. However, not all code segments of the program will affect the total execution time.

It is often difficult to know which code segments of a multithreaded program that affect the total execution time. Synchronizations are a major factor to decide whether a code segment of a program contribute to the total execution time. Consider a thread that will produce the result long before the result is needed. The thread does not share the processor it is executing on with any other threads. Optimizing that thread will *not* affect the thread that use the result and, thus, the total execution time. On the other hand, if the threads using the result where waiting for the result, optimizing the thread producing the result will decrease the total execution time.

Also the actual number of processors affects which code segments that constrain the total execution time. For example, consider a program that includes a watchdog. The number of processors are sufficiently large to allow the watchdog to execute on its own processor. Thus, there is no need to optimize the watchdog. However, when having only one processor, the execution time for the watchdog will affect the total execution time for the whole program.

Multithreaded programs may include hundreds of threads with thousands of different synchronizations. Thus the need for a tool that indicates which code segments of the program that is fruitful to optimize is crucial for effective optimization of multithreaded programs executing on a multiprocessor. It is also crucial that the developer gets useful information about the code segments, i.e., function names, what fraction of a function contributes to the total execution time. In this paper we present a tool that will give the developer information about which code segments of the multithreaded program that are beneficial to optimize, we call that the *critical path*.

The paper is structured as follows: A definition and discussion of the critical path is presented in Section 2. Section 3 describes the algorithm to calculate the critical path. In Section 4 we present a working tool with the critical path analysis implemented and show in a practical example in Section 5 for the need of such a tool. Discussion and related work is found in Section 6 and we conclude the paper in Section 7.

2. Overview of the Critical Path Analysis

The developer of a multithreaded program must have knowledge of which code segments of the parallel program that are fruitful to optimize. In general terms, code segments that are fruitful to optimize are called the *critical path*. We define the *critical path* as all the executed code segments of a program that when reduced with an ε will also shorten the total execution time. Consequently, all code segments of a *sequential* program will be considered as part of the critical path. This is because if we shorten *any* part of a sequential program it will result in a shorter total execution time.

In the case of a multithreaded program executing on a multiprocessor, not all code segments of the program are included in the critical path. Consider the program in Figure 1. We assume that all functions (a-d) take equal amount of time to execute and that signalling takes no (or negligible) time.

When executing the program on a multiprocessor with 3 processors (or more) the execution will look like in Figure 2. The critical path is (from the start) thread 1 executing `a()`, then thread 3 executing `c()`, thread 2 executing `d()`, and finally thread 1 executing `a()` again. It is obvious that shortening some code segment in the critical path will also decrease the total execution time. Shorten, e.g., `b()` will have no affect on the total execution time.

```
Thread 1:                Thread 2:                Thread 3:
Execute a()              Execute b()              Wait for event X
Signal for 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
```

Figure 1: A simple program with three threads.

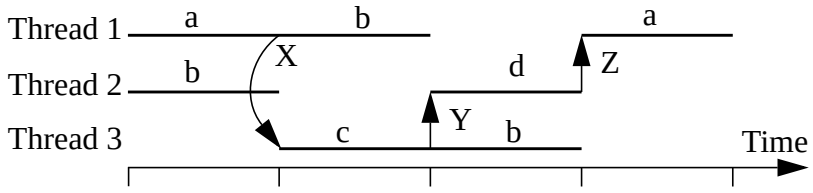


Figure 2: The execution of the simple program on a multiprocessor with three processors.

Unfortunately, ordinary profiling tools, such as Quantify [7], does not consider the critical path. They will indicate that the program spend most of the time in function `b()` and suggest that function for optimization. This is indicated in Figure 3 where function `a()` to `d()` is found at the top, the other functions shown are only for internal use in the Solaris operating system. In fact, the advice Quantify gives is the worst possible, due to the tool's incapability to analyze the semantics of the synchronizations within the program.

The need for a tool which consider the critical path is obvious in order to correctly optimize multithreaded programs on a multiprocessor.

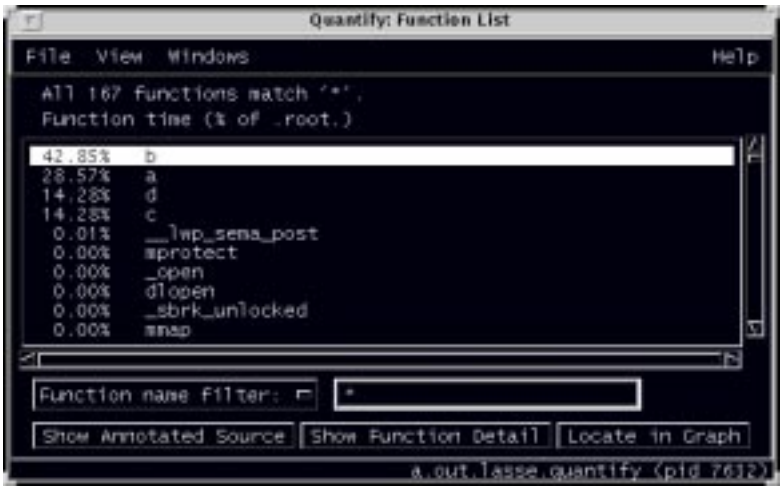


Figure 3: Quantify's list of the functions to optimize.

3. The Critical Path Algorithm

In this section we will describe how the critical path algorithm works. First we describe the algorithm to identify the critical path. Then we describe how we calculate how much time each function in the critical path executed during the whole execution. Both these parts are important in order to give useful information about where to start optimize the code. Finally, we will show how to calculate the critical path for multithreaded program with more runnable threads at the same time than there are processors.

3.1. Finding the critical path in the optimal case

First we will look at the critical path algorithm under the condition that the program is executed on a sufficient number of processor and thus no threads have to share a processor at any time. We assume that explicitly synchronized events happens at the exact same time, such as one thread releases another thread. This means that the thread starts exactly at the same moment as the other thread releases it. Unsynchronized events are assumed to occur at different times. Using a timing device with high resolution will solve the assumptions above. These assumptions are realistic and does not change the algorithm in principal. They are easily removed, but kept here in order to simplify the presentation. The algorithm for finding the critical works as follows.

The algorithm starts at the end of the execution and follows the thread backwards until the thread has been blocked for some reason. Then the algorithm finds the event that released the current thread. The algorithm then continues to follow the releasing thread backwards until it also becomes 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 critical path.

To illustrate the algorithm we use the example program with three threads in Figure 4. The execution times for the different functions are illustrated in Figure 5 where also the execution of the program on an optimal number of processors are illustrated.

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

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

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

Figure 4: A program with its three threads.

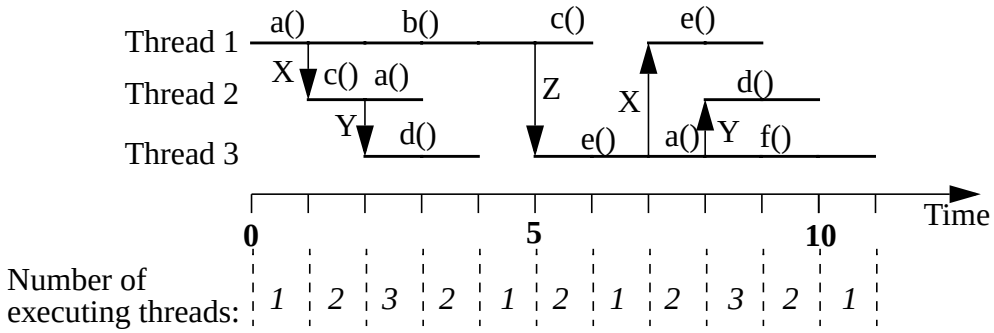


Figure 5: The optimal execution of the program. The values in *italic* indicates the number of executing threads.

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 function `f()`, `a()` and finally `e()`. At time 4 to 5 the thread was not executing, thus the thread 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 function `b()` and `a()` without being blocked until the start of the program at time 0.

Thus the critical path for the program is thread 1 executing function `a()` and `b()` and thread 3 executing function `e()`, `a()`, and `f()`. It is easily verified that optimizing any *other* part of the program will not affect the total execution time.

3.2. Calculating function times contributing to the critical path

In the previous section all the synchronization are done between the function calls. This is not always the case, since synchronization could appear inside the functions as well. This gives some complications when calculating the time spent in certain functions during the critical path. This calculation is done when the critical path analysis is already done, thus we only concentrate our effort on the parts that are in the critical path. We will calculate three values *per function*:

- The number of calls. This is the number of times the function was called during the critical path.
- The total execution time. This time is the accumulated time, during the critical path, the program was executing in the function.

• The total execution time including all descendant functions. This time is the accumulated execution time for the function and its descendants, during the critical path both for the function and the descendants.

We will use the program illustrated in Figure 6 to show how the values are calculated. In Table 1 the information needed to perform the calculations of the program in Figure 6 is found. The information in Table 1 is gathered by the tool described in Section 4. The events indicating a thread entering or exiting a function are considered to take no time and thus are unimportant to mark as part of the critical path. Basically the algorithm iterates over all rows in the table that is marked as part of the critical path.

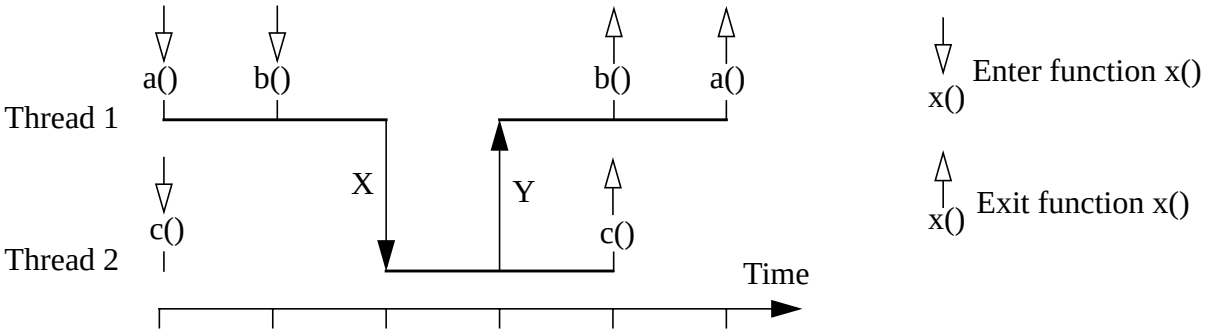


Figure 6: An example program with synchronization inside functions.

Table 1: The needed information for the analyze of the program shown in Figure 6. The Category column will be discussed in Section 4.

Row	Event	Thread	Start time	Length	Part of Critical Path	Category
1	enter function a()	1	0.000000	-	-	C
2	executing	1	0.000000	3.894285	yes	B
3	enter function c()	2	0.000000	-	-	C
4	wait for event X	2	0.000000	0.001680	no	A
5	enter function b()	1	3.894285	-	-	C
6	executing	1	3.894285	3.894284	yes	B
7	signal event X	1	7.789474	0.000177	yes	A
8	executing	2	7.789651	3.928914	no	B
9	wait for event Y	1	7.789658	0.000156	no	A
10	signal event Y	2	11.718565	0.000113	yes	A
11	executing	2	11.718678	3.832353	no	B
12	executing	1	11.718678	3.884579	yes	B
13	exit function c()	2	15.551031	-	-	C
14	exit thread	2	15.551031	0.000058	no	A
15	exit function b()	1	15.603257	-	-	C
16	executing	1	15.603257	3.884579	yes	B
17	exit function a()	1	19.487836	-	-	C
18	exit thread	1	19.487836	0.000054	yes	A

Each iteration first determines the thread's function call stack. The first part of the critical path in Table 1 (row 2) has the stack `a()`. Now the algorithm first look whether the function `a()` has been called. This is determined by comparing the time of the entering (row 1) and the start time of the execution (row 2). Since they are equal the thread just entered the function and a counter for

the number of entrances for function `a()` is increased. The time the critical path has been executing in the function is kept in another counter, which simply is increased with the length of the event (row 2). The total time the function has been executing, including its descendants, is kept track of in a similar manner.

In order to illustrate the handling of the two different time counters, we will take a look at the next part of the critical path (row 6). Now the stack is `b()` at the top and `a()` at the bottom. The counter for execution time is only increased for the function at the top of the stack, i.e., function `b()`. The counters for the execution time with descendents is increased for all functions in the stack, i.e., `b()` and `a()`.

This is then repeated for all parts of the critical path. Recursive calls are also easily managed since we keep only one set of counters per function, regardless of whether there are many occurrences of the function on the stack. Note, however, that the counters may only be updated once per part of the critical path, regardless if they occur several times on the stack.

3.3. Finding the critical path with CPU constraints

In Section 3.1 we defined the algorithm for finding the critical path when a unlimited number of processors are available. This is not always the case in real life, unfortunately. Thus, we need to adjust the algorithm to fit whenever there are more runnable threads than there are processors. This enforces that some threads at some times must be multiplexed by the scheduler on one processor.

We keep the example in Figure 5, but look at what happens if we only have two processors available. The number of executing threads are 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 without assuming any specific scheduling algorithm is to regard the processors to run slower 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 is not an unrealistic assumption since some performance prediction tools has used that assumption with good result [6] when compared with real scheduling.

First we identify the optimal critical path as described in Section 3.1. During time 2 to 3 in Figure 5 the number of thread exceeds the number of processors, thus the three thread must be multiplexed on the two processors. In practise time 2 to 3 will take 1.5 time units and thus extended the total execution time with 0.5 time units. If we are able to optimize any thread during time 2 to 3 we are interested in knowing which ones that will cut the total execution time. Thread 1 is already marked as part of the critical path. Optimizing thread 3 will make no difference since ε is to be considered a small amount of time. This is because optimizing thread 3 at time 2 to 3 will only shift the execution at time 3 to 4 to the left, leaving time 2 to 3 still having 3 threads executing simultaneously. Thread 2 can successfully be optimized, since each ε will also cut the time spent executing threads simultaneously. This is because thread 2 becomes blocked at time 3, and thus no part of the execution will be shifted in its place.

The critical path algorithm looks for all threads that stop executing (either becomes blocked or exits) at a time where there are more runnable threads than processors available. This, since a thread that stops executing is the only one that not will shift in a later part in the time period where there are more threads than processors as discussed above. After the algorithm has found one thread ending, the critical path algorithm described earlier in Section 3.1 will be applied on the thread from the time it was stopped. The algorithm in Section 3.1 will continue backwards since any part of the critical path can make the thread end earlier and thus shorten the time that more threads are runnable than processors. The algorithm in Section 3.1 continues until there are as

many threads as processors. A schematic view of this is found in Figure 7(a) assuming two processors. The two processors will have three threads to execute during time 2 to 3, thus optimizing thread 3 will shorten the period when the two processors have to execute three threads. The critical path algorithm can not continue to the time slot between time 1 and 2 since this will not shorten the time the two processors will execute three threads. The result is that thread 2 executing function `a()` is part of the critical path in the example in Figure 5.

The time 8 to 9 in Figure 5 will mark the second half of function `e()` executed by thread 1 as part of the critical path. However, when applying the critical path algorithm in Section 3.1 we cannot stop when the algorithm reaches a time with as many threads as processors. Obviously, optimizing any part of function `e()` will make thread 1 end its execution earlier. This will make thread 1 in executing function `e()` part of the critical path. The reason is schematically shown in Figure 7(b), again with two processors. It is obvious that shortening any part of thread 3 will shorten the time that the two processors must execute the three threads. Thus, the critical path algorithm may continue backward even if there are as many processors as threads. The difference between Figure 7(a) and (b) is that thread 3 synchronize, i.e., releases, a thread at time 2 in Figure 7(a). Thus the stop criteria must consider both these cases. The stop criteria is then defined as to stop whenever the thread is synchronizing with another thread and the number of processors is equal or greater than the number of runnable threads.

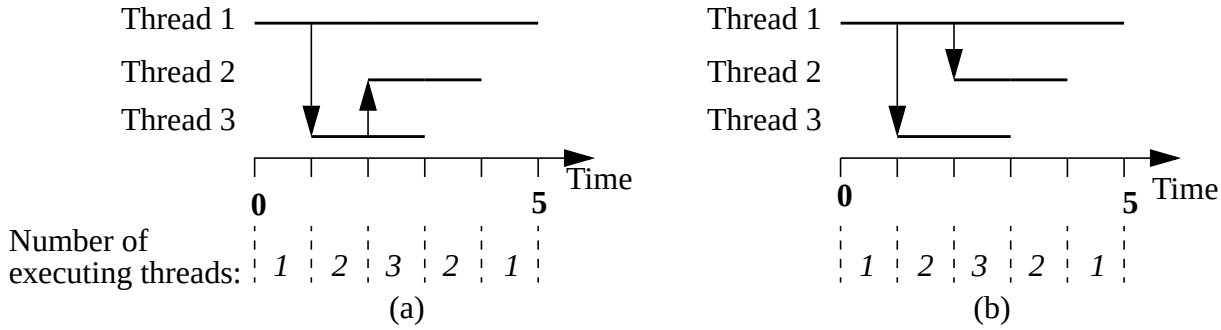


Figure 7: Two principal sketches to illustrate the critical algorithm stop criteria. The number of runnable threads during each time slot is shown in *italic*.

4. Implementation of the Critical Path Algorithm

The critical path analysis has been implemented in a tool called VPPB (Visualization and Prediction of Parallel Program Behaviour) [1, 2]. The tool already had much of the infrastructure to perform the critical path analysis. The critical path analysis fits well in this tool as it increases the capacity of the tool in a natural way.

4.1. 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 academea as well as in industry. The VPPB tool combines two things: visualization of a multithreaded program's behaviour, and prediction of how the program will execute on a multiprocessor.

The tool traces the execution of a multithreaded program on a uni-processor workstation. The tracing is performed by wrapping the thread and synchronization library available in Solaris 2.X and record the 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 stored on file upon program exit. Thus, the behaviour of the program is traced. Additional information about the pre-emptive scheduling of the LWPs (Light Weight Processes) [10] is also collected with a Solaris system command called `prex` [2]. The recorded information results in the rows categorized as A in Table 1. Based on that information the rows categorized as B can be computed. 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 [1].

The simulated execution is then displayed graphically with two graphs. The first graph shows the amount of parallelism over time, as well as the amount of waiting 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 signal on a semaphore at a given time) including the source code line that made the call. With this the developer can easily detect and identify performance bottlenecks.

4.2. Introducing critical path analysis

The introduction of critical path analysis requires that all the function entrances as well as exits must be traced as well. This corresponds to the rows categorized as C in Table 1. The insertion of function probes is done as a stage in the compilation phase of the program. Previously, there was no need for any re-compilation 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, this is done by the ordinary C/C++ compiler with the '-S' option. The assembler code is then parsed in the next stage and for each function entrance/exit probes are inserted to record the events. We use the same kind of recording probes, TNF [2], as previously used in the tool. The parser assumes that the compiler uses no optimization flags, since, e.g., optimization flags may in line functions instead of making explicit calls. The third step is then to compile the modified assembler code into ordinary object code. This stage is performed by the ordinary C/C++ compiler. The probes keeps count of the function call depth for each thread. The developer may set a function call depth limit where the function calls are not longer recorded. This, in order to avoid recursive algorithm to generate large amounts of recorded data. Upon execution all entrances and exits are recorded, corresponding to the rows categorized as C in Table 1.

The Simulator is extended with the algorithms discussed in Section 3. The different synchronizations in Solaris 2.X thread library includes semaphores, mutexes, read/write locks, and condition variables. The synchronization directly between threads are also supported, the obvious one is that one thread joins another thread, i.e., one thread wait for another thread to exit. Thread creation is also handled as a synchronization. A thread that start is dependent on the thread that issued the creation.

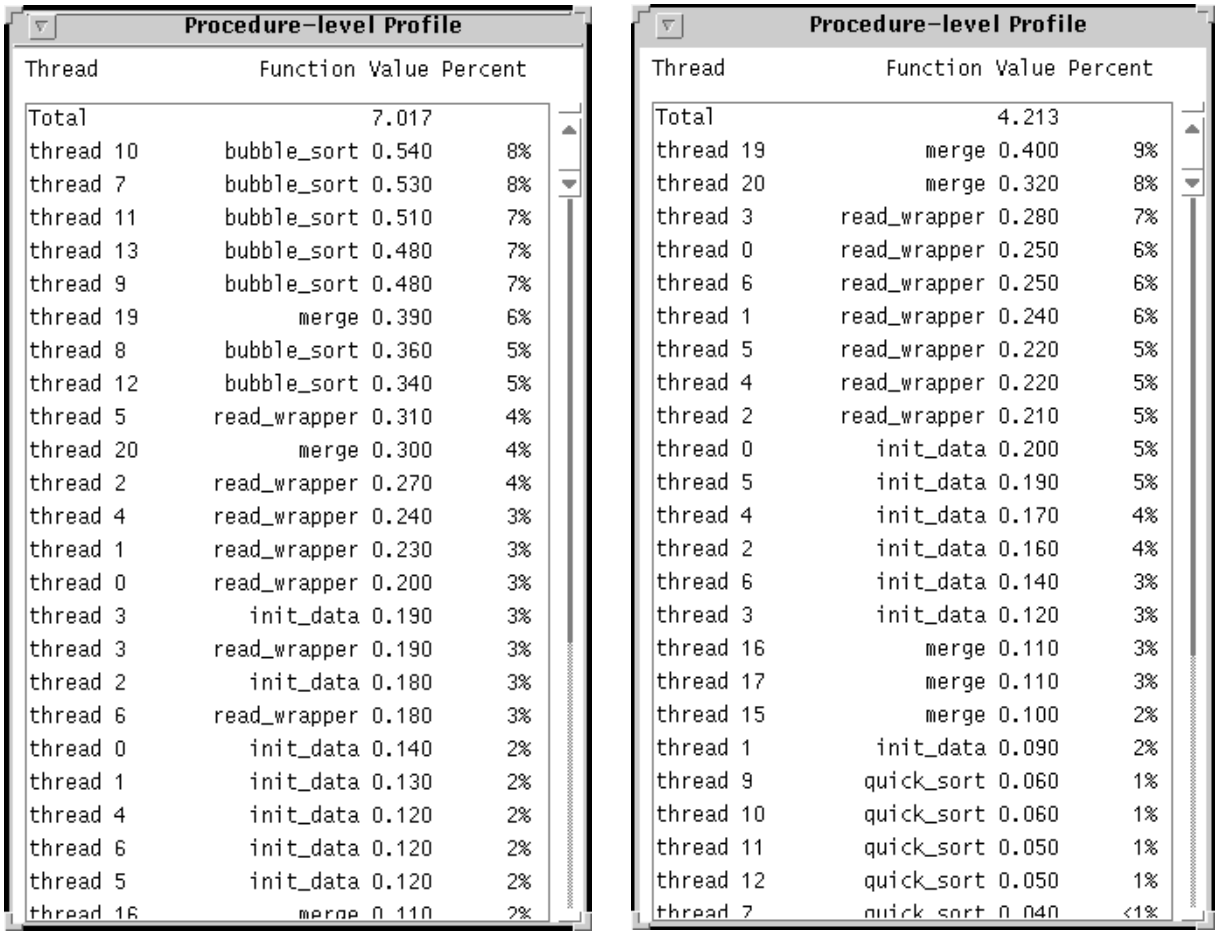
The Simulator is used to obtain the optimal execution of the program, by simply simulating a multiprocessor with as many processors as there are threads within the program. After the optimal (simulated) execution is obtained the critical path algorithm can be applied with any number of processors as argument, as discussed in Section 3.3, and the result is displayed on a function level as discussed in Section 3.2.

5. A Small Practical Example

To illustrate the critical path analysis we use a multithreaded integer sorting program. The program works in the following way. The program starts 7 threads (`init_data`) for reading random integers from a file (`read_wrapper`). When one of the threads has read its portion of the numbers it starts a new thread (`bubble_sort`) that does the actual sorting, using a the bubble sort algorithm. The file is locked using a mutex to ensure that only one thread reads the file at

once. When all threads that read the values from the file are finished, 7 new threads (`merge`) are started. Their task is to merge one of the sorted lists (from `bubble_sort`) with a global list, which will contain the sorted list of all integers. The global list is protected by a semaphore.

Intuitively, we assume that the easiest optimization effort should be to replace the bubble sort with a faster sorting algorithm such as quick sort. This is also what the Thread Analyser (`tha`) [9] suggest, see Figure 8(a), and Quantify, see Figure 9(a).



Procedure-level Profile

Thread	Function	Value	Percent
Total		7.017	
thread 10	bubble_sort	0.540	8%
thread 7	bubble_sort	0.530	8%
thread 11	bubble_sort	0.510	7%
thread 13	bubble_sort	0.480	7%
thread 9	bubble_sort	0.480	7%
thread 19	merge	0.390	6%
thread 8	bubble_sort	0.360	5%
thread 12	bubble_sort	0.340	5%
thread 5	read_wrapper	0.310	4%
thread 20	merge	0.300	4%
thread 2	read_wrapper	0.270	4%
thread 4	read_wrapper	0.240	3%
thread 1	read_wrapper	0.230	3%
thread 0	read_wrapper	0.200	3%
thread 3	init_data	0.190	3%
thread 3	read_wrapper	0.190	3%
thread 2	init_data	0.180	3%
thread 6	read_wrapper	0.180	3%
thread 0	init_data	0.140	2%
thread 1	init_data	0.130	2%
thread 4	init_data	0.120	2%
thread 6	init_data	0.120	2%
thread 5	init_data	0.120	2%
thread 16	merge	0.110	2%

(a)

Procedure-level Profile

Thread	Function	Value	Percent
Total		4.213	
thread 19	merge	0.400	9%
thread 20	merge	0.320	8%
thread 3	read_wrapper	0.280	7%
thread 0	read_wrapper	0.250	6%
thread 6	read_wrapper	0.250	6%
thread 1	read_wrapper	0.240	6%
thread 5	read_wrapper	0.220	5%
thread 4	read_wrapper	0.220	5%
thread 2	read_wrapper	0.210	5%
thread 0	init_data	0.200	5%
thread 5	init_data	0.190	5%
thread 4	init_data	0.170	4%
thread 2	init_data	0.160	4%
thread 6	init_data	0.140	3%
thread 3	init_data	0.120	3%
thread 16	merge	0.110	3%
thread 17	merge	0.110	3%
thread 15	merge	0.100	2%
thread 1	init_data	0.090	2%
thread 9	quick_sort	0.060	1%
thread 10	quick_sort	0.060	1%
thread 11	quick_sort	0.050	1%
thread 12	quick_sort	0.050	1%
thread 7	quick_sort	0.040	<1%

(b)

Figure 8: Data output from tha when analyzing the sort program. Bubble sort (`bubble_sort`) to the left and quick sort (`quick_sort`) to the right.

However, when running this application with 14336 integers (that is 2048 integers per thread) on an 8 processors Sun Enterprise 4000 we will gain nothing. Measurements have shown that the difference in execution time between the bubble sort version and the quick sort version is about 0.3%, which we consider is due to variations in load, paging, network traffic, etc., on the Enterprise 4000. The impact of the sorting (now called `quick_sort`), though, has dropped as can be seen in Figure 8(b) and Figure 9(b).

(a) (b)

Figure 9: Data output from Quantify when analyzing the sort program. Bubble sort (`bubble_sort`) to the left and quick sort (`quick_sort`) to the right.

Obviously, the critical path does not include the bubble sort threads. Using the enhanced version of VPPB with the critical path analysis we find that the 7 `bubble_sort` threads are not part of the critical path. Thus, if we instead had used VPPB in the first place, we had focused our efforts on the initiation and merging stage, since they in practice must be performed in sequential. This is shown in Figure 10.

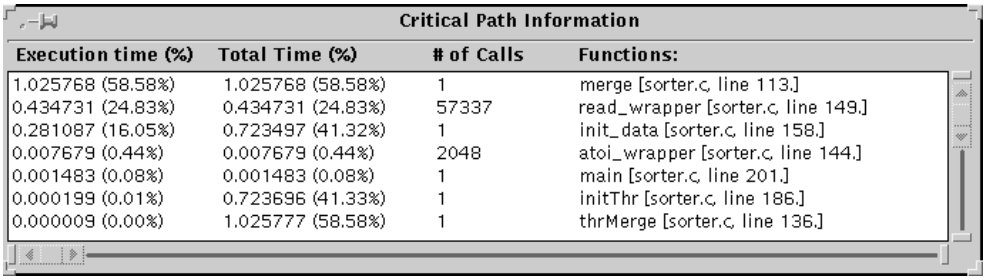


Figure 10: The critical path information given by the VPPB tool.

The overhead for recording all the information is of concern. The intrusion must not be too large, since this may change the critical path of the program. Using the program described in Section 5 we get the execution times found in Table 2. As can be seen VPPB has significantly lower overhead than `tha` and is close to Quantify.

Table 2: Normalized execution times for the same program using different tools (middle value of 5 consecutive executions).

Tool	Time
Without any data collection	1.0
tha	3.4
VPPB	1.3
Quantify	1.1

6. Discussion and Related Work

The critical path analysis is a well known technique, e.g., in combinatorial circuit design, such as we have in [3]. However applying the critical path to multithreaded programs is not very common. We have only found one case that has made use of critical path analysis [5] in the software area. The critical path analysis was applied on video applications, such as MPEG-3 decoders. The issue for the paper was to decide the critical path at the assembler level of sequential program executing on processors with an unlimited amount of ILP (Instruction Level Parallelism). Thus the effort was on breaking the dependencies found in the program at the ILP level.

Other optimization tools lack the possibility to identify the critical path. The Solaris program `tha` [9] is designed to work as `prof` [8] 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 [7]. While `tha` supports better information it also has some major drawbacks [9], e.g, it is not possible to use the standard C++ I/O primitives.

An issue for future work is how to visualize the critical path for the developer. As the VPPB tool do today, the critical path is shown as thicker lines in the execution flow graph. The information about different functions is simply a metric per function. The use of a function call graph found in many tools, e.g., Quantify [7] is not directly applicable, since the critical path may move from one thread to another due to synchronization. The synchronizations may be placed deep down in the functions and thus the critical path jumps from a function called deep down in one thread to another function deep down on another thread. The simple call graph will not cope with this kind of jumps.

Another way of presenting the performance problems of multithreaded programs is based on contention, as in Tmon [4]. The contention is based on locks and context switching overhead. Tmon use two uni-processors, one to execute the multithreaded program, and the other to gather the recorded data. The data is analyzed, to some extent, in real time. This set-up requires a fast interconnection between the uni-processors. The applicability of Tmon on multiprocessors is not addressed in [4].

7. Conclusion

Traditional performance optimization is done when the program is written. The main goal is to increase the performance of the application. The existance of a number of commercial profiling tools [7, 8, 9], shows the importance of the optimization task. Multiprocessors are used for the same reason, i.e., to increase performance. Performance optimization for programs designed for multiprocessors is at least as improtant as optimizations of sequntial programs, because tough performancd 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 total execution time. A simple example is a watchdog, which in the uni-processor case will be a part of the total execution time. However, on a multiprocessor, the watch dog may execute on its own processor. The watchdog will then not affect the total execution time of the program.

Traditional profilers, such as Quantify [7], 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 why these kind of tools gives misleading information is that they assume that all executed code segments contributes to the total execution time.

Some of the problems found in Quantify have been addressed in `tha` [9]. The data collected is presented on a thread level basis, instead of at the program level as in Quantify. This, however,

is not good either as we have shown in this paper. The reason is that `tha` assumes that all executed code segments contribute to the total execution time.

To perform efficient performance optimization we must concentrate the efforts on the critical path. The critical path is heavily dependent on the synchronizations behaviour in the multithreaded program and also the number of processors contribute. In this paper an algorithm to find the critical path has been presented. The algorithm does not only manage an ideal situation with an unlimited number of processors, but also realistic scenarios when there are a limited number of processors.

The need for the developer to connect the critical path to which functions that are involved is essential. We have presented an algorithm that does so. The algorithm aslo shows the amount of time spent in the functions during the critical path.

This method has been implemented in a performance optimization tool called VPPB. The tool pinpoints what parts of the code to optimize. A practical example shows that neither Quantify, nor `tha` can support the developer with correct information. VPPB, on the other hand, gives the correct information to support the developer in the optimization efforts.

References

[1] M. Broberg, L. Lundberg, and H. Grahn, “VPPB - A Visualization and Performance Prediction Tool for Multithreaded Solaris Programs”, *Proc. 12th Int’l Parallel Processing Symp.*, pp. 770-776, 1998.

[2] M. Broberg, L. Lundberg, and H. Grahn, “Visualization and Performance Prediction of Multithreaded Solaris Programs by Tracing Kernel Threads”, *Proc. 13th Int’l Parallel Processing Symp. (to appear)*, 1999.

[3] H-C Chen, D. H-C Du, and L-R Liu, “Critical path selection for performance optimization,” *Proc. 28th ACM/IEEE Design Automation Conf.*, pp. 547-550, 1991.

[4] M. Ji, E. Felten, and K. Li, “Performance Measurements for Multithreaded Programs,” *Performance Evaluation Review*, vol. 26, no. 1, pp. 161-170, Jun. 1998.

[5] H. Liao and A. Wolfe, “Available Parallelism in Video Applications,” *Proc. 30th Annual IEEE/ACM Int’l Symp. on Microarchitecture*, pp. 321 -329, 1997.

[6] L. Lundberg and M. Roos, “Predicting the Speedup of Multithreaded Solaris Programs,” *Proc. 4th Int’l Conf. on High-Performance Computing*, pp 386-392, 1997.

[7] Rational, “Quantify version 4.2,” *http://www.rational.com/products/quantify.*

[8] Sun Man Pages, “prof,” Sun Microsystems Inc., 1993.

[9] Sun Man Pages, “tha,” Sun Microsystems Inc., 1996.

[10] SunSoft, “Solaris Multithreaded Programming Guide,” *Prentice Hall*, 1995.