Visualization and Performance Prediction of Multithreaded Solaris Programs by Tracing Kernel Threads

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. We present a performance prediction and visualization tool called VPPB. Based on a monitored uni-processor execution, VPPB shows the (predicted) behaviour of a multithreaded program using any number of processors and the program behaviour is visualized as a graph.

The first version of VPPB was unable to handle I/O operations. This version has, by an improved tracing technique, added the possibility to trace activities at the kernel level as well. Thus, VPPB is now able to trace various I/O activities, e.g., manipulation of OS internal buffers, physical disk I/O, socket I/O, and RPC. VPPB allows flexible performance tuning of parallel programs developed for shared memory multiprocessors using a standardized environment; C/C++ programs that uses the thread package in Solaris 2.X.

1. Introduction

The thread concept in the Solaris 2.X operating system [10] makes it possible to write multithreaded C/C++ programs which can be executed in parallel. Having multiple threads does, however, not guarantee that a program will run faster on a shared memory multiprocessor. One major performance problem is that thread synchronizations may create serialization bottlenecks. Such bottlenecks are often hard to detect [6]. Removing serialization bottlenecks is referred to as *performance tuning*. In this context, it is important with tools that efficiently support the programmer in the performance tuning process, e.g., by visualizing the behaviour of, and thus the bottlenecks in, the parallel programs [2, 7, 8, 9, 11, 12, 14, 16].

In an earlier paper [2], we presented a performance prediction and visualization tool called VPPB (**V**isualization of **P**arallel **P**rogram **B**ehaviour). The target programs are written in C/C++ and run on the Solaris 2.X operating system, an environment commonly used in industry as well as in academia. Based on a monitored uni-processor execution, the VPPB system shows the (predicted) behaviour of a multithreaded Solaris program using any number of processors.

The first version of VPPB [2] could only monitor activities that took place in user space, i.e., only user level threads could be handled. Whenever a user level thread is blocked on an I/O operation, not only the user level thread is blocked, but also the corresponding kernel level thread (a.k.a. Light Weight Process, LWP) is blocked. The Solaris operating system then tries to dynamically create a new LWP to continue to execute some other user level thread. The first version of the tool could not manage several LWPs simultaneously and thus no blocking I/O.

The main contribution in this paper is an extension that overcomes the limitations above *by tracing the kernel level threads as well*. By recording all state transitions in the OS kernel for the LWPs, it is now possible to have several LWPs running at the same time. The tool can now handle various I/O activities, including physical disk I/O, socket communication, and RPC calls.

Validation has been done using ten benchmarks from the SPLASH-2 suite [15] and a skeleton of an I/O intensive commercial telecommunication application [6]. The simulated performance predictions were compared to real executions on a multiprocessor with eight processors. The maximum error of the predictions for those application are less than 10% for all applications and less than 4% for more than half of the applications.

The paper is structured in the following way. Section 2 gives a short overview of VPPB. In Section 3 the tracing part is described along with a discussion of how we sort and manage the collected data. The simulation part is described in Section 4. The validation part is found in Section 5 and the related work is found in Section 6. The paper concludes in Section 7.

2. Overview of VPPB

The VPPB consists of three major parts, the *Recorder*, the *Simulator*, and the *Visualizer*. The workflow when using the VPPB system is shown in Figure 1. The developer writes the multithreaded program (a) in Figure 1.

When starting the monitored execution (b) on a uniprocessor, the *Recorder* is automatically placed *between* the program and the standard thread library. Every time the program uses the routines in the thread library, the call

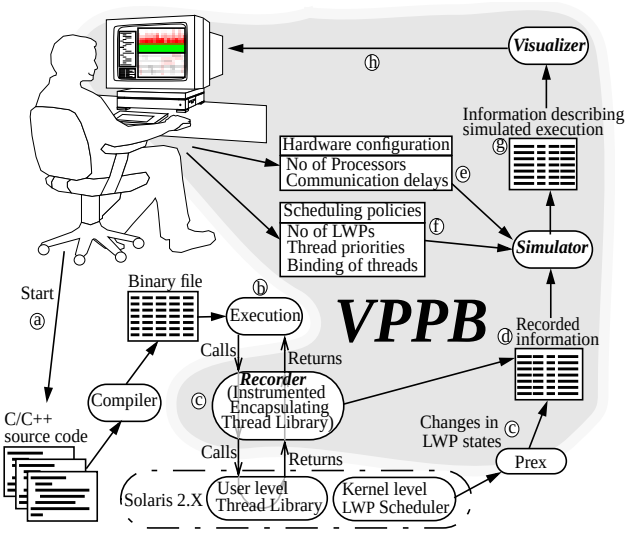


Figure 1: A schematic flowchart of the VPPB system.

passes through the *Recorder* (c) which records information about the call. The *Recorder* then calls the original routine in the thread library. Whenever an LWP does a state change the operating system informs a program called `prex`. `Prex` is a standard program on the Solaris platform used to create logfiles about LWP state changes. When the execution of the program finishes the two data files (from *Recorder* as well as `prex`) are sorted in time order and transformed into the file format (d) used in the first version of this tool. The recording is done without recompilation or relinking of the application.

The *Simulator* simulates a multiprocessor execution. The main input for the simulator is the *recorded information* (d) in Figure 1. The simulator also takes the hardware configuration (e) and scheduling policies (f) as input. The output from the simulator is information describing the predicted execution (g).

Using the *Visualizer* the predicted parallel execution of the program can be inspected (h). The Visualizer uses the simulated execution (g) as input. The simulated execution is shown as two graphs; one parallelism graph and one execution flow graph, as shown in Figure 2. It is possible for the developer to click on an event, get the source code displayed, and the line making the call that generated the event highlighted. With these facilities the developer may detect problems in the program and can modify the source code (a). Then the developer can re-run the execution to inspect the performance change.

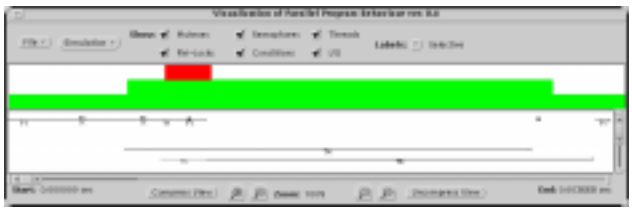


Figure 2: The execution flow and parallelism graphs.

3. Monitoring the Program Behaviour

The main contribution in this second version of VPPB is the ability to trace kernel threads in addition to user level threads. User level threads are non-preemtive and are executed on kernel threads. The kernel threads are preemtive and scheduled in a time-slice manner. This gave some implications in the first version of VPPB since only user level threads were traced and thus only one kernel thread was allowed. One example is the concept of spinning locks. Whenever a user level thread enters a spinning lock, the other threads can not pre-empt the thread. This makes the program to hang on the spinning lock. Another example is I/O. Whenever a user level thread blocks, the corresponding kernel thread is also blocked. The Solaris operating system will then, in order to continue execution of the multithreaded program, create new kernel threads for the other user level threads to execute on.

In Solaris, threads are used at two levels (see Figure 3) [13]. The application programmers use user-level threads for expressing parallel execution within a process. Kernel threads (a.k.a. LWPs) are used within the operating system kernel. The kernel knows nothing about user-level threads.

The LWPs are scheduled by the kernel in a preemtive round-robin fashion with timeslices from 20 milliseconds up to more than 200 milliseconds depending on the age of the LWP etc. A user level thread can be bound to a specific LWP or be floating around on the LWPs that are free.

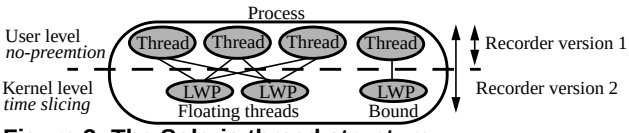


Figure 3: The Solaris thread structure.

3.1. Tracing user level threads

In order to trace the behaviour of the program when executed on a uni-processor, the Recorder dynamically inserts probes when the program starts. The probes are inserted at specific events, i.e., before and after calls to the thread library, and they do not affect the behaviour or function of the program. For each event, the probes record the following information: when an event has occurred; the type of event, e.g., locking of a mutex; which object the event concerns, e.g., the identity of the mutex being used; the identity of the thread generating the event; and the location of the event in the source code.

The user level tracing has been extended to capture some primitives in the `libc` library. In particular the `open`, `close`, `read`, `write`, and `fsync` primitives. To allow RPC calls [1] generated by, e.g., `rpcgen` [1], we also capture the primitives `putmsg`, `getmsg`, and `pipe`. One I/O operation will produce two different events in the log file; one event corresponding the CPU time needed to

execute the operation, and another event, called IO_wait, that corresponds to the time the thread was blocked.

One single I/O operation may consist of using the CPU several times with waiting times between. In order to keep the log file as small as possible all CPU time for one single I/O operation are concatenated into one. This concatenation is also done with the waiting time as well.

Our current implementation of the Recorder is based on TNF-probes [5]. The basic idea is to insert a new library between the program and the dynamically linked thread library. This is achieved by using the built-in facilities of Solaris with run-time linking and shared objects. The insertion is handled at program start-up by the run-time linker via the program `prex`. We also trace the source code location of the call to a probe.

3.2. Tracing kernel level threads

The first version of VPPB were restricted to have only one single LWP running at any time during the execution of the multithreaded program. The reason for this is that we could not trace the context switches of the LWPs in the kernel. This tracing can be done without any changes since Solaris 2.5 already have probes inside the kernel that trace this. The probes are implemented by using TNF-probes [5]. The super-user can extract the information by the `prex` command which gives as output a binary file which has to be merged with the ordinary user level trace from the Recorder. The approach that allows several LWPs to execute also makes it possible to trace programs with spinning locks. However, there is still a problem with poor performance prediction for programs with spinning locks as we will show in Section 5.1.

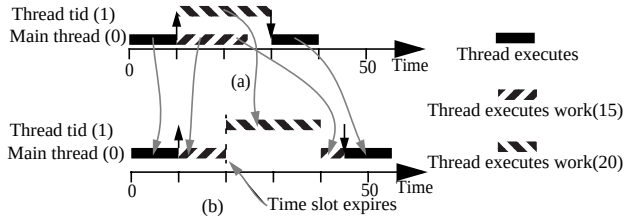
3.3. Basic merging and sorting of the two TNF-files

An example is used to illustrate the basic principles of how the sorting is done. The code is found in Figure 4. The main thread creates thread `tid` bound to an LWP. Then the main thread does some CPU bound work and then waits for thread `tid` and exits. The thread `tid` does some CPU bound work and exits.

The optimal execution on two processors can be found in Figure 5(a). Note that we have made some considerable simplifications in this example discussed at the end of this section, e.g., all thread primitives take no time to perform. The main thread starts at time 0 and creates thread `tid` at time 10. From time 10 to 25 the threads execute in parallel, then the main thread wants to join with thread `tid`. Thread `tid` will continue executing until time 30 and then exits. By this time the main thread can resume execution for the last 10 time units. Since the Recorder works on a uni-processor, the two threads can not execute in parallel, thus the execution may look like in Figure 5(b). At time 10 the main thread creates the thread `tid`, which waits for the CPU. At time 20 the time slot expires and the main thread leaves the CPU and the thread `tid` starts executing. When the thread `tid` has finished its execution, the main thread has 5 time units left until it reaches the join with thread `tid`. At time 45 the two threads has joined and the main thread executes until the end at time 55.

```
void *thr(void *in) {
  work(20); /* CPU bound work for 20 time units */
}
void main() {
  thread_t tid;
  thr_create(0, 0, thr, 0, THR_BOUND, &tid);
  work(15); /* CPU bound work for 15 time units */
  thr_join(tid, 0, 0); }
```

Figure 4: The source code of the example.



User level log file:

Time	LWP	Thread	Event
0	0	0	Start collect
10	0	0	Thread create 1
40	1	1	Thread exit
45	0	0	Thread join 1
55	0	0	Thread exit

Kernel level log file:

Time	LWP	Thread	Event
0	0	0	LWP starts running
20	1	1	LWP starts running
40	0	0	LWP starts running

(c)

Merged log file:

Length	Thread	Event
0	0	Start collect
10	0	Running
0	0	Thread create 1
15	0	Running
20	1	Running
0	1	Thread exit
0	0	Thread join 1
10	0	Running
0	0	Thread exit

(d)

Figure 5: Merging the user and kernel level log files.

The two log files generated during the execution are found in Figure 5(c). The user level log file consists of a start event at time 0 and at the same time in the kernel level log file the corresponding LWP starts running. The next event that occurs is the thread create event at time 10. However, the newly created thread `tid` does not start to execute until time 20, as seen in the kernel log file, when the corresponding LWP starts running. At time 40 the thread `tid` has finished, as indicated in the user level log file as well as in the kernel level log file, since the main thread continues to execute. At time 45, the threads joins as indicated in the user level log file, and finally, the main thread stops executing at time 55.

Now, we take a close look at the merging of the two log files into one single log file. During the merging we want to eliminate the concept of LWPs, thus only representing the behaviour of the user level threads. The resulting log file is shown in Figure 5(d). At time 0 we have the start event for the main thread and we see that its LWP is executing in the kernel log file in Figure 5(c). The LWP is running until time 20, i.e, the main thread is running 10 time units until it creates thread `tid` at time 10. At time 40 the thread `tid` has finished its execution, and we have to calculate how many time units it has executed as well as

the time the main thread executed. The LWP switch occurred at time 20, thus the main thread must have been running for 10 time units (from time 10 to 20) and the thread `tid` has been running for 20 time units (from time 20 to 40). The next event is at time 45. Since the main thread's LWP started executing at time 40, the main thread must been running for five time units before the join, and in order to make the merged log file compact, we add the previous running time for the main thread with this one. The resulting merged log file is found in Figure 5(d).

Other issues, must be considered, e.g., the kernel log file does not indicate which thread it is executing, the mapping must be done via the LWP identity. Also, the kernel threads are started before the corresponding user level thread starts and representing that time in the user level thread must be done in reverse order. The threads may float around on several LWPs, other processes may interact and put LWPs in both running and runnable states. Each user level event must have a start and stop time in order to measure how long the primitive took to execute. This made the user level log file to include nine lines in reality and the kernel level log file have 57 lines. Finally, things do not occur at the exactly same time in the kernel level log file as in the user level log file, and vice versa.

3.4. Merging and sorting the TNF-files with I/O events

We keep the example in Figure 4, but the main thread does not call `work(15)`, instead it writes to a file using the C standard primitive `write`. We have intentionally omitted the necessary `open` and `close` primitives in order to simplify the example. The optimal execution on two processors will look as in Figure 6(a). The main thread starts at time 0 and creates the thread `tid` at time 10. Immediately after, the main thread initiates a write to the disk. The writing of the file is finished at time 25 and the join with thread `tid` is reached at time 30. The main thread is finished at time 40. The thread `tid` is running between time 10 and 30, i.e., 20 time units.

The execution on a uni-processor system looks like in Figure 6(b), where we take a closer look at the write operation, time 10 to 45. The write operations include two important issues. The first issue is the time required by the processor to perform the write operation. The second is the time required by the disk to perform the write, meanwhile the processor may execute the other LWP and its thread. This is indicated in Figure 6(b) at time 15 since the thread `tid` starts executing when the main thread is waiting for the disk. Thread `tid` leaves the processor at time 35, and the write can be completed at time 40, the execution of the main thread continues and join with the thread `tid`, and finally ends at time 50.

The user level log file and the kernel level log file are found in Figure 6(c). In the kernel level log file at time 15 the main thread leaves the processor and the thread `tid` starts executing. At time 20, the waiting for the disk is over and the main thread's LWP can be put in the runnable state. At time 35 in the kernel level log file, the main thread's LWP starts executing to completion.

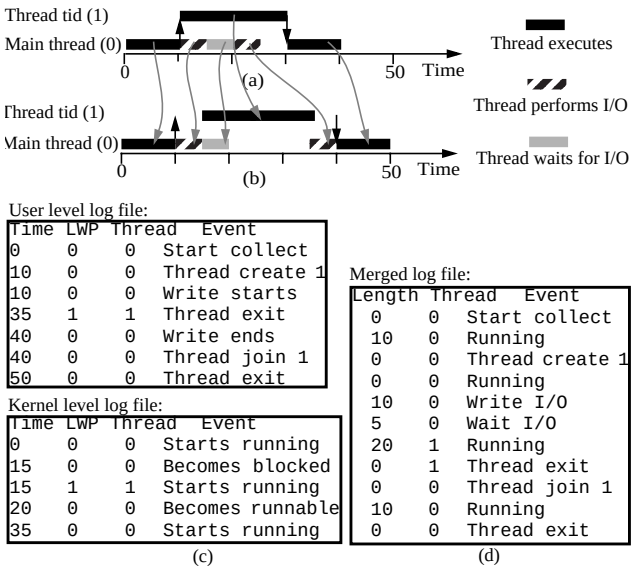


Figure 6: Merging the user and kernel level log files.

Whenever an I/O operation is performed, we collect two kinds of time information: The time that the I/O required the processor; and the time the I/O was waiting for the disk. These two times are represented as two events in the log file. The first event is the processor bound part of the I/O operation (write I/O) and the second event is the time the I/O operation was blocked (wait I/O) as shown in Figure 6(d). The I/O is the time from 10 to 40 in the user level log file. At time 15 in the kernel level log file the main thread becomes blocked. Thus, five time units execution on the CPU was needed before the actual writing to the (physical) disk. In the kernel level log file the main thread becomes runnable at time 20, i.e., the (physical) disk is ready and the wait for (physical) I/O is over. At time 35 the main thread start execute again as seen in the kernel log file. The write operation ends at time 40, i.e., the main thread needed yet five time units execution on the CPU in order to end the I/O operation. The two CPU bound parts of the I/O operation is added together. Otherwise the merging is performed as described in Section 3.3.

Note once again that the example is simplified, e.g., during one I/O operation the processor might have to wait several times for the (physical) disk and have to execute in between the waitings. Also, all state transitions between user space and kernel space are traced since each fundamental I/O operation is a system call. This make, together with the issues stressed earlier, that the user level log file consists of 15 lines in reality and the kernel level log file consists of 1340 lines in the case of writing a 10 million bytes large file. The merged log file consists of 15 lines.

4. Predicting the Program Execution

The Simulator mimics the scheduling in Solaris 2.5 [10]. The modeling of I/O follows the discussion in Section 3.1 with one part of the I/O to be considered as CPU bound and one part as waiting time. The Simulator first simulates the CPU bound time in a chunk just as any other event that only takes time. Then, the Simulator simulates the waiting for I/O to be completed, which is similar to a sleep primitive. However we simulate that only one thread can perform an I/O operation at the same time and only one I/O request, i.e., `IO_wait`, may be issued at the same time. This seems to mimic the OS quite accurate.

There are advantages and disadvantages of merging the parts of an I/O operation into one CPU bound part and one part that is representing the waiting time. The obvious advantage is that the log file will be shorter than if all the individual parts in were stored in the file. The obvious disadvantage is that we loose some information. However, this loss of information could actually be regenerated, to some extent, in the simulator by assuming that the internal I/O buffer within the kernel is of a particular size. Thus it is simple to calculate how many times the write operation must enforce a physical write, and thus a wait period, on the disk. This facility is not implemented and left to future development of the tool.

Much of the Simulator is kept from the first version of the tool [2], e.g., threads may be bound or unbound as well as the number of processor simulated is adjustable. Creating a bound thread is simulated to take 6.7 times longer than an unbound thread [13]. A synchronization on a semaphore takes 5.9 times longer with bound threads than unbound. The value is found in [13] and is incorporated in the simulator for semaphores as well as for mutexes, condition variables, and read/write locks.

5. Validation of the Predictions

The validation of the predictions was made using a subset of the SPLASH-2 benchmark suite [15] and a skeleton of a telecommunication application that uses a lot of I/O in different manners. All executions were made on a Sun Ultra Enterprise 4000 with eight processors and 512 MByte memory. Our measurements showed that the execution time overhead for doing the recordings was very small. The maximum overhead, was obtained for Raytrace in the SPLASH-2 benchmark suite, was 31% of the total execution time. More than half of the log files caused less than 2% overhead. More than 75% of the log files were less than 1.5 Mbyte in size. The largest log file, which was obtained for Radiosity, was 19 MByte. This file could be handled without any problems. Consequently, neither the execution time overhead, nor the size of the log files caused any problems for these programs.

5.1. The SPLASH-2 benchmark suite

The programs that we use from the SPLASH-2 suite are: Ocean (with data set 514-by-514 grid), Water-Spatial (512 molecules, 30 time step), FFT (4M points), Radix (16M keys, radix 1024), LU (contiguous, 768x768 matrix, 16x16 blocks), Raytrace (teapot), Barnes (2048 bodies), Cholesky (tk29.O), FMM (2048 bodies), and Radiosity (Default, batch mode, en 0.1). Since the SPLASH-2 programs are designed to create one thread per physical processor, one log file was generated for each processor setup.

The first version of VPPB could only use five of the benchmarks [2]. This was because of spinning locks, as described in Section 3. Another issue is task stealing, i.e., a thread steals a waiting job from another whenever the stealing thread is idle. In the first version of the tool we could not handle several LWPs. This led to the result that the first thread that begun execute would steal all jobs from the other threads (which never got a chance to execute on the CPU). Thus, the recording showed that all work were done by one single thread and the others did nothing. When simulating this on a multiprocessor the load imbalance would be at its maximum. In this second version of VPPB we can handle several LWPs and thus the jobs may distribute better.

The 5 benchmarks we could use in the first version of the tool are discussed first. Then, we will look at the other benchmarks as well. Table 1 shows the measured and predicted speed-up for 5 programs from the SPLASH-2 benchmark suite we could use in the first version of VPPB. The real speed-up is the middle value of 5 executions of the program. The error is defined as |((Real speed-up) - (Predicted speed-up))/(Real speed-up)|, where $|-x| = |x| = x$, for all $x > 0$. As we can see in Table 1 the maximum error is 5.2%, which we consider to be a very low error. It is also an improvement with more than 16% as compared to the first version of the tool [2].

Table 1: Speed-ups for the first 5 benchmarks.

Application	2 processors			4 processors			8 processors		
	Pred	Real	Error	Pred	Real	Error	Pred	Real	Error
Ocean	1.95	1.97	1.0%	3.75	3.87	3.1%	6.47	6.65	2.7%
Water-spatial	1.97	1.99	1.0%	3.86	3.95	2.3%	7.27	7.67	5.2%
FFT	1.52	1.55	1.9%	2.06	2.14	3.7%	2.57	2.62	1.9%
Radix	1.99	2.00	0.5%	3.98	3.99	0.3%	7.91	7.79	1.5%
LU	1.82	1.79	1.7%	3.08	3.15	2.2%	4.72	4.82	2.1%

The benchmarks Barnes, Cholesky, FMM, and Radiosity could not be used in the first version of the tool since they use spinning locks. When a thread runs into a spinning lock, it will stay there for (in average) half a time slot until another thread can execute and possibly change the value of the lock. Raytrace uses a task stealing scheme, that might cause load imbalance if the tasks were executed

in the same order on a multiprocessor as on a uni-processor. These problems are clearly shown in Table 2 since the error may be as high as 62% (FMM).

Table 2: Speed-ups for the 5 benchmarks with *(without in italic)* spinning locks / load imbalance.

Application	2 processors			4 processors			8 processors		
	Pred	Real	Error	Pred	Real	Error	Pred	Real	Error
Raytrace	1.67	1.73	3.5%	2.19	2.69	18.6%	3.38	3.73	9.4%
Raytrace	*1.71*	*1.72*	*0.6%*	*2.42*	*2.50*	*3.2%*	*3.24*	*3.28*	*1.2%*
Radiosity	1.74	1.91	8.9%	3.09	3.72	16.9%	5.25	6.20	15.3%
Radiosity	*1.91*	*1.86*	*2.7%*	*3.63*	*3.75*	*3.2%*	*5.97*	*6.31*	*5.4%*
Barnes	1.72	1.95	11.8%	2.85	3.34	14.7%	4.28	5.77	25.8%
Barnes	*1.97*	*1.97*	*0.0%*	*3.57*	*3.38*	*5.6%*	*5.84*	*5.33*	*9.6%*
Cholesky	1.37	1.62	15.4%	1.98	2.31	14.3%	2.42	2.89	16.3%
Cholesky	*1.59*	*1.62*	*1.9%*	*2.21*	*2.31*	*4.3%*	*2.80*	*2.85*	*1.8%*
FMM	1.58	1.90	16.8%	1.99	3.50	43.1%	1.98	5.19	61.8%

The error was caused by the time a thread was bound to stay spinning on a lock until the time slice was over. By increasing the number of threads executing on a uni-processor, and thus causing more threads spin on the spinning locks, we exaggerated that behaviour. Another way of testing this is to, for each iteration in the spinning loop, voluntarily give up the processor and thus decrease the overhead with spinning locks. Tests conducted on Cholesky and FMM confirmed our thoughts.

The spinning locks in Radiosity, Barnes, and Cholesky could easily be replaced by semaphores and mutexes. The spinning locks in FMM could not be replaced easily and we did not change that program, since we do not want to perform too large changes to the programs. The results after replacing the spinning locks with blocking locks are found as *italic* rows in Table 2. Further, Raytrace has been slightly modified to avoid task stealing. As can be seen, the error drop dramatically, in most cases with at least 83%. The maximum error is now 9.6%.

5.2. The Billing Gateway

None of the SPLASH-2 programs contained any (significant) I/O, and in order to validate the I/O we used a telecommunication application developed by Ericsson Software Technology called Billing Gateway (BGw) [6]. We used a skeleton version of the BGw for our validation because the real BGw has an advanced graphical user interface, and that the application uses customer adjusted data input formats which made it hard for us to find proper loads. The skeleton was created together with Ericsson to mimic the characteristics of the original BGw and ended up consisting of around 1000 lines of C++ code. The original BGw consists of about 100,000 lines of code.

A principal sketch over the BGw (skeleton) is found in Figure 7. The BGw (skeleton) works as a kind of filter. The Socket receivers get the information to be filtered through a socket. The information chunk is 1 Mbyte large and consists of integers. As soon as all data are received the information is stored on disk, the disk is synchronized, i.e., all data is physically written to disk, and the receiver is ready for the next chunk of data. The Sorter reads the file created by the Socket receiver and puts all the integers in a binary tree. As workload all integers are converted to floating points and back to integers again during a traversion of the tree, this is repeated 1024 times. Finally, the Sorter stores the odd integers into one file and the even integers into another file. As previously the information on the disk are synchronized. The Consumers then read the data and discards it. The skeleton we use has eight Socket receivers, eight Sorters, and 16 Consumers as shown in Figure 7. Each Socket receiver were fed with two 1Mbytes chunks.

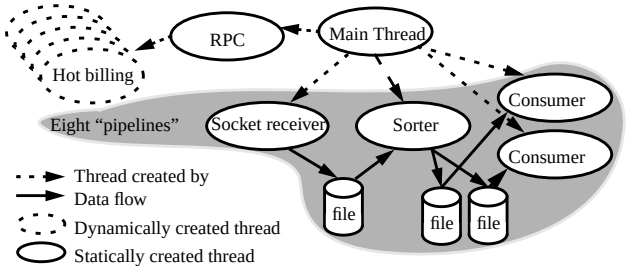


Figure 7: The organization of the BGw skeleton.

The skeleton is also able to consider hot billing which, as in the original BGw, is managed by RPC. Once an RPC call is made to the BGw, a new thread is created to process the data. The processing of the hot billing data is the same as described above. However no data are stored on disk. The skeleton received 5 RPC calls of 5 kbytes each of hot billing data. The skeleton performs, on average, approximately 800 Kbytes per second of I/O traffic on a Sun Enterprise 4000 with eight processors.

The result of the BGw skeleton can be found in Table 3. As can be seen, the predictions for this I/O application is very accurate, at most with 6.1% error.

Table 3: Speed-ups for the BGw skeleton.

2 processors			4 processors			8 processors		
Pred.	Real	Error	Pred.	Real	Error	Pred.	Real	Error
1.99	1.98	0.5%	3.98	3.75	6.1%	6.44	6.17	4.4%

6. Related Work

Some performance visualization tools show the behaviour of one particular monitored multiprocessor execution of the parallel program [3, 7, 16]. The problem with this approach is that there is no support for detecting bottlenecks which appear on another number of processors. TNF probes are used for a similar purpose in [3]. Another

tool [4] focus on the contention in multithreaded programs.

There are a number of tools, [8, 9, 14, 12, 11], which make it possible to visualize the (predicted) behaviour of a parallel program using any number of processors. However, these tools are either developed for message passing systems or for non-standard programming environments.

7. Conclusion

In this paper we have presented an improved version of the VPPB tool. This tool makes it possible to predict the speed-up and visualize the behaviour of a multithreaded C/C++ application using the Solaris 2.X thread package. It is a common environment in both industry and academia.

Our approach is based on a monitored uni-processor execution of the multithreaded program. Based on recordings from this execution and some parameters describing the target multiprocessor, the behaviour and execution time of the multithreaded program is predicted. The first version of the tool was not able to handle I/O. The main improvement in this version is that I/O can be handled because we monitor kernel threads as well. The monitoring is performed using the TNF probes in Solaris.

We have validated the predicted speed-up using the SPLASH-2 suite [15], an I/O intensive skeleton of a telecommunication application, and a multiprocessor with eight processors.

The first version of the tool could only handle five of the applications in the SPLASH-2 suite. The current version of the tool can handle all applications in the test suite. The maximum error in the speed-up predictions for the first five applications is 5%; in most cases the error is much smaller. The other applications in SPLASH-2 could not be handled by the first version of the tool because they contain spinning locks. These applications can now be handled. The maximum speed-up prediction error for these applications is relatively large, up to 62%. However, if we replace spinning locks with semaphores and mutexes the predictions become better. The maximum error was less than 10%, and more than half of the predictions had an error of less than 4%.

To validate the speed-up predictions for applications heavily depending on I/O we used a skeleton version of a large commercial telecommunication application. This validation shows that the simulation of I/O is very accurate; the maximum error is only 6%.

In the current version of the tool, we can handle any multithreaded Solaris program. Our technique requires no modification of the source code of the multithreaded program. The recording overhead is small for most applications, e.g., less than 2% of the total execution time for more than half of the SPLASH-2 applications.

References

[1] J. Bloomer, “Power Programming with RPC,” O’Reilly & Associates, Inc., ISBN 0-937175-77-3, 1992.

[2] 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.

[3] B. Cantrill and T. Doeppner, “ThreadMon: A Tool for Monitoring Multithreaded Program Performance,” *Proc. 30th Hawaii Int’l Conf. on System Science*, pp. 253-265 Vol. 1, 1997.

[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] S. Kleiman, D. Shah, and B. Smaalders, “Programming with threads,” Prentice Hall, 1996, ISBN 0-13-172389-8.

[6] L. Lundberg and D. Häggander, “Optimizing Dynamic Memory Management in a Multithreaded Application Executing on a Multiprocessor,” *1998 Int’l Conf. on Parallel Processing*, pp. 262-269, 1998.

[7] G. J. Nutt, A. J. Griff, J. E. Mankovich, and J. D. McWhirter, “Extensible Parallel Program Performance Visualization,” *Proc. Mascots ‘95*, pp. 205-211, 1995.

[8] E. Papaefstathiou, D. J. Kerbyson, G. R. Nudd, and T. J. Atherton, “An Overview of the CHIP^{3}S Performance Prediction Toolset for Parallel Systems,” *Proc. 8th ISCA Inte’l Conf. on Parallel and Distributed Computing Systems*, pp. 527-533, 1995.

[9] V. Pillet, J. Laboarta, T. Cortes, and S. Girona, “PARAVER: A Tool to visualize and Analyse Parallel Code,” University of Politencia, Catalonia, CEPBA/UPC Report No. RR-95/03, Feb. 1995.

[10] M. L. Powell, S. R. Kleiman, S. Barton, D. Shah, D. Stein, and M. Weeks, “SunOS 5.0 Multithreaded Architecture,” Sun Soft, Sun Microsystems Inc., Sep. 1991.

[11] S. R. Sarukkai and D. Gannon, “SIEVE: A Performance Debugging Environment for Parallel Programs,” *J. of Parallel and Distributed Computing*, Vol. 18, pp. 147-168, 1993

[12] Z. Segall and L. Rudolph, “PIE: A Programming and Instrumentation Environment for Parallel Processing,” *IEEE Software*, 2(6):22-37, Nov. 1985.

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

[14] S. Toledo, “PERFSIM: A Tool for Automatic Performance Analysis of Data-Parallel Fortran Programs,” *Proc. 5th Symp. on the Frontiers of Massively Parallel Computation*, IEEE Computer Society Press, pp. 396-405, Feb. 1995.

[15] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta, “The SPLASH-2 Programs: Characterization and Methodological Considerations,” *Proc. 22nd Annual Int’l Symp. on Computer Architecture*, pp. 24-36, Jun. 22-24, 1995.

[16] J. Yan, S. Surukkai, and P. Mehra, “Performance measurements, Visualization and Modelling of Parallel and Distributed Programs using the AIMS Toolkit,” *Software-Practice and Experience*, 25(4):429-461, Apr. 1995.