Performance Tuning of Multithreaded Applications for Different Multiprocessor Platforms

Magnus Broberg

Department of Computer Science
University of Karlskrona/Ronneby
Soft Center, S-372 25 Ronneby, Sweden
Magnus.Broberg@ipd.hk-r.se

Abstract

Performance tuning of a parallel application is often hard. The use of standards, such as POSIX threads, makes it possible to move a multithreaded application from one platform to another. Doing performance tuning for many platforms are even tougher since the implementation of the standards may vary on different operating systems. The developer need tools for analysing how the application will behave on different operating systems in order to do adequate performance tuning.

In this paper we present a technique based on cross-simulation that will solve the issues above. The technique uses a monitored execution of a multithreaded application on a single processor workstation running the Solaris operating system. Then the technique, which has been implemented in a tool, simulates a multiprocessor with an arbitrary number of processors running either Solaris or Linux. The tool then displays the behaviour of the application on the selected target configuration.

Validation, using a subset of the SPLASH-2 benchmark suite, shows that the tool predicts speed-ups correctly. The average error in the predicted speed-up when simulating Linux is 5.8%. All this can be done using the ordinary Solaris workstation on the developer's desk, without even having a multiprocessor.

1. Introduction

Writing parallel applications for multiprocessors is often hard. In many cases the reason for using a multiprocessor is to achieve more computing power for the application. Doing performance tuning of a multithreaded application for a multiprocessor is an important but tedious task and few tools are available for the developer. Most tools require the application to be executed on the target multiprocessor, e.g., [5, 7, 9, 18].

When introducing standards, such as POSIX threads [3], the aim is to make applications portable. This makes it possible to move one application from one operating system, e.g. Solaris, to another operating system, e.g. Linux. However, it is not obvious that an application tuned for one operating system will run efficiently on another operating system. Different characteristics for the operating systems may lead to different tuning or trade-offs when tuning the application for both operating systems.

The developer then has to tune the application for several operating systems, using different tools for the different operating systems. The developer must also have access to all combinations of multiprocessors and operating systems in order to do the performance tuning. Not only a tedious task to do, in many cases it is impossible (and expensive) to have all the machines needed.

In this paper we present a tool called VPPB (Visualization of Parallel Program Behaviour). The tool is capable of executing a multithreaded application (using POSIX Threads) on a single processor workstation with Solaris and shows how the execution would be if the application were executed on an SMP (Symmetric MultiProcessor) with an arbitrary number of processors, running

either Solaris or Linux. The predictions are with good accuracy. The tool has perviously been used for prediction on Solaris for Solaris-threads [1, 2, 15]. The tool has now been extended to cover the major parts of POSIX threads as well. The tool can now also mimic the Linux 2.2 operating system. Based on an execution of an application on a single processor Solaris workstation, the tool is able to predict the behaviour of the application on a multiprocessor running Linux. Thus, the tool is able to perform cross-simulation. We use the term cross-simulation when a program is monitored on one operating system and then simulated for another (target) operating system. The performance tuning of the application can be done on the developer's ordinary workstation without the need for a multiprocessor.

The rest of this paper is as follows. In Section 2 we give an overview of the VPPB system and in Section 3 Solaris, Linux, and POSIX threads are briefly discussed. Section 4 shows the validation of the predictions and Section 5 shows that the same application will execute differently on different operating systems. Section 6 discusses the result and points at some future work, the conclusions are found 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, compiles it, and an executable binary file is obtained. After that, the program is executed on a uni-processor. When starting the monitored execution (b), 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 passes through the *Recorder* (c) which records information about the call, i.e., the identity of the calling thread, the name of the called routine, the time the call was made, and other parameters. The *Recorder* then calls the original routine in the thread library. When the execution of the program finishes all the collected information is stored in a file, the *recorded information* (d). The recording is done without recompilation or relinking of the application, making our approach very flexible.

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 configuration (e) as input, such as the target operating system, number of processors, etc. The output from the simulator is information describing the predicted execution (f).

Using the *Visualizer* the predicted parallel execution of the program can be inspected (g). The Visualizer uses the simulated execution (f) as input. The main view of the (predicted) execution is a Gant diagram. When visualizing a simulation, it is possible for the developer to use the mouse to click on a certain interesting 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. The VPPB system is designed to work for C or C++ programs that uses the built-in thread package [7] and POSIX threads [3] on the Solaris 2.X operating system. The Simulator is able to simulate both Solaris and Linux.

3. POSIX Threads and Some Key Differences Between Solaris and Linux

Both Solaris and Linux 2.2 implement the POSIX thread interface [3]. However, there are differences between the implementations. Solaris uses a two-layered approach illustrated in Figure 2 [6, 12]. In the user level there are user level threads. These threads are executed in a non preemptive way. This means that the thread must, in some way, voluntarily give up the execution in favor of another thread. This could be done explicitly or implicitly by calling a synchronization primitive that blocks the thread or releases a previously blocked thread with higher priority.

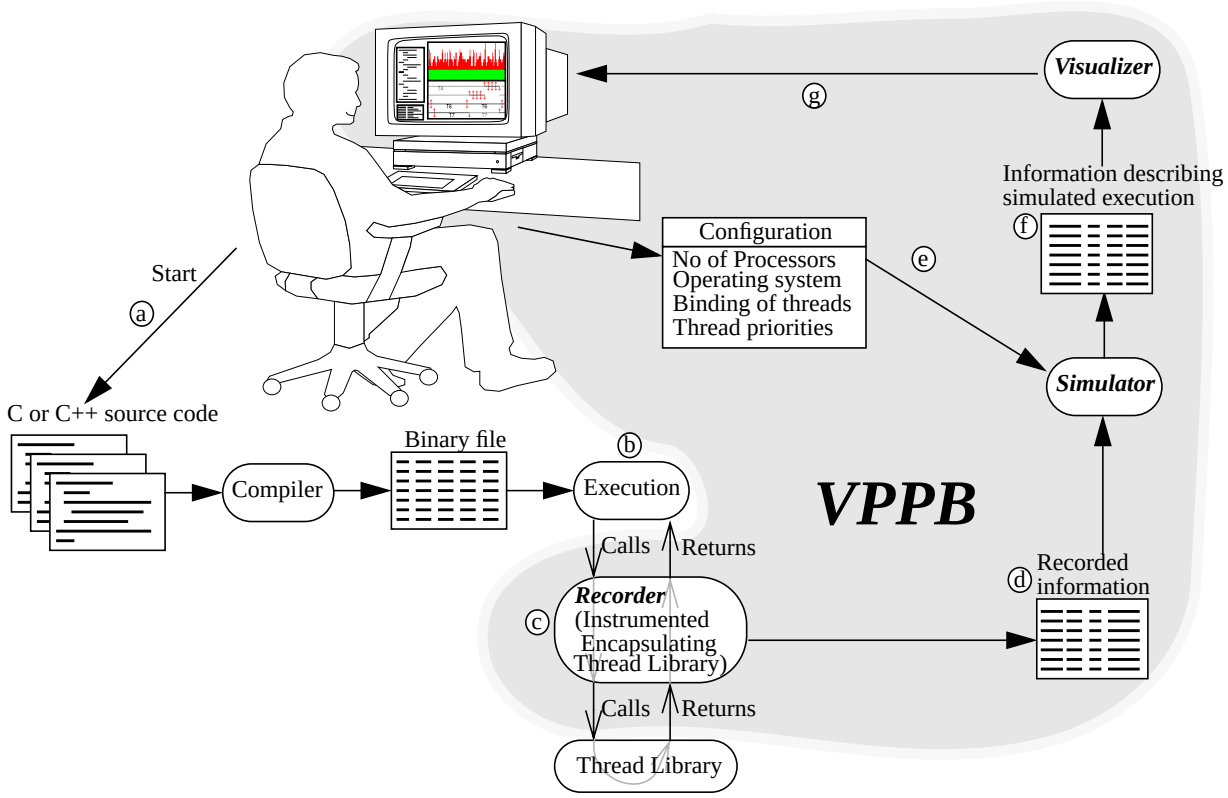

Figure 1: A schematic flowchart of the VPPB system.

The kernel level threads (also known as LightWeight Process, LWP [15]) are scheduled by the kernel in a pre-emptive fashion. The scheduling is priority based, with a round robin policy on each priority level. The priority is changed over time, the longer time an LWP executes the lower priority and longer time slices it gets. Each CPU has it's own run queue. In order to migrate an LWP to another CPU the LWP must have been in the queue for some pre-defined time without being selected by the current CPU. The user level threads are executed by the LWPs. This means that the kernel only schedules LWPs, the user level threads are not seen by the kernel. From the user level threads' point of view the LWPs could be seen as virtual CPUs. A user level thread might be bound to a specific LWP, i.e., the user level thread will (indirectly) be scheduled in a pre-emptive manner. Non-bound user level threads will execute on the LWPs that are not bound to any thread. There could be many LWPs to serve this kind of user level threads.

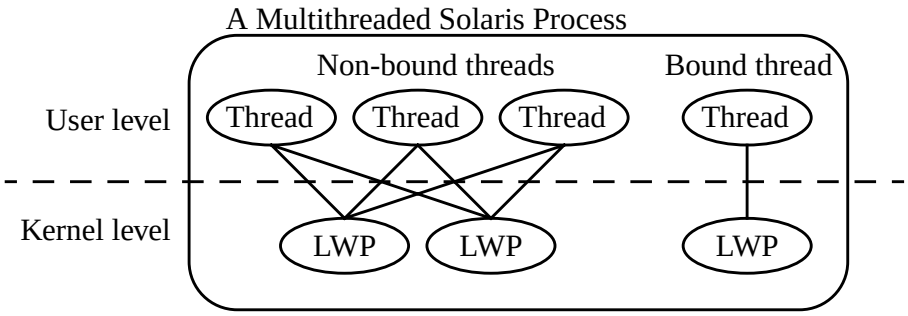

Figure 2: The thread model in Solaris. A bound thread may only execute on the LWP that it is bound to, whereas a non-bound thread may execute on any LWP that is not bound to a user level thread.

Linux [8] uses an single-layered approach, similar to merging the Thread and LWP concepts in Solaris together into one single unit. The creation of threads is made by cloning the parent thread, using the system call clone. This is quite similar to fork, but the cloned threads share the same address space, open files, etc. However, each thread gets its own process id and is scheduled as a process. The scheduling is pre-emptive and time-sliced, as for any other process in Linux. The time-slices are fixed (210 milliseconds). Linux calculates a goodness-value in order to choose between which thread/process to execute next. This goodness-value represents two things, first it represents how long the thread has executed in its current time slice. A thread that has been executed a small part of its time slice will be given a higher goodness value than threads that have executed a longer part of their time slice. The second thing is that threads are favourized to run on the same processor as previously. This is done by increasing the goodness value for the thread when the processor it previously executed on looks for a new thread to execute. Newly awakened threads can force another thread from a processor when the newly awakened thread has a higher goodness value than the other thread.

There are also differences between synchronizations as well. In Linux all locks has ordered queues. When a thread releases the lock, the first thread in the queue is given the lock and put in the running queue. In Solaris, there is no ordered queue (for guaranteed). When a thread releases a lock, it may lock it again, before any other thread (both those waiting in the queue or other) has been able to grab the lock.

4. Validation of the Predicted Execution Times

4.1. The SPLASH-2 benchmark suite

The validation of the predictions was made using a subset of the SPLASH-2 benchmark suite [17]. The applications that we used from the SPLASH-2 suite are listed with the data set in Table 1. All applications that we used are from the scientific and engineering domain.

Table 1: The parallel applications together with the data set sizes we used.

Application	Description	Data set size/Input data
Ocean (contiguous)	Simulate eddy currents in an ocean basin	258-by-258 grid
Water-Spatial	Molecular dynamics simulation, O(N) algorithm	512 molecules, 30 time steps
FFT	1-D $\sqrt{n}$ Six-step Fast Fourier Transform	4M points
Radix	Integer radix sort	16M keys, radix 1024
LU (contiguous)	Blocked LU-decomposition of a dense matrix	768x768 matrix, 16x16 blocks
Raytrace	Producing a raytraced picture	balls4
Barnes	Simulates interaction between a number of bodies in three dimensions	2048 bodies
Cholesky	Factors a sparse matrix into the product of a lower triangular matrix and its transpose	tk29.0
Radiosity	Producing a raytraced picture	Default, batch mode, en 0.1

All executions were made on a Sun Ultra Enterprise 4000 with 8 processors and 512 MByte memory. The operating systems in this validation was Solaris 2.6 and Linux 2.2 (RedHat 6.1). The compiler used for both Solaris and Linux was the egcs-1.1.2 (a.k.a. gcc 2.91.66). Since the SPLASH-2 applications are designed to create one thread per physical processor, one log file

were made for each processor setup when using the Recorder. Thus, 9 applications running each on 4 different CPU setups generated a total of 36 log files. The benchmarks were modified in order to remove spinning locks and task stealing as described in [2].

4.2. Validation results

Table 2 shows the measured and predicted speed-up for the 9 applications from the SPLASH-2 benchmark suit on the Solaris platform. The real speed-up is the middle value of 5 executions of the application. The error is defined as |((Real speed-up) - (Predicted speed-up))/(Real speed-up)|, where $|-x| = |x| = x$, for all $x > 0$.

Due to the recordings, the monitored uni-processor execution takes somewhat longer than an ordinary uni-processor execution of the application. However, our measurements showed that the execution time overhead for doing the recordings was very small. The maximum overhead, which was obtained for Radiosity, was 16.4% of the total execution time, but still more than half of the 36 log files caused less than 0.5% overhead. Another concern was the size of the log files. 75% of the log files were less than 2Mbyte in size. The largest log file, which was obtained for Radiosity, was 20.4 MByte. This file could be handled without any problems. Consequently, neither the execution time overhead, nor the size of the log files caused problems for these applications.

Table 3 shows the measured and predicted speed-up for the 9 applications from the SPLASH-2 benchmark suite on the Linux 2.2 platform. The columns are the same as in Table 2.

The mean error for Solaris was 1.6% while when simulating the Linux platform the mean error was less than 5.8%. Thus, the error on the Linux platform is then on average 3.6 times larger than on the Solaris platform.

4.3. Study of the application with the largest error, Ocean

There are three applications in Table 3 that has large errors in the predictions. These are Ocean, FFT, and Radiosity. Ocean has the largest error and we will focus on that application for this study.

The first thing to notice is the total CPU time needed to execute the application (without the Recorder) behaves quite differently on Solaris and Linux. Since the monitoring is done on Solaris all the overhead, etc., on Solaris is incorporated in the monitoring. Thus, if Solaris behaves differently than Linux, the estimation can be no good. The CPU time needed to execute Ocean is shown as the first row in Table 4. The values are normalized to the case for 1 processor for easy comparison reasons. As can be seen, Solaris increases the needed CPU time with up to twelve percent when executing the 8 threaded Ocean. On the other hand, Linux needs three percent less CPU time to execute Ocean with 8 threads than with one thread. The difference between Solaris and Linux is then 15.5% (1.12 / 0.97).

By increasing the data set for Ocean to 514 and 1026, theses differences decrease as shown in the two last rows in Table 4, to 5.0% and 1.1%, respectively for the case with 8 threads. It is then most likely that the predictions also will be more accurate for Linux as the needed CPU time does not differ. In Table 5 the predictions for Ocean on Linux is shown. As assumed the error in the predictions drops as the data set increases. The measured CPU time can then act as an indicator of the degree of reliability of the prediction.

The measured CPU time may not only act as an indicator, it could also be used to compensate the prediction. In the case of Ocean with data set 258 there is an difference of 15.5% between Solaris and Linux. Thus, the monitored execution on 8 processors was 15.5% longer on Solaris than on Linux and this made the predicted execution for Linux 15.5% longer as well, assuming an even distribution of the overhead over the whole execution. If the predicted execution was 15.5% longer, the predicted speed-up will only be 86.6% (1 / 1.155) compared to the speed-up without the differences in CPU time. Thus, by adding 15.5% more speed-up we are able to compensate for

Table 2: Measured and predicted speed-ups for the benchmark applications on Solaris 2.6.

Application		2 processors	4 processors	8 processors
Ocean	Real Speed-up	1.98	3.67	4.39
	Pred. Speed-up	1.90	3.35	4.35
	Error	4.0%	8.7%	0.9%
Water-spatial	Real Speed-up	1.99	3.93	7.41
	Pred. Speed-up	1.97	3.83	7.24
	Error	1.0%	2.5%	2.3%
FFT	Real Speed-up	1.58	2.22	2.76
	Pred. Speed-up	1.59	2.23	2.80
	Error	0.6%	0.5%	1.4%
Radix	Real Speed-up	1.99	3.96	7.66
	Pred. Speed-up	1.99	3.96	7.79
	Error	0.0%	0.0%	1.7%
LU	Real Speed-up	1.78	3.02	4.45
	Pred. Speed-up	1.78	3.00	4.50
	Error	0.0%	0.7%	1.1%
Raytrace	Real Speed-up	1.88	2.97	4.86
	Pred. Speed-up	1.88	2.94	4.83
	Error	0.0%	1.0%	0.6%
Radiosity	Real Speed-up	1.85	3.60	5.78
	Pred. Speed-up	1.81	3.42	5.89
	Error	2.2%	5.0%	1.9%
Barnes	Real Speed-up	1.94	3.56	6.35
	Pred. Speed-up	1.93	3.57	5.97
	Error	0.5%	0.3%	6.0%
Cholesky	Real Speed-up	1.60	2.30	2.94
	Pred. Speed-up	1.60	2.30	2.90
	Error	0.0%	0.0%	1.4%

the differences in CPU time. The predicted speed-up is 4.35 and with a 15.5% increase it will be 5.02. The latter is much closer the real measured value of 5.77.

The result of calculating in the same way for the other data sets and number of processors is shown in Table 5 within square brackets. The average error in Table 4 has decreased from 7.8% to 5.6% due to this compensation. In three cases the predictions become worse than without compensation, however, those errors are still within the range of errors for the Solaris prediction in Table 2 as well as the errors reported in [2].

Table 3: Measured and predicted speed-ups for the benchmark applications on LINUX 2.2.

Application		2 processors	4 processors	8 processors
Ocean	Real Speed-up	1.99	3.75	5.77
	Pred. Speed-up	1.90	3.35	4.35
	Error	4.5%	10.7%	24.6%
Water-spatial	Real Speed-up	1.99	3.95	7.75
	Pred. Speed-up	1.97	3.83	7.24
	Error	1.0%	3.0%	6.6%
FFT	Real Speed-up	1.71	2.56	3.40
	Pred. Speed-up	1.59	2.23	2.80
	Error	7.0%	12.9%	17.6%
Radix	Real Speed-up	1.98	3.93	7.17
	Pred. Speed-up	1.99	3.96	7.79
	Error	0.5%	0.8%	8.6%
LU	Real Speed-up	1.81	3.11	4.80
	Pred. Speed-up	1.78	3.00	4.50
	Error	1.7%	3.5%	6.2%
Raytrace	Real Speed-up	1.82	2.85	4.41
	Pred. Speed-up	1.88	2.94	4.83
	Error	3.3%	3.2%	9.5%
Radiosity	Real Speed-up	1.85	3.75	5.05
	Pred. Speed-up	1.81	3.42	5.89
	Error	2.2%	8.8%	16.6%
Barnes	Real Speed-up	1.94	3.51	6.03
	Pred. Speed-up	1.93	3.57	5.98
	Error	0.5%	1.7%	0.8%
Cholesky	Real Speed-up	1.60	2.30	2.93
	Pred. Speed-up	1.60	2.30	2.90
	Error	0.0%	0.0%	1.0%

5. Scheduling Differences

When dealing with performance debugging, the speed-up metric does not give all the information needed. If the speed-up is not as the desired speed-up there are some kind of performance bottlenecks in the application. The VPPB system has, as mentioned in Section 2, the ability to show the execution flow as a Gant diagram. This diagram can help the developer understand where the bottlenecks are and how to remove them.

Table 4: Normalized CPU time required to execute the OCEAN benchmark with different data sets on Solaris and Linux.

Data set	Operating System	1 Thread	2 Threads	4 Threads	8 Threads
258	Solaris	1.00	0.99	1.03	1.12
	Linux	1.00	0.96	0.97	0.97
514	Solaris	1.00	1.02	1.04	1.05
	Linux	1.00	1.00	1.01	1.00
1026	Solaris	1.00	1.00	1.00	1.01
	Linux	1.00	1.01	1.00	1.00

Table 5: Ocean on Linux with different data sets. A compensated values are given in square brackets.

Data set		2 Processor	4 Processor	8 Processor
258	Real Speed-up	1.99	3.75	5.77
	Pred. Speed-up	1.90 [1.96]	3.35 [3.56]	4.35 [5.02]
	Error	4.5% [1.5%]	10.7% [5.1%]	24.6% [13.0%]
514	Real Speed-up	1.84	3.67	5.27
	Pred. Speed-up	1.94 [1.98]	3.71 [3.82]	4.51 [4.74]
	Error	5.4% [7.6%]	1.1% [4.1%]	14.4% [10.1%]
1026	Real Speed-up	1.96	3.83	6.93
	Pred. Speed-up	1.90 [1.88]	3.74 [3.74]	6.66 [6.73]
	Error	3.1% [4.1%]	2.3% [2.3%]	3.9% [2.9%]

In Section 4 there is very little difference between the predicted speed-up for the Solaris platform and the Linux platform. Although the speed-up is similar, the execution flow may not be the same. To illustrate that issue Figure 3 shows the same execution segment of the Barnes benchmark. Each horizontal line in the figure represent an executing thread over time. Different symbols indicate different events, e.g., an arrow facing downwards represents a locking operation. Different colours (appear as different gray shades in black and white printing) are used for mutexes, semaphores, etc. Though the execution flow is quite different in Figure 3 the source code is the same. This shows that the execution flow is different when using different operating system, and thus the performance bottlenecks may also differ between the operating systems.

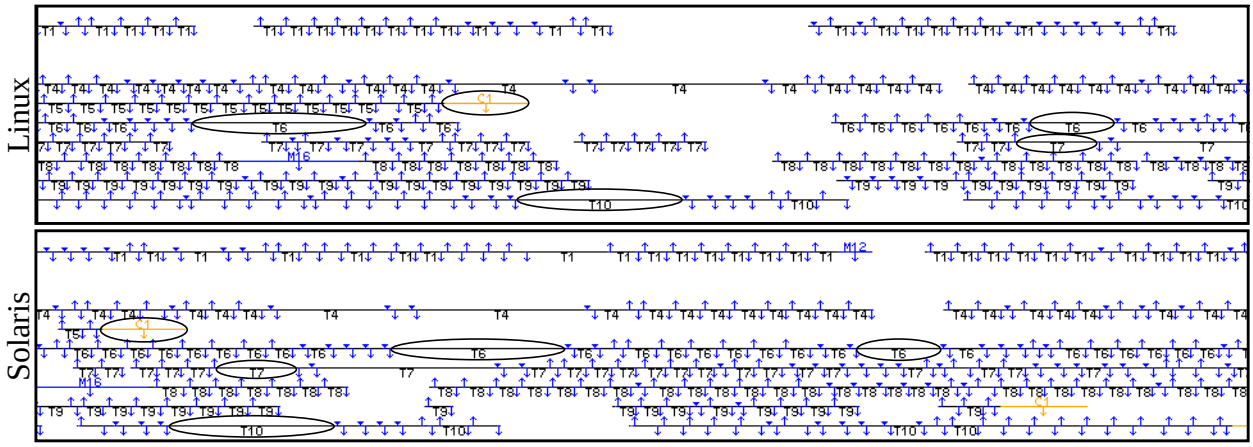

Figure 3: Execution flows for Barnes. Solaris is the upper graph, Linux is the lower. The five ovals in the Solaris graph indicates the same events as the five ovals in the Linux graph.

6. Discussion and Related Work

6.1. Comments on the validation

Validation of the cross-simulation was done on a Sun Enterprise 4000 with 8 CPUs by executing 9 benchmarks from the SPLASH-2 benchmark suite. Only three applications had larger than ten percent error in the predicted speed-up. A further study of the application with the largest error (Ocean) shows that the error is reduced when the data set grows.

As the data set increases the overhead for executing Ocean on Solaris decreases. Since this overhead is included in the monitored data used for the cross-simulation the simulation results for small data sets may not be very accurate. By compensating the predictions with the measured overheads, the error of the predictions was reduced with up to nearly a factor of two.

The reason for the increased usage of CPU time on Solaris, when increasing the number of threads in the Ocean application has not been found. A further study of the reason for this behaviour on Solaris is considered to be future work. Understanding why the error for the benchmarks FFT and Radiosity is almost twice as large as the largest error on Solaris could also be interesting future work.

Currently the VPPB system implements all the POSIX thread primitives that are common between Solaris and Linux except the thread cancelling primitives. The semaphore primitives are supported, although semaphores are not a part of the POSIX threads, but originate from POSIX.1b.

6.2. Other tools

There are a number of tools, shown in Table 6, which make it possible to visualize the (predicted) behaviour of a parallel application using any number of processors. However, these tools are either developed for message passing systems or for non-standard programming environments.

Only a few tools, PARAVER and SIEVE, can do some kind of cross-simulation. The PARAVER tool uses the DIMEMAS simulator [4]. DIMEMAS is a simulator for distributed memory multiprocessors, and the ability to re-configure in order to mimic different operating systems is limited. In [10] there is no validation of the predictions at all. In SIEVE all simulation is performed by scrips working like macros on a spread-sheet, where the spread-sheet is the trace file. By supplying different scripts different operating system can be simulated. The SIEVE system does not address the issue of data monitoring. In [13] there is no validation of the predictions at all.

Table 6: Comparison of some visualization tools similar to VPPB.

Name	Platform	Language	Cross-simulation	Comment	Ref-erence
CHIP^{3}S	Mathematica	CHIP^{3}S	Yes	Pseudo language	[10]
PARAVER	Any PVM3 platform	Any with PVM3	Yes	Message passing	[11]
PERFSIM	TMC CM-5	CM-Fortran	No	Limited visualization	[16]
PIE	VAX 11/780, VAX 11/784, MicroVAX	MP with Pascal	No	Pseudo language	[14]
SIEVE	BBN GP1000	pC++	Yes	Hard to use the scripts	[13]

7. Conclusion

POSIX threads are used to make multithreaded applications portable. However, the implementation of POSIX threads differs for different operating systems. Thus, an application will not always have the same performance and behaviour on different operating systems. Tuning multithreaded applications for a multiprocessor is hard. Tuning applications for good performance on several operating systems is even harder. Most tools use a real execution of the multithreaded application on a given operating system in order to give the developer support in the performance tuning, e.g., [5, 7, 9, 18]. It is often impractical and expensive to have several multiprocessors in order to run different operating systems.

In this paper we have presented a tool, called VPPB, that based on an execution of a multithreaded application on an ordinary single processor Solaris workstation can predict the behaviour of application on a multiprocessor with arbitrary number of processors, running Solaris or Linux 2.2. The predictions are accurate, with an mean error of 5.8% for the predicted speed-ups for Linux, and even better for Solaris. The validation was made by using 9 of the benchmarks from the SPLASH-2 benchmark suite on a Sun Enterprise 4000 with eight processors.

During a detailed study of the Ocean benchmark we have shown that it is possible to find an indicator that shows how reliable the predictions are. The indicator is based on the CPU time required to execute an application with a different number of threads. This indicator works on a single processor workstation with Solaris.

The Ocean benchmark also shown that the indicator can be successfully used to compensate the predictions in order to get more accurate predictions. With Ocean the error in the predictions was reduced with up to almost a factor of two.

Thus, we have shown that it is possible to use the described tool to predict the behaviour of an multithreaded application on a multiprocessor with either Solaris or Linux, with the means of a single processor workstation running Solaris.

8. Acknowledgments

We would like to thank Henrik Persson for rewriting the simulator for the Solaris threads in order to achieve a much faster simulator. This simulator was used as the base when adding ability for POSIX threads as well as support for the Linux platform.

References

[1] M. Broberg, L. Lundberg, and H. Grahn, “VPPB - A Visualization and Performance Prediction Tool for Multithreaded Solaris Programs”, in *Proceedings of the 12th International Parallel Processing Symposium*, Orlando, USA, pp. 770-776, 1998.

[2] M. Broberg, L. Lundberg, and H. Grahn, “Visualization and Performance Prediction of Multithreaded Solaris Programs by Tracing Kernel Threads”, in *Proceedings of the 13th International Parallel Processing Symposium*, San Juan, Puerto Rico, pp. 407-413, 1999.

[3] D. Butenhof, “Programming with POSIX Threads”, Addison-Wesley, 1997, ISBN 0-20-163392-2.

[4] S. Girona, T. Cortes, J. Labarta, V. Pillet, A. Peres, and E. Lopez, “Deriverable OPS4A of the project Basic research APPARC, Effect of short term scheduling on message passing multiprogrammed systems”, http://www.wi.leidenuniv.nl/CS/HPC/apparc-deliverables/OpS4a.html, 1994.

[5] M. Heath and J. Etheridge, “Visualizing the Performance of Parallel Programs,” *IEEE Software 8(9)*, pp. 29-39, 1991.

[6] S. Khanna, M. Sebrée, and J. Zolnowsky, “Realtime Scheduling in SunOS 5.0,” in *Proceedings of the Winter ‘92 USENIX*, June 1992.

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

[8] S. Maxwell, “Linux Core Kernel Commentary,” Coriolis Open Press, 1999, ISBN 1-57610-469-9.

[9] B. Miller, M. Callaghan, J. Cargille, J. Hollingsworth, R. Irvin, K. Karavanic, K. Kunchithapadam, and T. Newhall, “The Paradyn Parallel Performance Measuring Tools,” *IEEE Computer 28, vol 28, no. 11*, pp. 37-46, 1995.

[10] 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,” in *Proceedings of 8th ISCA International Conference on Parallel and Distributed Computing Systems*, Florida, USA, pp. 527-533, 1995.

[11] 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, February 1995.

[12] 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., September 1991.

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

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

[15] SunSoft, “Solaris Multithreaded Programming Guide,” Prentice Hall, 1995, ISBN 0-13-160896-7.

[16] S. Toledo, “PERFSIM: A Tool for Automatic Performance Analysis of Data-Parallel Fortran Programs,” in *Proceedings of the 5th Symposium on the Frontiers of Massively Parallel Computation*, McLean, Virginia, IEEE Computer Society Press, February 1995.

[17] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta, “The SPLASH-2 Programs: Characterization and Methodological Considerations,” in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp. 24-36, June 22-24, 1995.

[18] Q. Zhao and J. Stasko, “Visualizing the Execution of Thread-based Parallel Programs,” *Graphics, Visualization, and Usability Centre, Georgia Institute of Technology, Technical Report GIT-GVU-95-01*, 1995.