MODELING AND VISUALIZING KERNEL ACTIVITY IN A SHARED MEMORY MULTIPROCESSOR

J. Opsommer, W. Van de Velde, E. H. D'Hollander *
University of Ghent,
Department of Electrical Engineering,
St.-Pietersnieuwstraat 41,
B-9000 Ghent, Belgium

Abstract

A graphical environment is presented to visualize the kernel activity in a shared memory multiprocessor. An existing thread scheduler was modelled and simulated to study the behaviour of parallel thread execution. The model reveals the possible bottlenecks of the system and allows to optimize several thread scheduling alternatives.

Using the MODLINE programming environment, it is possible to obtain an animated execution showing the evolution of thread creation, scheduling and execution. The simulation was compared with a kernel executing on a shared memory multiprocessor.

In addition a post processor was developed to visualize the task execution on each processor and the shared resource accesses. In the corresponding Gantt charts, task dependencies are graphically represented.

1 Introduction

One of the key performance factors in a multiprocessor system is the scheduling and allocation of parallel tasks. However it is difficult to accurately monitor the low level kernel activity, because the measurement probes and associated instrumentation blur the performance picture.

To visualize the interaction of processors and processes in a shared memory multiprocessor, a graphical discrete event simulation environment, MODLINE, has been used [5]. Using animation the experimenter is able to observe the evolution of the parallel executing tasks and to zoom in on the hot spots when processors compete for shared resources.

We extended the MODLINE package with a post processor to visualize specific multiprocessing artefacts. The post processor is also able to show the history of the events traced with a non-intrusive timing analyzer during the execution of a real multiprocessor.

The system is used to simulate and validate the operating system behaviour of the VPS multiprocessor prototype [6]. Each processor board of the VPS locally executes a thread handling kernel, implementing system primitives of the μ-kernel [3].

The next section describes the models that were developed to simulate several alternative methods for scheduling threads on multiprocessor systems. The tools for visualizing and animating the kernel activity are discussed in section 3. Section 4 discusses the results and in section 5 the concluding remarks are given.

2 Multiprocessing Kernel Modeling

Thread Management

The scheduling policy of a multiprocessing operating system kernel significantly affects the execution of parallel tasks, or threads [8]. Often a fine task granularity is preferred because this enhances the parallelism. However, it is clear that more communication and scheduling activity will occur in taskgraphs with small-sized tasks [9]. To study the behaviour of different thread management alternatives, a detailed queueing model of a thread scheduling kernel was developed.

Different thread schedulers have been simulated. These schedulers vary in the way they handle the list of executable tasks, known as the *ready list*. The ready list varies dynamically as new tasks are *emitted* and ready tasks are executed. A task becomes ready when its predecessor tasks have terminated.

Basic Multiprocessor Kernel Model

In its simplest form, the ready list is stored as a single list in the shared memory. The model of a mul-

*This research is sponsored by the contracts OOA 87/93-117 and IT/IF/8 of the Belgian Ministry of Science

tiprocessor kernel executing tasks from a global ready list is represented in figure 1.

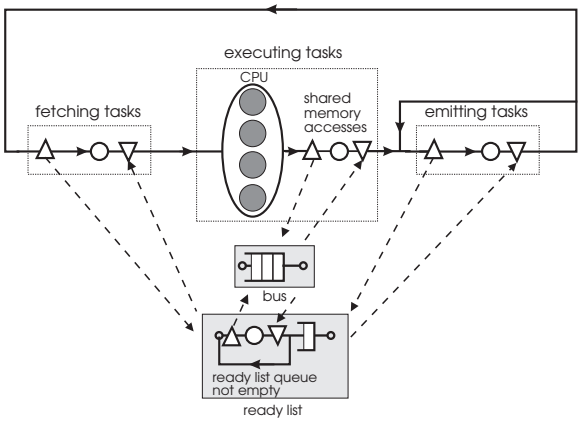

Figure 1: Basic thread scheduler model: the task is fetched from the global ready list and executed on the processor; ready successor tasks are emitted to the global ready list.

Each task has a shared memory semaphore representing the number of uncompleted predecessor tasks. In the first scheduling model, the execution of a task is performed in the following steps:

1. A processor fetches the task from the global ready list. The ready list is a shared resource, protected by a lock. If the list is locked by another processor, the processor keeps polling the lock until the list becomes accessible. An *idle* processor periodically tries to read the ready list using increasing time intervals when no ready tasks are available. This spin-waiting is similar to the Ethernet Backoff Algorithm [2,7] and reduces the bus traffic if there is not enough parallelism in the program.
2. The processor executes the task locally, using the bus for accessing shared data.
3. The semaphores of the successor tasks are decremented and if a semaphore becomes zero, the corresponding task is emitted on the ready list. Emitting a task involves the same resource accesses as in step 1.

This model has two potential bottlenecks: the ready list and the bus. Furthermore, the polling on the lock of the ready list has an avalanche effect on the bus usage.

Thread Scheduling Alternatives

The various thread scheduling techniques differ in the way the ready list is structured and accessed. The list can be stored in global or in local memory and the access to the list is private or shared (table 1). If the processors have enough local memory, it is never advantageous to store private lists in shared memory; so this possibility is not considered here (e.i. global lists are always shared).

	Local	Global
Private	LM, no lock	N/A
Shared	DPM, locked	SM, locked

Table 1: Overview of the different combinations to store and access ready lists. LM = local memory; DPM = dual-port memory, SM = shared memory

Local Private List. To reduce ready list contention, processors keep one or more ready tasks locally in a private list. A fetch from the shared list is only needed when the local buffer is empty; the number of tasks that is then transferred from the shared to the private list during the same lock operation is a configuration parameter. The local taskbuffer is especially useful for parallel chains of sequential tasks where each task triggers one successor task. If the local buffer is too large, the workload balance suffers, because a processor cannot fetch tasks from its neighbour processor's private local list.

Multiple Global Ready Lists. Another way to alleviate the contention is to provide multiple shared lists, each protected by a separate lock. Initially a processor selects an arbitrary list and skips to the next list when the previous one is locked. The first free list becomes the preferential list for the next accesses as long as it remains unlocked and contains ready tasks.

Local Shared Ready Lists. A third approach uses dual-port memories. This memory allows a processor to access the local memory of another processor via the shared bus. Now each processor only has a local ready list with a lock, allowing a processor with an empty ready list to fetch tasks from the list of another processor. This method reduces ready list contention and shared memory accesses, but an excessive dual-port traffic reduces the performance if the program is not well-balanced.

Both private and shared local and multiple global ready lists are scalable solutions to the contention for common data structures. Equally promising are scheduling strategies based on combinations of the

previous methods, e.g. a small private list for each processor together with multiple global lists in shared memory.

3 Visualization Tools

Graphical Simulation Environment

The different scheduling strategies were implemented using the MODLINE graphical programming environment [4]. This includes a graphical editor QNET for entering and animating queueing networks. The design is converted to QNAP2 code, a modeling language for the simulation or analytical solution of queueing networks [5]. The system provides a library to support a statistical analysis of the obtained results. An experimenter tool enables the user to specify interactively the parameters for a number of simulation runs. Reports are generated automatically and include a graphical representation of the model, the QNAP2 code, the input parameters and the simulation results.

The kernel activities are modelled by closed queueing networks where the CPU, the bus and the different kinds of ready list structures are rather unconventionally represented as *resources* instead of service stations [1]. This allows to model these resources as *independent entities*. Each resource has a waiting queue. However, an executing task may decide not to join a locked queue; e.g. when a ready list is locked, the kernel may try to access another list. Since the private lists are not shared, they are not implemented as resources but rather as internal structures of the fetch and emit server.

The QNET model of the kernel is represented in figure 2. The task execution is modelled as a sequence of stages (fetch, exec, emit), executed by different servers. The servers reflect the phases of scheduling (*fetch*), execution (*exec*) and synchronization (*emit*) and they require particular resources to accomplish their service. The model allows the user to compose the kernel list data structures from a palette of local, global; private and shared ready lists (cfr. section 2). If different policies are active, the fetch and emit operations first try to use private lists, then local shared lists and finally global lists.

The input for a simulation run consists of the following:

- Hardware parameters: during the edit session, it is possible to change the low-level characteristics of the simulated architecture. The parameters are the time to lock and access private and shared ready lists and the time to adjust semaphores and to access shared data.

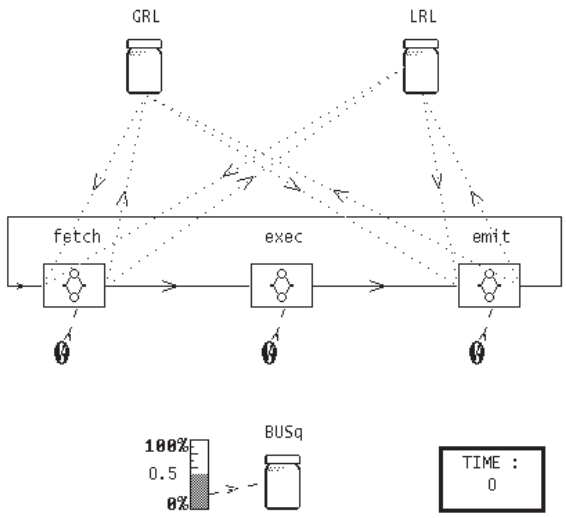

Figure 2: QNET representation of the kernel model: the subsequent stages in the execution of a task are modelled as servers (fetch, exec, emit), competing for shared resources (bus, global ready lists GRL, local shared lists LRL).

- Kernel configuration: for each run, the user specifies the number of processors and the type of ready lists to use. It is possible to limit the size of the private lists and to specify how many tasks are transferred from a shared to a private list during one lock operation. Also the different *backoff* parameters for the spin-wait of idle processors are defined as part of the kernel configuration. Several kernel configurations can be selected in a single simulation experiment.

- Program to simulate: the parallel program can be specified in two ways:

  1. using a *complete description of the task-graph*, containing all dependencies between tasks and an estimated length of each task
  2. using *average task characteristics* such as the average number of successors per task, the shared memory data communication per task and the average task execution time on a single processor. Although less accurate for studying a particular program, this method is useful for studying the effect of general program characteristics on the performance of a parallel execution.

Animation

During the animation session, a task residing at a particular server is represented by a large bullet (fig. 3). Initially, the resources are filled with tokens (small bullets). The number of tokens equals the number of available resources; e.g. in the simplest model there is only one global ready list token. Accessing a resource requires a resource token. Servers compete for the shared resources; if a token is not available, a server is blocked temporarily. When a service is completed, the task moves from one server to the next one.

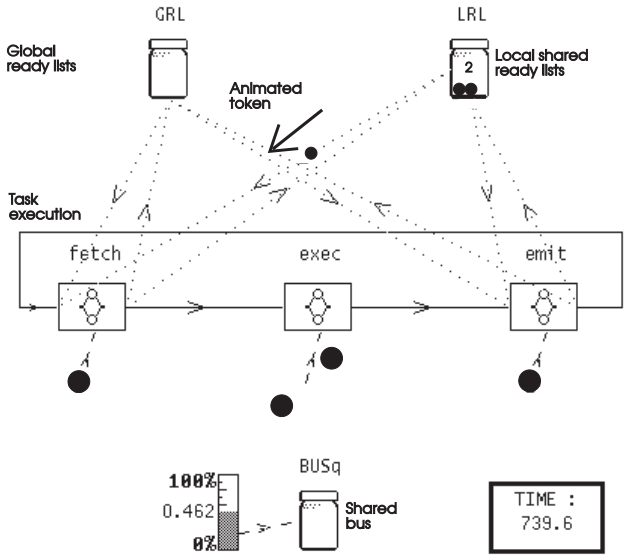

Figure 3: Snapshot of animated kernel visualization: tasks are represented by large bullets, tokens by small ones. The task in the fetch server is taking a local ready list token from the LRL resource

Using measurement icons, it is possible to indicate the average access fraction, the number of free resources, etc. These statistical variables are visualised on a scale and updated during the animation session. From the animation trace, a time plot of a selection of statistical variables can be generated.

Graphical Post Processor

A graphical post processor was developed to display the events recorded during the simulation run. The graphical post processor makes it possible to experiment with various scheduling alternatives and to obtain meaningful information about otherwise unobservable kernel statistics. Among these are Gantt charts of each processor, displaying task execution, shared resource accesses, idle times, bus locks and contention delays. Furthermore predecessor and successor links are indicated on the time charts and in a separate window statistical measures such as the speed-up, memory communication and synchronization times, task granularity and bus occupation are displayed.

The visualization system is also able to display a limited trace of a real execution on a coarse time scale using data recorded with a timing analyzer. This allows an easy comparison between the simulation and the execution profiles.

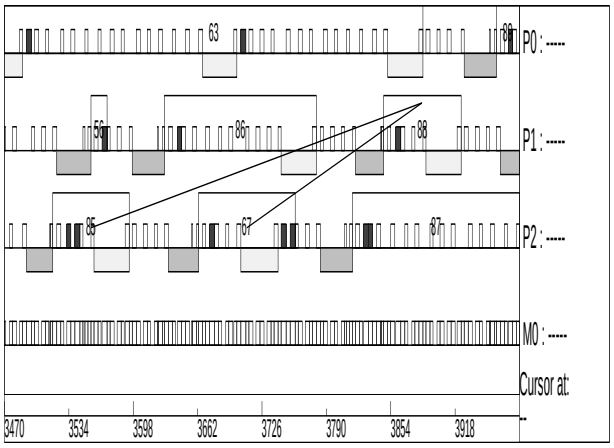

Figure 4: Screen output of graphical post execution analysis: the large numbered blocks represent tasks with the corresponding fetch and emit operations (dark and light gray blocks under the baseline), the bus accesses (small white blocks) and the semaphore updates (small black blocks). The Gantt chart shows a zoom of the execution on three processors (P0, P1, P2), using shared memory M0. The lines indicate the predecessors of task 88.

4 Results

Model Validation

The VPS multiprocessor system is observed non-intrusively by a timing analyzer. The access to critical kernel data structures can be clocked by triggering the correct addresses. In this way, accurate machine time parameters for the queuing models were obtained. These are a shared memory data word read/write, a semaphore synchronizing operation, the locking of a ready list, a ready list task fetch and a task emit.

Table 2 contains the list manipulation times in the absence of resource contention (i.e. measured on a single processor) for a local private list, a global shared list and a local shared list. The local shared list has two entries: a local access (processor uses its own list) and a dual-port access (processor uses list of another processor). Using these measured characteristics, the simulated execution time of a number of programs has been compared with the real execution times. The average deviation of all programs and kernels is limited to 3.5%, with a maximum of 11.7% [10].

Ready list type	lock	fetch/emit		
		cpu	bus	total
Local private	N/A	10.	0.0	10.0
Global shared	4.0	20.	3.0	23.0
Local shared:				
local access	1.0	15.	0.0	15.0
dual-port access	4.0	25.	3.0	28.0

Table 2: Times to lock and access (average times of fetch and emit operation) the different kernel data structures . The time a list is locked during a fetch and emit is partitioned in local processing ("cpu") and shared access time ("bus"). Note that the local shared list has an entry for a local and for a dual-port access. Times are in μs.

Impact of Scheduling Alternatives

The execution times and the mean shared bus occupation for a typical program execution on 4 processors are shown in figure 5. The different kernel configurations are ordered in decreasing execution time. All alternative schedulers are faster than the basic thread scheduler and the local shared lists are more efficient than the global shared lists.

Adding a private buffer is advantageous both for the singly and the multiply global list. Only the local shared list kernel has no execution time benefit of a private buffer; but using the buffer reduces the bus occupation from 57% to 45% which is important for the scalability of this kernel.

The private list utilization PL_u can be defined as:

$$PL\_u = \frac{\#\ tasks\ fetched\ from\ private\ list}{\#\ private\ and\ shared\ fetches}$$

The influence of the private list length on the performance and on the private list utilization is represented in table 3 for a kernel with one global ready list. Introducing a private list with length 1 doubles the execution speed and reduces the bus occupation

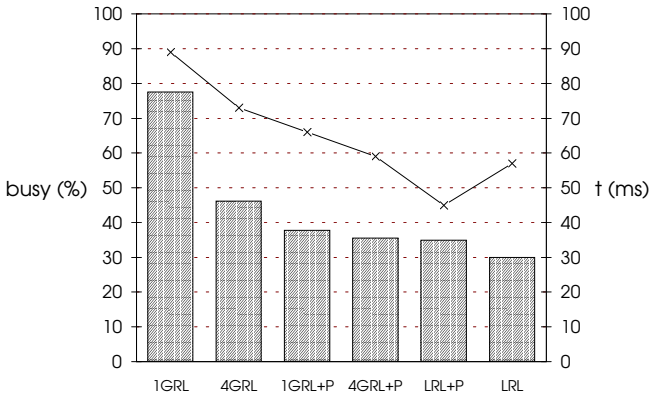

Figure 5: Execution time t in ms (bar graph) of a Gauss-Jordan 10 $\times$ 10 linear system solver (GJ10) on a selection of kernels: 1GRL, 4GRL = kernel with one and 4 global ready lists; LRL= kernel with local shared lists, P = private lists. The lines indicates the mean bus occupation.

significantly. With a larger size the utilization further increases but the impact on execution speed and bus occupation is less remarkable. Due to load inbalance, the execution speed drops with a list of length 5. The optimal length of the private list depends on the parallelism in the program.

length	t (ms)	bus occupation	PL_u
0	77.6	89.0%	0.0
1	36.9	66.0%	61.0
2	34.9	63.3%	67.6
3	32.8	60.8%	74.0
4	30.4	59.4%	80.4
5	33.7	53.5%	82.1

Table 3: Execution time t, bus occupation and private list utilization PL_u for a kernel with one global list executing the GJ10 program on 4 processors.

Figure 6 presents the evolution of the number of ready tasks on the local shared list of a processor; the other processors show a similar curve. The simulated program is GJ10 and the interesting spots are indicated on the figure:

(a) here the outer loop of the program generates a lot of successor tasks

(b) waiting processors succeed in getting three tasks from the list (this processor is the only one emitting tasks)

(c) the inner do loops are started, fetching ready iterations

(d) the local list is empty and the processor has to fetch tasks from other processors.

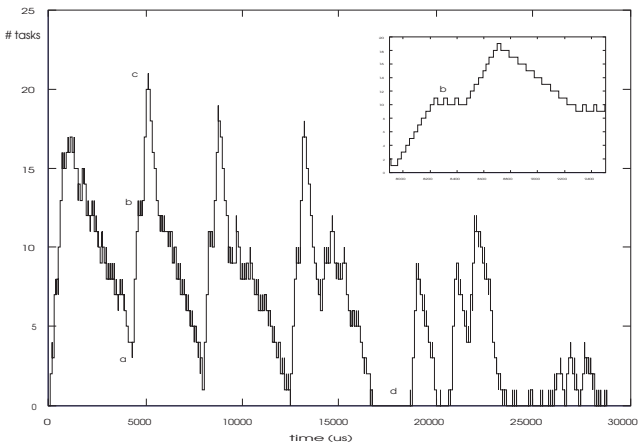

Figure 6: Number of ready tasks on local shared list of one processor, executing GJ10 on 4 processors; the inset shows a detail of artefact (b).

5 Conclusion

The main advantages of the graphical modeling environment are:

- the simulation with the animation tool offers an effective way to understand the inner working of an executing multiprocessor,
- the X-Window oriented user interface presents different simultaneous views of events, times and other characteristics,
- using the discrete event modeling system, the building blocks of the architecture are described as independent resource units, which allows a *resource* driven description of the environment,
- the model developer is assisted by a graphical programming interface.

The modeling approach permits to explain the behaviour of the scheduler in terms of program and machine characteristics. The model allowed to compare the scheduling policies, pin-point the bottlenecks in the algorithm and the shared data structures, and improve the scheduling discipline significantly. The improvements were also verified by implementing them on a real multiprocessor prototype.

References

[1] Akyildiz I.F., *On the Exact and Approximate Throughput Analysis of Closed Queueing Networks with Blocking,* IEEE Trans. on Software Engineering, Vol. 14, 1, pp. 62-70, 1988.

[2] Anderson T.E., Lazowska E.D., Levy H.M., *The Performance Implications of Thread Management Alternatives for Shared-Memory Multiprocessors,* IEEE Trans. on Computers, Vol. 38, 12, pp. 1631-1644, 1989.

[3] Buhr P.A., Strooboscher R.A., *μSystem. Providing light-weight concurrency on shared-memory multiprocessor computers running UNIX,* Software – Practice and Experience, Vol. 20, 9, pp. 929-964, 1990.

[4] Bull/INRIA, *MODLINE Users Guide,* Simulog, 1992.

[5] Bull/INRIA, *QNAP2 Users Guide,* Simulog, pp. 315, 1992.

[6] D'Hollander E.H., *The VPS, a Virtual MIMD Processor and its Software Environment,* Workshop on Compiling Techniques and Compiler Construction, for Parallel Computers, Keble College, Oxford UK, 13-15 September 1989., pp. 19-36, 1989.

[7] Gupta A., Tucker A., Urushibara S., *The Impact of Operating System Scheduling Policies and Synchronization Methods on the Performance of Parallel Applications,* ACM, pp. 120-132, 1991.

[8] Lehr T., Black D., Segall Z., Vrsalovic D., *Visualizing Context-Switches Using PIE and the Mach Kernel Monitor,* Intl. Conf. on Parallel Processing, II - Software, pp. 298-299, 1990

[9] Opsommer J., *A Taskgraph Clustering Algorithm based on an Attraction Metric between tasks,* Proc. Computer Systems and Software Engineering, IEEE COMPEURO92, pp. 77-82, 1992

[10] Van de Velde W., Opsommer J., D'Hollander E.H., *Performance Modeling of Micro-kernel Thread Schedulers for Shared Memory Multiprocessors,* Proc. of the PARLE93 conference, June, pp. 736-739, 1993