Uppsala Master's Theses in Computing Science 113
Examensarbete DV3
1998-01-15
ISSN 1100-1836

SICStus MT – A Multithreaded Execution Environment for SICStus Prolog

Jesper Eskilson

January 15, 1998

Computing Science Department, Uppsala University
Box 311, 751 05 Uppsala, Sweden

This work has been carried out at

Swedish Institute of Computer Science
Box 1263, 164 29 Kista, Sweden

Abstract

We have designed and implemented a multithreaded execution environment for SICStus Prolog. The threads are dynamically managed using a small and compact set of Prolog primitives and they are implemented completely on user-level, requiring almost no support from the underlying operating system.

The development of intelligent software agents has been one of the reasons why explicit concurrency has become a necessity in a modern Prolog system today. Such an application needs to perform several tasks which may be very different with respect to how they are implemented in Prolog. Performing these tasks simultaneously is very tedious without language support.

Keywords: Logic Programming, Threads, Message-passing, Concurrency

Supervisor: Stefan Andersson

Examiner: Mats Carlsson

Passed:

Acknowledgements

I would like to thank my supervisor Stefan Andersson who has patiently answered my questions about SICStus Prolog, my examiner Mats Carlsson for asking the right questions to me and having many valuable opinions about the implementation and on the report. Thanks also to Sverker Janson and to my friend and colleague Fredrik Larsson for many good ideas and fruitful discussions.

Special thanks to my wife Jenny for her support and endless love.

Table Of Contents

Tables & Figures

Chapter 1 - Introduction

The nature of *Logic Programming* in general and *Prolog* in particular, is that of proving a statement given a set of axioms and a set of rules. The axioms and the rules form what is called the *database*, to which queries are made. For example, assume that we have a database containing information on how to get from point A to point B. The query

```
| ?- route(kremlin, whitehouse, X).
```

might then give us the answer

```
X = [kremlin,sheremetyevo-2,heathrow,jfk,whitehouse] ?
yes
```

after searching for the fastest way of getting from the Kremlin to the White House.

This paradigm of modeling computation by making queries to a database has turned out to be very expressive in modeling a large number of problems, especially where some form of search is involved. However, larger software systems often contains several independent sub-programs which continuously interact with their environment using some kind of loop which accepts input and then acts on it. For example, a WWW-server normally consists of a part which continuously listens for connections on a socket and a word-processor with on-the-fly spell-check continuously scans the spelling dictionary for the words which are typed in. Modeling these kinds of programs using the traditional query-answering mechanism becomes inconvenient and inefficient since the only way in which we can switch from executing one sub-program to another is to interrupt the execution of the query, store away the entire computational state, and return to the top-level loop, allowing another sub-program to execute, and later restore the computational state and resume the query.

In other words, the nature of Prolog has historically been centered around the concept of having a s*ingle thread of control*, a concept which is insufficient in large software systems (regardless of the language used). This is where *multithreading* comes in.

1.1 Background

1.1.1 Concurrency

Let us first take a little broader view of the subject. The generalization of multithreading is called *concurrency*, which (in this context) refers to the simultaneous execution of smaller or larger parts of one or more computer programs [2]. Concurrency can be divided into:

1. *Instruction level* where two or more (assembler) instructions are executed simultaneously.
2. *Statement level* where two or more statements (a group of instructions) are executed simultaneously.
3. *Unit level* where two or more subprograms (functions, methods, predicates, depending on the paradigm) are executed simultaneously.
4. *Program level* where two or more programs are executed simultaneously.

Instruction level and program level concurrency require no language support but are instead supported at hardware and operating system level, respectively. Statement level concurrency is also referred to as *data-flow concurrency*, since the flow of control is governed by the availability of computed data values. Instead of executing statements sequentially, they are executed as soon as all of their input values are computed. Multithreading comes in at Unit level concurrency; it is the execution of two or more subprograms simultaneously. There is a variant of unit level concurrency called *co-routining* where program units called *co-routines* can cooperate to intertwine their sequence of execution. This type of concurrency provided by co-routining is called *quasi-concurrency* since only one co-routine can execute at one given time (even when multiple processors are present). *Physical concurrency* is when several subprograms are literally executing simultaneously. This requires that multiple processors be available, which is often not the case. A relaxed variant of physical concurrency is called *logical concurrency*, which *appears* to the user as physical concurrency. This is done by interleaving the execution of the different program units on the same processor. [2]

The central concept throughout this thesis is the *thread of control*. It is defined in [2] as a "sequence of program points reached as control flows through the program". This concept together with *logical*, *unit level* concurrency make up what is hereafter called *threads*.

1.1.2 History

In the late 1970s, when UNIX was young, digital watches high-tech and window-systems yet to be invented, there was little or no support for concurrency at language level (statement and unit level). However, there were languages, such as Concurrent Pascal and InterLisp ([3], [4]), with co-routining, but they were both relatively small, experimental languages with small industrial impact.

The only support for logical and/or physical concurrency that existed was at program level, in UNIX implemented by *processes*. This meant that a separate process had to be created for each individual activity which should be performed concurrently: daemons, user applications, printer spoolers, batch jobs, etc. Even if this was a large step forward from having no concurrency at all—not even at program level—it soon became obvious that this was inadequate. Concurrency was needed below program level; there was a need to execute *parts* of a program in parallel, not only whole programs or applications. In distributed systems, for example, servers were often found to be bottlenecks since they were unable to serve multiple clients at a time, resulting in long response times and irritated users. Also, the emergence of MIMD [5] architectures made it possible to exploit true (physical) concurrency to solve problems with inherent parallelism in them. In order to do this in a simple way, there had to be support for unit-level concurrency in the language (spawning off new tasks, critical regions, etc.).

Now, why was it not possible to use normal processes? The main problem with processes was (and still is) that they are too heavyweight. As the reader might be aware of, a process is a completely isolated unit with its own execution environment (signal dispatch tables, memory mapping tables, file descriptors, etc.). Creating a new process means that a complete execution environment needs to be created from scratch (or copied from an existing one), an operation which takes a considerable amount of time. The unsuitability of processes for unit-level concurrency becomes even more evident on NUMA (Non-Uniform Memory Access) [6] architectures where the difference in access speed between local and non-local memory is very large. A process-switch on such a machine will result in a address space change which in turn will cause expensive cache and TLB misses [7].

In AND/OR-parallel Prolog systems [33], the overhead of creating new processes has been eliminated by having a static (fixed) set of processes. The available work is distributed dynamically among these processes by a scheduler. However, static process creation only eliminates the overhead of creating new processes; the overhead of process-switching still remains.

It was realized that in order to implement low-overhead unit level concurrency, only those parts of the execution environment directly related to code execution (such as the program counter and the execution stack, for example), needed to be created for each concurrent subpro-

gram. The rest of the execution environment could be shared. Using these ideas it was possible to implement unit level concurrency without the overhead that came with using processes. Andersson et al. [8] mentions a factor 10 for the difference in overhead for creating a new thread and creating a new process. This number was obtained by the *Null Fork*-benchmark which creates a thread / process whose only task is to invoke the empty procedure.

During the late 1980s and early 1990s, support for unit level concurrency became publicly available in the form of *thread packages* of different kinds, and in 1995 every major operating system had integrated support for threads [4].

1.2 About This Thesis

The purpose of this thesis is to show the feasibility of incorporating support for multithreading in SICStus Prolog [9, 10]. The thesis describes the design and implementation of a prototype multithreaded execution environment. The working title has been *SICStus MT*, which will be used throughout this report. The design should be general enough to work on most platforms supported by SICStus and efficient enough—both with respect to memory consumption and execution speed—so that the usage of threads is not discouraged if and where it is appropriate.

1.3 Organization

Chapter 2 contains a discussion on the overall design issues. Chapter 3 describes the programming interface and the semantics of the predicates involved. The problem of blocking system calls is described in Chapter 4. Chapter 5 contains a discussion issues related to efficiency in terms of memory consumption and execution speed. Chapter 5 also contains a comparison between SICStus MT and other multithreaded languages. Chapter 6 describes future work; improvements and extensions to the implementation. Chapter 7 describes some related work, and Chapter 8 contains a short conclusion of the thesis. Chapter 9 contains the source code for the two benchmarks.

Chapter 2 - The Design of A Multithreaded Environment

This chapter will describe the design of a multithreaded execution environment for Prolog in general. Knowledge of the WAM [11, 12]—the abstract machine on which SICStus executes Prolog code—is not required, although a general knowledge on how abstract machines work can be helpful.

2.1 Multithreaded Execution and Virtual Machines

It is important to realize that the concept of a thread of execution is tightly connected to the concept of a machine executing a program. The machine is usually a *physical* device (a microprocessor, for example), but it can also be *virtual* and only exist in terms of a specification of the instructions it can execute (such as the WAM or the JVM). In the absence of appropriate hardware, virtual machines are emulated in software. The emulator-program is for efficiency usually written in assembler, C or another low-level language and executes directly (i.e. not emulated) on a physical device. This concept of using several (different) layers of interpretation/execution is generalized in [43].

Consider the scenario present in SICStus. We have a Prolog program which executes on the WAM which in turn is emulated by a program which executes on a CPU. There are two programs present here—the Prolog program and the emulator program—and therefore we have two threads of execution. One executing the Prolog program and one executing the emulator.

In the light of this, we introduce the concept of *Prolog* threads and *native* threads. *Native* threads refer to threads of execution on the level of the emulator program. *Prolog* threads refer to threads of execution on the level of the Prolog program. Naturally, this work is focused around Prolog threads. The incorporation of native threads into SICStus Prolog is discussed in Section 2.2.

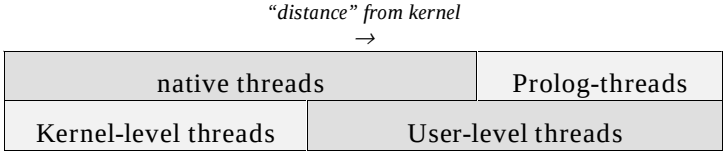

Figure 1: Relationship between different kinds of threads

In the same way as it is important to distinguish between native threads and Prolog threads, it is important to distinguish *Kernel-level* threads and *User-level* threads. The difference is simple and intuitive; kernel-level threads are threads which are scheduled by the operating system kernel while user-level threads are managed and scheduled without any knowledge of the kernel. This does not, however, mean that they are completely separate from the operating system, only from the *kernel*. They could, for example, be implemented in a user-level system library.

The relationship between native threads and Prolog threads on one side and Kernel-level threads and User-level threads on the other is shown in Figure 1. In the figure we can see that the definitions overlap; native threads can be either Kernel-level or User-level, and User-level threads can be either native threads or Prolog threads.

2.2 Should Native Threads Be Used?

One of the first questions, and undoubtedly the one that influenced the overall design most, was the question of whether native threads should be incorporated and if so, how do we map Prolog threads onto native threads?. There are several possible alternatives and they are not mutually exclusive.

1. Map each Prolog thread onto an native thread. This means that for every new Prolog thread created, a new native thread is created with the emulator as entry point, executing the Prolog code for the new thread. This would result in the scenario where we have an instance of the emulator running for each thread.
2. Introduce a new kind of thread, an *Native Prolog* thread. This means that, we allow the user to explicitly create Prolog threads which are mapped directly onto native threads, alongside with creating "normal" Prolog threads. This could, for example, be achieved by using two different predicates for spawning threads.
3. Allow the emulator to make its own choice on mapping Prolog threads onto native threads, possibly guided by some kind of user preference.
4. Do not use native threads at all.

We have chosen to use the last alternative, for a variety of reasons. First, even if there are fairly portable packages implementing native threads (POSIX, for example), they are basically non-portable since they do not behave in the same way on all platforms. Solaris threads are quite different from Windows NT threads which in turn are different from OS/2 threads. This is a major drawback. Second, by using native threads we lose control over scheduling algorithms. If the underlying package does not support preemptive scheduling (see Section 2.5.1), Prolog threads will not become preemptive and if the underlying package does not prevent starvation, there will be Prolog threads queuing for charity and we will stand helpless. Third, due to implementation details in SICStus, using native threads would mean rewriting large parts of code which assumes that there is a global reference to the current set of machine-registers and in order to fix this *without* rewriting code, one would need to hook the native thread scheduling mechanism so that it keeps the global reference updated each time a new thread is scheduled. this would cause even more non-portability. Fourth, even if native threads are very useful in order to utilize underlying machine-specific features (such as multiple processors, specialized scheduling algorithms), they are not essential in demonstrating the usefulness of Prolog threads. Of course, it is possible to incorporate some of the ideas of utilizing native threads as described above, but that is outside the scope of this thesis.

2.3 Storage Model

There are five kinds of data-areas in the SICStus emulator:

The Static Area contains a variety of objects, such as interpreted and compiled clauses, atoms, indexing tables, and so on. It expands and shrinks by calls to dynamic memory allocation functions à la `malloc()`, `realloc()` and `free()`.

The Local Stack is also called *environment stack* and contains procedure frames, which mainly consist of permanent variables (variables which survive predicate calls). It expands on predicate calls and shrinks on determinate returns and on backtracking. This includes the situation when exceptions are thrown, since exception handling is implemented in terms of "controlled backtracking".

The Global Stack contains Prolog terms. The term “stack” is a bit confusing, since it is not a strict LIFO-structure. It expands when terms are built and contracts by garbage collection and on backtracking. “Heap” is a more appropriate term.

The Choicepoint Stack contains choicepoints consisting of the machine and argument registers of the WAM. It expands whenever a non-deterministic predicate is called, and shrinks either when the last clause of the predicate has been tried, a call to `!` (cut) is made, or if an exception is thrown.

The Trail Stack contains conditional variable bindings, i.e. variables which should be reset to unbound upon backtracking. It expands during variable-binding in non-deterministic predicates and shrinks similarly to the choicepoint stack with the addition that it also shrinks by garbage collection. This is due to the fact that cuts cause garbage to be left on the trail stack.

In addition to this, we have the set of abstract machine registers organized as a data structure `struct worker`, or *WS* for *W*orker *S*tructure, which contains program counters, stack boundaries, choicepoint-information, etc.

In SICStus MT, the structure of this storage model needs to be modified. More precisely: some areas must be kept private to each thread. Fortunately, this matter is solved quite easily, under the assumption that we wish to keep the thread as light-weight as possible.

The bulk of the static area is kept global. There are some minor parts of the static area that might be better off being thread-specific—such as execution statistics, for example—but they do not affect the overall design or the implementation, so they have for the time being been kept in the static area. The local, choicepoint, and trail stack must be kept private, since they are directly related to how the program is executed. The same thing goes for the abstract machine registers, the WS. The WS is combined with thread-related information (such as status-flags, thread ID, message-port, etc.) to form a structure of type `Thread`.

The global stack has to be kept private to each thread. Even if it would be attractive to employ a shared global heap to be able to share data between threads (such as in Oz 2.0), this is not really a feasible solution. The reason is that since Prolog is a backtracking language (Oz 2.0 is not), a shared heap in multithreaded Prolog would mean that the heap management routines must be able to deal with the fact that several threads can backtrack simultaneously, causing the heap to shrink and expand in a very complex way.

2.4 Execution Model

Like the storage model, the execution model needs to be modified in order to support multiple threads. Since we do not have any unit-level concurrency (i.e. no native threads) in the emulator this means that the execution model must support time-sharing the emulator between the different threads.

The first problem to solve is to determine where in the emulator loop threads should be switched in and out. The place where this is done is called *synchronizing point*. It is conceivable to have more than one synchronizing point, but that would cause problems. If threads were allowed to be switched in and out anywhere in the emulator, it would be difficult (and error-prone) to make sure that they are switched in at the same place they switched out. Having a single synchronizing point is also desirable in order to minimize the number of places in the emulator that need modification.

A natural candidate for the synchronizing point is the *overflow-check*. This is a piece of code which the emulator executes periodically order to make sure that the data-areas do not overflow. It is also invoked explicitly by certain WAM instructions to ensure that sufficient stack space is available and that any goals unblocked by recent variable bindings are run.

The major benefit of using the overflow-check as location of the synchronizing point is that it already has a mechanism for invoking it. This means that we do not need to write any new code to be used by the scheduler to invoke the thread-switching mechanism. We simply fake a

signal to the emulator that a stack is about to overflow, which will cause the emulator to transfer control to the overflow-check where the thread (possibly) will be switched out. By "piggybacking" on the existing mechanism, we greatly reduces the impact on the emulator.

The second problem is to actually perform the switch. This is done by simply exchanging the reference to the currently executing thread and to the enclosed WS. Since all references to machine registers are made through the WS, this is a very easy procedure.

2.4.1 Runtime vs. Development Systems

SICStus has two modes of execution; *development systems* and *runtime systems*. Runtime systems are linked together with an native application (usually written in C) to create what is known as a *stand-alone application*. A runtime system is basically a subset of SICStus Prolog in the sense that many of the built-in predicates are omitted or have limited functionality [9]. For example, runtime systems have no top-level, no debugger, no profiling, and no save/restore facility. However, our intention is to integrate the two systems, simplifying design and maintenance of future versions of SICStus.

For this implementation, we have concentrated on one of the execution modes, the development system. The development system was chosen since it is commonly used for developing Prolog applications, and therefore more suitable for our purpose.

2.4.2 Exceptions

Runtime and development systems also differ in the way they handle uncaught exceptions. Runtime systems simply return them to the embedding application, while development systems have a catch-all mechanism built-in in the top-level interpreter which catches any exception that has not been caught by the program itself.

The issue which needs to be addressed here is about what happens when a sub-thread (i.e. a thread other than the thread running the top-level interpreter) throws an uncaught exception. As usual, the simplest solution is to "do nothing". Due to the implementation of exceptions in SICStus (exceptions are basically a form of "controlled backtracking" combined with `assert/1` and `retract/1`), this will force the exception to behave just as a normal failure. Therefore, if a predicate throws an exception which is not caught, it will fail all the way up to the thread's goal, where it will terminate the thread. This solution has the benefit of being simple.

The drawback of this "do-nothing" solution is that threads terminate unconditionally when an uncaught exception reaches the top-level goal. This goes also for "unintended" failures which depend on misspelled predicate names, etc. In order to be informed of such failures, a catch-all solution could be programmed using the standard exception primitives. See Figure 2.

```
goal(Arg) :-
        on_exception(Pattern,raw_goal(Arg),handler(Pattern)).

handler(Pattern) :-
        print_message(error,Pattern).

raw_goal(Arg) :-
        ...
        % Here goes the code for the subthread
```

Figure 2: How to implement a catch-all mechanism in a subthread

2.5 Scheduling

Scheduling [4, 13, 14] in multithreaded execution environments can be compared to motion picture soundtracks: if it is done well, it is not noticed—it just contributes to the overall impression of the performance.

A little note on the terminology used in this following section. The algorithms are general enough to be applicable to both low-level microprocessor scheduling and user-level abstract machine scheduling. The classification of algorithms is taken from [16], a text-book on operating system concepts. The terminology is therefore focused on the low-level scheduling: the units which compete for computing resource are called *processes* and the computing resource is called *CPU*. The corresponding terminology for this implementation would be *threads* and *WAM*, respectively.

2.5.1 Preemption

The most important aspect of the scheduling in systems that use logical concurrency is to create the *illusion* of concurrency—that is the whole idea—and in order to do that the scheduler has to be *preemptive*. Preemption means that a thread can be interrupted, letting another thread execute. Without preemption, a thread cannot be interrupted except at certain places, for example I/O calls.

If preemption is not used, the illusion of concurrency is in danger in two ways. First, the concurrency itself is in danger since if a thread cannot be preempted by force, the concurrency depends on the cooperation of the program to suspend itself allowing other threads to execute. The concurrency can therefore easily be destroyed by a vicious program. Second, the illusion is in danger since the cooperation of the program requires explicit calls (such as `yield()` in Java [15]) in order to suspend itself inside tight loops, for example. These explicit calls destroy the illusion because the concurrency, or traces of it, can be seen in the code.

2.5.2 Fairness

Fairness is an important aspect of a scheduler. It guarantees that a process will get access to the CPU at some time in the future. *Strong fairness* guarantees that a process will get an infinite amount of accesses to the CPU in the future (i.e. it will regularly be scheduled for execution, regardless of the CPU load). Fairness is more difficult to achieve than preemption, since it requires that the scheduler keeps some record of CPU usage for each process. However, it is relatively easy to achieve *conditional* fairness, i.e. fairness under certain circumstances. These circumstances are discussed in the next section.

2.5.3 The Scheduling Algorithm

The choice of scheduling algorithm is naturally the most important decision behind the design of a good scheduling mechanism.

2.5.3.1 Considerations

There are many different algorithms to chose from, both preemptive and non-preemptive. The simplest is called *first-come, first-served*, or FCFS. The *ready-set* (the set of processes which are waiting to execute) is organized as a FIFO queue. The first process in the queue is the next process waiting to execute. It executes until it suspends and is then inserted last in the FIFO queue. It is not preemptive; processes execute until they suspend themselves. The problem with this algorithm is that the average waiting time can be quite long if there are processes doing expensive computations without suspending themselves. Other processes will then have to wait until

the executing process is done, which may take quite a while. The FCFS algorithm is inadequate for our needs; it lacks preemption and does not guarantee fairness under any circumstance.

The next one is called *Shortest Job First*, abbreviated *SJF*. SJF scheduling is based on something called *CPU bursts*. A CPU burst is a period of uninterrupted CPU usage. In SJF scheduling, the process with the shortest upcoming CPU burst is scheduled first. This algorithm has been proven optimal [16] in the sense that it minimizes the average waiting time for a given set of processes. The major problem with SJF scheduling is that it is not possible to implement as a short-term scheduling algorithm since it is impossible to predict the size of the CPU burst exactly. It is, however, possible to *estimate* the size of the upcoming CPU bursts. This is usually done by calculating the exponential average of the measured lengths of the previous CPU bursts (process by process, of course). The details on how this is calculated is not very important, but the interested reader may take a look at [16], p. 139-140 for a detailed description of this measurement. The important characteristic of this measure is that it weighs previous CPU bursts differently depending on how long ago they occurred. Recent history gets more weight than less recent history. If the CPU bursts display a fairly localized pattern, i.e. if they do not vary very randomly, it is possible to make a educated guess about the coming CPU bursts.

SJF scheduling can be implemented both with and without preemption. SJF without preemption is the "normal case". With preemption it is usually called *remaining-time-first* scheduling, since the processes are scheduled according to the size of what remains of their CPU burst. However, the strength of SJF lies in non-preemptive scheduling, where the average waiting time can become quite long. If preemptive scheduling is to be used, there are better algorithms than SJF, since it becomes less important to predict the size of the next CPU burst. SJF is thereby not suitable to use in this implementation; it is best suitable in a batch-job system where it is important to minimize the average waiting time.

The third algorithm is called *priority* scheduling. It is basically very simple: each process is associated with a priority and the process with highest priority gets to execute. It can be either preemptive or non-preemptive. Preemptive priority scheduling *interrupts* the currently running process if a new process with higher priority is started, non-preemptive does not. Note that SJF is a special case of priority scheduling: the priority being the inverse of the (estimated) length of the next CPU burst.

None of these algorithms turned out to be suitable for our needs. Instead, we have adopted an algorithm called *Round-Robin*. This algorithm (with a touch of priority scheduling) is the one used in this implementation and is described in detail in the next section.

2.5.3.2 Priority Round-Robin Scheduling

The algorithm used in this implementation is a combination of priority scheduling and Round-Robin scheduling [16, 17], called *Priority Round-Robin* (*PRR*).

PRR scheduling is conducted in the following way: the set of threads ready to execute is kept sorted by priority. When a time-quantum (see Section 2.5.4 for details on the choice of time-quantum) is up, or a thread has suspended itself explicitly, the thread with highest priority is scheduled for execution and the current thread stored away. If it was suspended on a blocking system call, it is inserted in the list of blocked threads, otherwise it is inserted into the list of threads waiting to execute (the ready-list). The insertion into the ready-list is done so that the newly inserted thread is inserted last among all the threads of equal priority. If all threads have the same priority, the list works exactly like a FIFO-queue.

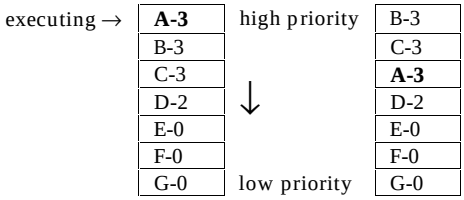

Figure 3: PRR Scheduling. To the left, A is executing with priority 3. To the right, A has been interrupted and inserted last among those threads with equal priority.

This algorithm is far from perfect. It is not fair; there is no guarantee that threads do not starve out other threads. However, it is fair as long as threads do not alter their priorities. One could, of course, remove the priority (and utilize simple Round-Robin), but that would make it impossible to let certain threads be "more important" than others, which is a desirable property in, for example, WWW servers.

A solution to this problem could be to take a glance at the UNIX scheduling model. In addition to the priority of the process, a UNIX process also stores information about what is has been doing lately, giving the scheduler a possibility to make decisions based on how much the process has been using the CPU (this is similar to SJF with estimated CPU bursts). The scheduler then bases the calculation of the *real* priority (used to determine which process should be executed) on the *base* priority (which is set by the user) together with the process' history of CPU usage. This calculation is done in such a fashion that the real priority decreases when the process uses the CPU a lot and increases when it does not. This guarantees that a process will not starve out other processes (at least not with respect to CPU time) while at the same time enable the user to indicate which processes are to get more CPU time than others.

This kind of scheduling algorithm could very well be implemented in SICStus, but the PRR algorithm has proven simple, robust, and providing good performance for most cases.

2.5.4 Choice of Time Quantum

Crucial to (P)RR scheduling is the size of the time quantum, i.e. the maximum period of time a process is allowed to execute before it is interrupted. The size of this period has a large impact on the efficiency of the scheduling. If we have too large a period, the response times (and thereby the concurrency) will suffer. Imagine the scenario where a large number of processes are waiting for input. If all these processes receive input at roughly the same time and the time quantum is large (and assuming that all processes consume their full quanta), it will take very long time before the last process gets to use the processor. For example, if the quantum is 0.5 seconds, the number of processes 50, it will take 0.5*50 = 25 seconds for the last process to get access to the CPU. On the other hand, if the time quantum is too small, the overhead of switching between threads will increase. For example, if the time quantum is smaller than the time it takes to switch between threads, the scheduling overhead will be more than 50%. So, the time-quantum needs to be carefully chosen.

When choosing the time quantum for an operating system, it is solely a question of raw time; a time of 100 ms could be a reasonable alternative [17]. However, in our case there are other possibilities:

2.5.4.1 Real-time Controlled Time Quantum

The variant is heavily influenced by the "classic" solution, which meant setting up a timer interrupt and thereby reschedule at a fixed real-time interval (for example, 50 ms). However, as discussed in Section 2.4, it is not possible to reschedule directly, but instead we have to wait until the emulator reaches the synchronizing point. This is achieved by setting a flag forcing the emulator to jump to the synchronizing point as soon as possible, and then rescheduling when

arriving there. This means that the chosen real-time interval is rounded up to the nearest arrival at the synchronizing point.

2.5.4.2 WAM Instruction Counting

Instead of rescheduling based on a real-time interval, it is possible to reschedule after the emulator has executed a certain number of WAM instructions. This method requires that a counter be kept updated and compared for each instruction in order to see if a reschedule should take place. This method has the benefit of not requiring any signals (timer interrupts, to be more precise), which means that it is more portable than the previous method.

However, this variant has two serious problem. The first is efficiency: counting each WAM instruction turned out to cause a large overhead (almost 20%). The second problem is related to native code execution. In order for this variant to work with native code, it would be necessary to modify the native code compiler so WAM instructions executed natively also are included in the count. Apart from being tedious to implement, this would presumably (we have not done any tests on this) degrade the performance of native code execution considerably.

2.5.4.3 Overflow-check Counting

Another method of determining the time quantum is to count the number of "spontaneous" arrivals at the overflow-check, and reschedule every *n*th time. This method has the benefit of the WAM instruction counting method of not needing any timer interrupts or other signals and at the same time avoiding both the performance trap and the problems with native code. It is also simple to implement.

There is a subtle problem with this method and it has to do with how often the emulator reaches the overflow check "spontaneously". Usually, it is reached often enough to be fairly close to the 50 ms we tried for the real-time method, so under normal circumstances this is a perfectly acceptable solution. There are, however, situations were the overflow-check is not reached at all. Consider the following example:

```
consume_very_much_cpu_and_do_very_little_work :-
    repeat,
    fail.
```

This program gets caught in a infinite backtracking-loop in which no overflow-checks are done. The predicate `repeat/0` basically pushes an infinite number choicepoints upon backtracking. For a more detailed explanation of `repeat/0`, see [9], p. 110. Basically, we are not guaranteed that there will be any overflow-checks at all, even if they for a normal program are performed quite regularly.

The solution we have adopted is a combination of the real-time controlled variant and the overflow-check counting variant. By rescheduling every overflow-check (setting *n* to 1) and using timer interrupts when multiple threads are active to make sure that the example above does not block the entire system, we get a solution which is simple to implement and efficient. The portability drawback of timer interrupts seems to be unavoidable.

Chapter 3 - Programming Interface

This section will describe the predicates used to create and destroy threads, communicate between threads, etc. It will also include a discussion on semantics of these predicates and the modified semantics of some of the built-in predicates of SICStus.

3.1 Primitives

The set of primitives used to create, destroy, and in other ways manipulate threads have deliberately been kept to a minimum. There are many more features one might wish to see in a threads implementation, but in order to keep the implementation simple and the ideas clear, these have been left out. See Section 6.3 for a discussion on how this interface might be extended.

Predicate	Description
`spawn(:Goal, -ThreadID)`	Creates a new thread and schedules it for execution. The new thread will execute the goal `Goal`, similarly to the predicate `call/1`. `ThreadID` will be bound to the identifier of the new thread. Together with the new thread, a message port (also referred to as input queue) will be created. This port is intended to be used for synchronization and communication between different threads The new thread will, if no measures are taken, complete execution and succeed or fail silently. In other words, there is no primitive `join/1` which waits for a thread to complete. However, this can easily be implemented using `send/2` and `receive/1`. See Figure 4.
`send(+ThreadID, +Term)`	Sends `Term` to the thread indicated by `ThreadID`. This predicate always succeeds (or throw a domain error exception). `Term` will be inserted last in the in the receiving thread's input queue.
`receive(?Term)`	Extract the first element in the thread's input queue which is unifiable with `Term`. If no such terms exist, the thread is suspended.
`self(-ThreadID)`	`ThreadID` is the thread identifier of the running thread.
`kill(+ThreadID)`	Causes `ThreadID` to terminate. Always succeed.
`wait(+Ms)`	Suspends the currently running thread and then waits at least `Ms` milliseconds before resuming. The actual time elapsed before the thread is resumed is guaranteed to be at least `Ms`.

Table 1: Primitives for manipulating threads in SICStus MT

```
spawn_joinable(Goal,ThreadID) :-
    self(Self),
    spawn(joinable(Goal,Self),ThreadID).

joinable(Goal,Parent) :-
    call(Goal),               % Do the actual work
    self(Self),
    send(Parent,done(Self)).  % Tell parent that the thread has completed.

join(ThreadID) :-
    receive(done(ThreadID)).

run :-
    spawn_joinable(..., ThreadID), % Same syntax as spawn/2 join(ThreadID).
                                   % Will suspend until ThreadID is done.
```

Figure 4: How to implement `join/1` using `send/2` and `receive/1`.

3.2 Semantics

The predicates can be divided into two groups. First, predicates *with* side-effects on the execution environment, such as `spawn/2` and `send/2`. Second, predicates without side-effects but which bind their arguments (or succeed/fail) depending on information in the execution environment.

3.2.1 Backtracking

Backtracking in the presence of side-effects is a not a trivial problem. De Bosschere describes four different ways of handling backtracking in predicates with side-effects [18]:

1. Disallow both undoing and redoing. This is the most restrictive one. No choicepoint is pushed and redoing a side-effect is not possible.
2. Disallow only undoing. This is the simplest solution and the one we have implemented. Simply avoid pushing a choicepoint. This will cause the side-effect(s) to be performed over and over again.
3. Disallow only redoing. This is conceptually a little more difficult to grasp. This means that the side-effect should be undone which is not always easy. In our case, for example, undoing `spawn/2` would have result in killing the spawned thread and undoing the side-effects of the new thread. Redoing is disabled, so we do not start a new thread.
4. Allow both undoing and redoing. This combines points 2 and 3.

We have the chosen the second solution: no choicepoint is pushed and no measure is taken to prevent redoing them. In the case of, for example, `spawn/2` this has the possibly unpleasant result that backtracking back and forth over the call will create several threads.

```
forkbomb :-
     repeat,
     spawn(forkbomb,_),
     fail.
```

Figure 5: Backtracking into `spawn/2`.

The code in Figure 5 is a cousin of the infamous fork-bomb, written in SICStus MT.

3.2.2 The Communication And Synchronization Mechanism

The model of communication and synchronization is heavily influenced by the model used in ERLANG [19]. The model is *message-based* as opposed to *blackboard-based* [20, 18] (also known as *tuple space based* [47]. This means that the communication is based on sending explicit messages as opposed to using a shared store (blackboard) of some kind. Message-based systems have the advantage over blackboard-based systems that they are inherently more scalable; there is no central point through where all message must pass. They are on the other hand less expressive since they require that messages are addressed to a certain destination. In blackboard-based systems, messages are simply posted on the blackboard.

Each thread has a unidirectional message port where it can receive messages, consisting of standard Prolog terms. *Unidirectional* simply means that messages only pass in one direction through the port; it is not used for outgoing messages. The port is invisible, i.e. it is treated as an integral part of the thread. A message can be any kind of Prolog term.

The communication mechanism is *asynchronous*. Asynchronous means that the sender does not need to wait until the receiver is ready to receive the message. This also means that the mechanism is *buffered*, i.e. the communication medium (the message queue) has a memory of its own where it can store messages until they are ready to be picked up by the receiving thread. The message port is basically a FIFO-structure, which means that it is completely ordered. However, as we will discuss later on, the programmer can specify the order in which terms are received.

Since shared heaps is not an option, sending a message must be done by copying it into the static area and inserting it into the receiving thread's input queue. If the receiving thread is suspended on a call to `receive/1` it is *lazily* switched in for execution, i.e. just moved to the ready-list. If the receiving thread is suspended for some other reason, nothing happens. The alternative would be *eager* thread switching, i.e. preempting the receiving thread, disregarding any priorities. See Section 5.4.1 for a discussion on the performance of these two approaches. When the receiving thread is eventually resumed, it must unpack the message on its own heap.

See Figure 6 and Figure 7 for an example on how Prolog threads may communicate and synchronize using `send/2` and `receive/1`.

```
echo :-
    receive(Term),
    write(Term),
    nl,
    echo.

run :-
    spawn(echo, EchoThread),
    send(EchoThread, term1),
    send(EchoThread, term2),
    ...
    send(EchoThread, termn),
```

Figure 6: Communication using `send/2` and `receive/1`. The example spawns a simple echo thread which executes in the background and echos everything sent to it.

```
  reader :-
    receive(readlock(NextReader)),
    read(Term),
    self(Self),
    send(NextReader, readlock(Self)),
    reader.

run :-
   spawn(reader, ReaderA),
   spawn(reader, ReaderB),
   send(ReaderA, readlock(ReaderB)),
   receive(dummy).
```

Figure 7: Synchronizing using `send/2` and `receive/1`. Two threads are spawned (`ReaderA` and `ReaderB`) cooperating to read terms from standard input.

3.2.2.1 Receiving Messages Out of Order

The ERLANG implementation of send/receive features a very practical construction: the ability to specify which messages to receive for a particular call to the receive-primitive. This is also referred to as *message non-determinism* [18]. This is in ERLANG implemented by pattern matching and we have used the Prolog unification mechanism to obtain a similar result.

A little note on the terminology. In [18], De Bosschere talks about *message non-determinism* and *media non-determinism* where the latter refers to the ability to be able to avoid specifying the origin of the message. This is implicit in our solution.

Let us take an example. In Figure 8, the thread denoted `ThrID` will be suspended on the call to `receive(start)` and will not be resumed until it has received the atom `start`; in this case after it has received all terms `term1`, …, `termn`. This is useful if, for example, the main thread needs to send all terms before the thread starts to process any of them.

```
thread :-
    receive(start),
    do_rest.

do_rest :-
    receive(Term),
    perform_action(Term),
    do_rest.

main :-
    spawn(thread,ThrID),
    send(term1,ThrID),
    send(term2,ThrID),
    ...
    send(termn,ThrID),
    send(start,ThrID).
```

Figure 8: Out-of-order receives in Prolog

In the Game-of-Life benchmark, described in Section 5.4.1, there is a particular construction which relies on out-of-order receives. Since the cells work asynchronously (without a global "conductor" telling them when to do a state transition), each cell needs to make sure that the incoming messages are grouped by generation. This means that if a cell in generation x receives a message from a cell which is in generation x+1, it must be able to defer that message until itself is in generation x+1. In our implementation, this is solved by letting each cell loop through

all its neighbors and for each one, wait for a message from that particular neighbor. In that way, we are guaranteed that the messages are processed in the correct generation. This would be very tedious to code without language support, since we would need to keep a separate list of terms which were received “too early”, a list which needs to be maintained, sent around to all predicates calling `receive/1`, and searched for each call to `receive/1`.

However, there are performance issues worth discussing here. Recall from the previous section that a message needs to be unpacked every time the receiving thread wishes to examine it; unpacked messages cannot be reused since they may have been garbage collected. Furthermore, out-of-order receives will result in a list of “currently unmatched” messages, i.e. messages which have arrived but are not unifiable with the argument to `receive/1`. This means that messages can be delayed in the message queue for an indefinite period of time. This has the effect that in order to examine the messages in input-queue, all of them (including the “currently unmatched” ones) needs to be unpacked.

3.2.3 Suspend and Resume

Some readers will probable have noticed the absence of the primitives *suspend* and *resume*. They are fairly common in thread implementations. POSIX implements them under the names `thr_suspend()` and `thr_continue()` [4]. They are, however, not needed in a basic set of thread primitives such as ours. In fact, there are only two situations where these two primitives are necessary. The first situation is if one would want to implement a debugger. The debugger would need to be able to step the threads in different ways. The second situation is if one would want to implement an external scheduler. Such a scheduler could be implemented by using a thread which controls which threads get to use the CPU (or the emulator in this case) by suspending and resuming these threads.

Another strong reason to exclude them from the set of primitives is that, besides them being halfway unnecessary, they can be implemented by using `send/2` and `receive/1`, as illustrated in Figure 9.

```
suspend :-
    receive(dummy).

resume(ThreadID) :-
    send(ThreadID,dummy).
```

Figure 9: Implementation of suspend/resume using send/receive

3.2.4 Exceptions—Where and Why?

Currently, none of the predicates throw any exceptions (for an explanation of the exception mechanism, see [9], p. 111-113), they only succeed or fail silently. This is not a desirable behavior. Instead, they should throw exceptions where this is appropriate (the implementation of this is out-of-scope for this thesis). There are two situations which are interesting:

3.2.4.1 Illegal Arguments

All the predicates should throw exceptions when they discover that their arguments are of the wrong type. For example, if they expect a thread-identifier and receives something that cannot be a thread-identifier, they should throw an exception.

3.2.4.2 Non-existent Target Threads

When the predicates which manipulate other threads than themselves (i.e. `send/2` and `kill/1`) discover that the specified thread (also called the *target thread*) does not exist, they need to take appropriate action. For `send/2` this amounts to throwing an exception. One might argue that `send/2` should simply succeed, to avoid threads failing when they do not care about whether or not the message arrived properly. However, by throwing an exception we keep the flexibility of allowing the user to handle the exception or ignoring it.

In the case of `kill/1` we decided not to throw an exception when the target thread could not be found, but instead let the predicate always succeed. One might use the same argument as for `send/2` and claim that we should allow the caller to take action if the target thread does not exist. On the other hand, if we view the semantics of `kill/1` as guaranteeing that the target thread does not exists after the call returns, always succeeding is a quite reasonable behavior.

Chapter 4 - The Problem Of Blocking System Calls

The problem of blocking system calls in user-level (as opposed to kernel-level) thread implementations is a well-known and well-investigated problem [21, 17]. The core of the problem is that the operating system kernel (by definition) is unaware of the existence of user-level threads. Therefore, when a blocking system call is performed, the kernel suspends the entire process for the duration of the system call, instead of scheduling another thread for execution, which is the desirable behavior.

4.1 Possible Solutions

There is not that many ways of solving the problem. We must in some way prevent a given blocking system call from blocking the entire process and find a way of scheduling another thread instead. We have explored two approaches to the problem, the *cautious approach* and the *cavalier approach*.

The cautious approach uses a relatively complex mechanism in order to examine system resources in order to determine, *without making the call*, whether or not the system call would block the process. If the call could not be performed without blocking the process, the thread is suspended and another thread is switched in. Otherwise, the thread continues with the read and returns normally. The main problem of this approach is complexity. Each system call which might block must be preceded by a piece of code (called *jacket* in [17]) in order to determine whether or not the system call would block or not. This turned out to be quite non-trivial—the documentation on when system calls block is often inadequate and examining system resources not a very portable procedure.

The cavalier approach relies on the fact that most operating systems support some form of performing system calls without blocking the process at all, also known as *asynchronous I/O*. Instead of trying to determine on beforehand whether or not a system call is about to block, we simply perform the system call asynchronously. A check is made after the call to determine if the call was completed and if not, the thread is suspended and then resumed when the asynchronous system call has completed.

The cavalier approach wins the game on the fact that it is simpler to implement and more robust. We leave it to the individual system call to determine whether or not it is about to block. This relieves us from having to write specialized code for each system call which is not only tedious but also error prone.

4.2 Emulator Support

The solutions discussed above both need a mechanism for communicating with the emulator. More precisely, they need to be able to inform the emulator when a thread should be suspended as a result of a blocking system call. The idea is to introduce an extra return code for predicate-calls in addition to `TRUE`/`FALSE` (which represent success and failure, respectively). The new return code is called `SUSPEND` and is returned when the thread executing the predicate should be suspended.

Since the process of suspending a thread as a result of making a blocking system call varies significantly depending on the nature of the system call, the actual work of suspending a thread (setting bits, moving threads between lists, etc.) is done by the code performing the system call

(see Figure 11). The only action required by the emulator is to immediately jump to the synchronizing point in order to perform a reschedule.

```
/*
 * All blocking system calls are made in predicates implemented in C.
 * These all appear as ENTER_C in the WAM code.
 */
CaseX(ENTER_C)
  ...
  switch ((*Func->code.cinfo)(Arg))
    {
    case FALSE:
      goto fail;

    case SUSPEND:     /* The C-routine was about to suspend... */
      goto heap_oflo; /* ... so force a new thread to be scheduled */

    case TRUE: /* fall through */
    }
  ...
  LoadH; goto proceed_w;
```

Figure 10: Emulator code for handling suspended C-predicates

4.2.1 Native Code

One of the features of SICStus is the possibility of executing native code [44, 46]. This means that instead of interpreting predicates or executing byte-compiled code, the Prolog code is compiled to native code and inserted directly into memory and executed as if it was a regular C function. The purpose of this is of course execution speed, and speedups of 3-4 times are not unusual.

The multithreaded execution model maintains full compatibility with native code execution, since the thread scheduling mechanism is built upon an already existing mechanism—the overflow-check—for which the native kernel already has support for. By simulating a heap-overflow when rescheduling should take place, the native code kernel will automatically escape back to the emulator, perform an overflow-check and thereby reschedule. When the thread later on is scheduled again, the native code execution will continue as normal. However, the implementation is not yet fully compatible, due to its lack of support for blocking system calls. The native kernel must be modified to support the `SUSPEND` return code indicating that a immediate reschedule should take place. Since this requires hacking into the native kernel, the implementation of this has been left outside this thesis.

4.2.2 Suspending The Emulator

The emulator will sometimes be in the situation where there are no more threads to schedule. For example, this happens immediately at startup when the top-level thread waits for input from the user. This should cause the Prolog process to suspend itself, just as if it would have if it had performed a normal, blocking system call.

This is implemented by a small piece of code in the scheduler. When the scheduler has suspended the top-level thread and realizes that the list of threads waiting to execute is empty, it suspends the process[1] by calling `pause()`. The process is then resumed (i.e. `pause()` returns)

[1] Suspending and resuming processes is—as one would expect—platform dependent. The case described here is for Solaris, but most modern operating systems implement something similar. See also Section 4.4.

when a signal is received. The signal is triggered by one of two reasons. Either the synchronous I/O mechanism sends a signal to indicate that the I/O call was completed, or the timer mechanism sends a signal to indicate that we have one or more threads suspended on a call to `wait/1` and that we need to examine the queue to schedule those threads for which the timer-period has expired. These actions are taken in the signal-handler routines, so that when `pause()` returns we examine the ready-list and if everything went right we should have a thread waiting to execute. However, for different reasons we might have received a false alarm, in case we will simply be suspended again.

4.3 Predicates

This section described some of the predicates which need to be modified in order to support blocking system calls.

4.3.1 Character Input Predicates

There are quite a few predicates in SICStus in some way concerned with reading data from a file or from the terminal. A complete list is available in [9]. Four of these are concerned with reading characters and returning them to the calling Prolog predicate. These are `get/1-2` and `get0/1-2`. They all read a character from either the standard input or from the specified stream. They are from our point of view practically identical (they differ only on how they treat white-spaces and from where they read their data).

```
BOOL prolog_get1(Arg)
     Argdecl;
{
  int i;
  SP_stream *s = w->input_stream_ptr;

  i = readchar(s,TRUE,preds.get1);

  if (check_susp(activeThread, s))
    return SUSPEND;

  if (i > -2)
    {
      Unify_constant(MakeSmall(i),X(0));
      return TRUE;
    }

  if (i == -2)
    raise_error("get",1,0);

  return FALSE;
}
```

Figure 11: Source code for the `get/1` predicate

The predicate `get/1` (see Figure 11) demonstrates the cavalier approach quite well. The call to `readchar()` is made and a small piece of code checks whether or not the thread should be suspended and if so return `SUSPEND`. The first approach looked very similar, with the difference that it made a call to `select()` before the call to `readchar()`, and had no code at all after.

4.3.2 Socket I/O

One of the important external libraries which needed attention in this matter was the socket-library. The socket-library enables Prolog applications to talk TCP/IP with other applications. Supporting the socket-library was not at all trivial, but the problems were not so much related to the actual I/O itself as to the general question of performing blocking system calls in foreign code, a topic addressed in Section 4.5.

The SICStus implementation of generic streams enables programmers to create their own kind of streams which then can be treated as any other stream by other Prolog predicates. The socket-library takes advantage of this fact and creates a generic stream encapsulating the underlying TCP/IP-socket. Thereby predicates can operate on socket streams exactly in the same way as they operate on, for example, terminal streams. This has a big advantage from our point of view. There is no new set of I/O predicates for sockets, except those predicates used to manage the sockets themselves, such as `socket_bind/2-3`, which means that the support for blocking stream primitives comes for free.

4.3.3 Output Primitives

The direct output primitives (such as `put/1` and `format/2-3`; calls used to output data to a terminal, file or some other sort of output device) used in SICStus are all based the low-level system-call (`write()` [22]). This call behaves similarly to `read()` with respect to blocking, but does not block during "normal" use. Therefore, support for blocking system calls in direct output primitives has not been implemented. The call `write()` does, however, block under certain file-system conditions such as mandatory file/record locking and this should of course be supported in a released version of SICStus.

There are other output primitives of a more "implicit" nature. An example of such a call is the socket library call `connect()` which is called in order to establish a connection to a socket. This call may suspend if the connection cannot be established directly. The reason why this call is classified as an output predicate is that in order to find out if it is going to block is to select it (i.e. use `select()`) for writing.

4.4 Portability Aspects

This implementation has been done on Solaris, and while most parts of the solution are directly portable to most operating systems supported by SICStus, there are some issues worth special attention.

4.4.1 Signals

One problem inherent in this implementation is that it relies on the existence of signals, a mechanism which exists in most UNIX-like operating systems, but there are some operating systems which do not support signals the way they are implemented under UNIX-like operating systems. Most notably among these are Win32.

One possible solution to this is to emulate the behavior of signals by using native threads. Instead of setting up a signal handler to inform the emulator that the system call has completed, we let a native thread do the job instead. When that thread has completed, it simulates a signal by, for example, calling the signal handler with the appropriate arguments. Obviously this requires that the OS supports native threads. If that is not the case (and no signaling mechanism is available), SICStus MT cannot be supported. The problems of this solution is that we need to have an native thread for each system call, presumable created every time a blocking system call is made. This might cause some performance problems.

4.4.2 Performing Asynchronous System Calls

Recall the discussion in Section 4.1 and the fact that the cavalier approach relies on the fact that there is some way of executing system calls *asynchronously*, i.e. so that they do not block the entire process. This can be done under most operating systems, but not in a very portable manner.

What happens if the underlying operating system does not support any way of performing asynchronous system calls? Well, if it supports native threads, one solution is to emulate the asynchronous call by spawning an native thread which performs the call. In this way we buy into the operating system's way of handling blocking system calls to the expense of spawning a new thread for each time a blocking system call is made. If the underlying operating system does not support either asynchronous system calls, threads, or another way of getting around the problem of performing blocking system calls , we have a dead end. The only way out in that case is to give up, execute the blocking system call and accept that the entire process is blocked.

4.5 Foreign Language Support

The SICStus foreign language interface currently only supports C, but the C support subsystem is quite extensive. It is possible to call C from Prolog, the other way around, and recursively (C to Prolog to C to Prolog to C to …). The support consists of a set of C-functions for controlling the emulator, creating and manipulating terms, and a set of predicates to specify argument information (type, instantiation, etc.) and dynamically link the C routines. It also consists of a mechanism for generating *glue-code*, stubs which convert Prolog arguments into a C argument list before the call and unifies uninstantiated arguments on return, and so on. For more details on how the foreign language interface works, see [9].

4.5.1 Blocking System Calls In Foreign Code

There is a major difference between supporting blocking system calls in built-in C predicates (i.e. such as `get/1-2`) and doing it for predicates in foreign code. The difference is that the built-in predicates are quite few and uncomplicated and are statically linked into the emulator. We have full control over the structure of the functions. The external libraries, on the other hand, can be written by practically anybody, and might be very complicated in terms of its call graph. Specifically, it can system calls located very deep in the call graph. What we have to do is to extend the foreign language interface, specifically the functions used to control the emulator, to include support for blocking system calls. Such an interface should be small and easy to understand. Otherwise the possibility of misusing it becomes too large. It must also be general. The built-in predicates which support blocking system call are quite simple. More specifically, the blocking system calls are not located deep in the call graph (i.e. they only occur in the function directly called by the emulator). Take a look at Figure 11. There is a single test with an immediate return which directly returns control to the emulator which then can examine the exit-code. If the system call would be located deep inside a call graph (as it might be in an external library), it would still be necessary to be able to return all the way back to the emulator, and preferably without any requirements on the user. Of course, it would be possible to document that any call to a potentially blocking system call has to be accompanied by code enabling immediate return to the emulator, but that would clutter the code of the external library and put a lot of responsibility in the hand of the programmer of the external library.

Our solution consists of two parts. The first part consists of an extension to the set of functions for the foreign language interface and the second part is a modification of the glue-code generator. The function is basically the same function as `check_susp()` which is called in the `get/1-2` and `get0/1-2` predicates (see Figure 11), i.e. if the system call was about to block, it marks the thread as suspended, makes sure that the emulator gets a `SIGPOLL` when I/O is possible on the specified file-descriptor, and finally returns `TRUE` if the thread should be suspended

and FALSE otherwise. This function is then called directly after the potentially blocking system call in the external library. See Figure 12.

```
if ((msgsock = accept((SPSock)socket, [...] )))
  {
    if (SP_handle_blocking_syscall(socket, S_INPUT, EWOULDBLOCK))
      return (SP_stream *)SUSPEND;

    ...
  }
```

Figure 12: How blocking system calls can be handled in foreign code. This example comes from the socket-library.

The modification of the glue-code generator is necessary mainly due to the fact that when a external library call returns, output-argument unifications are done. These has to be ignored if the thread is about to block -- the library call is not really returning but just releasing control to the emulator.

Chapter 5 - Discussion

One of the central points of this thesis was to implement an efficient multithreaded execution environment, both with respect to execution speed and to memory consumption.

5.1 Memory Consumption

The main issue here is the question “how lightweight Prolog threads can be?” It is immediately obvious that the data occupied in the static area by the extended WS is negligible compared to the size of the stack-consumption; if the extended WS consumes 1 kb, it is outweighed 64:1 by the stacks if the stack sizes are those given in figure Table 2. The size consumed by the compiled messages depends naturally on the number of messages, but the overhead of storing them separately as compiled messages in the static area is negligible. The stack-consumption is partly determined by the initial size of the stacks (which are controlled by environment variables, see [9], p. 12-13), but after some experimenting with these it became obvious that there was a lower limit which was very quickly exceeded and setting an initial size below this limit caused unnecessary stack shifting. These number are shown in table Table 2.

Data Area	Minimum Size	
	kbytes	kwords
The Global Stack	32	8
The Local Stack	16	4
The Choicepoint Stack	8	2
The Trail Stack	8	2

Table 2: Recommended minimum stack sizes (in bytes/words on a 32-bit architecture) for SICStus MT

If we use these numbers to calculate the maximum number of threads which fit into the address space of 256 MBytes (the problem of the small address space is discussed in the next section) we get 4096 threads. 4096 threads may seem like a lot, but in certain applications, 4096 threads is not a very large number. For example, the Game-Of-Life benchmark exceeds 4096 threads with a board-size of 64 by 64, which is not a very large board. There are very many interesting patterns larger than that.

We have not found a real solution for this problem; the heap consumption of the WAM is not easy to change. However, there is a more immediate problem than simply large memory consumption, and that is the problem with the 256 Mbyte address space which will be addressed in the next section.

5.2 Address Space Fragmentation

As mentioned above, the address space of SICStus is only 256 Mbytes on a 32 bit architecture. The reason for this is that SICStus uses the upper 4 bits of each data-cell for tags to distinguish different kind of data-cells, such as atoms, lists, structure cells, etc. There are also bits used by

the garbage collector, but these occupy the lower 2 bits which are always unused since all data-areas are 4-byte aligned. These bits reduces the amount of addresses expressible to 2^{28} bytes = 256 Mbytes.

Not only does this mean that the Prolog stacks (those data-areas which needs to be addressed by data-cells, which excludes the static area which is only referenced by full 32-bit pointers) cannot exceed 256 Mbytes in size, it means that they have to be located at optimal places in order for this to be possible; i.e. the address space can not be *fragmented*. Unfortunately, the address space *will* become fragmented due to how stacks are expanded. Stack expansion, also called *stack shifting*, is done by allocating a new stack with doubled size and then shifting over the stack to the new area. If we expand a stack of size *n* bytes, there has to be 2*n* bytes free, and after shifting the stack to the new position, the *n* bytes from the old location will be left unused, fragmenting the address space. This may, under unfavorable situations, cause address space exhaustion even though the stacks total size is far below 256 Mbytes.

5.2.1 Optimal Address Space Utilization Using `mmap()`

There is one way to avoid address space fragmentation. By using the system call `mmap()` [22], the data-areas can be mapped to fixed locations in the address space. This has two advantages. First, expanding the stacks becomes very efficient, since no copying needs to be done. Second, since the stacks are not reallocated in the way described above, the address space does not become fragmented.

Unfortunately, this solution only allows one set of stacks and it is therefore not very suitable to use in SICStus MT. It is possible, but it would require that the number of threads be determined on beforehand. It would also require that the address space be equally (or at least statically) divided among the threads. Both of the consequences are unacceptable.

5.2.2 Proposed Solution

The most attractive solution is to ignore the problem for now and await the arrival of 64-bit architectures. SICStus has already been ported to one such architecture, the DEC/Alpha platform. However, time has not permitted testing this implementation on that platform, but this should (theoretically, at least) not be a problem. On 64-bit platforms we get rid of the 256 Mbytes limit which solves the problem of a fragmented address space, so we can utilize all the physical and virtual memory available. This becomes especially important on machines with large amounts of RAM. Previously it was impossible to utilize more than 256 Mbytes on those machines, even with no fragmentation at all.

5.3 Comparison With Other Multithreaded Environments

This section will compare SICStus MT with other multithreaded execution environments. Performance issues are discussed in Section 5.4.

5.3.1 ERLANG

The functional programming language ERLANG [19], developed at the Computer Science Laboratory at Ericsson, has had a heavy influence on the design of this SICStus MT and the influence can most clearly be seen on the programming interface and the choice of primitives and their semantics. The ideas of communication and synchronization using a single message port per thread and the absence of the primitives *suspend* and *resume* both originate from ERLANG.

ERLANG itself is a functional language which is very suitable to write process-oriented programs in. The syntax and semantics for creating and manipulating threads is very simple and

easy to understand. The piece of code in figure Figure 13 spawns a thread which increments a counter each time it receives a message containing the atom `increment`. The corresponding piece of code in Prolog can be seen in figure Figure 14.

```
start() ->
    spawn(counter, loop, [0]).

loop(Val) ->
    receive
      increment ->
        loop(Val + 1)
    end.
```

Figure 13: Example of inter-thread communication in ERLANG. This example spawns a thread which increments a counter each time it receives the atom `increment`.

```
start :-
    spawn(loop(Parent,0),_),
    receive(_).

loop(Parent,Val) :-
    receive(increment), !,
    Val0 is Val + 1,
    loop(Parent,Val0).

loop(Parent,Val) :-
    receive(die),
    send(Parent,done).
```

Figure 14: Example of inter-thread communication in Prolog. Roughly the same example as in Figure 13, but in Prolog. The thread terminates when it receives the atom `die`. Note the `!` (cut) after `receive(increment)`. Without the cut, a choicepoint would be pushed for each incoming message.

5.3.2 Oz 2.0

Oz 2.0 [23] is a so called *multi-paradigm* programming language developed at the Programming Systems Lab at DFKI, the German Research Center for Artificial Intelligence. It is called multi-paradigm since it incorporates ideas from so conceptually different programming language paradigms such as functional, declarative (logic programming), concurrent, constraint, imperative, and object-oriented.

Oz 2.0 creates threads by using the construct `thread ... end`, which does not only create a thread, but constitutes a ordinary block which has a value. This gives the programmer flexibility and expressive power (which sometimes limits the readability of the program). See Figure 15.

```
declare fun {Fib X}
  case X of
      0 then 1
   [] 1 then 1
   else thread {Fib X-1} end + {Fib X-2} end
end
```

Figure 15: A concurrent Fibonacci function in Oz 2.0. This version is however very inefficient, since it creates an exponential number of threads.

Oz 2.0 does not take quite as a minimalist approach as our implementation (or ERLANG, for that part), which is noticeable both in the language as a whole and in the implementation of threads. For example, the main message passing and thread synchronization mechanism in Oz 2.0 is built upon the abstract datatype `Port`. A port is more expressive (but also more complex to implement) than the message-ports in SICStus MT and in ERLANG. Unlike a message port, ports in Oz 2.0 can be shared among threads, be embedded in other data structures, and passed between threads. In general, ports in Oz 2.0 can be treated just as another datatype.

Unlike SICStus MT and ERLANG, Oz 2.0 implements the suspend and resume primitives (see Section 3.2.3). Oz 2.0 also implements a primitive `injectException` which raises an exception in another thread.

A major difference between Oz 2.0 and SICStus MT (together with ERLANG) which has considerable performance impact (see Section 2.3), is that Oz 2.0 implements a shared global stack. By having a shared global stack, it is possible to send references to terms and other structures between threads without any need for copying data. This can be utilized in for example inter-thread message passing and thereby reducing the work done in sending a message to sending a reference. In some situations (such as the Game-of-Life benchmark, discussed in Section 5.4.1) we can avoid message passing entirely by be able to simply read information directly from a data structure.

5.3.3 Java

Java [15, 24, 25] (developed by Sun Microsystems, Inc.) is, if all buzzwords are removed, an object oriented programming language with C-like syntax and extensive WWW and Internet support. Naturally, its capabilities in the area of Internet, WWW, and so on, is quite irrelevant here; we are mainly interested in its multithreaded execution environment.

A little surprising is the fact that neither Java's language specification [25] nor the virtual machine specification specifies any restraints on the scheduling algorithm of the virtual machine. As a consequence, a (concurrent) Java program cannot presume that the underlying scheduler is neither fair nor preemptive. This means that a concurrent Java program needs to insert calls to `yield()` at strategic locations in order to make sure that other threads in the application are allowed access to the CPU.

In Java, a thread is created by creating a subclass from the base class `Thread`, thereby associating an object with the thread. To start the thread, the method `start()` is executed on the thread-object. This will spawn a new thread which will, when scheduled, execute the method `run()` in the thread-object. The somewhat unnatural association between a thread and an object makes it a little tedious to create a thread. Consider the code in Figure 16. The corresponding code in Prolog would consist of a simple call to the predicate `spawn/2`, while the Java code is at least 10-12 lines.

```
public class MyThread extends Thread
{
  public static void main(String argv[])
  {
    MyThread mt;

    mt = new MyThread();
    mt.start();
  }

  public void run()
  {
    // do work...
  }
}
```

Figure 16: Sample code for creating a thread in Java.

Not only is it more tedious to create a thread, the expressive power and the possibility of fine-grain concurrency is diminished since a new class needs to be created every time the programmer wants to execute something concurrently. For example, a raytracer might want to execute each separate trace concurrently in order to utilize multiple processors. In Java, the programmer would need to create a new class for the very specific purpose of tracing a single pixel. This has been simplified a little by the introduction of *inner classes* which supports declaring classes encapsulated in other classes. In Oz 2.0, it would be enough if the loop through the pixels was encapsulated in a `thread ... end` construct.

Like Oz 2.0, but unlike ERLANG and Prolog, Java implements inter-thread communication by using datatypes. However, in Java this is considerably more complicated than in Oz 2.0, even if the structure is similar. The datatypes `PipedInputStream` and `PipedOutputStream` (which correspond to the Oz 2.0 datatype `Port`) supply the functionality of writing at one end and reading at the other. However, using these alone is not enough; they only supply the "pipe" characteristic of connecting two streams. The streams themselves are subclasses of `InputStream` or `OutputStream` and must be created separately and then connected together using the functionality of the piped stream classes.

This sounds no more complicated than it is and the mechanism has more drawbacks than being simply complex and difficult to understand. The inter-thread pipes are designed to be able to use over arbitrary serial communication devices (such as a TCP/IP-connection). To accomplish this, all objects which are to be sent between threads must implement the `Serializable` interface, which means that they can be written upon a serial device and thereafter reconstructed at the other end. This interface also enables the programmer to have full control on how the object is written down on the stream and how it should be read, enabling an easy implementation of, for example, a transparently compressing stream.

The actual problem with the Java implementation of inter-thread communication is that it is too general. The piped streams and the serializable interface are important issues in Internet and WWW-based applications—it enables them to send objects over the Internet in a generic manner—but they are too clumsy and complex too use for light-weight inter-thread communication. It is, for example, not necessary to support arbitrary communication devices when communicating between threads on shared memory architectures, neither is it necessary to have full control on how the data is stored while in transfer.

As a remedy, we have implemented an extended thread class with a light-weight built-in message-port. Sending a message then amounts to passing a reference to the receiving thread. See Figure 17.

```
private List messagePort; // doubly-linked list with messages (= objects)
  ...

  public synchronized void send(Object obj)
  {
    messagePort.insertLast(obj);
    this.notify();
  }

  public synchronized Object receive()
      throws InterruptedException
  {
    // Suspend if message queue is empty
    while (messagePort.length == 0)
       this.wait();

    return messagePort.removeFirst();
  }

  ...
```

Figure 17: Implementing send/receive in Java without using the piped input and output streams.

See Section 5.4 for a discussion on the performance of this implementation.

5.3.4 CS-Prolog Professional

CS-Prolog Professional [35, 45] is a Prolog system developed in Hungary. It supports several independent processes (to avoid ambiguity, we will refer to them as *threads*) similar to SICStus MT. However, there are a couple of important differences.

Thread are not created dynamically, but "pseudo-static". This means that the execution of a CS-Prolog program is divided into two phases, the *prelude phase* and the *working phase*. Threads can only be created during the prelude phase. The prelude phase is terminated by a call to a predicate `start_processes/0`, which causes all threads which have been created to start executing. This has the effect that before any threads can start, the application needs to determine the number of threads it will need. This is a quite serious drawback, since it would, for example, prevent a WWW-server from creating a thread for each accepted connection.

5.3.4.1 The Communication Mechanism

CS-Prolog provides two different kinds of threads—*self-driven* and *event-driven*. The self-driven process is the normal case. Its purpose is to execute a goal (in parallel) and terminate when it is done, just like SICStus MT threads. Event-driven (or *real-time*) are designed in order to continuously respond to events. When an event-driven thread is created, the programmer supplies two goals: one for initialization and one for handling events. The latter will be restarted for each incoming event. In other words, CS-Prolog has integrated support for event-driven applications.

Like SICStus MT, CS-Prolog communicates between threads by sending messages. However, there are two major differences. The first is that CS-Prolog uses *channels* as medium. These channels need to be created separately and are not bi-directional. This gives flexibility on one hand: it is easy to generate broadcasts since many threads can read on the same channel. On the other hand, the programmer needs to create all the channels explicitly; it is not possible to send a message to a thread given only the thread's identifier.

The second difference is that the message passing is *synchronous* (and *buffered*). This means that when a message is sent to a channel, the sending thread will be suspended until the receiving thread is ready to accept the message. This tends to cause deadlocks more frequently

and does not match the concept of “independent” threads very well since threads will be suspended depending on the state of the thread they send messages to.

5.3.4.2 Scheduling

CS-Prolog supports physical concurrency, i.e. it is possible to utilize multiple-processor architectures. Threads are automatically distributed as evenly as possible. If there are more threads than processors, some or all processors will be time-shared as in the single-processor case. For single-processor scheduling, CS-Prolog behaves similarly to SICStus MT with the difference that CS-Prolog threads do not have priorities, so the algorithm is a simple Round-Robin with a configurable time-slice which defaults to 2 seconds.

An interesting feature of CS-Prolog is that it has a built-in deadlock detection mechanism (DDM). By keeping track on the current state of all threads, it can detect global deadlocks. This happens when all threads are suspended on communication points (send, receive, etc) and there are no more threads to execute. When the DDM detects a global deadlock it signals a runtime error in the main-thread.

A DDM should certainly be considered for inclusion in a released version of SICStus MT. The problem is how to act when a deadlock is discovered. Since SICStus MT really does not have a “main thread” (the top-level loop is not sufficiently “special”), we cannot imitate CS-Prolog’s solution.

5.4 Performance

5.4.1 Game of Life

We have measured the performance of SICStus MT on two benchmarks. The first one is *Game of Life* [26]. It is a little program that simulates a primitive biological environment (such as a microorganism culture or a student party). The environment consists of a board of size *n* by *m* cells. A cell can either be alive (containing a living organism) or dead (no organism). Their initial state is determined either randomly or using a preset pattern before the game starts. Time is counted in generations and the game is played one generation at a time. For every generation, the state of each cell (alive or dead) is calculated by counting the number of alive neighbors in the previous generation. The new state of the organism can be seen in Table 3.

Sum	New State	Explanation
0-1	dead	Dies of loneliness
2	unchanged	Happy organism
3	alive	3 neighbors creates a new organism
4-8	dead	Dies of overcrowding

Table 3: State transitions for Conway’s Game of Life

The game itself is very fascinating, and there are many initial patterns which display intriguing behavior. However, this is not the interesting part. The interesting aspect of Conway's Game of Life from our perspective is the fact that implemented in a certain way, it becomes an excellent benchmark for measuring thread scheduling overheads and the efficiency of synchronizing and communicating between threads. Our special implementation uses one thread for each cell and all communication between cells is done by using the message passing mechanism. It is easily

realized that it quickly becomes a lot of threads and even more messages; the ideal situation for a benchmark. The source code for the Prolog implementation of this benchmark can be seen in Section 9.1. Table 4 shows some figures, measured on a Sun Microsystems 248 MHz UltraSPARC.

Since the implementation heavily relies on the usage on threads and it would be difficult to implement a comparable version *without* using threads, this benchmark has only been used to compare different multithreaded execution environments.

Language/Implementation	Time (ms)	Factor
SICStus MT	10000	1.00
ERLANG	3100	0.31
Oz 2.0	820	0.082
Java (JIT compiled)	17000 / 29000	1.7 / 2.7

Table 4: Execution times for Conway's Game of Life. The parameters were 10x10 cells and 500 generations. Times are in milliseconds.

The efficiency of Oz 2.0—12 times faster than SICStus MT—is partly due to the fact that it does not communicate nor synchronize using message passing. Instead of sending messages between the threads to inform the neighbors about state changes, the Oz 2.0 version reads the state from the thread's data structure explicitly. This is possible since threads in Oz 2.0 uses a shared heap from which all of them can read / write. See Figure 18.

```
proc {Cell G CurrentS FinalS History NW N NE W E SW S SE}
   case G == 0 then
      FinalS = CurrentS
      History = nil
   else
      NextS Hr
   in
      % Read the state of the neighbors and calculate next state.
      % No message passing!
      NextS = {NewState CurrentS NW.1+N.1+NE.1+W.1+E.1+SW.1+S.1+SE.1}
      History = NextS|Hr
      {Cell G-1 NextS FinalS Hr NW.2 N.2 NE.2 W.2 E.2 SW.2 S.2 SE.2}
   end
end
```

Figure 18: The inner-loop of the Oz 2.0 version of Game-of-Life. Note the absence of message passing.

As mentioned earlier, ERLANG has the possibility of compiling receive statements very efficiently.

The Java benchmark was implemented in two different ways. The first was implemented using the same structure as the Oz 2.0 implementation; i.e. it does not use the message passing mechanism to communicate and synchronize. The second mimicked the Prolog / ERLANG implementation by communicating and synchronizing using message passing. Still, the message passing did not include copying between heaps as in Prolog and ERLANG, but by sending simple references to objects. Even so, the benchmark rendered much higher execution times for the Java version[2] than for the others. It seems that Java has a very inefficient thread scheduling

[2] Using Sun's own Java implementation.

mechanism, but since the internals of the JVM were not available for us to examine, we can not really say anything other than that the Java benchmark was far slower than could be expected. An interesting note is that the JVM showed (practically) no difference between interpreted and JIT compiled code, indicating that it is the scheduling mechanism of the JVM itself which is inefficient.

Comparing lazy switching and eager switching (see Section 3.2.2) using the Game-of-Life benchmark showed—as can be expected—that lazy switching is more efficient (approximately 700 ms or 7% in the example above). This is caused by the fact that lazy switching decreases the number of messages which are unpacked in vain (i.e. messages which do not match the first argument to the call to `receive/1`).

5.4.2 Matrix Arithmetic

While the Game of Life benchmark was written to compare different multithreaded execution environments, the Matrix Arithmetic benchmark was written to compare SICStus MT with single-threaded SICStus.

The benchmark is very simple; it multiplies two matrices. The single-threaded version multiplies them straight away, i.e. takes one cell at a time and traverses the corresponding row and column by standard Prolog list-traversal. The multithreaded version, however, spawns a thread for each cell, and each thread then performs the same action as the top-level thread did for each cell in the single-threaded implementation. The source code for this benchmark can be seen in Section 9.2.

Size	ST	MT	Factor
10*10	30	60	2.0
20*20	170	350	2.1
30*30	500	900	1.8

Table 5: Execution times in milliseconds for the Matrix Arithmetic benchmark. Observe that the number of threads increase quadratically.

As we can see, in the multithreaded version the execution times are roughly doubled compared to the single-threaded version. We can also see that the factor does not increase at all when we increase the number of threads, indicating good scalability.

5.4.3 Profiling Data

By profiling the SICStus emulator, we can obtain figures for how much time is spent doing different things. The data in Figure 19 and Figure 20 was obtained by running the Game-of-Life benchmark on a board of 10 by 10 cells and iterating 500 generations.

```
 %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 33.28      6.64     6.64        2  3320.00  6776.64  wam
  [...]
  3.71     13.19     0.74   342876     0.00     0.00  compile_term
  3.71     13.93     0.74      166     4.46     4.46  SP_bcopy
  3.01     14.53     0.60   955264     0.00     0.00  cunify_args
  2.31     14.99     0.46  2706151     0.00     0.00  cunify
  2.21     15.43     0.44   955129     0.00     0.00  get_instance
  2.06     15.84     0.41      164     2.50     7.53  heap_overflow
  1.70     16.18     0.34   342876     0.00     0.00  ct_mark_phase
  1.50     16.48     0.30       44     6.82     6.82  _open
  1.35     16.75     0.27  1009674     0.00     0.00  prolog_receive
  1.10     16.97     0.22   740871     0.00     0.00  htLookUp
  1.05     17.18     0.21   704709     0.00     0.00  checkalloc
  1.00     17.38     0.20      332     0.60     0.60  _ioctl
  1.00     17.58     0.20      134     1.49     1.49  _write
  0.90     17.76     0.18   687733     0.00     0.00  checkdealloc
  0.65     17.89     0.13   562348     0.00     0.00  clInsert
  0.65     18.02     0.13   562346     0.00     0.00  clRemove
```

Figure 19: Profile data obtain from the Game-of-Life benchmark (10x10, 500 generations), using `gprof`.

```
-----------------------------------------------
                0.06    1.20  342663/342663      msgCreate [8]
[9]      6.4    0.06    1.20  342663         set_compound_data [9]
                0.74    0.46  342663/342876      compile_term [11]
-----------------------------------------------
...
-----------------------------------------------
[ uninteresting calls here]
                0.00    0.00     706/2706151     bu3_functor [159]
                0.13    0.00  739931/2706151     prolog_self [31]
                0.33    0.00 1964545/2706151     prolog_receive [13]
[20]     2.3    0.46    0.00 2706151         cunify [20]
                0.00    0.00      13/955264      cunify_args [16]
-----------------------------------------------
...
-----------------------------------------------
                0.00    0.00     258/955129      wam [3]
                0.44    0.00  954871/955129      prolog_receive [13]
[21]     2.2    0.44    0.00  955129         get_instance [21]
-----------------------------------------------
```

Figure 20: Extract from the call graph data obtained from `gprof()`.

These figures[3] are quite informative. They can, for example, tell us that

- Apart from the function `wam()`, most of the time is spent doing the following: compiling and packing up terms (`compile_term()`, `get_instance()`), unifying terms in general (`unify_args()`, `unify()`), and expanding/shrinking stacks (`SP_bcopy()`). However, the stack expansion probably depends on badly chosen initial stack values and not on excessive stack usage.
- All calls to `get_instance()` are made by `receive/1` (i.e. `prolog_receive`).
- Most calls to `cunify()` were made by `receive/1` and `self/1`. However, using `cunify()` in `self/1` is a little overkill and can be optimized.

Figure 20 displays parts of the call graph which can tell us from where the functions were called. By analyzing this information, we can see that the unification algorithm is called nearly 2 million times in order to match terms in the input-queue. We can also see that messages are un-

[3] A little note on the percentages. Among the functions removed from figure Figure 20 is the profiler main-function `internal_mcount()`, which is responsible for roughly 25% of the execution time. The "true" percentage can therefore be calculated from the raw percentage by dividing by 0.75. As a result, the true percentage of for example packing and unpacking terms is (3.71% + 2.21%)/0.75 = 7.9%.

packed roughly 1 million times (`get_instance()`), and packed about 350,000 times (`set_compound_data()`). This means that each message is unpacked on average 3 times.

These figures can also give us a hint of how much speedup the special unification algorithm described in Section 6.2 would give. This algorithm would eliminate the unnecessary (i.e. two-thirds of it) unpacking. If we assume that the new unification algorithm is as fast as the existing ones this means that the speedup will be 2.21%/(0.75 * 3/2) = 2.0%, not a very large number.

5.4.4 Raw Overhead

An interesting performance issue is how large the *raw overhead* is. By raw overhead we mean the overhead associated with *supporting* multiple threads, not the overhead associated with executing two or more threads in parallel. More concretely it is the difference in execution time between a program executed on single-threaded SICStus and the same program executed on SICStus MT. To measure this, we executed the *nreverse* benchmark on a 100-element list 2000 times (for accuracy). First *with* support for multiple threads and then, using preprocessor switches, *without* support for multiple threads.

The results were a little strange. Executing *with* support for threads gave consistently lower execution times than without. Not much, around 0.5% (150 ms for a total of about 27500 ms), but very consistently. This should not be possible, since if thread-support is switched off, there is less code to be executed. The probable reason for this is that the branch-prediction of the UltraSPARC can eliminate the overhead of checking if there are any threads to schedule. In any case we know that the raw overhead for supporting multiple thread is very low.

5.5 Conclusion

There is a considerable overhead in the message passing mechanism, mostly caused by the out-of-order receive mechanism. The total number of messages sent in the benchmark was only a third of the total number of unpackings which means that each message is unpacked nearly three times. Not only are the messages unpacked in vain, the (that itself is not a very big part of the overhead), but the receiving thread has to be rescheduled and allowed to execute in order to determine if the message was the one expected or not. By using indexing (as described in Section 6.2.2) it should be possible to eliminate all the unnecessary unpackings and it should also be possible to eliminate the resulting useless scheduling of threads blocked on `receive/1`, since the indexer should be able to directly determine whether or not a message was the expected one and thereby only wake up the receiving thread when the correct message arrives.

By implementing the improvements described in Section Chapter 6 - , we expect SICStus MT to perform comparably to the ERLANG implementation with respect to the Game-of-Life benchmark.

Chapter 6 - Future Work

This chapter describes some areas which should be pursued in the future development of SICStus MT.

6.1 Critical Regions and Database Synchronization

The programming interface outlined in Section Chapter 3 - must be extended to include some form of critical regions, for example to support synchronized access to the Prolog database. Currently, database transactions such as `assert/1-2` are performed atomically. The atomicity comes from the fact that threads cannot be interrupted inside a C function—threads are only switched in and out at the synchronizing point, so all C-predicates are *guaranteed* to be executed atomically.

However, it is desirable to be able to atomically execute larger parts of a Prolog program than a single statement. Take a complex database-update as an example. It may consist of an initial test of some condition followed by a call to `assert/1` which should only be performed if the initial test succeeded. After the test is done, it is vital that no other thread is allowed to interrupt since that could make the test invalid.

In the current implementation, there is no mechanism to do this unless we fall back on the message passing mechanism, but that would cause unnecessary overhead in sending messages between the threads (both in extra lines of code and in execution time). Also, the situation of critical regions and database synchronization occurs frequently enough to indulge it with a separate solution. An example on how this could look in the Prolog code is displayed in Figure 21. A discussion about process synchronization in general can be found in [16].

```
transaction(Arg1,Arg2,Arg3) :-
    assert(Arg1),
    assert(Arg2),
    retract(Arg3).

run :-
    ... ,
    lock(transaction(foo,bar,frotz),L),
    ... .
```

Figure 21: Example of a synchronized database operation.

In this example, the predicate `transaction/3` is executed atomically using the predicate `lock/2` and the lock denoted `L`. `lock/3` works similarly to `call/1`. The lock `L` is created somewhere and distributed to all threads which wants to execute the critical region.

6.1.1 Semantics for Synchronized Database Operations

In Figure 21 the predicate `transaction/3` is executed *atomically* with respect to the lock `L`. This means that if another thread attempts to call `lock(...,L)` before the first thread's call to `lock/2` (and thereby to `transaction/3`) has completed, it will be suspended and then released when the first thread has completed its call. The lock maintains a list of threads which are sus-

pended on the lock. These threads will be allowed to enter the critical region in a FIFO-manner. In the literature, locks of this kind are called *monitors* [27, 28] since they make sure that only one thread is inside the critical region at any time and thereby can be said to "monitor" the critical region. Java's `synchronized`-construct is implemented using monitors (see [15], p. 38).

There are a couple of important points to make here. First, the lock needs to be *reentrant*. This means that when `lock/2` executes, it examines the lock and suspends only if someone else has it (i.e. has used it in a call to `lock/2`). It does not suspend if the lock is owned by the executing thread. If locks are not reentrant, it becomes very hairy to use nested atomic transactions, since nested calls to `lock/2` using the same lock would then suspend and cause a deadlock. Second, we need to make sure that `lock/2` behaves correctly when the atomic transaction fails or raises an exception. This will cause backtracking and we must therefore make it possible for `lock/2` to be undone, i.e. it must release the lock.

The third point is also concerned with backtracking and what happens if the call to `lock/2` is backtracked into, i.e. if the program fails after the call to `lock/2` has completed. Here the choice of semantics is not obvious. Assume that there are choicepoints pushed inside the transaction. Should they be taken into account and thereby enabling backtracking into `lock/2`? If so, we would have to make sure that no one else has entered in the meanwhile and suspend if so.

Since the `lock/2` predicate basically is a form of meta-call where we are guaranteed that we will execute the predicate atomically with respect to the lock `L`, it is desirable that `lock/2` has the same model of execution as, for example, `call/1`. In terms of locking, this means that the lock is acquired at call/redo and released at exit/fail/exception.

6.2 The Message Passing Mechanism

In any communication-intensive application, the message-passing mechanism is a potential bottle-neck, so also in SICStus MT. This section describes two possible improvements.

6.2.1 Avoiding Message Copying

One possible solution is to create a special unification implementation which is capable of unifying a term on a heap with a compiled term. This would get rid of the overhead of constantly unpacking the message, however the message would still need to be compiled (copied), which is not the case with the solution above. The unification necessary would not cause any overhead, since it would need to be done anyway, even if the terms were located on the same heap. This is a perfectly reasonable solution, and will probably be implemented later on. See Section 5.4.3.

6.2.2 Indexing

Recall that when the out-of-order receive mechanism is used, the entire message queue needs to be searched each time `receive/1` is called; the message queue must store the messages which cannot be passed directly to the user. This has the effect that the time complexity for receiving messages is linear in the number of "currently undeliverable" messages.

Fortunately, the process of searching the message queue for terms unifiable with the argument of `receive/1` is essentially equivalent to the search done during standard query resolution: searching the Prolog database for matching clauses. This is a central issue in the WAM and it has been the subject of extensive research [29, 30].

The key mechanism used in searching the Prolog database for matching clauses is called *indexing*. Basically, it finds and excludes those clauses which cannot possibly result in a solution. The difference between this and actual query resolution is that indexing only examines the head of the clause. If all clauses except one can be excluded then the predicate becomes deterministic and no choicepoint needs to be pushed. Of course, since only the head of the clause is

examined, only those clauses which differ in the head can possibly be distinguished. Due to efficiency reasons, indexing is often only performed on the principal functor, but there are other algorithms which are more efficient (see [29]).

The core of the indexing mechanism is, given an arbitrary term, to search a set of clause-heads (i.e. terms) and produce a subset of these which are unifiable with the given term. This can be utilized in the receive-mechanism in order to find messages which are to be passed to the user. The message queue then corresponds to the Prolog database and the term passed to `receive/1` corresponds to the query.

By using indexing, we do not need to do a linear search and we can thereby decrease the time spent in searching the message queue. However, the actual improvement depends on the efficiency of the indexing algorithm and how well it performs on the set of messages which the application sends between its threads, meaning that the programmer can improve the efficiency of the indexing mechanism by sending "suitable" messages which are easy to index on.

6.3 Improved Syntax

Comparing Figure 13 and Figure 14 we see that the Prolog code is considerably less clear and more verbose than the ERLANG code. This is mainly due to the fact that we have no syntax support for the synchronization and communication primitives; they have to be implemented as ordinary predicates with no special syntax. See Figure 22 for a suggestion of how the syntax for receive constructs could look like.

```
thread :-
   receive( Template,          % Incoming term must match Template
          ( Guard1 -> Body1    % If the Guard (a goal) succeeds,
          ; Guard2 -> Body2    % the corresponding body is executed.
          ; ...
          ; GuardN -> BodyN
          ),
          Timeout, Body ).     % If no term is received within 'Timeout'
                               % millisecs, execute 'Body' and return.
    ...
```

Figure 22: Suggestion for improved syntax for receive constructs.

The improved syntax enables us to specify in a very dynamic way the order in which terms are received and we are no longer restricted to unification against a single term.

The semantics and eventually necessary restrictions of the new `receive/4`-predicate must of course be thoroughly investigated. For example, should it be possible to call `receive/4` from inside a guard?

6.4 Improvements Under the Hood

6.4.1 Blocking System Calls in Foreign Code

In Section 4.5 we outline a possible solution for supporting blocking system calls in foreign code. This solution, however, has two serious drawbacks. The first is how to get back to the emulator when the system call suspends. The current solution is not transparent to the user. Explicit jackets needs to be inserted, since the C code in the external libraries has full freedom to perform any low level system call, and the user needs to manually make sure to return the `SUSPEND`-code all the way up to the emulator.

The second problem is how to redo the system call. The emulator would want to be able to resume the library call immediately before the system call which caused the blocking, as if the library call was never interrupted. This would require that the entire execution environment of the library call was saved.

The first problem is possible to solve fairly transparently by using the standard C functions `setjmp()` and `longjmp()` to obtain some sort of escape mechanism. This would eliminate the manual stack unwinding, but not the explicit jackets. The second problem is not solvable at all (within the limits of the C language) since it requires that the execution environment (especially the C-stack) can be saved and restored again when the system call should be performed again.

The most attractive solution is to allow the program to do this manually, i.e. checking the system call if it was about to block, and return `SUSPEND` to the emulator.

6.4.2 The Prolog/C Interface

The interface between Prolog and C has to be properly extended in a released version of SICStus MT. It has to be extended to fully support the proposed solution for blocking system calls in foreign code outlined in Section 4.5. The proposed solution is quite feasible, but it has to be complemented with a deeper look into the different forms of blocking system calls and their characteristics to make sure that they are supported. Similar support is needed for the Prolog/C interface support functions, such as `SP_fgetc()` (see [9], p. 159). These must be able to suspend themselves in a compatible manner. It might be desirable to extend the Prolog/C interface to supply a similar functionality as the Prolog programming interface described in section Chapter 3 - does, such as sending terms to other threads. Generally, the Prolog/C interface needs to be thoroughly tested with respect to multiple threads, reentrancy, and blocking system calls.

6.4.3 Runtime vs. Development Systems

This implementation does not support stand-alone applications (also called Runtime Systems, see Section 2.4.1), but only the development system. However, the intention is that the distinction between runtime and development systems should be eliminated and we have therefore not spent any resources on implementing support for both.

6.4.4 Retracted Clauses

There is also a problem with removing retracted clauses. The problem has to do with the algorithm used to determine if a retracted clause can be physically erased [31]. The current algorithm searches the choicepoint stack for references to the retracted clause (since it can be activated due to backtracking). The current implementation has no support for this and it is therefore possible that a retracted clause is removed too early.

6.4.5 Access-control for Sub-threads

When running the system as a development system with a top-level loop, restrictions must be enforced on sub-threads and what they are allowed to do. For example, it might be unsuitable to allow a sub-thread to terminate the entire process by calling `abort/0`. Simply forbidding sub-threads to use certain predicates is not always suitable. In some cases, it might desirable to implement alternative semantics for some of these; for example, `halt/0` might be restricted to terminate only the currently executing thread (and therefore be equivalent to `self(X),kill(X)`).

Other predicates which need attention on this issue are `reinitialize/0`, `save/1`, `restore/1`, and `load_foreign_resource/1`.

Chapter 7 - Related Work

This chapter will describe some related work done in the area of parallelism and concurrency in Prolog and some references for further reading.

A survey of the basic issues and an outline of some existing Prolog implementation which support concurrency / parallelism in different ways can be found in [18]. A comparison of SICStus MT with the languages ERLANG [19], Java [25, 24], Oz 2.0 [23], and CS-Prolog Professional [45] is provided in Section 5.3.

The AND / OR-parallel Prolog systems [32, 33, 34, 18] are worth a separate mention. AND / OR parallelism exploit the inherent parallelism in Prolog programs by utilizing shared-memory MIMD architectures and creating processes as a result of conjunctions or disjunctions in the Prolog code. This kind of parallelism is also called *goal parallelism*, as opposed to the *process parallelism* which this thesis is focused around. Goal parallelism is *implicit*, i.e. the parallelism is exploited without any special predicates and can therefore preserve full Prolog semantics.

PAN [20] is a process based Prolog system built on SICStus Prolog which uses multiple SICStus processes which are statically created at startup as opposed to the light-weight Prolog threads employed in SICStus MT. *PMS-Prolog* [36] is a parallel Prolog system which also uses the message-passing paradigm for inter-thread / process communication. *Multi-Prolog* [37] and *BlackLog* [38] are parallel Prolog systems which use the blackboard paradigm for inter-thread / process communication. *IC-Prolog II* [39, 40] is a multithreaded Prolog system developed at Imperial College, London. It supports multithreaded Prolog similar to the implementation described in this work.

Chapter 8 - Conclusion

We have shown that it is possible to support multiple threads of execution in SICStus Prolog. The threads are light-weight, dynamically managed using a small and compact Prolog interface, and implemented entirely at user-level.

The implementation has a very low raw overhead. This means that the overhead of executing single-threaded code (i.e. a regular non-threaded Prolog program) on SICStus MT is very low which makes it very reasonable to include support for multiple threads in a released version of SICStus. The implementation maintains full native code support. This is important since a considerable part of SICStus efficiency stems from its being able to execute Prolog code compiled to native code. The implementation does have a few problems, mainly concerned with memory consumption together with small address spaces and the efficiency of the message passing mechanism. There are solutions or suggestions for improvements for most of these problems, even if they can not all be solved.

Future work in the area of SICStus MT include support for native threads, support for critical regions and database synchronization, ensuring portability and improved syntax.

Chapter 9 - Program Listings

9.1 Game Of Life

The following code is the Prolog code for the Game Of Life benchmark. It also serves as a good example on how threaded Prolog code might look like.

```
% Document type:  -*- Prolog -*-
% Filename:   /a/if/home1/jojo/xjobb/misc/threadtest/life.pl
% Author:         Jesper Jonsson <jojo@sics.se>
% Last-Update:    Time-stamp: <1997-05-07 1728 jojo>

:-
            use_module(library(random)),
            use_module(library(lists)).


%%%%%%%
%
% life( +N, +M, +NumGens, +Mode )
%
% Description:
% Implements Conway's Game Of Life. The game is run in a MxN-matrix with
% Num generations with one thread per cell. The game is entirely built on
% communication between thread and therefore a very good benchmark for
% a multithreaded environment.
%
% The source is highly inspired by Johan Montelius' <jm@sics.se> Erlang
% version.
%


life(N, M, Num) :-
            M >= 1, N >= 1,
            statistics(runtime, _),
            matrix(N, M, Matrix),  % Create matrix of cells
            !,                     % Don't redo spawns
            link(Matrix),                       % Inform each cell about its neighbors
            start(Matrix, Num),    % Start the game
            Size is N * M,
            wait_cells( Size, Sum ),
            statistics(runtime, [_,Runtime]),
            format( "Done. Surviving organisms = ~w.~n", [Sum]),
            format( "Runtime = ~w.~n", [Runtime]).

life(_, _, _) :-
            format( "Matrix too small. Minimum = 2x2.~n", [] ),
            !,
            fail.

%%%%%%%
%
% matrix( +N, +M, -Matrix )
%
% Description: Creates a N by M matrix of threads, one for each cell.
% The matrix is organized as a list of lists of thread-identifiers.
%
% [ [ <first row> ],
%   [ <second row> ] ]
%

matrix(N, M, [Dummy|Matrix]) :-
            DLen is M + 2,
            length( Dummy, DLen ),
            matrix0(N, M, Matrix).

matrix0(0, M, [Dummy]) :-
            DLen is M + 2,
```

```
            length( Dummy, DLen ).
matrix0(N, M, [Line|Matrix]) :-
            make_line(M, N, Line),
            N0 is N - 1,
            matrix0(N0, M, Matrix).

% Aux: make_line/2 creates a line in the matrix.
make_line(M, N, [_|Line]) :-
            make_line0( M, N, Line ).

make_line0(0, _, [_]).

make_line0(M, N, [CellThread|Line]) :-
            spawn(cell, CellThread),          % Spawn cell-thread
            M0 is M - 1,
            make_line0(M0, N, Line).

%%%%%%%
%
% link( +Matrix )
%
% Description: Links the matrix together by informing each cell about its neighbors.
%

link([_,_]).

link([ North | Rest ]) :-
            Rest = [This, South |_],
            link_line( North, This, South ),
            link( Rest ).

link_line( [NW | RestN], [W | RestW], [SW | RestS] ) :-
            RestN = [N, NE | _ ],
            RestW = [This, E | _ ],
            RestS = [S, SE | _ ],
            Neighbors = [NW, N, NE, W, E, SW, S, SE],
            remove_noncells( Neighbors, RealNeighbors ),
            send( This, RealNeighbors ),
            link_line( RestN, RestW, RestS ).

link_line( [_,_], _, _ ).

remove_noncells( [], [] ).
remove_noncells( [Neighbor|Neighbors], [Neighbor|RealNeighbors] ) :-
            nonvar( Neighbor ),
            remove_noncells( Neighbors, RealNeighbors ).
remove_noncells( [Neighbor|Neighbors], RealNeighbors ) :-
            var( Neighbor ),
            remove_noncells( Neighbors, RealNeighbors ).

%%%%%%%
%
% start( +Matrix, +NumGens )
%
% Description: Send the term go( MasterThread, NumGens ) to each thread
%

start( [], _ ).
start( [Line|Matrix], NumGens ) :-
            start_line( Line, NumGens ),
            start( Matrix, NumGens ).

% Aux: start_line/2 starts all threads in a list.
start_line( [], _ ).
start_line( [Thread|Line], NumGens ) :-
            self( This ),
            ( nonvar( Thread ) ->
                random( 0, 2, X ),
                send( Thread, go( This, NumGens, X ))
            ;
                true
            ),
            start_line( Line, NumGens ).



%%%%%%%
%
% wait_cells( +NumCells, -Sum )
%
% Description:
%
```

```
wait_cells( NumCells, Sum ) :-
            wait_cells( NumCells, Sum, 0 ).

wait_cells( 0, Sum, Sum ).
wait_cells( NumCells, Sum, SumAck ) :-
            receive( final( Final )),
            NumCells0 is NumCells - 1,
            SumAck0 is SumAck + Final,
            wait_cells( NumCells0, Sum, SumAck0 ).



%%%%%%%
%
% cell
%
% Description: The cell-thread.
%

cell :-
            receive(Xs),                        % Receive list of neighbors.
            receive(go( MasterThread, NumGens, Initial )), % wait for start-signal.
            cell_iter( NumGens, Xs, Initial, Final ),
            send( MasterThread, final( Final )).

%%%%%%%
%
% cell_iter( Neighbors, MasterThread, NumGens, Initial )
%
% Description: The iteration-loop of the cell-thread.
%

cell_iter( 0, Neighbors, State, State ) :-
            !,
            send_nghs( Neighbors, State ).

cell_iter( NumGens, Neighbors, State, Final ) :-
            send_nghs( Neighbors, State ),
            receive_nghs( Neighbors, NeighborStates ),
            reproduce( NeighborStates, State, NewState ),
            NumGens0 is NumGens - 1,
            cell_iter( NumGens0, Neighbors, NewState, Final ).

send_nghs( [], _ ).
send_nghs( [Neighbor|Neighbors], State ) :-
            self( This ),
            send( Neighbor, state( This, State )),
            send_nghs( Neighbors, State ).

receive_nghs( [], [] ).
receive_nghs( [Neighbor | Neighbors], [NState | NStates ] ) :-
            receive(state( Neighbor, NState )),
            receive_nghs( Neighbors, NStates ).

reproduce( NeighborStates, State, NewState ) :-
            sum_list( NeighborStates, Sum ),
            repro( Sum, State, NewState ).

repro( 0, _, 0 ).
repro( 1, _, 0 ).
repro( 2, State, State ).
repro( 3, _, 1 ).
repro( 4, _, 0 ).
repro( 5, _, 0 ).
repro( 6, _, 0 ).
repro( 7, _, 0 ).
repro( 8, _, 0 ).
```

9.2 Matrix Arithmetic

The following code is the Prolog code for the multithreaded Matrix Arithmetic benchmark.

```
% Document type:  -*- Prolog -*-
% Filename:  /amd/home/jojo/src/prolog/matrix.pl
% Author:         Jesper Jonsson <jojo@sics.se>
% Last-Update:    Time-stamp: <1997-08-15 1304 jojo>

:-
            use_module(library(random)),
            use_module(library(lists)).

maximum(42).

matrix(M,N) :-
            statistics(runtime,_),
            make_matrix(Matrix1,M,N),
            make_matrix(Matrix2,M,N),                    % Actually transposed
            spawn_multipliers(Matrix1,Matrix2),
            MN is N * M,
            wait_for_results(MN),
            statistics(runtime,[_,Runtime]),
            format("Time consumed: ~w~n", [Runtime]).

make_matrix(Matrix1,M,N) :-
            length(Matrix1,M),
            fill_matrix(Matrix1,N).

fill_matrix([],_).
fill_matrix([First|Rest],N) :-
            maximum(Max),
            randseq(N,Max,First),
            fill_matrix(Rest,N).


wait_for_results(0).
wait_for_results(Total) :-
            receive(Result),
            T0 is Total - 1,
            wait_for_results(T0).


spawn_multipliers([],_).
spawn_multipliers([Row|Rest],Matrix2) :-
            spawn_multipliers_2(Matrix2,Row),
            spawn_multipliers(Rest,Matrix2).

spawn_multipliers_2([],_).
spawn_multipliers_2([Col|Cols],Row) :-
            self(Self),
            spawn(multiplier(Row,Col,Self),_),
            spawn_multipliers_2(Cols,Row).

multiply_rows([],[],0).
multiply_rows([RowElem|RowElems],[ColElem|ColElems],Result) :-
            multiply_rows(RowElems,ColElems,Result0),
            Result is Result0 + (RowElem * ColElem).

multiplier(Row,Col,Parent) :-
            multiply_rows(Row,Col,Result),
            send(Parent,result(Row,Col,Result)).
```

Chapter 10 - References

1 Leon Sterling and Ehud Shapiro. *The Art of Prolog*. MIT Press, 1986.

2 Robert W. Sebesta. *Concepts of Programming Languages*. Addison-Wesley, third edition, 1996.

3 The Free On-Line Dictionary of Computing. URL: *http://wagner.princeton.edu/foldoc/*.

4 Bil Lewis and Daniel J. Berg. *Threads Primer—A Guide To Multithreaded Programming*. Prentice Hall, 1996.

5 Kevin Dowd. *High Performance Computing*. O'Reilly & Associates, Inc., 1993.

6 Benjamin Gamsa. Region-Oriented Memory Management in Shared-Memory NUMA Multiprocessors. Master's thesis, Department of Computer Science, University of Toronto, October 1992.

7 Christoph Koppe. NUMA architectures and user level scheduling—a short introduction. URL: *http://www4.informatik.uni-erlangen.de/ELiTE/numa_ult_intro.html*, 1996.

8 Thomas E. Andersson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler Activations: Efficient Kernel Support for the User-level Management of Parallellism. In *Proceedings of the 13th ACM Symposium on Operating System Principles*, pages 95-109, 1991.

9 Mats Carlsson, Johan Widén, Johan Andersson, Stefan Andersson, Kent Boortz, Hans Nilsson, and Thomas Sjöland. SICStus Prolog User's Manual. SICS Technical Report T91:15, Swedish Institute of Computer Science, June 1995. Release 3 #0.

10 Mats Carlsson. The SICStus Emulator. SICS technical report T91:15, Swedish Institute of Computer Science, 1991.

11 Hassan Aït-Kaci. *Warren's Abstract Machine—A Tutorial Reconstruction*. MIT Press, 1991.

12 David H. D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, 1983.

13 L. Kleinrock. *Queueing Systems, Volume II: Computer Applications*. Wiley-Interscience, 1975.

14 B. W. Lampson. A scheduling philosophy for multiprocessing systems. *Communications of the ACM*, 11(5):347-360, May 1968.

15 David Flanagan. *Java in a Nutshell*. O'Reilly & Associates, second edition, 1997.

16 Abraham Silberschatz and Peter B. Galvin. *Operating Systems Concepts*. Addison-Wesley, fourth edition, 1994.

17 Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice-Hall, 1992.

18 Koen De Bosschere. Process-based parallel logic programming: A survey of the basic issues. In Bosschere et al. 41].

19 Joe Armstrong, Robert Virding, Claes Wiström, and Mike Williams. *Concurrent Programming In ERLANG*. Prentice Hall, second edition, 1996.

20 Hamish Taylor. Design of a resolution multiprocessor for the parallel virtual machine. In Bosschere et al. [41].

21 George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems—Concepts And Design*. Addison-Wesley, second edition, 1994.

22 John P. Mulligan. The Unofficial Guide To Solaris 2.x, Online Manual Pages. URL: *http://www.lafayette.edu/cgi-bin/mulligaj/rtfm*.

23 Seif Haridi. *A Tutorial of Oz 2.0*. Swedish Institute of Computer Science, 1996.

24 Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, September 1996.

25 James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, September 1996.

26 John Horton. Computer Recreations. *Scientific American*, March 1984.

27 P. Brinch Hansen. *Operating System Principles*. Prentice-Hall, New Jersey, 1973.

28 C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549-557, October 1974.

29 R. Ramesh, I.V. Ramakrishnan, and D.S. Warren. Automata-Driven Indexing of Prolog Clauses. In *Proceedings of the Principles of Programming Languages*, 1990.

30 S. Dawson, C. R. Ramakrishnan, I. V. Ramakrishnan, K. Sagonas, S. Skiena, T. Swift, and D. S. Warren. Unification factoring for efficient execution of logic programs. In *Papers of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 247-258, 1995.

31 T. Lindholm and R. A. O'Keefe. Efficient implementation of a defensible semantics for dynamic PROLOG code. In Lassez 42], pages 21-39.

32 Mats Carlsson. *Design and Implementation of an OR-Parallel Prolog Engine*. SICS Dissertation Series 02, The Royal Institute of Technology, 1990.

33 J. Conery. *The AND/OR Process Model for Parallel Interpretation of Logic Programs*. PhD thesis, University of California at Irvine, 1983.

34 Roland Karlsson. *An High Performance OR-Parallel Prolog System*. SICS Dissertation Series 07, The Royal Institute of Technology, 1992.

35 S. Ferenczi and I. Futo. CS-Prolog: A Communicating Sequential Prolog. In P. Kacsuk and M. J. Wise, editors, *Implementations of Distributed Prolog*, pages 357-378. John Wiley, 1992.

36 M. J. Wise, D. G. Jones, and T. Hintz. PMS-Prolog: A Distributed, Coarse-grain-parallel Prolog with Processes, Modules and Streams. In P. Kacsuk and M. J. Wise, editors, *Implementations of Distributed Prolog*, pages 379-404. John Wiley, 1992.

37 K. De Bosschere and J.-M. Jacquet. Multi-Prolog: Definition, Operational Semantics and Implementation. In D. S. Warren, editor, *Proceedings of the ICLP'93 conference*, pages 299-313, Budapest, Hungary, June 1993. The MIT Press.

38 D. G. Schwartz. *Cooperating Heterogenous Systems: A Blackboard-based Meta Approach*. PhD thesis, Department of Computer Engineering and Science, Case Western Reserve University, 1993.

39 Yannis Cosmadopoulos and Damian A. Chu. *IC Prolog II version 0.92 Reference Manual*. London, 1992.

40 Damian Chu. I.C. Prolog II: a Multi-threaded Prolog System. In Evan Tick and Giancarlo Succi, editors, *ICLP-Workshops on Implementations of Logic Programming Systems*, pages 17-34. Kluwer Academic Publishers, 1993.

41 Koen De Bosschere, Jean-Marie Jacquet, and Antonio Brogi, editors. *ICLP94 Post-Conference Workshop on Process-Based Parallel Logic Programming*, June 1994.

42 Jean-Louis Lassez, editor. *Proceedings of the Fourth International Conference on Logic Programming*, MIT Press Series in Logic Programming, Melbourne, 1987. The MIT Press.

43 Neil D. Jones, Carsten K. Gomard, Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

44 Kent Boortz. *SICStus maskinkodskompilering*. SICS Technical Report T91:13, Swedish Institute of Computer Science, August 1991.

45 *CS-Prolog Professional User's Manual*. Version 1.1. ML Consulting and Computing Ltd, Applied Logic Laboratory, Budapest, Hungary, 1997.

46 R.C. Haygood. *Native code compilation in SICStus Prolog*. In Proceedings of the Eleventh International Conference of Logic Programming, MIT Press Series in Logic Programming, 1994.

47 D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and systems*, 7(1):80–112, January 1989.