SICStus MT—A Multithreaded Execution Environment for SICStus Prolog

Jesper Eskilson and Mats Carlsson

Intelligent Systems Laboratory
Swedish Institute of Computer Science
Box 1263, SE-164 29 Kista, Sweden
Email: jojo@sics.se, matsc@sics.se
Phone: +46-8-7521500, Fax: +46-8-7517230

Abstract. The development of intelligent software agents and other complex applications which continuously interact with their environments has been one of the reasons why explicit concurrency has become a necessity in a modern Prolog system today. Such applications need 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.
This paper describes the design, implementation and evaluation of a prototype multithreaded execution environment for SICStus Prolog. The threads are dynamically managed using a small and compact set of Prolog primitives implemented in a portable way, requiring almost no support from the underlying operating system.

Keywords: logic programming, implementation, multithreading, abstract machines

1 Introduction

Prolog has historically been centered around the concept of asking queries to a database and receiving answers and other output on a terminal or other output device. This has proven to be an excellent way of providing many software solutions.

But computer applications have become more and more complex, involving many independent and different subproblems. For example, a WWW-server normally contains a part which continuously listens for connections on sockets and a word-processor with on-the-fly spell-check has a part which continuously scans the spelling dictionary for the words which are typed in. This is usually done by employing multiple *threads of execution*, or just *threads*. Doing it without threads forces the programmer to manually switch between executing the different subproblems, making the program inefficient and difficult to write and maintain.

Now, threads is not a new concept in Prolog or in other languages. Most modern languages support threads in some way. Most notable are C [22, 18], ERLANG [1], Java [14], and Oz 2.0 [30, 31]. Many logic programming systems

implement threads (although they are mostly referred to as "processes"), such as CS-Prolog [15], KL1 [8], IC Prolog II [10], Multi-Prolog [4], but these are all relatively small and experimental systems. There are no widely-used commercial Prolog systems supporting threads (to the best of our knowledge). EMRM [32] includes a multi-client WWW-server based on Aurora [23], an or-parallel extension to SICStus Prolog. Amzi! Prolog + Logic Server supports multiple threads, but not in the same Prolog engine; they have to be coded using C or Java threads.

Summing up, multithreading is a very useful language extension for developing and maintaining large software systems, such as WWW-servers and intelligent software agents. This paper presents the design, implementation and evaluation of such an extension to SICStus Prolog [7]. Different options are discussed for the following design issues: native threads, scheduling algorithm, time quanta and thread switching, programming interface, communication and synchronization, and blocking system calls. A preliminary performance analysis is given.

The rest of the paper is organized as follows: We state the requirements of our prototype in Section 2. We then describe the important design issues (Section 3, 4, 5) and their impact in the implementation. Section 6 describes the programming interface and Section 7 describes the solution to the problem of blocking system calls. Sections 9 and 10 evaluate the performance of our implementation and include references to related work. We end with some conclusions.

2 Requirements

The purpose of this work has primarily been to prove that support for multiple threads is realistic in an industrial-strength Prolog system, executing *full* Prolog. There are no restrictions, for example, on backtracking as in committed-choice languages. Also, threads performs separate and independent computations, not to be confused with *parallel* Prolog, where threads (or processes as they are sometimes referred to) cooperate to perform the same computation in parallel.

To prove that multiple threads are realistic under these requirements, we must show that introducing support for multiple threads does not cause any significant inconvenience for the user such as reduced execution speed or excessive memory consumption. In other words, the prototype must provide the benefits of multiple threads without removing any (or as little as possible) of the benefits of *not* supporting threads.

3 Native Threads

The first design issue which needs to be settled is whether to use native threads. Native threads have two major benefits. The first, and perhaps most important, is that they make it possible to utilize multiple CPUs (and other machine- and operating-system specific features). The second is related to blocking system calls and will be discussed in Section 7.

We have chosen not to use native threads at all in this first version of SICStus MT, for a couple of reasons. First, native threads are not available on all platforms. If this prototype relies on native threads we restrict our results to those platforms with native thread support. Second, native threads are not crucial in order to prove the feasibility of threads in a industrial-strength Prolog. Third, native threads can be implemented later on if desirable; avoiding native threads in the prototype does not rule them out completely. See also Section 11.

4 Necessary Modifications

The two basic characteristics of any language implementation are its storage and execution models. In order to support multithreading in SICStus, changes are needed to both of these.

4.1 Storage Model

The storage model need to be modified so that some data-areas are kept private to each thread. There are four data-areas in a WAM-based [2] emulator:

The Static Area contains a variety of objects, such as interpreted and compiled clauses, atoms, indexing tables, and so on.
The Local Stack contains procedure frames and choice point records.
The Global Stack contains Prolog terms. This is usually the largest area.
The Trail Stack contains conditional variable bindings, i.e. variables that should be reset to unbound upon backtracking.

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

The bulk of the static area is kept global in order for threads to be able to share code. The local and trail stacks must be kept private, since they are directly related to how the program is executed. The same 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 data-structure representing a thread.

In a WAM-based emulator, all three stacks shrink on backtracking. Thus, if threads perform independent computations, the global stack too must be kept private. This unfortunately incurs an overhead on communication between threads (see Section 6.1), as messages need to be copied between stacks. It is conceivable to have a design that avoids copying, with a shared global stack that does not shrink except on garbage collection, but that would represent a major departure from the WAM and is out of scope of this work.

4.2 Execution Model

Like the storage model, the execution model needs to be modified in order to support multiple threads. Since native threads are not utilized, the execution model must support time-sharing of the emulator between the different threads (see also Section 7.2).

The main 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 *the synchronization point*. A natural candidate for the synchronization point is the *event-handler*, which handles certain kinds of asynchronous events, such as stack overflows, goals woken by variable bindings, and goals invoked by interrupts.

The benefit of using the event-handler as the location of the synchronization point is that the execution state is fully well-defined, so that a context switch between two threads is trivially implemented by exchanging references to the WS.

5 Scheduling

Scheduling [21] 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.

The main requirements of the scheduler, apart from having a low scheduling overhead, is that it should be *preemptive*. Without preemption, it is impossible to write applications which use threads to perform independent work simultaneously (like serving two clients at the same time).

The algorithm used in SICStus MT is a variant of Round-Robin scheduling [29, 33], called Priority Round-Robin (PRR). PRR scheduling is conducted like Round-Robin scheduling with the difference that Round-Robin is only practiced among threads with equal priority. Among threads with different priority, the priority determines which thread is scheduled for execution.

This algorithm is far from perfect. The main disadvantage is that it is not *fair* (a *fair* scheduling mechanism guarantees that threads will not starve, regardless of the number of threads waiting to run). If a thread decides to increase its priority above everybody else and then initiate a lenghty calculation, all other threads will starve for sure.

On the other hand, by not having priorities, threads with important tasks will have to wait for threads with very low priority tasks. For example, a spell-checker in a word-processor would naturally run with very low priority in order to avoid stealing resources from the more important task of handling user input and displaying it on the screen. Without priorities, the spell-checker would take up as much computing resources as the actual word-processor.

Summing up, PRR does not guarantee fairness, but it is robust, easy to implement, and provides good performance in most cases.

5.1 Choice of Time Quantum

The size of the time quantum is crucial to (P)RR scheduling. The time-quantum is defined as the maximum period of time a thread is allowed to execute before it

is interrupted by the scheduler. The size of this period has a large impact on the efficiency of the scheduling. If the period is too large, response times will become too long. If the time-quantum is allowed to be infinite, we lose the preemptive property of PRR. If the period is too small, the scheduling overhead will become too large.

We obtained a time quantum by using using a timer interrupt to generate an asynchronous signal, and then switch out the thread when it reaches the event-handler (as described in Section 4.2). If timer signals occurs every, say, 50 ms, the length of the time-quantum would then be 50 ms plus the time taken to reach the synchronization point. This will guarantee a finite time-quantum.

This solution has the drawback of relying on signals to ensure preemption. ANSI C does support timer signals [18], but only with low resolution (seconds). For non-ANSI C implementations which lack signals entirely, PRR cannot be implemented with guaranteed preemption. See Section 8 for a further discussion of the portability aspects of using timer signals.

It is also possible to base the time quantum on the number of executed WAM instructions. This has quite a large overhead (almost 20% in some cases), but it has a major benefit: it is possible to repeat the same execution sequence (for debugging purposes, for example). However, it would also require substantial modification to the native code kernel, which would be tedious to implement and would also degrade the performance of native code execution.

Another possibility is to base the time quantum on the number of executed procedures. This is more efficient than counting WAM-instructions and still supports repeatable execution sequences. This method is used by Oz and ERLANG [30, 31, 1].

6 Programming Interface

The following predicates are introduced in SICStus MT. The usage of each predicate is indicated by prefixing each argument X by `+`, denoting that X should be instantiated to a non-variable in any call; `-`, denoting that X should be unbound; or `?`, denoting no restrictions on X.

`spawn(+Goal, -ThreadID)` Launches a new, independent computation in a separate thread. Thus, the declarative semantics of this predicate is "true". The new thread will execute the thread `Goal`. `ThreadID` will be bound to the identifier of the new thread. Also, a message-port is created for *send/receive* operations. See Section 6.1.

`send(+ThreadID, +Term)` Sends `Term` to the thread indicated by `ThreadID`. This predicate always succeeds (or throws a domain error exception). `Term` will be inserted last in the receiving thread's input queue. Unbound variables are consistently copied, i.e. `f(X,X)` becomes `f(Y,Y)` on the other side.

`receive(?Term)` Extracts the first element in the thread's input queue that is unifiable with `Term` and unifies it with `Term`. If no such term exists, the thread is suspended.

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

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

Fig. 1. Example of using send/receive. The example spawns a simple echo thread which lies in the background and echos everything sent to it

`self(-ThreadID)` `ThreadID` is the thread identifier of the running thread.

`kill(+ThreadID)` Causes `ThreadID` to terminate. Always succeeds, even when the thread does not exist or has died, i.e. the semantics is that after the call to `kill/1`, the specified thread is dead, regardless of its state before the call.

`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` but could exceed this limit depending on two factors; the frequency of the timer-interrupts and the number of threads waiting to run. See Section 8.

See Figure 1 and 2 for examples of how to use these predicates. Figure 1 is a trivial example while Figure 2 is a little more complex example on how to code the predicate `join/2` using SICStus MT.

6.1 Communication and Synchronization

The model of communication and synchronization is *message-based* as opposed to *blackboard-based* [5, 34], also known as *tuple space based* [16]. This means that the communication is based on sending explicit messages as opposed to using a shared store of some kind. A message can be any kind of Prolog term. Unbound variables are allowed in messages, but they will be renamed.

Furthermore, the message-port is *anonymous*, i.e. it is an integral part of the thread and addressed using the thread's identifier. Messages are addressed by direct naming, but only of the destination thread; the source thread is not specified, allowing a server, for example, to receive messages from unnamed clients. Direct naming is not as flexible as indirect communication using named *channels* [9]. More specifically, channels allow many-to-many communications which

```
spawn_joinable(Goal, JoinableThreadID) :-
    self(Self),
    spawn(joinable_wrapper(Goal, Self), JoinableThreadID).

joinable_wrapper(Goal,Parent) :-
    self(Self),
    ( on_exception(Exception,Goal,
                   joinable_handler(Exception,Parent,Self)) ->
        send(Parent,Self -> success(Goal))
    ;   send(Parent,Self -> fail)
    ).

joinable_handler(Exception,Parent,Self) :-
    send(Parent, Self -> exception(Exception)).

join(ThreadID,Result) :-
    receive(ThreadID -> Result).

run(Goal) :-
    spawn_joinable(Goal,ThreadID),
    ...
    join(ThreadID,Result),
    format('~w has completed. Result = ~w.~n', [Goal,Result]).
```

Fig. 2. Example of how to implement `join/2` using SICStus MT. The semantics of `join/2` is to suspend until the specified thread has completed. This implementation catches exceptions which are thrown in the thread and also sends back goal-term with eventual variable bindings.

direct naming does not. The main reason for using direct naming in the prototype was to keep the implementation simple, and this might well be reconsidered in a released version of SICStus MT.

Oz and Gypsy [30, 17], are examples of language implementations using channels. PLITS and ERLANG [13, 1] is an example that uses direct naming and allows the receiver to leave out the sender address.

The communication mechanism is *asynchronous*. This means that the sender does not need to wait until the receiver is ready to receive the message, i.e. the *send*-primitive is *non-blocking*. This means also that the mechanism is buffered, i.e. the communication media (the input 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.

Due to our design choices, messages must be copied when sending them between threads. The absence of a shared heap causes at least one copying to

be done. Full Prolog (with unrestricted backtracking) prevents us from copying directly to/from another thread's heap, so we must copy the message twice, via the static area.

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 9 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.

Message Non-determinism Recall that `receive/1` extracts the first *matching* message. The makes it possible to specify which messages to accept for a particular call to `receive/1`. This is also referred to as *message non-determinism* [5] and is also used in ERLANG.

In the Game-of-Life benchmark, described in Section 9, there is a construction which relies on this particular feature. 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 this 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 such call.

However, there are performance issues worth discussing here. Each time the receiving thread tries to execute a `receive(Term)` goal (either the first time around, or upon a thread switch in case it was suspended), it has to traverse the input queue, unpack each message, delete it from the queue if it unifies with `Term`, and otherwise keep searching. Unpacked messages that do not unify are reclaimed by Prolog's backtracking and so must be unpacked again the next time around. Even though the time to unpack a message is linear in its size, a given message may get re-examined many times. Also, message non-determinism will result in a list of "currently unmatched" messages, i.e. messages that have arrived but are not unifiable with the argument to `receive/1`. This means that messages can be delayed in the input queue for a potentially indefinite period of time. See Section 9 for some performance figures.

7 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 [33, 12].

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.

7.1 Possible Solutions

There are 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 its complexity. Each system call which might block must be preceded by a piece of code (called *jacket*) 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. Another drawback is the additional overhead of the jacket code.

The cavalier approach relies on the possibility of performing system calls without blocking the process at all, commonly known as *asynchronous I/O*. Instead of trying to determine 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 can be retried (as a result of a `SIGIO` signal indicating I/O completion).

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. The drawback is that is relies on the availability of asynchronous I/O (see Section 8).

We have not made any measurements of the performance penalty of these mechanisms (i.e. what the overhead is when threads are not used at all), but we do not believe that it is significant.

7.2 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. The only action required by the emulator is to immediately jump to the synchronization point in order to perform a thread switch.

Native Code SICStus Prolog compiles Prolog code to native code on some platforms [3, 19]. Thus 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 were a regular C function. The purpose of this is, of course, execution speed, and speedups of 3-4 times are typical.

The multithreaded execution model maintains full compatibility with native code execution, since the thread scheduling mechanism is built upon an already existing mechanism—the event handler—for which the native kernel already has support. The scheduler raises an asynchronous event which will cause the native code kernel to automatically escape back to the emulator, jump to the event-handler and thereby reschedule. When the thread later on is scheduled again, the native code execution will continue as normal. See also [27].

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 in the development system 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 ready-list is empty, it suspends the process by calling `pause()`. Suspending and resuming processes is—as one would expect—platform-dependent. We will only describe the procedure used under Solaris; we expect the procedure to be fairly similar on most modern operating systems. See also Section 8.

The process is then resumed (i.e. `pause()` returns) when a signal is received. The signal is triggered by one of two reasons. Either the asynchronous 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 time-quantum 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 which case we will simply be suspended again.

8 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.

Asynchronous I/O relies on the use of signal `SIGPOLL` to indicate I/O completion. This signal is not ANSI-C, but is included in POSIX and thus available on most UNIX dialects. The most notable exception is the Win32 platforms. However, on those platforms there are other (Win32-specific) ways of performing asynchronous I/O which mimic the use of `SIGIO/SIGPOLL`.

The same applies to the predicate `wait/1` (Section 6) and the mechanism for suspending the emulator (Section 7.2). They both use POSIX extensions which have counterparts in the Win32 interface.

Implementing SICStus MT in a bare ANSI-C environment is certainly possible, but it would mean some restrictions in functionality: Time-quantums cannot be specified with higher resolution than seconds, the `wait/1` predicate will have reduced accuracy, and blocking system calls will suspend the entire process.

9 Performance Evaluation

We briefly evaluate the space complexity and execution speed of our implementation. We have used two benchmark programs: *Game-of-Life* [20], a simulation of a simple biological environment, and a matrix multiplication program. The purpose of the former benchmark is to measure message-passing overheads, whereas the latter benchmark gives an idea of the intrinsic scheduling overheads.

9.1 Memory Consumption

Naturally, the objective during the implementation process has been to keep the threads as lightweight as possible. It is quickly realized that the space occupied by the WS is negligible compared to the stackconsumption.

Related to this is the fact that the address space in SICStus is only 2^{28} = 256 Mb (on a 32-bit architecture) since the 4 upper bits are used for tagging data-cells. This not only means that the data-areas cannot exceed 256 Mb, it also means that they have to be located at optimal places for this to be possible. In other words, if the address space becomes fragmented (by stack-shifting, for example), the limit of 256 Mb will drop even further. The impact of this limit will become clear in the next section.

9.2 Execution Speed

The figures in the following section have been obtained by running SICStus MT executing emulated code on a Sun UltraSPARC 248 MHz (unless otherwise noted).

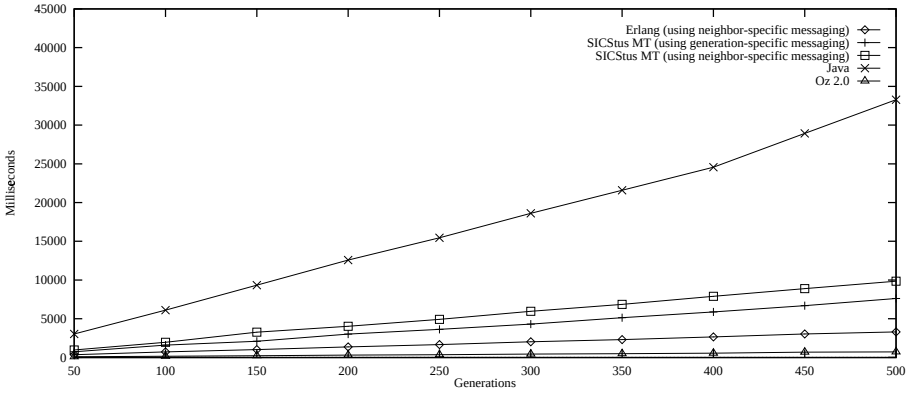

Fig. 3. Execution times for the Game-of-Life benchmark. The board is 10 × 10 cells.

Game-of-Life This is a variant of Conway's Game Of Life [20], a simulation of a simple biological environment. Basically, it consists of a matrix of cells, each of which may or may not contain an organism. There are rules for how the live cells reproduce for each generation and, depending on the initial population, many interesting and fascinating patterns occur. We have implemented the game in a way which makes it an excellent benchmark for measuring the efficiency of employing several (hundreds or even thousands) threads. By spawning a separate thread for each cell in the matrix and using the inter-thread communication mechanism to propagate information, we can get good information about the scheduling and communications overhead. See Figure 3.

The Oz 2.0 benchmark is not as relevant as the other three since it does not use the message passing mechanism to propagate information. Since Oz 2.0 has a shared heap (see Section 4.1), the threads can simply read the state of their neighbors directly from the heap. This eliminates two overheads: suspension overhead while sending and receiving messages and message copying overhead.

The Java-implementation (Sun's JDK 1.1.4) displays surprisingly bad performance, despite the fact that no message copying is done. We also found that there is also only a very small difference between JIT-compiled Java and byte-interpreted Java, which leads us to believe that it is the Java scheduler which is inefficient.

Comparing lazy switching and eager switching (see Section 6.1) using the Game-of-Life benchmark showed that lazy switching is more efficient. The difference was approximately 700 ms or 7%, measured on neighbor-specific messaging (the square markers in Figure 3). The difference is caused by the fact that lazy switching decreases the number of messages that are unpacked in vain (i.e. messages that do not match the first argument of the call to `receive/1`).

There is a considerable overhead in the message passing mechanism. The overhead can be divided into two parts. The first part is caused by having to copy messages between heaps. This is not a very big overhead, especially when

the messages are small. The dominating overhead is caused by the message nondeterminism (see Section 6.1). This mechanism allows the receiving thread to specificy which messages should be received. The problem is that the sending thread has no way of finding out if a message is "appropriate", i.e. if it matches the first argument to `receive/1`. Instead, it must always resume the receiving thread so it can make the decision itself. This is where *generation-specific* and *neighbor-specific* messaging come in. The former means that messages are only identified by their generation and the latter means that messages are identified by the sending cell (or thread). Neighbor-specific messaging causes more messages to arrive "out-of-order". This induces a higher overhead in the parsing of the message queue which can clearly be seen in the figure.

This overhead can be reduced by being more selective about when to resume threads blocked on `receive/1`, so we minimize the number of threads which are resumed in vain. Lazy switching is one way of doing this. Another way is to allow the *sending* thread to inspect the first argument to `receive/1` (in the receiving thread, that is) and avoid resuming threads when the message does not match. This should enable us to eliminate all unnecessary scheduling of threads blocked on `receive/1`.

Another improvement is to utilize indexing [26] to search the input queue. The queue is currently searched linearly, but by using indexing, the search would become considerably more efficient. The actual improvement, however, depends heavily on the application. If the average length of the input queues is large (which is the case if there are many messages arriving out of order), this improvement will have larger impact than if the queues are mostly empty.

Yet another, more orthogonal way of reducing this overhead is to introduce named channels (see Section 6.1). Named channels can be used to reduce the amount of out-of-order messaging, since messages of a certain kind can be sent to a dedicated channel.

Matrix Arithmetic Since our implementation of the Game-of-Life benchmark relies on the existence of threads, it is difficult to use that benchmark to get a grip on the overhead of SICStus MT compared with single-threaded SICStus.

To get some figures on this, we used a very simple matrix multiplication benchmark which can operate in two different modes, *threaded* and *meta-called*. The first spawns a thread for each cell in order to compute its value and the second simply performs a meta-call (using `call/1`), thereby not spawning any threads. These two modes correspond to the *Parallel MT* and *Sequential MT* lines in the graph in Figure 4, which were obtained using SICStus MT. The third line is obtained by running the matrix benchmark in sequential mode but using plain SICStus (version 3#5).

While the difference between lazy and eager switching was quite small in the Game-of-Life benchmark, it made a considerable difference here. The main thread receives a messages from each thread when it has completed, and eager switching caused the main thread to start executing for each such message while lazy switching only let the main thread execute once all multiplier threads had

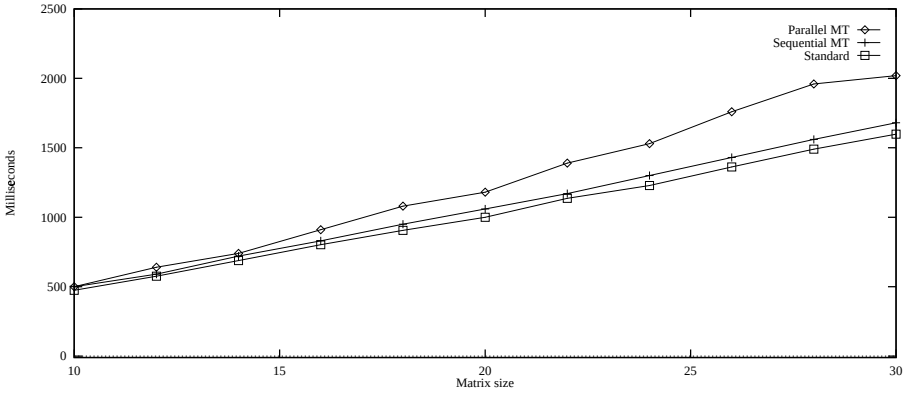

Fig. 4. Execution times for multiplying a matrix using different version of SICStus.

completed. The difference was large enough (roughly a factor 2 compared to Sequential MT) to exclude eager switching from an interesting comparison.

The data in Figure 4 tell us that the overhead of supporting threads is very small (the benchmark had to be executed several times in order to obtain a reliable difference). This means that it is reasonable to include support for multiple threads in a released version of SICStus.

The benchmark also shows that the overhead of spawning a thread as opposed to doing a meta-call is reasonably small. Also, the overhead of the meta-call itself (not shown in the graph) was insignificant; we did not observe a significant difference between the meta-called version and the standard sequential version.

Performance Conclusion Added up, it is reasonable to include support for multiple threads in a released version of SICStus, but the message passing overheads needs to be reduced in order to be competitive with implementations such as ERLANG and Oz 2.0.

10 Related Work

A great deal of work on parallel execution of Prolog has been done during the past 15 years; see e.g. [11, 23]. However, this work has mostly been concerned with exploiting the parallelism which is implicit in logic programs.

Several concurrent logic programming system based on explicitly creating processes (or threads) have evolved during the last 15 years. A survey of the general issues can be found in [5].

CS-Prolog Professional [24] supports multiple processes and uses the message-passing paradigm for communication using explicitly created channels as media. Asynchronous message passing is not supported; instead a rendezvous

(or hand-shake) model is used. However, forcing threads to synchronize whenever communication takes place tends to encourage deadlocks and is generally restrictive to the programmer.

KL1 [8] is a concurrent logic programming language based on *GHC* (Guarded Horn Clauses). The concurrency is similar to that of SICStus MT; threads are managed explicitly and communication is done using blocking variable bindings.

Multi-prolog [4] and *BlackLog* [28] use the blackboard paradigm for communicating between processes. Blackboard-based systems are inherently less scalable in applications where there is heavy communication between processes since all messages need to pass through one single point. On the other hand, blackboards tend to give the programmer more expressiveness since messages do not need to be sent to an explicit destination.

11 Future Work

11.1 Implementing Native Threads

The perhaps most interesting area for future work is the incorporation of native threads. As mentioned in section 3, native threads are not used at all in this prototype implementation. However, the benefits of using native threads (multiple CPU utilization, blocking systems calls, etc.) are important enough to justify native threads in a released version of SICStus. Incorporating native threads raises a couple of design issues itself.

First, should native threads replace the emulated threads of the current design? The answer to this question is "no". Not all platforms have multiple CPUs and there is a certain amount overhead with using native threads; both in the handling of the native threads themselves but also with the additional cost of needing to synchronize accesses to global data in the emulator. So, in some cases it might well be justified (from a performance point of view) to avoid native threads to some extent.

Second, how should the underlying native threads implementation be chosen to minimize portability problems? Are Pthreads (POSIX threads) [25] suitable for our purposes? This question is probably the most difficult one to give a good answer to. There are many different native threads implementations with different properties regarding portability, user-space vs. kernel-space, scheduling, etc.

Answering the first question with "no", raises another question: How do we map Prolog threads onto native threads? Should it be automatic or user-controlled? This question is still open for discussion. Since native threads do use resources in the operating system, using too many will degrade overall system performance. On the other hand, a Prolog application (such as Game-of-life) might want to create a very large number of threads. In such a situation, it is necessary to strike a balance between the number of native threads used and how the Prolog threads should map onto these. This dynamic scenario makes it unlikely that a static mapping—i.e. a Prolog thread is attached to a specific

native thread, for example when calling `spawn/2`, and is thereafter fixed to that native thread—is suitable. A dynamic mapping where Prolog threads could be dynamically scheduled on the existing native threads could more easily adopt to different user-needs and different platforms.

11.2 Debugger

Debugging a multithreaded application is not an easy task. However, we are not really talking about debugging Prolog code (for which SICStus has very good support), but rather about debugging the concurrency introduced by multithreading. Extending the existing debugger with the basic support for debugging multithreaded code is not that difficult; the problem is the support for more advanced features such as deadlock detection, debugging race conditions, etc.

12 Conclusion

We have presented the design and implementation of SICStus MT, a multithreaded extension to SICStus Prolog. The following design issues were discussed in detail: whether to use native threads, the scheduling algorithm, time quanta and thread switching, programming interface, communication and synchronization, and blocking system calls. We gave a preliminary performance analysis, indicating minimal scheduling overheads, but significant message passing overheads in the implemented prototype. Work is under way to reduce these overheads.

References

1. Joe Armstrong, Robert Virding, Claes Wiström, and Mike Williams. *Concurrent Programming In Erlang*. Prentice Hall, second edition, 1996.
2. Hassan Aït-Kaci. *Warren's Abstract Machine—A Tutorial Reconstruction*. MIT Press, 1991.
3. Kent Boortz. SICStus maskinkodskompilering. SICS Technical Report T91:13, Swedish Institute of Computer Science, August 1991.
4. 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.
5. Koen De Bosschere. Process-based parallel logic programming: A survey of the basic issues. In Bosschere et al. [6].
6. Koen De Bosschere, Jean-Marie Jacquet, and Antonio Brogi, editors. *ICLP94 Post-Conference Workshop on Process-Based Parallel Logic Programming*, June 1994.
7. 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.
8. Takashi Chikayama, Tetsuro Fujise, and Hiroshi Yashiro. A portable and reasonably efficient implementation of KL1. In Warren [35], page 833.

9. Randy Chow and Theodore Johnson. *Distributed Operating Systems & Algorithms*. Addison-Wesley, March 1997.
10. 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.
11. J. Conery. *The AND/OR Process Model for Parallel Interpretation of Logic Programs*. PhD thesis, University of California at Irvine, 1983.
12. George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems—Concepts And Design*. Addison-Wesley, second edition, 1994.
13. J. A. Feldman. High Level Programming for Distributed Computing. *Communications of the ACM*, 22(6):353–368, 1979.
14. David Flanagan. *Java in a Nutshell*. O'Reilly & Associates, second edition, 1997.
15. Iván Futó. Prolog with communicating processes: From T-Prolog to CSR-Prolog. In Warren [35], pages 3–17.
16. D. Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1989.
17. D. I. Good, R. M. Cohen, and J. Keeton-Williams. Principles of Proving Concurrent Programs in Gypsy. In *Proceedings of the Sixth ACM Symposium on Principles of Programming Languages*, pages 42–52, 1979.
18. Samuel P. Harbison and Guy L. Steele Jr. *C, A Reference Manual*. Prentice Hall, third edition, 1991.
19. R. C. Haygood. Native code compilation in SICStus Prolog. In *Proceedings of the Eleventh International Conference of Logic Programming*. The MIT Press, 1994.
20. John Horton. Computer Recreations. *Scientific American*, March 1984.
21. B. W. Lampson. A scheduling philosophy for multiprocessing systems. *Communications of the ACM*, 11(5):347–360, May 1968.
22. Bill Lewis and Daniel J. Berg. *Threads Primer—A Guide To Multithreaded Programming*. Prentice Hall, 1996.
23. Ewing Lusk, Ralph Butler, Terrence Disz, Robert Olson, Ross Overbeek, Rick Stevens, David H. D. Warren, Alan Calderwood, Péter Szeredi, Seif Haridi, Per Brand, Mats Carlsson, Andrzej Ciepielewski, and Bogumil Hausman. The Aurora or-parallel Prolog system. *New Generation Computing*, 7(2,3):243–271, 1990.
24. ML Consulting and Computing Ltd, Applied Logic Laboratory, Budapest, Hungary. *CS-Prolog Professional User's Manual, Version 1.1*, 1997.
25. Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads Programming*. O'Reilly & Associates, 1996.
26. R. Ramesh, I.V. Ramakrishnan, and D.S. Warren. Automata-Driven Indexing of Prolog Clauses. In *Proceedings of the Principles of Programming Languages*, 1990.
27. John H. Reppy. Asynchronous Signals in Standard ML. Technical Report 90-1144, Department of Computer Science, Cornell University, Ithaca, NY 14853, 1990.
28. D. G. Schwartz. *Cooperating Heterogenous Systems: A Blackboard-based Meta Approach*. PhD thesis, Department of Computer Engineering and Science, Case Western Reserve University, 1993.
29. Abraham Silberschatz and Peter B. Galvin. *Operating Systems Concepts*. Addison-Wesley, fourth edition, 1994.
30. Gert Smolka. The Oz programming model. In Jan van Leeuwen, editor, *Computer Science Today*, Lecture Notes in Computer Science, vol. 1000, pages 324–343. Springer-Verlag, Berlin, 1995.
31. Gert Smolka. Problem solving with constraints and programming. *ACM Computing Surveys*, 28(4es), December 1996. Electronic Section.

32. Péter Szeredi, Katalin Molnár, and Rob Scott. Serving multiple HTML clients from a Prolog application. In Paul Tarau, Andrew Davison, Koen de Bosschere, and Manuel Hermenegildo, editors, *Proceedings of the 1st Workshop on Logic Programming Tools for INTERNET Applications, in conjunction with JICSLP'96, Bonn, Germany*, pages 81–90. COMPULOG-NET, September 1996. Available from: `http://clement.info.umoncton.ca/~lpnet/lp-internet/archive.html`.
33. Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice-Hall, 1992.
34. Hamish Taylor. Design of a resolution multiprocessor for the parallel virtual machine. In Bosschere et al. [6].
35. David S. Warren, editor. *Proceedings of the Tenth International Conference on Logic Programming*, Budapest, Hungary, 1993. The MIT Press.