SICStus MT—A Multithreaded Execution Environment for SICStus Prolog

Jesper Eskilson Mats Carlsson
Intelligent Systems Laboratory
Swedish Institute of Computer Science
Box 1263, SE-164 29 Kista, Sweden
`jojo@sics.se`, `matsc@sics.se`

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.

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 consists of a part which continuously listens for connections on sockets. This is usually done by employing multiple *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 notably are C [16, 19], ERLANG [1], Java [11], and Oz 2.0 [23, 24]. Many logic programming systems implement threads (although they are mostly referred to as "processes"), such as CS-Prolog [12], KL1 [7], IC Prolog II [8], Multi-Prolog [4], but these are all relatively small and experimental systems. There are no (to the best of our knowledge) widely-used commercial Prolog systems supporting threads.

Summing up, multithreading is a necessary language requirement in order to facilitate the development and maintenance of 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 [6]. Different options are discussed for the following design issues: whether to use 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: The necessary concepts are defined in Section 1.1. We then state the requirements of our prototype in Section 1.2. We then describe the important design issues (Section 2, 3, 4) and their impact in the implementation. Section 5 describes the programming interface and Section 6 describes the solution to the problem of blocking system calls. Sections 7 and 8 evaluate the performance of our implementation and include references to related work. We end with some conclusions.

1.1 Preliminaries

Since SICStus Prolog (like most other Prologs) is executed on a virtual machine–also referred to as *emulator*–it is important to realize where the multithreading comes in. This paper discusses the implementation of *Prolog threads*, i.e. where two or more predicates appear to be running in parallel. This should not be confused with using *native threads* where parts (functions) of the emulator appear to run in parallel by using a native threads package such as *Pthreads* [19]. See Section 2 and Section 9.

1.2 Requirements

The purpose of this work has primarily been to prove that support for multiple threads is realistic in an industrial-strength Prolog system. To prove this, we must show that introducing support for multiple threads does not cause any significant inconveniences 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.

2 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 6.

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. That is, avoiding native threads in the prototype does not rule them out completely. See also Section 9.

3 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.

3.1 Storage Model

The storage model need to be modified to that some data-areas are kept to each thread. There are five kinds of 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 is also called environment stack and contains procedure frames, which mainly consist of permanent variables (variables that survive predicate calls).

The Global Stack contains Prolog terms. This is usually the largest area.

The Choicepoint Stack contains the choicepoint records. These record contain information about what to do during backtracking.

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, choicepoint, and trail stack 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.

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), there are several complications. For example, it would be very difficult to support multiple threads backtracking at the same time.

3.2 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 and due to our requirements on the scheduler (see Section 4), the execution model must support time-sharing the emulator between the different threads.

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 *overflow-check*. This is a handler for certain kinds of asynchronous events, such as stack overflows, goals woken by variable bindings, and goals invocated by interrupts. The benefit of using the overflow-check 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.

4 Scheduling

Scheduling [15] 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 [22, 25], 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*[1]. However, it is robust, easy to implement, and provides good performance in most cases.

4.1 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 period is small, the scheduling overhead will become too large.

1. One possible way of obtaining a time quantum is setting up a timer interrupt and thereby rescheduling at a fixed-time interval (for example, 50 ms), rounded up to the next arrival at the synchronization point (see Section 3.2).
2. It is also possible to reschedule based on the number of executed WAM instructions. This has the benefit of not requiring any timer interrupts. It can also be used if the same execution sequence must be repeated (for debugging purposes, for example). The major problem is efficiency; the overhead turned out to be almost 20%. 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.
3. By reducing the granularity of WAM instruction counting and instead reschedule every nth arrival at the overflow-check, we obtain a reasonable time-quantum (empirical tests showed that the overflow-check occured spontaneously roughly with the same frequencies as a timer-interrupt set of 50 ms), and at the same time we are not bound to using timer-interrupts.

5 Programming Interface

The set of predicates used to create, destroy, and in other ways manipulate threads has deliberately been kept to a minimum in order to keep the implementation simple and the ideas clear. The following predicates are defined by SICStus MT.

`spawn(:Goal, -ThreadID)` Spawns 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.

`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 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.

---

[1] A *fair* scheduling mechanism guarantees that threads will not starve, regardless of the number of threads waiting to run.

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

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

`kill(+ThreadID)` Causes `ThreadID` to terminate. Always succeeds.

`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 Figure 1 for examples of how to use these predicates.

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

run :-
    spawn(reader, ReaderA),
    spawn(reader, ReaderB),
    send(ReaderA, readlock(ReaderB)),
    wait(120000),
    kill(ReaderA),
    kill(ReaderB).
```

Figure 1: Example of using the "send" and "receive" primitives. This example spawns two threads which take turns reading one term each from standard input. After 2 minutes, both threads are killed and the program terminates.

5.1 Communication and Synchronization

The model of communication and synchronization is *message-based* as opposed to *blackboard-based* [3, 26], also known as *tuple space based* [13]. This means that the communication is based on sending explicit messages as opposed to using a shared store of some kind.

In our solution, each thread has a *unidirectional* message port where it can receive messages, consisting of standard Prolog terms. 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. Unbound variables are allowed, but these will be duplicated.

The communication mechanism is *asynchronous*. This means that the sender does not need to wait until the receiver is ready to receive the message. 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.

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 7 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.

5.1.1 Receiving Messages "Out of Order"

The ERLANG implementation of the `send` and `receive` primitives features a very practical construction: the ability to specify which messages to accept for a particular call to `receive`. This is also referred to as *message non-determinism* [3].

In the Game-of-Life benchmark, described in Section 7, 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 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 call to `receive/1`.

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, unpacking each message, delete if from the queue if it unifies with `Term`, 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, out-of-order receives 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 7 for some performance figures.

6 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 [25, 10]. 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.

6.1 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 fact that most operating systems support some way of performing system calls without blocking the process at all, commonly 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 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.

6.2 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.

6.2.1 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 [2] by calling `pause()`. 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

---

[2] 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.

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.

7 Performance Evaluation

We briefly evaluate the space complexity and execution speed of our implementation. We have used two benchmark programs: *Game-of-Life* [14], 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.

7.1 Memory Consumption

Naturally, the objective during the implementation process has been to keep the threads as light-weight as possible. It is quickly realized that the space occupied by the WS is negligible compared to the stack-consumption. The WS is outweighed 64:1 if the global, local, choicepoint, and trail stacks are 32k, 16k, 8k, and 8k, respectively. These figures are the recommended ones. Setting them lower causes excessive stack-shifting and memory fragmentation.

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.

7.2 Execution Speed

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

7.2.1 Game-of-Life

This is a variant of Conway's Game Of Life [14], 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 2.

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. This eliminates the overhead of scheduling threads in order to receive messages (which turns out to be the major bottleneck in our implementation).

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

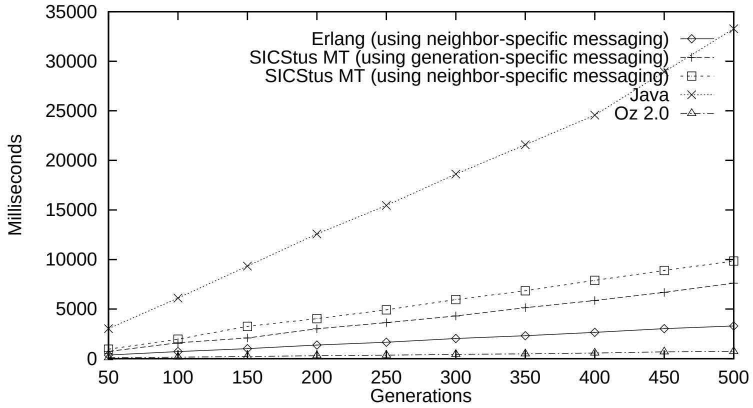

Figure 2: Execution times for the Game-of-Life benchmark. The board is 10 × 10 cells.

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 5.1) using the Game-of-Life benchmark showed that lazy switching is more efficient (approximately 700 ms or 7%). This 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 out-of-order