SPAM : a multiprocessor execution driven simulation Kernel

Alain Gefflaut, Philippe Joubert

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

SPAM: A Multiprocessor Execution Driven Simulation Kernel

Alain Gefflaut, Philippe Joubert

N° 1966

mars 1993

PROGRAMME 1

Architectures parallèles,
bases de données,
réseaux et systèmes distribués

ISSN 0249-6399

INRIA
RENNES

SPAM: A Multiprocessor Execution Driven Simulation Kernel

Alain Gefflaut, Philippe Joubert*

Programme 1 — Architectures parallèles, bases de données, réseaux
et systèmes distribués
Projet LSP

Rapport de recherche n ° 1966 — mars 1993 — 22 pages

Abstract: Trace driven simulation is a well known technique for performance evaluation of single processor computers. However, trace driven simulation introduces distortions when used to simulate multiprocessor architectures. Execution driven simulation is the only technique that gives accurate simulation results for multiprocessor architectures though it is difficult to implement.

This paper presents SPAM, a simulation kernel that simplifies the construction of execution driven simulators for shared memory multiprocessors. The kernel provides a tracing tool and a set of primitives which allow the execution, tracing and simulation of shared memory parallel applications on a single processor computer. The performance of the kernel allows the simulation of real sized parallel applications in a reasonnable time.

Key-words: execution driven simulation, parallel address traces, shared memory multiprocessors

(Résumé : tsvp)

*email :Alain.Gefflaut@irisa.fr, Philippe.Joubert@irisa.fr

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : (33) 99 84 71 00 – Télécopie : (33) 99 38 38 32

SPAM: un noyau de simulation coordonnée à l'exécution pour multiprocesseurs

Résumé : La simulation à partir de traces d'adresses est une technique classique pour l'évaluation des performances des monoprocesseurs. Toutefois, l'application de cette technique à l'étude des architectures multiprocesseurs donne des résultats imprécis. Seule la simulation coordonnée à l'exécution (*execution driven simulation*) donne des résultats précis pour ce type d'architecture mais est difficile à mettre en œuvre.

SPAM est un noyau de simulation qui simplifie la construction de simulateurs de multiprocesseurs à mémoire partagée. Le noyau propose un outil de traçage ainsi qu'un ensemble de primitives permettant l'exécution, le traçage et la simulation d'applications parallèles à mémoire partagée sur une machine monoprocesseur. Les performances du noyau de simulation sont telles qu'elles rendent possible la simulation d'applications parallèles réalistes dans un temps raisonnable.

Mots-clé : simulation coordonnée à l'exécution, traces d'adresses de programmes parallles, multiprocesseurs à mémoire partagée

1 Introduction

Simulation is one of the most widely used techniques for performance evaluation of computer systems. In the early design stage, simulation provides a cheap way to compare competing design solutions as well as the behaviour of a given design under varying workloads. Simulation is particularly useful for shared memory multiprocessors whose performance heavily depends on tiny interactions within the memory system of the architecture.

The classical implementation of an architectural simulator uses as input to the architectural model an address trace collected during a previous execution of a program. This trace driven simulation technique has been widely used for the study of cache memories on single processor systems. However trace driven simulation is not well suited to the simulation of parallel applications running on multiprocessor architectures because the execution path through the application, and hence the address trace to be simulated, depends on the target architecture. Execution driven simulation corrects this drawback by interleaving the execution and tracing of the parallel application with its simulation. Execution driven simulation controls the address trace generation to ensure that the trace corresponds to the one that would be obtained if that application was actually executed on the architecture being simulated. Unfortunately, execution driven simulation is much more complicated to implement than trace driven simulation because of the necessary interactions between the simulator and the simulated application.

SPAM is a simulation kernel and tracing tool which relieves architecture designers from the burden of implementing a complete execution driven simulator. Along with a simple parallel programming model for shared memory applications, SPAM offers a set of primitives to retrieve the address traces from the application processes and to enforce the correct execution order of the application under simulation.

The remainder of this paper is organised as follows. The next section examines the fundamental problems of simulating shared memory parallel architectures and introduces the execution driven simulation technique. Section 3 presents the programming model supported by the simulation kernel and section 4 describes how an execution driven simulator can be constructed with SPAM. Finally, section 5 details the current implementation and performance of the tracing tool and simulation kernel.

2 Simulation Principles

2.1 Trace driven Simulation

The traditional way of implementing an architectural simulator is to use *trace driven simulation*. This technique divides the simulation system into two independent components: an address generator and a memory/network timing simulator. The purpose of the address generator is to produce a stream of execution events, mostly addresses, which are of interest to the simulator. The simulator then emulates the target architecture to estimate the time taken to perform each event on the architecture under study. This partitioning into two distinct components allows the use a of wide variety of address collection methods. As a side effect, the same address trace can be used when simulating other hardware configurations or to study the influence of varying parameters.

Trace driven simulation gives accurate results when studying uniprocessor architectures because the address trace of a uniprocessor application does not depend on the simulated architecture. A given uniprocessor trace can hence be used to simulate any uniprocessor architecture. This is obviously not the case for multiprocessor architectures because the execution path through a parallel program is not necessarily the same on two different architectures and can even vary from one execution to another on the same architecture. The execution path through a parallel program is determined by the order of the accesses to synchronization objects. This order of accesses depends on the relative speeds of the processes constituting the program which are in turn determined by the characteristics of the architecture on which the parallel program is executed. Since the address traces of the different processes of a parallel program are determined by the execution path through the program, different executions generate different address traces.

Hence, a parallel address trace collected on a given architecture may not reflect the execution path that should be observed when executing the program on a different architecture. During the simulation, this trace may get out of phase with the execution that would be observed on the simulated architecture. This *trace-shifting* phenomenon [Dubois *et al.* 86] can heavily perturbate the simulation results.

Figure 1 presents a simple example of trace-shifting. The first process entering the critical section executes code sequence C1 whereas the second executes code sequence C2.

Assume that process P1 executes C1 and process P2 executes C2 when the trace is collected. Now let us assume that this trace is used to simulate a shared memory multiprocessor where each processor has a private cache. The size and the replacement policy of the caches can change the relative speed of the two processes, that is the speed at which their address traces are consu-

```
var int i init 0
var lock v

process P1 ::          process P2 ::
  lock(v)                lock(v)
  i := i + 1             i := i + 1
  if i = 1 then          if i = 1 then
    unlock(v)              unlock(v)
    C1                     C1
  else                   else
    unlock(v)              unlock(v)
    C2                     C2
  fi                     fi
end                    end
```

Figure 1: Example of indeterminism in a parallel application

med by the simulator. So, during the simulation, the event corresponding to the locking operation in the trace of process P2 may be consumed by the simulator before it has consumed the corresponding event in the trace of process P1. If this program was actually executed on the simulated architecture, process 2 should execute C1 whereas its trace contains the events relative to the execution of C2. Obviously, the trace is not correct for the simulated architecture. This problem can also occur for processes communicating by shared memory or by message passing.

Some trace driven simulators try to solve this problem by generating the address traces corresponding to all the possible executions of a parallel program [Holliday & Ellis 92]. During simulation, the simulator chooses the address trace to be used. Because of the potentially unbounded number of different executions, very few programs can be traced with this technique. Another alternative is to perform the simulation with a set of sequential programs that do not interact with each other [Chiang & Sohi 92]. This solution precludes the *trace-shifting* phenomena because each execution stream is independent but it is impossible to study the behaviour of parallel programs on parallel architectures with this technique.

2.2 Execution driven Simulation

Execution driven simulation [Covington *et al.* 88] is a simulation technique that corrects the drawbacks of trace driven simulation for parallel applications. The basic idea of execution driven simulation is to interleave the execution and tracing of the application with the simulation of the target architecture. Since the execution of the application is controled by its simulation, the address trace simulated is exactly the one that would be observed if the application was actually executed on the simulated machine.

The execution order of a parallel application is determined by the ordering of the operations which can modify the execution path through the program. The globally visible events are typically accesses to shared variables, message passing or synchronization operations. The order of these globally visible events must reflect the behaviour of the application processes on the simulated architecture and must therefore be computed by the simulator. In execution driven simulation, the execution of the application and the generation of the address traces are carried out simultaneously with the simulation of the target architecture The execution of the application is under the control of the simulator. As soon as an application process attempts to execute a globally visible operation, the execution of this process is stopped and the process waits for an acknowledgement from the simulator to resume its execution.

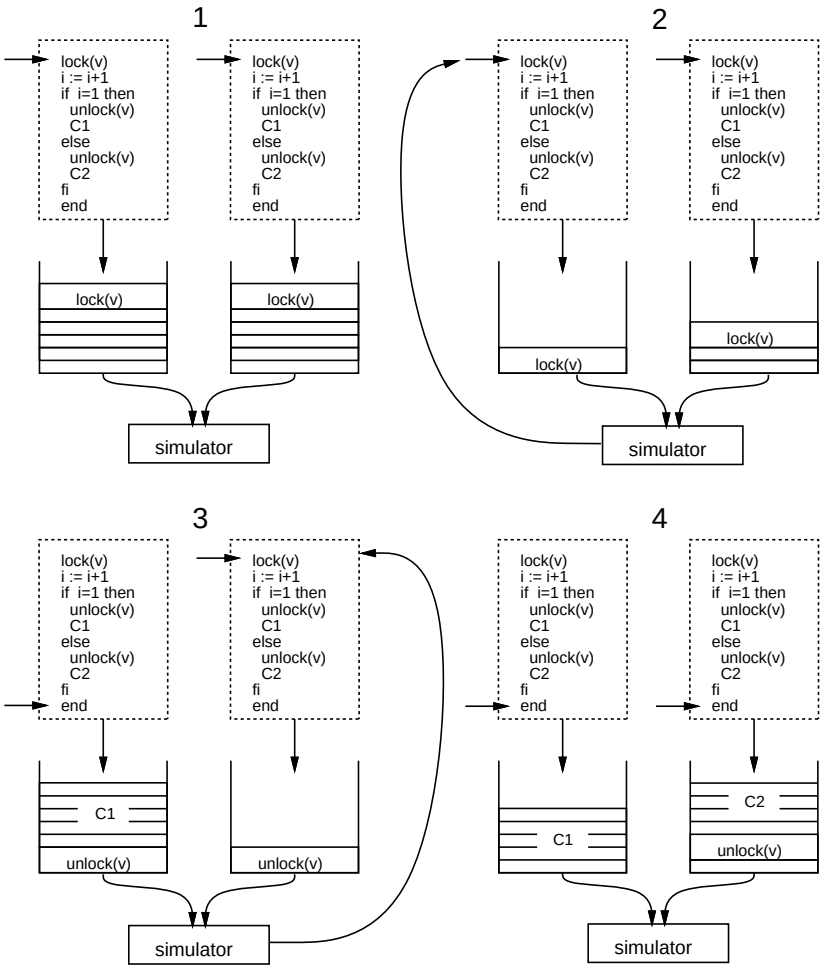

Figure 2: Execution driven Simulation

Figure 2 shows the execution driven simulation of the previous example. The simulation can logically be divided into four phases:

1. The two processes generate their traces until they encounter the *lock* operation. Both processes stop their execution at this point. The simulator starts consuming the traces with respect to the timing characteristics of the simulated architecture.

2. When the simulator finds the first *lock* operation in one of the traces, it simulates the operation and resumes the execution of the corresponding process (P1 in our example). This process then generates the *unlock* operation and the trace corresponding to the execution of C1.

3. When the simulator consumes the *unlock* operation from the trace of process P1, it resumes the execution of process P2 which enters the critical section and generates the trace corresponding to the execution of C2.

4. The simulator finishes the consumption of the two traces.

As presented in the figure, the correct execution order of the application is enforced. Moreover, the waiting times on the synchronization operations are respected.

Usually, the implementation of execution driven simulation uses simulated clocks. Each process of a parallel application owns a simulated clock which represents its elapsed time since the beginning of the execution on the simulated architecture. The simulator is used to update these simulated clocks, either by a simple increment or by performing a more detailed emulation of the target architecture. When a process issues a globally visible operation, this process stops and waits until it is its turn to perform the operation according to the values of the different clocks. As the application continues its execution, the correct ordering of program events can be ensured since all the simulation times are known.

The update of the simulated clocks is determined by the events selected to be simulated and influences the accuracy of the simulation because when an event is not simulated, the clock must be updated with an approximate value. A very detailed simulation will consider all the memory references and synchronization operations. A study that is not concerned with private data references or whose access time to private data is known could choose to consider only the accesses to shared data and to synchronization operations. One last option is to simulate only the synchronization operations or the message communications. These solutions constitute a trade-off between accuracy and simulation speed.

Some execution driven simulators have been implemented. In Tango [Davis *et al.* 91], compiled application processes are multiplexed to run in a way

that preserves the ordering and timing of the events of interest with respect to the simulated time of the target system. This is ensured by using software clocks associated with each process of the application. A scheduler algorithm elects the process whose clock value is the smallest, thus enforcing the correct ordering of the events. The different clocks represent the simulated time for each process. An option allows the selection of a set of operation types for which accurate global ordering is ensured. Since a process execution between events of interest is assumed not to be affected by any other process, the events are correctly ordered by switching processes immediately before these events. The Rice Parallel Procesing Testbed [Covington *et al.* 88] proposes a similar solution but considers that the clock can be updated with an estimate value as long as a process does not interact with another process. When a process interaction occurs, the process is suspended until the estimated time and the time of the communication evaluated by the simulator have elapsed.

Execution driven simulation has several advantages over trace driven simulation. The most important is the absence of distortion in the simulation results whichever the architecture and application simulated since the simulator ensures that the execution of the application conforms to the simulated architecture. Execution driven simulation also avoids the need to store large address trace files. Finally, since the simulator drives the execution and tracing of the application processes, any multiprocessor system can be simulated on a single processor system.

The primary drawback of this technique is that it requires an efficient trace generator to provide acceptable simulation performance since the application processes are traced during each simulation. The second problem is that an execution driven simulator is much more complex to implement than a trace driven simulator due to the necessary synchronization mechanisms between the simulator, trace generator and application processes.

3 Programming Model of Parallel Applications

This section presents the programming model used by the parallel applications whose addresses traces are simulated. In this programming model, a parallel application is made up of a set of processes communicating through shared memory. The application processes are dynamically created by an initial or parent process which also performs the initialization part of the application. The synchronization structures used are locks and barriers.

The applications are written in C and use the parmacs macros from the Argonne National Laboratory [Lusk & Overbeek 87] for managing shared memory, synchronization and process creation. In particular, this programming

model is used by the SPLASH (Standford Parallel Applications for Shared Memory) benchmark suite [Singh *et al.* 91]. SPLASH is a set of scientific parallel applications for use in the design and evaluation of shared memory multiprocessing systems. These macros are available for various architectures and simulators allowing the same applications to run on different platforms without any modifications.

A short description of the available macros is given in table 1 and a simple parallel program that demonstrates the use of these constructs is presented in figure 3.

MAIN_ENV	declare and include global data structures
MAIN_INITENV (size)	declare a shared segment of *size* bytes
G_MALLOC (size)	allocate *size* bytes of memory within the shared segment
MAIN_END	deallocate the shared segment and exit
CREATE (func)	create a child process which executes function *func*
GET_PID	return the pid of the process
WAIT_FOR_END	wait until the end of all the children processes
BARDEC(barrid)	declare a barrier
BARINIT (barrid, nproc)	initialize a barrier to block *nproc* processes
BARRIER (barrid)	block on a barrier
LOCKDEC(lockid)	declare a lock
LOCKINIT (lockid)	initialize a lock
LOCK (lockid)	acquire a lock
UNLOCK (lockid)	release a lock

Table 1: the parmacs macros

This program increments and then decrements a shared variable i. The first phase, incrementation, is separated from the second phase, decrementation, by a barrier. Locks ensure the atomicity of the incrementation and decrementation of i. Any number of processes can be used but at the end of the first phase, i must be equal to the number of children processes and at the end of the execution the father process writes 0 for the value of i.

3.1 Globally visible events

Execution driven simulation ensures that the execution of the application which produces the address trace to be simulated is the one that would occur if the application was actually executed on the architecture being simulated. This implies that each process stops its execution each time it needs to execute a globally visible operation that may change the execution path

```
MAIN_ENV

struct gmem {                          /* shared memory structure */
        BARDEC (barrier)
        LOCKDEC (lock)
        int             i;
}               *mem;

void
child ()
{
        LOCK (mem->lock)
        mem->i = mem->i + 1;
        UNLOCK (mem->lock)

        BARRIER (mem->barrier)         /* wait for the others */

        LOCK (mem->lock)
        mem->i = mem->i - 1;
        UNLOCK (mem->lock)
}


main (int argc, char **argv)
{
        int             NCPU, i;

        NCPU = atoi (argv[1]);

        MAIN_INITENV (1024) /* declare 1 Kbytes of shared memory */

        mem = (struct gmem *) G_MALLOC (sizeof (struct gmem));
        /* allocate memory for the shared data strucure */

        BARINIT (mem->barrier, NCPU)
        /* mem->barrier blocks all children processes */

        LOCKINIT (mem->lock)

        for (i = 1; i <= NCPU; i++) { /* create children processes */
                CREATE (child)
        }

        WAIT_FOR_END  /* wait until all children have ended */
        printf ("value of i : %d\n", mem->i);
        MAIN_END
}
```

Figure 3: A parallel application

through the parallel application. When such an operation is issued, its execution is delayed until the simulator has consumed all the previous events in the trace of the process. When the simulation has reached the same point as the execution, the process can resume its execution.

With this programming model, there are two types of globally visible operations: accesses to shared variables and synchronization operations. In our implementation, the only globally visible operations considered are the synchronization operations. The main reason for this restriction is that it would otherwise be necessary to identify all accesses to shared variables which is a very time consuming task.

As a consequence, an application cannot use shared variables to implement synchronization (for example, spin-wait locks will not generate the correct trace) and all shared variables must be accessed within critical sections. The reason is that the simulator cannot enforce the correct order of access to shared variables outside critical sections since the processes can access shared variables without stopping their execution. This restriction is in accordance with a weakly consistent shared memory semantics [Nitzberrg & Lo 91].

4 Execution Driven Simulation with SPAM

The SPAM kernel is made up of two parts: a tracing tool and a simulation library. The tracing tool generates the address traces and synchronization events of the processes of the parallel application under simulation. The simulation library allows the simulator to retrieve the events reflecting the activity of the processes and to synchronize their execution according to the execution driven simulation technique.

In this section, we first describe the process events delivered by the tracing tool to the simulator and the simulation library. We then demonstrate how the simulation library can be used to implement an execution driven simulator.

4.1 Process events

The process events reflect the activity of each process of the parallel application and are delivered to the simulator as the execution of the application proceeds. The process events are listed in table 2.

For the memory accesses, four types of memory segments are considered: text segments, data segments, stack segments and one shared memory segment. An address is an offset from the beginning of the memory segment. This permits the simulation of any kind of memory organization. For synchronization operations, the identity of the synchronization object is delivered. When a process calls a synchronization operation, it delivers the correspon-

INST_FETCH(address)	instruction fetch in the text segment
TEXT_READ(address)	data read in the text segment
DATA_READ(address)	data read in the data segment
DATA_WRITE(address)	data write in the data segment
STACK_READ(address)	data read in the stack segment
STACK_WRITE(address)	data write in the stack segment
SHARED_READ(address)	data read in the shared memory segment
SHARED_WRITE(address)	data write in the shared memory segment
LOCK(lock_id)	lock (globally visible event)
UNLOCK(lock_id)	unlock
BARRIER(bar_id)	wait on a barrier (globally visible event)
WAIT_END	the parent process waits until all its children terminate
CREATE_PROC(pid)	creation of a new process
HALTED	the process is halted at a synchronization point
TERMINATED	the process has terminated

Table 2: Process events

ding event to the simulator and suspends its execution until the simulator decides to resume it. The creation of a new process is also reported to the simulator to allow the initialization of the data structures needed for the simulation of this process. When a process is halted on a lock or a barrier or has terminated its execution, the corresponding event is reported to the simulator. Other event types could be easily added in order to simulate other process activities, for example operating system calls.

4.2 Simulation library

This section presents the set of primitives necessary to implement an execution driven simulator. These primitives are presented in table 3.

The *spam_boot* primitive is called at the beginning of the simulation to start the parallel application. The parent process then starts to execute and produces an event stream. The simulator retrieves the process events with the *spam_get_next_event* primitive. When a *LOCK* event is delivered to the simulator, the simulator has to emulate the operation. Remember that when this event is consumed by the simulator, the execution of the process is suspended. The *spam_lock* primitive resumes the execution of the process if the emulated lock is free, otherwise the process has to wait until another process releases the lock. When an *UNLOCK* event is issued, the simulator emulates the operation with the *spam_unlock* primitive. The first process waiting for this lock resumes its execution and enters the critical section. *BARRIER* events are treated similarly. A process blocking on a barrier suspends its exe-

*spam_boot(char *appl[])*	start the parent process of the application
spam_get_next_event(int pid)	get next event from process *pid*
spam_end(void)	return true if the application has ended
spam_lock(int pid, int lockid)	acquire a lock (the process resumes its execution and enters the critical section if the lock is free)
spam_unlock(int pid, int lockid)	release a lock (grant the lock to the first process blocked and resume its execution)
spam_barrier(int pid, int barid)	block on a barrier (if this process is the last to block, all processes resume their execution)

Table 3: SPAM primitives

cution and generates the corresponding event. When the simulator finds the *BARRIER* event, it emulates the operation with the *spam_barrier* primitive. If this process was the last to block on the barrier, the execution of all the processes blocked on the barrier is resumed otherwise the calling process remains blocked. The *spam_end* primitive is used to detect the termination of the application.

4.3 Simulator example

To illustrate the use of the simulation library, we now give a simple example of a simulator (figure 4). This simulator simply consumes all the events coming from processes and counts the number of memory references. This example can form the core of a more complex simulator in which memory latencies and contentions between processors can be modeled. After starting the simulated application, the simulator enters a loop in which it retreives process events and ensures correct synchronization of the execution of the processes until the application ends.

Since no timing nor interaction are modeled in this example, the synchronization primitives are executed as soon as the corresponding event is retrieved by the simulator. Should a shared bus be modeled, the execution of a synchronization primitve would be delayed until the bus is granted to the calling processor, thus resulting in a possibly different execution order.

Since the simulator has complete control over the execution of the application processes, the impact of multiprogramming or process migration could trivially be studied by scheduling application processes to processors on the simulated architecture.

```
void main(int argc, char *argv[])
{
  int       i;
  int       nproc = 1;                          /* number of processes */
  spam_ref  *ref;

  spam_boot(&argv[1]);   /* start the application given in parameter */

  while (1) {
    for (i = 0; i < nproc; i++) {

      ref = spam_get_next_event(i);  /* get next event from process */

      switch (ref->type) {
      case LOCK:
        spam_lock(i, ref->ref_data.lock_id);
        break;
      case UNLOCK:
        spam_unlock(i, ref->ref_data.lock_id);
        break;
      case BARRIER:
        spam_barrier(i, ref->ref_data.barrier_id);
        break;
      case CREATE_PROC:
        nproc++;               /* increment the number of processes */
        break;
      case WAIT_END:
        break;
      case HALTED:  /* process is waiting on a synchronization call */
        break;
      case TERMINATED:
        if (spam_end()) {         /* test the end of the simulation */
          printf("simulation ended\n");
          exit(0);
        }
        break;
      default :  /* memory reference */
        printf("memory reference : 0x%x\n", ref->ref_data.address);
        break;
      }
    }
  }
}
```

Figure 4: Simulator example