Efficient Execution Replay for ATHAPASCAN-0 Parallel Programs

Jacques Chassin de Kergommeaux, M. Ronsse, K. de Bosschere

INRIA
INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Efficient execution replay for ATHAPASCAN*-0 parallel programs*

J. Chassin de Kergommeaux, M. Ronsse, K. De Bosschere

No 3635

Mars 1999

THÈME 1

ISSN 0249-6399

RHÔNE-ALPES

Efficient execution replay for ATHAPASCAN-0 parallel programs

J. Chassin de Kergommeaux*, M. Ronsse†, K. De Bosschere†

Thème 1 — Réseaux et systèmes
Projet APACHE

Rapport de recherche n ° 3635 — Mars 1999 — 28 pages

Abstract: ATHAPASCAN-0 programs are executed by a network of communicating threads evolving dynamically. Within the same node, threads communicate through shared memory and synchronization primitives. Between two different nodes, threads communicate by message passing. Execution replay of ATHAPASCAN-0 programs addresses the non-determinism arising from synchronization races, from promiscuous messages received from non specified source and from the varying number of operations testing the completion of non blocking ATHAPASCAN-0 primitives. The execution replay mechanism is mainly control-based since, in addition to recording the results of test operations, only the order of accesses to synchronization functions and the order of arrival of promiscuous messages need to be recorded. The efficiency of the recording comes from the use of Lamport clocks to reduce drastically the number of records associated to synchronization operations and from the reduction to a single record of the information necessary to reproduce a series of unsuccessful tests.

Key-words: Execution replay, Threads, Synchronization, Message Passing, Lamport clocks

(Résumé : tsvp)

This work was sponsored by "Programme d'Actions Intégrées franco-belge Tournesol No. 98114"

* B.P. 53, F-38041 Grenoble Cedex 9. Jacques.Chassin-de-Kergommeaux@imag.fr
† Vakgroep Elektronica en Informatiesystemen, St. Pietersnieuwstraat 41, B-9000 Gent. {kdb,ronsse}@elis.rug.ac.be

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN (France)
Téléphone : 04 76 61 52 00 - International: +33 4 76 61 52 00
Télécopie : 04 76 61 52 52 - International: +33 4 76 61 52 52

Réexécution déterministe efficace de programmes ATHAPASCAN-0

Résumé : Les programmes ATHAPASCAN-0 sont exécutés par un réseau de processus légers communicants, évoluant dynamiquement. Les processus légers d'un même nœud du système parallèle communiquent par mémoire partagée et par des primitives de synchronisation. En revanche, les processus légers de nœuds différents communiquent par passage de message. La réexécution déterministe de programmes ATHAPASCAN-0 traîte le non déterminisme provenant des conditions de concurrence de synchronisation, des messages en concurrence lorsque la source n'est pas spécifiée par la primitive de réception ainsi que du nombre variable de primitives exécutées pour tester la terminaison des primitives non bloquantes de ATHAPASCAN-0. Le mécanisme de réexécution déterministe est essentiellement basé sur le contrôle puisque, en plus de l'enregistrement des résultats des primitives de test, il n'est nécessaire d'enregistrer que l'ordre selon lequel sont effectuées les opérations de synchronisation ainsi que l'ordre d'arrivée des messages conflictuels. L'efficacité du mécanisme d'enregistrement provient de l'utilisation d'horloges de Lamport pour réduire considérablement le nombre d'enregistrements associés aux opérations de synchronisation ainsi que de la réduction à un seul enregistrement de l'information nécessaire à la reproduction d'une série de tests infructueux.

Mots-clé : Réexécution déterministe, Processus légers, Synchronisation, Transfert de Messages, Horloge de Lamport

1 Introduction

This report describes an execution replay system for ATHAPASCAN-0 [5, 4]. ATHAPASCAN-0 is a multi-threaded, portable, parallel programming runtime system [5] combining threads and communications. The ATHAPASCAN-0 runtime system was designed for parallel hardware systems composed of shared-memory multi-processor nodes connected by a communication network. It exploits two levels of parallelism: inter-nodes parallelism and inner parallelism within each of the nodes. The first type of parallelism is exploited by a fixed number of system-level processes while the second type is implemented by a network of communicating threads evolving dynamically. The main functionalities of ATHAPASCAN-0 are dynamic local or remote thread creation and termination, sharing of memory space between the threads of the same node which can synchronize using locks or semaphores, and blocking or non-blocking message-passing communications between non local threads, using ports.

The aim of ATHAPASCAN-0 and similar programming models are to ease programming of irregular applications, mask communication or I/O latencies, to exploit shared-memory parallelism and implement remote memory accesses [12]. Achieving the same results using (heavy) processes, communicating through a message-passing library such as PVM [14] or MPI [24], involves considerable programming efforts. All possible cases of unbalance must be predicted by the programmer of an irregular application. Masking communication and I/O latencies requires to manage a communication automaton, on each of the nodes of the parallel system. System (heavy) processes having disjoint address spaces are not suited for exploiting shared memory parallelism. On the contrary, it is fairly simple to spawn several threads to cope with the evolution of an irregular problem or mask communication latencies. In addition, inner parallelism of shared memory multiprocessor nodes can be exploited by several threads sharing the same memory. Remote memory accesses can be serviced by dedicated threads. ATHAPASCAN-0 can be used directly to program parallel irregular applications [3] or as a runtime kernel to implement data or control parallel programming models of higher level of abstraction such as ATHAPASCAN-1, built on top of ath-0 [6, 13].

The execution replay facility described in this report is to be part of a debugging environment for clusters-based parallel systems. This environment will include a parallel debugger based on the coordination of local debuggers [7] commanded by a graphical interface. In addition to the textual representation of the debugged program provided by the parallel debugger, it will be possible to have a high level representation of the execution of the program using the Pajé visualization tool [8]. The main reason for providing a replay system for ATHAPASCAN programs is to make cyclic debugging possible. Without execution replay, the "probe effect" caused by the use of a debugger is likely to prevent programmers from being able to reproduce erroneous executions as many times as necessary to identify the origin of their errors. The execution replay will make possible to collect traces and visualize the execution of the program being debugged during a replayed execution, without worrying about the considerable perturbation introduced by the visualization.

The execution replay mechanism is based on the "Instant Replay" technique [16]: control information is recorded during an initial phase, this information being used to force subsequent "replayed" executions to execute deterministically with respect to the initial recording. Recording of traces should perturbate parallel executions as little as possible so that errors do not vanish, due to the "probe" effect arising from trace recording, when recording an erroneous execution is attempted. This objective is difficult to achieve because of the wide variety of possible interactions between the threads of ATHAPASCAN-0 programs: within a node they communicate using shared memory and synchronization primitives while threads executing on different nodes communicate through message passing.

The proposed implementation combines features of the RecPlay[33, 32] system developed in the ELIS group at the University of Ghent – which makes it possible to record and replay shared memory programs on a shared memory multiprocessor [28, 29]– with new developments for message passing communications.

The organization of this report is the following. After this introduction, the Athapascan-0 application programming interface and system implementation are presented. Then a sketch of the execution replay mechanism of Athapascan-0 is presented. The design and implementation of the execution replay for the shared memory primitives of Athapascan-o is then described. The next section is dedicated to the recording and replaying of the distributed memory functions of Athapascan-0 before the conclusion.

2 Athapascan-0

2.1 Athapascan-0 application programming interface

Athapascan-0 programs are executed by dynamically evolving sets of interconnected threads. Threads can be created locally or remotely. Once created, threads cannot be migrated. Local threads communicate using shared memory and synchronization primitives. Remote threads communicate using message passing through communication ports (Figure 1).

The Athapascan-0 application programming interface includes a large number of functions:

Initialization: declaration of global ports and services. Each remote thread creation request includes the name of a service as a parameter, the service being the first function to be executed by the new thread when it is created.

Thread creation: blocking or non blocking remote thread creation to execute previously declared services or local (slave) thread creations. It is also possible to terminate the current thread.

Synchronization: classical synchronization functions on mutexes locks, condition variables and semaphores.

Communications: blocking or non blocking message passing communication primitives using local – associated to a single node of the system – or global system wide ports.

Test primitives: since many Athapascan-0 requests are non blocking, it is possible to test for the completion of one or several non blocking requests; in the latter case, the call succeeds if one of the requests has completed.

2.2 Non determinism in Athapascan-0

A program is said to be *externally deterministic* if there is only one possible output for each input. A program is said to be *internally deterministic* if and only if the program is externally deterministic and if the sequence of instructions each thread executes, along with the values of the operands used by each instruction is deterministic [9]. A program is non deterministic if its behavior not only depends on its input. Cyclic debugging assumes that a program execution can be faithfully re-executed any number of times. Therefore, non deterministic parallel programs are hard to debug by means of cyclic debugging because subsequent executions with identical input are not guaranteed to have the same behavior. Known sources of nondeterminism are: certain system calls (like `random()`

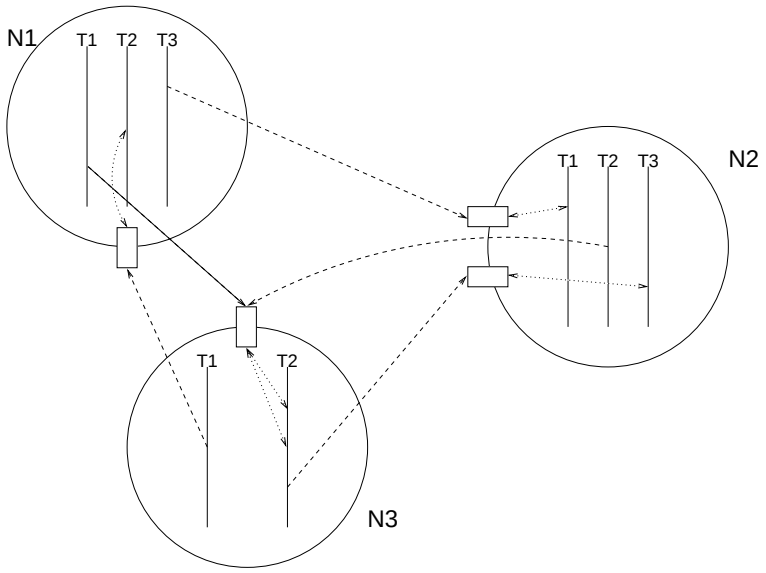

Figure 1: Message passing using ports in ATHAPASCAN

and `gettimeofday()`), interrupts, traps, signals, non-initialized variables, unsynchronized accesses to shared memory (for parallel programs) and message exchanges (for distributed programs).

There exist effective and fairly efficient ways to remove the sources of nondeterminism that also occur in sequential programs: the output of nondeterministic system calls can be traced; the nondeterminism caused by interrupts, traps and signals can be dealt with by using the technique of Interrupt Replay [1]; non-initialized variables and dangling pointers can be detected by packages like Purify[1] or Insight[2]. The nondeterminism that is specific for parallel and distributed programs is however much harder to deal with during debugging. This report uniquely focuses on solutions for the nondeterminism that is specific for parallel and distributed programs.

There exist various sources of non determinism in ATHAPASCAN. Accesses to shared memory may create *race conditions* which can be of two different classes. *Synchronization races* are race conditions that cause the program to execute nondeterministically in a way that was intended by the programmer. E.g. the competition by threads to enter a critical section or to obtain a semaphore causes a synchronization race. The other race conditions are not intended by the programmer and as such are bugs that should be removed. These *data races* (Figure 2) are the result of a lack of proper synchronization. By adding proper synchronization, data races can always be removed. Removing synchronization races means that a program is made completely deterministic. Excluding synchronization races means that dynamical management techniques of parallel programs such as load balancing would no longer be possible which would contradict the objectives of ATHAPASCAN-0.

The second source of nondeterminism in ATHAPASCAN is message passing. In ATHAPASCAN, it is possible to receive messages from "any source". This results in a race if a "receiving" thread can choose from different messages. Replayed executions need to ensure that messages received from "any source" are received in the same order as during the reference recorded execution. Non-blocking requests introduce further non-determinism since they have to be associated with test functions in

[1] `http://www.rational.com`
[2] `http://www.parasoft.com`

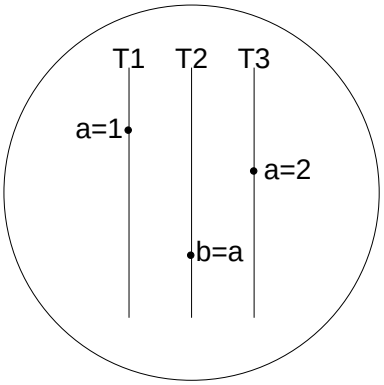

Figure 2: A program containing a *data race* on variable a.

order to detect their completion. During replay, the same number of calls to the test functions should be executed.

2.3 Implementation of ATHAPASCAN-0

In order to be portable on a large number of different parallel platforms, the ATHAPASCAN-0 runtime kernel was implemented on the top of existing standard thread libraries complying to the Posix standard [25] and to communication library MPI [24]. To ease portability, ATHAPASCAN is built using a layered approach (Figure 3). The layer on top of the hardware (shared memory and network) is the MPI layer for sending messages and the Posix layer for creating and synchronizing threads. The Posix library offers mutexes and condition variables for synchronization purposes. In this layer, the Posix functions are renamed: function akXYZ simply calls Posix function XYZ. The ATHAPASCAN kernel, aKernel, is built on top of this layer and, basically, extends the Posix and MPI layer in two ways: semaphore functions are added and a thread aware version of the MPI functions are provided. The latter is necessary as most MPI implementation are not thread-safe (the functions are nonreentrant) and most thread-safe MPI implementations are not thread-aware, that is block all the threads of a node (Unix process) each time a thread calls a blocking communication primitive. Therefore the ATHAPASCAN-0 kernel transforms all blocking communications into non-blocking ones which are performed by a communication daemon thread. This daemon performs non-blocking communications on behalf of the threads created by the user program and checks for the arrival of communications regularly (using `MPI_IProbe`). Similarly, remote thread creations are serviced asynchronously by specific daemons on the caller and callee sides, even if the caller thread's request is synchronous. The layer on top, the ATHAPASCAN-0 layer, implements the ATHAPASCAN-0 application programming interface, using the aKernel functionalities.

3 Overview of the ATHAPASCAN-0 execution replay implementation

3.1 Theoretical basis

The execution replay system of ATHAPASCAN-0 is backed by the theoretical work of Éric Leu [17, 18]. The main results presented in his PhD dissertation can be summarized as follows. First of all a formal definition of the equivalence between two program executions is given. The intuitive meaning of this definition is that, two executions X and Y of a parallel program P are equivalent if each of

the processes executing the program P walks through the same sequence of states during each of the executions. Two equivalent executions will therefore produce the same result and/or the same error(s).

Then a theorem, giving sufficient conditions for two program executions to be equivalent, is proven. This theorem assumes that both executions have the same initial state and that all events of both executions are elementary events such that each event depends solely of the previous state of the process where the event takes place. With the assumptions above, **two parallel program executions are proved equivalent if the reception order of messages by each of the processes and the partial order of access to shared memory cells are identical in both executions**

This result is used to implement the execution replay mechanism of ATHAPASCAN-0. During an initial execution, the reception order of messages as well as the order of access to synchronization objects are recorded. Non synchronized accesses to shared memory are not recorded since this recording would increase the execution time a lot and require an enormous amount of time. The traces are used to force subsequent replayed executions of parallel ATHAPASCAN-0 programs, fulfilling the hypotheses of the equivalence theorem – same initial state, no usage of functions such as random – and performing no non synchronized access to shared memory, to be equivalent to the initially recorded execution. Since non synchronized accesses to shared memory are not recorded, it will not be possible to replay programs presenting data races. However, the execution replay mechanism can help finding such errors. If it is possible to adapt the JiTi race detection mechanism [34] to ATHAPASCAN-0, it will then be possible to detect data races during replayed executions. In any case, since the replayed program execution will behave deterministically at least until the first data race, it should be possible to perform repeated executions of faulty programs until the first error and it should be much easier to identify it.

3.2 Abstraction level of recording

What to record is an important question. The information we record directly relates to the level of equivalence that will exist between the recorded and the replayed execution. In ATHAPASCAN-0 it should be possible to trace events at the application programming interface level (Figure 3). As it was demonstrated in an early prototype [11, 10], the recording at this level of abstraction reduces the amount of information to be recorded with respect to lower abstraction levels since some communications, at the highest level of abstraction, require several communications at the lowest level. However, this is not a good idea:

- most ATHAPASCAN-0 functions would have to be instrumented; as indicated in section 2, there exist a lot of ATHAPASCAN-0 functions and this would involve numerous modifications. In addition, because of the variety of possible cases, there would be many different types of traces whose management at record and replay would likely be costly in time and memory.
- each time a function is added to ATHAPASCAN-0, instrumentation would have to be added;
- at such a high level, only the application is replayed, not the ATHAPASCAN-0 layers themselves, making (correctness and performance) debugging of ATHAPASCAN itself impossible.

Tracing at the lowest possible layer is also a possibility. Tracing at this layer makes it possible to replay the whole system: not only the application but also the ATHAPASCAN-0 libraries. However, it is clear that this comes with a cost: tracing all memory references introduces an intolerable overhead. Therefore, the proposed record/replay mechanism traces all events at the abstraction

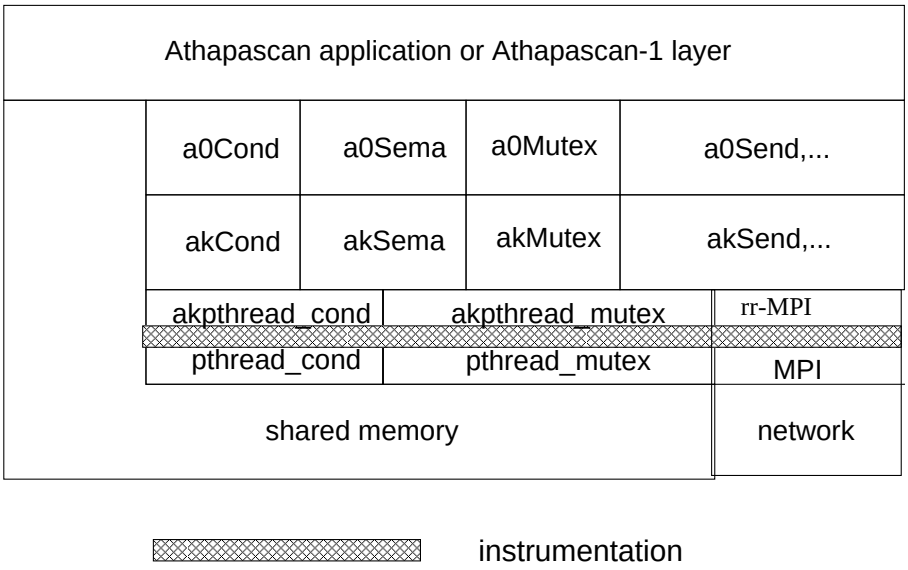

Figure 3: The instrumentation of ATHAPASCAN for record-replay

The instrumentation is placed in the kernel layer of ATHAPASCAN whose aim is to ease portability on various software platforms.

level represented on Figure 3. Note that this line does *not* extend from the left to the right on the figure, meaning that some events may pass unnoticed from the application layer to the hardware. As this means that some events will not be traced, a faithful replay will not be guaranteed for all possible executions. The next sections explain why this choice is nevertheless acceptable. Compared with a recording at the application level [10], this solution requires the recording of fewer functions, but these functions are called more frequently, by the communication daemon thread in particular. However this is not a problem since solutions were found to reduce drastically the number of records to be performed for synchronization (see section 4.1) and message passing communications (see section 5.4). Figure 3 shows the instrumented layers of ATHAPASCAN: e.g. `akpthread_cond` is no longer another name for `pthread_cond`. The function `akpthread_cond` has become a real function that performs instrumentation routines and calls `pthread_cond`.

4 Shared Memory: Dealing with Synchronization Races

4.1 Optimized tracing using logical clocks

As explained previously, a synchronization race is a possible source of non determinism in a program, but we do not consider a synchronization race a bug if the program is externally deterministic. Nevertheless, the non determinism that is caused by it often prevents cyclic debugging of the program. Therefore, we must at least record the order in which synchronization operations are executed, and use this information during replay (and debugging).

The record phase is clearly the most difficult part of a record/replay system. During replay, the execution of a program can be delayed arbitrarily without changing the order of synchronization

operations. During the record phase, this is certainly not true. The overhead caused by recording a trace should be minimal to limit the probe effect.

The tracing can be done using hardware [2], software or hybrid probes. Probably the most important issue in record/replay is the question what and how much to record during the initial execution. There are two possible approaches that force executions of a non deterministic parallel program to be "equivalent" to a traced execution. The first one is to force the threads to read the same values of shared variables as during the traced execution by recording the original value (*contents based or data driven replay*). The second one, using the result presented in section 3.1 is to force the threads to access the shared variables in the same order as during the original execution, forcing the variables to undergo the modifications in the same order as during the record phase (*ordering based or control driven replay*). The first approach is not considered feasible because it leads to huge trace files (a data rate of 1 MB/s on a VAX 11/780 has been reported [27]). Fortunately, the second approach allows us to dramatically reduce the trace sizes.

The overhead introduced by the tracing must be as small as possible (in time [11] and in space [16, 19, 26]). The *time overhead* should be small to circumvent Heisenbugs and to limit the probe effect. The *space overhead* for the trace files should be small too because (i) this is necessary to be able to trace long running program, and (ii) storing the trace file requires bandwidth, which must be shared with the target program. In practice, space overhead can be traded for time overhead by using compression.

The techniques developed to deal with ATHAPASCAN-0 synchronization races are based on ROLT (Reconstruction of Lamport Timestamps) [22, 23, 21, 20, 29, 31], an ordering-based record/replay method, where only the partial order of synchronization operations is traced. The method [22] has the advantage that it produces small trace files and that it is less intrusive than other existing methods [26]. Moreover, the method allows the use of a simple compression scheme [30] which can further reduce the trace files.

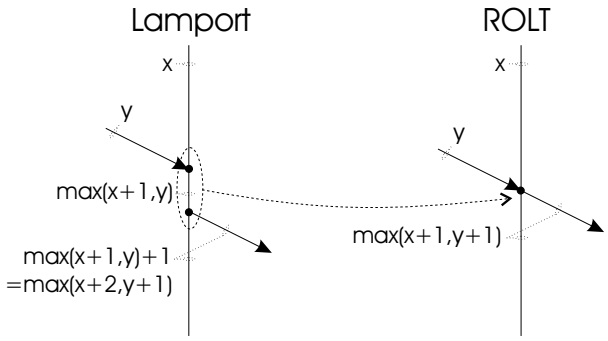

Figure 4: Comparing Lamport Clocks with ROLT Clocks.

During the record phase we do not trace the order of all synchronization operations. A scalar logical clock is attached to each synchronization operation. This logical clock is actually an optimization of the classical Lamport Clock [15] (see Figure 4). Lamport assigns logical clocks to threads, and clocks are communicated between thread by piggybacking them on the messages they exchange. A characteristic of a message is that it is a consumable, it is created by a sender, and consumed by a receiver. Every message is a new message by definition. In shared memory parallel programs, threads communicate by means of shared variables, but shared variables are not consumables, but can be recycled, and a value that is written by one thread can be read by multiple other threads. Furthermore, the shared locations that are used by synchronization operations (the ones

that we want to trace) are either written to (as is the case with a V-operation), or are first read, and then written (as is the case with a P-operation). Hence, a synchronization operation is always a *sender*, and sometimes a *receiver* and a *sender*. The latter combination (*receiver* and *sender*) is in ROLT considered as one single event, which has two advantages: (i) the Lamport clocks are incremented more slowly, and (ii) there are less events to be traced, reducing the size of the trace files.

A further optimization, which is a consequence of the recyclable nature of the shared variables as communication channels is that Lamport clocks do not have to be piggybacked on the messages, but can be permanently attached to the shared variables. Hence, not only the threads are given a Lamport clock, but also the shared variables that are used for synchronization purposes (semaphores, mutexes, condition variables, etc.).

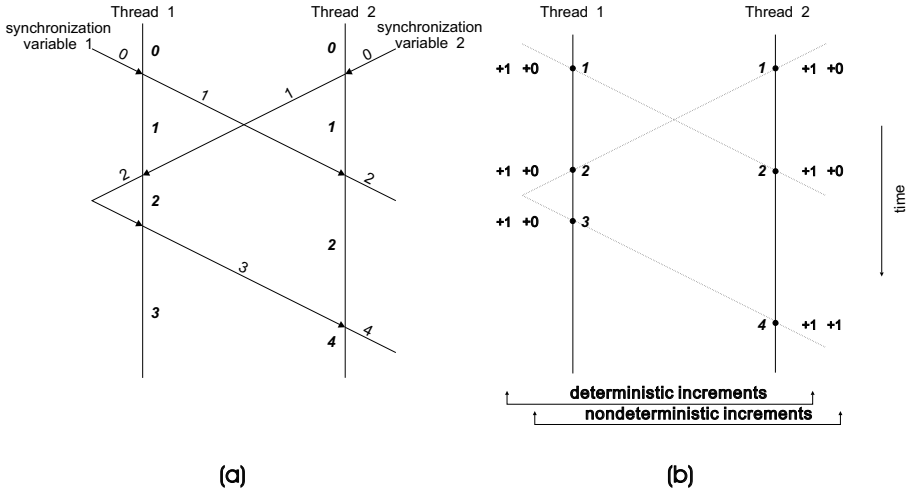

Figure 5: ROLT: updating the scalar clocks.

Operations on the same synchronization variable are connected by an arrow, pointing from the synchronization operation that occurred first to the next synchronization operation on that synchronization variable.

The logical clocks are updated as follows: when a synchronization event takes place, both the logical clock of the thread executing the event, and the logical clock of the synchronization variable are incremented. Then the processor and the synchronization variable are *synchronized* by both storing the maximum of the two clock values as the new common logical clock value (Figure 5.a):

$$LC_t \doteq LC_o \doteq max(LC_t, LC_o) + 1$$

(X' is the new value, X is the old value). This method of updating the scalar clocks is not exactly the same as the method proposed by Lamport [15], but is has the same consistency properties.

The execution traces consist of a sequence of timestamps, one trace per thread. In order to reduce the amount of information to store, not the actual timestamps, but the timestamp increments are stored. This information can be further reduced by splitting the increments into a deterministic increment (the most common case: +1), and a nondeterministic increment (an update due to a synchronization variable that has a higher logical clock value) (Figure 5.b). In principle, storing the nondeterministic increments per processor is sufficient to allow a correct replay as the deterministic increments can be recomputed.

In [21] it is proven that logging only the synchronization operations for which the nondeterministic increments are nonzero (i) still allows a correct replay, (ii) does automatically eliminate the trivial transitive orderings, (iii) allows for an efficient compression scheme.

To get a faithful replay, it is sufficient to stall each synchronization operation until all synchronization operations with a smaller timestamp have been executed.

4.2 Encoding of synchronization traces

During the record phase we might consider to log the sequence of Lamport timestamps for each thread. Suppose a thread adapts the following sequence of timestamps:

$$0, 1, 2, 4, 7, 8, 9, 10, 11, 12, 15, 17, 18, 19, 21 \tag{1}$$

If we log each timestamp as a 32 bit number, this approach would generate an amount of data to be logged of $15 \times 4 = 60$ bytes, which is the same as for Instant Replay [16]. With the ROLT method only a subset of all timestamps are logged and therefore smaller traces will be generated. The method is based on the observation that in most cases, the clock is augmented by one, corresponding to a deterministic increment. Since the same deterministic update algorithm is used during replay, we don't have to log those updates. However, the nondeterministic increments must be logged in order to be able to reconstruct the original Lamport timestamps. Thus, we only have to log an update of a Lamport clock, if the increment is larger than one. Therefore, the ROLT method logs a pair of timestamps when the difference between them is larger than one. So, for the above example we can log

$$(2, 4), (4, 7), (12, 15), (15, 17), (19, 21). \tag{2}$$

This time, we only need $5 \times 2 \times 4 = 40$ bytes. By only logging this subset, we obtain a reduction w.r.t. Instant Replay. The ROLT method makes it possible to reduce the number of log entries. However, each log entry still requires 32 bits. Now we will try to reduce the number of bits for each log entry, and obtain an additional reduction of the traces.

With the ROLT method a list of growing numbers has to be traced. Such a list can be encoded very efficiently after applying a transformation. The transformation is based on two facts:

- the first item of a pair of numbers is always larger than or equal to the last item in the previous pair;
- for each pair (a, b), b is always greater than $a + 1$.

This means that we can transform the list

$$(a, b), (c, d), (e, f)$$

into the list

$$(a, b - a - 2), (c - b, d - c - 2), (e - d, f - e - 2).$$

For the example we obtain the list

$$(2, 0), (0, 1), (5, 1), (0, 0), (2, 0) \tag{3}$$

Experiments conducted on different examples show that the numbers we obtain by using this transformation are very small. In fact, a simple method is used to encode the increments of the logical time-stamps in an even more compact way. If the increment is smaller than 255 we use 8 bits to write it to the log file. If the increment is larger than 254, we use 40 bits: the value 255 (8-bit) followed by the increment (32-bit). This is a very simple compression algorithm, allowing the logical clocks to be coded and decoded very fast.

4.3 Dealing with Synchronization Races in ATHAPASCAN-0

This section explains in detail how to update the logical clocks at the explicit (mutexes, condition variables and semaphores) and implicit (thread creation and exit) synchronization operations. Information on replaying these functions is also given. The main data types used in the following are:

`rr_pthread_mutex_t:` structure including a posix mutex and its associated logical clock.

`rr_pthread_cond_t:` structure including a posix condition variable and its associated logical clock (see section 4.3.3).

The main data structure used during the replay phase is a table containing the current values of the logical clocks of the threads of the node. Each thread reads and writes its own logical clock value and reads the values of the logical clocks of the other threads of the node. Accesses to this table is protected by a lock. In the text, the following functions are used:

`increment_lc:` this function increments the thread's logical clock.

`calc_lc:` this function updates the logical clocks of both the thread executing the function and the object involved. It takes the old logical clock of both, takes the maximum, adds one and updates the clock of both the thread and the object. This function is only executed during the record phase. If the increment of the clock of the thread is larger than one, we log the increment (see further).

`replay_try:` this function checks if the current thread can proceed with its current logical clock. To do this, the timestamp is compared with the clocks of all the other threads. If the thread executing `replay_try` has the smallest clock it proceeds, if not the thread goes to sleep. This function is only executed during the replay phase.

`replay_next:` this function calculates the next timestamp for the current thread (from the information recorded during the record phase) and awakens the thread with the smallest logical clock. This function is also only executed during replay;

4.3.1 Thread Creation and Exit

The creation of a thread (`a0NewSlave`) and the joining of a thread (`a0JoinSlave`) are implicit synchronization points (Figure 6): all events executed by a newly created thread happen strictly after the execution of `a0NewSlave` and the events executed by a thread after the execution of `a0JoinSlave` happen strictly after all the events of the thread that died (using `a0ExitThread`). Although it is not necessary to record these events for a correct replay, it is necessary for ROLT that the clocks are updated. E.g. in Figure 7 the clocks are not updated at thread creation or exit. This will lead to deadlock during replay: the event on thread 1 (timestamp=1) has to be executed before the second event on the second thread (timestamp=2) while the `ExitThread/JoinSlave` couple demands for the opposite ordering. Therefore, the logical clocks are updated during the record phase. Note however that it is **not** necessary to enforce a wait condition (based on the logical clocks) during the replay phase: the operations are properly synchronized in a deterministic way. One must however execute a `replay_next()` to update the logical clock.

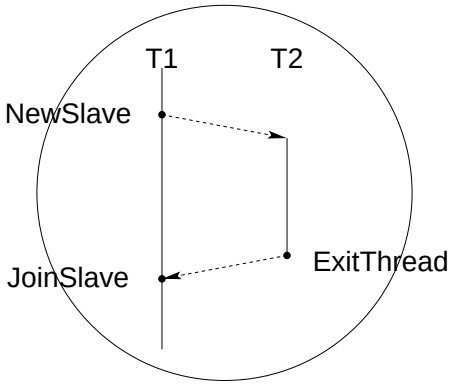

Figure 6: Implicit synchronization in ATHAPASCAN: thread creation and exit.

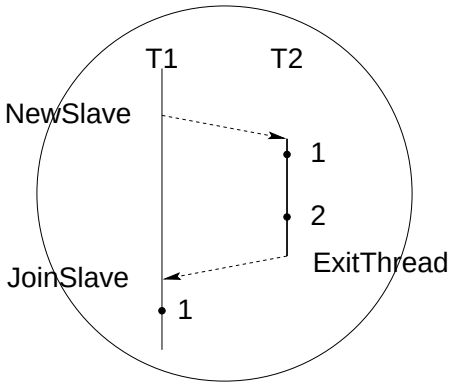

Figure 7: A recorded execution that will lead to deadlock during replay.

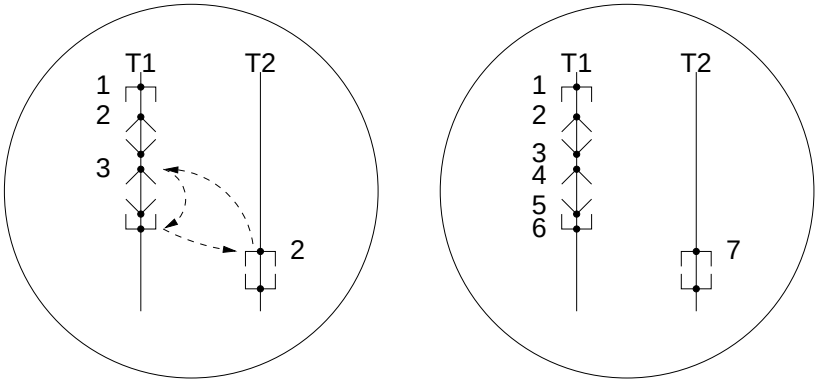

Figure 8: Nested mutex operations leading to deadlock: the execution in the left figure leads to deadlock during replay: a circular dependency arises. Tracing the unlock operations alleviates this (execution depicted in the right figure).

4.3.2 Mutexes

Mutexes are always used is pairs: the same thread executes the *lock* and *unlock* procedures. For a correct replay, we have to trace the order of the lock operations. It seems unnecessary to record the order of the unlock; as unlock operations always follow a lock operation, and no other operations on the same mutex can intervene. However, this lead to deadlock if an application uses nested lock/unlock pairs. For example, the left execution in Figure 8 leads to deadlock during replay, while the right execution does not. The latter execution also updates the logical clock at unlock operations. Note however that these updates are always deterministic, and hence they are never recorded.

The instrumented `akpthread_mutex_lock` is shown in Table 1 and the instrumented `akpthread_mutex_unlock` is shown in Table 2.

Table 1: The instrumented akpthread_mutex_lock procedure.

record	replay
	`replay_try();`
`pthread_mutex_lock();`	`pthread_mutex_lock();`
`calc_lc();`	`replay_next();`

Table 2: The instrumented akpthread_mutex_unlock.

record	replay
`calc_lc()`	`replay_try();`
`mutex_unlock();`	`mutex_unlock();`
	`replay_next();`

4.3.3 Condition Variables

Condition variables should always be used in connection with a mutex. In theory, a procedure on a certain condition variable can only be executed when the corresponding mutex is held. The function

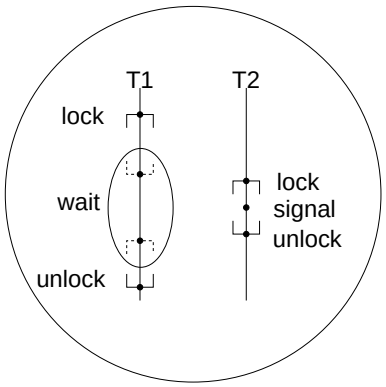

Figure 9: The actual synchronization events during a `pthread_cond_wait` - `pthread_cond_signal` pair.

`pthread_cond_wait` will unlock the mutex and wait for a `pthread_cond_signal` or `pthread_cond_wait`. If a condition is signaled `pthread_cond_wait` will re-acquire the mutex and proceed (see Figure 9). In fact, making a condition true can be regarded as delivering a signal. As such, it is not really a synchronization point; the synchronization should always be performed using the associated mutex. Therefore, it should be sufficient to log the ordering of the mutex operations. For the `pthread_cond_wait` procedure, the "hidden" lock and unlock operations have to be logged, and replayed, as well. Delivering and awaiting the signal is however no longer necessary.

However, it may occur that the `pthread_cond_signal` is done without the lock being held. Although it is not recommended, it is allowed by the Solaris `pthread` library and this possibility is actually used by the ATHAPASCAN-0 implementation. To cope with this problem, the condition variable data structure is modified so that it is possible to indicate whether another thread is waiting during the record phase, in which case the logical clock of the signaling thread is synchronized with the logical clock of the mutex. This is not necessary if no thread is waiting and the signal is lost.

The instrumented `pthread_cond_wait`, `pthread_cond_signal` and `pthread_cond_broadcast` procedures are shown in Tables 3,4 and 5.

Table 3: The instrumented `akpthread_cond_wait` procedure.

record	replay
`calc_lc();`	`replay_try();`
`cond->mutex = mutex /* Indicates that`	`pthread_mutex_unlock();`
`a thread is waiting */`	`replay_next();`
`pthread_cond_wait();`	
`cond->mutex = NULL /* Not waiting`	
`any more */`	`replay_try();`
`calc_lc();`	`pthread_mutex_lock();`
	`replay_next();`

4.3.4 Semaphores

As semaphores are built on top of mutexes and condition variables and these two are properly replayed, there is no need to instrument the semaphore procedures themselves.

Table 4: The instrumented `akpthread_cond_signal` procedure.

record	replay

`if (cond->mutex) calc_lc(cond->mutex)`
`else increment_lc()`
`pthread_cond_signal();`
	`replay_try();`
	`replay_next();`

Table 5: The instrumented `akpthread_cond_broadcast` procedure.

record	replay

`if (cond->mutex) calc_lc(cond->mutex)`
`else increment_lc()`
`pthread_cond_broadcast();`
	`replay_try();`
	`replay_next();`

5 Distributed Memory: Dealing with Promiscuous Messages and Test functions

As the sending and receiving of messages is done using MPI and a mutex is grabbed while executing an MPI function (one single mutex for all the MPI functions), the execution of these functions is serialized. Most MPI implementations are not thread-safe and if this is the case they are usually not thread-aware, that is the use of a blocking communication primitive by one of the threads of a node is likely to block all the other threads of the node. To avoid such blocking, all communication primitives are implemented as non blocking requests. Further progresses of the communications are then managed by a special communication daemon thread, using mainly the `MPI_IProbe` function and an internal data structure called `Wait_Context` (see sections 5.3 and 5.4).

As the mutex operations are traced during the record phase, the same serial ordering will be imposed during the replay phase. Figure 10 shows, on the left, a recorded execution and, on the right, the serialized re-execution. However, this is not sufficient for a deterministic replay: promiscuous receive operations and test functions require special attention. The following sections explain this in greater detail.

5.1 Sending Messages

MPI uses FIFO channels for its communication. This means that if a thread sends two messages to a port[3], the messages will be received in the same order. As the sending of messages by different threads on the same node are serialized this means that a replayed execution will deliver the messages in the order they were sent, *if they were sent by the same node*. As the send events on different nodes are not ordered, and hence not recorded, care should be taken that they are received in the correct order. This requires an intervention at the receiving side and not at the sending side. Therefore, the send operations need not be changed for a record/replay mechanism.

5.2 Receiving Messages

On the receiving side, events are also serialized. This time, they are not serialized by the mutex guarding the MPI functions, but by the mutex (`akDaemonMutex`) of the ATHAPASCAN kernel daemon thread (`akDaemon`). As such, the receive operations (`a0IReceive, a0Receive`,...) are serialized and

[3]The destination of a message is in fact a triple (node,port,tag). But as the couple (port,tag) is regarded as being an identifier for a unique port, this technical report will denote destinations by a (node,port) couple.

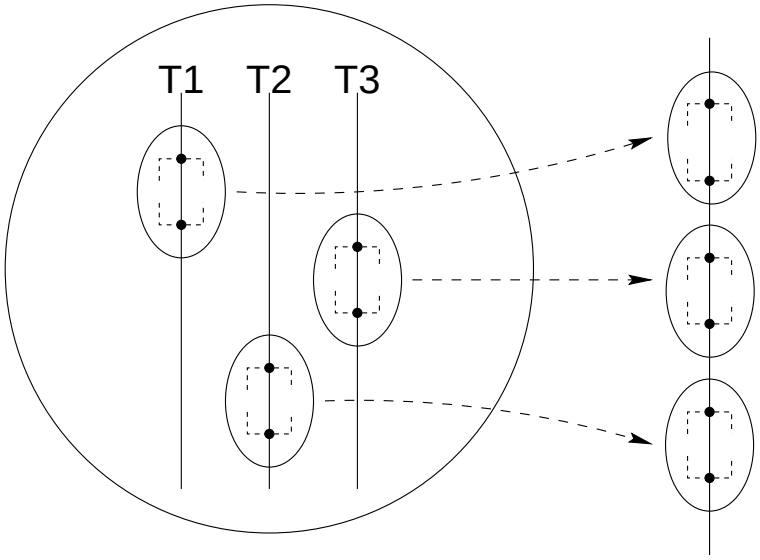

Figure 10: Replaying (right figure) a previously recorded execution (left figure) serializes the synchronization operations in the same order as during the recorded execution.

replayed in the correct order. Note that this will not lead to deadlock during replay. E.g. Figure 11 show three nodes: a thread on the first node sends a message to the third node and a thread on the second node send a message to *another* port on the third node. On the third node, two different threads receive these messages. During the receive operation, the daemon mutex is grabbed, and hence these receive operations are ordered and traced. During replay, the same ordering is imposed on these receive operation. Suppose, during replay, the first thread on the receiving node tries to execute its receive operation first (Figure 12). The replay mechanism will prevent this, and this receive operation will be stalled, till the receive operation of the second thread is executed (or submitted in case of `a0IReceive`). Note that the first thread is stalled *before* grabbing the mutex and entering the kernel. As long as the kernel is concerned, the receive operations are executed in the correct order, and hence the internal kernel structures will be built in the same order as during the original execution.

5.3 Promiscuous Functions

Problems will however arise if the receive operation is promiscuous. This occurs if the receive event does not specify a source node but allows the message to be sent from any node (specified by using the special symbol `A0AnySource` as source node). Figure 13 shows an example: two threads on two different nodes send a message to the *same* port on a third node. On this node, two threads execute a promiscuous receive operation. Therefore, both threads can receive either of the two messages sent. Although the ordering of the receive operations is logged, this is not sufficient as the actual order in which the messages arrive at the nodes is not known. This ordering is only known *and* preserved if the messages arrive from the same node, as shown in Figure 14. In that case a correct replay is possible. If the messages are sent by different nodes, additional information has to be recorded.

At the kernel layer, receive operations are implemented by the `akIRecv` function, which calls `MPI_IRecv`. `MPI_IRecv` was instrumented both to record the source of a message, if it can be sent

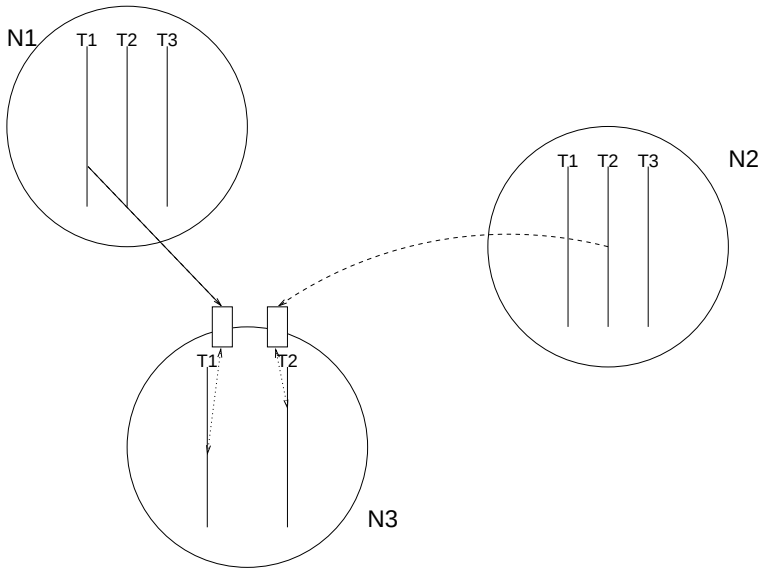

Figure 11: Record phase: two different threads on two different nodes and a message two different ports on a third node.

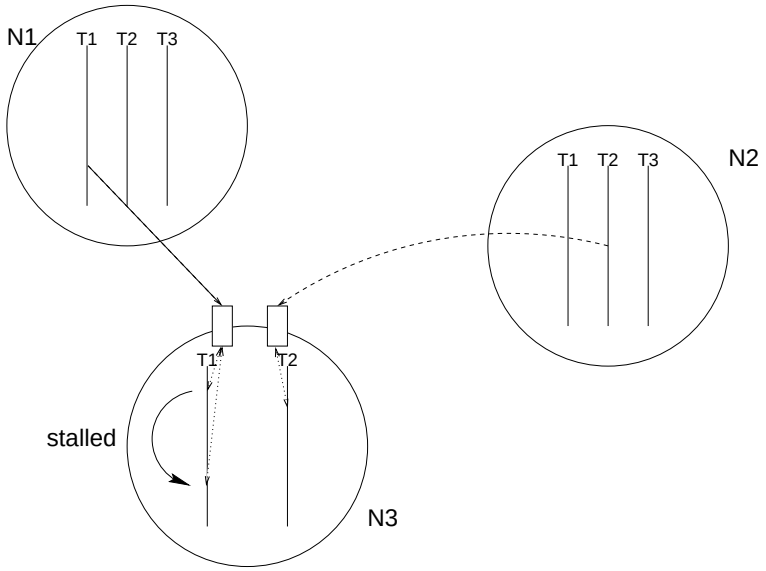

Figure 12: Replay phase: the receive operation by the first thread on the third node is stalled as the receive operation by the second thread has to proceed first.

by any node, and an operation number. The operation number is to be used by subsequent test primitives and stored with the actual source node, in the `Wait_Context` of the associated request, when the message arrives (see section 5.4). Since the arrival of the expected message might take place after a large number of test requests (see section 5.4), it is necessary to *sort* the trace file before replay, so that as soon as the `MPI_IRecv` is performed, the identification of the emitting node, during the recorded execution, can be fetched from the trace file. Provision must also be

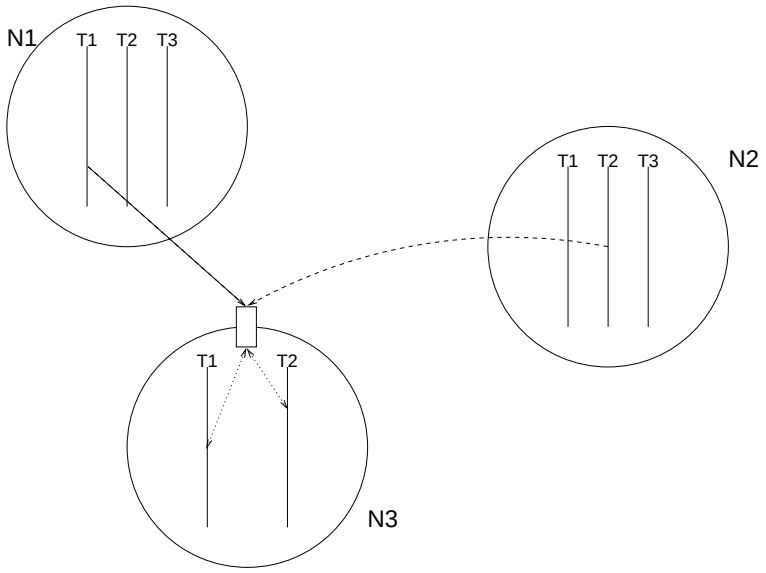

Figure 13: A re-execution that is nondeterministic if one or both receive operations are promiscuous.

made to record the number of tests associated to the request. Table 6 shows the instrumented code for `MPI_IRecv`.

In the text the following functions are used:

`getnumber:` gets a number from the trace file;

`putnumber:` puts a number at the head of the trace file.

`get_operation:` returns a request number to be used as an identifier of an MPI request.

The `MPI_Probe` has a similar behavior to `MPI_Recv` in case the indicated source node is `MPI_ANY_SOURCE`, in which case the source needs to be recorded. Table 8 shows the instrumented code for `MPI_IProbe`.

Other types of "promiscuous" functions are `a0TestAnyRequest`, to test if a request has finished, and `a0WaitAnyRequest`, to wait until a request finishes. These functions do not need to be instrumented since they are implemented by calls to `MPI_Test` requests which are themselves instrumented.

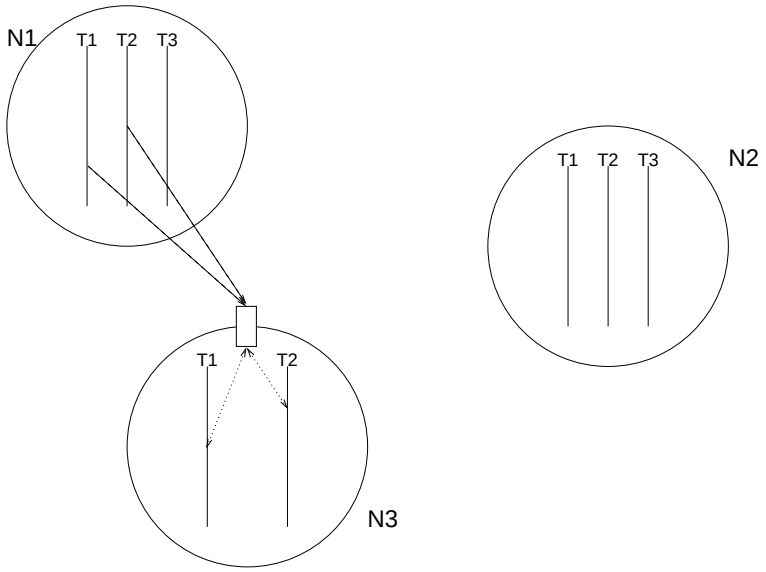

Figure 14: A re-execution that is deterministic, even if one or both receive operations are promiscuous.

Table 6: The instrumented `MPI_IRecv` procedure.

record

```
int RR_MPI_IRecv(..., int source, ..., WaitContext * wcp, ...){
  if (source==MPI_ANY_SOURCE) {
     wcp->operation_No = get_operation();
     wcp->Test_operation=FIRST_TEST; /*No. of tests for this request*/
  }else {
     wcp->RS_operation=DONT_TRACE;
     wcp->Test_operation=FIRST_TEST;
  }
  MPI_IRecv(..., );
}
```

replay

```
int RR_MPI_IRecv(..., int source, ..., aktWaitContext * wcp, ...){
  if (source==MPI_ANY_SOURCE) source=getnumber();
  wcp->Test_operation=FIRST_TEST; /* No. of tests for this request */
  MPI_IRecv(..., source, ....);
}
```

Table 7: The instrumented `MPI_ISend` procedure.

record

```
int RR_MPI_Isend(..., WaitContext * wcp, ...){
     wcp->RS_operation = DONT_TRACE;
     wcp->Test_operation=FIRST_TEST; /*No. of tests for this request*/
  MPI_ISend(..., );
}
```

replay

```
  /* Same as record */
```

Table 8: The instrumented `MPI_Probe` procedure.

record

```
int RR_MPI_Probe(int source, ..., MPI_Status * status, ...){
  MPI_Probe(..., int source, ..., MPI_Status * status, ...);
  if (source==MPI_ANY_SOURCE)
     putnumber(get_operation(), status->MPI_SOURCE);
}
```

replay

```
int RR_MPI_Probe(..., int source, ..., MPI_Status * status, ...){
  if (source==MPI_ANY_SOURCE) source=getnumber();
      MPI_Probe(source, ....);
}
```

5.4 Test Functions

Test functions require special attention. The function `a0TestRequest` verifies if a request is complete. The function `a0TestAnyRequest` tests for the completion of one of several requests. These functions are associated to non blocking communication primitives. The functions return either TRUE or FALSE. In the positive case, the request structure is updated. A typical usage for these functions is loop: one loops until the result is TRUE. For deterministic replay, it is important that the same number of test requests are issued and that the same result is delivered when they succeed.

As before and for the sake of simplicity, the execution replay mechanism is implemented at the `MPI` level. Two `MPI` functions are involved: `MPI_Test` and `MPI_IProbe`. Because of the way communications are implemented in ATHAPASCAN, these two functions are often called by the daemon threads to test the progresses of the pending communications (tests) or the arrival of new messages (probes). In both cases, the number of unsuccessful tests (probes) executed during a record phase needs to be recorded. A simple implementation would assign an operation number to each `MPI_Test` and `MPI_IProbe` function call and record this operation number together with the result of the call (success or failure). To reduce the amount of traces to be recorded, an operation number is associated to the first `MPI_IProbe` or to the request (`MPI_IRecv` or `MPI_ISend` function call) and stored with the number of consecutive unsuccessful tests.

Table 9 shows the instrumented code for the `MPI_IProbe` function. If the `MPI_IProbe` function uses a source `MPI_Any_Source`, it is necessary to trace the source node of the incoming message when the request succeeds.

The `MPI_Test` function requires some form of "communication" with related non-blocking `MPI_ISend` and `MPI_IRecv` communication primitives. This is done by extending the `WaitContext` data type used in the `Athapascan` implementation. Three new fields were added to the `WaitContext` data type:

Test_operation: stores the operation number (allocated by a call to get_operation) of the `MPI_IRecv` function call. This is used to associate the source node – that may only be identified after several `MPI_Test` calls – when source is `MPI_Any_Source`.

Test_number: stores the number of tests already done for the same request.

RS_operation: indicates whether the source node of the received message needs to be traced or not.

Table 10 shows the instrumented code for the `MPI_Test` function.

Table 9: The instrumented `MPI_IProbe` procedure.

record

```
int RR_MPI_IProbe(int source,..., int *flag, MPI_Status *status, ...){
  static unsigned int probe_operation=FIRST_PROBE;
  MPI_IProbe(int source, ..., int *flag, MPI_Status * status);
  probe_number++;
  if (probe_operation==FIRST_PROBE)
    probe_operation=get_operation();
  if (*flag) {   /* flag set by MPI: message received */
    putnumber(probe_operation, probe_number);
    probe_number=0;
    probe_operation=FIRST_PROBE;
    if (source==MPI_ANY_SOURCE)
      putnumber(get_operation(), status->MPI_SOURCE);
  }
}
```

replay

```
int RR_MPI_IProbe(int source, ..., int *flag, MPI_Status * status){
  static unsigned int probe_operation=FIRST_PROBE;
  if (probe_operation==FIRST_PROBE){
    probe_operation=get_operation();/* Used for checking */
    probe_number=getnumber()-1;
    }
  if (probe_number) {  /* no message received */
    *flag=0;
    probe_number--;
   } else {            /* message received */
       if (source==MPI_ANY_SOURCE) source=getnumber();
       MPI_Probe(source, tag, comm,  status); /* blocking probe */
       *flag=1;
       probe_operation=FIRST_PROBE;
     }
}
```

Table 10: The instrumented `MPI_Test` procedure.

record

```
int RR_MPI_Test(WaitContext * wcp, int * flag, MPI_Status *status) {
   MPI_Test(&(wcp->u.request), flag, status);
   if (wcp->Test_operation==FIRST_TEST) {
     wcp->Test_operation=get_operation();
     wcp->Test_number=1;
   }  else wcp->Test_number++;
   if (*flag) {  /* message received/send */
     putnumber(wcp->Test_operation, wcp->Test_number);
     if (wcp->RS_operation!=DONT_TRACE)
       putnumber(wcp->RS_operation, status->MPI_SOURCE);
     wcp->Test_operation=FIRST_TEST;
     wcp->RS_operation=DONT_TRACE;
   }
}
```

replay

```
int RR_MPI_Test(WaitContext * wcp, int * flag, MPI_Status *status){
   if (wcp->Test_operation==FIRST_TEST)
     wcp->Test_number=getnumber()-1;
   else wcp->Test_number--;
   if (wcp->Test_number) *flag=0;
   else {
     MPI_Wait(&(wcp->u.request), status);  /* forced wait */
     wcp->Test_operation=FIRST_TEST;
     *flag=1;  /* message received/send */
   }
}
```

5.5 Encoding of message passing traces

During the recording phase, couples of values are recorded for messages. Each couple includes an operation number, used to sort the trace file at the beginning of the replay phase and a data value which can be:

- the source node of a message in case of successful `MPI_IProbe` with source `MPI_Any_Source` or a successful `MPI_Test` relative to an `MPI_IRecv` with source `MPI_Any_Source`.
- The number of consecutive unsuccessful calls to `MPI_IProbe` or the number of unsuccessful calls to `MPI_Test` relative to the same non blocking request.

In the current implementation, these data couples are stored as plain integers, without any optimization. Would the trace file size become a problem, it should be possible to use compression schemes similar to those used for synchronization traces.

5.6 Trace files

One trace file is produced for each node involved in the computation. In each of the files, data is stored by blocks, each block being produced by a single thread, its size being limited by the size of the trace buffer allocated to each thread. The first two words of each block indicate the thread number and the number of bytes of the block. The message passing traces are stored as "virtual" thread 0 traces. These message passing traces need to be sorted before execution replay. This sorting is done once, at the beginning of the first replayed execution. To limit the number of trace files, the initial and final values of the logical clocks of the threads of the node are stored as "virtual" threads 1 and 2 traces, at the end of the trace file.

6 Conclusion and future work

The ATHAPASCAN-0 execution replay mechanism was tested on several toy programs featuring all possible cases of non determinism of ATHAPASCAN-0. It was also tested on the available program examples of the ATHAPASCAN-0 distribution. It was also used to debug the ATHAPASCAN-1 system, which represents a large ATHAPASCAN-0 program: this was an opportunity to discover some missing features of the current implementation such as being able to save traces of crashing programs.

Up to now, the size of the recorded traces remained limited as well as the overhead of recording. Surprisingly, the replayed executions were faster than the corresponding record execution. Such a phenomenon comes probably from the behavior of the daemon thread. The daemon thread was designed to optimize the performances of communications, that is to reduce their latency. Since it is scheduled by Solaris as an ordinary thread, it takes an important proportion of CPU time for non compute intensive programs. Since most test programs used so far are communication intensive programs, the communication daemon thread is activated frequently. When activated, a large part of its activity is probing communications and testing for the completion of blocking MPI requests. During replayed executions, most of the calls to `MPI_test` or `MPI_probe` are replaced by simple integer comparisons, which might explain the performance gains.

Future work concerns integrating the record-replay in the ATHAPASCAN-0 debugger being built so that tracing and visualization of programs can be performed during replayed executions. Another interesting extension would be to adapt the JiTi race detection tool [34] to other platforms that Sparcs so that it can be used in conjunction with the execution replay on the platforms running ATHAPASCAN-0 or ATHAPASCAN-1 [6] programs.

References

[1] K.M.R. Audenaert and L.J. Levrouw. Interrupt replay: a debugging method for parallel programs with interrupts. *Microprocessors and Microsystems*, 18(10):601–612, December 1994.

[2] David F. Bacon and Seth Copen Goldstein. Hardware-assisted replay of multiprocessor programs. *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, 26(12):194–206, December 1991.

[3] P.-E. Bernard and D. Trystram. Report on a Parallel Molecular Dynamics Implementation. In *Parallel Computing*, Bonn, Germany, May 1997.

[4] J. Briat, I. Ginzburg, and M. Pasin. *Athapascan-0 Reference and User Manuals.* LMC-IMAG, B.P. 53, F-38041 Grenoble Cedex 9, March 1998. http://www-apache.imag.fr/software/ath0/.

[5] J. Briat, I. Ginzburg, M. Pasin, and B. Plateau. Athapascan runtime : Efficiency for irregular problems. In *Proceedings of the Europar'97 Conference*, pages 590–599, Passau, Germany, Aug 1997. Springer Verlag.

[6] Gerson G. H. Cavalheiro, Fran ois Galil e, and Jean-Louis Roch. Athapascan-1: Parallel Programming with Asynchronous Tasks. In *Proceedings of the Yale Multithreaded Programming Workshop*, Yale, USA, june 1998. http://www-apache.imag.fr/ gersonc/publications/yale98.ps.gz.

[7] J. C. Cunha, J. Lourenco, and T. Antao. An experiment in tool integration: the ddbg parallel and distributed debugger. In S. Winter P. Milligan, editor, *To appear in Journal of Systems Architecture, Special Issue on Tools and Environments for Parallel Processing.* Elsevier Science, 1997.

[8] B. de Oliveira Stein and J. Chassin de Kergommeaux. Interactive visualisation environment of multi-threaded parallel programs. In *Parallel Computing: Fundamentals, Applications and New Directions.* Elsevier, 1997.

[9] Perry A. Emrath and David A. Padua. Automatic detection of nondeterminacy in parallel programs. *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, 24(1):89–99, January 1989.

[10] A. Fagot and J. Chassin de Kergommeaux. Formal and experimental validation of a low-overhead execution replay mechanism. In *S. Haridi, K. Ali, and P. Magnusson, editors, Euro-Par'95 Parallel Processing, volume 966 of LNCS*, pages 167–178, August 1995.

[11] A. Fagot and J. Chassin de Kergommeaux. Systematic assessment of the overhead of tracing parallel programs. In *E.L. Zapata, editor, Proceedings of the 4th Euromicro Workshop on Parallel and Distributed processing, PDP'96, Braga. IEEE/CS*, January 1996.

[12] I Foster, C. Kesselman, and S. Tuecke. The nexus approach to integrating multithreading and communication. *Journal of Parallel and Distributed Computing*, 37(1):70–82, 1996.

[13] F. Galil e, J.-L. Roch, G. G. H. Cavalheiro, and M. Doreille. A general modular specification for distributed schedulers. In *Pact'98*, Paris, France., Oct 1998.

[14] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM 3 User's Guide and Reference Manual.* Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831, May 1994.

[15] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[16] Thomas J. LeBlanc and John M. Mellor-Crummey. Debugging parallel programs with Instant Replay. *IEEE Transactions on Computers*, C-36(4):471–482, April 1987.

[17] E. Leu, A. Schiper, and A. Zramdini. Execution Replay on Distributed Memory Architectures. In *Proceedings of the 2nd IEEE Symposium on Parallel and Distributed Processing*, pages 106–112, Dallas, USA, December 1990.

[18] Eric Leu. *La r execution pierre angulaire de la mise au point des programmes parall les.* PhD thesis, cole Polytechnique F d rale de Lausanne, 1992. In French.

[19] Eric Leu, André Schiper, and Abdelwahab Zramdini. Efficient execution replay technique for distributed memory architectures. In *Proceedings EDMCC2, Second European Conference on Distributed Memory Computing*, pages 315–324. Springer-Verlag, 1991.

[20] L.J. Levrouw and K.M.R. Audenaert. Minimizing the log size for execution replay of shared-memory programs. In Bruno Buchberger and Jens Volkert, editors, *Parallel Processing: CONPAR 94 – VAPP VI*, LNCS 854, pages 76–87, Berlin Heidelberg, 1994. Springer-Verlag.

[21] Luk J. Levrouw, Koenraad M. Audenaert, and Jan M. Van Campenhout. Execution replay with compact logs for shared-memory programs. In C. Girault, editor, *Applications in Parallel and Distributed Computing*, IFIP Transactions A-44: Computer Science and Technology, pages 125–134. Elsevier Science B.V., North-Holland, 1994.

[22] Luk J. Levrouw, Koenraad M. Audenaert, and Jan M. Van Campenhout. A new trace and replay system for shared memory programs based on Lamport Clocks. In *Proceedings of the Second Euromicro Workshop on Parallel and Distributed Processing*, pages 471–478. IEEE Computer Society Press, January 1994.

[23] Luk J. Levrouw and Koenraad M.R. Audenaert. Minimizing the log size for execution replay of shared-memory programs. Technical report, Universiteit te Gent, ELIS, Februari 1994.

[24] Message Passing Interface Forum, University of Tennessee, Knoxville, Tennessee. *MPI: A Message-Passing Standard*, May 1994.

[25] Frank Mueller. A library implementation of POSIX threads under UNIX. In *Proc. of the Winter USENIX Conference*, pages 29–41, San Diego, CA, January 1993.

[26] Robert H.B. Netzer. Optimal tracing and replay for debugging shared-memory parallel programs. In *Proceedings ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 1–11, May 1993.

[27] D. Pan and M. Linton. Supporting Reverse Execution of Parallel Programs. In *SIGPLAN/SIGOPS Workshop on Parallel and Ditributed Debugging*, pages 124–129, May 1988.

[28] M. Ronsse and L. Levrouw. An experimental evaluation of a replay method for shared memory programs. In E. D'Hollander, G.R. Joubert, F.J. Peters, D. Trystram, K. De Bosschere, and J. Van Campenhout, editors, *Parallel Computing: State-of-the-Art and Perspectives*, pages 399–406. North-Holland, Gent, 1996.

[29] M. Ronsse and L. Levrouw. On the implementation of a replay mechanism. In L. Bouge, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Proceedings of Euro-Par '96*, volume LNCS 1123, pages 70–73. Springer-Verlag, Lyon, August 1996.

[30] M. Ronsse, L. Levrouw, and K. Bastiaens. Efficient coding of execution-traces of parallel programs. In J. P. Veen, editor, *Proceedings of the ProRISC / IEEE Benelux Workshop on Circuits, Systems and Signal Processing*, pages 251–258. STW, Utrecht, March 1995.

[31] Michiel Ronsse. An efficient implementation of a replay mechanism. In *Parallel Computing: Software, Architectures and Operating Systems.*, pages 112–122. Department of Computer Science, Clausthal, Germany, Noordwijk, May 1997.

[32] Michiel Ronsse. Recplay: an integrated practical record/replay system. Technical Report DG 97-06, ELIS, October 1997.

[33] Michiel Ronsse and Koen De Bosschere. An on-the-fly Data Race Detector for RECPLAY, a Record/Replay System for Parallel Programs. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, page (on CD), Saint-Malo, October 1997.

[34] Michiel Ronsse and Koen De Bosschere. JiTI: Tracing Memory References for Data Race Detection. In *Parallel Computing: Fundamentals, Applications and New Directions.* Elsevier, 1997.

Unit´e de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399