Proceedings of the 1996 Winter Simulation Conference
ed. J. M. Charnes, D. J. Morrice, D. T. Brunner, and J. J. Swain

A TESTBED FOR PARALLEL SIMULATION PERFORMANCE PREDICTION

Alois Ferscha
James Johnson

Institut für Angewandte Informatik
Universität Wien
Lenaugasse 2/8, A-1080 Vienna, AUSTRIA

ABSTRACT

The overwhelming complexity of influencing factors determining the performance of parallel simulation executions demands a performance oriented development of logical process simulators. This paper presents an incremental code development process that supports early performance predictions of Time Warp protocols and several of its optimizations. A set of tools, N-MAP, for performance prediction and visualization has been developed, representing a testbed for a detailed *sensitivity analysis* of the various Time Warp execution parameters. As an example, the effects of various performance factors like the event structure underlying the simulation task, the average LVT progression per simulation step, the commitment rate, state saving overhead, etc. are demonstrated. We show how the scenario management features provided by the N-MAP tool can be efficiently utilized to predict performance sensitivities. For the particular example, the Time Warp protocol, though highly involved, N-MAP was able to predict the performance sensitivity that was measured from the full implementation executing on the Meiko CS-2.

1 INTRODUCTION

A huge variety of Time Warp (TW) parallel simulation (PS) protocols have been proposed in the literature, with a primary focus on correctness issues in their presentation. Performance aspects of the protocols, due to an overwhelming interweaving of influencing factors, have mostly been studied on the basis of abstractions (*models*) of the target execution platform, the TW implementation and the simulation problem. In most of the literature, performance analysis is used to motivate the optimization of the TW protocol with respect to one or more of these concerns, or to assert the performance gain obtained with optimizations. Specifically the question on the relative qualities of TW protocols (in protocol optimization studies) has often been raised, but general rules of superiority cannot be formulated since performance cannot be sufficiently characterized by models.

In the first category of TW performance analysis, all of the three influence factors (target platform, TW implementation, simulation problem) are implicitly or explicitly abstracted into *models*, mostly stochastic models. As an example, in the analysis by Gupta et. al. (1991) event processing time is assumed to be exponentially distributed, time stamps of messages are Poisson distributed in virtual time, the destination logical process (LP) for each message is randomly chosen and equally likely for every LP, etc. Performance investigations based on analytical models often fail to achieve a satisfactory accuracy due to unrealistic assumptions in the modeling process itself, as well as simplifying assumptions that make the evaluation of those models tractable (e.g. symmetry, homogeneity, $M \rightarrow M$ property). Since performance evaluation based on analytical models is prone to modeling errors, only relative trust can be placed in the results obtained (See Gupta et al. 1991 and Akyildiz et al. 1993 for exceptions).

As an improvement, performance investigations have been conducted upon full TW implementations, but operating under synthetic workloads, thus defining a second category of TW performance analysis. In this category, the analysis is based on a *real system* as far as the hardware and TW software is concerned, but the simulation task to be executed in this environment is still an abstraction (*model*) of the real simulation task - leaving just a single source of potential performance analysis bias due to modeling errors. The most prominent synthetic workload model for TW is PHOLD (Fujimoto 1990) which has been widely used in the community to demonstrate the *performance sensitivity* of TW implementations to the (event) structure of the simulation task. Another approach where a mixture of a real simulation task and a model of the execution environment is studied is trace driven simulation. Here, the behavior of a hy-

pothetical environment is studied under the real load.

Finally, in the third analysis category, even full TW implementations with real workload sets often prohibit performance comparisons if different implementation strategies were followed or different target platforms were selected for the execution. Here, performance analysis potentially suffers from an "incomparability" dilemma addressed in (Ferscha and Chiola 1995). There we have developed a performance comparable implementation design, which isolates hardware from implementation and TW protocol related performance influences. A maximum of source code reuse gained from a conditional compilation implementation technique allows for a further reduction of incomparability to an acceptable level.

1.1 The need for performance predicition

In an existential discussion within the parallel and distributed simulation community (Fujimoto 1993), the availability of *performance prediction methods* for PS protocols has been pointed out as being critical for the future success and general acceptance of PS methods in practice (Lin 1993). For example, it is important for a simulationist to be able to evaluate the suitability of certain PS protocols for a specific simulation task *before* substantial efforts are invested in developing sophisticated PS codes. Another aspect is the early evaluation of the anticipated performance of a new PS protocol being developed.

None of the three performance analysis categories above appears adequate for these demands. Models, although able to give fast predictions, are too vague to be meaningfully applied. Analysis that considers full implementations, although accurate, just brings to light the design flaws committed earlier, which are irrevocable or very hard to repair at the time when major part of the development work is already done. In this paper we present a *performance prediction testbed for Time Warp*, N-MAP, designed and implemented to support performance engineering endeavors from the early design phase of TW protocols in order to avoid late and costly re-engineering. The performance engineering activities in a supervisory role escort the development of TW, ranging from *performance prediction* in the early development stages, to *measurements* of performance metrics of the preliminary or final program in the testing, debugging and tuning phase. Implementing TW *incrementally* in N-MAP, i.e. starting from a code skeleton and providing more and more detailed program code towards the full implementation, allows for very early performance based design decisions, systematic investigations of performance sensitivities using an automated scenario manager, and a maximum of code reuse when trying different TW optimizations using an automated version manager. In the next chapter we briefly recall TW performance factors. In Section 3 we develop a TW skeleton in N-MAP, upon which - as a demonstration - we investigate the performance effects of global virtual time (GVT) computation, throttling the optimism via the available memory, and the choice of the size of the checkpointing interval. We have executed the performance scenarios on the Meiko CS-2, and present the results in Section 4.

2 TW PERFORMANCE FACTORS

Collected arguments on the TW performance characteristics and influences have been reported in (Ferscha 1996). (See also Rönngren et al. (1993) for performance issues related to the implementation of Time Warp and some of its protocol optimizations, or Das et al. (1994) for a TW implementation with a minimal amount of event processing overhead.) Here we enlist some of them, outlining the overwhelming complexity of the performance issue:

- **Simulation Task** The structure of events underlying the simulation task exhibit properties such as persistency, concurrency, mutual exclusion, synchronization, causal connectedness, etc., which determine the potential TW performance. TW optimizations often utilize these properties.
- **Partitioning** A paramount TW performance factor is how the global simulation task is decomposed into LPs, and how these are assigned to processors.
- **Target Hardware Raw Performance**: processor speed, communication latency, memory hierarchies/size, cache levels, etc.
- **Communication/Synchronization Model** The target hardware together with several layers of system software influence performance via the routing strategy, policies for multicasting, scattering, buffering, etc.
- **Implementation related Optimizations** "Tricky" implementations of e.g. memory allocation at run time, active messages, interrupts, data referencing etc. can considerably accelerate TW simulations.
- **Protocol related Optimizations** e.g. aggressive/lazy cancellation, lazy re-evaluation, rollback filtering, infrequent/incremental state saving, cancelback, artificial rollback, time windows/buckets, GVT calculation, fossil collection and many others.

- **Partitioning related Optimizations** like the balancement of inter- and intra-LP load, the event-per-message ratio, rollback prevention by blocking, etc. have severe performance impact. Usually, information necessary for partitioning decisions is not available statically, claiming for methods optimizing the execution performance at runtime.
- **Simulation Engine** Yet another source of potential accelerations is the organization and implementation of the event list (binary heaps, splay trees, calendar queues, skip lists, etc.), other data structure manipulations (input queue, output queue, state stack, etc.), time progression, random number generation, etc.

3 IMPLEMENTING TW IN N-MAP

The N-MAP toolset aims at providing the software developer with an integrated environment for the development of performance efficient parallel programs. Starting from a rough description of the program's algorithmic structure in the form of skeletal code, the program is iteratively and incrementally refined by providing a more and more detailed description of the program's component behavior and execution time requirements in each successive development step. Under the constant supervision of performance prediction tools, the thus emerging program source code is tuned and modified in the direction of the most promising implementation strategy ultimately yielding a fully functional, performance efficient parallel program.

In the early development phases of TW implementations, the software developer is confronted with the difficult problem of choosing the TW implementation strategy which is most efficient and suited to the specific simulation task at hand from an huge wealth of possible implementation strategies which, each in its turn, offer a wide spectrum of possible performance optimizations. As a testbed for the development of parallel simulation protocols, N-MAP provides tools for performance prediction which allow the developer to determine the influence of performance critical factors and detect sources of performance loss. Furthermore, a meaningful TW performance analysis demands an in-depth investigation of all performance influencing factors and their interrelationship. For this, the N-MAP *scenario management* comes into play (see Figure 1).

At any point in program development, any variable used in the program source may be declared as a *mutable* and is subsequently handled by the N-MAP scenario manager. Each such mutable may then be assigned a unique value, or a selection of values to be systematically altered in different scenarios by the scenario editor. N-MAP then automatically creates a suite of simulation/execution runs by taking the Cartesian product over all mutable values in which each point in the resulting space represents a specific setting of mutable values.

For gathering performance data, N-MAP defines a standard set of *responses* which represent common performance metrics (e.g. execution time; busy, idle and communication time; packets/bytes sent/received etc.) which may be chosen for inclusion in the scenario. In addition to the standard responses, N-MAP also allows the definition of new, application specific responses.

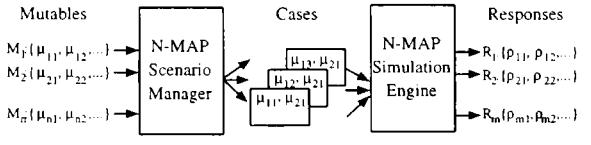

Figure 1: N-MAP Scenario Management

Contrary to other modeling techniques which, as a final result of modeling efforts, yield a *model* of program performance, the N-MAP methodology produces *operable source code* which may be translated into native C code for a variety of parallel platforms as well as for simulated execution on the local uniprocessor by means of N-MAP's built-in translation capabilities.

In the following, a general purpose TW simulator for timed Petri Nets is developed in the N-MAP environment. The principle features of the N-MAP toolset and methodology are demonstrated.

3.1 The Time Warp Task Structure

In the N-MAP environment, program behavior is described in the form of a *task structure specification* (TSS) which defines the sequential stream of task and communication calls to be performed on each processor. The syntax of the TSS is basically that of C with extensions for representing parallelism. The TSS for the TW implementation under investigation is shown in Figure 2.

The code segment labeled `LP` defines `MAXP` logical processors `lp(0)` through `lp(MAXP-1)` operating in an SPMD mode of execution. The `for` loop which encapsulates the whole of the TW source code serves to gather performance data over the number of simulation runs specified in the mutable `RUNS`. Each run begins with the initialization of the simulator and model (i.e. Petri net) in the code segment `INIT`.

During initialization the net description file (specified in the mutable `NETFILE`) is read and partitioned

```
process lp(i) where (i=0:MAXP-1;)                                    LP
{
/* Run several simulations one after the other                        */
for (run=0; run<RUNS; run++)
{
/* Initialize simulator and model */                                 INIT
initialstate(NETFILE,EVENT_POOL);
/* Insert initial internal events into EVL                            */
while(ie=next_ie()) chronological_insert(ie,EVL);
/* Log this state on state stack                                      */
log_new_state();
/* Main simulation loop  */                                          LOOP
while (GVT < ENDTIME)
  {
  /* Read all messages from the input buffer */                      INPUT
  while (m=next_IB())
    {
    /* Straggler message?                                             */
    if (ts_less_than_LVT(m))
      {
      /* Invoke rollback?                                             */
      dual=dual_exists(m,IQ);
      if ((positive(m) && !dual) || (negative(m) && dual))
        {
        /* Rollback to earliest state before timestamp */
        LVT=restore_earliest_state_before(ts(m));
        /* Generate antimsgs resulting from rollback    */
        generate_antimessages(LVT);
        /* Insert antimessages into OQ                  */
        while(ee=next_ee()) chronological_insert(ee,OQ);
        }
      }
    /* Insert message into IQ or annihilate                           */
    if (!remove_dual(m,IQ)) chronological_insert(m,IQ);
    }
  /* Was a GVT packet received? */                                   GVT/FOSS
  if (gvt_packet_received)
    {
    advance_GVT(); /* Calculate GVT estimate                        */

    /* Perform emergency fossil collection if memory                 */
    /* exhausted and checkpointing is used                           */
    if (percent_events_used()>MEMORY_LIMIT&&CHECKP_INTERVAL)
       incr_fossil_collection();
    else
       fossil_collection();
    }
  /* Loop back if insufficient memory */                             GUARD
  if (percent_events_used()>MEMORY_LIMIT) continue;
  /* Get the next event to process from EVL or IQ                   */
  /* If no events to process loop back  to INPUT                    */
  if (!e = get_first_EVL_or_IQ()) continue;
  /* Set LVT to timestamp and event and simulate */                  SIM
  LVT = ts(e);
  simulate_occurrence_of(e);
  /* Insert the internal events generated into the EVL              */
  while(ie=next_ie()) chronological_insert(ie,EVL);
  /* Remove pre-empted internal event from the EVL                  */
  while(ie=next_preempted_ie()) remove_event(ie, EVL);
  /* Insert external events into output queue only if               */
  /* dual does not exist (lazy cancellation)                        */
  while(ee=next_ee())
    if (!dual_update(ee,OQ)) chronological_insert(ee,OQ);
  /* Log this state incrementally onto the stack  */                 LOG
  /* if checkpointing used and checkpoint not yet reached */
  if (!checkpoint && CHECKP_INTERVAL) log_incr_state();
  /* Log the complete state information onto the stack              */
  /* if checkpoint reached or checkpointing not used                */
  if (checkpoint || !CHECKP_INTERVAL) log_new_state();
  /* Fill the output buffer */                                       OUTPUT
  fill_OB(LVT);
  /* Send out messages in the output buffer                         */
  send_out_contents(OB);
  }
/* Free up all memory used during simulation */                      INIT
clean_up();
}
/* Print out performance data                                       */
print_results();
}
```

Figure 2: The Task Structure Specification for TW

among the processors and a pool of free events allocated on each processor. The number of available events (specified in the mutable `EVENT_POOL`) remains constant during simulation thus providing a means for limiting optimism through the limitation of available memory. Model initialization produces a list of initial internal events which are inserted into the event list (EVL) and the resulting initial state is subsequently logged on the state stack. The main simulation loop is then executed until GVT reaches the value contained in the mutable `ENDTIME`.

In the `INPUT` segment, incoming messages are read from the input buffer (IB) and their time stamps are checked against the current LVT. In the case of a causality violation, the rollback mechanism is invoked to restore the first consistent state prior to the time stamp of the straggler message and antimessages generated in the course of the rollback are inserted into the output queue. Finally, each incoming message is inserted into the input queue (IQ) or annihilated.

If a GVT calculation packet has been received during the `INPUT` phase, the `GVT/FOSS` code segment is executed. Using the information contained in the packet, the LP calculates a new GVT estimate, updates its own information in the GVT packet and forwards it to its successor (the GVT algorithm implemented is described in detail below). Based on the new GVT estimate, fossil collection is then performed in the state stack.

The following segment `GUARD` checks if sufficient memory (specified in the mutable `MEMORY_LIMIT`) is available to perform the local simulation of the next event. If not, the simulator loops back to `INPUT` to await the arrival of further messages or GVT calculation packets until sufficient memory is freed through the occurrence of a rollback (arrival of a straggler message) or fossil collection (arrival of a GVT calculation packet). In the case that no events are scheduled for local simulation in the IQ or EVL (i.e. the partition has become depleted of tokens), the simulator also loops back to the `INPUT` segment.

The occurrence of the next scheduled local event is simulated in the `SIM` segment and LVT is set to the occurrence time of event in the model. The model returns three lists of events to the simulator: 1) a list of the new internal events (transition firings) resulting from the occurrence of the event in the model which are to be scheduled for future simulation, 2) a list of previously scheduled events (transition firings) which have now been pre-empted by the occurrence of the event and 3) a list of new external events (token arrivals in other partitions) which must be sent to the respective LPs. The state of the simulator is updated to reflect the occurrence of the event by inserting the internal events into the EVL and removing the pre-empted events. External events are inserted into the OQ or alternately annihilated if a dual message is present in the OQ (lazy cancellation).

The current state of the simulation is saved in the `LOG` segment by copying the EVL to the state stack as well as the state variables used by the model. If checkpointing is enabled (mutable `CHECKP_INTERVAL` set to a value > 0), state information is saved incrementally on the stack between checkpoints and the complete state information only every `CHECKP_INTERVAL` simulation steps.

Finally, messages stored in the OQ having time stamps less or equal to the current LVT are moved to the output buffer (OB) and sent to the respective

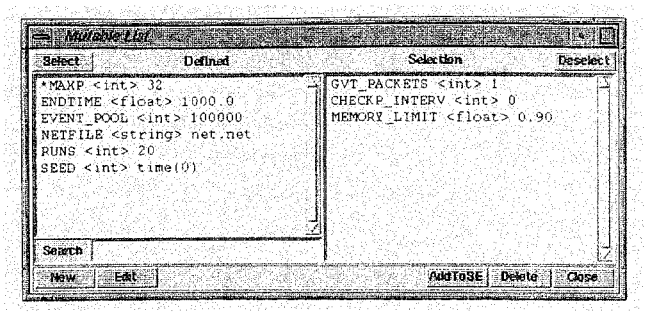

Figure 3: N-MAP Mutables List

LPs in the `OUTPUT` code segment and the simulator loops back to the `INPUT` phase. The current simulation run terminates when GVT reaches `ENDTIME` and all memory used by the simulation is freed before the next run begins. Upon program termination, the performance data gathered over all runs is averaged and the results printed.

3.2 Mutables

The mutables used in the TSS are shown in the *Mutables List* (Figure 3). Here, mutables may be assigned a unique value (left list) or alternately, a subset of mutables selected for variation in a scenario (right list). The following list summarizes the mutables defined in the TSS:

- `MAXP`: number of processors.
- `ENDTIME`: GVT at which simulation terminates.
- `EVENT_POOL`: number of events allocated on each processor during initialization.
- `NETFILE`: name of the Petri net description file.
- `RUNS`: number of simulation runs to execute.
- `SEED`: random number generator initialization.
- `GVT_PACKETS`: number of GVT packets.
- `CHECKP_INTERVAL`: steps between checkpoints.
- `MEMORY_LIMIT`: amount of memory available.

3.3 Execution Time Requirements

A schematic representation of the algorithmic structure and execution time requirements of the TW implementation is shown in Figure 4. The tasks defined in the TSS are grouped according according to their requirement types and the primary run-time factors influencing their execution times.

The simulation of the occurrence of an event in the model as well as GVT calculation are assumed to have fairly *constant* execution times. The group of tasks

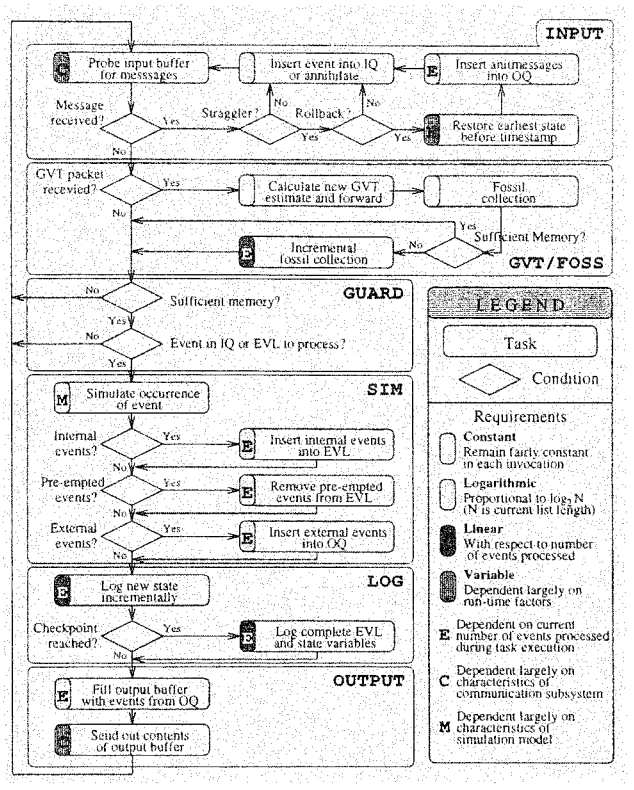

Figure 4: Algorithmic Structure and Execution Requirements of the Time Warp Simulation Protocol

involving list operations can be assumed to be *logarithmic* in their requirements since sorted lists have been implemented as "skip lists" (Pugh 1990) which have optimal element insertion/deletion costs of the order $O(\log_2 N)$, N being the current list length. Furthermore, skip lists have low overall execution times.

A further group of tasks are characterized by *variable* execution requirements since their execution times are dependent largely on run-time factors, e.g. tasks which involve interaction with the communication system or state restoration which is dependent on the "severity" of the rollback. The last group of tasks which deal with state saving and fossil collection can be considered to have *linear* requirements with respect to the number of events processed in the invocation of the task. These linear requirements stem primarily from the fact that during state saving each individual event to be saved must be duplicated and placed on the state stack. Similarly during fossil collection, each event must be freed individually and returned to the pool of free events.

Judging from the structure of the TW implementation under investigation, it appears as if optimization efforts should be concentrated primarily on reducing state saving/fossil collection costs as well as lowering the probability and severity of rollbacks.

3.4 Limiting Available Memory

As a mechanism to limit the optimism in TW, the potential performance gain of memory based throttling (Das and Fujimoto 1994) is investigated, here in the context of a distributed memory implementation of TW. In this approach, whenever an event occurrence is to be simulated, an artificial throttle (mutable `MEMORY_LIMIT`) will delay the execution for a constant amount of CPU time. For example, a value of 0.7 for `MEMORY_LIMIT` will cause the effect of throttle to set in when 70% of available memory has been used. Thus, the willingness of the simulator to execute the next event occurrence is related to the degree of memory exhaustion.

3.5 Checkpointing Interval

The checkpointing method implemented here saves a complete copy of the event list and state variables every `CHECKP_INTERV` simulation steps. Between checkpoints, only those events which have been added to or removed from the event list and state variables which have changed as a result of the simulation of an event are stored on the state stack in a form suitable to allow for subsequent restoration of any intermediate state in the case of a rollback.

The cost and memory requirements for incremental state saving in the case of Petri net simulation (Chiola and Ferscha 1993) is much lower than for complete state saving since each simulation step (transition firing) alters only a small portion of the event list and state variables. Especially when a large number of transitions in the partition have been scheduled for firing in the event list (resulting most likely from a large token population), incremental state saving costs can be considerably lower. For very large checkpointing intervals, however, the coast forward costs incurred during rollbacks may serve to mitigate the positive effect of incremental state saving.

3.6 GVT Calculation

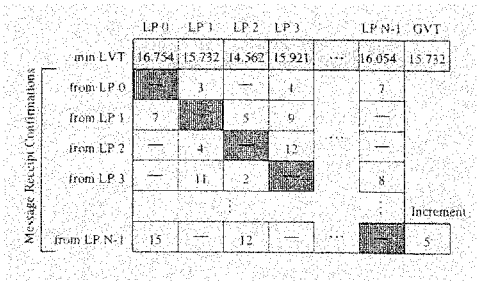

Figure 5: Structure of a GVT Packet

The GVT calculation method implemented uses one or more *GVT packets* which circulate on complete, closed pre-defined paths among the processors. Figure 5 shows the structure of a GVT packet. To account for unprocessed "in transit" messages, outgoing messages are stamped with a sequence number for each output channel (Lin and Lazowska 1990). Upon receipt of a GVT packet, each processor posts message receipt confirmations in the GVT packet on a per channel basis by writing the sequence number of the last packet received on that channel which does not break the complete series of sequence numbers. Each processor can then calculate the set of unconfirmed messages by inspecting the message receipt confirmations posted in the GVT packet by the other processors and then calculate the minimum of its present LVT and the time stamps of all unconfirmed messages (`minLVT`) and post this in its respective column in the GVT packet. A GVT estimate is calculated by taking the minimum of all `minLVT`s in the GVT packet before forwarding the GVT packet to its successor on the path.

The *increment* field of the GVT packet contains an integer which is relatively prime to the total number of processors, `MAXP`, and which each processor adds to its own processor number (modulo `MAXP`) to determine its successor processor in the path. Each GVT packet in the network has a different increment so that no two GVT packets circulate along the same paths. Thus, in a network of 8 processors, the GVT packet with increment 1 travels on the path 1-2-3-4-5-6-7-0-1 whereas the packet with increment 5 is forwarded along the path 5-2-7-4-1-6-3-0-5.

In a network of `MAXP` processors and using one GVT packet, each processor must wait `MAXP-1` GVT calculation steps until it again receives the GVT calculation packet. By allowing 2 GVT packets to circulate in the network, the latency between the receipt of GVT packets is reduced and the frequency of GVT progression is increased on each processor thus allowing for more frequent fossil collections. For large processor counts or for simulations with large memory requirements, a more frequent GVT calculation (i.e. more GVT packets) may be advantageous despite increased communication costs. In other cases, however, a single GVT packet may provide for best performance.

4 SCENARIO EXECUTION ON THE CS-2

Figure 6 shows N-MAP's *Scenario Editor*. The mutables which have been chosen for variation in the scenario (`GVT_PACKETS`, `CHECKP_INTERV`, `MEMORY_LIMIT`) are displayed in the left listbox of the window. Each mutable is assigned a list of values which it is to assume in the scenario in a separate dialog window. The values may be given as a simple white space sep-

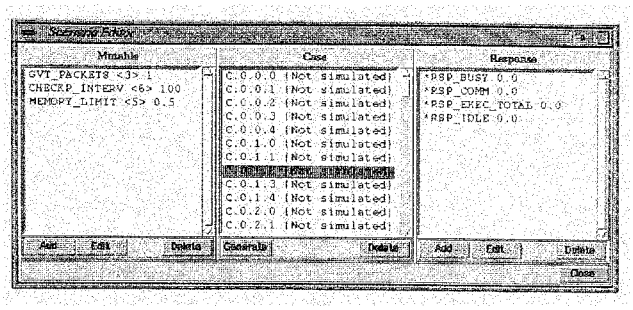

Figure 6: N-MAP Scenario Editor

arated list of values or alternately as a range of values by specifying an upper and lower bound and an increment which may be additive or multiplicative. The following values have been chosen for investigation:

Mutable	Values
GVT_PACKETS	1 2 4
CHECKP_INTERV	0 100 200 300 400 500
MEMORY_LIMIT	0.1 0.3 0.5 0.7 0.9

Figure 7: Scenario Variation of Mutable Values

The Petri net chosen as the simulation task (Figure 8) consists of 64 "regions" linked together in a ring fashion. Each region R_i can be considered to emulate the behavior of a more complex Petri net partition containing one input place P_i, one output transition T_i and a subnet Ψ_i. Both T_i and Ψ_i are infinite servers with exponentially distributed service times $\lambda_{ext} = 0.1$ and $\lambda_{int} = 1.0$ respectively. Initially, each place P_i contains $n = 8$ tokens.

The results of scenario execution are shown in Figure 9. The best results are obtained for the mutables setting `GVT_PACKETS=1`, `CHECKP_INTERVAL=100` and `MEMORY_LIMIT=0.5`.

The use of checkpointing (`CHECKP_INTERVAL>0`) is the most important factor contributing to performance improvement. The resimulation cost incurred during rollback or are obviously insignificant compared to the state saving and fossil collection costs

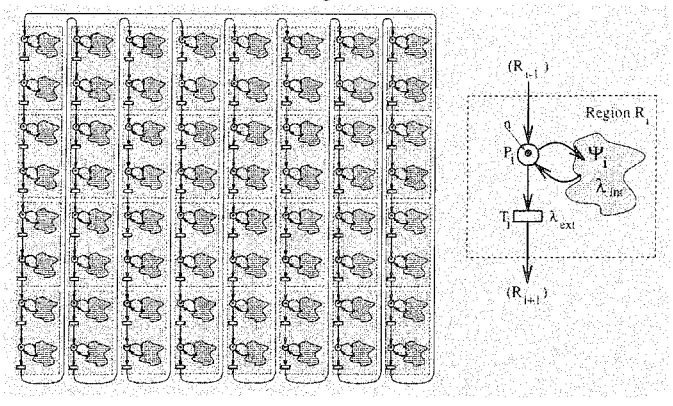

Figure 8: A Petri Net Simulation Task Consisting of 64 "Regions" Mapped onto 32 Processors

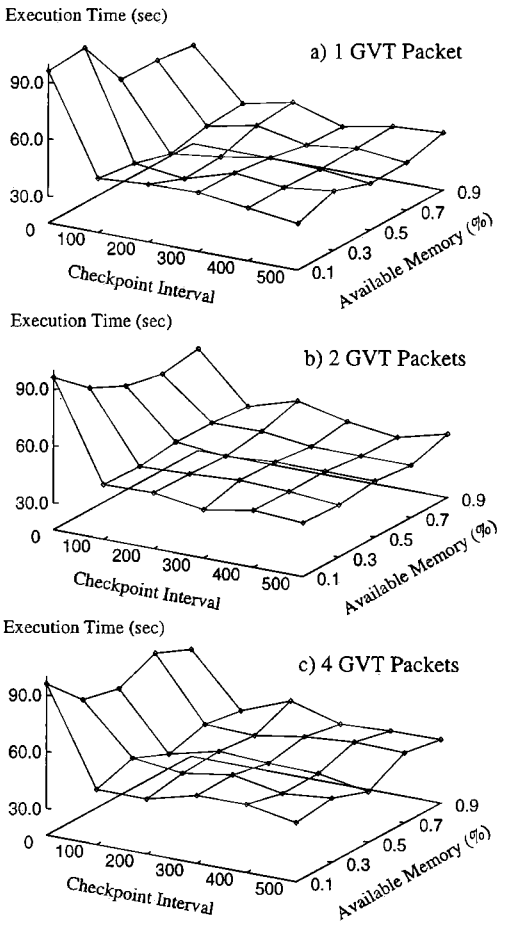

Figure 9: Results of Scenario Execution on CS-2

avoided through the use of checkpointing. The actual checkpointing interval used (100-500) appears to have little effect on performance. Point investigations for the mutable settings `GVT_PACKETS` = 1, `MEMORY_LIMIT` = 0.5 with `CHECKP_INTERVAL` = 1000, 2000, 5000 showed significant performance loss only after an interval of 2000.

In most cases, performance drops only slightly by using more `GVT_PACKETS` although an increased number of packets, providing for a more frequent GVT calculation, does serve to improve performance in cases of high memory requirements (`CHECKP_INTERVAL=0`) and where memory is very limited (`MEMORY_LIMIT` < 0.5). For this particular simulation task partitioned and mapped onto 32 processors, however, the GVT calculation frequency achieved with one GVT packet appears quite sufficient.

5 CONCLUSIONS

The availability of *performance prediction methods* and *tools* for parallel simulation protocols is without

any doubt critical for the future success and general acceptance of parallel simulation in practice. For a simulationist it is of utmost importance, to to be able to evaluate the suitability of a certain parallel simulation protocol for a specific simulation task, for a certain multiprocessor system and a certain operational environment *before* substantial programming efforts are invested.

A performance prediction methodology and set of tools, the N-MAP testbed, has been developed, easing performance engineering endeavors of PS protocols from the early design phase in order to avoid late and costly re-engineering. As a testbed, N-MAP in a fully graphical user interface offers very early performance based implementation design decisions, systematic investigations of performance sensitivities using an automated scenario manager, and a maximum of code reuse when trying different TW optimization using an automated version manager. N-MAP is publicly available.

To demonstrate some of the features of the testbed, we have investigated the performance effects of GVT computation, of throttling the optimism via the available memory, and of the choice of the size of the checkpointing interval in a distributed memory implementation of Time Warp. Performance scenarios were defined and executed on the Meiko CS-2.

ACKNOWLEDGEMENTS

This work was partially supported by the Oesterreichische Nationalbank under grant No. 5069, and the Human Capital and Mobility program of the EU under grant CHRX-CT94-0452 (MATCH).

REFERENCES

Akyildiz, I. F., L. Chen, R. Das, R. M. Fujimoto, and R. F. Serfozo. 1993. The effect of memory capacity on Time Warp performance. *Journal of Parallel and Distributed Computing*, 18(4):411–422.

Chiola, G., and A. Ferscha. 1993. Distributed simulation of timed Petri nets: Exploiting the net structure to obtain efficiency. In *Proceedings of the 14th Int. Conf. on Application and Theory of Petri Nets 1993*, ed. M. Ajmone Marsan, 146–165. Lecture Notes in Computer Science, Springer Verlag, Berlin.

Das, S., R. Fujimoto, K. Panesar, D. Allison, and M. Hybinette. 1994. GTW: A Time Warp system for shared memory multiprocessors. In *Proceedings of the 1994 Winter Simulation Conference*, ed. J. D. Tew and S. Manivannan, 1332–1339.

Das, S., and R. M. Fujimoto. 1994. An adaptive memory management protocol for Time Warp parallel simulation. In *Proc. of the 1994 ACM Sigmetrics Conf. on Measurement and Modeling of Computer Systems*, 201–210. ACM.

Ferscha, A. 1996. Parallel and distributed simulation of discrete event systems. In *Parallel and Distributed Computing Handbook*, ed. A. Y. Zomaya, 1003–1041. McGraw-Hill.

Ferscha, A. and G. Chiola. 1995. Adaptive distributed simulation of Petri net models. In *Proceedings of 1995 Summer Computer Simulation Conference (SCSC '95)*.

Fujimoto, R. M. 1990. Performance of Time Warp under sythetic workloads. In *Proceedings of the SCS Multiconference on Distributed Simulation*, ed. D. Nicol, 23–28.

Fujimoto, R. M. 1993. Parallel discrete event simulation: Will the field survive? *ORSA Journal of Computing*, 5(3):218–230.

Gupta, A., I. Akyildiz, and R. Fujimoto. 1991. Performance analysis of Time Warp with multiple homogeneous processors. *IEEE Transactions on Software Engineering*, 17(10):1013–1027.

Lin, Y-B. 1993. Will parallel simulation research survive? *ORSA Journal of Computing*, 5(3):236–238.

Lin, Y-B. and Lazowska, E. 1990. Determining the global virtual time in a distributed simulation. In *1990 International Conference on Parallel Processing*, (III)201–209.

Pugh, W. 1990. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–677.

Rönngren, R., R. Ayani, R. M. Fujimoto, and S. R. Das. 1993. Efficient implementation of event sets in Time Warp. In *Proceedings of the 7th Workshop on Parallel and Distributed Simulation*, ed. R. Bagrodia and D. Jefferson, 101–108. IEEE Computer Society Press, Alamitos, Calfornia.

AUTHOR BIOGRAPHIES

ALOIS FERSCHA is an Associate Professor at the Department of Applied Computer Science and Information Systems at the University of Vienna, Austria. His current research interests include performance modeling and prediction, computer aided performance engineering of parallel software, distributed simulation and neural networks.

JAMES JOHNSON has been a Research Assistant the Department of Applied Computer Science and Information Systems at the University of Vienna since 1994. His research interests are in tools for computer system performance analysis.