Performance-Oriented Development of Irregular, Unstructured and Unbalanced Parallel Applications in the N-MAP Environment

Alois Ferscha and Allen Malony

CIS-TR-95-10
March 1995

Submitted for review to:

Joint conference

PERFORMANCE TOOLS '95 AND MMB'95

8TH International Conference on Modelling Techniques & Tools for Computer Performance Evaluation

8th Conference on Measuring, Modelling & Evaluating Computing & Communication Systems

September 20-22, 1995
Heidelburg, Germany

Department of Computer and Information Science
University of Oregon

Performance-Oriented Development of Irregular, Unstructured and Unbalanced Parallel Applications in the N-MAP Environment

Alois Ferscha*

Institut für Angewandte Informatik
Universität Wien
Lenaugasse 2/8, A-1080 Vienna, AUSTRIA
Email: ferscha@ani.univie.ac.at

Allen D. Malony

Computer Science Department
University of Oregon
Eugene, OR 97403, U.S.A.
Email: malony@cs.uoregon.edu

Abstract

Performance prediction methods and tools based on analytical models often fail in forecasting the performance of real systems due to inappropriateness of model assumptions, irregularities in the problem structure that cannot be described within the modeling formalism, unstructured execution behavior that leads to unforeseen system states, etc. Prediction accuracy and tractability is acceptable for systems with deterministic operational characteristics, for static, regularly structured problems, and non-changing environments.

In this work we present a method and the corresponding tools that we have developed to support a *performance-oriented* development process of parallel software. The N-MAP environment incorporates tools for the specification and early evaluation of skeletal program designs from a performance viewpoint, providing the possibility for the application developer to investigate performance critical design choices far ahead of coding the program. Program skeletons are incrementally refined to the full implementation under N-MAP's performance supervision, i.e. the real code instead of an (analytical) performance model is "engineered". We demonstrate the use of N-MAP for the development of a challenging application with extensive irregularities in the execution behavior, unstructured communication patterns and dynamically varying workload characteristics, thus resisting an automatic parallelization by a compiler and the respective runtime system, but also being prohibitive to classical "model based" performance prediction.

Keywords: Performance Prediction, Parallel Programming, Task Level Parallelism, Irregular Problems, Parallel Simulation, Time Warp, CM-5, Cluster Computing.

*This work was conducted while Alois Ferscha was visiting the University of Oregon, Computer Science Department, supported by a grant from the Academic Senate of the University of Vienna and the Department of Computer and Information Science at the University of Oregon. The research is also supported by a NSF National Young Investigator (NYI) award.

1 Introduction

Early approaches of relating performance engineering activities to the development process of parallel software appeared when parallel machines started to reliably generate computational results [22, 23, 20]. Early work in the performance analysis of parallel systems mainly focused on the mechanics of empirical analysis: measurement and monitoring, automatic probe insertion, trace generation, post execution trace analysis, perturbation analysis [21] and trace visualization [16]. It was soon recognized that starting with performance debugging activities after fully functional executable code has been developed, reduces the degrees of freedom for program modifications to those that do not require significant changes to the original program structure [9]. Large-scale code redesigns for performance optimization are costly. Hence, a large body of work in classical *performance analysis* appeared to offer alternative engineering solutions, contributing markovian stochastic process, queueing network, timed Petri net, series parallel task graph, etc. *models* to different parallel performance problems. Modeling has been applied to understand performance interdependencies of hardware resources [19], to characterize the behavior of parallel program components, and to abstract the workload for parallel systems [3]. Also the integration of parallel program, multiprocessor hardware and mapping factors for performance evaluation have been reported. The natural abstraction process inherent in the design and use of performance models lends itself to a *model-based* performance analysis approach to performance engineering that can be induced in earlier phases of parallel program development.

Recent success in integrating performance engineering into the parallel software development process can be found in the application of parallel performance prediction tools. Performance estimates obtained from these tools can be shown to provide feedback to high level design considerations, such as choice of partitioning strategy, that often are difficult, if not impossible to obtain from traditional parallelizing compiler approaches, where the fully specified source code is required. However, prediction accuracy and tractability often depends on specific assumptions about the target system and on the *deterministic* operational characteristics of (mostly) *static* problems executing in *regularly structured*, non-changing environments [8].

In practice, however, applications fail to achieve the performance predicted by models. There are many possible reasons: inappropriateness of model assumptions (like Markovian timing), irregularities in the problem structure that cannot be described within the modeling formalism, unstructured execution patterns of parallel applications, widely nondeterministic behavior in the software layers of runtime environments, data dependencies determining the performance of memory hierarchies, and unforeseen dynamics of interacting parts and layers of the application at run time. In general, performance prediction accurate enough to rank different implementation alternatives of an application based on program or workload *models* is just not feasible. To overcome the limitations of the pure modeling approach for parallel performance engineering, a technique for the performance behavior prediction of parallel programs based on *real* (skeletal) *codes* rather than on models has been proposed [12]. A set of tools for an incremental development process of parallel SPMD programs driven by performance engineering activities, called the N-MAP (N-(virtual) processor map) toolset, has been developed [13], following the idea of launching performance en-

gineering activities as early as possible in the development process, namely in the implementation design phase. N-MAP supports the specification and early evaluation of skeletal program designs from a performance point of view, such that performance critical design choices can be investigated far ahead of the full coding of the application. N-MAP aims to discover flaws that degrade performance in a parallel program as early as possible, based on a skeletal representation covering just those program structures that are most responsible for performance. Program skeletons can be provided very quickly by the application programmer and are incrementally refined to the full implementation under N-MAP's performance supervision. We call this implementation strategy *performance oriented* parallel program development.

1.1 Problem Classes

In previous work [13], we have shown the successful use of N-MAP in the context of numerical applications with a static structure of task occurrences and regular communication patterns. By incrementally developing the implementation of parallel Householder reduction algorithms [2] we have demonstrated that very early performance predictions are possible with very little specification efforts, and that arbitrary prediction accuracy could be achieved by adding more and more detailed functionality specifications of the application. In this paper, we demonstrate N-MAP abilities in cases where conventional program model based performance prediction methodologies fail. Specifically, problems with the following properties will be addressed:

Irregular. Many important scientific applications like molecular dynamics codes, fluid dynamic solvers, Monte Carlo codes, multigrid solvers, etc. makes extensive use of indirectly indexed arrays in data parallel embeddings of HPF, Vienna Fortran or Fortran-D codes onto distributed memory multiprocessors. Since an efficient address translation cannot be generated at compile time, compiler runtime support libraries for automatic data partitioning, for sharing data along partitioning boundaries, for communication cost optimization and for automatic dynamic load balancing are necessary for gaining reasonable performance on irregular problems [17]. However, a general purpose compiler, even together with a specialized runtime system, cannot be expected to deliver good or near optimum performance on problems that do not reflect regular data reference patterns, thereby leaving open a set of applications where a direct programming approach still remains the most promising one. In this work, we will demonstrate the tool-support provided by N-MAP for the development of an application with extensive irregularities.

Unstructured. Many common techniques for the acceleration of solving partial differential equations involve irregular meshes. For instance, the multigrid method employs regular meshes, but with different resolution at different levels, and the multiblock method, couples regular meshes in an irregular arrangement [1]. In both cases, non-unitary data movements caused when shifting between different multigrid levels or updating the boundaries among adjacent blocks intrude the independent local computations (on a multigrid level or block), which are basically regular. Since the communication patterns induced by the data references do not exhibit any particular structure,

we refer to these kind of problems as "unstructured". The problem to be studied using N-MAP does not exhibit any kind of communication pattern whatsoever, representing a stress case for previous performance prediction methodologies.

Unbalanced. For certain simulation applications, like N-body codes [25], that simulate the movement of objects (particles) through a bounded area of the k-dimensional space, subject to mutual forces and adhesion influenced by the respective masses, velocities etc., data reference locality is exhibited, but patterns of data interaction change over time. Load balancing must be dynamic and is determined by the specific characteristics of the simulated system. In these cases, it is not the source code of the program, but the characteristics of the system to be simulated (or in other words the program input) that needs to be understood and exploited to generate good performance decisions. We shall present how N-MAP, by automated scenario management, lets the user figure out the sensitivity of parallel program performance to program input, requiring merely an abstraction of the input data.

This paper will present the N-MAP approach for performance engineering parallel applications with unstructured communication and computation behavior. In Section 2 we will present the aims and the methodological approach implemented in the N-MAP tool, and how early performance and behavior predictions are generated. Section 3 is devoted to the N-MAP capabilities for automated scenario management and performance sensitivity analysis. In Section 4 we demonstrate how N-MAP copes with the crucial problem of predicting the performance of statically nondetermined problems. As a stress test case we have chosen to study a parallel simulation protocol (Time Warp) in the context of Petri net simulation models. A detailed *sensitivity analysis* will work out the performance influence of the various Time Warp execution parameters (simulation model, LVT progression, rollback costs, message load, internal event simulation costs, etc.).

2 The N-MAP Toolset

2.1 N-MAP: Rationale and Environment

As a *performance oriented* parallel program development environment, the goal of N-MAP is to provide performance and behavior prediction for early program specifications, namely "rough" but quick specifications of program skeletons that reflect the constituent and performance critical program parts of eventually fully operable program code. The N-MAP specification language is based on C, and includes language constructs for intuitive components of parallel programming like *tasks*, *processes* and (communication) *packets*. By providing direct language support for parallelism at the task level, N-MAP encourages a top-down, stepwise refinement approach of program development. Initially in this process, describing the task behavior and functionality is of lesser importance than capturing its performance impacts; with progressive refinements of the specification, full functionality will eventually be reached.

A *task* in the N-MAP sense is self contained, contiguous block of program code. Its functionality is described like an ordinary C-procedure (called the *task behavior*), whereas the quantification of

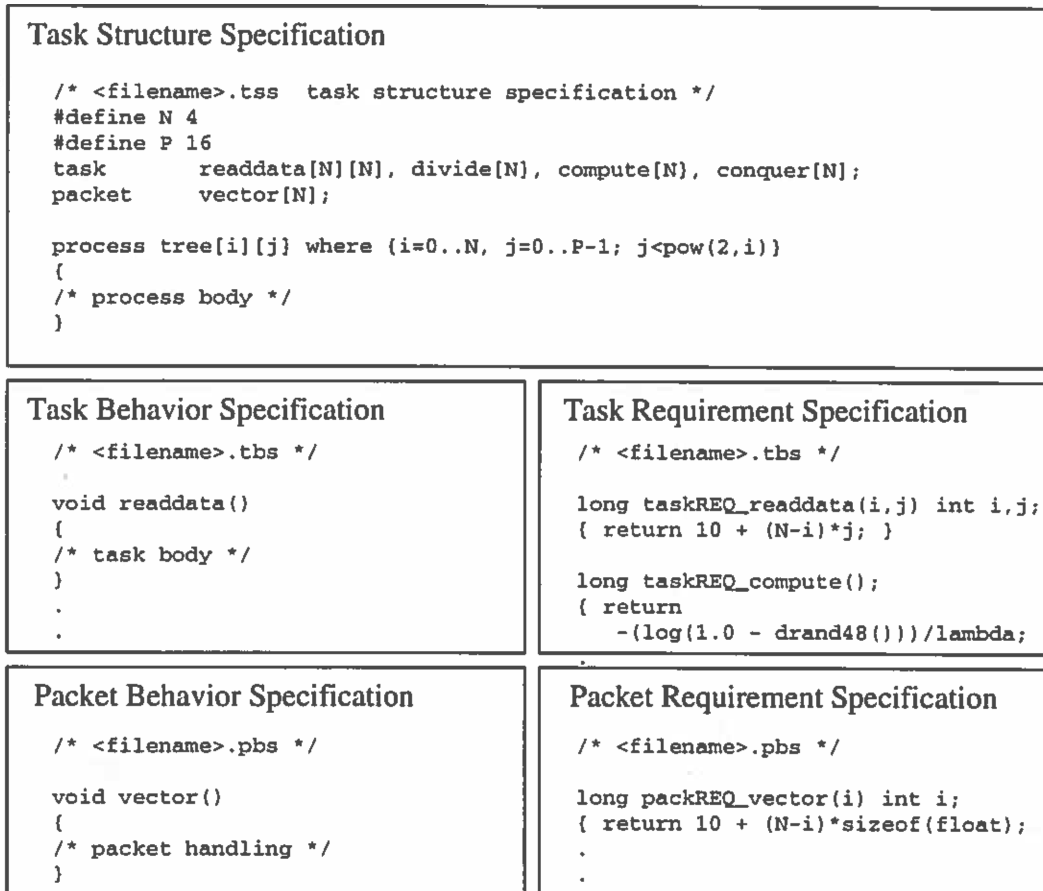

Figure 1: Specifications in N-MAP

its real or expected execution time is expressed with a C-function (the *task requirements*). The task requirements reflect the resource demands for executing the task. A task can implement computation or communication as its behavior. An ordered set of tasks defines a *process*, which appears as a stream of task calls. Since the mental model of a process being a piece of code to be assigned to a physical processor is appropriate, we often refer to processes as *"virtual processors"*. Communication objects, are referred to as *packets* (i.e. the actual data to be transferred among virtual processors). Similar to tasks, packets are characterized by their *behavior* (functionality to gather and pack the data), and their *requirements* (quantification of the amount of data to be transferred among virtual processors). The respective specification sources representing the input to the N-MAP compilation and simulation system are shown in (Figure 1). The

`tss` (Task Structure Specification) describes the application skeleton in terms of *processes*, *tasks* and *packets*,

`tbs`, `pbs` (Task Behavior Specification) and (Packet Behavior Specification) contains C-code for the task and actions to be taken (buffer allocation, data transfer to buffer, etc.) in order to

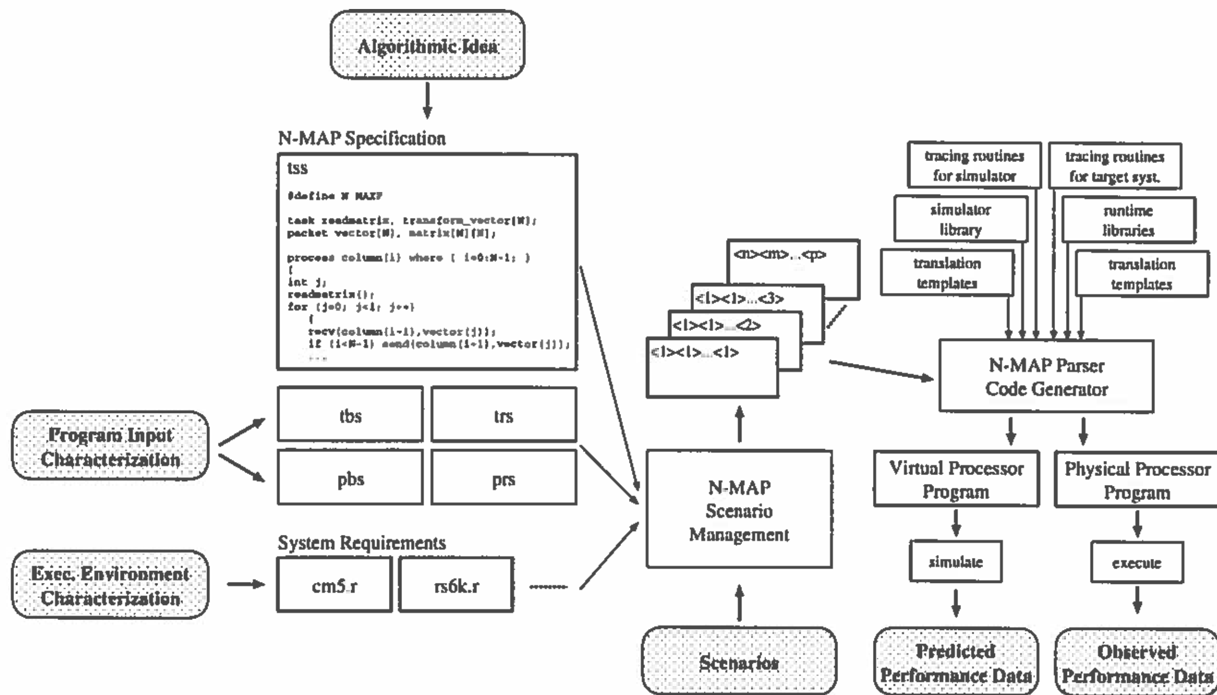

Figure 2: The N-MAP Compilation and Simulation System

send a packet,

trs, **pbs** (Task Requirements Specification) and the (Packet Behavior Specification) express as a C-function the anticipated resource consumption in terms of execution time or message propagation time.

Note that requirements for tasks with non-deterministic execution times can be described by random variates (e.g. exponentially distributed in task `compute()`), and that requirements can be made dependent on parameter values coming from the calling virtual processor (e.g. the requirement for sending a vector (in terms of bytes to be transferred) in `packREQ_vector(i)` is made dependent on the actual vector length `i`).

The tool architecture of N-MAP is depicted in Figure 2. The N-MAP specification sources **tss**, **tbs**, **pbs**, **trs** and **pbs**, together with a system requirements specification, are parsed and translated into a discrete event simulation program. The execution of this simulation program generates the behavior of the parallel program as if it were executed on a set of N virtual processors, and performance data are collected to analyze this behavior. The statistics computed include busy, overhead and idle time for each virtual processor, as well as the number of messages/bytes sent/received. Additionally, trace files can be generated that meet standard file formats (PICL [15]), giving access to standard analysis visualization tools (e.g. ParaGraph [16]), which can further characterize the simulated behavior. It should be noticed at this point that the N-MAP compilation system merely requires the **tss** to be available to generate the simulated execution behavior, allowing a very fast performance prediction and visualization from the program skeleton. If in further development

steps program information is provided to N-MAP via tbs, pbs, trs and pbs, the prediction accuracy can be subsequently improved. Once the characteristics of a certain execution environment are described in terms of <sys>.r file (e.g. cm5.r or rs6k.r shown in Figure 2), N-MAP will automatically make use of this information to mimic a parallel execution of the program on that platform.

A thread mechanism is used in N-MAP to simulate the behavior of the virtual processors. Fast context switching calls (setjmp() and longjmp()) transfer control from one "virtual processor" thread to the other, which is necessary whenever a simulated message is to be transferred among them. Because each virtual processor is executing in its own context, no adjustments must be made to the program code to be simulated in order to allow for concurrent operation. Tasks for which full code has been developed in the respective tbs are executed directly and the *actual* execution time as measured during the simulation is used to progress the simulated time. With this "simulated direct execution" N-MAP achieves excellent prediction accuracy for CM-5 codes on a Sun Sparc workstation [13].

Methodologically, a parallel application is developed in the N-MAP environment by firstly expressing the "algorithmic idea" (i.e. the intended implementation strategy) in the tss using the constructs of the N-MAP language. In a second step, the program input is characterized and related to the tss qualitatively in the tbs and pbs, as well as quantitatively in the trs and pbs; again, immediate feedback can be produced after each incremental extension of the N-MAP specification. To make performance predictions for a concrete target platform (instead of an idealized set of virtual processors), an execution environment characterization can be enlisted by providing a system requirements file. Naturally, the tss can be modified at any point of the development process, should it turn out that the implementation strategy is not promising a performance efficient application. Thus, the developer can immediately react to potential performance pitfalls, even if detailed program code is yet developed.

While this methodology has proven to be a powerful one in its application to regularly structured problems, the difficulties of specifying irregular, unstructured, and unbalanced computations and predicting their behavior challenges the N-MAP process. As a stress test of N-MAP, we choose the Time Warp application and attempt to determine implementation functions from multiple experiments for different system scenarios that would enable a prediction of a high performing Time Warp solution. This work is described below.

2.2 Time Warp

Time Warp [18] is a well established parallel and distributed discrete event simulation (DDES) protocol, aiming to accelerate simulations that take exceedingly long to execute on a single processor. Basically a global simulation task is divided into a set of subtasks, each of which is executed by a *logical process* (LP) that simulates occurrences of events in a spatial subspace of the space-time. Time Warp as a synchronization protocol that can guarantee that the partial event ordering as generated by the individual asynchronous LPs is *consistent* with the total event ordering that would have been generated by a sequential event driven simulator, can automatically make use of

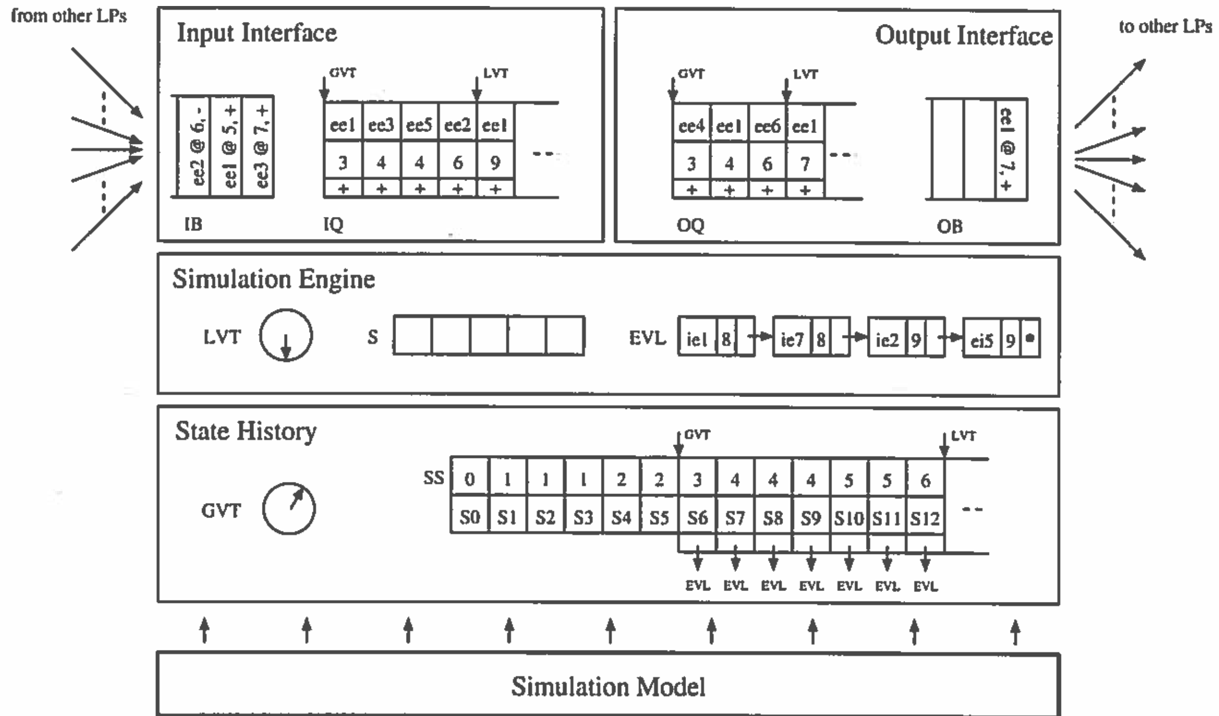

Figure 3: Architecture of Logical Process Executing the Time Warp Protocol

the parallelism among events in the different subspaces.

The operational principle of a single LP is basically the same as for a sequential simulation engine: A set of variables S representing the current state of the simulated system is modified over simulated time due to event occurrences, the only difference being that each LP_i in Time Warp has access only to a (static) subset of the state variables $S_i \subset S$ (disjoint to state variables assigned to other LPs). An LP executes a sequence of event occurrences (potentially modifying the local state variables) prescheduled in an chronologically ordered *event list* (EVL), thus advancing a *local clock* (local virtual time, LVT). Two kinds of events are processed in LP_i: *internal events* which have causal impact only to $S_i \subset S$, and *external events* that also affect $S_j \subset S$ $(i \neq j)$ the local states of other LPs. A communication interface (CI) attached to each LP takes care for the propagation of effects causal to events to be simulated by remote LPs, and the proper inclusion of causal effects to the local simulation as produced by remote LPs. The main mechanism for this is the sending, receiving and processing of (event-)messages piggybacked with copies of the senders LVT at the sending instant. As depicted in Figure 3, CI has two components, the input interface (II) accepts messages from remote LPs, while the output interface (OI) takes care of the distribution of messages concerning remote LPs. The simulation engine (SE) itself keeps and maintains the local state variables S, LVT and EVL.

A locally induced mechanism for causality preservation across LPs called *rollback* is employed to take care of proper synchronization of LPs with respect to timestamps of events as carried by

messages. If an external event arrives out of chronological order (i.e. having a timestamp less than LVT, *straggler message*), then the Time Warp scheme *rolls back* to the most recently saved state in the simulation history consistent with the time stamp of the arriving external event, and restarts simulation from that state on as a matter of causality violation correction. Rollback therefore requires a record of the LP's history with respect to the simulation of internal *and* external events. To keep sufficient internal state information the implementation of a *state stack* SS which allows for restoring a past state is appropriate. The messages sent and received can be administrated in an *input queue* IQ and an *output queue* OQ respectively. For reasons to be seen, this logging of the LP's communication history must be done in chronological order. Since the arrival of event messages in increasing time stamp order cannot be guaranteed, two different kinds of messages are necessary to implement the synchronization protocol: first the usual external event messages ($m^+ = \langle ee@t, +\rangle$), (where ee is the external event and t is its timestamp) which we will subsequently call *positive* messages. Opposed to that are messages of type ($m^- = \langle ee@t, -\rangle$) called *negative-* or *antimessages*, which in case of rollback are transmitted among LPs as a request to annihilate the prematurely sent positive message containing ee, for which it meanwhile turned out that it was computed based on a causally erroneous state. In our example in Figure 3, the message $\langle ee2@6, -\rangle$ residing in the LP's IB is supposed to annihilate the dual positive message $\langle ee2@6, +\rangle$ previously received and stored in the IQ.

The Time Warp protocol in every LP mainly loops over four algorithm parts: (i) an input-synchronization part that percepts messages from other LPs, (ii) a local event processing part that simulates event occurrences, (iii) the propagation of external effects via the sending of locally generated messages, and (iv) the (global) confirmation of locally simulated events, i.e. the verification for each event whether it can become the subject of a future rollback or not. To assure from which state in the history (and back) computations can be considered fully committed, the determination of the value of *global virtual time* (GVT) is necessary, which is the minimum over all LVTs and all timestamps of messages currently in transit. With the knowledge of GVT, a procedure called *fossil collection* can return memory space used by history records that will no longer be used by the rollback synchronization mechanism. The Time Warp protocol terminates the execution of all LPs once GVT has reached a predefined simulation endtime.

2.3 Time Warp Performance

Due to an overwhelming degree of interweaving factors influencing the performance of the Time Warp protocol, a general statement about its performance behavior cannot be formulated [14]. A few of those factors are:

Event Structure of Simulation Model Certainly, overall performance is determined by the amount of model parallelism inherent to simulation model. Specifically, the rollback mechanism is known to be prone to inefficient behavior in situations where event occurrences are highly dispersed in space and time. Such "imbalanced" simulation models can yield recursive rollback invocations over long cascades of LPs which will eventually terminate. An excessive amount of local and remote state restorations is the consequence of the annihilation requests

for effects that have been diffused widely in space and too far ahead in simulated time, consuming considerable amounts of computational, memory and communication resources while not contributing to the simulation as such. This pathological behavior is referred to as *rollback thrashing*, and is influenced by the simulation model.

Partitioning Dividing the global simulation task into subtask therefore is crucial for avoiding the possibilities of rollback thrashing. If partitioning can be done such that the average LVT progression as seen across all LPs is about the same, this effect can be widely avoided. But also other methods like controlling the amount of available memory or the optimism by artificially throttling LVT progression can achieve the same result for such "imbalanced" partitionings.

Hardware Performance The processor speed on the individual nodes executing the various LPs contributes to the overall Time Warp performance. But neither can high processor speed, nor large amounts of memory, nor low communication latency guarantee good performance. Time Warp can potentially perform better on "slow" processor than on fast ones if the communication-computation speed ratio of the paltform fits better to the event structure induced by the partitioning. High memory cycle time on the other hand can considerably accelerate the rollback procedure, but again this does not necessarily need to contribute to an overall acceleration, since an early annihilation message sendout could induce substantially more rollbacks than a more conservative policy.

Protocol Optimizations A variety of amendments and optimization to the standard Time Warp protocol have been proposed in the literature [10], mostly intending to reduce rollback overhead. But even for the *aggressive* and the *lazy cancellation* strategy (the former sends out antimessages immediately after receiveing a straggler, while the latter waits whether recomputation after rollback does not generate the same output messages again and thereby potentially gains form a local message annihilation) it was proven [24] that depending on the event structure one can arbitrarily outperform the other.

Implementation Optimizations Experiences with implementing Time Warp on shared [7, 4] or distributed memory machines [5] revealed that the protocol can gain considerably if nonstandard techniques are used for implementing memory management, event list manipulation, and interrupt based handling of messages [6]. "Tricky" implementations of fossil collection and GVT algorithms can have a significant performance influence.

The diverse set of factors contributing to the Time Warp performance makes an optimal implementation of the protocol for a particular platform problematic. Implementation design considerations that lead to good Time Warp performance for a class of simulation models that the user is interested in and for a machine environment that the user has access to require an early evaluation to avoid the development of a large, poorly performing Time Warp code. Is it possible, to follow a performance engineering methodology for irregular, unbalanced, and unstructured computations such as Time Warp? If so, one might be interested in whether

(i) Time Warp can accelerate the execution of particular simulation models, given an implementation on a dedicated platform. In our case we are interested in evaluating simulation models that evaluate performance behavior of data parallel computations and cluster based parallel systems.

(ii) Given the simulation model (e.g. of a HPF program), one might ot target a particular type of machine, where it is most promising to have Time Warp implemented as the *parallel* simulation engine.

Below, we show how N-MAP has been used to conduct a performance engineering study of Time Warp, avoiding a full, cumbersome implementation of the protocol for evaluation.

2.4 Time Warp Task Structure

The verbal description of the Time Warp protocol lets us directly provide an implementation prototype. Figure 4 shows the `tss`, defining a set of virtual processors `LP(i)` with indices assigned by the selector `where {i=0:MAXP-1;}`, that execute tasks declared separately (`taskdeclarations.h`). A typical task declaration consists of the keyword `task` followed by a list of toggles enclosed in curly braces, the task data type and the task identifier itself. Valid task toggles are `+t` to enable the tracing option, `+id="<id>"` to assign a task identifier to the task for future visualization, `+b` to force N-MAP to look for a behavior specification file (`tbs`), and `+r` to force N-MAP to look for a requirements specification file (`trs`) (these toggles are set by the graphical user interface of N-MAP). As an example, the task declaration that stands for the insertion of an event in the EVL, or a message into IQ, OQ, IB, or OB, respecting the timestamp appears as:

```
task{+t +b +r +id="0"} void chronological_insert();
```

For a better interpretation of performance data and readability of traces, N-MAP provides the possibility of grouping task into sets. For the Time Warp `tss` we have chosen the task grouping in Figure 5.

2.5 Adding Requirements

With the `tss` given in Figure 4, N-MAP would already be able to generate a simulated execution of the Time Warp protocol, using the default settings of unitary task timing. In order to generate more meaningful performance predictions we would have to add requirements reflecting the relative resource consumptions caused by task executions. For example, while the test `ts_less_than_LVT()` requires just a negligible amount CPU cycles, other tasks like `fossil_collection()` can be rather CPU time consuming.

Another observation about the requirement specifications is that some tasks would cause the same requirements irrespective of the state of the simulation (e.g. removing the first element from a list like in `remove_first()`), while others would entail state dependent requirements (e.g. like inserting an element in an ordered list like in `chronological_insert()`, which depends on the

```
/* Task Structure Specification */

 1  #include "datadeclarations.h"
 2  #include "taskdeclarations.h"

 3  process LP(i) where {i=0:MAXP-1;}
 4  {
 5  GVT=0.0; LVT=0.0; EVL=NULL; S=initialstate();
 6  while(ie=next_ie()) chronological_insert(ie, ts(ie), '+', EVL);

 7  while (GVT <= endtime) {
 8     while (m=next_IB())    {
 9        if (ts_less_than_LVT(m)){
10           if ((positive_and_not_dual_in_IQ(m)) ||
11              (negative_and_dual_in_IQ(m))) {
12              restore_earliest_state_before(ts(m));
13              generate_and_sendout_antimessages(ts(m));
14              LVT=earliest_state_timestamp_before(ts(m));
15              }
16           }
17        if (dual_in_IQ(m)) remove_dual_from_IQ(m);
18        else chronological_insert(m, ts(m), sign(m), IQ);
19        }
20     if (first_EVL_less_than_first_IQ()) e = remove_first(EVL);
21     else e = remove_first_nonnegative(IQ);
22     LVT = ts(e);
23     S = modified_by_occurrence_of(e);
24     while(ie=next_ie()) chronological_insert(ie, ts(ie), '+', EVL);
25     while(ie=next_preempted_ie()) remove_event(ie, EVL);
26     log_new_state(LVT, S, EVL, SS);
27     while(ee=next_ee()) {
28        deposit(ee, ts(ee), '+', OB);
29        chronological_insert(ee, ts(ee), '+', OQ);
30        }
31     send_out_contents(OB);
32     GVT = advance_GVT();
33     fossil_collection(GVT);
34     }
35  }
```

Figure 4: The Task Structure Specification for the Time Warp Protocol

Taskgroup	Tasks

SIMULATION (+id="0")	struct state *initialstate();
	void chronological_insert();
	int ts_less_than_LVT();
	int positive_and_not_dual_in_IQ();
	int negative_and_dual_in_IQ();
	int dual_in_IQ();
	void remove_dual_from_IQ();
	int first_EVL_less_than_first_IQ();
	struct event *remove_first();
	struct event *remove_first_nonnegative();
	struct state *modified_by_occurrence_of();
	struct event *next_ie();
	struct event *next_preempted_ie();
	void remove_event();
	void log_new_state();
	struct event *next_ee();
	void deposit();
	float advance_GVT();
	void fossil_collection();

COMMUNICATION (+id="1")	struct event *next_IB(); void send_out_contents();

ROLLBACK (+id="2")	void restore_earliest_state_before();
	void generate_and_sendout_antimessages();
	float earliest_state_timestamp_before();

Figure 5: Task Grouping

Task Behavior Specification	Task Requirement Specification

void chronological_insert
(event, time, sign, list)
struct Event *event; float time;
char sign; struct Event *list;
{ switch (list)
{ case EVL:
EVL_cnt++; break;
case IQ:
IQ_cnt++; break;
...
}
}
	void REQ_chronological_insert(event, time, sign, list)
	struct Event *event; float time;
	char sign; struct Event *list;
	
	{ switch (list)
	{ case EVL:
	return log2(EVL_cnt)*REQ_seek+REQ_insert; break;
	case IQ:
	return log2(IQ_cnt)*REQ_seek+REQ_insert; break;
	...
	}
	}

Figure 6: State dependent requirement specifications

current list length). A major feature of the N-MAP simulation engine is the possibility of handling *state dependent requirement specifications*, which makes it superior over model based performance predictions not providing this option. As an example, suppose the cost for inserting an element in an odered list is given by the seeking cost (logarithmic in the list length) and the insertion cost (constant). In this case the `trs` and `tbs` for `chronological_insert()` would be written as shown in Figure 6. Here the `tbs` of `chronological_insert()` is used to increment list element counters for the `EVL`, `IQ`, `OQ`, `IB` and `OB` (which are subject to insertion calls), such that the `trs` for `REQ_chronological_insert()` could then use the actual count values to return state dependent requirements. (Element counters would be decremented in the `tbs` of `remove_dual_from_IQ()`, `remove_first()`, etc.)

Another option is to provide *probabilistic* requirement specifications. The test whether an arriving message is a straggler (`ts_less_than_LVT()`) depends on the timestamp of the message as compared to the local value of LVT. In an early development phase where no explicit implementation of LVT progression is available, one might at least conduct a "what-if" analysis by providing an explicit rollback probability like:

```
int ts_less_than_LVT(); {return (drand48() < 0.25);}.
```

Here, the number of new events generated by `modified_by_occurrence_of()` could be described by a random variate drawn from an arbitrary theoretical or empirical distribution.

3 Scenarios and Performance Sensitivity Analysis

N-MAP provides the concept of *mutable* (workload) characterization parameters ("mutables", for short) as a means for scenario control. For the N-MAP parser and code generator, a mutable appears as a symbolic name, and therefore does not necessarily need to represent a program constant, but can be a parameterized function call as well. To ease the use of mutables, their values (or symbolic instancies) are varied by the scenario manager automatically, once the user provides certain parameter ranges and stepping frequencies. A specific setting of mutables is called a *case*, a set of cases (automatically generated or composed by hand) is called a *scenario* (Figure 2). Provided with a scenario, N-MAP iterates over all the contained cases and produces performance predictions for all those cases in a systematic way. Figure 7 shows how the user can compose a case by selecting mutables contained in the `tbs` of tasks and setting values for the selected mutables. A case can then be embedded into a scenario. Specifically, in Figure 7 the user for a Time Warp scenario controls the number of internal events generated in one iteration step (within the task `S = modified_by_occurrence_of(e)`) by the mutual `Internal_Event_Count_Distribution`, the number of output messages (external events) generated in one iteration by the mutual `External_Event_Count_Distribution` and LVT increment imposed by an event scheduled in the local EVL by the mutual `LVT_Progression`.

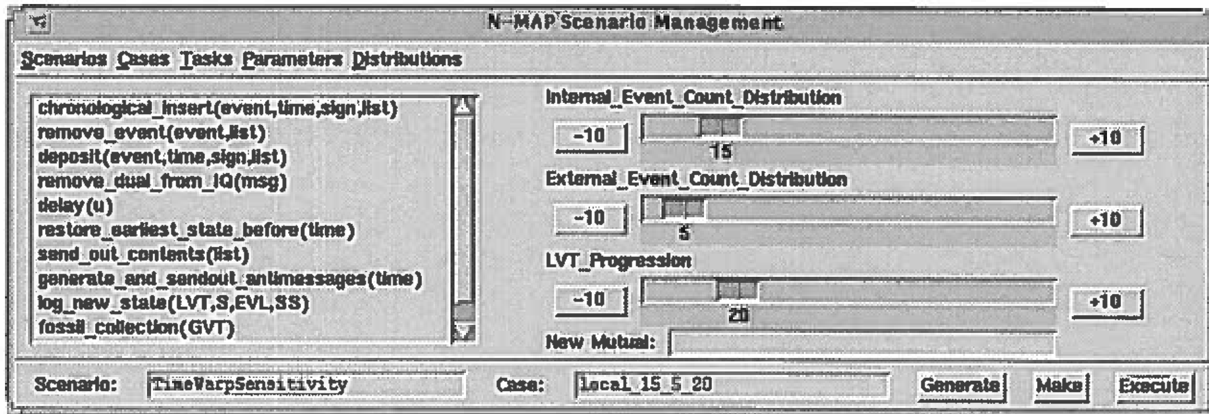

Figure 7: Composing a case in the Scenario Manager

3.1 Characterizations for Time Warp

For a performance sensitivity analysis of Time Warp, in addition to the "algorithmic idea" as developed in the `tss` above, we have to provide a characterization of the *program input* which in our case is the simulation model executed by the Time Warp protocol, as well as a characterization of *execution environment* (see Figure 2). From prior work [12], we used the characterization of two execution environments: the CM-5 being programmed using CMMD message passing calls, and a cluster of RS6000 workstations with Token Ring interconnect and PVM 3.2. The respective system requirements files for those two environments are readily available.

To support a program input characterization at the level of events to be simulated by Time Warp, tasks in the `tss` (Figure 4) that cause modifications of LVT and GVT have been assigned `tbs`'s that actually do handle timestamp manipulations based on values of the mutable `LVT_Progression`. For example, if the occurrence of the event `e` as simulated by statement 23 in Figure 4 causes the scheduling of a new event, then the `tbs` of `modified_by_occurrence_of()` inserts a newly created event in EVL with the increment `ev->lvt = LVT+LVT_progression;`. The following mutables were defined for a characterization of the simulation (Figure 8):

- `Model_Parallelism`
  to control for every `LP(i)` the amount of events initially scheduled in the local EVL (encoded in the `tbs` of task `initialstate()`). In an abstract sense, `Model_Parallelism` represent the number of objects initially assigned to one LP, but subject to migration to other LPs.

- `Event_Grain_Size`
  to control the amount of CPU consumption for the execution of a single event (object) (encoded in the `tbs` of task `modified_by_occurrence_of()`).

- `LVT_Progression`
  to control the amount of LVT increment imposed by an event scheduled in the local EVL (encoded in the `tbs` of task `modified_by_occurrence_of()`).

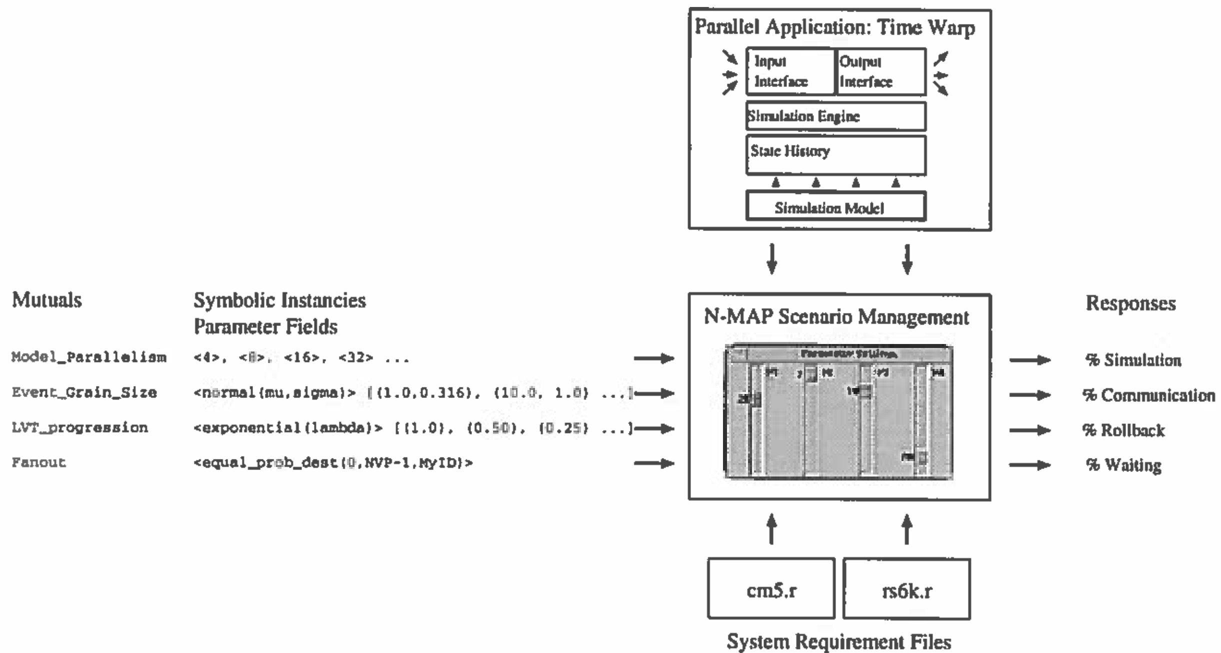

Figure 8: Early Behavior Prediction: Mutuals and Responses

- `Fanout`
  to control the selection of a target LP to which an output message is generated for (object migration) (encoded in the `tbs` of task `modified_by_occurrence_of()`).

For various variations and combinations on the mutuals as performance (output) parameters (*responses*) to be generated by N-MAP, we have chosen the percentages of CPU time expected to be used for executing tasks as grouped in Figure 5. Any other performance characteristic like absolute number of messages generated, speedup, processor utilization, etc. could have been used, but our main goal was the *sensitivity* of Time Warp to certain simulation model characteristics and target platform performance.

3.2 Early Behavior Predictions

Using the mutuals defined above, N-MAP can generate simulated execution behaviors from the code skeleton in Figure 4. As an example, for the *case*:

```
Model_Parallelism = 4;
Event_Grain_Size  = (normal(10.0,1.0));
LVT_Progression   = (exponential(1.0/8.0));
Fanout            = (equal_prob_dest(0,NVP-1,coMyID));
```

i.e. 4 objects scheduled initially scheduled in every LP, a normally distributed event grain size of $N(10.0, 1.0)$ μsecs, an exponentially distributed LVT increment of $exp(\frac{1}{8})$ per executed event and

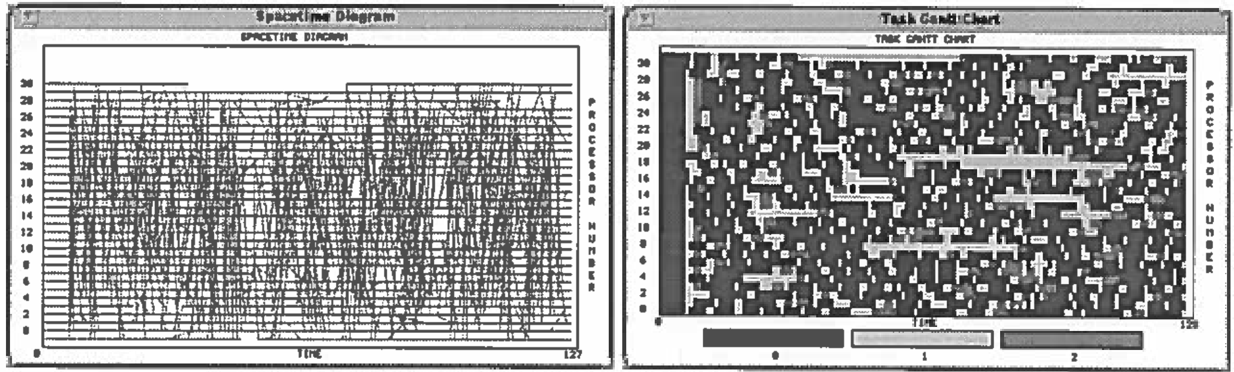

Figure 9: Predicted Time Warp Execution Behavior for CM-5

an equally distributed destination selection probability for migrating objects, N-MAP predicted the execution behavior for a 32 processor CM-5 as depicted in Figure 9 (0 simualtion, 1 communication, 2 waiting for messages). The displays clearly indicate the lack of any structure in the communication pattern (left), as well as the irregularities in computations and the imbalance occasionally induced due to waiting for messages (right). Point-predictions (evaluations for a single *case*) like this can help when reasoning about Time Warp performance on a *particular* simulation model and a *particular* target platform. In general, however, one would be interested in how sensitive the parallel application is to the variation of one or more of the mutuals before a full implementation of the application is undertaken. Setting up N-MAP scenarios would be the method of choice in this case (Figure 8).

3.2.1 Scenario 1: Model Parallelism and LVT Imbalance

In a first performance scenario, we are interested in the impact of different LVT progressions in the various LPs. Specifically we are interested in the *rollback thrashing* effect described in Section 2.3, and its relation to the model parallelism on a target system with relatively high communication performance (CM-5). To make effects more transparent, we only consider the response of two LPs. Suppose also that due to the simulation model one LP increments LVT in larger steps than the other (think of two interacting objects, one with high, the other with low response time). The scenario set-up is as follows:

Scenario 1: Model Parallelism and LVT Imbalance		
Mutual	Symbolic Instance	Parameters
`Model_Parallelism`	`MP`	`MP` = 4, 8, 12, 16, 20, 24
`Event_Grain_Size`	`normal(mu,sigma)`	$(\mu, \sigma) = (10.0, 2.0)$
`LVT_progression_LP0`	`exponential(lambda)`	$\lambda = 1$
`LVT_progression_LP1`	`exponential(lambda)`	$\lambda = 1, \frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}$
`Fanout`	`equal_prob_dest(0,NVP-1,MyID)`	`NVP` = 2

Figure 10: Settings for Scenario 1

The responses for Scenario 1 are depicted in Figure 12. We observe that at an `Event_Grain_Size` of 10 μsec the LPs cannot utilize more than about 75 % of the overall execution time for doing 'real' simulation work (i.e., simulation work that would also have to be executed on a sequential simulator). With the same argument, given that a simulation model employs events that have an average CPU requirement of about 10 μsec, a Time Warp simulation of that model running on only 2 processors would already accelerate the sequential execution by a factor of 1.5. The more the overal simulation system becomes imbalanced (e.g., LP1 progressing LVT1 16 times ($1/\lambda$) as fast than LP0), the more the data structures become loaded in LP1, causing a shift of workload from LP0 to LP1 (a similar phenomenon has been observed in [11] on the iPSC/860). On the other hand, LP0 becomes increasingly depleted, and starts wasting CPU-cycles due to message waiting. This is clearly seen in the "% Waiting" response for LP0, but also the one for LP1 vanishes in cases of slight imbalance. Counterintuitively, model parallelism (at least for this event grain size) does not have that much influence. However, it is interesting that with increasing load in LP1 the percentage of CPU time for executing rollbacks at high model parallelism becomes smaller than the one for low model parallelism. This is due to the fact that at a higher object (message) population in the system, communication load starts dominating rollback load at some point of imbalance (here in between $\lambda = \frac{1}{2}$ and $\lambda = \frac{1}{4}$).

3.2.2 Senario 2: Event Grain Size, CM-5 vs. RS6000 Cluster

The second scenario is devoted to address the questions raised in Section 2.3 on issues of simulation model and execution platforms combinations. For a simulation model with stationary LVT increments, we investigate the impact of average CPU time consumed per executed event, related to an increasing number of processors and model parallelism. Predictions for a CM-5 are related to the ones for an RS6000 workstation cluster:

Scenario 2: Event Grain Size, CM-5 vs. RS6000 Cluster		
Mutual	Symbolic Instance	Parameters
`Model_Parallelism`	`MP`	`MP` $= 4$ (each LP)
`Event_Grain_Size`	`normal(mu,sigma)`	$(\mu, \sigma) = (1.0, \sqrt(0.1)), (10.0, \sqrt(1.0)),$ $(100.0, \sqrt(10.0)), \ldots, (10000.0, \sqrt(1000.0))$
`LVT_progression_LP`	`exponential(lambda)`	$\lambda = \frac{1}{8}$
`Fanout`	`equal_prob_dest(0,NVP-1,MyID)`	`NVP` $= 2, 4, 8, 16, 32, 64, 128$

Figure 11: Settings for Scenario 2

Looking at the responses for Scenario 2 (Figure 13) tells that the RS6000 cluster is not a promising target platform for Time Warp in conjuction with simulation models that have small grain size. Simulation models with event executions of at least 1000 μsec of CPU demand are required in order to obtain speedup over a sequential simulation. Clearly, this is due to the comparably high communication latencies on the Token Ring. A CM-5 implementation of Time Warp, however, is promising also for simulation models with far smaller event grain size, and it appears that this platform reacts more sensitive to low grain workloads if the number of processors involved is increased. Indeed we find the best relative performance if just a few processors are used (e.g.

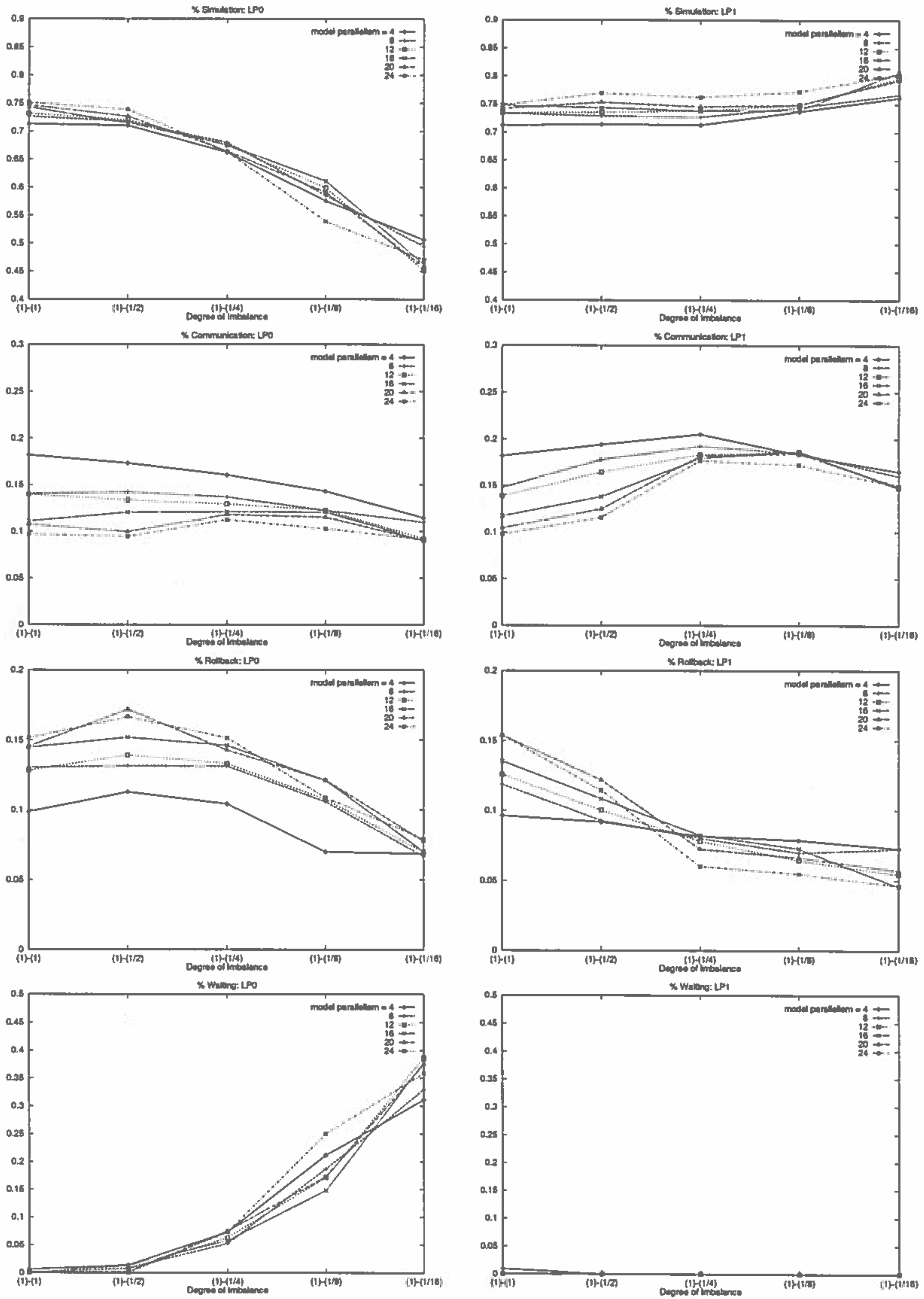

Figure 12: Responses for Scenario 1: Model Parallelism, LVT imbalance, CM-5

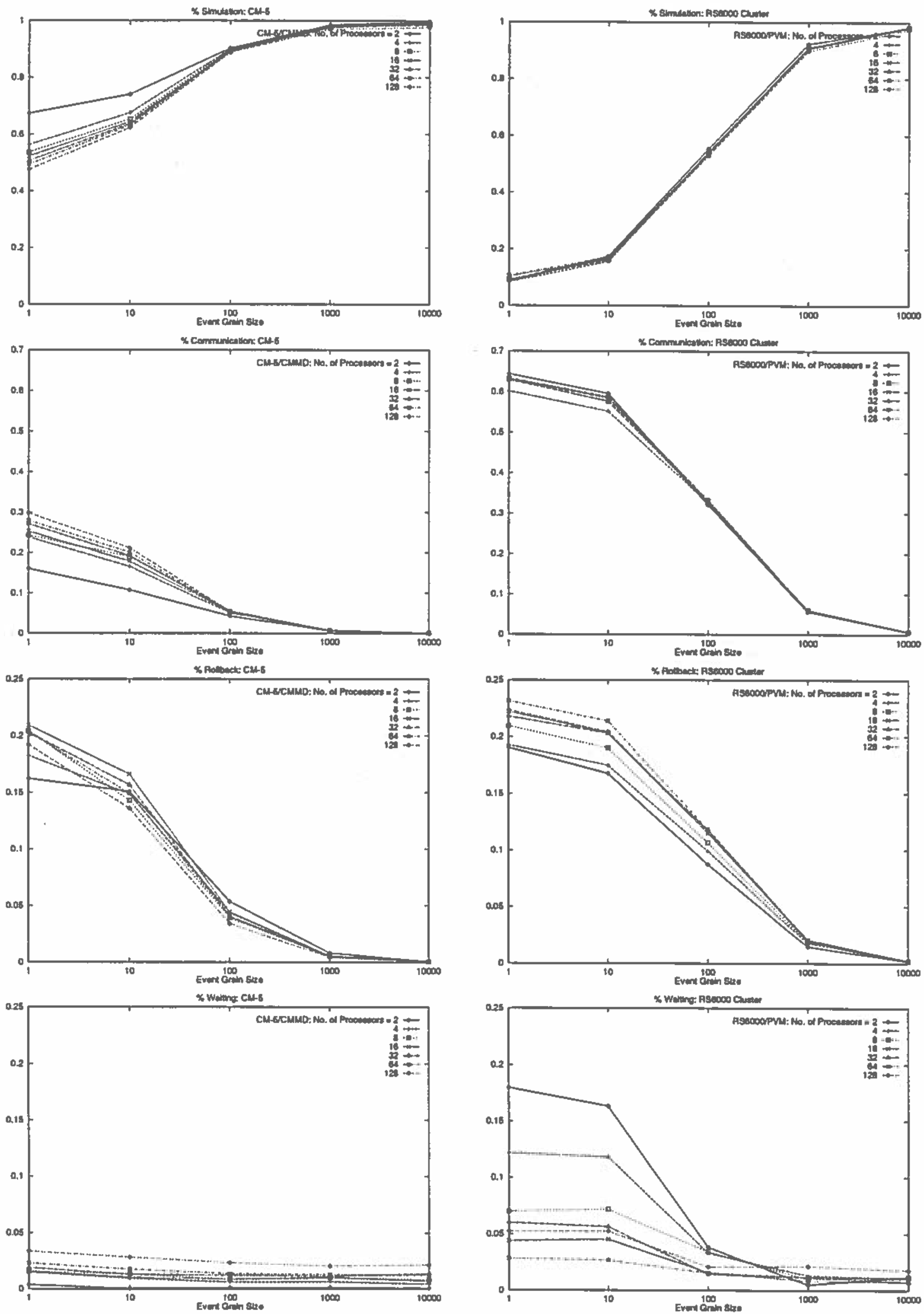

Figure 13: Responses for Scenario 2: Event Grain Size, CM-5 vs. RS6000 Cluster

NVP=2), since contention on the CM-5's data network can be kept small, but this effect vanishes the higher the event grain size becomes: less frequent messages reduce contention (and by that communication overhead) in a natural way. On the RS6000 cluster, for small event grain sizes, blocking of LPs frequently occurs since messages are longer in transit than the simulation of the corresponding events takes in terms of CPU time ("% Waiting").

4 Conclusions

Traditionally, performance engineering approaches for parallel software development have focussed on applications that behave in predictable ways. That is, the applications have enough structure in problem decomposition, regularity in execution behavior and balance in computational load that assumptions about deterministic operational characteristics and static execution environments do not disqualify the performance evaluation obtained from parallel prediction models. For irregular, unstructured and unbalanced problems, full code implementation is often the only available alternative to determine the performance consequences of algorithm design choices.

Here, we have demonstrated a performance engineering methodology for this class of problems that merges a performance-oriented parallel software design system, N-MAP, with a systematic approach to scenario management and sensitivity analysis. As a tool, N-MAP accepts program descriptions at the task structure level (task structure specifications) as input. Parametrized with quantitative workload parameters and target specific performance characteristics, the specifications are parsed and translated into a thread based virtual processor parallel program, the execution of which on-the-fly generates execution statistics, or trace from which all relevant program performance characteristics can be deduced in a subsequent analysis. After a selection of mutable (input) parameters, N-MAP automatically manages the construction of scenarios and visually reports back performance sensitivities on the response parameters of interest.

This methodology, captured in the N-MAP system, was tested on the Time Warp parallel and distributed discrete event simulation protocol, which has resisted a general performance characterization in the past because of its strong interdependence between multiple performance factors. As Time Warp stresses the ability of performance prediction for irregular, unstructured and unbalanced computation, the results produced are quite encouraging. The task structure specification of the Time Warp protocol is significantly less than what would be required of a fully operational implementation. Nevertheless, the N-MAP predictions for the different scenarios described were able to reveal certain performance sensitivities that would be important for parallel simulation application developers to know, as they relate to issues of appropriate simulation model partitioning (Scenario 1), preferred execution architecture, and potential gains of a parallel simulation for certain simulation models (Scenario 2). The prediction quality of the N-MAP scenarios shown in this work finds a solid empirical validation in our full Time Warp implementations [6].

References

[1] G. Agrawal, A. Sussman, and J. Saltz. Efficient Runtime Support for Parallelizing Block

Structured Applications. In *Proc. of the Scalable High Performance Computing Conference*, pages 158 – 167. IEEE CS Press, 1994.

[2] P. Brinch Hansen. Householder Reduction of Linear Equations. *ACM Computing Surveys*, 24(2):185 – 194, June 1992.

[3] M. Calzarossa and G. Serazzi. Workload Characterization: A Survey. In *Proceedings of the IEEE*, 1993.

[4] Ch. D. Carothers, R. M. Fujimoto, and P. England. Effect of Communication Overheads on Time Warp Performance: An Experimental Study. In D. K. Arvind, Rajive Bagrodia, and Jason Yi-Bing Lin, editors, *Proceedings of the 8th Workshop on Parallel and Distributed Simulation (PADS '94)*, pages 118–125, July 1994.

[5] G. Chiola and A. Ferscha. Distributed Simulation of Petri Nets. *IEEE Parallel and Distributed Technology*, 1(3):33 – 50, August 1993.

[6] G. Chiola and A. Ferscha. Performance Comparable Design of Efficient Synchronization Protocols for Distributed Simulation. In *Proc. of MASCOTS'95*, pages 343 – 348. IEEE Computer Society Press, 1995.

[7] S. Das, R. Fujimoto, K. Panesar, D. Allison, and M. Hybinette. GTW: A Time Warp System for Shared Memory Multiprocessors. In J. D. Tew and S. Manivannan, editors, *Proceedings of the 1994 Winter Simulation Conference*, 1994. *to appear.*

[8] T. Fahringer and H.P. Zima. A Static Parameter based Performance Prediction Tool for Parallel Program. In *Proc. 1993 ACM Int. Conf. on Supercomputing, July 1993, Tokyo, Japan*, 1993.

[9] A. Ferscha. A Petri Net Approach for Performance Oriented Parallel Program Design. *Journal of Parallel and Distributed Computing*, 15(3):188 – 206, July 1992.

[10] A. Ferscha. Parallel and Distributed Simulation of Discrete Event Systems. In A. Y. Zomaya, editor, *Handbook of Parallel and Distributed Computing*. McGraw-Hill, 1995.

[11] A. Ferscha and G. Chiola. Accelerating the Evaluation of Parallel Program Performance Models using Distributed Simulation. In *Proc. of. the 7th Int. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation.*, Lecture Notes in Computer Science, pages 231–252. Springer Verlag, 1994.

[12] A. Ferscha and J. Johnson. Performance Oriented Development of SPMD Programs Based on Task Structure Specifications. In B. Buchberger and J. Volkert, editors, *Parallel Processing: CONPAR94–VAPP VI*, LNCS 854, pages 51–65. Springer Verlag, 1994.

[13] A. Ferscha and J. Johnson. N-MAP: A Virtual Processor Discrete Event Simulation Tool for Performance Predicition in CAPSE. In *Proceedings of the HICCS28*. IEEE Computer Society Press, 1995. *to appear.*

[14] R. M. Fujimoto. Performance of Time Warp under Sythetic Workloads. In D. Nicol, editor, *Proc. of the SCS Multiconf. on Distributed Simulation*, pages 23 – 28, 1990.

[15] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley. A users' guide to PICL: a portable instrumented communication library. Technical Report ORNL/TM-11616, Oak Ridge National Laboratory, August 1990.

[16] M. T. Heath and J. A. Etheridge. Visualizing Performance of Parallel Programs. Technical Report ORNL/TM-11813, Oak Ridge National Laboratory, May 1991.

[17] Y-S. Hwang, B. Moon, Sh. Sharma, R. Das, and J. Saltz. Runtime Support to Parallelize Adaptive Irregular Programs. In L. L. Dongarra and B. Tourancheau, editors, *Proc. of the 2nd Workshop on Environments and Tools for Parallel Scientific Computing*, pages 19 – 32. SIAM, 1994.

[18] D. A. Jefferson. Virtual Time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[19] I. O. Mahgoub and A. K. Elmagarmid. Performance Analysis of a Generalized Class of m-Level HIerarchical Multiprocessor Systems. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):129 – 138, March 1992.

[20] A. D. Malony. *Performance Observability.* PhD thesis, University of Illinois, Department of Computer Science, University of Illinois, 1304 W. Springfield Avenue, Urbana, IL 61801, October 1990.

[21] A. D. Malony, D. A. Reed, and H. A. G. Wijshoff. Performance Measurement Intrusion and Perturbation Analysis. *IEEE Transactions on Parallel and Distributed Systems*, 3(4):433 – 450, July 1992.

[22] D. C. Marinescu, J. E. Lumpp, T. L. Casavant, and H. J. Siegel. Models for Monitoring and Debugging Tools for Parallel and Distributed Software. *Journal of Parallel and Distributed Computing*, 9:171–184, 1990.

[23] B. P. Miller, M. Clark, J. Hollingsworth, S. Kierstead, S.-S. Lim, and T. Torzewski. IPS-2: The Second Generation of a Parallel Program Measurement System. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):206–217, April 1990.

[24] P. L. Reiher, R. M. Fujimoto, S. Bellenot, and D. Jefferson. Cancellation Strategies in Optimistic Execution Systems. In *Proceedings of the SCS Multiconference on Distributed Simulation Vol. 22 (1)*, pages 112–121. SCS, January 1990.

[25] J. P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared Memory. Technical report, Computer Systems Laboratory, Stanford University, CA 94305, 1993.