



# N-MAP — an environment for the performance oriented development process of efficient distributed programs

Alois Ferscha\*, James Johnson

*Universität Wien, Institut für Angewandte Informatik und Informationssysteme, Lenaugasse 2/8, A-1080 Wien, Austria*

---

## Abstract

The performance engineering activities in a *performance oriented* development process of distributed (and parallel) software have to cover a range of crucial issues: performance prediction in early development stages, (analytical or simulation) modeling in the detailed specification and coding phase, monitoring/measurements in the testing and correction phase, and — most importantly for distributed applications executing in heterogeneous distributed environments — automated performance management at run-time.

In this paper we present a development environment, N-MAP (N-(virtual) processor map), to tackle these issues. Within N-MAP, the challenging aspect of performance prediction to support a performance oriented, incremental development of distributed programs is addressed, such that design choices can be investigated far ahead of the full coding of the application. Our approach considers the *algorithmic idea* as the first step towards an application, for which the programmer should not be forced to provide detailed program code, but just to focus on the constituent and performance critical program parts.

Due to time-varying workloads and changing system resource availability in heterogeneous multicomputer environments in which the distributed application is embedded, self-managing applications are demanded which allow for dynamic reconfiguration in response to a new system states. N-MAP aims at the development of applications able to induce and assess the impact of such re-configurations at run-time in a “pro-active” way. As opposed to “re-active” systems which bring reconfigurations in effect after the system state has changed, the timeliness of reconfiguration activities is attempted by a future state forecast mechanism and performance management decisions are based on anticipated future system states. © 2000 Published by Elsevier Science B.V. All rights reserved.

*Keywords:* N-(virtual) processor map; Performance oriented development process; Distributed programs

---

## 1. Introduction

Since the use of distributed systems and massively parallel processing hardware in science and engineering is primarily motivated to increase computational performance, it is obvious that a performance

orientation must be the driving force in the development process of parallel programs. Only very few efforts have been devoted to the *rigorous* integration of performance engineering activities in the phases of distributed and parallel application software; computerized tools suitable for supporting performance efficient programming are rarely in existence today.

The N-MAP (N-(virtual) processor map) environment aims at remedying this situation by providing

---

\* Corresponding author.

*E-mail addresses:* ferscha@ani.univie.ac.at (A. Ferscha), johnson@ani.univie.ac.at (J. Johnson)

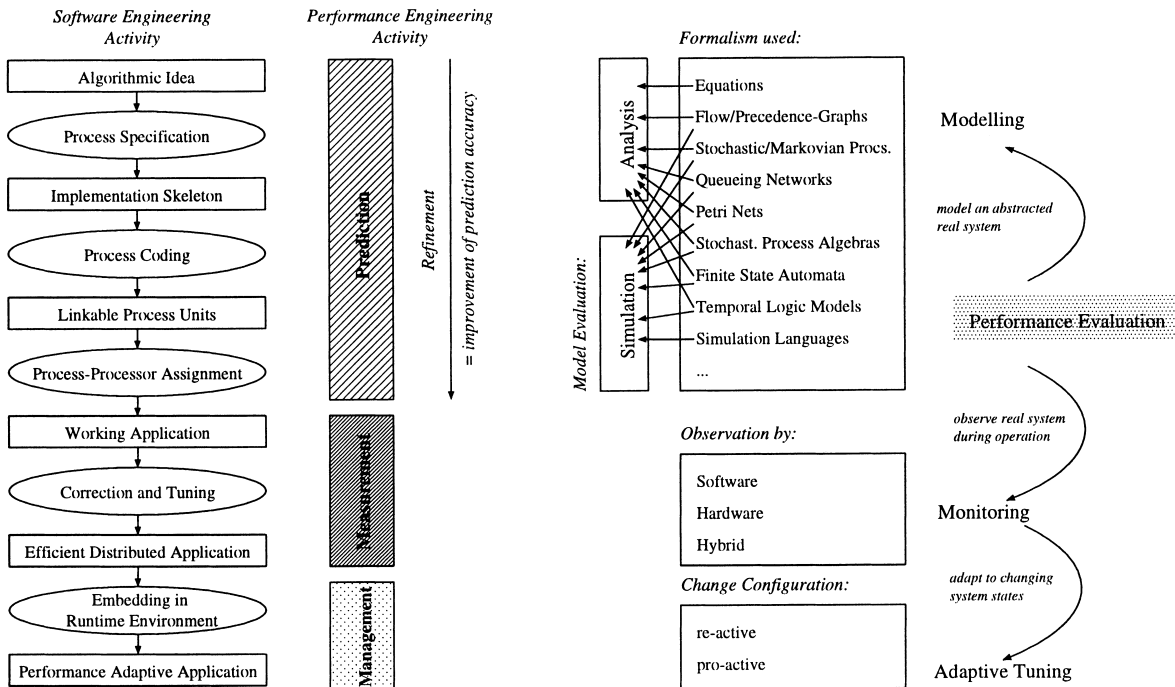


Fig. 1. Performance engineering activities.

performance prediction methods early in the development process based on *skeletal program code* which, during the course of program development, is iteratively and incrementally refined under constant performance supervision until a fully functional, performance efficient program is created.

N-MAP provides support for programming and performance engineering activities in all phases of a *performance oriented* distributed program development process ranging from *performance prediction* in the early stages (design), *modeling* and *simulation* in the detailed specification and coding phase, *monitoring* and *measurements* in the testing and correction phase, to finally the automated performance management at run-time (see Fig. 1).

As a *performance oriented* 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.

We have developed N-MAP and demonstrated its successful use in a variety of different contexts. The development of an application for  $Ax = b$  by means of a series of Householder transformations  $(HA)x = Hb$  on the CM-5 is reported in [1]. The process of performance prototyping was exercised in [2]. The challenging performance assessment of distributed simulation protocols was attempted in [3] and in [4], where a  $2^k r$  full factorial design simulation experiment was conducted as a means for systematically searching huge performance spaces of distributed applications. Recently we have demonstrated N-MAP’s abilities with respect to adaptive performance tuning [5].

In this paper we present the architecture and the key components of N-MAP, which has become a stable and mature environment, publicly available to the performance and software engineering community. In Section 2 we present N-MAP’s software architecture. Section 3 is devoted to scenario management and the simulation engine. Section 4 explains adaptive performance management.

## 2. The N-MAP environment

Features that make N-MAP diverse from existing tools are situated in the area of performance predictions. Since any kind of predictive analysis is based on models (see Fig. 1), the primary drawback of model based performance prediction is that the quality of predictions is entirely dependent on the way how system characteristics are abstracted into models, and the ability of tools to (accurately enough) evaluate these models. In the classical approach of model based performance prediction, models are built explicitly with the motivation to be able to derive performance indices for a range of different parameter settings, i.e. workload or system parameters. Explicit modeling is not new as a performance engineering method for distributed systems [6,7], and could be applied successfully in areas where model assumptions are sufficiently close to system characteristics. Success of the approach is almost guaranteed when problems of *static* nature are investigated in *homogeneous* environments with *deterministic* operational characteristics [8]. *Explicit modeling* appears convincing for multiprocessor platforms with deterministic operational characteristics, for programs with static structure and balanced computational loads as well as for regularly structured problems and dynamically non-changing environments — at least for toy examples. For more general cases, however, the explicit modeling approach is suspect to fail. Reasons for these shortcomings, jokingly referred to as the “art of modeling”, are inadequate model assumptions (exponential timing, stochastic independence, finite state space, memoryless property, etc.), “sloppy” modeling (to preserve tractability, reduce state-space, keep convenient/efficient solution algorithms/tools applicable, etc.), an irregular problem structure, unstructured execution pattern (e.g. state/data dependent branching), non-deterministic behavior of software layers and the run-time system, unforeseen execution dynamics (like contention, congestion, thrashing, etc.) and an overwhelming degree of performance interdependencies, etc.

### 2.1. Performance prototyping

N-MAP, as a prototyping environment, although reliant on models in predictive performance analysis, avoids engaging in explicit models. Early code

skeletons, together with resource requirement specifications, serve as (performance) prototypes for distributed programs, and, simultaneously, as their *implicit* performance models. Implicit, because a simulated execution of the prototype (real code) yields a prediction of performance indices, and no explicit (static) model representation is involved. As such, N-MAP is not liable to modeling lapses.

More specifically, a performance prototype notated in the N-MAP language, serves two purposes. First it represents the implementation skeleton subject to stepwise refinement. Secondly it can be parsed/translated into a discrete event simulation program, the execution of which mimics the execution of the prototype on an idealized multicomputer, or, provided information on execution characteristics of a real multicomputer, on that one. The N-MAP language is based on C, and includes language constructs for intuitive components of distributed/parallel programming like *tasks*, *processes* and (communication) *packets*. The concept of *virtual processors* allows for a development process that is preliminarily independent of a concrete set of real processors, easing a *problem oriented* development strategy. 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. The simulation engine in N-MAP progresses individual simulated (CPU) times for a set of virtual processors, according to (user-)defined requirement specifications. Since requirements are subject to repeated refinements too, prediction accuracy increases steadily.

A distributed application is developed within in the N-MAP environment (Fig. 2) by first expressing the “algorithmic idea” in the form of a task structure specification (TSS) using the constructs of the N-MAP language. The program input may then be characterized *quantitatively* by providing task and packet requirement specifications (TRS and PRS) which give the estimated execution time requirements for task and the size of the data to be transferred in each packet. Tasks and packets may furthermore be characterized *qualitatively* by providing task and packet behavior specifications which describe the functionality (i.e. code

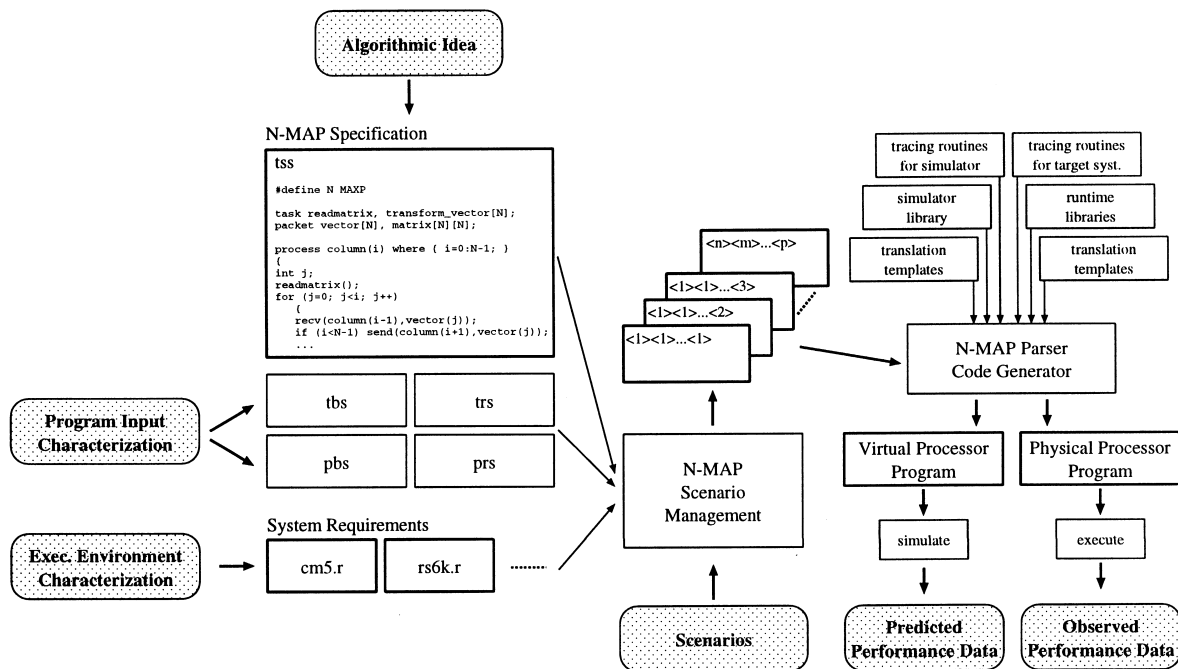


Fig. 2. Architecture of the N-MAP environment.

to be executed) of the tasks and the actual data to be transferred in packets. This, together with a system requirements specification (SRS) describing the run-time behavior of the target system in the form of a *model of execution*, is parsed and translated into a discrete event simulation program.

The execution of the simulation program generates the behavior of the distributed program as if it were executed on a set of  $N$  virtual processors, and performance data is collected to analyze this behavior. The statistics compiled include busy, overhead and idle times 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) giving access to standard analysis visualization tools (e.g. ParaGraph) which can further characterize the simulated behavior.

Another remarkable feature unique in N-MAP is the way in which task attributes like resource requirements, tracing options etc. are specified. From an automatically assembled “task list”, generated after the lexical analysis of the prototype source code, the users in a direct manipulative way collects tasks of

this choice to be presented via the *task attribute panel* for further manipulation. In the example of Fig. 3 the possibility of stochastic requirement specification is illustrated, which are particularly desirable for task structures with non-deterministic behavior. For `taskA`, at every simulated invocation of the respective task, virtual CPU time is progressed by the value of a truncated normally distributed random variate, for `taskB` by the value of an exponential variate. `taskC` is parameterized with a deterministic virtual CPU time progression of  $100\ \mu\text{s}$ . The `trace` toggle allows for switching the tracing option for every task invocation in the simulated execution on or off. If the TBS toggle is switched on, the program code associated with the task and the respective actual CPU time of its execution will increment virtual time. This feature allows for a smooth transition from performance prediction to performance measurement as soon as a working application emerges from the prototype.

Consider as an example the development of the Householder transformations  $(HA)x = Hb$  for the CM-5 ([1]) as illustrated in Fig. 4. In the first step, using a straightforward task structure specification

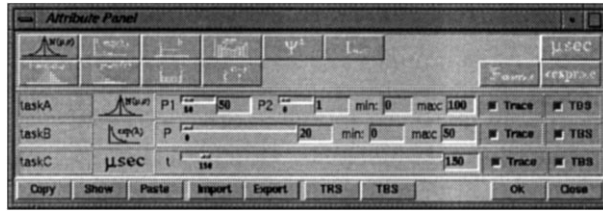


Fig. 3. The task attribute panel.

and default settings, N-MAP generates a predicted behavior as in the space time chart on the right hand side. Providing additional requirement specifications in step 2 (e.g. defining the CPU requirements for the readmatrix() task to be a normally distributed random variate, or the transform(i, j) being dependent on the iteration index *j* and the virtual processor number *i*) yields a more adequate prediction (task instances at higher iteration indices take less CPU resources). After providing full source code

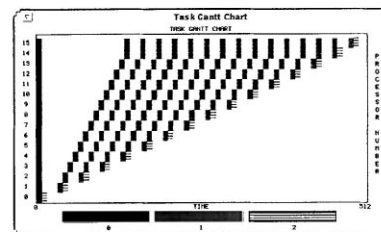
for the application and considerable debugging efforts, N-MAP collects the behavior in step 3 from a true execution on the target platform (CM-5). It is convincingly seen, that a severe misdesign of the application was already observed in step 1 (after very few development efforts), and further steps towards an inefficient distributed application could have been avoided.

To support the experimental way in which performance efficient applications often need to be built,

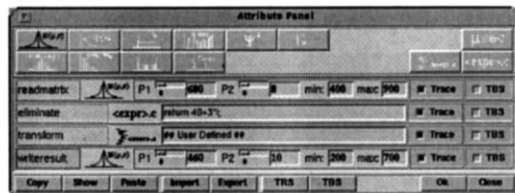
1) Notating the Algorithmic Idea in the N-MAP Language

```
process column(i) where (i=0:MAXP-1;)
{
  int j;
  readmatrix(i);
  for (j=0; j<i; j++)
  {
    recv(column(i-1), v_pack(j));
    if (i<MAXP-1) send(column(i+1), v_pack(j));
    transform(i, j);
  }
  eliminate(i);
  if (i<MAXP-1) send(column(i+1), v_pack(i));
  writeresult(i);
}
```

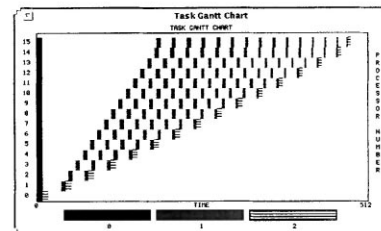
N-MAP Simulation



2) Adding Resource and System Requirements



N-MAP Simulation



3) Add Source Code

```
void eliminate(i)
{
  int i;
  float anorm, dii, fi, wi; int k;
  anorm=sqrt(product(i, a, a));
  if (a[i] > 0.0) dii=-anorm;
  else dii=anorm;
  wi=a[i]-dii;
  fi=sqrt(-2.0*wi*dii);
  v[i]=wi/fi;
  a[i]=dii;
  for (k=i+1; k<MAXP; k++)
  { v[k]=a[k]/fi; a[k]=0.0; }
  return;
}
```

Generate Target

Execution on CM-5

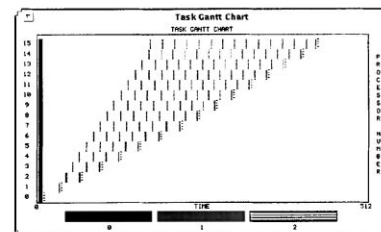


Fig. 4. From the task structure specification to target code.

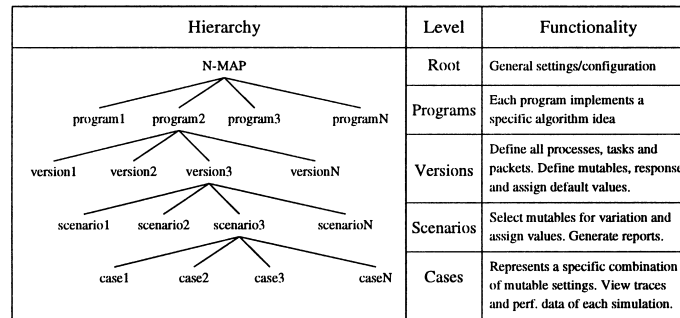


Fig. 5. The N-MAP object hierarchy.

N-MAP implements performance engineering activities along a hierarchy of objects subject for analysis (see Fig. 5). At the top level, N-MAP allows the user to organize different variants of programs that stand for different implementation strategies of a given algorithm (algorithmic idea). During the prototyping phases, each program undergoes several versions, each of which can be analyzed as a single instance (case), or as a systematic set of performance experiments (scenarios). Attribute inheritance and code cloning is possible through all levels of the hierarchy, and considerably eases experimental prototyping. Intercomparability of performance indices generated at any level of the hierarchy is preserved, thus yielding a self contained experimentation environment.

N-MAP’s graphical user interface presents the top level tool functionality as in Fig. 6.

### 2.2. Scenario management

A distinguished feature of N-MAP is the possibility for an automated scenario creation from a user defined set of parameters or constraints of particular interest that allows for a systematic search in the performance space. Opposed to repeated performance prediction invocations by the user, N-MAP automatically generates all meaningful performance point estimates by means of the scenario editor (Fig. 7). The user expresses parameters of interest in terms of so called *mutables*, i.e. symbols in the prototype subject to variation (e.g.

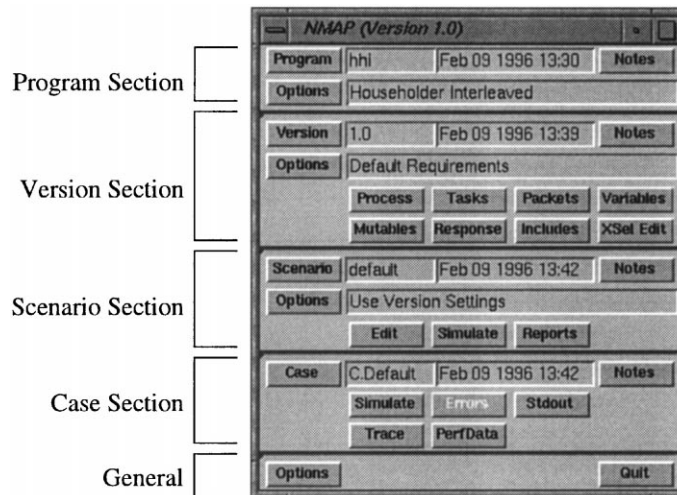


Fig. 6. The N-MAP main window.

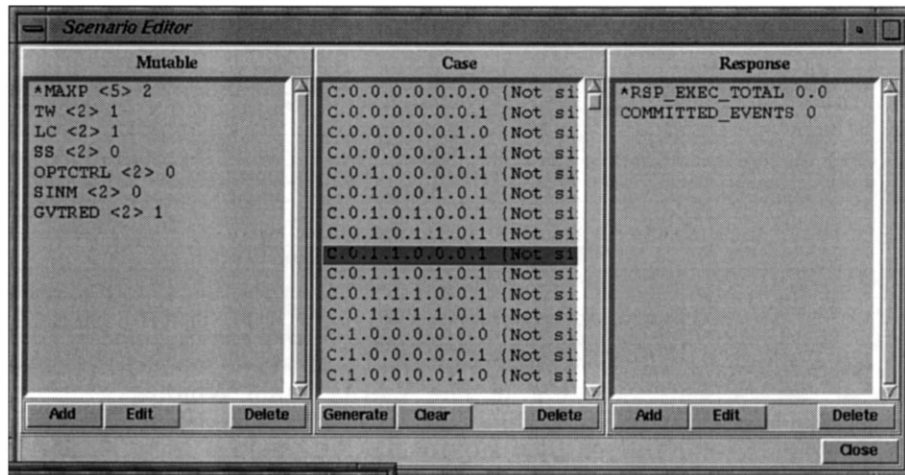
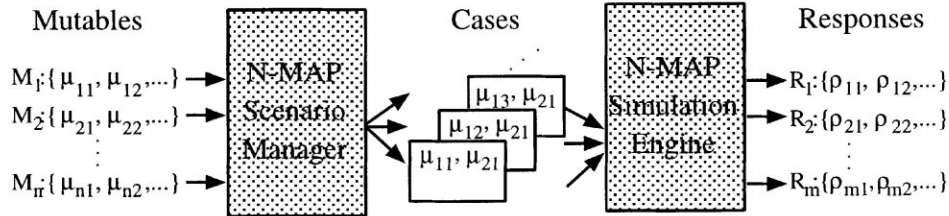
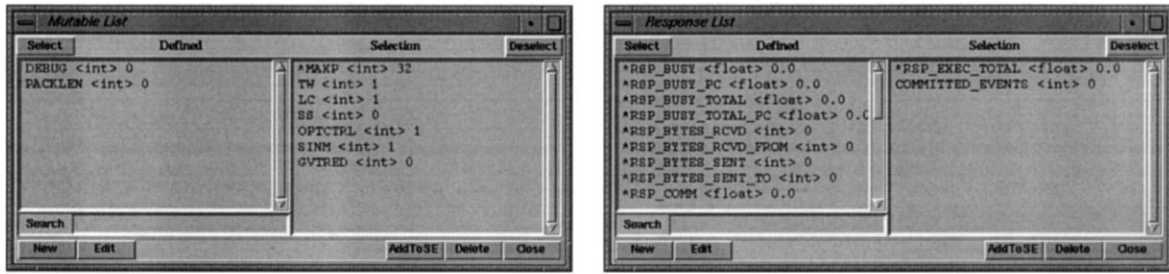


Fig. 7. N-MAP scenario editor.

variables, program constants, tasks, file names) and defines *response* variables which collect the desired results (e.g. performance metrics such as aggregated task invocation times, communication overheads etc.).

From the list of defined mutables and responses, a subset is selected for investigation and inserted into the scenario editor. The desired range of variation and stepping frequency for each mutable is then specified by user. N-MAP can now automatically generate a set of “cases” by taking the Cartesian product over all mutable settings. Each case is then executed/simulated and results are recorded in the corresponding response

variables. The results generated with the scenario manager undergo statistical analysis and are returned in aggregated performance reports. A variety of form customizations are available to meet individual users needs. Multiple resources in a network of workstations can be allocated to accelerate the scenario experiment via parallel execution.

A graphical user interface provides the user with methods for conveniently accessing all features of the N-MAP tool. The hierarchy of programs, versions of programs, scenarios and individual cases is reflected in the main window of the tool allowing for easy

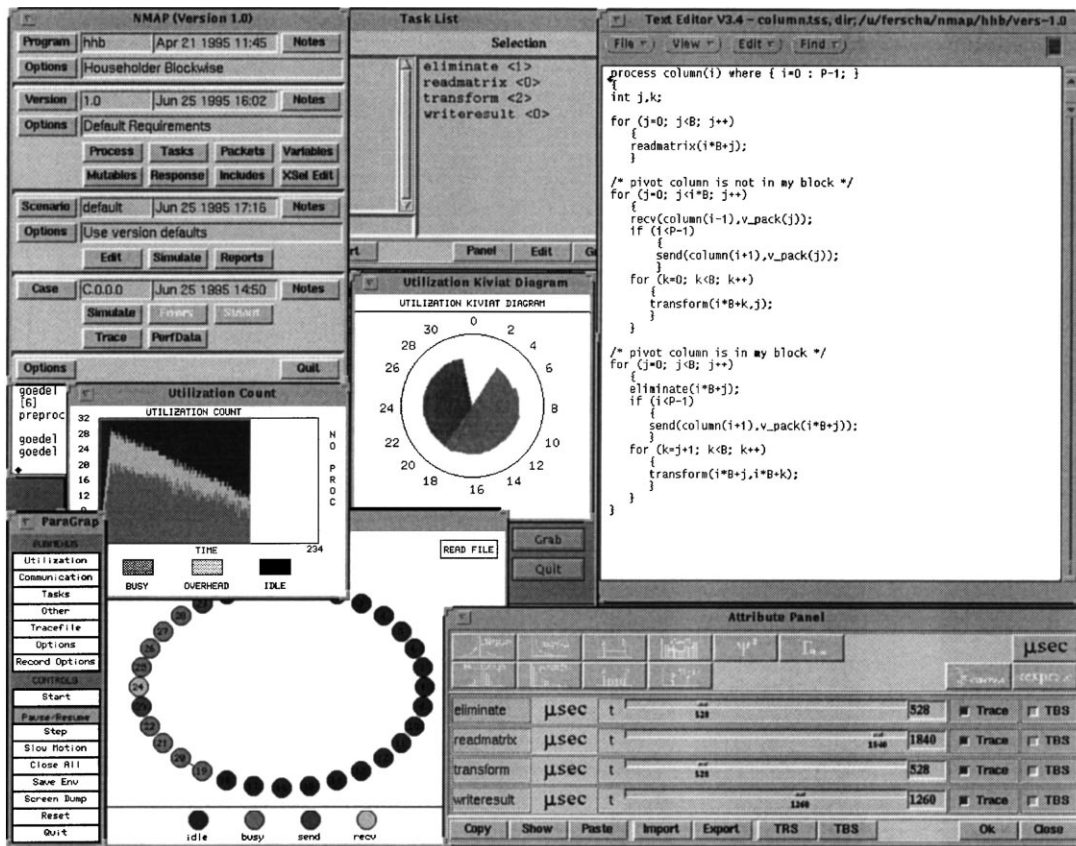


Fig. 8. A screen shot of a typical N-MAP session.

accessing of all program components from a single window. Listbox windows are provided for managing the individual elements of the N-MAP language, e.g. processes, task, packets, mutables etc. and control panels are provided for tasks and packets allowing the user to view at a single glance the various attribute settings of all tasks and packets or groups thereof. Performance data and program execution traces can be viewed with a single button press in the main window as well as program output and errors stemming from the parse, compile or simulation phases. Fig. 8 shows a screen shot of a typical N-MAP session.

### 2.3. N-MAP's direct simulation engine

N-MAP employs a *discrete event simulation* scheme for the simulation of distributed programs. As shown in Fig. 9, the N-MAP simulation environment consists

of a *simulator engine* and a number of *virtual processors* (VPs) which all execute in the common memory of the simulation program. Both the simulator engine and the VPs are implemented as *light-weight processes* within the running UNIX process. During simulation, control is transferred from the simulator engine to the VPs and vice versa by means of a simple *context switch*. Since scheduling and flow control must be handled directly by the simulator engine in order to maintain causal consistency, the context switch suffices as control transfer mechanism and possible overheads which could be introduced by standard threads packages are thus avoided.

When simulating the execution of virtual processors in the memory space of a single program, it is necessary to provide a separate *stack* and *data space* for each VP. In the initialization phase of the simulation, a memory area of fixed size is allocated for the stack of

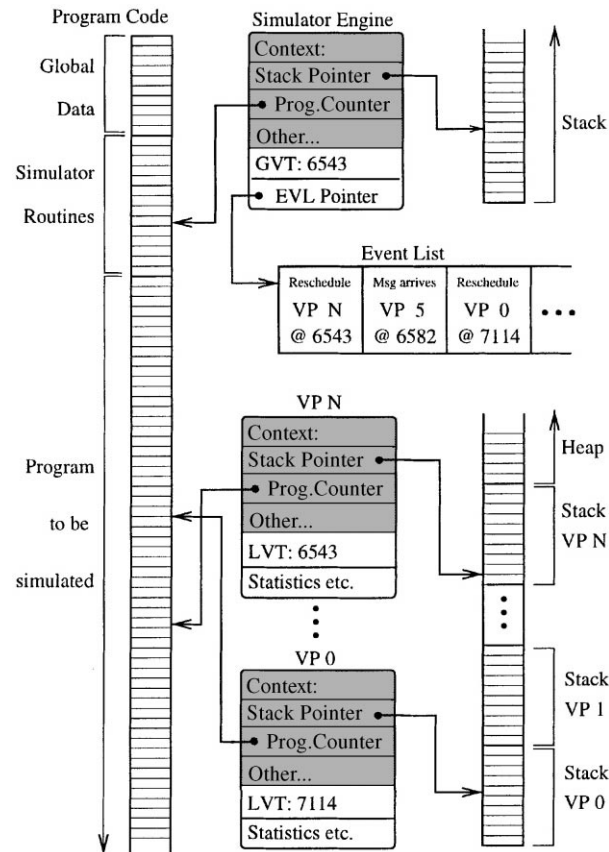


Fig. 9. N-MAP simulation engine architecture.

each VP from the heap. The VP's *stack pointer* is then set to point to the highest address in this memory block (as the stack will grow downward during program execution). This context, i.e. the state of the stack and the processor registers (which include the stack pointer and program counter), is saved with `set jmp()` and stored in the VP's control data structure. Control is then returned to the simulator engine by restoring its previously saved context with `long jmp()` and the next VP is initialized. The question of data is resolved in the N-MAP environment by replicating each data element for each VP at translation time. This must be done only for *global variables*, i.e. variables which may be referenced for any point in the code since variables defined locally within functions are allocated on the VP's stack at execution time and are thus inaccessible for other VPs.

With this mechanism, N-MAP can perform a seamless transition from virtual time progression based on task requirements specifications to the true execution of the task behavior specification.

### 3. Pro-active performance management with N-MAP

Developing distributed applications at the confluence of software engineering and performance engineering activities as described so far cannot be guaranteed to achieve satisfactory performance in every case. In other words, static performance tuning remains inadequate for applications which exhibit non-predictable execution dynamics, and for computing platforms characterized by a high dynamicity. For

these dynamic systems and environments, *adaptive* [9–13] performance tuning strategies appear necessary.

Traditionally, adaptive distributed programs are able to *react* in response to changes by altering the system configuration, so that the action most appropriate to preserve (or even to increase) the efficiency of the application is taken when a change occurs. Although the whole dynamic load balancing research area [14–16] has followed this approach, one faces the severe drawback of lack of timeliness in “reactive adaptivity”: when an action takes place, the system might be in a state different from the one in which it was when the above action was considered appropriate. The time required to determine that the system entered a given state, to decide what action is appropriate, and to actually perform the above action, may indeed be longer than the time taken by the system to enter a different state. In the worst case, the effect of this action may be exactly the opposite of the desired one, so performance may decrease even further instead of increasing.

To overcome this problem, we have introduced the concept of *pro-active* performance tuning in N-MAP, which is based on the principle of predicting the future evolution of the system in order to be prepared to perform a given action at the appropriate time. While a reactive application acts only *after detecting* that a given event (e.g. load imbalance) has occurred, a pro-active one tries to forecast such events and by appropriate preparatory actions tries to avoid entering an undesirable state.

Technically, as opposed to “classical” load sharing or load balancing strategies for distributed applications executing on networks of workstations (NOWs) with node heterogeneity [15,16] that attempt to dynamically redistribute initial and forthcoming load configurations in order to optimize overall execution performance of a particular application, N-MAP does not follow this *reactive* performance management policy: traditional load management literature follows a scheme of collecting state information  $S(t)$  at certain (mostly periodic) instants of time  $t$ , and enacts control decisions  $CD(S(t))$  based on  $S(t)$ . Since establishing the control decision  $CD(S(t))$  uses system resources and takes time, as the diffusion of  $CD(S(t))$  to the involved nodes does in a centralized scheme,  $CD(S(t))$  in most cases will not be timely. Assum-

ing  $\Delta$  time having passed between the time the control decision was made, and the time when the control decisions becomes effective, the probability of a system change in the time frame  $\Delta$  can then be considered a quality parameter for the “timeliness” of control decisions. Opposed to these schemes (“*re-active*” because load management is a re-action to a certain system state) N-MAP applies “*pro-active*” load management: based on a statistical analysis of historical state information,  $\dots, S(t - 2\Delta), S(t - \Delta), S(t)$ , the future state of the system  $\hat{S}(t + \Delta)$  is estimated for the time when the control decision  $CD(\hat{S}(t + \Delta))$  is supposed to become effective. Depending on the forecast quality, this pro-active scheme will presumably exhibit performance superior to re-active schemes, at least for cases where state changes in the time frame  $\Delta$  are very likely.

### 3.1. Distributed application wrapping

N-MAP provides a wrapping facility for distributed applications such that performance data can be periodically relayed from the application to a pro-active performance tuning module. Wrapping is implemented through the insertion of sensors into the program code at code generation time. Based on the metrics delivered by sensors, indicator values are derived which serve to characterize the application’s momentary behavior. Program execution can thus be envisioned as a trajectory through the  $n$ -dimensional indicator space. In order to record the execution trajectory in a memory and time efficient manner, incremental clustering techniques are applied. The resulting dynamically evolving clusters serve to characterize reachable program states and the coordinates of their centroids serve as guideposts for steering program execution towards higher performance. To avoid state exchange overheads and to preserve scalability, the scheme restricts itself to local state information and enacts control decisions locally.

As outlined in Fig. 10, a distributed application, composed of a set of application processes executing remotely on different nodes of e.g. a NOW (network of workstations), is wrapped with a performance management component which automatically controls the applications’ performance. As a matter of system performance observation, each application

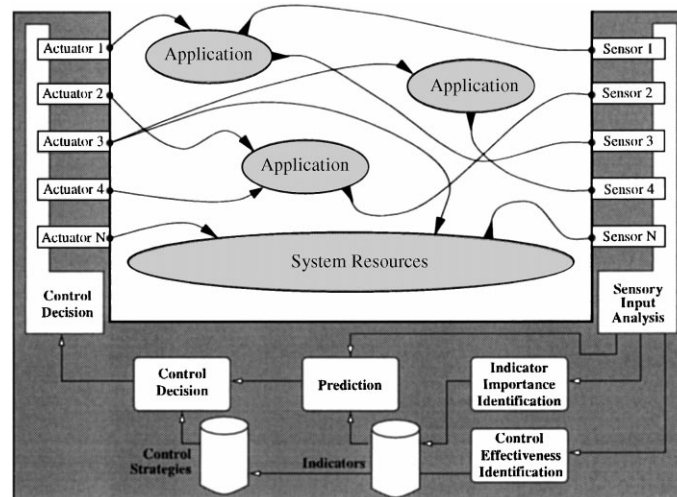


Fig. 10. Application wrapping.

process is monitored by a dedicated “*sensor*”. (Besides application specific sensors, system resource sensors deliver utilization information directly from the underlying system resources.) Sensory input is reported to a statistical analysis component that extracts the relative importance of sensor values, and by that establishes “*performance indicators*”. A prediction component uses the indicator history together with the sensory input to forecast the next system state, upon which a control decision is made. “*Actuators*” then expose the application process(es) to the control decision, thus closing the performance management loop.

### 3.2. Sensors, indicators and actuators

In its simplest implementation, a *sensors* is defined as a global variable which is assigned the current value of its corresponding performance/state metric at appropriate points in the application program code. Although this appears sufficient for most cases, more complex sensor implementations may require a more extensive data collection strategy and/or additional preprocessing of performance/state metrics before assigning a value to a sensor variable. Moreover, various types of sensors are possible depending on the characteristic of the performance/state metric in question. The current implementation of our pro-active

performance management module (PPMM) module allows for *point sample sensors* which register the *momentary* value of a performance/state metric and *cumulative sensors* which sum a specific metric (e.g. occurrence of an event).

Technically, assuming  $n_S$  sensors are defined, PPMM is invoked with a state update function  $PPMM\_update(S_1, S_2, \dots, S_{n_S})$ . Based on  $(S_1, S_2, \dots, S_{n_S})$ , the module then calculates the value of  $n_I$  *indicators* which are used to characterize the current state of program execution. In the most general case, the value of an indicator may be a function of any number of sensor values  $I_i = f(S_1, S_2, \dots, S_{n_S})$ . The task of the PPMM is to optimize the value combination of the indicator vector by setting the values of *actuators* according to the control decision based on the analysis of the indicator data.

As opposed to sensors which simply relay state information from the executing application to the PPMM, *actuators* are inserted at appropriate points in the program source which *set* the values of execution program parameters affecting the run-time behavior of the application directly. Since indicator values may also depend on the momentary values of the  $n_A$  actuators:  $I_i = f(S_1, \dots, S_{n_I}, A_1, \dots, A_{n_A})$  inverse functions must be given that allow the actuator values to be calculated based on a given indicator value  $I_i$ :  $A_k = f_k^{-1}(I_i, S_1, \dots, S_{n_I})$ .

### 3.3. Execution trajectories in indicator space

In a geometric interpretation, each vector of sampled indicator data  $\mathbf{p}(t) = (I_1(t), I_2(t), \dots, I_{n_I}(t))$  collected at time  $t$  can be viewed as a point in an  $n_I$ -dimensional vector space. The sequence of sampled points gathered during the course of program execution thus describes a *trajectory* through the  $n_I$ -dimensional *indicator space*. With each point of the trajectory representing a system state that has been visited, the entire execution trajectory can be viewed as the ordered set of all system states which have been recorded during the course of program execution. Using the trajectory information, the PPMM can identify similar states and steer program execution (by controlling actuators) towards those states which have better performance components.

Storing the coordinates of every point of the trajectory, is not feasible due to the enormous amount memory this would require. Additionally, sorting and data retrieval would be too time consuming. A recording method is needed which is fast and makes state information retrieval quick. Since the objective of the PPMM is to identify reachable states similar to the current state and to steer program execution towards them, a method which classifies similar states appears adequate for “storing” the trajectory data. Clustering fulfills these requirements in that similar states can be grouped together, the centroid of the resulting cluster then serves as a representative for all points within the cluster. Static clustering methods are obviously not feasible due to the dynamic nature of the data since new points must be repeatedly added to the data set and clustered. To overcome this difficulty, the PPMM implements an incremental clustering algorithm which allows for continuous, dynamic reclustering of the data points in the indicator space.

### 3.4. Performance forecasting and tuning

With each update of the PPMM, a decision must be made as to which values must be assigned to the actuators in order to reach a state with better performance. Based on the current system state and the clustering information, which represents the execution trajectory in a compact form and thus a set of possible system states, the PPMM must choose to move (in the actuator subspace) towards clusters of states with

better system performance. A number of methods for cluster selection appear appropriate, some promising techniques have been studied in [5]. In any case, after the most promising cluster has been selected, the PPMM controls current actuator values to go for the selected cluster. Execution control is then returned to the application which modifies its execution behavior according to the current actuator settings.

With all processors following this strategy simultaneously, a negative shift in program performance can be prevented, since each processor moves towards that same point in indicator space which promises highest performance given the overall system state. This method of run-time performance tuning, furthermore, has the advantage of being truly distributed, avoiding the communication overheads that the exchange of system state information among processors can induce.

## 4. Conclusions and summary

An incremental performance oriented development tool for distributed programs has been presented in which performance and behavior prediction are treated as performance engineering activities in the early development stages, allowing for the evaluation of alternate implementation strategies far ahead of the actual coding. Guided by the performance predictions given for the initial skeletal program code, the developer may choose only the most promising implementation alternatives for further refinement and investigation.

In the ensuing specification and coding phase, N-MAP provides tools for modeling and simulation which allow for an incremental specification of executional requirements and program behavior until a fully functional and efficient distributed program is achieved. With the integrated scenario manager, detailed and automated investigation of performance and behavior is possible at any time during the development cycle. Through the systematic variation of performance influencing factors, detailed reports are generated which provide valuable insight into program behavior, even for very large parameter spaces.

Furthermore, N-MAP provides a wrapping facility which implements a pro-active strategy for performance management at run-time. Opposed to

“classical” performance (or dynamic load) management methods which *react* after having detecting that an undesirable state (system state of poor performance), has been entered, the pro-active tuning module in N-MAP makes distributed applications “future aware” by wrapping a prediction and control decision component to it, such that based on sensory input applications can plan their individual resource consumption with respect to their anticipated availability.

The key features of N-MAP can be summarized as:

- N-MAP is fully integrated development environment with emphasis on performance prediction, static performance tuning and run-time performance management.
- N-MAP provides language support for data parallelism and parallelism at the task level, together with a virtual distribution model.
- Support is provided for a top-down development approach by progressive improvement of task requirement and behavior specification until full functional code is available.
- Prediction of execution behavior is possible already from early program (task structure) specifications (trace analysis, performance visualization and execution animation).
- For performance prediction, N-MAP supports deterministic and stochastic requirement (workload) specifications, i.e. random variates are made available both from theoretical as well as empirical distributions. Arbitrarily complex, even state based requirement expressions are possible.
- Arbitrary performance prediction precision is possible via incremental requirement (workload) specification and refinement (at low cost due to an efficient simulation kernel, portable to basically any UNIX environment).
- N-MAP allows for an automated generation of (instrumented) source code for simulation of *virtual processor* distributed programs on single processor, as well as an automated generation of (instrumented) source code for execution on real multi-computers.
- A fully automated scenario management generates a systematic set of workload scenarios which can be executed even in a distributed way (i.e. simultaneously on several uniprocessor workstations).
- A graphical user interface is provided for all tool components with the possibility to customize preferred editing facilities.
- A full software distribution is freely available for SGI/Irix and SunSparc/Solaris platforms.

## References

- [1] A. Ferscha, J. Johnson, N-map: a virtual processor discrete event simulation tool for performance prediction in capse, in: Proceedings of the HICSS-28, IEEE Computer Soc. Press, Silver Spring, MD, 1995, pp. 276–285
- [2] A. Ferscha, J. Johnson, Performance prototyping of parallel applications in N-MAP, in ICA<sup>3</sup>PP96, Proceedings of the Second International Conference on Algorithms and Architectures for Parallel Processing, 11–13 June 1996, Singapore, IEEE Computer Soc. Press, Silver Spring, MD, Press, 1996, pp. 84–91.
- [3] A. Ferscha, J. Johnson, A testbed for parallel simulation performance prediction, in: J.M. Charnes, D.J. Morrice, D.T. Brunner, J.J. Swain (Eds.), Proceedings of the 1996 Winter Simulation Conference, 1996, pp. 637–644.
- [4] J. Ferscha, A. Johnson, St. Turner, Early performance prediction of parallel simulation protocols, in: Y.M. Teo, W.C. Wong, T.I. Oren, R. Rimane (Eds.), Proceedings of the First World Congress on Systems Simulation, WCSS'97, Singapore, 1997, September 1–3, IEEE Computer Soc. Press, Silver Spring, MD, pp. 282–287
- [5] A. Ferscha, J. Johnson, M. Kotsis, C. Anglano, Pro-active performance management of distributed applications, in: Proceedings of MASCOTS'98, IEEE Computer Soc. Press, Silver Spring, MD, 1998.
- [6] H.V. Sreekantaswamy, S. Chanson, A. Wagner, Performance prediction modeling of multicomputers, in: Proceedings of the 12th International Conference on Distributed Computing Systems, Los Alamitos, California, IEEE Computer Soc. Press, Silver Spring, MD, 1992, pp. 278–285.
- [7] A.J.C. van Gemund, Performance prediction of parallel processing systems: the pamela methodology, in: Proceedings of the 1993 ACM International Conference on Supercomputing, July 1993, Tokyo, Japan, ACM, New York, 1993.
- [8] T. Fahringer, H.P. Zima, A static parameter based performance prediction tool for parallel program, in: Proceedings of the 1993 ACM International Conference on Supercomputing, July 1993, Tokyo, Japan, 1993.
- [9] G. Shao, R. Wolski, F. Berman, Modeling the cost of redistribution in scheduling, in: Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing, 1997.
- [10] J.N.C. Arabe, A. Beguelin, B. Lowekamp, E. Seligman, M. Starkey, P. Stephan, Dome: parallel programming in a distributed computing environment, in: Proceedings of the 10th International Parallel Processing Symposium (IPPS 96), IEEE Computer Soc. Press, Silver Spring, MD, April 1996.
- [11] M. Cermele, M. Colajanni, S. Tucci, Adaptive load balancing of distributed SPMD computations: a transparent approach,

- Tech. Rep. DISP-RR-97.02, Dipartimento di Informatica, Sistemi e Produzione, Università di Roma Tor Vergata, 1997.
- [12] J. Gehring, A. Reinefeld, MARS – a framework for minimizing the job execution time in a metacomputing environment, *Future Generation Computer Systems* 12 (1) (1996) 87–99.
- [13] J. Casas, D. Clark, R. Konuru, S. Otto, R. Prouty, J. Walpole, Adaptive load migration systems for PVM, in: *Proceedings of Supercomputing 94*, IEEE Computer Soc. Press, Silver Spring, MD, 1994, pp. 390–399.
- [14] T.V. Gopal, N.S. Karthic Nataraj, C. Ramamurthy, V. Sankaranarayanan, Load balancing in heterogeneous distributed systems, *Microelectronics Reliability* 36 (9) 1996.
- [15] D. Arredondo, M. Errecalde, S. Flores, F. Piccoli, M. Printista, R. Gallard, Load distribution and balancing support in a workstation-based distributed system, *ACM Operating Syst. Rev.* 31 (2) 1997.
- [16] S.P. Dandamundi, M.K.C. Lo, A hierarchical load sharing policy for distributed systems, in: *Proceedings of the Fifth International Symposium on Modeling, Analysis and*

*Simulation of Computer and Telecommunication Systems (MASCOTS'97)*, IEEE Computer Soc. Press, Silver Spring, MD, 1997, pp. 3–10.

**Alois Ferscha** received the Mag. degree in 1984, and a PhD in business informatics in 1990, both from the University of Vienna, Austria. In 1986, he joined the Department of Applied Computer Science at the University of Vienna, where he is presently an Associate Professor. He has been active and has published about 50 technical papers on topics related to parallel and distributed computing, like e.g. CAPSE (Computer Aided Parallel Software Engineering, Performance Oriented Distributed/Parallel Program Development, Parallel and Distributed Discrete Event Simulation, Performance Modeling/Analysis of Parallel Systems and Parallel Visual Programming. Currently he is focussed on Object Oriented, Write-Once Run-Anywhere Software Models, Global Net-Centric Computing and Multiuser Distributed Cooperative Work Environments. He has been the project leader of several national and international research projects.