Simulation of Parallel Computations

Lloyd D. Fosdick
&
Carolyn J. C. Schauble

CU-CS-438-89 May 1989

University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

SIMULATION OF PARALLEL COMPUTATIONS

Lloyd D. Fosdick
Carolyn J.C. Schauble

CU-CS-438-89 May 1989

Department of Computer Science
Campus Box 430
University of Colorado
Boulder, CO 80309-0430

SIMULATION OF PARALLEL COMPUTATIONS*

Lloyd D. Fosdick
Carolyn J. C. Schauble
University of Colorado
24 May 1989

Simulation of computing systems has gained increasing importance as the complexity of these systems has grown. It is done for a variety of reasons: to predict performance of systems before they are built, to validate logic at the gate level, to permit software production and testing to proceed in parallel with the construction of a new computer, to test algorithms on speculative systems, and so forth. Our own concern with this topic is in predicting performance of multiprocessor systems, especially in an environment where a designer is interested in exploring the effect of design changes. In this paper we will describe how a new general-purpose simulation tool can be used for this purpose. The description will be illustrated with results from simulations we have done on two-processor, shared-memory systems with different numbers of memory modules, with and without cacheing.

OLYMPUS

Olympus is a general-purpose modeling system, running on a network of Sun workstations. The elements of this system that are pertinent to our application are described here. Further details and applications can be found in several Colorado technical reports (Nutt 1988, 1989; Demeure 1988).

Olympus supports a graph model description of the simulated system. This graph is called a *bilogic precedence graph* (BPG) which is similar to a petri net graph in its use and interpretation. The simulation proceeds with tokens moving along the edges of the graph from one node to the next; nodes "fire" according to certain rules allowing tokens to advance. In the BPG a node is typically associated with a task that is realized by a short computation, a procedure call, that is activated when a token visits the node. In our application tokens represent machine instructions for the simulated machine and the graph represents the architecture of the machine: as the token progresses through the graph the actions associated with the

* This research has been supported by grants from NITS 60NANB8DO838, and NSF DCR-8420944.

execution of that instruction are simulated and elapsed times are recorded.

There are three types of control nodes in the BPG: single-entry-single-exit (SESE); AND; and OR. They are illustrated in Fig. 1. In this figure, and in others we will use later, SESE nodes and OR nodes are

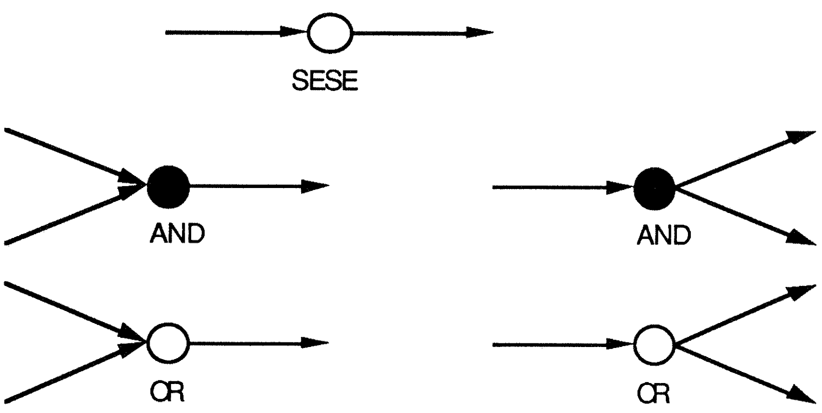

Figure 1: BPG primitives

represented by open circles, and the AND nodes are represented by filled circles. Both AND and OR may have one or more input edges and one or more output edges. The AND node is activated (fires) when it has tokens on all of its input edges; firing implies that the tokens then disappear from the input edges, the task associated with the node is executed, and then tokens appear on all of its output edges. Similarly, the OR node fires when it has a token on one input edge; after firing only one output edge receives a token. (This can occur probabilistically but in our application it is completely determined). The SESE node simply fires when a token appears on the input edge, the token is removed from the input edge and placed on the output edge. The BPG allows another type of node, a data node, but we do not use it. While Olympus takes care of advancing tokens over the graph, it is the modeler's responsibility to supply the procedures that are executed when a node fires.

An important feature of Olympus is that it supports a convenient user interface. The BPG can be drawn on the bit-mapped display of the workstation using a mouse and a palette of objects. An illustration of the display is shown in Fig. 2. The palette appears in the band at the top of the window. The figure shows an intermediate stage in the drawing of a BPG, with the cursor appearing as a cross. One weakness of the present version is that layering is not supported. Thus it is not possible to hide a structure inside a node, so at this time the screen can become rather full and difficult to read even with fairly simple models. A new version that supports layering will be finished soon.

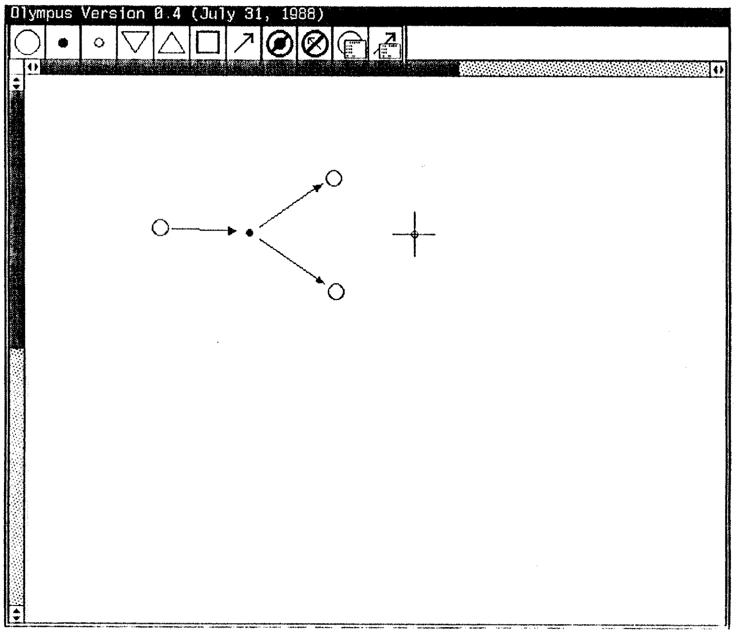

Fig 2: Drawing a BPG

Olympus can be run in step mode in which the user can step through a simulation, one firing at a time, with the display being updated for each step; animate mode, in which the display shows every active token moving once a second; and simulate mode, in which the display is updated every 10 or 15 seconds. In all cases the location of the tokens on the graph is shown on the display. In Fig. 3 the menu for making these selections is illustrated with *Animate* selected.

Olympus runs interpretively, thus it is slow. For our current work this is not a problem, but it would be in any serious production simulation. On the other hand the freedom to interrupt and make changes in the middle of a simulation is an advantage. A compilable version of this system is planned.

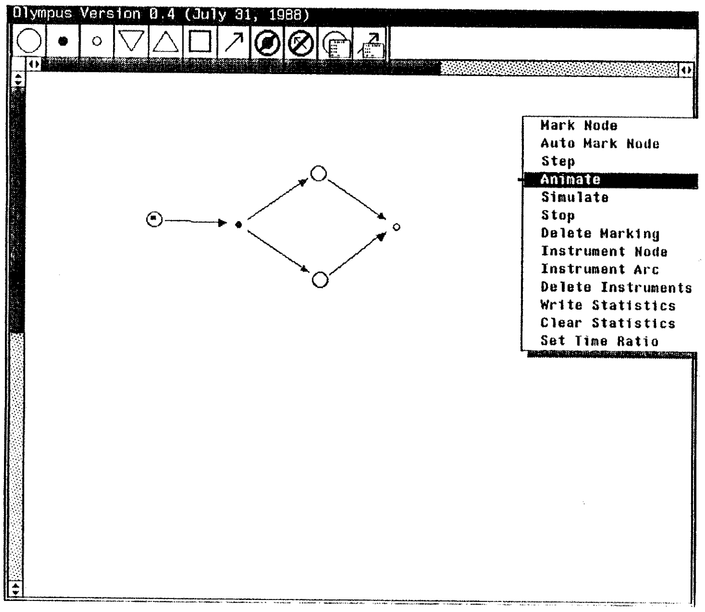

Figure 3: Olympus menu

SOME SIMPLE ARCHITECTURAL MODELS

In this section we will illustrate the use of BPGs to represent architectures of simple systems. We begin with a one-bank memory module, then show a simple uniprocessor, and finally a two-bank memory module. In the subsequent discussion we will see that two kinds of tokens are used: data tokens and control tokens. The data tokens represent instructions and carry on them the following information fields: instruction type, memory address for fetch or store, memory address of instruction itself, time. Control tokens are used for synchronization and carry on them a time field only.

A Simple Memory Module

A memory bank can be modeled by the BPG in Fig. 4. This BPG is initialized with a single control token whose time value is zero. An incoming data token, representing a memory access request, enters this graph on the edge marked *enter*. The AND nodes, U and Z, assure that only one data token may be processed by the graph at a time. Node U is a smart node: it compares the times on the control token and the data token and puts the maximum of these on the data token that emerges from U. When the data token finally causes Z to fire the data token is replicated, the copy going along the edge directed to U and acting as a control token, the original proceeding along the *exit* edge.

Within the memory bank itself the data token will need to fetch or store a value: the two possible paths are shown in the figure. When V fires it will put the data token on the left branch or the right branch according to whether the data token requires a fetch or store. Both operations will take some time and the time on the data token is updated accordingly when W or X fires.

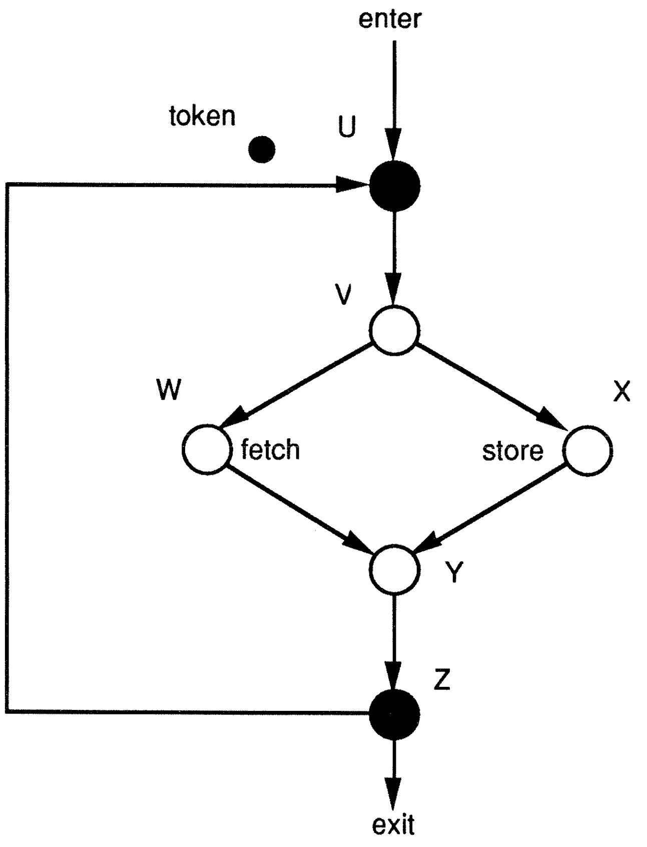

Figure 4: BPG for one memory bank.

A memory request may come from any one of several points in the BPG for a machine model. Thus in such a graph the memory module illustrated in Fig. 4 acts like a subroutine for handling references to a particular memory bank. In the next example we will represent the memory module by the structure shown Fig. 5.

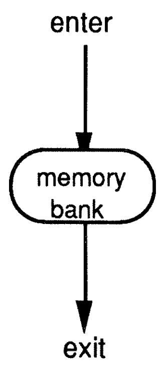

Figure 5: Abstraction of memory module.

A Simple Machine Model

Our machine models are trace-driven. Thus the input to the model is a file of tokens representing the complete sequence of instructions that is to be executed. In a multiprocessor model the input consists of traces for each processor. The implications of this will be discussed later. Now we look at a single processor system.

The BPG in Fig. 6 represents a simple processor. It has two basic parts: instruction prefetching and instruction execution. The prefetch part is the subgraph consisting of nodes {A,B,C,D}. Synchronization in each part is done with AND nodes and control tokens, as in the memory module example.

Notice that a token cannot enter the prefetch part unless there is a control token on edge (D,A). Similarly, one cannot enter the execution part unless there is a control token on (P,C). The initial state of the BPG is shown in the figure. After a data token passes through the execution part and causes P to fire a token will appear on (P,C) which allows the next token to enter the execution part and another to enter the prefetch part.

The execution part allows for four kinds of instructions: those which fetch data from memory; those which store data in memory; those that do not access memory; and those that fetch and store. Each of the four branches from E handles one of these cases. When the data token passes through G, H, J, or L the time is updated with a value appropriate to the instruction.

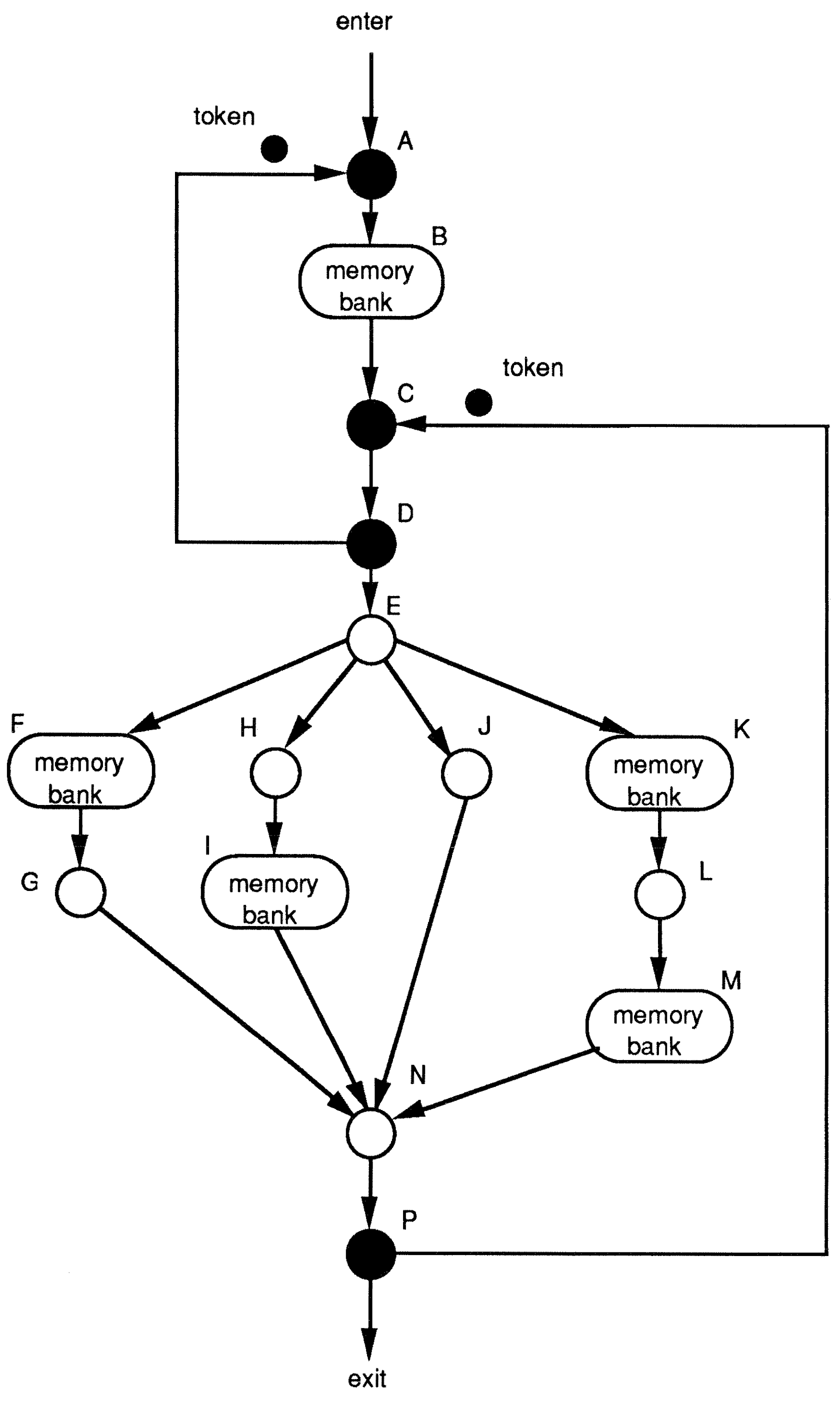

Figure 6: BPG for simple processor

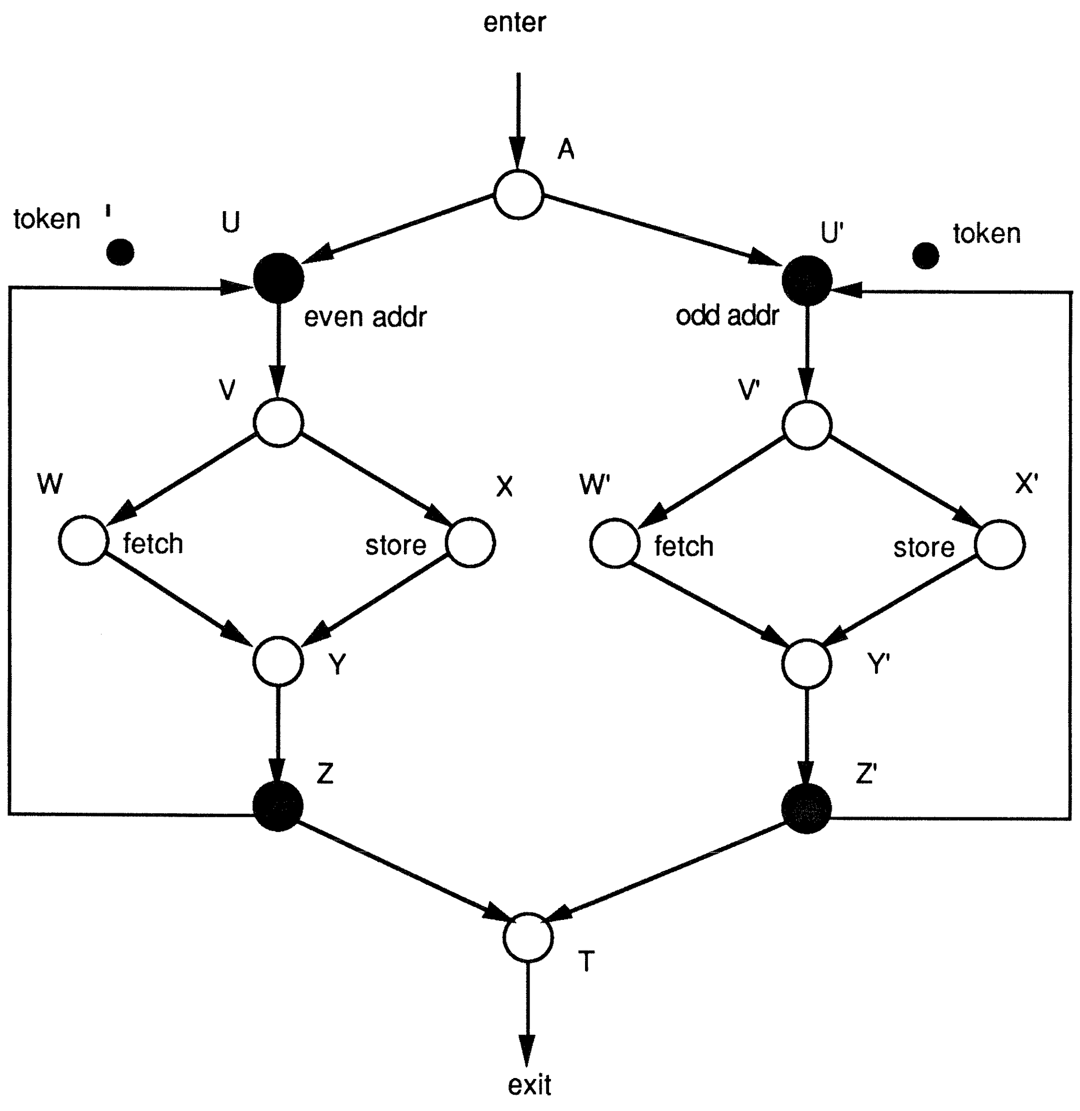

Figure 7: BPG for two memory banks

Two-bank Memory Module.

Here we illustrate the modeling of a two-bank memory in which even addresses are attached to one bank and odd addresses to the other. The BPG is shown in Fig. 7. The OR node, A, allows two memory accesses to go on simultaneously, so long as each is concerned with a different bank. When it fires it sends the data token to the out-edge according to the address it sees on the token: even addresses on edge (A,U), odd addresses on edge (A,U'). It should be evident how an n-bank system can be modeled in this way.

TWO-PROCESSOR MODEL AND THE OLYMPUS DISPLAY

In this section we present an example illustrating the appearance of the Olympus display for a model consisting of two processors and four memory banks. In Fig. 8 a snapshot of the SUN display for this model is shown.

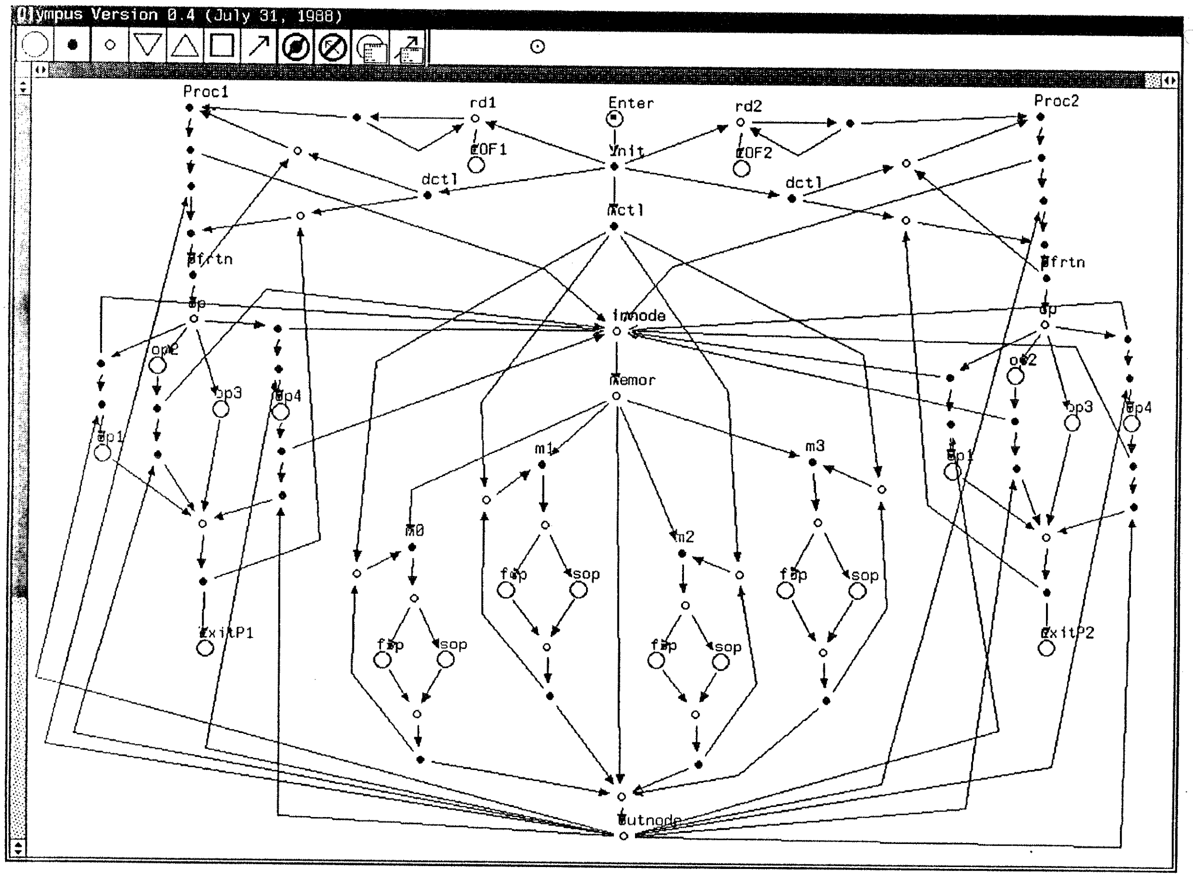

Figure 8: Two processors on Olympus display

The two processors are represented in the subgraphs on the left and right sides. They contain nodes labeled *op1, op2, ...* and should be fairly easy to recognize from our earlier discussion. The four memory banks are in the center of the picture containing nodes labeled *fop* and *sop* (for fetch operation and store operation), and again they should be easy to recognize.

Execution of the model begins when a control token is placed at the node labeled *enter* at the top-center of the graph. This will cause *init* to send control tokens to a number of points on the graph in order to initialize it. Also, the control tokens it sends out cause *rd1* and *rd2* to fire. When *rd1* fires it reads a data token from a file and puts it on its out-edge. This token represents an instruction that is to be executed by processor 1 in the model. The short loop containing *rd1* thus provides the sequence of tokens that is to be executed by processor 1. These tokens enter the processor portion of the graph at *Proc1*. Similarly, *rd2* reads tokens that are to be executed by processor 2.

When the simulation is executed the tokens can be seen moving around this graph. In step mode each new state is displayed under mouse control, providing a convenient tool for debugging. Bottlenecks that affect performance become visible in animation mode. However, without a layering feature the picture is rather cluttered and so the effectiveness of the tool for these purposes is now limited to rather simple systems.

ILLUSTRATION OF RESULTS

The simulation tool that we have described here can be used to obtain useful information about the effect of varying design parameters such as memory latency, number of memory banks, and cache size. For example, it is possible to estimate execution times and speedup as a result of such variations. We illustrate this for two very simple computations, a SAXPY operation and an insertion sort.

The basic model that we have used in these illustrations is a 2-processor, shared-memory machine, essentially like that shown in Fig. 8. In examples with multiple memory banks the addresses are interleaved in the usual way. The cache is direct mapped and shared by the two processors. The traces are constructed from a sequence of National Semiconductor 3200 instructions for the computation.

The SAXPY computation evaluates $s \times X + Y$ where s is a scalar and X and Y are vectors. The computation is distributed over the processors in round-robin fashion, and the vectors have length 20. The simulation was run on a single SUN 3/60 processor and took about 10 minutes.

The sort computation is done by giving each processor half of the list to sort by insertion; after the two parts are sorted they are merged by one of the processors. Because of some current limitations of Olympus we were restricted to a list of nine elements. The simulation was run on a single SUN 3/60 processor and took about 20 minutes.

In Figs. 9, 10, 11 we show a series of results for SAXPY, with each figure corresponding to a different cache size. The abscissae in these figures represent values of f which is the ratio of memory latency (m) to the time (a) for an integer addition; the ordinates represent total computation time in units of a: notice that the scale on the ordinate axis varies. The trace for each processor consisted of 71 tokens. In Figs. 12, 13, 14 we show a corresponding series of results for sort. The trace for the first processor consisted of 64 tokens, and for the second, where the merge was also done, was 238.

In Fig. 15 we show the speedup for the SAXPY computation on two processors. To obtain this we simply ran the trace on one processor and then split it across two processors as described above: the ratio of the times is the speedup. Here all computations are done with f = 1.0. The ordinate shows the speedup, the abscissa shows the number of memory banks. We have not yet been able to explain the anomolous speedup for a cache size of 64 and 4 memory banks.

With information like this gathered from an ensemble of realistic problems and knowing costs (for increasing cache size, memory banks, or f), we can see how it would help to make design decisions. However, before this system can be used in this manner a number of important things remain to be done. We comment on this below.

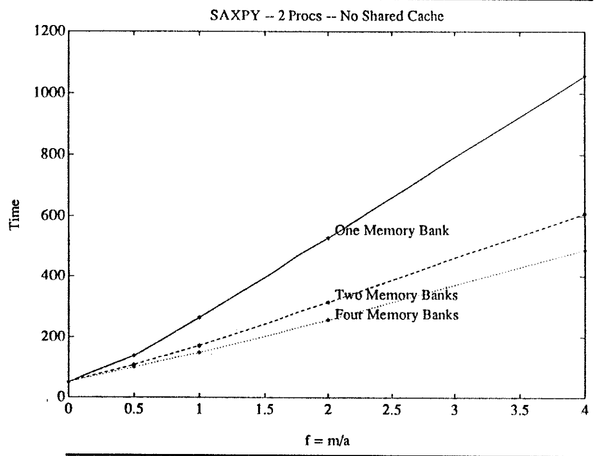

Figure 9

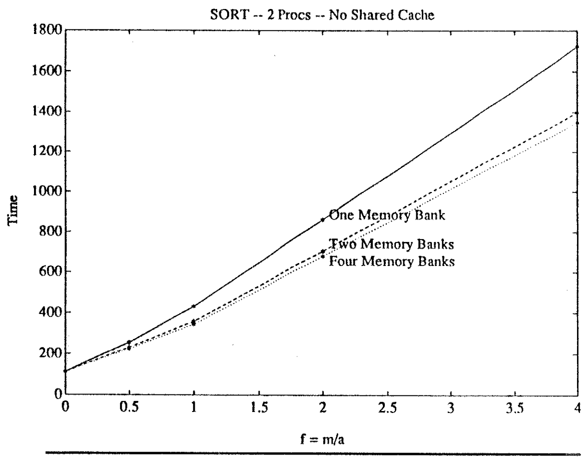

Figure 12

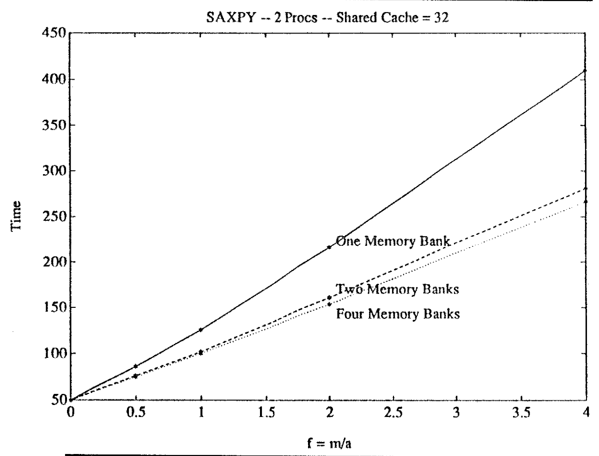

Figure 10

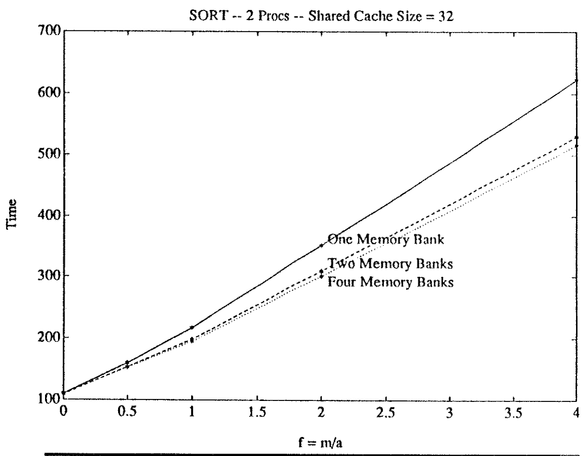

Figure 13

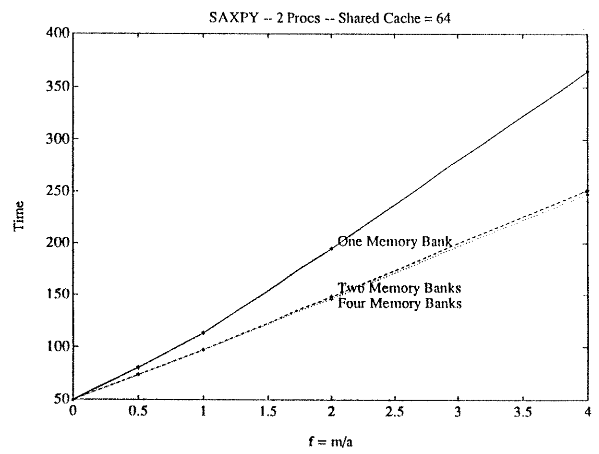

Figure 11

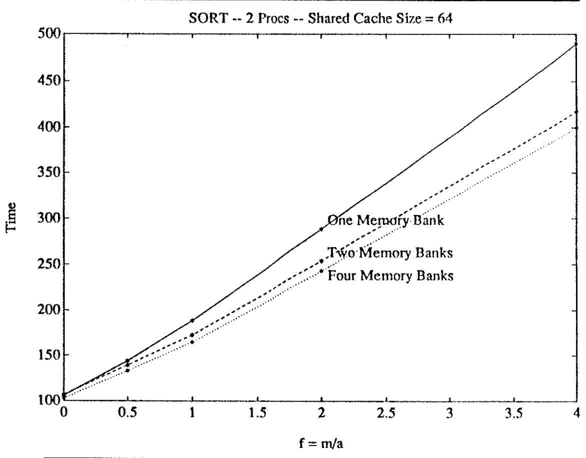

Figure 14

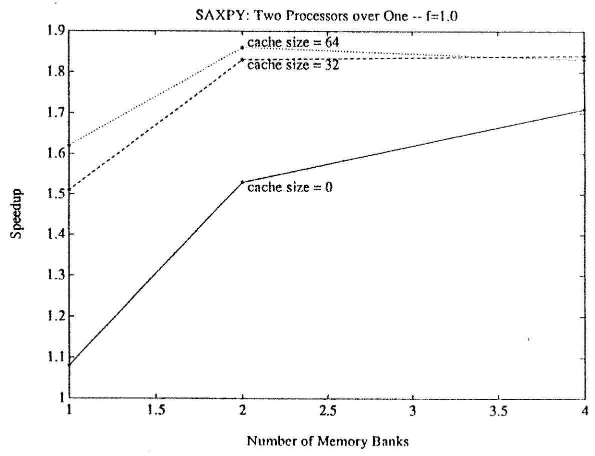

Figure 15

CONCLUSION

Our objective here has been to describe a tool we are working on for the simulation of multiprocessor systems. It is based on a discrete event simulation tool that provides a convenient user interface and a great deal of flexibility making it easy to develop different computer models quickly.

Our long-range goal is to further our understanding of how to predict the performance of complex multiprocessor systems by simulation. We are interested in practical tools that provide a user with the capability for exploring the effect of altering designs and design parameters with relative ease. The tool we have described here shows some promise for meeting our objectives but there remain a number of problems to be solved.

Any practical scheme must include an automatic scheme for developing tokens for the model. Thus a tool for translating a parallel program into an appropriate token stream must be developed. We must also provide some facility for modeling alternate execution traces that fairly represent the nondeterministic behavior of a multiprocessor. This can be done by putting some nondeterminism in the model or by using a monte carlo scheme to construct an ensemble of execution traces for a deterministic model. We intend to explore both of these avenues. We note that some experimenters [Darema-Rogers, Pfister and Ko, 1987] have used the nondeterminism of their host system and have relied on it to supply the nondeterminism for the model. This seems to us like an approach that may not be expected to provide very satisfactory results.

It is also evident that the slow speed of the current system is far too restrictive for serious modeling. What needs to be done is to estimate from the current system the kind of improvements that would result from moving it on to a more powerful computer, at least for the backend processing. Another avenue that needs to be explored is simplification. In particular, we would like to see to what extent the traces can be compressed without sacrificing too much predictive power.

Finally, it is necessary to provide a mechanism for validation of the models. Our first validation experiments will be done with a 20 processor Encore Multimax system. In a parallel research effort at Colorado techniques for careful performance measurements are being explored based on this machine. These techniques are based entirely on software "probes" which are intended to be suitable for application to a range of machines not too disimilar from each other.

REFERENCES

Darema-Rogers, F., G. F. Pfister, and K. So. 1987. Memory access patterns of parallel scientific programs. *Proceedings ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*. 46-58.

Demeure, I.M. 1989. A model (DCPG/PAM) and a graphic tool (VISA) for distributed computations. *Thesis proposal,* Department of Computer Science, University of Colorado, Boulder CO 80309.

Nutt, Gary J. 1988. Olympus: An extensible modeling and programming system. *Tech. Rept. CU-CS-412-88*, Department of Computer Science, University of Colorado, Boulder CO 80309.

Nutt, Gary J. 1989. A flexible distributed simulated system. *Tech. Rept.*, Department of Computer Science, University of Colorado, Boulder CO 80309.