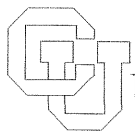
Some Uses of Simulation in System Design *

Gary J. Nutt

CU-CS-059-75



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

* This work was supported by the National Science Foundation under grant number GJ-42251.

SOME USES OF SIMULATION IN SYSTEM DESIGN*

by

Gary J. Nutt
Department of Computer Science
University of Colorado
Boulder, Colorado 80302

Report #CU-CS-059-75 January, 1975

* This work was supported by the National Science Foundation under grant number GJ-42251.

ABSTRACT

In this paper, some simulation design techniques used in the study of a new architecture called the multi associative processor (MAP) architecture are discussed. The architecture executes multiple single-instruction-multiple-data stream (SIMD) programs simultaneously (i.e., it is a specialized multiprocessor). The design analysis of the system includes topics such as the generation of a suitable job mix for the machine; simulating the execution of a SIMD program; and simulating the competition between executing programs for various shared resources.

The approach to analysis of the MAP system design is based on a highly detailed (low level) simulation model that interpretively executes a single SIMD program at a time. This interpreter provides a medium for testing various SIMD algorithms that could be executed on a real system corresponding to the hypothetical MAP system. Although the interpreter does not allow for the simulation of the simultaneous execution of two or more SIMD programs, the single executions are measured with a (simulated) hardware or software monitor to obtain data indicating the performance of that program in an isolated (non-competitive) environment. These data are then used to drive higher level (lower detail) simulation models that focus on particular aspects of the design, e.g., the competition for main memory cycles.

INTRODUCTION

The multi associative processor (MAP) computer system is a hypothetical machine employing eight control units and an arbitrary number of identical processing elements (see Figure 1), [3]. Each control unit, in combination with a subset of the processing elements, operates on a single instruction stream through the control unit and multiple data streams, one through each processing element (i.e. SIMD operation). The instruction execution cycle for one control unit and one or more processing elements can be described as follows: The control unit fetches an instruction from the main memory and then decodes the instruction to obtain a series of low level operations that each processing element must execute in order to carry out the instruction. The control unit "broadcasts" this set of operations over the instruction bus shown in Figure 1, one at a time, to the set of processing elements that are currently assigned to that control unit. The time required to broadcast one operation is called a control unit cycle. Each processing element executes the operations on data that resides in a memory that is local to the processing element. Since it is not likely that all instructions should apply to each processing element that is assigned to the control unit, a mechanism to temporarily activate or deactivate a given processing element is required; we refer to this mechanism as an "associative unit" and further discussion of the unit is provided in the paper by Arnold, [1].

The processing element memories are loaded sequentially by the control unit from the main memory. This loading amounts to data loading rather than program loading, since the processing element memory is not used to store instructions. In order to provide this

capability to transfer data from the main memory to the processing element memories, a data bus is required, as indicated in Figure 1. Note that the philosophy of this machine will require high utilization of the instruction bus system and relatively low utilization of the data bus system provided that high I/O programs are not executed on the machine. For this reason, and since the data bus will generally be wider than the instruction bus, the architecture incorporates dedicated instruction busses and shared data busses. Although the general architecture of the machine is specified above, and in Figure 1, the parameters of the design are subject to the actual condition under which the machine might be used. For example, the main memory is shared amongst eight control units and must be designed to support the simultaneous access of all control units and the I/O system. What bandwidth requirements are imposed on the memory system if we assume that global data and instructions are stored in separate banks? Given a job load, how many processing elements should be assigned to a given data bus connection (as indicated in Figure 1). It is these aspects of the system design that we discuss in the remainder of this paper.

EVALUATION OF THE DESIGN

The plan of attack to the evaluation of the design of the hypothetical MAP system is summarized in Figure 2. The first requirement is to generate a set of programs that might represent a job load on a MAP system. Each program is then executed on an interpreter to allow data to be gathered about the execution in an isolated environment. After several copies of data have been obtained, these data are used to drive simpler simulation programs that model resource competition. In the following, a more detailed discussion of each phase in the design evaluation is given.

The MAP program interpreter, called MAPSIM, is a complex simulation program used to interpret a single MAP program stored in a pseudo main memory in absolute binary format. Thus, it is used to model a single control unit and a multiplicity of processing elements, allowing actual programs to be written and executed in a noncompetitive environment (i.e. resources are always immediately available as required). The input data to MAPSIM includes a designation of the number of processing elements to be used, the amount of processing element memory required, and a MAP program in absolute binary form. MAPSIM is written in the assembly language of the Control Data 6400 (called COMPASS) since it is desirable that the program be as efficient as possible during execution. This is necessary due to the large degree of parallel processing element activity that must be simulated on the sequential Control Data machine. Even with MAPSIM coded in assembly language, the simulated time/real time ratio is much greater than one. This ratio is highly dependent on the number of processing elements being simulated and on the activity of the set of processing elements with respect to

the set of operations being broadcast by the control unit.

In order to see why this overhead is incurred by MAPSIM, consider the action taken for each machine instruction execution. First, MAPSIM fetches an instruction from the pseudo main memory, this instruction is then decoded into a set of actions that the active processing elements must execute. Note that these operations required to simulate the effect of a machine instruction do not directly correspond to the theoretical set of operations broadcast over the instruction bus as described previously. It is only necessary that the whole machine instruction have the same effect in MAPSIM as the uninterrupted sequence of operations would have had in a real MAP system. The set of processing elements currently allocated to a processing element is separated into two lists; the first list chains together all currently active processing element descriptors, and the second list includes the remaining processing elements. Therefore, to apply the instruction to the set of active processing elements, the list of active processing elements must be traversed.

All MAP program input and output (to and from the main memory) is accomplished by including a FORTRAN subroutine, called IOSUBS, with MAPSIM. The MAP program then makes "supervisory calls" which are passed on to the FORTRAN input/output routine.

Since MAPSIM expects the program to be defined in an absolute binary format, it was necessary to provide an assembler to translate symbolic MAP programs into a binary format. Initially, some crude attempts were made at writing an assembler from scratch, but no really good software resulted. For example, the early assemblers would not handle expressions in the operand field, they had no macro capability, the pseudo operations were very limited, etc. It finally occurred

to us to use the macro capability of the COMPASS assembler itself. Then we would have a very powerful assembler with features that would have required several man-months for us to build from scratch. Each symbolic MAP program immediately calls a macro that purges all of the usual COMPASS statements. Subsequent macros are defined for each operation code such that the operand field can be evaluated by the usual COMPASS mechanism and then a predetermined bit pattern and address will be generated for the operation. This approach to assembler writing has been quite lucrative, allowing for easy assembler definition, implementation, and modification.

As is noted in Figure 2, MAPSIM is used to execute a MAP program, producing user-defined output. This has proven to be a worthwhile approach, since it has provided a medium for writing and testing a diversity of programs that illustrate several application areas for MAP, [4]. Additionally, the existence of MAPSIM gives one the opportunity to write monitoring routines to simulate software and hardware monitors. MAPSIM is designed to call a subroutine named MONITOR at the completion of each instruction cycle.* If the user does not wish to monitor his program, the default MONITOR simply returns control to MAPSIM. Otherwise, this option allows one to define any special purpose simulated monitor to satisfy the current need. Once the monitor has been called, it is free to inspect any of the tables maintained by MAPSIM, e.g. the active processing element list, the simulated time, whether or not the instruction made a data reference to main memory, whether or not the instruction required the data bus, etc. The output

---

* It is also possible to have MAPSIM call MONITOR only after certain instruction have been executed. This option is invoked by reassembling MAPSIM with appropriate assembly-time options set.

provided by the monitoring routine (and the other routines it may invoke) is defined by the user. He may generate a set of full trace data, a partial trace, or merely collect statistics to define a distribution reflecting the character of the MAP program. It is also possible to use the monitoring routines to perform certain computations that would normally be classified as artifact, (for an example of such an application, see [4]).

Although MAPSIM has been carefully written in assembly language to minimize the required execution time, the monitoring routines will usually be written in some combination of COMPASS and FORTRAN. Although these routines may be called very frequently they will most likely perform only a minimal amount of computation. They must also be able to do flexible I/O. Furthermore, we see the MAPSIM code as being very static, while the monitoring routines will be everchanging to reflect the current interest of the analyst.

As an example of the use of the monitoring routines, consider the problem of determining the load on the interface between the control unit and the main memory as generated by a given program. One can distinguish between instruction fetches and global data references to the main memory. In order to determine the load, then, the monitor must examine the activity caused by each instruction executed on the control unit. After each instruction is processed, its execution time is looked up in an operator table. This time is used as the time since the last instruction fetch, and the data can be written to an auxiliary file to generate the trace of instruction fetches or it can be entered into a data structure to generate a distribution of the frequency of instruction fetches. Figure 3 is a distribution of the instruction fetch frequency for a given program to compute the shortest path between two

nodes in a graph.

In order to determine the global data reference distribution, one must determine if the current instruction referenced memory or not. This information can also be saved in a table. If the instruction does not reference data, the instruction time is added to an accumulator that keeps track of the time since the last data reference; otherwise, a portion of the instruction execution time for this instruction is added to the time since the last reference and then the sum is entered into a distribution. Again, this data could be used to generate a full trace of data references. Figure 4 indicates the distribution for the same program execution used to derive Figure 3.

MAPSIM has been a useful tool for analyzing single program execution. It has allowed us to explore such potential application areas as numerical mathematics, operating systems, and operations research. Furthermore, we have been able to expose several weaknesses in the original instruction set for the machine; this has resulted in the implementation of a second version of the assembler and interpreter.

However, this approach to design evaluation avoids all forms of resource competition, hence, the design of the critical portions of the hardware cannot be directly studied. For example, MAP must allow for up to nine simultaneous requests for access to the main memory (see Figure 1). There are several strategies to the design of such a memory: One may choose a n-way interleaved memory, hoping to reduce conflicts; or each control unit may have a dedicated instruction memory, and share a ninth module among all control units and the I/O system. In this latter design we are still led to consider interleaving. The other critical resource for which competition is ignored in MAPSIM is the data

bus system.

There are two approaches that could be taken at this juncture: MAPSIM could be refined to execute eight instruction streams in a quasi-parallel fashion, or data can be taken from the single control unit version and used to drive simpler (i.e. less detailed) models of MAP that simulate the competition for resources and the parallel activity. Although the former possibility has not been disregarded, it is not the initial method of investigation that has been applied to the problem. The current version of MAPSIM is already a complex program; the introduction of more parallelism into the interpreter would require a significant programming effort that we are not willing to attempt with our current resources. A benefit of the multi control unit MAPSIM approach that is not available with the alternate approach has to do with the design and evaluation of operating systems for MAP. Without simultaneous multiple instruction execution, one cannot really test certain facets of the machine having to do with interprocess communication and coordination.

Alternatively, the low level models are relatively inexpensive to build and execute. The particular portion of the system that is the subject of interest can be isolated and rigorously tested with the trace data or distributions from monitored MAPSIM runs. In the next section, we discuss an example of the use of such a model to examine parameters for the memory system design. These models are generally much less complex than MAPSIM, and tend to focus on a particular aspect of the system. The emphasis here is generally on the time required to generate the model, hence we may use a variety of high level languages to implement the simulation program rather than assembly language.

A MAP MEMORY SYSTEM MODEL

In this section we discuss a simple simulation model of the connection between the main memory system and the control units. It is assumed that the the memory system is composed of two separate subsystems: One to store all control unit programs and the other to store all global data. The memory cycle time for the model is determined as some multiple of control unit cycle time (as mentioned in the introduction). Current technology for building main memories and control units would suggest that a memory cycle is equal to about eight control unit cycles, i.e., a control unit cycle is assumed to be about 100 ns and a physical memory cycle would be about 800 ns using a core memory technology. In the memory model, the memory cycle time is an input parameter; a value of sixteen is consistent with the technology for building auxiliary core storage, a value of eight corresponds to current technology for primary memory core storage, and a value of four corresponds to semiconductor memory technology. In one case we use a value of two for the memory cycle time and in this case we are supposing that the average amount of time that a control unit requires for an instruction fetch is diminished by the existence of an instruction look-ahead buffer in each control unit. In order to produce a better model of this latter situation, one really should redefine the distribution of control unit references to the instruction memory reflecting the less frequent number of memory references. The model would also have to be changed to reflect the interleaving that would be required with the instruction look-ahead approach. The current model still produces useful results, since it will indicate if further refinement is necessary, or if the present approach is sufficient.

The simulation language used to write the memory conflict model is the E-net system, [2]. This language was chosen since the analysis is somewhat preliminary in nature, (i.e., can the memory system be designed to handle the load without resorting to the instruction buffering within control units or to separate memory modules), therefore, it is desirable to be able to rapidly produce the simulation program without being too concerned about the run-time efficiency of the program. The model incorporates eight control units, each operative, independent of the others, and a memory system with a seven place queue. The inter-reference times for the memory system are specified either by the distribution shown in Figure 3 (for the instruction references) or by Figure 4 (for the data references).

Our premonition about the heavy use of the instruction memory was born out by the simulation runs. Noting that the mean time between instruction fetches was found to be about 5 cycles, then a single memory requiring eight cycles for a reference was destined for trouble. Therefore, the model was exercised with a memory cycle time of 2, 4, and 8 units of time and a summary of the results is shown in Figure 5. It was unnecessary to run the model with a cycle time of one, since this amounts to specifying a private instruction memory for each control unit. The figure plots the per cent of the total simulated time that <u>exactly</u> n jobs were waiting for the memory against n, ($0 \leq n \leq 7$). For the cycle time of eight units, the mean number of jobs waiting for the memory is about 6.3; if the cycle time is set to four, the mean number of jobs drops to about 5.7; for a cycle time of two units, the mean is about 4.4 jobs. Our conclusion is that distinct instruction memories are required for each control unit in order to keep the memory conflict

down to some reasonable figure, since even the cycle time of two control unit cycles produced an excessive amount of blockage at the memory. This experiment is slightly pessimistic since control unit blockage due to other factors such as bus conflict have been ignored.

In the case of the data memory, our intuition told us that we could probably get by with a single uninterleaved memory. The results of the simulation, shown in Figure 6, proved this not to be the case. The model was run with cycle times of 4, 8, and 16 units and the memory interference, even for a cycle time of four, was significant. The mean number of jobs waiting for memory was found to be about 2.7, with a cycle time of four. Again, the control unit is assumed to be free from other resource conflicts. The program that was used to generate the memory reference distribution was a relatively short program, and a significant portion of the execution time for the program is used to load data into processing element memories from the main memory. In a more realistic situation, programs would be expected to make fewer data references, thus increasing the mean time between references to the data memory. Although more MAPSIM programs should be tested before gathering any firm conclusions about the memory design, the current data indicates that an effective memory cycle time of four control unit cycles is plausible for the data memory.

SUMMARY

The multi associative processor system is a new architecture that is highly dependent upon the design of the methods of connecting the control units both to the main memory system and to the set of processing elements. The design of MAP is based wholly on a set of simulation models ranging from the detailed single program interpreter written in assembly language to high level models such as the memory system model written in a special purpose simulation language.

The current status of the MAP project is that a detailed interpreter with a simulated monitoring capability is in production and sample array processor programs are being written and executed. Data from these sample programs are now being collected and used to drive higher level models, such as the memory model and a FORTRAN model of the bus system.

REFERENCES

1. Arnold, D., "Multi Associative Processor Systems Architecture", University of Colorado, Department of Computer Science, Technical Report No. CU-CS-051-74, (August, 1974).

2. Nutt, G. J., "Evaluation Net Simulation System Reference Manual", University of Colorado, Department of Computer Science, Technical Report No. CU-CS-042-74, (April, 1974).

3. Nutt, G. J., "An Overview of a Multi Associative Processor Study", Proceedings of the 1974 ACM Conference, San Diego, (November, 1974).

4. Nutt, G. J., "Sample Programs for a Hypothetical Computer", University of Colorado, Department of Computer Science, Technical Report No. CU-CS-058-74, (October, 1974).

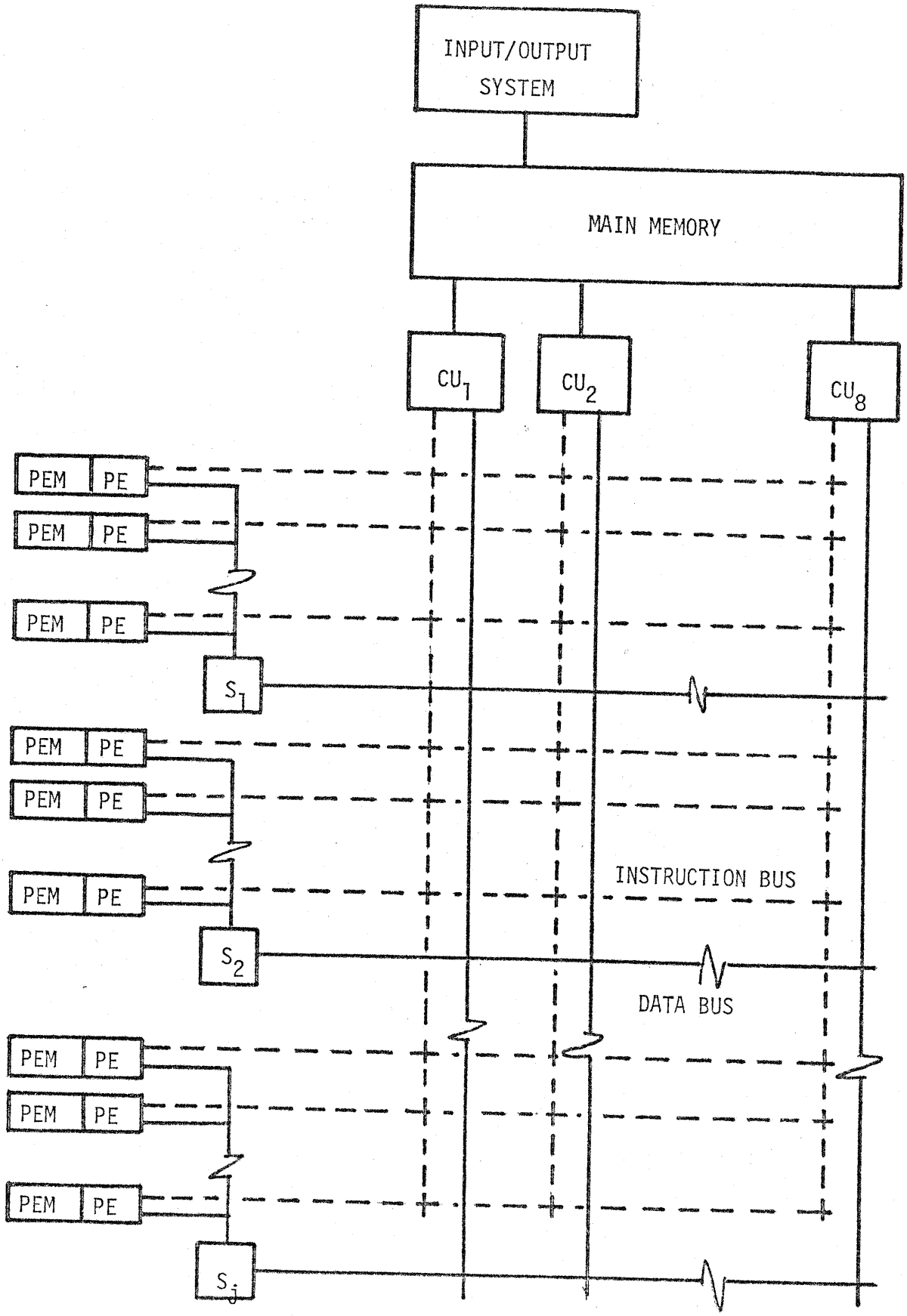
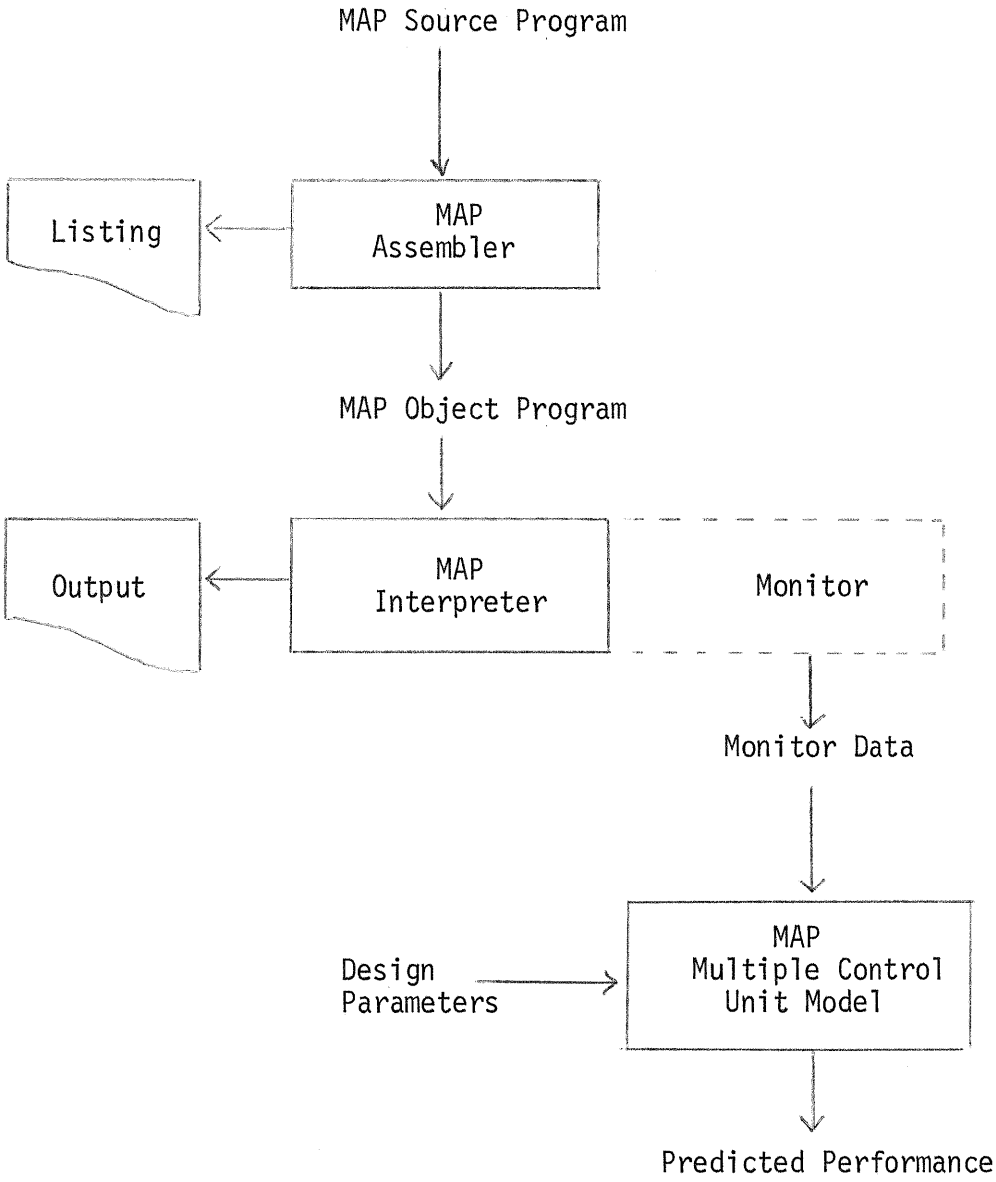
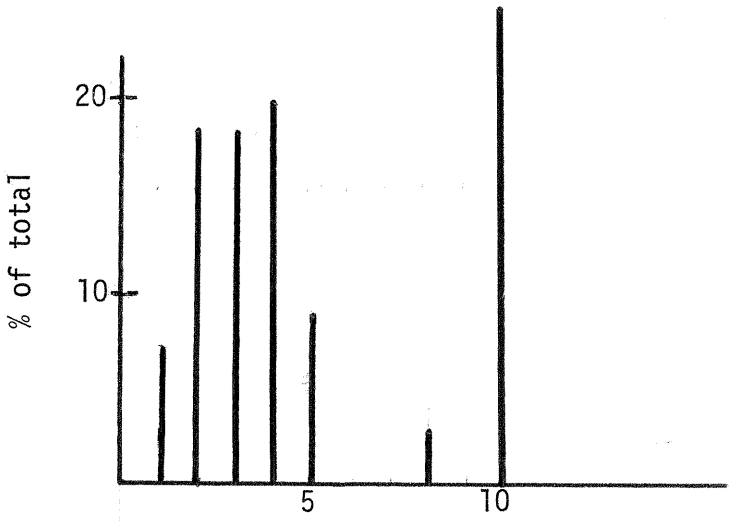

INPUT/OUTPUT SYSTEM
MAIN MEMORY
CU1
CU2
CU8
PEM
PE
S1
S2
Sj
INSTRUCTION BUS
DATA BUS

FIGURE 1



Modeling Components

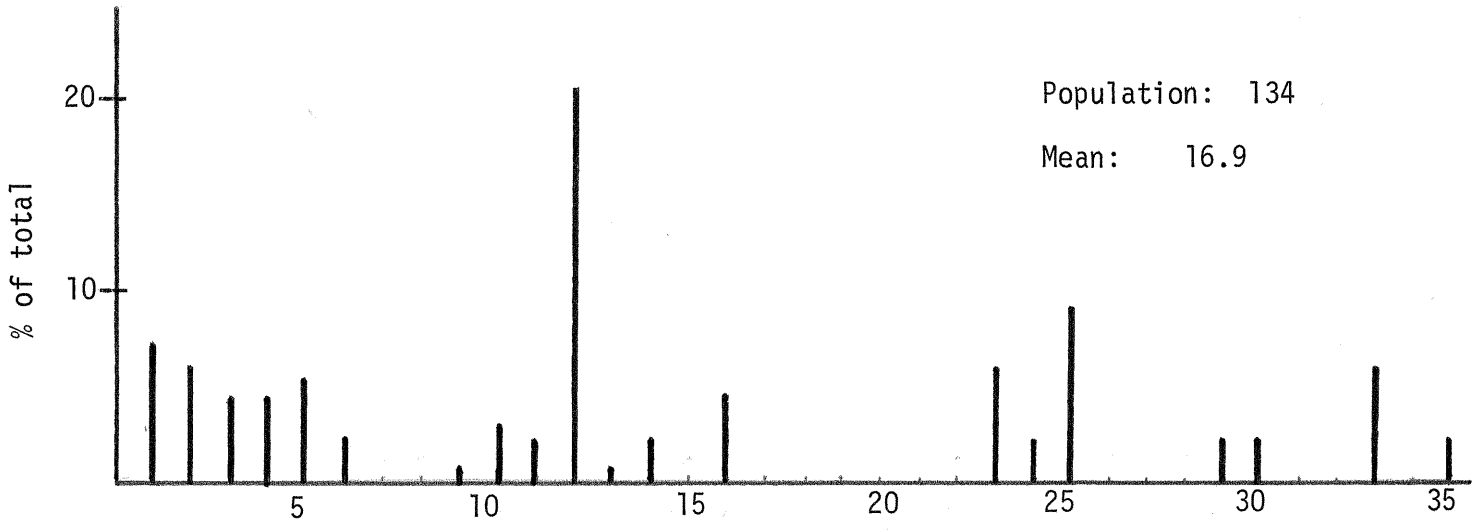
Figure 2



Population: 461
Mean: 5.1

CU cycles between instruction references

Figure 3



Population: 134
Mean: 16.9

CU cycles between data references

Figure 4

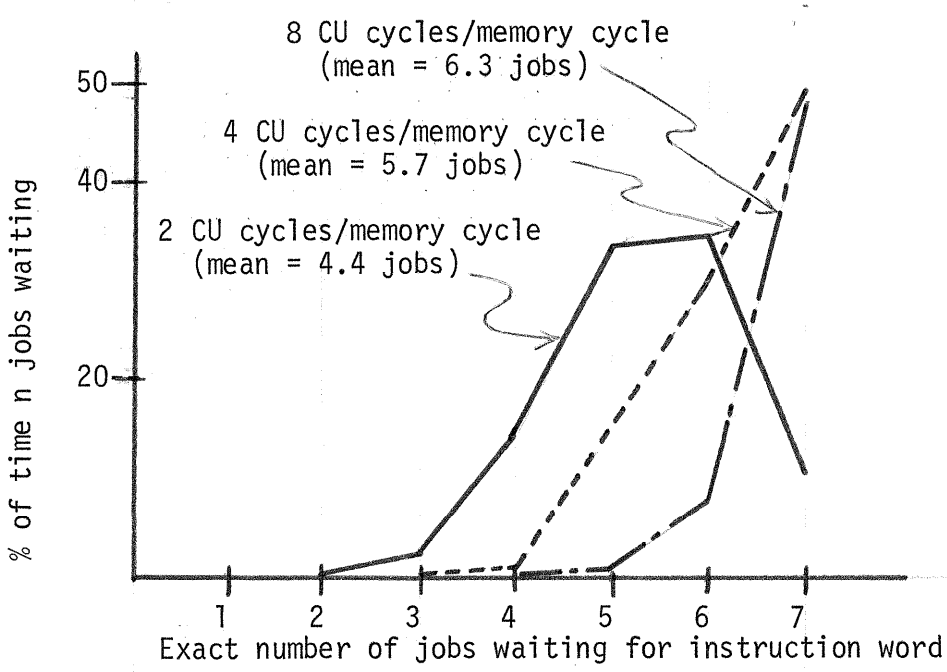


Figure 5

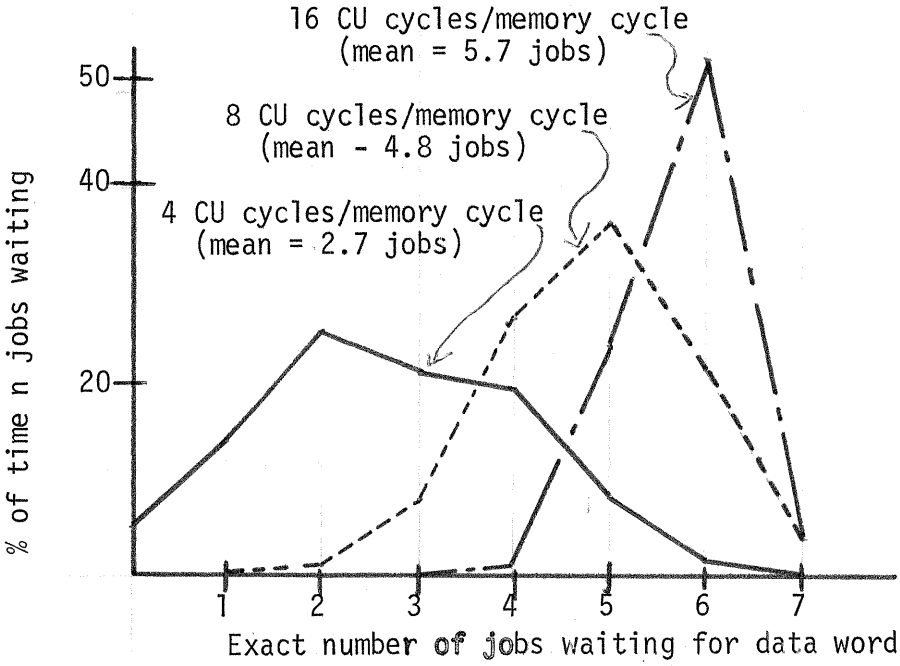


Figure 6