PERFORMANCE EVALUATION OF A PIPELINED VLSI ARCHITECTURE USING THE GRAPH MODEL OF BEHAVIOR (GMB)

by

J. T. Hsieh
A. R. Pleszkun
M. K. Vernon

Computer Sciences Technical Report #589

March 1985

Performance Evaluation of a Pipelined VLSI Architecture Using the Graph Model of Behavior (GMB)*

J. T. Hsieh
A. R. Pleszkun
M. K. Vernon

Computer Sciences Department
University of Wisconsin
Madison, WI 53706

Abstract

The PIPE project is a research project, at the University of Wisconsin-Madison, investigating high performance computer architectures appropriate for implementation with VLSI technology. The performance of the PIPE architecture has been evaluated previously using a detailed simulator written in Pascal. In this paper, we report the results of a simpler and more flexible approach. We have used a hardware description language, or more specifically a system modeling language known as the Graph Model of Behavior (GMB), to represent the PIPE architecture. A GMB simulator is used to derive performance estimates from the model, including queue length distributions, memory utilizations, and instruction issue rates. We have also represented "important" characteristics of program behavior probabilistically, rather than running actual benchmarks on our system model. Part of our study involved determining which program characteristics have a significant impact on performance and must be captured in the model. The probabilistic description necessarily omits specific program behavior which can affect system performance. However, our results show that the model is surprisingly accurate over a wide variety of program characteristics and system parameters. The probabilistic program representation also allows us to vary key program characteristics independently, providing performance new results that can be used to guide efforts to develop efficient compilers for PIPE. Based on the results of our experimental study, we conjecture that this simpler and more comprehensive approach is generally viable for initial studies of the performance characteristics of system architectures.

1. Introduction

The PIPE project is a research project, at the University of Wisconsin-Madison, investigating high performance computer architectures appropriate for implementation with VLSI technology. Principal features of the PIPE architecture include: (1) it is pipelined and supports programming constructs for efficient use of the pipeline, (2) it makes extensive use of architectural queues, (3) it is capable of a decoupled mode of data access/instruction execution, and (4) it has an instruction cache. The features of this architecture have been evaluated previously using a detailed simulator written in Pascal. The detailed simulator gathers performance statistics as it executes the assembly-language instructions of a benchmark program. Compiled and hand-coded versions of the first 12 Lawrence

* This research was supported by the National Science Foundation Grants, DCR-8202952 and DCR-8402680.

Livermore Loops [McMa72], as well as two programs that perform repeated procedure calls, were run on the detailed simulator to evaluate system performance. System parameters were varied extensively to study the effects of cache size, memory speed, queue sizes, and other parameters on the instruction issue rate. Results of those studies are reported in [HsPG84, SPKG83, YoGo84].

In this paper, we present a different approach to evaluating the PIPE architecture. First, we have used a hardware description language, or more specifically a system modeling language known as the Graph Model of Behavior (GMB), to represent the PIPE architecture. A GMB simulator is used to derive performance estimates from the model, including queue length distributions, memory utilizations, and instruction issue rates. Second, we have represented "important" characteristics of the benchmark programs probabilistically, rather than running the actual code on our model. Part of our study involved determining which program characteristics have a significant impact on performance and must be captured in the model. The model is validated by determining the program characteristics of the Loop benchmarks, and comparing model estimates with the detailed simulator results.

The graph model is simpler to develop and easier to comprehend than the detailed simulator. One measure of the reduced complexity is the "size" of the representations. The GMB model contains roughly 200 lines in its textual form, as compared with approximately 2000 lines of Pascal code for the detailed simulator. The probabilistic description necessarily omits specific program behavior which can affect system performance. However, our results show that the model is surprisingly accurate over a wide variety of program characteristics and system parameters. Furthermore, the probabilistic program representation allows us to vary key program characteristics independently, providing performance results that are not easily obtained using the detailed simulator. The results of these new experiments can be used to guide efforts to develop efficient compilers for PIPE. Based on the results of our experimental study, we conjecture that this simpler and more comprehensive approach is generally viable for initial studies of the performance characteristics of system architectures.

Section 2 of this paper describes the PIPE architecture in more detail, with an emphasis on the high-performance features which are evaluated in our study. The GMB models of the PIPE processor and memory subsystems, including the probabilistic representation of program characteristics, are described in Section 3. Section 4 reports the results of validating the GMB model, comparing model estimates with results from the detailed simulator for the benchmark programs. Characteristic probability distributions of the Loop benchmarks are summarized in that section. Section 5 then presents the results of our experiments on the effects of various program characteristics on PIPE performance, and Section 6 contains the conclusions of our study.

2. The PIPE Architecture

The PIPE architecture takes advantage of the concepts of decoupled access/execute architectures [CoSt81, Ples82, Smit82] to achieve its performance. In its original design, a PIPE system uses two processors to operate on a single task. This is the so-called decoupled mode of operation. One processor performs computations that calculate the addresses of data references and make the memory requests for the data items. The other processor uses the data operands requested by the first processor to perform the algorithmic computations of a program. For example, in a program that multiplies two matrices, the first processor will generate the addresses and requests for the elements of the matrix, while the second processor actually performs the multiplication. As specified in the PIPE architecture, the two processors in such a system were designed as be identical. Furthermore, each processor, by itself, can operate as a complete computer. Although the performance of a single processor will not be as great as that of a pair of processors, the single processor has features that make it a processor of significant power. It is the single-processor mode of operation that is studied in this paper.

The instruction issue and execution phases are pipelined in the PIPE processor. There have been other proposed single chip pipelined processors (MIPS for one [HJBG82]). However, the PIPE processor is unique in that interlocks are handled in hardware. The instruction set has been designed to permit the resolution of pipeline interlocks at the instruction issue stage. This follows the

philosophy of the CRAY-1 [Russ78] architectures. Such an approach permits the use of relatively simple hardware that can be implemented with VLSI technology.

Another feature of the PIPE architecture is the use of three architectural queues: a load data queue (LDQ), a store data queue (SDQ), and a store address queue (SAQ). All communication with the memory is performed via these queues and load and store instructions. A load instruction retrieves an item from memory and places it at the tail of the LDQ. One of the general purpose registers is designated as the head of the LDQ. When that register is read, the top item is popped off the queue. A write to that register does not affect the LDQ but instead places a data item at the tail of the SDQ. A store instruction places a memory address at the tail of the SAQ. The address is paired with data in the SDQ, in FCFS order, and the data value is then stored in the specified memory location.

Architectural queues are necessary for proper decoupled operation when two processors are operating together. These queues also turn out to be useful in the context of a single chip computer. Data is efficiently referenced through the use of the architectural queues. A similar effect could be achieved by a large number of registers, however this can have a profound effect on the efficiency of instruction set coding and on the ease of generating code. Using the SDQ, a finite number of store instructions can be issued at a faster rate than data can be written in memory. Using the LDQ, load instructions can be issued several instructions before the data operands are used, such that the overhead of going to memory can be hidden by other operations. The number of instructions between a load instruction and the instruction which uses the data is called the "load distance".

The use of queues to buffer data items can be appreciated by considering programs that manipulate vectors or arrays. Such programs are typically composed of loops that access each element of an array during each pass through the loop. By using techniques such as software pipelining, requests for data items can be scheduled in such a way that the LDQ is always kept filled. In this case, the processor never has to wait on a memory request. The importance of the architectural queues can be appreciated if one considers the delays involved in sending a data item off-chip, the

limited processor memory bandwidth in a VLSI environment, and the urgency of a memory request in a conventional architecture.

It is well known that due to the relatively heavy load of an output pin, sending a data item off-chip requires a relatively long period of time. In conventional machines, caches are included to minimize the latency associated with a memory request. For a single chip processor, a general purpose cache is not a particularly attractive alternative due to bandwidth limitations and the difficulty of placing a sufficiently large cache on the same chip with the processor. In the PIPE processor, we have placed an instruction cache on the chip. The architectural queues provide buffer space for data. In contrast to a general purpose cache, a relatively small instruction cache offers a high hit rate.

A final feature of the PIPE architecture, related to efficient operation of the cache and the pipeline, is the prepare-to-branch instruction. This instruction could also be called a delayed branch. Branches have a major impact on a pipelined machine. Typically, a branch instruction must pass through the entire pipeline before the branch target can be determined and start executing. Without a delayed branch instruction, the target of the branch experiences the complete pipeline latency, plus a higher probability of delay due to instruction cache miss. A delayed branch has been proposed several times recently [HJBG82, Radi82]. In these versions, the delayed branch instruction executes one instruction following the branch before the change in control flow occurs. In the PIPE architecture, we have generalized this. After a prepare-to-branch instruction, between zero and seven instructions may be placed. The number, or count, of such instructions is specified by a field in the prepare-to-branch instruction. These instructions are executed regardless of the branch outcome. The increase in the number of instructions after the prepare-to-branch was motivated by a desire to always have an instruction available at the issue stage.

In summary, the high-performance features of the PIPE architecture include the capability to operate in a decoupled mode, pipelined instruction issue and execution phases, architectural queues for off-chip communications with memory, an instruction cache, and a generalized prepare-to-branch instruction. In addition to the instructions which use the special features described above, the PIPE

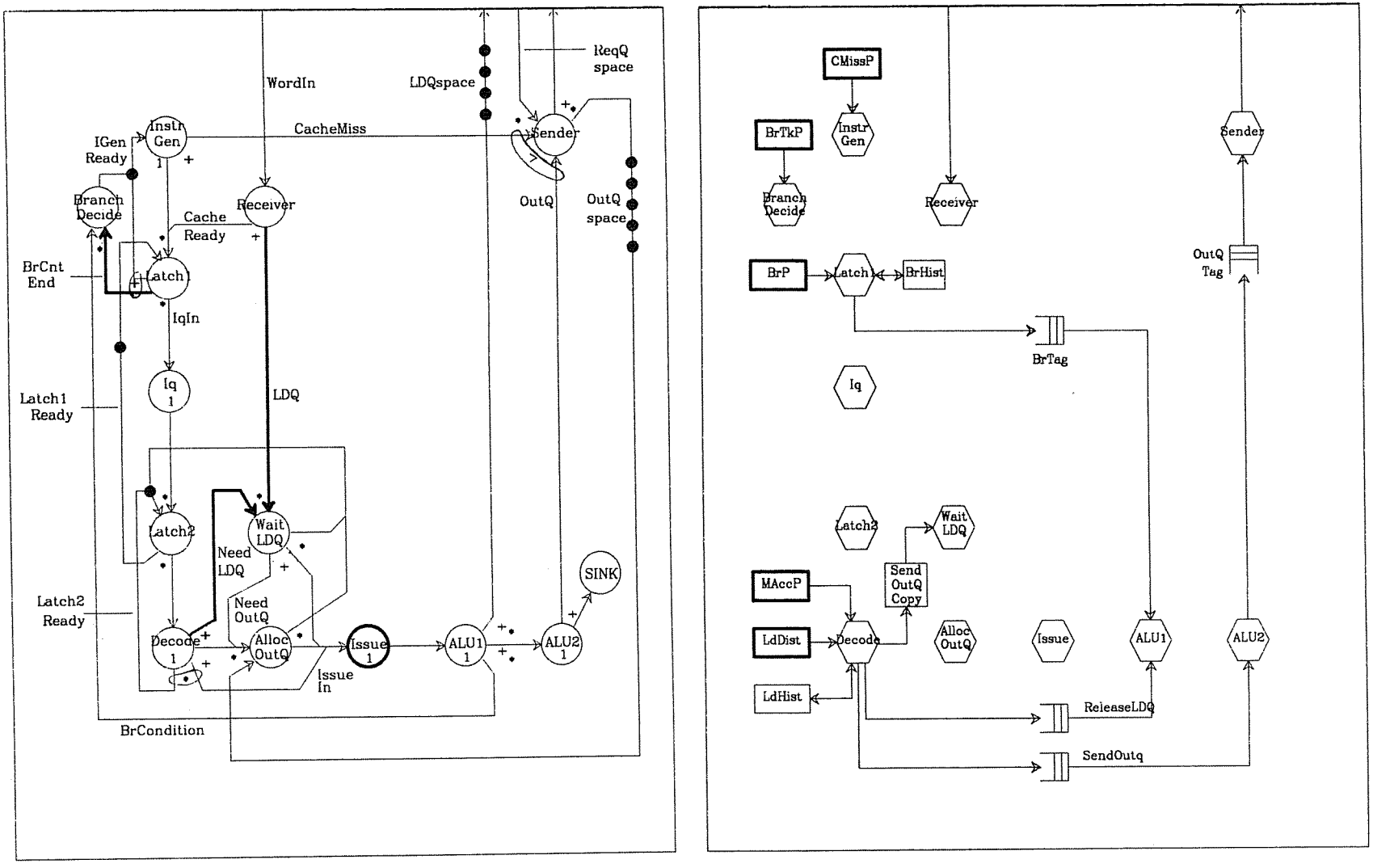
architecture has a fairly standard instruction set composed of arithmetic, logical, shift, and move instructions.

3. The Model

The GMB performance models which represent the behavior of the PIPE processor and memory subsystems, are shown in Figures 3.1 and 3.2. The two models connect via control and data arcs at their boundaries. Only the "control graph" and related "data graph" portions of the models are shown. A third part of the models, the "interpretation", defines the functional behavior of nodes in the graphs. Since most of the detailed functionality of the PIPE system is omitted in the GMB model, the interpretation for the model is very simple, and will be summarized briefly as we describe the models below. (The complete text of the GMB model appears in Appendix A.)

In Section 3.1 we discuss the dynamic behavior of the models, focussing primarily on the flow of control in the control graphs. These graphs contain "nodes" which represent processing activity, and "control arcs" which define the sequencing of node execution events. The control logic at the inputs and outputs of each node further characterizes the sequencing behavior by defining conditions before initiating and after terminating the execution of the node (* = "AND", + = "OR", and $+_*$ = "inclusive OR"). A control node is initiated when there are sufficient "tokens" on its input arc to satisfy its input logic. The control graph model for the PIPE processor (Figure 3.1a), contains initial tokens on *IGenReady*(1), *Latch1Ready*(1), *Latch2Ready*(1), *OutQspace*(k), and *LDQspace*(n). The dynamic behavior of the control graphs is then characterized by the flow of tokens among the nodes. One unit of time in the model represents one clock cycle in the PIPE processor. The nodes with a delay of one time unit have a "1" in the control graph in Figures 3.1a and 3.2a. All other nodes serve only to synchronize tokens, and thus have a delay of zero.

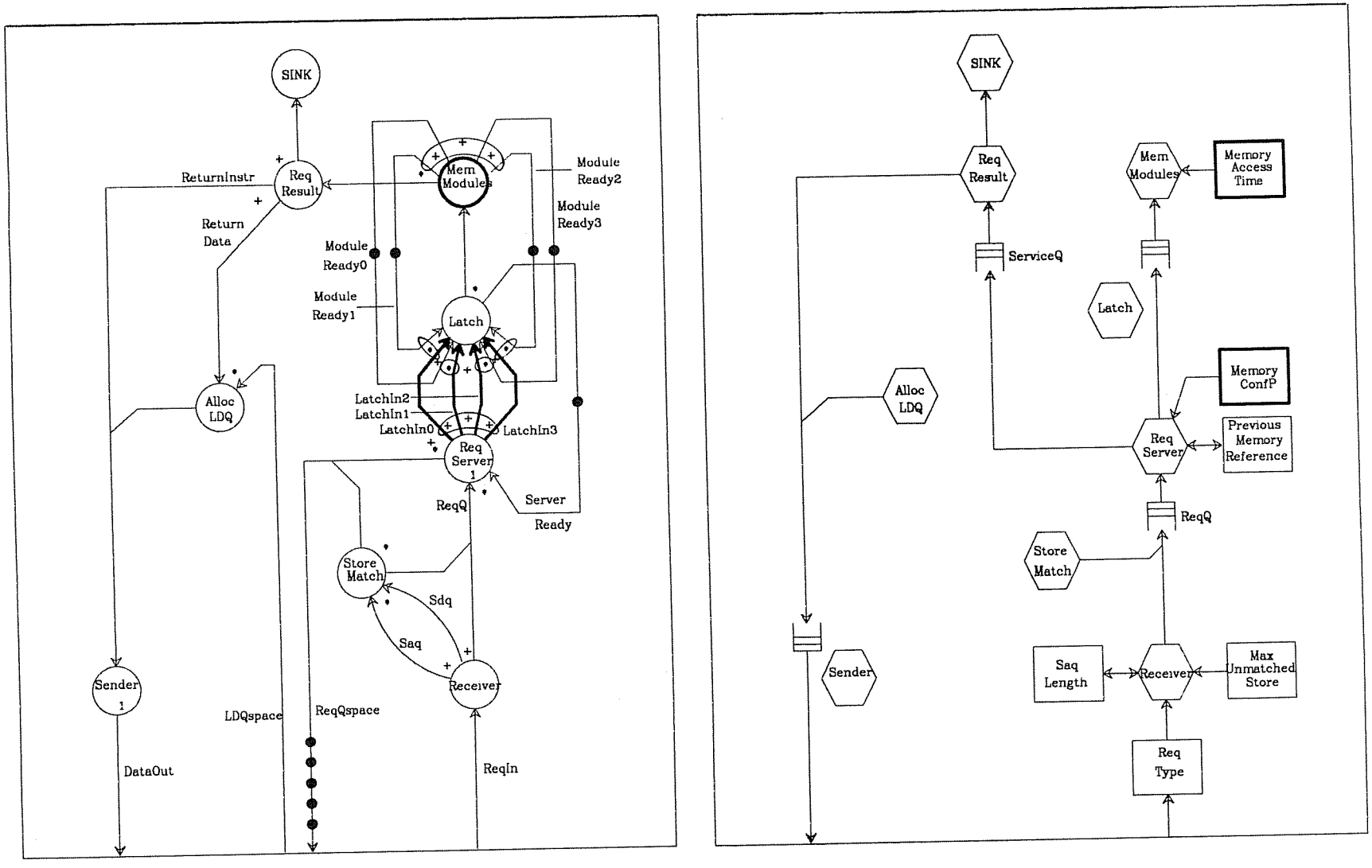
In Section 3.2 we discuss the parameters which characterize program behavior probabilistically in the model, and how these parameters are represented in the data graphs. The data graphs consist of "processors" (i.e. hexagons) which correspond one-to-one with the nodes in the control graph, "datasets" which represent data values, and "data arcs" which define the read/write dataset access



(a) Control Graph

(b) Data Graph

Figure 3.1: GMB Performance Model of the PIPE Processor



(a) Control Graph

(b) Data Graph

Figure 3.2: GMB Performance Model of the Memory Subsystem

capabilities for each node/processor pair. The read-only datasets which are highlighted in the PIPE model (e.g. *CMissP* in Figure 3.1b) contain input parameters for a simulation experiment. All other datasets are used to maintain state information necessary to model particular system behavior, or to pass such state information between two nodes. For example, *BrHist* in Figure 3.1a is read and written by *Latch1*, and is used to record information about the branch count for prepare-to-branch instructions. On the other hand, *BrTag* is written by Latch1 and read by *ALU1*, whereby *ALU1* can determine if the expression it is currently evaluating is the conditional of a prepare-to-branch instruction.

The GMB is defined and described more completely in [Estr78, RaVE79, Vern82, and VdSE83]. A reader who is not familiar with the GMB notation can follow the discussion below to gain further understanding of the PIPE architecture and the amount of detail represented in the model.

3.1. Processor and Memory Subsystems

The control graph model of the PIPE processor (Figure 3.1a) contains a node which sends requests to the memory subsystem (*Sender*), and a node which receives data from the memory subsystem (*Receiver*). All other nodes represent stages in the instruction fetch (*InstrGen, Latch1, Iq*), decode (*Latch2, Decode*), issue (*WaitLDQ, AllocateOutq, Issue*), and execute (*ALU1, ALU2*) pipeline. As long as there are no cache misses, no branch instructions, and no need for data from memory, tokens flow along this pipeline, (from *InstrGen to CacheReady, Latch1 to IGenReady*IqIn, Decoder to NeedOutq+IssueIn*), one at each stage in the pipe per clock cycle. The *Issue* node will be active during every time unit under these circumstances, and the measured fraction of time that *Issue* is busy, which corresponds to the "instruction issue rate", will be 1.0.

InstrGen represents a test for whether the next instruction to be issued is in the cache. The unit delay for this node models the time to copy the instruction from the cache into the instruction queue. The occurrence of a cache miss is determined probabilistically from specified input parameters, as described in Section 3.2. In the case of a cache miss, *InstrGen* will output a token on *CacheMiss*,

which will cause *Sender* to send a "request" token to the memory subsystem. The flow of tokens in the execution pipeline is temporarily interrupted[1] until a token representing the transferred instruction word from memory arrives on *WordIn*. If the incoming data is a response to a request for the instruction cache, *Receiver* places a token on *CacheReady*, and the execution pipeline resumes operation.

Latch1 models the effects of branch instructions on the execution pipeline. If the current instruction is a "prepare-to-branch" instruction, then *Latch1* sets the branch count (dataset *BrHist* in Figure 3.1b), and decrements this count during subsequent activations. The prepare-to-branch event and branch count are again determined probabilistically (see Section 3.2). A prepare-to-branch instruction continues through the pipeline. When it completes the *ALU1* stage, a token is output on *BrCondition*, indicating that the conditional expression which determines whether the branch will be taken, has been evaluated. When the branch count is exhausted, *Latch1* places a token on *BrCntEnd* instead of on *IGenReady*. If the branch count is exhausted before the prepare-to-branch completes the *ALU1* stage, then the token on *BrCntEnd* will wait for the token to appear on *BrCondition*, temporarily interrupting the flow in the execution pipeline, and reducing the instruction issue rate. The waiting time distribution for tokens on *BrCntEnd* is a performance measure of interest in the model.

The remaining behavior to be described for the execution pipeline, is the effect of "Load" and "Store" instructions, and the use of data from memory. The load and store instructions are "recognized" (i.e. determined probabilistically) by *Decode*, and routed to *AllocateOutq*. These instructions will not be issued (i.e. *AllocateOutq* will not be initiated) until there is space in the "output queue" for the memory request. The output queue is not visible to the programmer, but is necessarily limited in size and can be full if memory accesses are backlogged. *ALU2* outputs a token on *OutQ* for load or store instructions, and otherwise outputs a token to *SINK*. Requests generated by load instructions will cause the memory subsystem model to return a token to *LDQ*.

[1] When the pipeline is interrupted, *Issue* will be idle, and its measured utilization will be less than 1.

A token which represents an instruction that requires operands from the LDQ is routed by *Decode* to *NeedLDQ*. If the data (requested by a previous load instruction) has not yet been retrieved from memory, the token will wait on *NeedLDQ*, thus reducing the instruction issue rate.

To summarize the processor system model, there are three significant events which can interrupt the steady flow of tokens ("instructions") through the execution pipeline: (1) instruction cache misses (*CacheMiss*), (2) branch count completion (*BrCntEnd*), and (3) the need for data operands from memory (*NeedLDQ*). The control points (i.e. the "+" output logic expressions) for these events are highlighted in Figure 3.1a. Interruption of the pipeline causes the instruction issue rate to be less than 1. The *Issue* node is highlighted in the model because estimation of the fraction of time it is busy, which can be requested in the GMB Simulator, corresponds to the "instruction issue rate" system performance measure. Similarly, the waiting time distribution for tokens on highlighted arcs *BrCntEnd* and *NeedLDQ*, and the queue length distribution on the highlighted arc *LDQ*, are interesting performance measures that the Simulator can provide.

In the memory subsystem model (Figure 3.2), "store data" requests (i.e. tokens) are paired with "store address" requests (*StoreMatch*), and these paired requests, as well as all load requests, are routed to one of four memory modules (probabilistically) by *RequestServer*. A request token routed to memory module *i* will wait on arc *LatchIn*[*i*] if the requested module is busy, thus blocking other memory requests which may be pending. *MemModules* represents all (four) modules in the memory subsystem, and can be active a total of four times simultaneously. The *MemoryAccessTime* (i.e. the delay for the *MemModules* node), measured in CPU clock cycles, is a parameter of the model, and is thus represented as a dataset in Figure 3.2b. Upon completion of a memory access which is a response to an instruction cache request or a load data request, *ReqResult* will output a token on *ReturnInstr* or *ReturnData*, respectively. Otherwise (store requests), *ReqResult* outputs a token to *SINK*.

The utilization of node *MemModules*, and the queue length and waiting time distributions on arcs *LatchIn[i]*, highlighted in the memory subsystem control graphs, are again interesting perfor-

mance estimates that can be requested at the start of a model simulation run. This completes a brief summary of the PIPE system operation as represented in the GMB models.

3.2. Program Characteristics

There are a large number of program characteristics, and a variety of ways in which these properties might be represented in the model. For example, we might store a benchmark assembly language program in a dataset, and process each statement at each node in the pipeline in turn, as is done in the detailed simulator. The reader should note that in this case we would need to add datasets for program variables and to pass specific instructions between each processor in the pipeline. On the other hand, we are only interested in the characteristics which have a primary effect on system performance (e.g. instruction issue rate). Given such a set of characteristics, we might again take a specific assembly language program, convert all instructions which do not have these "important" properties to "no-op" instructions, and then run the program on the model. This would reduce the complexity of the Decode and ALU pipeline stages, but would still leave a considerable amount of detail in the model.

In the interest of creating a significantly simpler and more flexible model, we have pursued a different approach [Vern82]. Instruction characteristics are determined by specified probability distributions in our model. Furthermore, the characteristics are determined at the pipeline stages where the information is pertinent, as explained later in this section. The probability distributions may be determined by a statistical analysis of program behavior and/or by the desire to see how the system will perform for particular parameter values. In Section 4 we discuss the results of experiments in which the probabilities are set according to values determined from statistical analysis of the Loop benchmarks. Based on our initial validation experiments, the following program characteristics have an important effect on performance:

(1) The probability that an instruction is in the cache ("cache miss rate").

(2) The probability that an instruction is a "prepare-to-branch" instruction.

(3) The branch count distribution (i.e. the distribution of the number of instructions between a prepare-to-branch instruction and the branch decision point).

(4) The probability that a branch condition evaluates to "true". (Note: the probability of finding the branch target in the cache may be different than the probability of finding other instructions.)

(5) The probability that an instruction is a "load" instruction. Furthermore, this probability is broken down into two cases, based on whether the immediately preceding instruction is a load instruction or not.

(6) The probability that an instruction is a "store" instruction.

(7) The load distance distribution (i.e. the number of instructions between a load instruction and the instruction which uses the data as an operand).

(8) For both load and store instructions. the first-order memory conflict probability (i.e. the probability that a memory request accesses the same memory module as the previous load or store instruction).

Initially, branch count distributions and load distance distributions were assumed to be uniform, and only the endpoints were specified in the model. Memory module accesses were also assumed to be uniform. However, comparison of initial GMB performance estimates with detailed simulator results indicated that distributions that more accurately reflected program behavior were needed. Similarly, the conditional probabilities of load instructions, which model sequences of load instructions more accurately, were found to be important during initial validation studies. Conversely, the branch outcome probability was not used in our GMB model to determine the probability of cache miss, since results agree well with the detailed simulator without this factor.

The datasets which store the probability distributions are highlighted in the data graph models (Figures 3.1(b) and 3.2(b)). For example. the probability that an instruction is a conditional "prepare-to-branch" instruction. and the branch count probability distribution, are stored in the dataset *BrP*. *Latch1* reads this dataset and uses the values to determine whether the current instruction is a conditional prepare-to-branch. and to select a branch count. The cache miss probability (*CMissP*) is similarly represented and used, as are the load and store instruction probabilities (*MemAccP*), load distance distribution (*LdDist*), branch conditional result probability (*BrTakenP*), and first-order memory conflict probability (*MemConfP*).

The values of the datasets are set at the start of a simulation run. Upon request, the GMB Simulator will calculate 1) utilization estimates for any control node, and 2) queue length and waiting time distributions for any control arc in the model. In the next section we discuss validation of the GMB model, in which the Livermore Loops were statistically analyzed and used to parameterize the model. Performance estimates are compared with the measures reported by the detailed simulator. In section 5, some of the probabilistic program parameters are varied independently, to explore their impact on performance.

4. Model Validation Experiments

The first 12 Lawrence Livermore Loops, and two additional programs ("Ackerman" and "sieve"), have been written in Pascal and compiled for the PIPE machine. The Loop benchmarks have also been hand-coded in PIPE assembly language to optimize for larger branch counts, longer load distances, and smaller program size. We used the detailed simulator to analyze the dynamic behavior of these benchmarks, to determine the characteristic parameters needed in the GMB model.

The parameters for the compiled and hand-coded benchmark characteristics are summarized in Tables 4.1 and 4.2, respectively.[2] The details of the branch count and load distance distributions are not given in the tables due to space constraints. (The detailed distributions are given in Appendix 2.) The tables show the range of program characteristics which are captured by the benchmarks. For example, the compiled benchmarks contain between 10% and 33% load instructions, and the mean load distances in the hand-coded benchmarks range from 2.0 to 16.2. The tables also show that branch counts and load distances are generally longer in the hand-coded benchmarks. The smaller program sizes for the hand-coded programs are reflected in the higher probability of conditional prepare-to-branch instructions (i.e "% branch"). More precisely, the hand-coded programs have fewer instructions in the loop(s), resulting in a higher ratio of branch to non-branch instructions.

[2] Second order memory conflict probabilities are shown in the tables, but were not used in our model. Conditional probabilities for the load instruction are derived from the mean load sequence lengths and the total percentage of load instructions.

Table 4.1: Characteristics of the Compiled Benchmarks

program	% load	% store	% cache misses	% branch instr	memory conflict		branch count		load distance		load sequence	
					1	2	range	mean	range	mean	range	mean
LLL1	22.72	13.64	0.12	2.28	0.3084	0.2321	2-3	2.0	0-2	1.5	1-3	1.7
LLL2	28.43	9.18	0.21	1.84	0.5017	0.1318	0-2	1.0	0-5	1.9	1-3	1.5
LLL3	16.66	11.11	0.06	5.56	0.4959	0.0498	2-3	2.0	1-2	1.3	1-2	1.5
LLL4	27.46	10.00	0.16	5.02	0.3401	0.1857	0-2	1.0	0-5	2.5	1-2	1.2
LLL5	27.36	7.37	0.13	2.11	0.5527	0.1574	0-2	1.0	0-5	3.0	1-3	1.4
LLL6	26.73	7.92	0.13	1.98	0.5761	0.1212	0-2	1.0	0-5	2.8	1-3	1.3
LLL7	26.88	14.28	0.35	0.85	0.1104	0.3564	1-2	2.0	0-3	1.7	1-3	1.9
LLL8	33.16	2.86	1.08	0.32	0.1573	0.1668	0-3	2.8	0-11	6.6	1-7	2.4
LLL9	30.76	20.50	0.44	0.65	0.3392	0.1209	1-2	2.0	0-6	1.9	1-4	1.5
LLL10	17.28	12.34	0.41	0.62	0.2297	0.2137	1-2	2.0	2-9	3.1	1-2	1.5
LLL11	10.00	10.00	0.06	5.00	0.3708	0.2475	1-1	1.0	1-1	1.0	2-2	2.0
LLL12	10.00	10.00	0.06	5.00	0.3713	0.0619	1-1	1.0	1-1	1.0	2-2	2.0
Ackerman	18.87	18.34	2.69	0.0496	0.0767	0.1985	0-5	1.0	0-5	2.8	1-6	1.7
sieve	20.86	13.44	0.63	11.53	0.2238	0.2711	0-3	0.6	0-3	1.1	1-2	1.3

Table 4.2: Characteristics of the Hand-Coded Benchmarks

program	% load	% store	% cache misses	% branch instr	memory conflict		branch count		load distance		load sequence	
					1	2	range	mean	range	mean	range	mean
LLL1	29.98	10.00	0.20	9.98	0.2446	0.0031	5-5	5.0	5-6	5.3	2-3	3.0
LLL2	24.98	2.50	0.09	14.99	0.4502	0.00	4-4	4.0	3-4	3.5	2-2	2.0
LLL3	33.30	0.02	0.08	16.65	0.4953	0.0005	4-4	4.0	1-3	2.0	1-1	1.0
LLL4	27.18	9.09	0.20	9.09	0.4497	0.0818	4-4	4.0	2-8	7.3	1-3	3.0
LLL5	33.32	14.28	0.14	4.76	0.1979	0.1988	3-3	3.0	5-7	6.0	7-7	7.0
LLL6	30.42	13.04	0.13	4.35	0.3958	0.0009	3-3	3.0	5-7	6.0	7-7	7.0
LLL7	29.99	3.33	0.33	3.33	0.1936	0.0984	7-7	7.0	4-16	11.7	2-9	8.9
LLL8	36.04	7.21	1.02	1.26	0.3718	0.0945	1-4	3.9	7-12	9.1	6-8	7.5
LLL9	38.45	2.56	0.49	2.56	0.4443	0.2293	6-6	6.0	4-11	8.5	1-7	5.0
LLL10	24.37	24.37	0.41	2.44	0.5401	0.3791	4-4	4.0	9-25	16.2	10-10	10.0
LLL11	33.32	16.66	0.07	16.66	0.00	0.6631	3-3	3.0	3-4	3.5	2-2	2.0
LLL12	33.32	16.66	0.07	16.66	0.3316	0.0007	3-3	3.0	3-4	3.5	2-2	2.0

4.1. Benchmark Validations

One measure of PIPE system performance is the instruction issue rate. The issue rate indicates how close we have come to our goal of issuing once instruction per clock. An issue rate of less than one occurs due to memory delays, inter-instruction dependencies, and branches. The issue rate

can be misleading because it can vary inversely with the efficiency of the code that is executing. For example, in the extreme case, we can insert "no-op" instructions to fill in for issue delays and bring the issue rate up to one. However, given a particular benchmark or set of program characteristics, the issue rate corresponds directly to program execution time. We use the issue rate as our primary performance measure, but consider the effects of code efficiency when interpreting the results.

Each set of benchmark characteristics (26 in all), were used to parameterize the GMB model described in section 3. Estimates of instruction issue rate of PIPE produced by the GMB Simulator for each set of model parameters, are compared with the measured issue rate in the detailed simulator to determine the accuracy of the probabilistic GMB model. The GMB simulations were run for 3000 clock cycles. Due to the exploratory nature of these experiments, confidence intervals were not calculated for the performance estimates. It was observed, however, that the estimates were consistent at 1000, 2000, and 3000 clock cycles in each of the runs. The models were run with a memory access time of 4 clock cycles, and a (very fast) memory access time of 1 clock cycle. The results of the 52 model validation runs are shown in Figures 4.1 and 4.2.

Although the GMB model omits a considerable amount of detail, we find that the model estimates are in good agreement with the detailed simulations for the majority of the experiments. For the compiled benchmark characteristics, 75% of the model estimates have relative errors less than 5%. Except for LL9, which has a relative error of 16.2% when the memory access time is 4, the remaining estimates for the compiled benchmarks have relative errors of less than 10%. The results are similar for the hand-coded benchmarks, although discrepancies are somewhat larger for the slower memory speed. Five of these estimates are above 10% relative error. Of these estimates, three are within 20% relative to the detailed values, and the largest relative error is 27.7% (or absolute error of 12.7%). We conclude that there are specific program dependencies, such as complex relationships between branch count delays and memory access delays, which occur in some programs but cannot be represented easily in the probabilistic GMB model. On the other hand, the GMB is qualitatively correct in all of the benchmarks estimates, and yields good initial quantitative

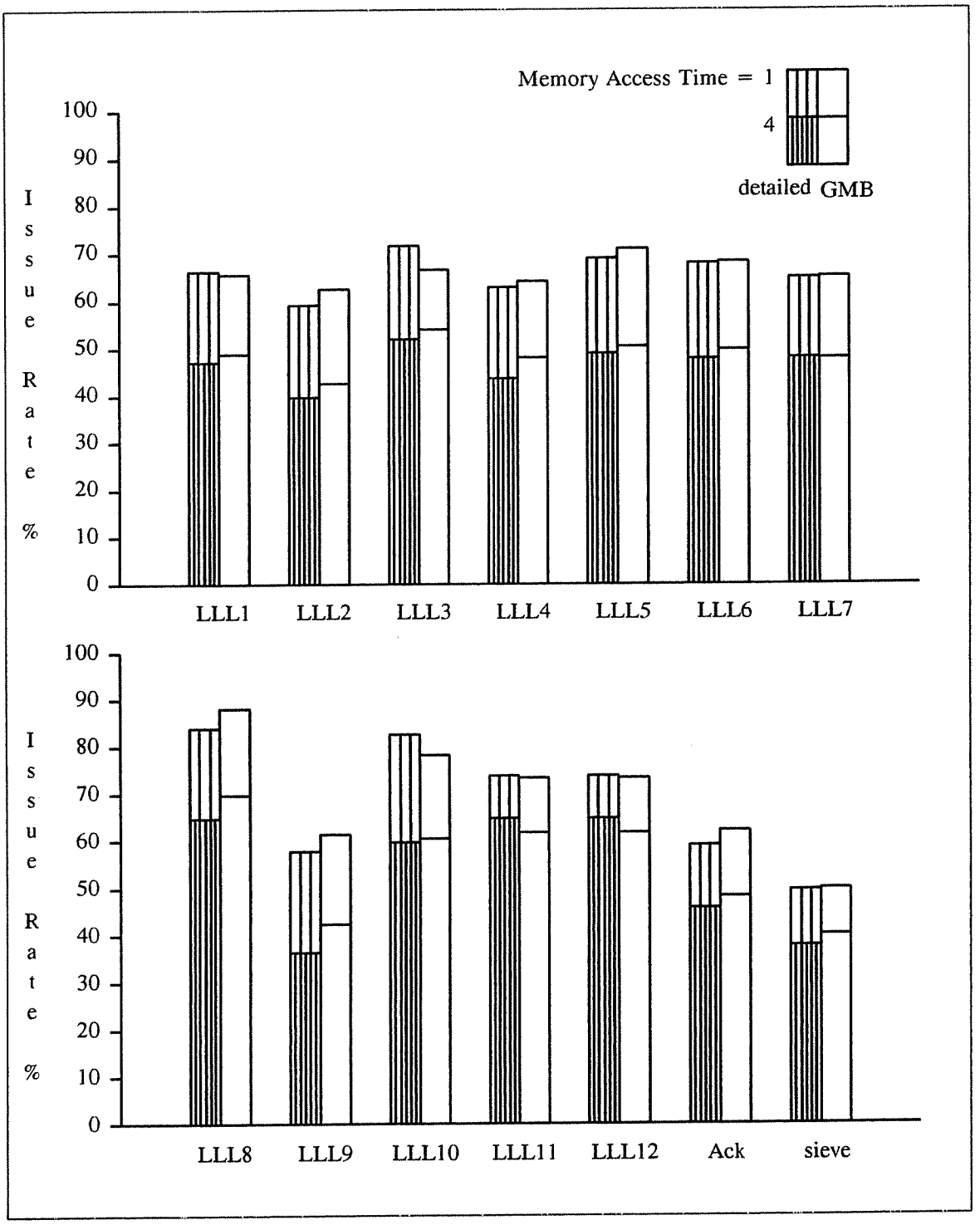


Figure 4.1: Validation Results for Compiled Benchmarks

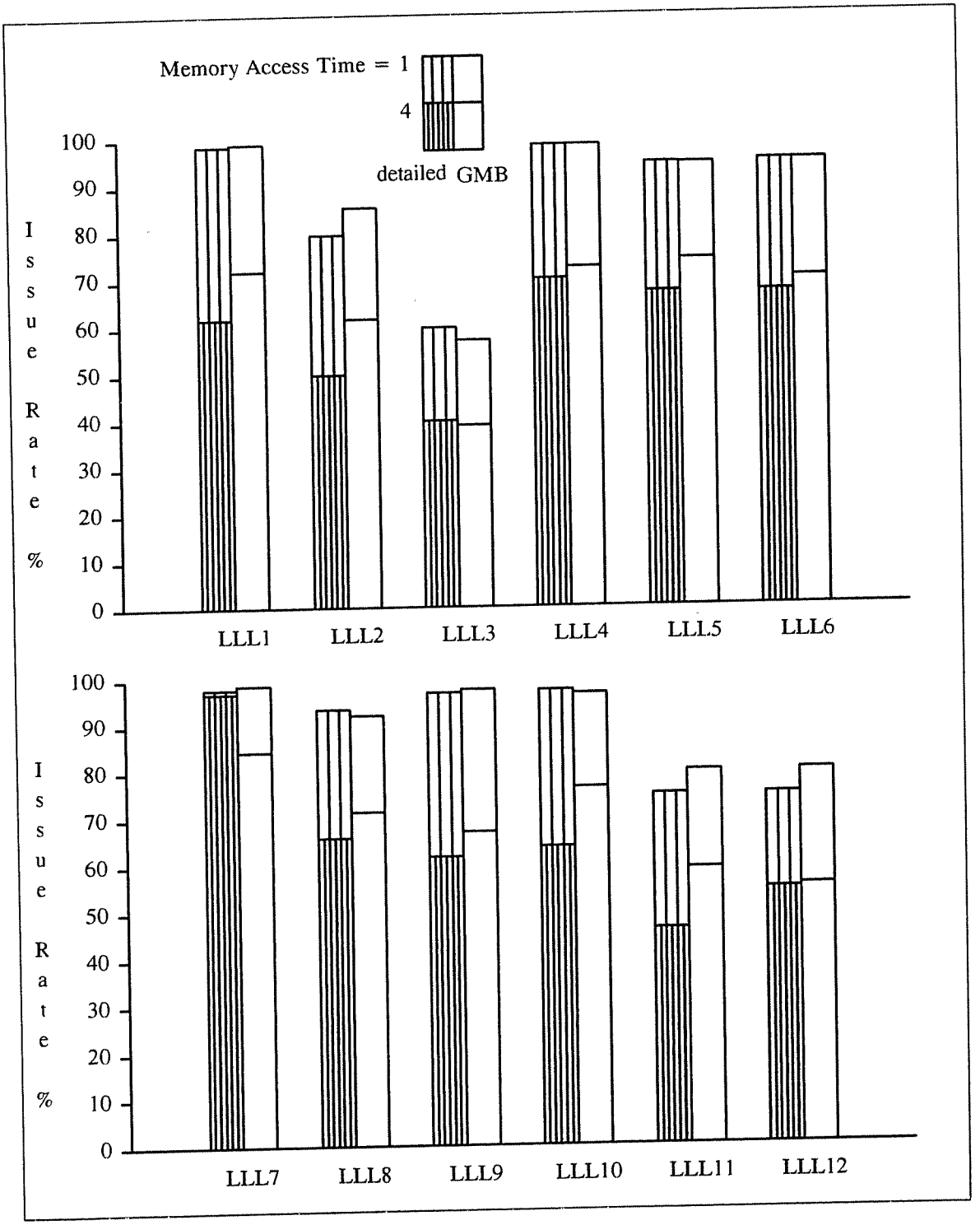


Figure 4.2: Validation Results for Hand-Coded Benchmarks

estimates in the vast majority of the cases we examined.

4.2. Memory Access Time Experiment

Both the GMB model and the detailed simulator can be used to study the effects of PIPE system parameters on performance. For example, the effect of memory access time on the issue rate of a benchmark can be studied. Although the architectural queues in PIPE support code scheduling of memory requests, memory latency will not be completely eliminated. As the memory access time becomes large, code scheduling will be less effective. In addition, data accesses compete with instruction accesses on a cache miss. A longer memory access time will magnify the effects of cache misses. We ran a second set of validation experiments to compare the performance estimates of the model and the detailed simulator for various memory access times.

The experiments were run using the compiled version of Lawrence Livermore Loop 1, and the hand-coded version of Loop 6. These loops were selected because they are somewhat representative of the the two classes of programs that were used in the model validation experiments. Memory access times were varied from 1 clock period to 16 clock periods. The numerical results of these simulations are given in Table 4.3, whereas the results are plotted in Figures 4.3a and b. Again, the results of the GMB model are in close agreement with the detailed simulator. In Figure 4.3(a) a log scale is used for the memory access times. The results indicate that the issue rate of the compiled benchmark is less sensitive to the memory access time than the issue rate of the hand-coded bench-

Table 4.3: Issue Rate vs. Memory Access Time
for Selected Lawrence Livermore Loop Benchmarks

Memory Access Time (Clock Periods)	Loop 1 (Pascal Compiler)		Loop 6 (Hand-Compiled)	
	Detailed	GMB	Detailed	GMB
1	66.37%	65.70%	95.00%	94.97%
2	58.81%	58.43%	87.69%	89.17%
3	52.32%	54.30%	76.04%	78.67%
4	47.12%	48.83%	67.12%	70.13%
6	39.31%	40.37%	53.11%	58.20%
8	33.21%	35.46%	43.12%	46.55%
12	25.22%	27.24%	31.32%	35.23%
16	20.22%	21.33%	24.58%	25.45%

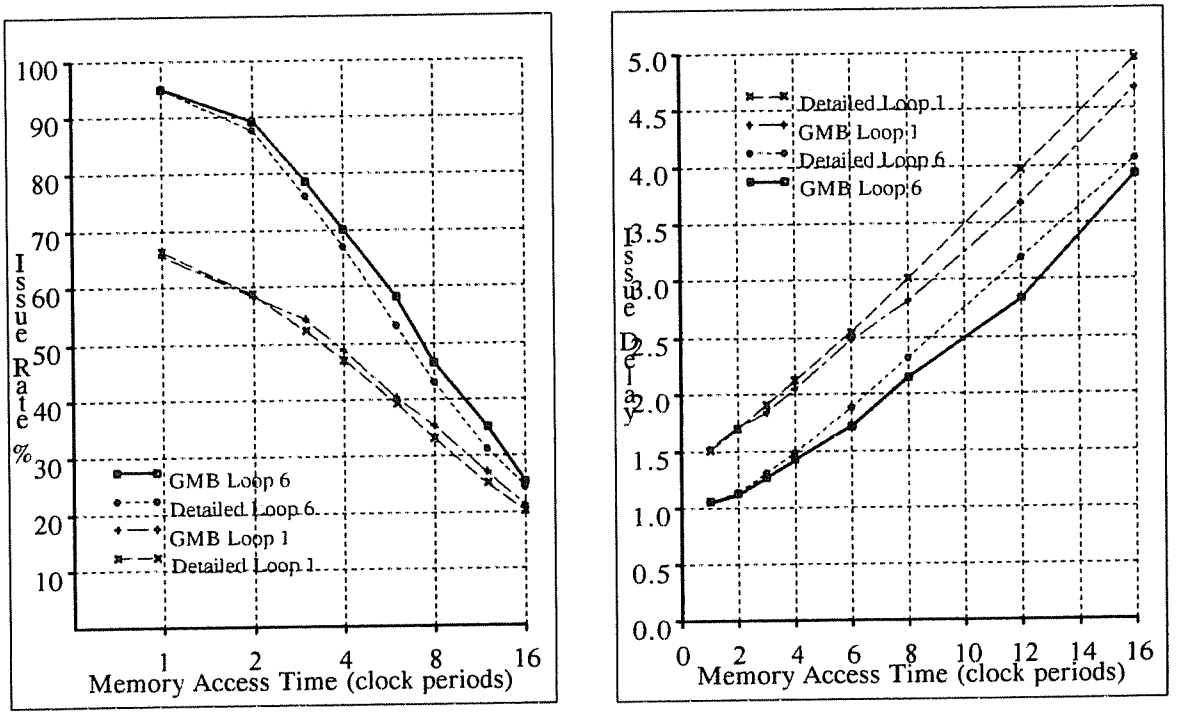


Figure 4.3: (a) Issue Rate and (b) Issue Delay vs. Memory Access Time for Selected Lawrence Livermore Loop Benchmarks

mark. When the memory access time has been increased by a factor of 16, the issue rate of the compiled benchmark has decreased by a factor of 3, whereas the rate for the hand-compiled code decreases roughly by a factor of 4. The greater sensitivity to memory access time for hand-coded programs may be explained by the relatively large percentage of the load and store instructions. Because the code is more efficient, there are fewer instructions that need to be executed for each load or store. Thus there are fewer instructions to "hide" the effects of memory accesses. Figure 4.3(b) shows the issue delay, which is the inverse of the issue rate, as a function of the memory access time, using a linear scale. It is interesting to note that the GMB curves of Figure 4.3(b) suggest that the issue delay is approximately a linear function of memory access time for access times greater than 6.

4.3. Summary

The agreement between GMB model estimates and detailed simulation results for experiments described in this section is encouraging. We have some confidence that further use of the GMB model will give an accurate indication of PIPE system performance under a wide variety of model parameters. In the next section, we report the results of evaluations made by the GMB simulator which are not easily obtained using the detailed simulator.

5. PIPE Performance Evaluation

The performance results in Figures 4.1 and 4.2 illustrate the increased instruction issue rates for the hand-coded benchmarks as compared with the compiled benchmarks. An interesting question is: which of the optimizations in the hand-coded programs have the most impact on instruction issue rate? A related issue is where should major effort be placed in developing a compiler for the PIPE architecture. It is difficult to study the effects of program characteristics using the trace-driven simulator, since this would require that the programs driving the simulation have very specific characteristics. The probabilistic nature of the GMB model has enabled us to modify these parameters very easily. The results for some initial experiments are reported below.

5.1. Typical PIPE Programs

We have two options when studying the impact of a particular program characteristic on system performance. We can set all other parameters to values which optimize performance (e.g. low branch probabilities, long load distances), and thus obtain estimates of the impact of the characteristic of interest in a controlled setting. Alternatively, we can set all other parameters according to some "typical" values, to study the impact of the parameter of interest in a "realistic" environment. The second environment includes complex interactions between various parameters which are not well understood. Both types of experiments are equally easy to implement in the model.

For the experiments reported in this section, we used the approach of defining "typical" program characteristics. Two typical sets of program characteristics were developed: one for programs

compiled by the Pascal compiler and another for hand-coded programs. These characteristics were constructed from an examination of the benchmark program characteristics (Tables 4.1 and 4.2, and Appendix 2). The features of our typical programs are summarized in Table 5.1.

Table 5.1: "Typical" Program Parameters

program type	% load	% store	% cache misses	% branch instr	memory conflict	branch count		load distance		load sequence	
					1	range	mean	range	mean	range	mean
Pascal Compiled	20.0	10.0	1.0	5.0	0.35	0-3	2.0	0-5	2.1	1-2	1.5
Hand-Compiled	30.0	12.0	0.5	10.0	0.35	4-6	5.0	5-7	6.0	3-3	3.0

The GMB model was run using these parameters. The resulting issue rate for the Pascal compiled parameters is 52.27%, while the issue rate for the hand-coded parameters is 68.57%. These results can be used as benchmarks against which further simulations may be compared.

5.2. Cache Hit Rate

The first set of experiments that we performed involved an evaluation of the effectiveness of the cache. One of the important features of the PIPE implementation is the inclusion of an on-chip instruction cache. A large instruction cache can minimize one of the impacts of inefficient code generation by a compiler. The instruction cache must, however, share chip real estate with the processor. Increasing the cache miss rate parameter in the model, will give an indication of how important it is to spend time designing a dense cache and/or building an optimizing compiler to produce compact code. If a higher miss rate does not result in a significant reduction in system performance, then it is not necessary to expend a great amount of effort on these activities. Design time could perhaps be more effectively spent on other parts of the implementation.

Using parameters for a typical hand-coded program, the cache miss rate was varied between 0% and 20%. The experiment was repeated for memory access times of 4 and 8 clock cycles, to study the impact of cache miss rate for different memory speeds. The resulting PIPE performance

estimates are shown in Figure 5.1.

A 0% cache miss rate indicates that the entire program is initially loaded and fits in the instruction cache. The issue rate for this cache miss rate reflects the blocking that occurs due to memory access for data items. Our first observation is that increasing cache miss rates to 2%, 5%, and 10%, substantially decreases system performance for both memory speeds. The relative effect of the cache miss rate is approximately the same for the two memory access times for the parameters examined.

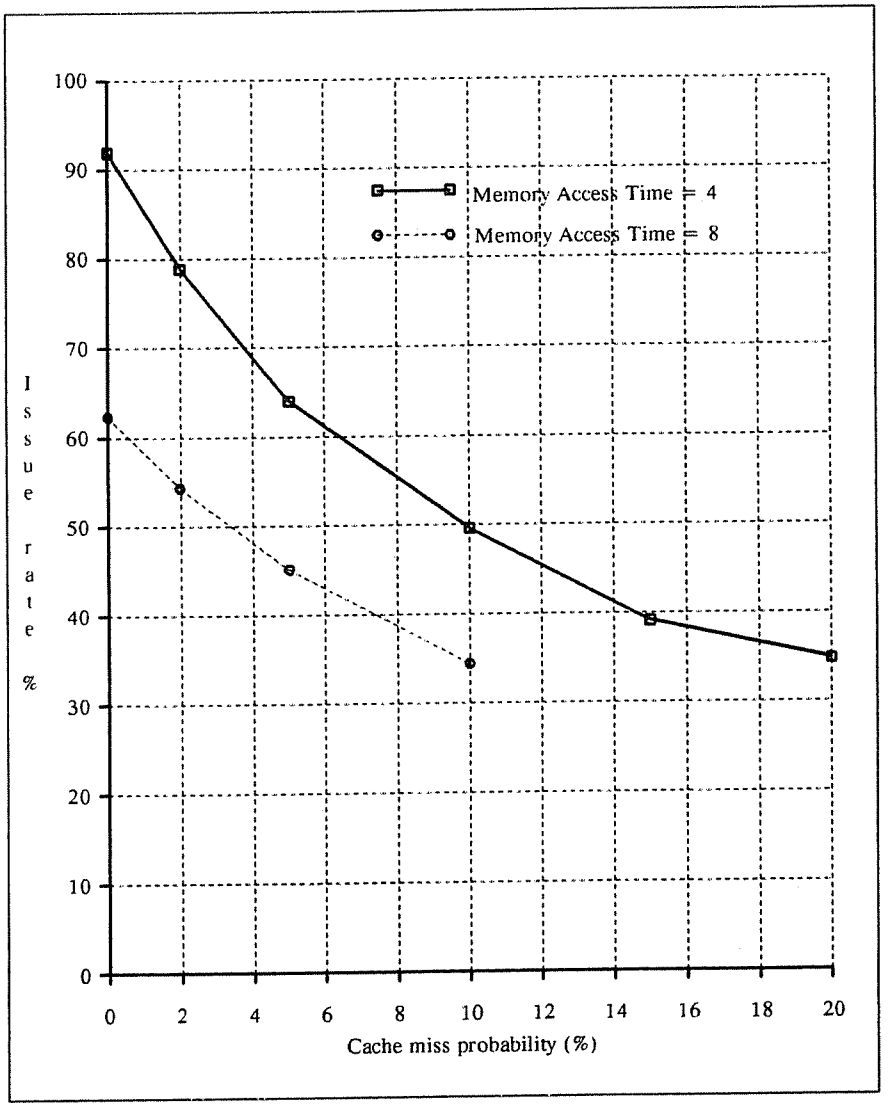


Figure 5.1: Issue Rate vs Cache Miss Rate
for "Typical" Hand-Coded Programs

5.3. Branch Count

The prepare to branch instruction of the PIPE architecture permits the change of control flow to be delayed for a variable number of instructions, as specified by the count field in the branch instruction. As with the inclusion of an instruction cache, there is a trade-off between the performance improvement and the amount of effort needed to support this generalized feature. Providing such a feature makes the implementation somewhat more difficult than that of a conventional machine. Also. to take advantage of this feature, the compiler must perform more work than it would for a conventional instruction set.

The impact of varying the branch count is necessarily related to the frequency of branch instructions in the program. Thus, in addition to the typical compiled and hand-coded sets of program characteristics, we have varied the branch count for the typical hand-coded parameters with branch instruction probability of 20% instead of 10%. A higher branch probability indicates that there are relatively fewer instructions in the inner loops of programs. This is exactly what can be achieved with hand-coded programs.

For each experiment. the branch count was fixed at the indicated value. For the typical hand-coded program characteristics (10% and 20% branch probability), the branch count was varied from 0 to 4. For the compiled set of parameters (5% branch probability) experiments were run for branch counts of 0 and 4. Notice that a branch count of 0 is comparable to a branch instruction in a conventional machine. Increasing the branch count beyond four does not result in improved performance, primarily because the modeled execution pipeline contains four stages between the time an instruction is determined to e a branch and the time the branch condition is evaluated. The results of this study are summarized in Figure 5.2.

There are several interesting conclusions which can be drawn from these results. First, a longer branch distance does not have a significant impact on the performance of the Pascal compiled code. This is due to the low probability of a branch occurring, and the larger impact of short load distances. On the other hand, looking at the hand-coded programs, the performance improves with a

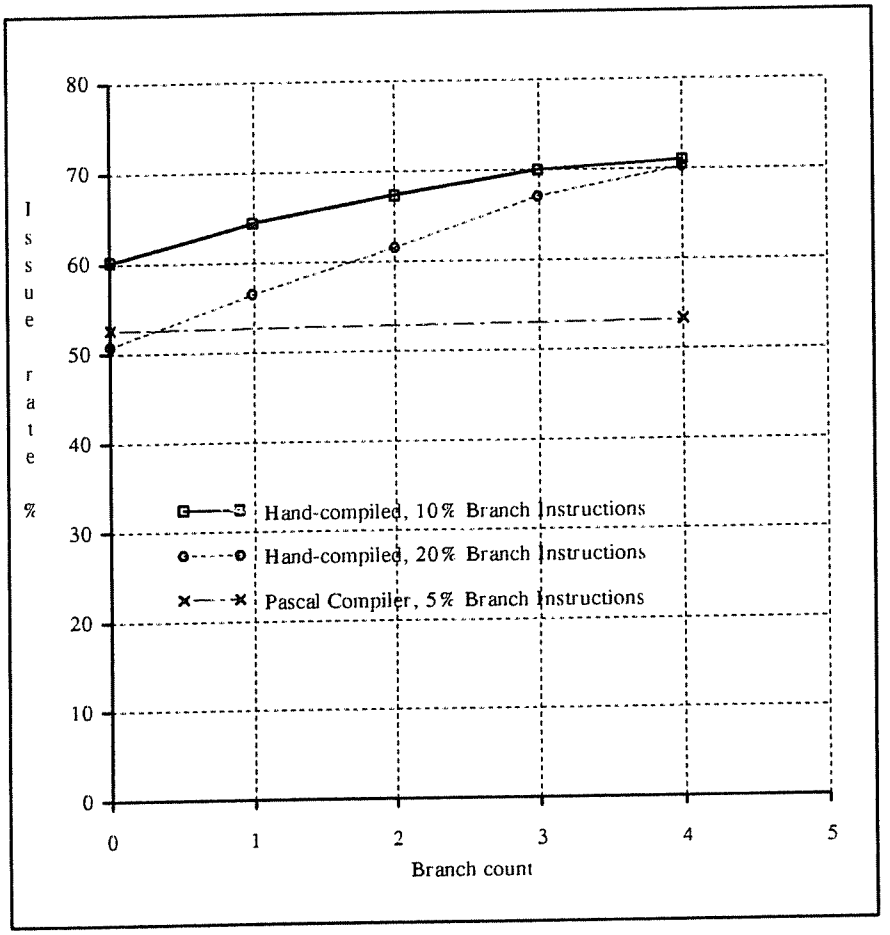


Figure 5.2: Issue Rate vs. Branch Count
for "Typical" Programs

large branch count. Furthermore, very tight code that has a high frequency of branch instructions can achieve issue rates equal to programs with lower branch frequencies, if the branch count is large.

5.4. Load Distance

The final feature that was evaluated with the GMB model was the effect of load distance on performance. The architectural queues found in the PIPE architecture make it easier to effectively perform code scheduling to minimize the effective latency of the memory. Code scheduling for load distance involves maximizing the number of instructions between the instruction that makes a memory request (i.e. places a data item on the LDQ) and the instruction that uses the data item (removes the

data item from the LDQ). Code scheduling to maximize load distance demands extra work of the compiler. How much effort should be expended in writing this portion of the compiler and how much time the compiler should use to do this scheduling can be evaluated by the effect of load distance on system performance.

In evaluating the effects of load distance, we have used the typical compiled program parameters, to determine the performance we could expect if this were the only optimization to be implemented in our current compiler. Using the "typical" compiled program characteristics, the load distance was varied from 0 to 16. The load distance was assumed to be fixed in each experiment, and the mean length of a load instruction sequence was assumed to be 1.5. The results of this study are shown in Figure 5.3. These results indicate that under the given conditions, performance increases significantly until memory contention becomes a limiting factor. The reader will note that memory utilization can be calculated in a straightforward way from the percentage of loads, stores and cache misses, the issue rate, and the memory access time. Thus, the memory utilization curves have the same shape as the issue rate curves for this experiment.

5.5. Summary

Using "typical" compiled and hand-coded PIPE program characteristics, we have evaluated system performance by varying program parameters one by one. In particular, we varied three sets of program characteristics: (1) cache hit rate, (2) branch count, and (3) load distance. Each of these parameters is an important aspect of the PIPE architecture.

The evaluation of varying cache miss rate indicates the sensitivity of the architecture to memory access times for even a small miss rate. The branch count studies indicate that if the probability of branch is low (5%), the prepare-to-branch feature does not significantly improve performance. If the inner loops of a program can be efficiently compiled, to raise the probability of a branch occurring to 10%, the prepare-to-branch with a branch count of 4 can improve performance. Increasing the probability of a branch occurrence to 20% increases the advantage provided by a branch count of 4. These results indicate that this feature of the architecture is worthwhile if one is willing to support

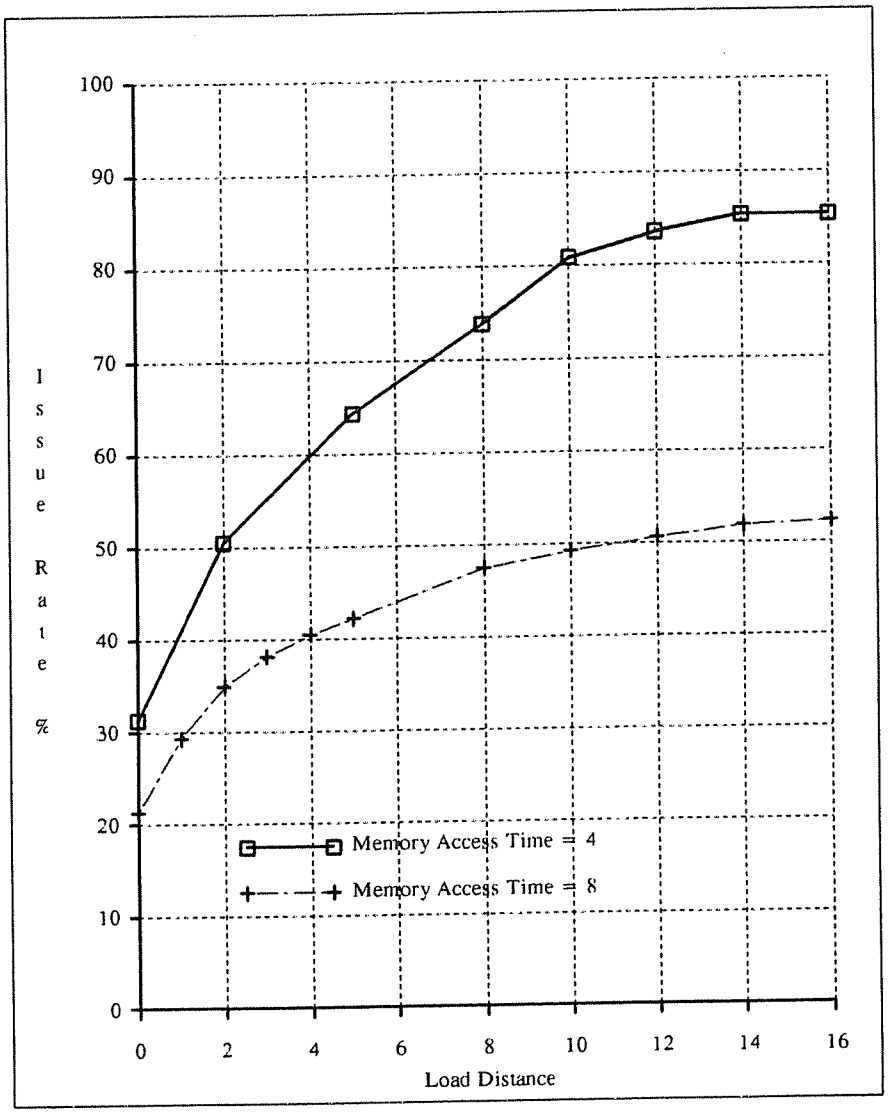


Figure 5.3: Issue rate vs. Load Distance
for "Typical" Compiled Program

the architecture with a compiler that produces efficient code. Load distance was the final parameter varied. These results indicated that scheduling loads is a significant way to improve the performance of the architecture.

6. Conclusions

A GMB performance model of the PIPE architecture, including a probabilistic characterization of program behavior, has been validated by comparison of results with detailed trace-driven

simulations. As part of this research, we determined the program characteristics that have a primary impact on the performance of the PIPE architecture. and have measured these characteristics for compiled and hand-coded versions of the first 12 Lawrence Livermore Loops. The model has proven to be accurate in estimating system performance for a wide range of program and system parameters. We were thus able to use the model to vary program characteristics, such as branch counts and load distances, independently, to obtain some preliminary estimates of the impact of each of these factors on system performance.

The experiments to determine the impact of program characteristics are very preliminary and serve to illustrate the potential utility of the model. In the future we plan to study the effects of program characteristics more systematically, as well as in more detail. Additional performance measures, such as waiting time distributions for *NeedLDQ* and *LatchIn[i]*, which can be obtained from the GMB simulator, may be useful in interpreting performance results. We assumed very fast memory access times in our experiments (1, 4, and 8 clock cycles). We are interested in performing additional experiments with slower memory access times. We also plan to use the model to study the performance of PIPE with a longer execution pipeline, and of the decoupled mode of execution. We expect that model estimates will be useful in guiding future design of PIPE hardware and software.

Acknowledgements

The GMB editor and simulator used in these experiments were designed and implemented by Jeff Meyer and Jane Kasper-Bagley at the University of Wisconsin-Madison.

References

[CoSt81] E. U. Cohler and J. E. Storer, "Functionally Parallel Architecture for Array Processors," *Computer*, Vol. 14, No. 9, pp. 28-36, September 1981.

[Estr78] Estrin, G., "A Methodology for Design of Digital Systems - Supported by SARA at the Age of One," *AFIPS Proceedings of the National Computer Conference*, 1978.

[HJBG82] J. Hennesy, N. Jouppi, F. Baskett, T. Gross and J. Gill, "Hardware/Software Tradeoffs for Increased Performance," *Symposium on Architectural Support for Programming Languages and Operating Systems*, pp. 2-11, March 1982.

[HsPG84] J. T. Hsieh, A. R. Pleszkun and J. R. Goodman, "Performance Evaluation of the PIPE Computer Architecture," Technical Report #566, Computer Sciences Department, University of Wisconsin-Madison, November 1984.

[McMa72] F. H. McMahon, "FORTRAN CPU Performance Analysis," Lawrence Livermore Laboratories, Livermore, CA, 1972.

[Ples82] A. R. Pleszkun, "A Structured Memory Access Architecture," Computer Systems Group Report CSG-10, Coordinated Science Lab., Univ. of Illinois, Urbana, Illinois, October 1982.

[Radi82] G. Radin, "The 801 Minicomputer," *Symposium on Architectural Support for Programming Languages and Operating Systems*, pp. 39-47, March 1982.

[RaVE79] Razouk, R., M. Vernon, and G. Estrin, "Evaluation Methods in SARA - The Graph Model Simulator," *1979 Conference on Simulaton, Measurement, and Modeling of Computer Systems*, Boulder, Colorado, August 1979.

[Russ78] R. M. Russel, "The CRAY-1 Computer System," *Communications of the ACM*, Vol. 21, No. 1, pp. 63-72, January 1978.

[Smit82] J. E. Smith, "Decoupled Access/Execute Computer Architectures," *Proc. of the Ninth Annual Symposium on Computer Architecture*, pp. 112-119, May 1982.

[SPKG83] J. E. Smith, A. R. Pleszkun, R. H. Katz and J. R. Goodman, "PIPE: A High Performance VLSI Architecture," *IEEE Workshop on Computer Systems Organization*, New Orleans, LA, pp. 131-138, March 1983. Also available as Technical Report #512, Computer Sciences Department, University of Wisconsin-Madison, September 1983.

[VdSE83] Vernon, M., E. de Sousa e Silva, and G. Estrin, "Performance Evaluation of Asynchronous Concurrent Systems: The UCLA Graph Model of Behavior," *9th International Symposium on Computer Performance Modeling, Measurement, and Evaluation*, College Park, Maryland, May 25-27, 1983.

[Vern82] Vernon, M., "Performance-Oriented Design of Distributed Computer Systems," *Technical Report No. UCLA-CSD-821217*, UCLA Computer Science Department, December 1982.

[YoGo84] H. C. Young and J. R. Goodman, "A Simulation Study of Architectural Data Queues and Prepare-to-branch Instruction," *Proceedings, IEEE International Conference on Computer Design*, pp. 544-549, October 1984.

Appendix 1: GMB Models of the PIPE Processor and Memory Subsystem

```
(herald pipe (env t))

;   PIPE Processor and Memory System.
;   (Control, Data and Interpretation Domains)

;  NODE DEFINITIONS
(def-nodes        InstrGen Latch1 Iq Latch2 Decode WaitLDQ AllocOutQ Issue
                  ALU1 ALU2 PSender PReceiver BranchDecide MReceiver
                  StoreMatch ReqServer Latch ReqResult AllocLDQ MSender)
(def-multiserver (MemModules 'infinite) (SINK 'infinite))

;  NODE INPUT and OUTPUT LOGIC for PIPE PROCESSOR MODEL
(node-io InstrGen          (* IGenReady) (+ (CacheReady) (CacheMiss)))
(node-io Latch1            (* CacheReady Latch1Ready)
                           (+ (IqIn BrCntEnd) (IqIn IGenReady)))
(node-io BranchDecide      (* BrCntEnd BrConditon) (IGenReady))
(node-io Iq                (* IqIn) (IqOut))
(node-io Latch2            (* IqOut Latch2Ready) (DecodeIn Latch1Ready))
(node-io Decode            (* DecodeIn)
                           (+ (NeedLDQ) (NeedOUTQ) (Latch2Ready IssueIn)))
(node-io WaitLDQ           (* NeedLDQ LDQ) (+ (Latch2Ready IssueIn) (NeedOUTQ)))
(node-io AllocOutQ         (* NeedOUTQ OutQspace) (IssueIn Latch2Ready))
(node-io Issue             (* IssueIn) (IssueOut))
(node-io ALU1              (* IssueOut)
                           (+ (al2 LDQspace BrConditon) (al2 extral BrConditon)
                              (al2 LDQspace extra2) (al2 extral extra2)))
(node-io ALU2              (* al2 ) (+ (ToSINK) (OutQ)))
(node-io PSender           (> (* ReqQspace CacheMiss) (* ReqQspace OutQ))
                           (+ (ReqIn extral) (ReqIn OutQspace)))
(node-io PReceiver         (* WordIn) (+ (CacheReady) (ToSINK) (LDQ)))
(node-io SINK              (+ (* ToSINK) (* extral) (* extra2)) ( ))

;  NODE INPUT and OUTPUT LOGIC for MEMORY SYSTEM MODEL
(node-io MReceiver         (* ReqIn) (+ (Saq) (Sdq) (ReqQ extral) ((ReqQ 4))))
(node-io StoreMatch        (* Saq Sdq) (ReqQ ReqQspace))
(node-io ReqServer         (* ReqQ ServerReady)
                           (+ (LatchIn0 ReqQspace) (LatchIn0 extral)
                              (LatchIn1 ReqQspace) (LatchIn1 extral)
                              (LatchIn2 ReqQspace) (LatchIn2 extral)
                              (LatchIn3 ReqQspace) (LatchIn3 extral)))
(node-io Latch             (+ (* LatchIn0 ModuleReady0) (* LatchIn1 ModuleReady1)
                              (* LatchIn2 ModuleReady2) (* LatchIn3 ModuleReady3))
                           (MemIn ServerReady))
(node-io MemModules        (* MemIn)
                           (+ (MemOut ModuleReady0) (MemOut ModuleReady1)
                              (MemOut ModuleReady2) (MemOut ModuleReady3)))
(node-io ReqResult         (* MemOut) (+ (MSenderIn) (ReturnData) (ToSINK)))
(node-io AllocLDQ          (* ReturnData LDQspace) (MSenderIn))
(node-io MSender           (* MSenderIn) (WordIn))

;  DATASET and QUEUE DEFINITIONS
(def-datasets     CMissP CacheMissCount CacheMissTag BrTakenP BrTakenStatus
                  BrDist BranchP BrHist MAccP LdHist lds LdDist SendOutQCopy
                  MaxUnmachedStores SaqLength MemoryConfP
                  PreviousMemoryReference MemoryAccessTime)
(def-dataqueues   BrTag ReleaseLDQ SendOutQ OutQTag ReqType ReqQType
                  MemInQ ServiceQ MSenderQ)

;  PROCESSOR READ/WRITE ACCESS SPECIFICATION for PIPE PROCESSOR MODEL
(simple-io InstrGen-proc          (CMissP BrTakenStatus CacheMissCount)
                                  (BrTakenStatus CacheMissCount))
(simple-io Latch1-proc            (BrDist BranchP BrHist) (BrHist BrTag))
(simple-io BranchDecide-proc      (BrTakenP) (BrTakenStatus))
(simple-io Decode-proc            (MAccP LdDist LdHist lds)
                                  (LdHist lds SendOutQCopy ReleaseLDQ SendOutQ))
(simple-io WaitLDQ-proc           (SendOutQCopy) ( ))
(simple-io ALU1-proc              (ReleaseLDQ BrTag) ( ))
(simple-io ALU2-proc              (SendOutQ) (OutQTag))
(proc-io PSender-proc     (PSenderIn (controlled read (CacheMissTag OutQTag) (CacheMiss OutQ)))
                          (ReqType))
(simple-io PReceiver proc         (MSenderQ) ( ))
```

```
;  PROCESSOR READ/WRITE ACCESS SPECIFICATION for MEMORY SYSTEM MODEL
(simple-io MReceiver proc        (ReqType MaxUnmachedStores SaqLength)
                                 (ReqQType SaqLength))
(simple-io StoreMatch proc       () (ReqQType))
(simple-io ReqServer-proc        (ReqQType PreviousMemoryReference MemoryConfP)
                                 (MemInQ ServiceQ PreviousMemoryReference))
(simple-io MemModules-proc       (MemInQ MemoryAccessTime) ())
(simple-io ReqResult-proc        (ServiceQ) (MSenderQ))
(simple-io AllocLDQ-proc         () (MSenderQ))
(simple-io MSender-proc          () ())

;  INTERPRETATION DOMAIN for PIPE PROCESSOR MODEL
(def-function InstrGen-proc (delay 1)
        (read-from CMissP) (read-from BrTakenStatus)
        (if (< (random) (if BrTakenStatus
                                 (block  (write-to BrTakenStatus ())
                                         (car CMissP))    ; branch taken
                                 (cadr CMissP)))          ; branch not taken
                (block  (output to-arc (CacheMiss))      ;instruction cache miss
                        (write to CacheMissCount (1+ (read-from CacheMissCount))))
                (output-to-arc (CacheReady))))           ;instruction cache ready

(def-function Latch1 proc (delay 0)
        (lset br ()) (read-from BrHist)
        (if (<0? BrHist)         ; No pbr is pending.  May generate one.
                (if (< (random) (read-from BranchP))     ; generate pbr instr.
                        (block (set BrHist (prand (read-from BrDist))) (set br t))))
        (if (>=0? BrHist)        ; Is there a pending pbr instruction?
                (block ; check and update current branch count
                        (if (=0? BrHist)
                                (output-to-arc (BrCntEnd IqIn)) ; branch count exhausted
                                (output-to-arc (IGenReady IqIn)))        ; not exhausted
                        (set BrHist (-1+ BrHist)) (write-to BrHist BrHist))
                (output-to-arc (IGenReady IqIn)))        ; No pbr is pending.
        (write-to BrTag br))

(def-function Decode-proc (delay 1)
        (read-from MAccP) (read-from LdHist) (read-from lds)
        ; Generate possible memory accesses
        (set MAccP (if lds (car MAccP) (cadr MAccP))) ; conditonal probability
        (lset tag (prand MAccP))
        (write-to SendOutQCopy tag) (write-to SendOutQ tag)
        (set LdHist (map -1+ LdHist)) ; decrement all entries in the load history
        (if (and (not (null? LdHist)) (<0? (car LdHist)))
                (block  (set LdHist (cdr LdHist))
                        (output-to-arc (NeedLDQ)) ; This instr uses LDQ.
                        (write-to ReleaseLDQ t))
                (block  (write-to ReleaseLDQ ())
                        (if (eq? tag 'none-tag)
                                (output-to-arc (Latch2Ready IssueIn))
                                (output-to-arc (NeedOUTQ))))) ; This instr uses Outq.
        (if (eq? tag 'load-tag) ; Generate load distance and insert it into
                (block  (read-from LdDist)       ; the load history
                        (set LdHist (insert (prand LdDist) LdHist))
                        (write-to lds t))
                (write-to lds ()))
        (write-to LdHist LdHist))        ; update load history

(def-function WaitLDQ-proc (delay 0)
        (if (eq? (read-from SendOutQCopy) 'none-tag)
                (output-to-arc (Latch2Ready IssueIn))
                (output-to-arc (NeedOUTQ))))

(def-function ALU1-proc (delay 1)
        (if (read-from BrTag)
                (output-to-arc (a12 BrConditon)) ; branch condition evaluated
                (output-to-arc (a12 extra2)))
        (if (read-from ReleaseLDQ)
                (output-to-arc (LDQspace))       ; release LDQ space
                (output-to-arc (extra1))))

(def-function ALU2-proc (delay 1)
        (read-from SendOutQ)
        (if (eq? SendOutQ 'none-tag)
                (output-to-arc (ToSINK))
                (block (output to-arc (OutQ)) (write to OutQTag SendOutQ))))

(def-function PSender-proc (delay 0)
        (read-from PSenderIn) (write-to ReqType PSenderIn)
        (if (eq? PSenderIn 'CacheMiss-tag)
                (output-to-arc (ReqIn extra1))
                (output-to-arc (ReqIn OutQspace))))      ; release OutQ space
```

```
(def-function PReceiver-proc (delay 0)
        (read-from MSenderQ)
        (cond   ((eq? MSenderQ 'CacheMiss-last) (output-to-arc (CacheReady)))
                ((eq? MSenderQ 'load-tag) (output-to-arc (LDQ)))
                (else (output-to-arc (ToSINK)))))

(def-function BranchDecide-proc (delay 0)
        (if (< (random) (read-from BrTakenP)) (write-to BrTakenStatus t)))

;  INTERPRETATION DOMAIN for MEMORY SYSTEM MODEL
(def-function MReceiver-proc (delay 0.000000001)
        (read-from ReqType)
        (cond   ((eq? ReqType 'store-tag)
                        (read-from MaxUnmachedStores) (read-from SaqLength)
                        (if (or (<=0? (+-SaqLength MaxUnmachedStores))
                                (and (< (random) 0.5) (< SaqLength MaxUnmachedStores)))
                                (block  (output-to-arc (Saq))
                                        (write-to SaqLength (1+ SaqLength)))
                                (block  (output to-arc (Sdq))
                                        (write-to SaqLength (-1+ SaqLength)))))
                ((eq? ReqType 'load-tag)
                        (write-to ReqQType ReqType) (output-to-arc (ReqQ extra1)))
                ((eq? ReqType 'CacheMiss-tag)   ; request a whole cache line
                        (write-to ReqQType 'CacheMiss-tag)
                        (write-to ReqQType 'CacheMiss-mid)
                        (write-to ReqQType 'CacheMiss-mid)
                        (write-to ReqQType 'CacheMiss-last)
                        (output-to-arc ((ReqQ 4))))
                (else (ERROR "MReceiver-proc gets ReqType=~s~%" ReqType))))

(def-function StoreMatch-proc (delay 0) (write-to ReqQType 'store-tag))

(def-function ReqServer-proc (delay 0.999999999)
        (read-from ReqQType)
        (lset module (case ReqQType     ; determine memory module of the request
                ((CacheMiss-tag) 0)
                ((CacheMiss-mid CacheMiss-last) (1+ (read-from PreviousMemoryReference)))
                (else (if (< (random) (read-from MemoryConfP))
                        (read-from PreviousMemoryReference) ; first-order conflict
                        (mod (+ (read-from PreviousMemoryReference) (funifor-m 1 4))
                                4)))))  ; uniform otherwise
        (cond   ((= module 0) (output-to-arc (LatchIn0)))
                ((= module 1) (output-to-arc (LatchIn1)))
                ((= module 2) (output-to-arc (LatchIn2)))
                ((= module 3) (output-to-arc (LatchIn3)))
                (else (ERROR "module = ~d~%" module)))
        (if (and (neq? ReqQType 'CacheMiss-tag) (neq? ReqQType 'CacheMiss-mid))
                (output-to-arc (ReqQspace))
                (output-to-arc (extra1)))
        (write-to PreviousMemoryReference module)
        (write-to MemInQ module) (write-to ServiceQ ReqQType))

(def-function MemModules-proc
        (read-from MemoryAccessTime) (delay MemoryAccessTime)
        (read-from MemInQ)
        (cond   ((= MemInQ 0) (output-to-arc (MemOut ModuleReady0)))
                ((= MemInQ 1) (output-to-arc (MemOut ModuleReady1)))
                ((= MemInQ 2) (output-to-arc (MemOut ModuleReady2)))
                ((= MemInQ 3) (output-to-arc (MemOut ModuleReady3)))
                (else (ERROR "MemInQ = ~d~%" MemInQ))))

(def-function ReqResult-proc (delay 0)
        (read-from ServiceQ)
        (cond   ((eq? ServiceQ 'store-tag) (output-to-arc (ToSINK)))
                ((eq? ServiceQ 'load-tag) (output-to-arc (ReturnData)))
                (else (block    (output-to-arc (MSenderIn)) ; i.e. ReturnInstr
                                (write-to MSenderQ ServiceQ)))))

(def-function AllocLDQ-proc (delay 0) (write-to MSenderQ 'load-tag))

(def-function Iq-proc (delay 1))
(def-function Issue-proc (delay 1))
(def-function MSender-proc (delay 1))
(def-function Latch2-proc (delay 0))
(def-function AllocOutQ-proc (delay 0))
(def-function SINK-proc (delay 0))
(def-function Latch-proc (delay 0))

; UTILITY FUNCTIONS
(define (prand pdist)   ; Generate random elements from a probability distribution
        (let* ((rand (random)) (car>= (lambda (x) (if (>= (car x) rand) (cadr x) ()))))
                (any car>= pdist)))
```

Appendix 2: Load Distance and Branch Count Distributions for the Loop Benchmarks

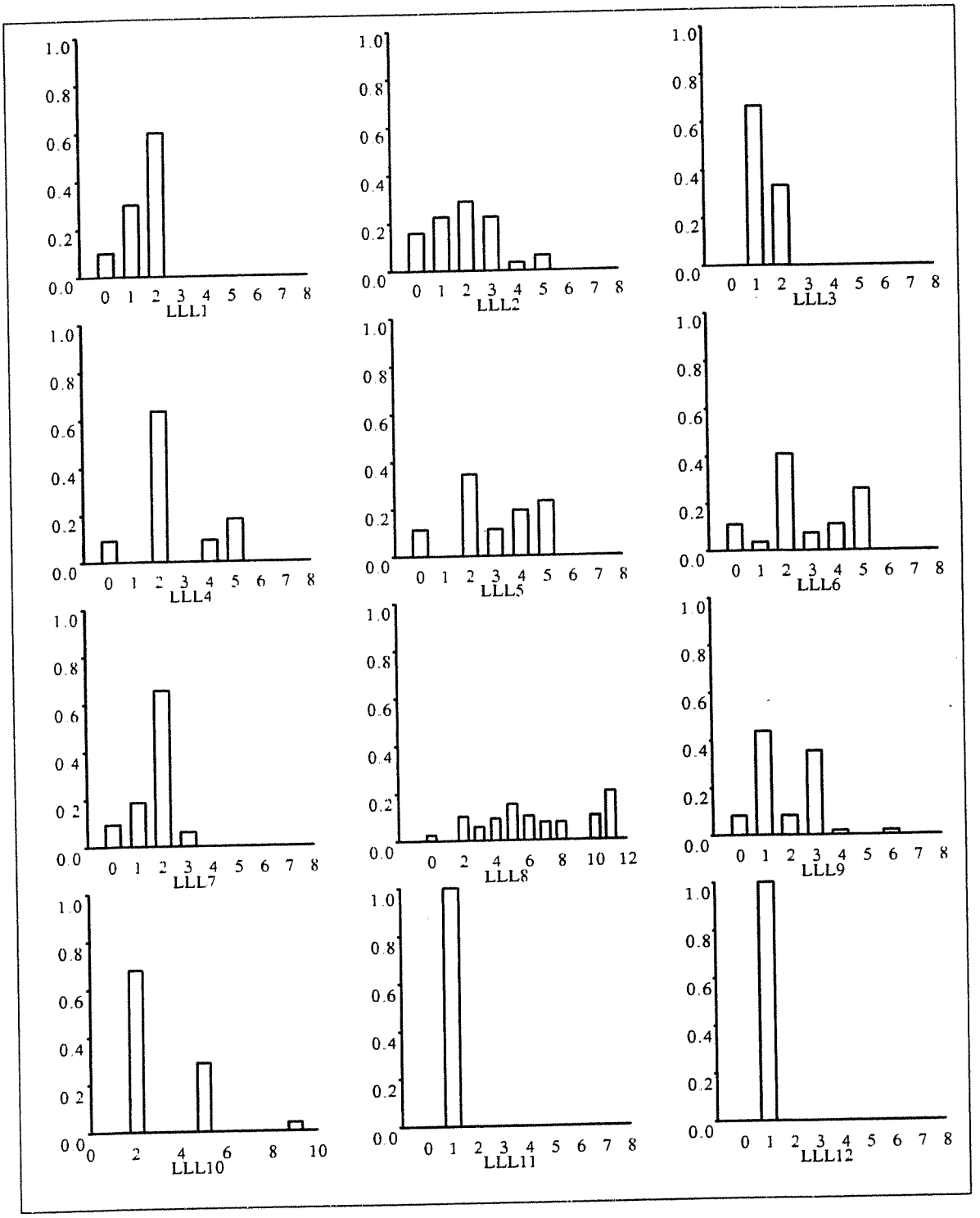


Figure A2.1: Load Distance Distribution for Compiled Benchmarks

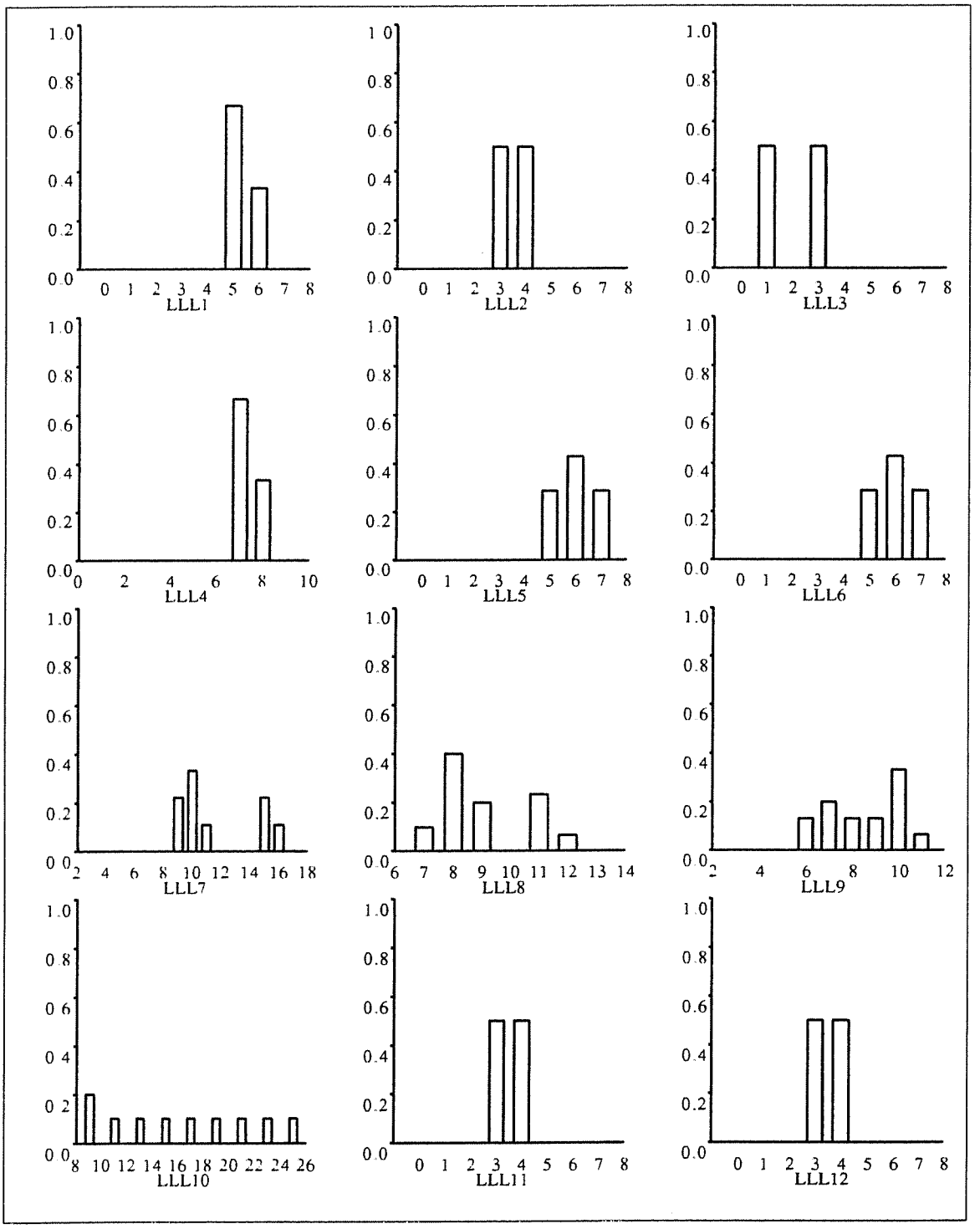


Figure A2.2: Load Distance Distribution for Hand-coded Benchmarks

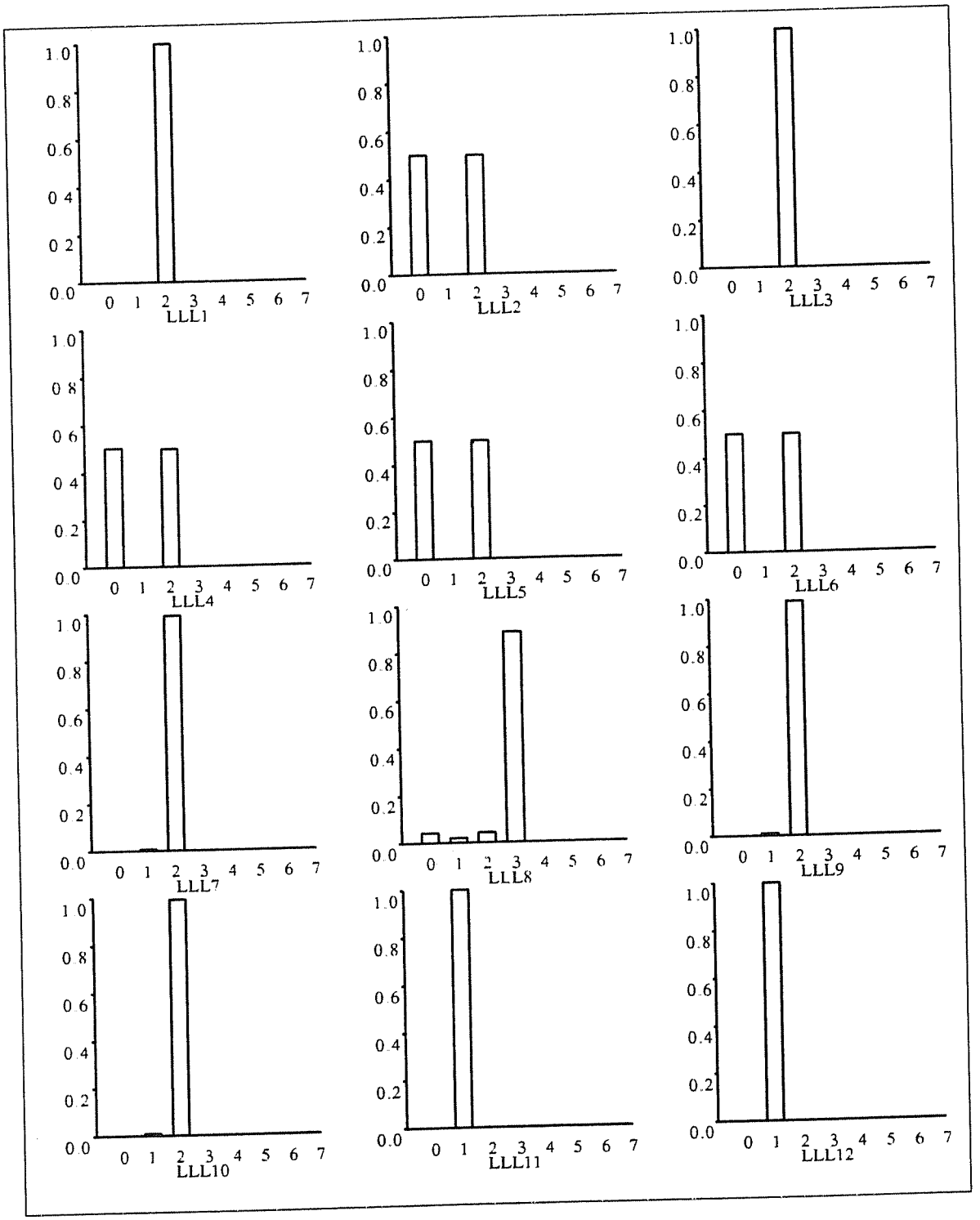


Figure A2.3: Branch Count Distribution for Compiled Benchmarks

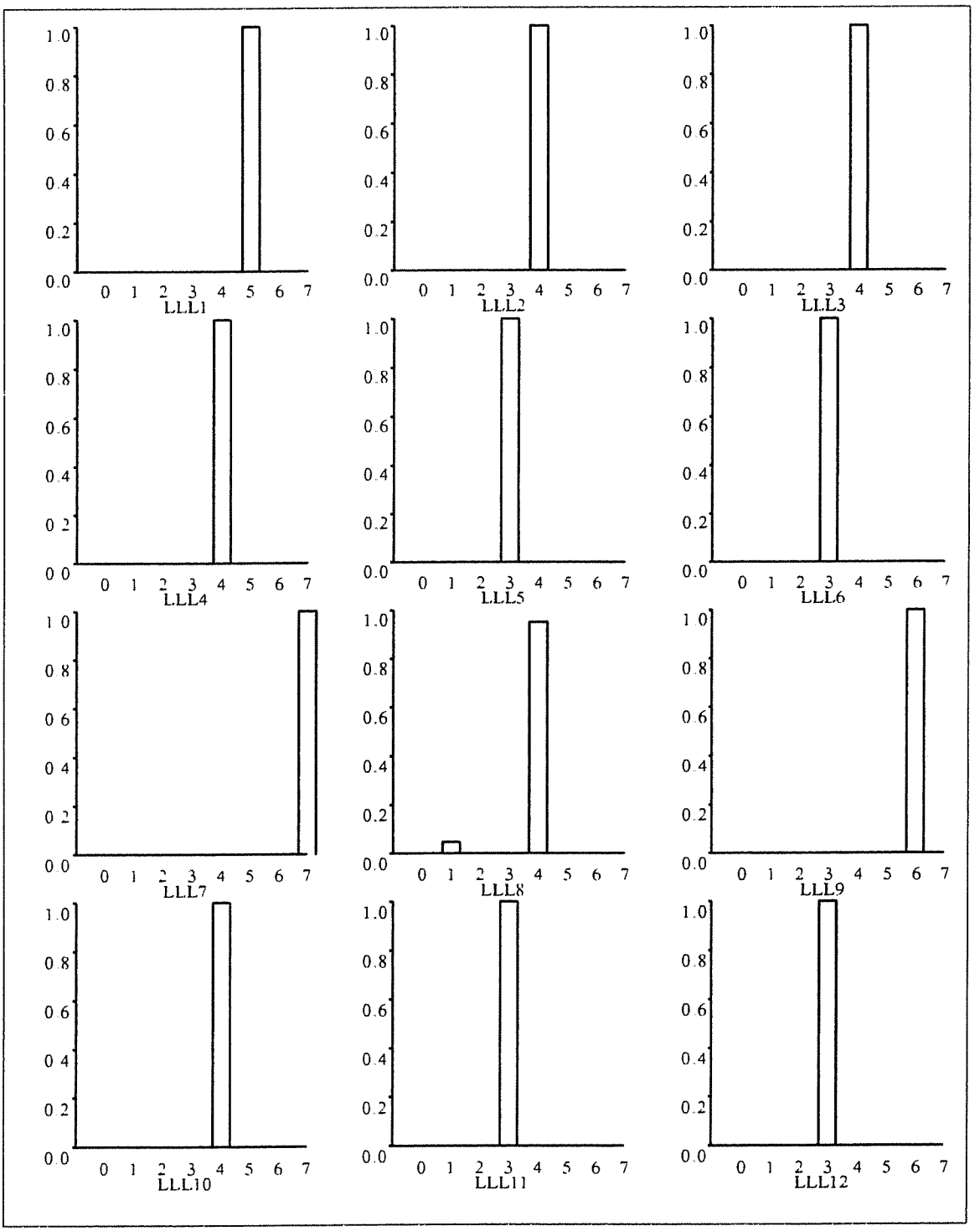


Figure A2.4: Branch Count Distribution for Hand-coded Benchmarks

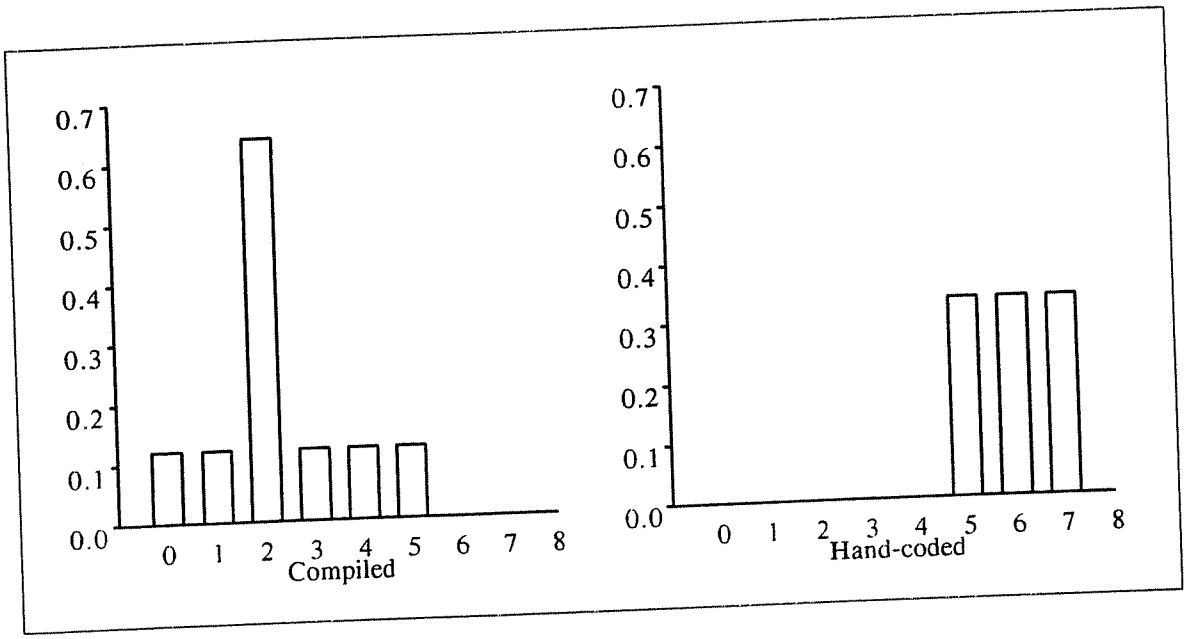


Figure A2.5: Typical Branch Count Distribution

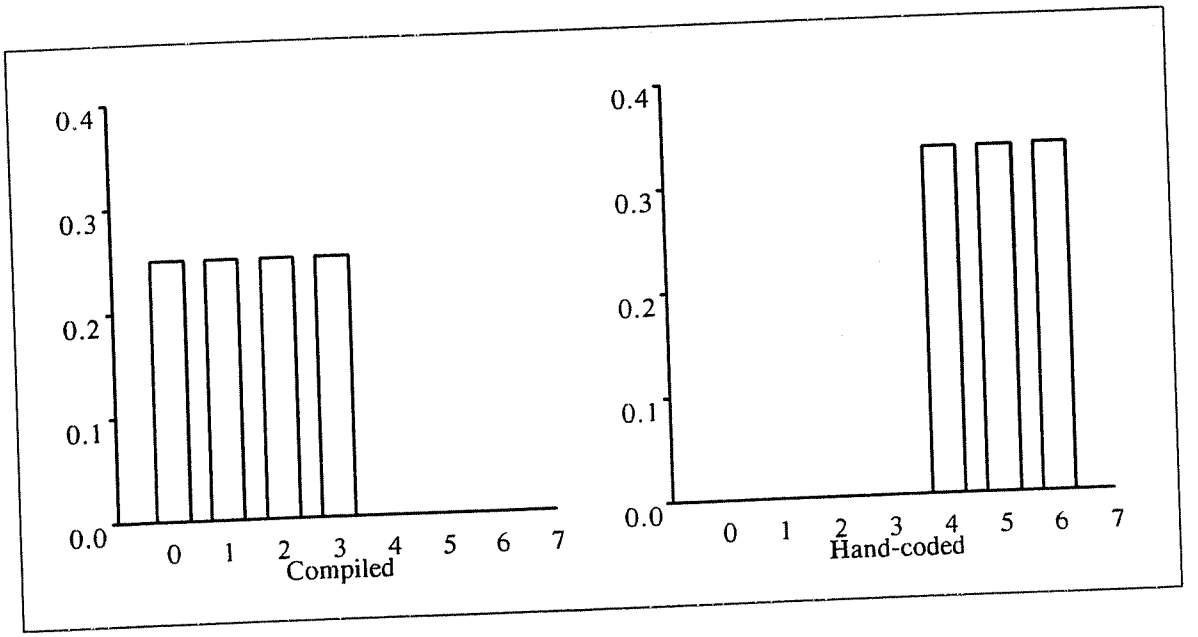


Figure A2.6: Typical Load Distance Distribution