Processor design and program behavior are inseparable. It is this relationship that allows us to determine how a component will perform before it is implemented.

Instruction-Level Program and Processor Modeling

Myron H. MacDougall, Amdahl Corporation

As a processor is developed from high-level design to logic design and implementation, a large number of design decisions are constantly being made—each of which involves evaluating the relative cost and performance of a set of design alternatives. Whatever the performance metric used, it reflects the rate at which work is being accomplished, and the design decision, in turn, is made with an eye toward the work to be performed. This association between work and design is reflected in the adjectives applied to "processor"—general-purpose, real-time, signal, or scientific—and continues as design objectives evolve from general to specific. That is, as the design level of detail increases, so does the level of detail at which work is described and performance evaluated. The performance evaluation process, then, involves two interlocking hierarchical models: a model of the design and a model of the workload.

This relationship of design and work is illustrated here in an instruction-level workload model and its decomposition in design analysis. Although details of this discussion pertain to a specific architecture—the IBM 370 and compatible systems—and to a particular implementation, the general approach is applicable to most high-performance hardware and software design environments. In either kind of environment, the performance issue is how fast a specific piece of software runs on a specific piece of hardware. For an analysis of that performance, it largely is irrelevant whether we approach from a hardware perspective, a software perspective, or both— the analysis is essentially the same; what matters is the direction we take to improve performance.

System-level workload characterization and decomposition

While some processors are designed for a single, specific application, such as signal processing, most are intended for a range of applications. Consequently, we begin the development of the instruction-level workload model presented here with a survey of current and projected user environments, followed by a breakdown of key environments in terms of processor use and program or application type. Samples of each application type are collected; they may comprise programs and data files when the application can be easily separated from the rest of the system, or may be a statistical description of database transactions or timesharing system commands.

This process is illustrated in Figure 1. In this example, a system-level analysis of a second-shift operation in an insurance company (a "commercial batch" environment), showed that 47 percent of processor time was operating system execution time, 30 percent was Cobol batch program execution, 14 percent was sort program execution, and the remaining time was associated with various utilities such as program loading. To analyze these components of the system's workload, programs representative of the Cobol execution portion were identified and collected.

Environment definition is, for the most part, judgmental. Some environments are clearly unique, such as airline reservation systems; many more, however, have essentially similar applications but in different proportions. Characterization and decomposition of the work performed in a given environment is based largely on statistical analyses of standard system accounting data augmented by hardware measurements. When feasible, samples of applications of the same type collected in different environments are merged to form a composite. For example, Cobol programs from varied environments collectively form a representative sample of such programs.

The application sample sets are then used, directly or indirectly, to define instruction-level workloads; these, in turn, are used to estimate processor performance at the instruction level. To estimate performance at the system level, the entire process is simply reversed. Program estimates are combined to estimate component performance, and component estimates are combined to estimate system-

0018-9162/84/0700-0014$01.00 © 1984 IEEE

level performance for various environments. (Note that the component proportions originally measured may no longer directly apply because of differences in relative component performance.)

Instruction-level workload characterization

The method used to collect data for characterization of an application sample set at the instruction level depends on the nature of the set. The basic methods used in our environment are illustrated in Figure 2. When the set is a collection of programs, instruction-level data can be collected by interpretively executing and tracing each program. When the sample set is a description of, for example, database transactions, these descriptions are used to define scripts, which are then used with a terminal simulator to generate a workload for the database system. Program trace and hardware monitor tools then are used to obtain data on the instruction-level behavior of the database system. Much the same approach is used for the operating system; a workload of the appropriate type

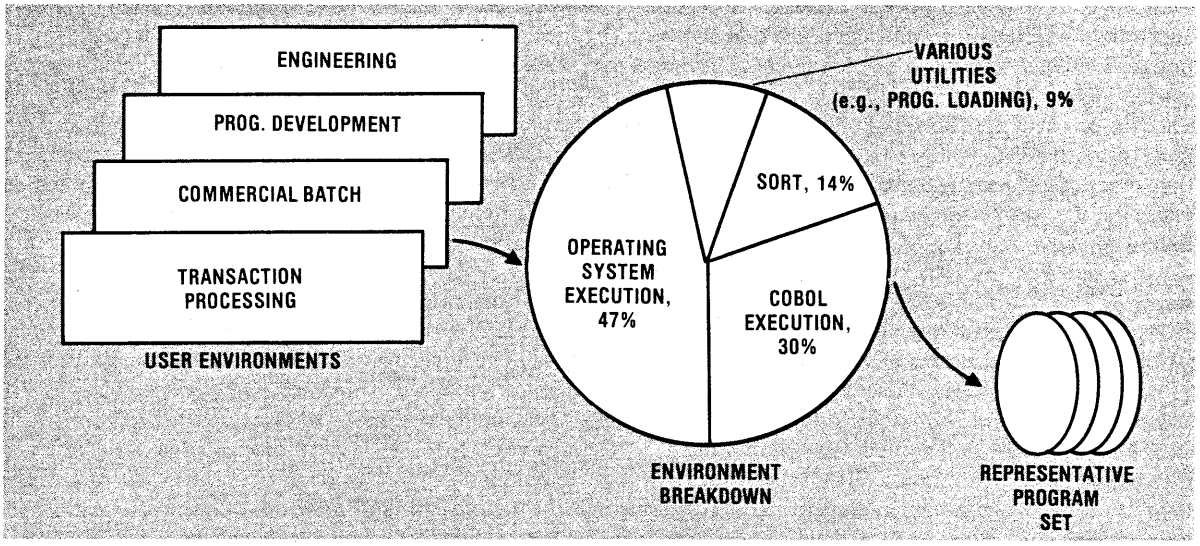


Figure 1. The workload decomposition process. In our example, Cobol execution programs were collected as the representative program set.

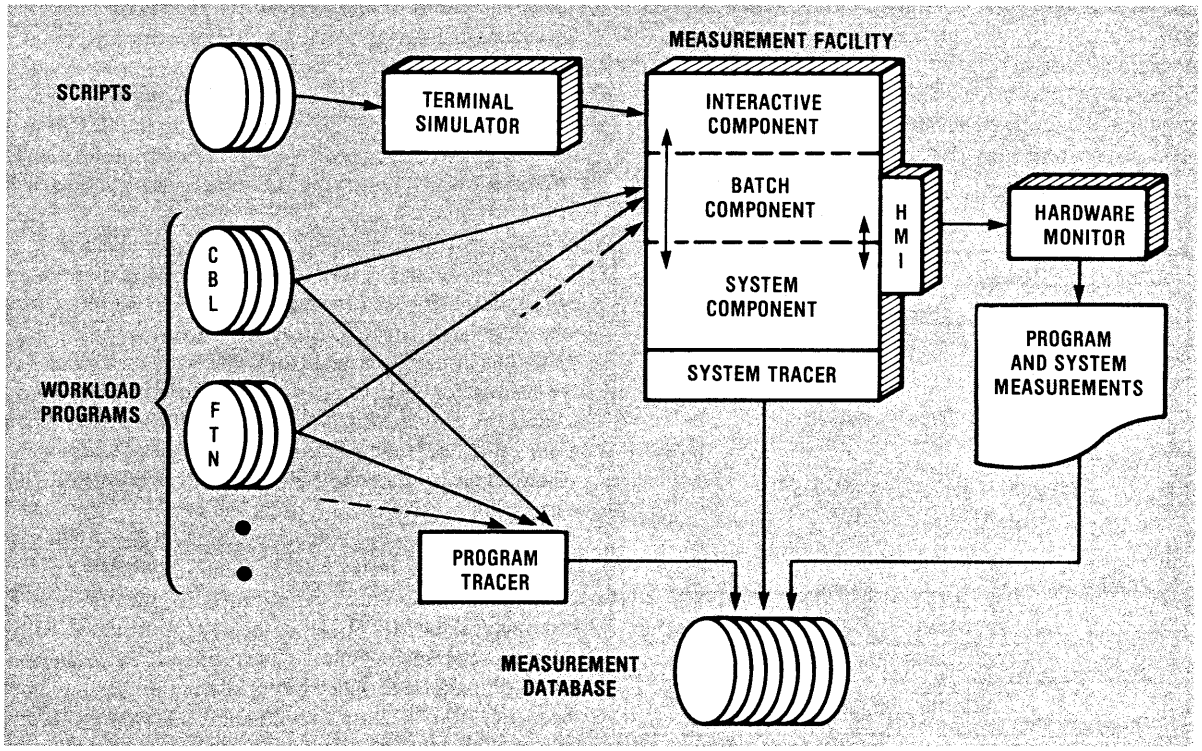


Figure 2. The measurement environment used in characterizing an application sample set at the instruction level.

(e.g., batch, timesharing) is constructed, generally from programs collected in the workload characterization process, which provides the desired system load point. Instruction-level measurement data for application and operating system components then is collected during execution of this workload.

Tools. The primary collection tools used are a program tracer, a system tracer, and a hardware measurement interface with its associated hardware monitor equipment. Both the program and system tracers interpretively execute machine instructions and produce a trace record for each instruction. The trace record includes the instruction address, operation code, operand addresses and lengths, and, when specified, operand values. For branch instructions, the record also includes the target address and a branch-taken indicator. The program tracer is used to trace the problem state portion of individual programs (in most cases, the dominant portion of the program's execution time). This tracing operation executes as a simple batch job that can be run at any time on any of our computer center systems. The system tracer requires a dedicated system but can trace both problem and supervisor state portions of a workload.

During the design of Amdahl processors, signals of interest from a performance viewpoint are identified and provisions made to route these signals to the *hardware measurement interface.* In some instances, these signals exist in the basic design; in other cases, additional logic is incorporated to generate them. The HMI provides a variety of facilities for signal manipulation (including counting, scaling, signal stretching, and sampling) as well as an electrical interface compatible with commercial hardware monitors.

A number of tactical issues must be weighed when using these tools. The data obtained by tracing is comprehensive, but the process is relatively expensive and, in operating system tracing, significantly perturbs the behavior of the system. Perturbation is not generally a problem when tracing individual programs; however, since a program may execute billions of instructions, a sampling mechanism is used to keep trace output to a reasonable volume. (The number of instructions that must be traced to get an accurate representation of a program's behavior depends on the complexity of the program. As part of our research into program behavior, we have developed trace reduction algorithms that essentially regenerate an instruction-level program graph from the trace; we hope to build on this work to develop heuristics for trace reduction "on the fly.") Sampling is also used to reduce the perturbation inherent in operating system tracing. Hardware measurement is efficient and does not perturb the system, but it limits the kinds of data that can be collected. One important application of hardware measurement data is the validation of sampled trace data.

Source-level instrumentation (for collecting data on statement execution frequencies and times, or for analyzing source statement flow) is an important tool in environments concerned with instruction set specification or compiler design. In our environment, source-level analysis is used primarily to identify instruction sequence and address reference patterns generated for common operations such as procedure calls and data type conversion.

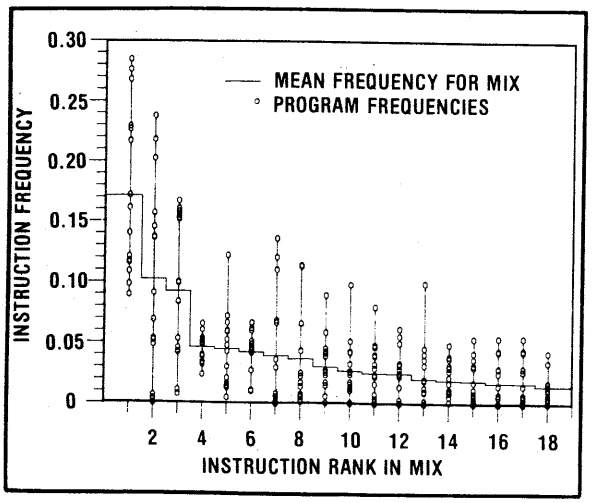


Figure 3. Variation in intramix instruction frequency.

Instruction mix descriptions. The data obtained from instruction trace analysis and hardware measurement is used to generate instruction mix descriptions for each key application type. Thus, to create an instruction mix description for Cobol execution, each program in our sample set for that type is traced, and the trace is analyzed to obtain data on instruction frequencies, operand length distributions, and taken branch frequencies. Data is then combined to form a composite of the Cobol programs. The detail of these instruction-mix descriptions expands as the design evolves and design questions arise. For example, designers concerned with branch instruction performance may want to know how often the condition code is set by the instruction immediately preceding the branch; when told "96 percent" they then want to know which instructions are the most frequent branch predecessors. Hundreds of such questions arise during the design process, often accompanied by performance questions like "What's it worth to set the condition code a cycle early in this instruction?"

These instruction-mix descriptions are illustrated in Tables 1 through 4. Table 1 shows an instruction frequency tabulation for a Cobol execution (CBL/X) instruction mix. This and similar tabulations encompass all the instructions in the programs from which the mix is derived, not just the 20 or 25 most frequent. (Certain instructions, while low in frequency, contribute significantly to program execution time.) The graph of Figure 3 provides a visual indication of how instruction frequencies vary among the programs composing the mix.

Table 2 shows a tabulation of the operand length distribution for the move character instruction in the CBL/X instruction mix. (This data is for the nonoverlapped operand, or MOVE, case; the MVC instruction with a one-byte operand overlap is frequently used to clear or blank-fill data fields.) The MVC instruction has a relative frequency of 0.0422 in this mix; approximately 96 percent of these are MOVES. From the cumulative distribution, note that two thirds of these instructions have operand

lengths of eight bytes or less. However, this high proportion of short moves is only part of the picture. Figure 4 shows the proportion of MVCs with operand lengths greater than *b* together with the proportion of total bytes moved by this instruction that are moved by MVCs with operand lengths greater than *b*. Clearly, most bytes moved by this instruction are moved by a small proportion of MVCs with relatively long operands. This behavior is a challenge to the ingenuity of the designer: to handle the frequent short moves efficiently, the MVC algorithm must have minimal initialization time; at the same time, the algorithm must be sophisticated enough to handle long moves at the maximum rate permitted by path widths, for example.

Branch instructions can represent up to one third of the nstructions in some mixes and are thus an important consideration in any design—especially in a pipelined design. Table 3 shows part of the instruction description for the Branch on Condition in an instruction mix representing IBM's MVS SP1.3 operating system. The BC instruction is the single most frequent instruction in this mix; the sequences TM-BC and LTR-BC account for 25 percent of the instructions in this mix, and so draw a great deal of design attention. Other data in branch instruction descriptions may include branch direction and distance statistics, the distribution of the number of instructions in the target block (word, double-word, etc.), branch repeat frequency (how often a branch is taken or not taken on successive executions), and target repeat frequency (how often successive executions of a taken branch go to the same target address).

Table 1. Mean instruction frequencies: Cobol execution instruction mix.

RANK	OP CODE	f	Σ(f)
1	L	.17116	.17116
2	BC	.10217	.27333
3	BCR	.09220	.36553
4	ST	.04643	.41187
5	LA	.04470	.45657
6	MVC	.04220	.49878
7	PACK	.03928	.53806
8	LH	.03667	.57473
9	CLC	.03033	.60506
10	CLI	.02720	.63226
11	LR	.02543	.65769
12	TM	.02493	.68263
•	•	•	•
•	•	•	•
•	•	•	•
44	EX	.00264	.97239
45	SH	.00257	.97496
46	STC	.00243	.97739
47	ED	.00224	.97963
48	ALR	.00199	.98163
49	AL	.00198	.98360
50	MP	.00175	.98536
51	XC	.00153	.98688
52	CL	.00147	.98836
53	DP	.00140	.98975
54	MVO	.00117	.99092
55	C	.00099	.99191
56	ICM	.00098	.99288
57	LCR	.00091	.99380
•	•	•	•
•	•	•	•
•	•	•	•
89	SRDL	.00000	1.00000
90	LNR	.00000	1.00000

Table 2. Distribution of operand lengths for Move Character instructions (MVCs) with nonoverlapped operands in Cobol execution Instruction mix: pr (instrs) = 0.4220; pr (MVCs) = 0.96188; mean length = 32.56 bytes.

LENGTH IN BYTES (b)	pr (i,b)	Σpr (i,b)
1	0.10182	0.10182
2	0.11899	0.22081
3	0.13007	0.35089
4	0.11932	0.47030
5	0.04411	0.51441
6	0.04879	0.56320
7	0.05449	0.61769
8	0.05513	0.67282
9	0.04502	0.71784
10	0.02816	0.74600
•	•	•
•	•	•
•	•	•
256	0.03837	1.00000

Table 3. Branch on Condition (BC) in an IBM MVS SP1.3 operating system instruction mix.

CONDITIONAL BC PREDECESSOR	FREQUENCY	BC SUCCESS RATIO
TM	0.3578	0.5049
LTR	0.1599	0.4554
CLI	0.0770	0.5000
CR	0.0666	0.6143
C	0.0572	0.4690
CLC	0.0507	0.5577
CL	0.0334	0.1864
CLR	0.0325	0.7042
CH	0.0278	0.4727
OTHER	0.1100	0.5579

	pr(INSTRS)	TAKEN	$\overline{\text{TAKEN}}$	TOTAL
Conditional	0.1652	0.3527	0.5386	0.8913
Unconditional	0.0201	0.1081	0.0006	0.1087
Total	0.1853	0.4608	0.5392	1.0000

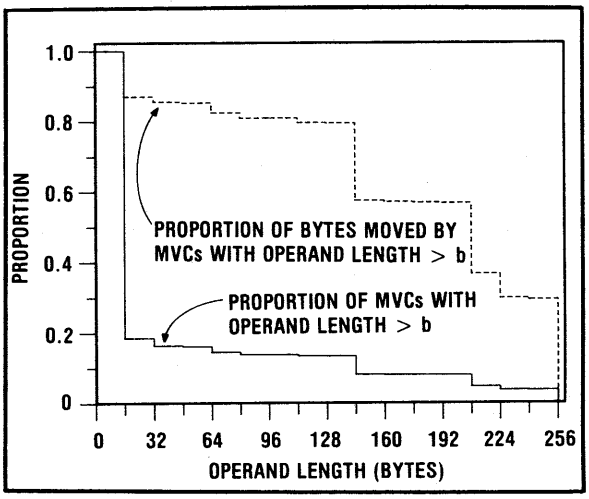


Figure 4. Proportions of Move Character Instructions (MVCs) and bytes moved versus operand length in a Cobol execution instruction mix.

Taken branches divide the instruction stream into a series of disjointed execution sequences. An execution sequence is a set of sequentially located instructions beginning with the target of a taken branch and ending with the next taken branch. The distribution of execution sequences is of interest in fetching pipelined instructions and designing instruction buffers. Figure 5 shows the distribution of execution sequence lengths for the SP1.3b operating system instruction mix. While the mean execution sequence length is approximately 7.5 instructions, 70 percent of the sequences are seven or fewer instructions long, and almost eight percent are one instruction. Sequences of one instruction represent a taken branch to a taken branch; they may result from the use of branch vector tables by the system or from procedure call protocols, for example. The high proportion of short sequences means that the taken branch "hole" must be made as small as possible to maintain pipeline efficiency. On the other hand, more instructions are executed in the long sequences. Figure 6 shows the proportion of sequences of length greater than s with the proportion of total instructions represented by sequences of length greater than s. While more than 70 percent of the sequences in this mix are shorter than the mean of 7.5 instructions, sequences greater than the mean account for more than 65 percent of all instructions executed. Thus, while frequent branches are of concern to the pipline designer, they do not prevent a pipelined design from being effective. (Kobayashi[1] discusses the characteristics of instruction sequences in greater detail.)

As a final example of the types of data composing an instruction-mix description, one kind of interinstruction dependency is shown in Table 4. There are several kinds of interinstruction dependencies; an address generation dependency exists when the result register of one instruction (the setting instruction) is used to form the operand address or target address of a following (using) instruction. When this situation occurs, operand or target address generation for the instruction must be delayed until the contents of this register become available, which creates a "hole" in the instruction pipeline. For the Cobol execution instruction mix, an address generation dependency exists for almost 12 percent of instructions; about 88 percent of these dependencies are caused by Load (L) or Load Register (LR) instructions. In this instruction mix, about two thirds of the Load instructions are used to load an index or base register for operand or target address generation. One frequent case is the sequence L-BCR, where the load sets the target address for the RR-type branch.

Using knowledge of what causes dependencies, the pipeline designer can reduce, and in some cases eliminate, address generation delays by incorporating bypasses at appropriate points in the pipeline. A bypass is a data path that routes a value to the address generator from a stage in the pipeline earlier than that at which the value is actually stored in its destination register. Software designers need to consider the impact of interinstruction dependencies in code optimization. Rymarczyk[2] discusses this and related topics.

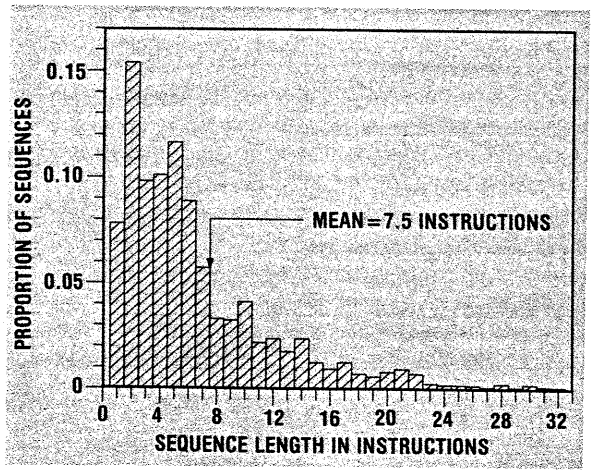


Figure 5. Distribution of execution sequence lengths for the SP1.3b operating system instruction mix.

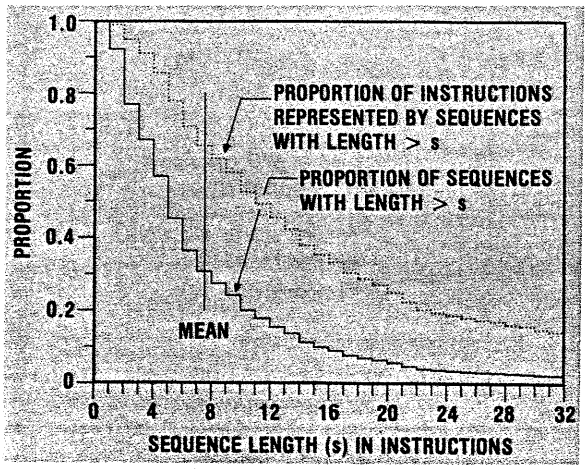


Figure 6. Proportions of sequences and instructions executed versus execution sequence length.

Models. The static model of program behavior represented by instruction mix descriptions is useful for answering questions about the execution of individual instructions and for analyzing some aspects of instruction interaction. Performance questions dealing with the concurrent execution of instructions and interactions among processing units are generally analyzed in relation to dynamic models. These range from simple analytic models used to "size" queueing delays at various points in the system (bus and memory conflicts) through a hierarchy of trace-driven simulation models that, as the design progresses, approach the register-transfer level very closely. Cycle-by-cycle simulation of a large set of programs, each represented by a trace of several million instructions, requires a substantial amount of computation time. There are a variety of ways to reduce these requirements, including decomposition, multilevel simulation, and trace reduction and synthesis. The full program set can be completely simulated only at points in the design process that require overall, absolute predictions of processor performance. While a variety of models may be used to analyze different aspects of performance, all resulting estimates are combined in the instruction-level model discussed below.

An instruction-level performance model

Performance measures. The common measure of performance for 370-compatible systems is the mean instruction execution rate, usually stated in millions of instructions per second, or MIPS. As a measure of processor performance, instruction execution rate is often misused (and, consequently, manufacturers tend to specify processor performance in relative terms). On any particular processor, the execution rate of applications can vary greatly. Thus, it is not uncommon for a Cobol compiler to execute its instructions at a rate twice that of the object program it compiles. Minor changes within a given architecture may decrease instruction rate performance and increase (or, for that matter, decrease) system performance as measured by throughput. For example, IBM has extended the 370 architecture several times over the last few years by incorporating various operating system functions into the processor (usually via microcode). An instruction-rate comparison of different versions of the operating system is not valid. For an invariant instruction stream, the mean execution rate is a useful tool for comparing different processors or different processor designs.

Mean instruction execution time. The reciprocal of the mean instruction execution rate in MIPS is the mean instruction execution time in microseconds. This, in turn, is—at least for synchronous machines—the product of the processor cycle time C and the mean instruction execution time in cycles I. That is,

$$\text{MIPS}^{-1} = I * C \qquad (1)$$

Most of our analysis work uses the processor cycle as the unit of time, rather than microseconds or nanoseconds. Machine organization decisions made early in the design process establish a cycle-time objective, but the exact cycle time may not be determined until late in the design, when chip layout is complete and path timing can be analyzed. More important, however, is that machine designers tend to think in terms of cycles, while software designers are concerned with the total number of cycles required to perform some function. Conversion from instruction execution time to instruction execution rate is generally made only to compare performances of processors with different cycle times.

Decomposition of instruction execution time. The mean instruction execution time I is assumed to be the sum of three basic components, such that

$$I = E + D + S \qquad (2)$$

where E is the mean nominal instruction execution time, that is, the mean instruction execution time if the pipeline could be kept fully busy; D is the mean pipeline delay per instruction, representing pipeline "holes" caused by path conflicts, register use dependencies, and taken branch delays; and S is the mean storage access delay per instruction representing the delays when an instruction or operand is not found in the buffer and must be fetched from mainstore. E, D, and S are expressed in cycles per instruction.

Equations (1) and (2) define a simple macroscopic model of CPU performance: Each term of (2) is, in turn, defined by submodels of increasing detail.

Mean nominal instruction execution time. For a pipelined processor with serial execution, E is the mean number of cycles per instruction spent in the execute stage of the pipeline. (A processor with parallel execution units will have overlapped execution cycles and requires a more complex model of E than that considered here.) For a given instruction mix, E is the weighted sum of the nominal execution times of all instructions in the mix. If $f(i)$ is the relative frequency of instruction i (where i can be the rank of a given operation code in the mix), and $e(i)$ is the mean nominal execution time of that instruction, then

$$E = \Sigma e(i) * f(i) \qquad (3)$$

where $f(i)\ e(i)$ is the contribution of instruction i to the mean nominal instruction execution time of the mix. It is assumed that the nominal execution time of each instruction can be computed independently of execution or delay times for any other instruction. Each $e(i)$ represents a submodel of E. Many of these are trivial, such as $e(i) = 1$. Others, such as the move character instruction are more complex. The MVC instruction has two uses, moving and clearing, which have different operand length distributions and different frequencies of buffer line and memory page crossings. Consequently, a fair amount of arithmetic is needed to compute e for this instruction.

The complexity in computing e for the MVC instruction arises from the large number of subcases involved: Design analysis to determine the execution time for a given subcase generally is simple. For some instructions, it is the design analysis which is complex. To determine e for the 370's Start IO Fast Release (SIOF) instruction, central processor and channel processor designs and their intercommunication must be analyzed. For the Start IO (SIO) instruction, we also need to know the timing characteristics of the device controllers to which various proportions of I/O operations are directed.

The values of $e(i)$ computed from various instruction submodels are multiplied by $f(i)$ for the particular instruction mix, and the resulting products are summed to compute the mean nominal execution time for the mix, as illustrated in Table 5.

Mean pipeline delay time. There are a variety of situations in which the execution of an instruction in a pipelined processor may be delayed. For example, an instruction

Table 4. Address generation dependencies in a Cobol execution instruction mix: pr(dependent instrs) = 0.1179.

OP CODE	PROP. OF DEPENDENCIES CAUSED BY OP CODE	PROP. OF OP CODE CAUSING DEPENDENCY
L	0.7979	0.6684
LR	0.0816	0.2544
LA	0.0168	0.0867
A/SR	0.0550	0.4015
A/S	0.0342	0.2235
other	0.0146	--

may have to wait for an address or operand value to be computed by a preceding instruction, or an instruction's operation fetch may be blocked because the buffer is busy performing a store for a preceding instruction. Most such delays result from interactions between instructions, which can extend across several instructions. As instruction execution times decrease, these interactions can increase because instructions are executed more closely together; at the same time, the proportionate effect of delays on performance increases. To counter this effect, current pipeline designs incorporate a variety of delay reduction mechanisms, including multiple copies of registers, special data paths (bypasses), multiple instruction stream buffers, and various kinds of branch prediction algorithms. In software design, code optimization for pipelined processors includes loading address and operand registers as early as possible before their use, making the most frequently followed paths corresond to not-taken branches, for example. Rymarczyk[2] describes various design tactics.

Delay analysis is not simple. An instruction incurring a delay may be several instructions away from the instruction causing the delay, and potential delays may be partly or entirely masked by other delays or by the execution time of intervening instructions. A single instruction may be involved in several different delays; for example, a branch may have a potential wait-for-condition-code setting overlapped by a wait-for-computation-of-a-target-address-register-value, and there may be a delay between the execution of the branch and the execution of its target. Consequently, delay analysis involves examining a sequence of instructions in terms of their relative positions in the pipeline and determining exactly which resources and paths are used by each instruction as the sequence advances from stage to stage.

Delays can be divided into three classes: register access, buffer access, and branch/miscellaneous. A delay submodel is associated with each delay condition, the most important of which are

Table 5. Nominal instruction execution times in an IBM SP1.3 MVS operating system instruction mix.

RANK	OP CODE	f	e	e*f
1	BC	0.18532	1	0.18532
2	L	0.14447	1	0.14447
3	TM	0.06289	1	0.06289
4	ST	0.05122	1	0.05122
5	LR	0.04864	1	0.04864
6	LA	0.04645	1	0.04645
7	BCR	0.03035	1	0.03035
8	LTR	0.02903	1	0.02903
9	MVC	0.02226	4.667	0.10389
10	IC	0.01790	1	0.01790
11	LH	0.01765	1	0.01765
12	BALR	0.01647	1	0.01647
13	STM	0.01563	5.543	0.08664
•	•	•	•	•
•	•	•	•	•
•	•	•	•	•
104	SCKC	0.00001	4	0.00004
				E = Σe*f

- Register Access Delays

  GI (generate interlock) delay. This delay can occur when the result register of one instruction is an index or base register of a following instruction.

  EI (execute interlock) delay. This delay can occur when the result register of one instruction is an operand register of a following instruction.

- Buffer Access Delays

  PI (priority interlock) delay. This delay occurs when an instruction attempts to fetch an operand from the buffer at the same time a preceding instruction is storing an operand into the buffer; the store is given priority, and the fetch is delayed.

  SFI (store-fetch interlock) delay. This delay occurs when an instruction accesses a storage location being stored to by a preceding instruction; the fetch access is delayed until the store is complete.

- Branch and Miscellaneous Delays

  TFCH (target fetch) delay. This delay represents the pipeline "hole" that may occur between a taken branch instruction and its target.

  CCI (condition code interlock) delay. Conditional branches are almost always preceded by condition code setting instructions. Some of the latter can be implemented to set the condition code at an early stage in their execution, but some set the condition code in a late stage, causing a delay in the execution of the subsequent branch instruction.

  STIS (store in instruction stream) delay. This delay occurs when a store address falls within the address range of the instructions already in the pipeline and instruction fetch buffer; these instructions are discarded, and instruction fetching is reinitiated when the store is completed.

The submodels themselves may be further decomposed according to causing-instruction type and incurring-instruction type. For example, a key submodel of the GI delay is the load-branch instruction sequence.

Rough estimates of these delays can be obtained from analytic models using instruction mix description data. However, since the type, frequency, and length of delays depend very much on the details of a particular design, accurate estimates require a detailed model of pipeline operation driven by instruction traces. The computational cost of this modeling can be substantially reduced by trace reduction methods. For example, a program loop may generate repetitive sequences of trace records, each sequence representing one iteration of the loop. Frequently, these sequences differ only in operand addresses and values. From the standpoint of pipeline delays and, generally, execution times, the set of sequences can be represented by a single sequence and a loop count. Pipeline delays are typically one cycle, and rarely more than two or three cycles. Any individual pipeline delay contributes very little to the pipeline delay per instruction (D). Consequently, low-frequency sequences can be ignored without introducing much error in D. The net result is that a program trace comprising millions of instructions can be represented, for pipeline modeling, by a relatively

small number of instruction sequences totaling a few thousand instructions.

Delay estimates are reported in several different forms: by cause, by causing-instruction type, and by incurring-instruction type. An example of the last form is shown in Table 6. Here, f is the relative frequency of each op code, and d is the mean number of delay cycles incurred by all instances of that op code for all delay conditions. (D, the mean pipeline delay time for the instruction mix, is

$$\Sigma d(i) * f(i), \tag{4}$$

where i denotes the ith-ranked op code.) In analyzing software, execution and delay cycles are reported instruction by instruction so that delay cause and effect can be easily pinpointed.

Mean storage access delay time. A storage access delay occurs when an instruction or operand access references a line not in the buffer, and the pipeline must wait for the line to be fetched from mainstore. A reference to a line not in the buffer is called a *buffer miss,* or simply a miss. S is the product of the mean number of misses per instruction (miss rate) u and the mean delay per miss (mean miss penalty) M in cycles. That is,

$$S = u * M \tag{5}$$

The miss rate u and buffer hit ratio h are related, such that

$$h = 1 - u/a \tag{6}$$

where a is the mean number of buffer accesses per instruction; h increases as a increases; and a is (among other things) a function of the width of the path between the pipeline and the buffer. Everything else being equal, a system with a four-byte buffer path, for example, will have a higher hit ratio—but lower performance—than a system with an eight-byte buffer path. Even for the same path width, a can vary from one design to another because of differences in operand alignment handling. Consequently, the miss rate is a more revealing measure of buffer performance than the buffer hit ratio.

While E depends on individual instructions and D depends on instructions sequences, S—and especially u—generally depends on the overall system environment. Current large-scale processors typically have buffer sizes of 64K bytes, and that figure is likely to increase by a factor of four to eight in future systems, most likely in the form of a buffer hierarchy. (The Fujitsu M-380 already has a two-level buffer hierarchy with 64K bytes at the first level and 256K bytes at the second.) A typical application task may reference the equivalent (in buffer lines) of 20K to 30K bytes in an execution interval before control is switched (voluntarily, because the task issued an I/O or some other system request, or involuntarily, because of an interrupt) to another task. If control is quickly returned to the first task, it will find many of its lines still in the buffer. Conversely, if the delay before resumption of the task is long, (in terms of the life of buffer lines, a disk request service time is very long), the task will find few, if any, residual lines, and will spend a good part of its execution interval bringing lines back into the buffer. The miss rate realized by a particular task depends not only on its intrinsic characteristics (number of instructions executed between I/O and other system requests, buffer line locality, address reference patterns, etc.) but also on the characteristics of all other concurrently executing tasks. Thus, by itself, analyzing the buffer behavior of individual application programs does not provide sufficient basis to estimate miss rates in a system environment. For example, in the typical 370 environment, two thirds or more of all buffer misses occur in supervisor state. What we require is a system view of buffer behavior.

Although we are focusing on mean overall values of u and M, equation (5) can be further decomposed in terms of instruction, target, and operand reference behavior, such that

$$S = u(i) * M(i) + u(t) * M(t) + u(o) * M(o) \tag{7}$$

where $u(i)$, $u(t)$, and $u(o)$ are, respectively, the mean miss instruction fetch, branch target fetch, and operand fetch (and store) miss rates; and $M(i)$, $M(t)$, and $M(o)$ are the associated mean miss penalties. Generally, $u(i) < u(t) < u(o)$; a sequential instruction fetch is less likely to cause a miss than a branch target fetch, and the locality of operand references is usually less than that of instruction references. The mean miss penalties also differ, and each miss penalty can be further decomposed as

$$M(j) = m(j) + p(j) * Tr \tag{8}$$

where $j = i, t,$ or o; $m(j)$ is the cost of fetching the instruction, target, or operand line itself; $p(j)$ is the proportion of misses of type j requiring an entry to be made in the translation lookaside buffer (typically from 0.1 to .01); and Tr is the TLB miss penalty, which may include mainstore accesses for page table and/or segment table entries.

Major components of the *mean miss penalty M* include

- *Buffer miss processing time,* the time required by the buffer to determine that the referenced line is not present; issue a mainstore read request for the missing line and, if the incoming line is to be replaced, issue a mainstore write request for a modified line. Typically, the line to be written is buffered and sent to mainstore after the read is initiated.
- *Mainstore processing time,* including memory bank queueing and access times. Some bank contention is

Table 6. Pipeline delay times by incurring instruction in an IBM MVS SP1.3 operating system instruction mix.

RANK	OP CODE	f	d	d*f
1	BC	0.18532	1.029	0.1907
2	L	0.14447	0.428	0.0618
3	TM	0.06289	0.408	0.0256
4	ST	0.05122	0.580	0.0297
5	LR	0.04864	0.087	0.0042
6	LA	0.04645	0.313	0.0145
7	BCR	0.03035	1.437	0.0436
8	LTR	0.02903	0.773	0.0224
9	MVC	0.02226	0.982	0.0219
10	IC	0.01790	0.802	0.0144
11	LH	0.01765	0.625	0.0110
12	BALR	0.01647	1.601	0.0264
13	STM	0.01563	0.559	0.0087
•	•	•	•	•
•	•	•	•	•
•	•	•	•	•
				D = Σd*f

caused by I/O operations, but most memory traffic is generated by the central processor (so that M depends on u). Some buffer designs may permit as many as six memory requests to be in process at any time: operand fetch, operand prefetch, replaced operand line store, instruction fetch, instruction prefetch, and replaced instruction line store.

- *Buffer interference time,* which occurs when the missing line is sent to the buffer from mainstore. The originally referenced portion of the line may be directly sent to the pipeline so execution can continue, and the line then is stored in the buffer. The number of buffer cycles required for the store depends on the path width and line size; these store cycles may interfere with buffer accesses from the pipeline and cause pipeline delays. (Since these delays are a function of u, they are included in S rather than D.)

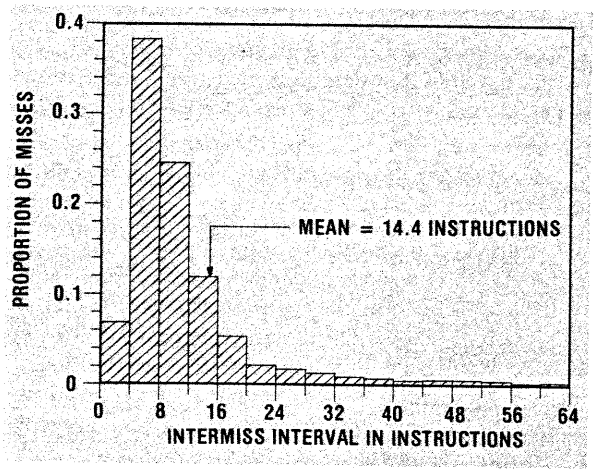


Figure 7. Intermiss interval distribution measured during supervisor state in a timesharing environment (without prefetch).

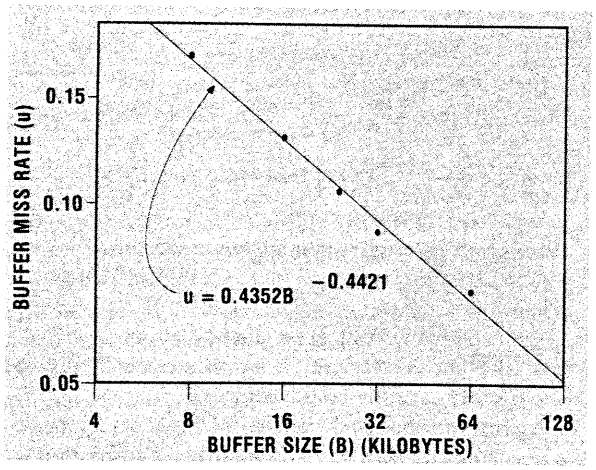


Figure 8. Buffer miss rate compared to buffer size during supervisor state in a batch environment (without prefetch).

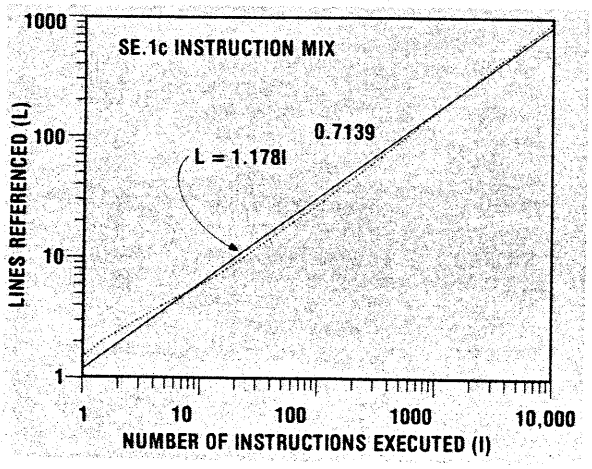


Figure 9. A stack growth function model.

Other contributors to M include, as noted earlier, TLB miss processing and, in a multiprocessor system, interprocessor communication to determine if a buffer line referenced by one processor exists in modified form in the buffer of another processor. Part of the time required to process a miss may be overlapped by the execution time of instructions preceding the instruction causing the miss. Consequently, the effective value of M may be less than the sum of these components; penalties for missing a sequential instruction fetch will usually be less than operand or target miss penalties.

Given the miss rate, proportion of modified lines, and pipeline, buffer, and mainstore operation details, we can obtain a reasonable approximation to M using analytic methods. However, because of the "bursty" nature of miss generation, the number of units involved (pipeline, buffer, and mainstore), and the high degree of concurrency in the processing performed by these units, we need a detailed simulation of the entire process to get an accurate estimate. Figure 7 shows the distribution of intervals between misses measured during supervisor state execution in a timesharing environment. The average interval between misses was greater than 14 instructions, but 45 percent of intermiss intervals were less than eight instructions long. This type of behavior causes designers to focus on miss response time, rather than simply buffer-mainstore bandwidth.

The buffer miss rate u depends on many buffer design parameters, including buffer size, line size, set size, and address mapping, replacement, and prefetch algorithms, for a specific workload. These parameters have traditionally been investigated with trace-driven models. However, it is extremely difficult to reproduce the reference stream seen by a buffer in actual system operation as control switches among various application and operating system tasks, so modeling results must be carefully interpreted. For example, investigations of line size based on application program traces suggest that long lines are more efficient than short lines. On the other hand, similar investigations based on operating system traces suggest that short lines are more efficient than long lines. The operating system tends to reference only a small portion of each line, so long lines dilute the effective capacity of the buffer. Since the system generates most misses, short lines are preferable; the final choice depends on the relative values of buffer-miss processing time and mainstore access time.

The different conclusions are due to behavioral differences. The typical batch application program tends to

have relatively strong spatial locality; line utilization (bytes referenced divided by line size in bytes) is high even with long lines, and a long line size expedites buffer reloading after a long suspension interval. The operating system is a collection of different tasks providing a variety of services and exhibits relatively little spatial locality either in its instruction or operand behavior: envision, for example, a long linked list search. Studies of other parameters, such as set size and address mapping, may also yield different conclusions, depending on the data used.

Because miss rate depends on system-level behavior, it is extremely difficult to model the relationship between miss rate and buffer size. For workloads and buffer designs that are similar except for size, empirical relationships between buffer size and miss rates have been determined through hardware measurement studies. These relationships have the form $u = a * B^{-k}$ (9) where B is the buffer size in kilobytes and a and k are constants for a particular workload or workload component. Typically, k is from 0.35 to 0.50, depending on the workoad. Figure 8 shows an example of equation (9) for the supervisor state component of a batch workload. Smith[3] provides a more elaborate form that includes problem state data and the effect of set size.

One approach to modeling this relationship (now under investigation) involves two submodels: a model of line reference behavior for individual tasks and a model of task switching behavior. The first model relates the number of unique lines referenced in a task's execution interval to the length of the interval in instructions. This model, called the stack growth function model,[4] is based on LRU stack behavior determined by instruction trace analysis. In many cases, an excellent fit to a trace-derived model is given by $L = r * I^d$ (10) where I is the number of instructions executed, L is the number of unique lines referenced by these instructions, and r and d are parameters for each program or task. Typically, d is from 0.35 to 0.65 for application programs and from 0.65 to 0.80 for operating system tasks. Figure 9 shows equation (10) plotted together with the trace-derived model for an operating system instruction trace. The task switching model being developed is based on data for a system event trace and gives transition frequencies for switches between both operating system and application tasks together with task execution interval lengths. The next step is to integrate the stack growth function models for individual tasks into the task switching model and validate the behavior of the composite model against system measurements.

While our discussion of buffer behavior has been from a processor design perspective, software performance can also benefit from an awareness of buffer organization and operation. Tactics are similar to those used for improving paging behavior—separation of modified and unmodified parts and aggregation of frequently referenced code and operands—but on a smaller scale.

Starting with mean instruction execution rate as a measure of instruction-level performance, we have traced several decomposition levels of a performance model (Figure 10), which encompasses a wide range of program-

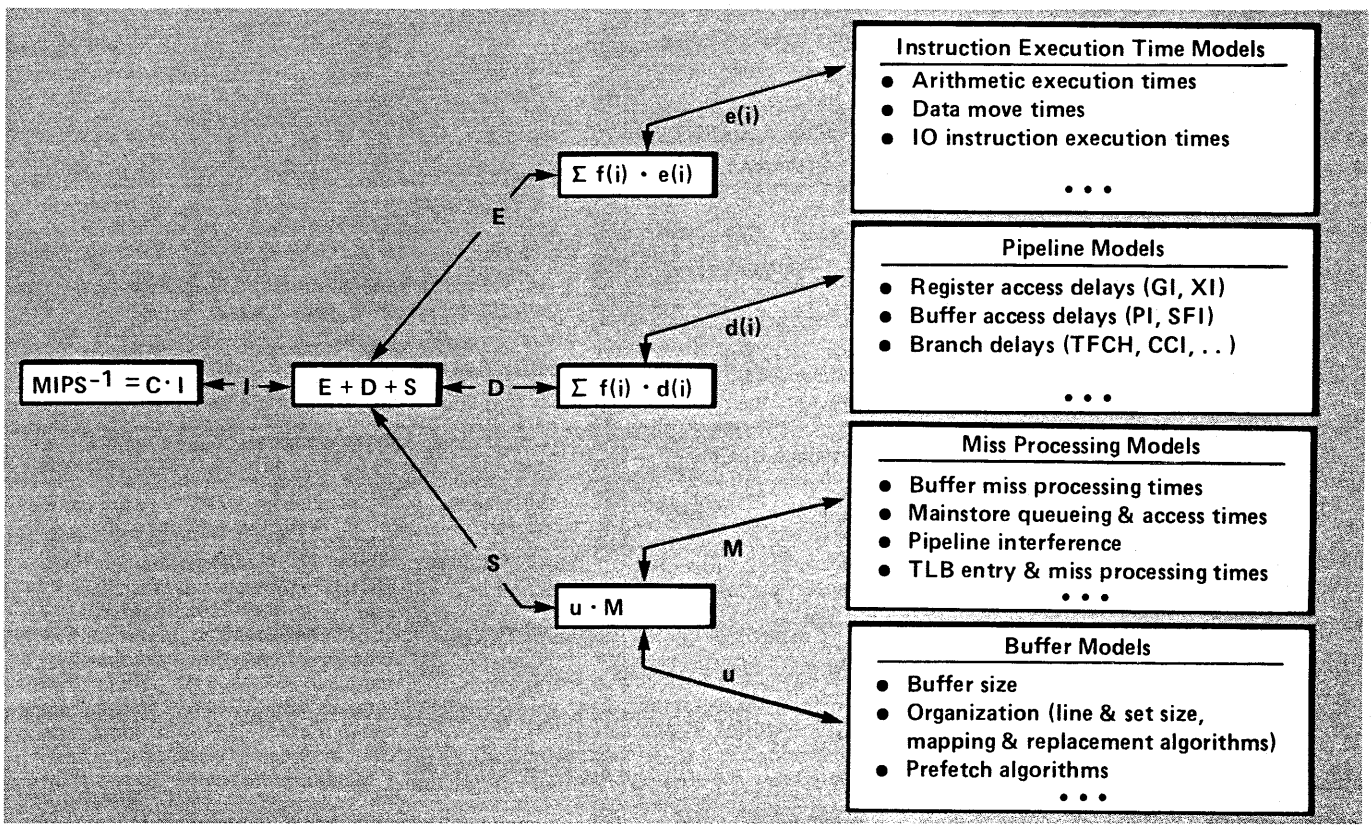


Figure 10. Summary of instruction-level model decomposition.

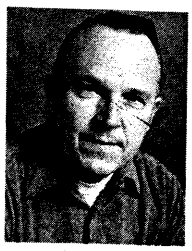
processor interactions. Although this model is oriented toward a specific architecture and a particular implementation view, the underlying concepts have a much broader application. A model of this kind can provide a taxonomy for identifying and describing performance components at successive levels of decomposition. It provides a consistent framework in which to incorporate estimates from different and possibly dissimilar submodels, including measurement as well as analytic and simulation models. When possible, its decomposition should follow lines of minimum component interaction. Decomposition is done not only in terms of processor design, but also in terms of program behavior. For current large-scale processors, performance depends on—literally—hundreds of different low-level interactions between program and processor, and we should be able to represent each of them at some level of the model. While performance analysis may be approached from a hardware or a software perspective, both require a performance model that is a composite of program behavior and processor design models. *

Acknowledgments

The work described in this article represents workloads, systems, tools, models, and methods developed over years. Many individuals have made important contributions to this effort and I feel privileged to describe some of their accomplishments.

References

1. M. Kobayashi, "Dynamic Profile of Instruction Sequences for the IBM System 370," *IEEE Trans. Computers,* Vol. C-32, No. 9, Sept. 1983, pp. 859-861.
2. J. W. Rymarczyk, "Coding Guidelines for Pipelined Processors," *Proc. Symp. Architectural Support for Programming Languages and Operating Systems,* 1982 (also in *Computer Architecture News,* Vol. 10, No. 2, Mar. 1982, pp. 12-19).
3. A. J. Smith, "Cache Memories," *Computing Surveys,* Vol. 14, No. 3, Sept. 1982, pp. 473-530.
4. M. H. MacDougall, "The Stack Growth Function Model," tech. report 820228-700A, Amdahl Corp., Sunnyvale, Calif., Apr. 1979.



Myron H. MacDougall is director of Systems Performance Architecture for Amdahl Corporation. Prior to joining Amdahl in 1976, he worked for 12 years at Control Data Corporation. He began working with computers at RCA in 1959. His interests encompass all aspects of computer performance analysis and include a long-term interest in system-level simulation.

MacDougall attended Rutgers University. He is a member of the ACM and IEEE Computer Society. His address is Amdahl Corporation, M/S 139, PO Box 470, Sunnyvale, CA 94086.