Proc. 4'th Annual Symposium on Computer Architecture, March 23,-25, 1977.

AN INSTRUCTION TIMING MODEL OF CPU PERFORMANCE *

Bernard L. Peuto +
Zilog, Inc.
Cupertino, California

and

Leonard J. Shustek
Stanford Linear Accelerator Center
and
Computer Science Department
Stanford University
Stanford, California

Abstract

A model of high-performance computers is derived from instruction timing formulas, with compensation for pipeline and cache memory effects. The model is used to predict the performance of the IBM 370/168 and the Amdahl 470 V/6 on specific programs, and the results are verified by comparison with actual performance. Data collected about program behavior is combined with the performance analysis to highlight some of the problems with high-performance implementations of such architectures.

Introduction

General Goals

One of the most important tasks for a computer designer is the evaluation of a computer architecture and its implementation. As two specific instances of that task, we consider (1) a comparison of the performance of the IBM 370/168 Model 1 and the AMDAHL 470 V/6, which are two machines with the same architecture but different implementations, and (2) an analysis of some of the properties of the IBM 370 instruction set.

The basic goal is to apportion the time spent by an executing program among the various system components such as the cache memory, the instruction pipeline and the individual instructions, so that resource utilization and system bottlenecks will appear. This is achieved by using models of the CPU of each machine which also provide estimates of the total CPU times. The total time is important insofar as it is used to verify the accuracy of the model, since the predicted times are compared to the actual performance of the machines.

The decision to make implementation dependent measures of CPU performance for two members of a specific architecture family has several advantages: (1) Some of the traditionally difficult problems encountered when comparing two different architectures are not present, since many confounding factors relating to performance evaluation have the same effect on both machines. (2) The success of one of the levels of a complex system can often be measured by the characteristics of the levels below. Performance evaluation which is close to the implementation level of a computer gives valuable design information at the architecture level. (3) The speed of collection and the precision of the results are greatly enhanced by having tools that are tailored for a specific instruction set. (4) Practical and useful results can be obtained quickly, paving the way for more general studies.

* Work supported in part by the U.S. Energy and Research Development Administration under contract E(043)515.

+ Work done while a Visiting Scientist at the Stanford Linear Accelerator Center.

Previous Studies

The evaluation of computer systems from the buyer's point of view has traditionally received a great deal of attention. The system software often requires careful and tender tuning, and bottlenecks which can have dramatic effects on performance must be identified and removed. An abundant literature addresses these problems and provides techniques for solution [AGA75].

The computer system designer has similar problems to solve, but most of the existing literature is not written for his viewpoint. One explanation for this phenomenon is the lack of feedback; users seldom complain about hardware design because they feel that their complaints will have little effect. The result is a scarcity of information for use by the designer. Most of the studies closest to this work deal with the collection of data on instruction frequencies. The most frequent objectives involve (1) benchmark studies, (2) computer design, (3) language design, and (4) general programmer curiosity.

Some studies leave all interpretation to the reader, and become a useful source of primary data [GIB, CON]. The studies most applicable to the computer designer's point of view often provide instruction frequencies, register utilization, opcode pairs, and static vs dynamic frequency comparisons, but little timing or performance information [LUN, FLY, WIN, HAN, AGA73, ANA, FOS71ab]. The language-oriented studies have provided similar information for specific languages, studying the match between the language and the machine code to which they must be translated [ALE72, HEN, ALE75].

When their interest is only in performance evaluation, users have generally been advised to use benchmark runs instead of instruction mixes based only on instruction frequencies. [ARB,SNI]. The use of timing information with these instructions mixes is made difficult by the lack of published information from the manufacturers, in particular for the high-performance machines. (Amdahl is an exception in this regard [AMD]). This has forced users to produce their own documents [LIP, EME]. The manufacturers themselves must have studied these questions, and some expurgated papers reveal glimpses of large-scale

efforts and sophisticated tools but offer little results [VAN, HUG, MUR].

The previous studies have shown that very few instructions (often four or five) represent 50% of those executed, and a few more (often 20 to 30) represent 90%. This would seem to justify the idea that a few instructions will account for most of a program's behaviour and one can neglect instructions whose frequencies are below a certain threshold. Unfortunately this applies only to a specific program. No trend has been shown in the importance of instructions, because the instructions which make up the 50%, 90% and 100% groups of a program are dependent on the program, the programmer, and the language used. The only instructions which seem universally important are the branches, which most often account for about 15-30% of the instruction counts, but which still show wide variation.

The difficulty with the frequency analysis approach is that for performance evaluation the designer needs information about the instructions which account for most of the execution time. Attempting to derive performance conclusions from an instruction frequency list yields poor results because some instructions can hundreds of times slower than others. To obtain acceptable performance results the designer needs to consider machine dependent variables because they are required for precise evaluation of the instruction execution time.

The Instruction Timing Model

The Methodology

The models of the CPUs used here are based on the instruction timing formulas available from the manufacturers' documents which describe their computers [AMD,IBM]. These documents sometimes sacrifice details for ease of exposition (which is not to say that they are easy to read!) and represent only the best efforts of an engineer to describe the existing machine. (In deriving the model for the Amdahl machine we were quite fortunate to get some help from the designers.)

The programs to be measured were traced in user state, and all the information required to compute the instruction execution time from the formulas was collected. A record was made of counts of occurrences, values of instruction variables used in the formulas, and information about memory performance. Typical variables depend on the specific instruction but may also depend on the implementation details. For example, the number of bytes moved is implementation independent, but measures of pipeline interlocks and timing delays are not. Some variables depend on instruction environment and therefore require information about instruction pair and triple distributions.

Two primary constraints caused us to trace only user-state instructions. (1) Tracing system software, with the attendant performance degradation of at least 50 to 1, would modify operating system behavior in timing dependent I/O sections. By tracing only in user mode, which is basically not speed dependent, we eliminate a source of error which would necessitate a complicated interpretation of the results. (2) Tracing the operating system introduces a large number of problems involving the recording of the trace data. One standard solution is the use of samples rather than complete traces, but then the verification of the predicted CPU time is nearly impossible.

Since the timing formulas do not include the effects of cache memory misses, the cache memory is simulated for each machine. The cache penalty is added to the instruction execution time to obtain the expected program execution time. To verify the model the expected time is compared to the operating system accounting time corrected to compensate for the differences between the measurement methods.

The effects of instruction interaction, which can generally be attributed to pipeline resource interlocks, are rather explicitly accounted for in the Amdahl formulas. For IBM, however, the pipeline effects seem to have been averaged into the formulas in a way which was not clearly indicated. This was a potential source of difficulty, but the effort required to obtain this information from the logic diagrams and microcode listings was prohibitive, and unjustified when an error of a few percent is acceptable.

The techniques used here are much more complex than benchmarking, but not as costly as total hardware simulation. The tools are general enough so they can be -- and have been -- used for other studies. The importance, however, lies in the ability to change the model variables to reflect proposed changes to the existing hardware and to accurately predict the performance effects of those changes.

Choice of Factors

The development of the CPU model has been greatly influenced by the idea of an evolving system of tools -- development by successive refinement. A crude model and simple tools were first assembled and by successive iteration new tools, new measurements, and a more refined model were designed. We think this approach reduced the number of false starts and the elapsed time of the whole study by allowing us to concentrate quickly on the most important factors.

The CPU model used is an intermediate one between full simulation at the hardware register level and a machine-independent representation of performance. The decision to include some factors and exclude others was based on our estimation, often supported by experimentation, of the effect of those factors on the final results. Some of the justification for the decisions are presented below.

The accuracy of the model is supported by the match between the program execution time as predicted by the model and the same time measured by the operating system during actual runs. Performance evaluation by benchmarking is repeatable only within 2-3% because of the large number of uncontrollable variables, and this therefore defines the required precision of the model.

An examination of previously published instruction frequencies might suggest that the more frequent instructions are those whose duration is constant and therefore do not heavily depend on execution variables like the length of operands. If this were true, then those variables could be set to program-independent values without introducing a significant error in the result. To test this hypothesis, the program which computes execution times was given three sets of execution variables with which to predict program running time. One was a programmer's best guess of the true values, and the other two were the smallest and largest extremes which could realistically be expected. The results showed that an instruction could jump from 4% to 50% of the total time depending on the value of its variables with all others remaining the same. This is an unacceptable error, especially since errors in the variables for many instructions could combine to form large systematic errors. Most of the variables which affect execution time were therefore measured

exactly or estimated from related measurements.

The predicted execution time is composed of the aggregate instruction timing results and a penalty for cache memory misses. The aggregate instruction timing results have already taken into account the instruction counts and basic execution speed, as well as the pipeline interlocks. The cache miss penalty depends on the reference pattern of the program, the cache organization, and the data flow pattern within the machine. The two machines differ rather markedly in those respects: the 370/168 uses aligned doubleword (8-byte) accesses and an associative set size of 8, while the 470 accesses unaligned fullwords (4-bytes), uses a set size of 2, but has the same total amount of data (16K bytes). There are also rather significant differences in the amount and type of instruction lookahead performed. To accurately measure the cache penalty, the trace analysis program has a detailed simulation of the cache and instruction fetch mechanism of both machines.

Although cache memory miss ratios are known to be low [MER], it is easily shown that the contribution of the time penalty for the misses is too large to be neglected. If the miss ratio is 5%, with a 480 nsec penalty for a miss, 2 memory requests per instruction, and an average instruction execution time of 300 nsec (reasonable values for the 370/168) then the time for the cache misses represents 16% of the execution time.

Two other cache organization features must be considered in the cache penalty correction. For IBM, stores always access main memory ("store-through") which may cause extra delays. For Amdahl, there is an extra penalty when a 4-byte access crosses a cache line boundary. These and the other cache corrections are not attributed to the instructions which caused them, but rather accumulated separately.

The execution time reported by the operating system includes all user-state and some supervisor-state instructions [BEN], whereas the trace program measures only user-state instructions. The time attributed to these supervisor-state instructions executed in the processing of user-initiated supervisor calls (SVCs) must be subtracted from the reported CPU time. Measurements were made of the charged time for all the relevent SVCs as the programs were traced. The correction is very significant for almost all programs, since both the number and cost of the SVCs are high. For the 168, for example, the time charged varies from 107 usec for an I/O operation to 26 msec for opening a file.

Although the SVC time correction could have been measured for the original benchmark programs, they were somewhat modified in view of the substantial correction required (as much as 20%). Wherever possible, the number of I/O operations was reduced by increasing the file blocking factors, but we did not otherwise alter the operation of the programs. Despite this effort, the SVC time correction remained the factor which introduced the largest error in the measurements. We also added a FORTRAN numerical analysis program from which the I/O parts were excised, so that few supervisor services were requested.

Since supervisor-state and user-state instructions share the same cache, there will be some displacement of the user's "working set" from the cache in response to an SVC, which will manifest itself as a lower than normal hit ratio when the user's program is resumed. An unpublished note by Rossman suggested that this would have a significant effect [ROS]. To verify this we simulated the cache activity for one job with a large number of SVCs first assuming a 100% cache flush for each SVC, and then again with no flush; the number of cache misses changed by a factor of 10. Measurements showed that the actual fraction of the cache displaced by an SVC varies from 0.16 to 1.0, and that almost all non-trivial requests completely replace the cache.

Interrupts which occur during the execution of the program do not account for a significant increase in accounted time (since the user-state CPU timer is disabled during interrupt processing) but there could be an effect due to cache displacement caused by the interrupt routine. On a heavily loaded machine interrupt rates as high as 4000 per minute are common, representing 16.4 ms of extra time (1.7% for IBM) to completely refill the cache for each second of CPU time. Since most of those interrupts are due to other jobs, this effect was reduced to a negligible level by running the job on on otherwise idle system, so that only the few interrupts caused by the benchmark job itself could cause interference. This is unlike the SVC correction, for which no change in the number of cache flushes is possible simply by controlling the environment of the benchmark run. Similar calculations for the effect of channel I/O transfers to memory show that they have even less effect on CPU performance. This is true both for IBM, where the channels transfer directly to main memory and invalidate corresponding cache entries, and for Amdahl, where the channels transfer into the cache.

Overview of the Measurement Programs

An interpretive trace program (TRACE) generates a record for each user-state instruction of the measured program. The record contains the instruction type, memory addresses referenced, and the other required information. These records are processed by a trace analysis program (ANALYSIS) which generates instruction counts, variable values, and memory access statistics such as cache memory miss counts, which are stored in a summary file. In order to avoid saving massive amounts of intermediate trace information (25 megabytes per traced second), the TRACE and ANALYSIS programs execute as coroutines. The combined overhead of the trace and trace analysis programs amounts to 300 seconds per second of real time. This compares favorably to other more detailed hardware simulations, where the overhead has been as high as 6000 seconds per second of real time [VAN].

The summary file is converted into a count file by an intermediate program (CONVERT). The count file contains all the information required to compute the timing formulas for both machines condensed into about 500 numbers. An instruction statistics program (INSTAT) uses the count file and files of encoded instruction timing formulas to produce the final timing and performance information.

We devised several test programs for verifying the formulas and understanding the measurement factors. A general instruction timing program (LTIMER) was designed for precise measurements of instruction times, cache memory miss penalties, SVC times, and the effects of SVCs on cache memory contents.

The Instruction Timing Formulas

An instruction may have several timing formulas associated with it, corresponding to different modes of execution. Each individual timing formula may depend linearly on the variables (the most common case) or have a more complicated dependence. In general, three types of linear formulas are encountered.

Some timing formulas reduce to a constant, and

often only one formula is associated with an instruction. Examples of this case are most register-to-register arithmetic or logical instructions.

ADD REGISTER (AR)	IBM	.080 usec
	Amdahl	.065 usec

Many formulas have a simple linear dependency on execution variables. An example is a Load Multiple (LM) instruction which can be expressed as

Load Multiple (LM)	IBM	.520+.080*R usec
	Amdahl	.065+.065*R usec

where R is the number of registers loaded.

Some formulas may involve variables which are concerned with the general environment of the instruction. These are often measures of the effect of pipeline interference which causes a delay in the execution of an instruction. Examples are the Amdahl variables S1 and DWD. S1 accounts for some cases of pipeline interlocks, and ranges from 0 to .065 usec depending on the "number of execution cycles attributable to the three words of the instruction stream following the instruction of interest" [AMD]. DWD, which is either 0 or .0325 usec, compensates for the occurrence of a doubleword result instruction before the subject instruction, because the machine is fundamentally single word oriented.

Store (ST)	Amdahl	.065+S1+DWD

When several formulas are associated with one instruction, each formula applies only to a specific case of its execution. For example, the Move Character instruction execution formulas depend in important ways on the degree of overlap of the two operands. The different cases involve not only different coefficients, but often different variables.

Move Character (MVC)	IBM	.760+.040*B usec	(no overlap)
		.640+.240*B usec	(any overlap)

Amdahl .195+S1+.130*WB+MV usec

where MV = .130*W (no overlap, or overlap>32 bytes)
MV = .1625*W (3<overlap<=32 bytes)
MV = .130*B (1<overlap<=3 bytes)
MV = .195*B (overlap=1 byte)

and where B = number of bytes moved
W = number of words moved
WB = number of bytes which must be moved to have the destination on a word boundary when b>63.

For all the individual linear formulas, we need only accumulate the counts and average variable values for each of the timing formula cases.

Unfortunately, some formulas are not linear in their variables. Typical examples are the decimal arithmetic instructions, where the duration depends on the product of the lengths or the average value of the digits used. For these we compute the appropriate products of variables at the time the program is analyzed, and average these values for use by the other programs in an equivalent linear form. These cases of non-linear formulas are sufficiently infrequent to justify this special treatment, but the effect on timing values is too important to ignore them. A simpler approach would assume that the product of the averages is a sufficient estimate of the average product, but the potential error is great.

The formulas are encoded as a string of records, each corresponding to the coefficient of a term in a subcase of a timing formula for a particular instruction; there are a total of 3200 variable names and coefficient values. A numbering and naming scheme was devised that allows variables which are common to many formulas to be propagated to all appropriate places, as well as giving individual identities to variables which are more specific.

Verification of the Model

Measurement of Cache Miss Penalty

Although cache miss penalty information is available from the manufacturers, it was difficult to interpret precisely what the effect on instruction time is. Since measurements are not difficult and the correction could be significant, the values were verified experimentally. To determine the cost of a cache miss, a test program simply fills the cache with known data. A second loop is then timed, in which either the same data is reloaded, or new data displaces the old. The difference in time between the two versions of the second loop, divided by the number of cache misses caused by the loop which displaces the data, provides the cache miss time. The value found for IBM is 480 nsec, which is not inconsistent with information from the hardware manuals. For Amdahl, cache misses are found to cost 650 nsec, which also agrees with information from the designers.

Once the cache miss penalty is established, the effect of a supervisor request on the user data in the cache can be measured easily. In a similar fashion the cache is filled with known data, the SVC is issued, and the cache is refilled with the same data. The second loop is timed, and compared to the identical loop when the SVC is not present. The time difference divided by the cache miss penalty gives the number of cache lines that were displaced by the SVC. Note that the second loop must fill the cache in the opposite order from the first loop, otherwise the LRU replacement algorithm would cause the original data to be removed instead of the data added by the SVC. Table 1 shows the fraction of cache displacement for some of the more common supervisor requests.

****** TABLE 1 — SVC TIMES AND CACHE EFFECTS (AVERAGED FOR ALL PROGRAMS)

	IBM		Amdahl	
Name	CPU time usec.	% cache displaced	CPU time usec.	%cache displaced
OPEN	26658	100%	17605	100%
CLOSE	16929	100%	13488	100%
EXCP I/O	107	58%	101	24%
WAIT	234	16%	139	7%
REGMAIN	394	30%	219	17%
LINK	3629	100%	1613	41%
OVERLAY	5214	100%	N/A	N/A

One of the most interesting differences of implementation between the two machines is the effect of data stores on the cache. The IBM approach is to always store data directly into main memory, and to update the cache only if the line already exists. The Amdahl machine updates the cache line if the data is present without storing into main memory. If the data is not in the cache, the line will be read from memory. If the replacement algorithm must remove a line which was modified in the cache, the memory is updated at the time the line is replaced. The IBM method, called "store-through", has often been criticized because it

requires a main memory access for all stores [KAP]. Although the store can proceed in parallel with subsequent instructions, any subsequent main memory accesses must be suspended until the memory becomes available. Since the timing formulas do not explicitly account for this effect, it is important to determine its magnitude.

There are three factors which combine to minimize the possible deliterious effects of the store-through policy used by IBM. The first is that the memory is organized with four-way interleaving of adjacent doublewords, so that consecutive stores may well reference separate memory banks. The second is simply that based on the opcode pair distribution we have accumulated, consecutive instructions which store data into memory are relatively infrequent. The third is that even for pairs of such instructions, there appears to be a level of buffering for data that must be written to main memory, at least for the case when that data is also in the cache. A penalty appears only for the third consecutive store, and then is 360 nsec. The full write cycle time penalty of 640 nsec occurs only for the fourth and subsequent store. These factors are sufficient to justify not including a difficult-to-compute correction for store-through writes.

SVC Time Measurement

As previously discussed, the CPU time charged for SVCs was measured in order to be able to correct the time given by the operating system. The time charged for each SVC is often large and varies from program to program even for the same SVC type. To account for these variations we measured the time charged to the user for each SVC as the benchmark programs were being traced. The SVC correction computed by summing the measured SVC times is therefore quite accurate for the 168 because it was the machine used for the tracings. For the 470, the timing program LTIMER was used to give estimates of the average SVC costs. This latter method does not take into account the variation from program to program and the SVC corrections are much less accurate than for the 168. Table 1 shows the time charged for some important SVCs averaged over all programs.

It is interesting that the time charged for supervisor services is often comparable to what would be required if there were no operating system. For I/O operations, previous measurement have shown that the hardware I/O instructions (SIO, TIO, etc.) are incredibly expensive; 100 usec is not unusual [JAY]. This is to be compared with, for instance, the measured charge of 107 usec for the request to the operating system for an I/O operation. Note that both of these are more than two orders of magnitude larger than, for example, the 0.61 usec needed for a double precision floating point multiplication. It would seem that improvements in the arithmetic units of computers have not been accompanied by similar improvements in the I/O interface despite the existence of I/O channels.

The Benchmark Jobs

The results presented here are derived from the analysis of seven benchmark jobs written at SLAC. Except for one (LINSY2) they were all production jobs written for purposes other than performance evaluation. To avoid biasing the results with artifacts from specific languages or programs, we purposely chose the three most used language compilers and programs compiled by them.

(1) FORTC is a compilation by the IBM Fortran-H optimizing compiler.

(2) FORTGO is the execution of the FORTRAN program compiled by FORTC. It is a numerical analysis program which solves partial differential equations.

(3) PL1C is a compilation by the IBM PL/I-F compiler.

(4) PL1GO is the execution of a PL/I program which accumulates and prints accounting summaries from computer use information.

(5) COBOLC is a compilation by the IBM ANSI Standard COBOL compiler.

(6) COBOLGO is the execution of a COBOL program which reformats and prints computer use accounting information.

(7) LINSY2 is the execution of a FORTRAN subroutine which solves large-order simultaneous equations. No I/O is done.

Table 2 summarizes some characteristics of the benchmark jobs.

****** TABLE 2 — PROGRAM CHARACTERISTICS

Program	# Instr.	Data reads per inst	Data writes per inst	Inst/Cache Miss IBM	Amdahl
COBOLC	6,048,476	0.431	0.130	82.57	36.95
FORTGO	23,865,168	0.352	0.204	104.06	28.07
PL1GO	23,863,497	0.473	0.261	73.28	61.16
LINSY2	11,719,853	0.195	0.067	20597	19598
COBOLGO	3,559,533	0.738	0.453	13.42	30.93
FORTC	17,132,697	0.433	0.146	39.86	24.47
PL1C	24,338,101	0.379	0.137	145.33	63.48

Model validation

Verification basically consists of comparing the time predicted by our model for each benchmark job with the corrected real execution time. The time predicted for each benchmark, Tpred, consists of the following terms:

Tins, the total time predicted from the timing formulas, which does not include the cache miss penalty.

M * Tmiss, where M is the number of cache misses as reported by the cache simulator, and Tmiss is the cache miss penalty. The number of cache misses includes the effect of SVC execution on the cache contents.

Tcross, the time penalty, for Amdahl only, paid when references to the cache cross a line boundary. The penalty is two cycles (.065 usec) for reads and three cycles (.0975 usec) for writes, and is computed using numbers provided by the cache simulator. Virtually all the penalty arises from instruction fetch, since none of the programs access unaligned data. There is no equivalent penalty for IBM because its larger instruction buffer prefetches enough so that two successive doublewords can be accessed without introducing an additional delay.

The corrected time for the actual execution, Trun, consists of the following terms:

Tacc, the time as given by the standard IBM accounting routines.

Tsvc, the time attributed to the user for the execution of all the supervisor calls, which must be subtracted from Tacc.

Table 3 provides the values for each of these times for each of the benchmarks. For Tpred and Trun, the relative percentage of each of their components is given. The absolute error, Trun-Tpred, and the percent error, (Trun-Tpred)/Trun, appears on the last lines. The verification process points to large discrepancies between raw execution speed (Tins) and the speed as perceived by the user (Tacc).

The results for IBM are generally extremely good; for all except one program the differences between the predicted and actual running time are less than 2%. The agreement for Amdahl is not as good, but we attribute most of the error to the crude method for measuring the SVC time correction. A factor of two in the the SVC correction, which is certainly conceivable when an OPEN as measured on the 168 can vary from 6 to 33 msec, could easily account for all the the error.

****** TABLE 3 -- MODEL AND BENCHMARK TIMES

COBOLC	IBM Time	IBM %	Amdahl Time	Amdahl %	RATIO IBM/Amd
Tins	2.213	98.44	1.179	88.45	1.878
M*Tmiss	.035	1.56	.106	7.95	.330
Tcross			.048	3.60	
Tpred	2.248	100.00	1.333	100.00	1.686
Tacc	2.57	100.00	1.71	100.00	1.503
-Tsvc	.348	13.54	.320	18.71	1.088
Trun	2.222	86.46	1.390	81.29	1.599
Trun-Tpred	-.026		-.057		
% error	-1.170		-4.101		

FORTGO	IBM Time	IBM %	Amdahl Time	Amdahl %	RATIO IBM/Amd
Tins	6.176	98.25	3.286	83.81	1.879
M*Tmiss	.110	1.75	.553	14.10	.199
Tcross			.082	2.09	
Tpred	6.286	100.00	3.921	100.00	1.60
Tacc	6.42	100.00	N/A		
-Tsvc	.082	1.28			
Trun	6.338	98.72			
Trun-Tpred	.052				
% error	0.82				

PL1GO	IBM Time	IBM %	Amdahl Time	Amdahl %	RATIO IBM/Amd
Tins	4.561	96.69	2.233	85.88	2.042
M*Tmiss	.156	3.31	.254	9.77	.614
Tcross			.113	4.35	
Tpred	4.717	100.00	2.600	100.00	1.814
Tacc	5.45	100.00	3.42	100.00	1.594
-Tsvc	.293	5.38	.206	6.02	1.422
Trun	5.157	94.62	3.214	93.98	1.604
Trun-Tpred	.440		.614		
% error	8.53		19.10		

LINSY2	IBM Time	IBM %	Amdahl Time	Amdahl %	RATIO IBM/Amd
Tins	1.970	100.00	1.561	96.48	1.262
M*Tmiss	.000	0.00	.000	0.00	1.000
Tcross			.057	3.52	
Tpred	1.970	100.00	1.618	100.00	1.218
Tacc	1.98	100.00	1.69	100.00	1.172
-Tsvc	.040	2.02	.031	1.83	1.290
Trun	1.940	97.98	1.659	98.17	1.169
Trun-Tpred	-.030		.041		
% error	-1.55		2.47		

COBOLGO	IBM Time	IBM %	Amdahl Time	Amdahl %	RATIO IBM/Amd
Tins	4.291	97.13	2.451	95.67	1.751
M*Tmiss	.127	2.87	.075	2.93	1.693
Tcross			.036	1.40	
Tpred	4.418	100.00	2.562	100.00	1.724
Tacc	4.82	100.00	2.92	100.00	1.651
-Tsvc	.428	8.88	.289	9.90	1.481
Trun	4.392	91.12	2.631	90.10	1.669
Trun-Tpred	-.026		-.069		
% error	-0.59		2.62		

FORTC	IBM Time	IBM %	Amdahl Time	Amdahl %	RATIO IBM/Amd
Tins	3.711	94.74	1.886	77.62	1.968
M*Tmiss	.206	5.26	.455	18.72	.452
Tcross			.089	3.66	
Tpred	3.917	100.00	2.430	100.00	1.612
Tacc	4.64	100.00	3.10	100.00	1.497
-Tsvc	.652	14.05	.430	13.87	1.62
Trun	3.988	85.95	2.670	86.13	1.494
Trun-Tpred	.071		.239		
% error	1.78		8.95		

PL1C	IBM Time	IBM %	Amdahl Time	Amdahl %	RATIO IBM/Amd
Tins	7.372	98.93	3.846	88.94	1.917
M*Tmiss	.080	1.07	.250	5.78	.320
Tcross			.228	5.27	
Tpred	7.452	100.00	4.324	100.00	1.723
Tacc	8.16	100.00	4.93	100.00	1.655
-Tsvc	.794	9.73	.388	7.87	2.046
Trun	7.366	90.27	4.542	92.13	1.622
Trun-Tpred	-.086		.218		
% error	-1.17		4.80		

Analysis of Results

Opcode Distributions

It has been observed many times that very few opcodes account for most of a program's execution. The COBOLC program, for example, uses 84 of the available 183 instructions, but 48 represent 99.08% of all instructions executed, and 26 represent 90.28%. Table 4 gives the opcodes which account for at least 50% of all instructions executed for each of the benchmark jobs. In addition to the frequencies of execution, the table gives the fraction of execution time attributable

to each of the instructions listed. Note that it is common for an instruction to have a ratio of 2 to 5 in execution time percentage versus execution frequency. For example, the "Move Chararacter" (MVC) instruction in the COBOLC job represents 3.92% of all instructions executed, but accounts for 14.97% of IBM execution time, and 16.47% of the Amdahl execution time. In contrast, the "load" (L) instruction in the COBOLGO job represents 16.58% of all instructions executed, but accounts only for 1.65% of IBM execution time, and 1.57% of Amdahl execution time.

****** TABLE 4 -- OPCODE FREQUENCY DISTRIBUTIONS

COBOLC

	Inst Name	%of Inst Count	% of Execution Time IBM	Amdahl
1	BC	22.32	18.81	13.83
2	LA	7.10	2.52	2.37
3	L	6.21	2.03	2.07
4	TM	4.87	1.60	1.62
5	CLI	4.19	1.37	1.40
6	MVC	3.92	14.97	16.47
7	BCR	3.31	2.84	2.64
	Totals	51.91	44.15	40.40

FORTGO

	Inst Name	%of Inst Count	% of Execution Time IBM	Amdahl
1	L	14.05	6.54	6.64
2	AR	12.06	3.74	5.70
3	LE	11.12	5.17	5.26
4	STE	10.54	9.80	5.33
5	ST	7.81	7.27	3.95
	Totals	55.58	32.52	26.87

PL1GO

	Inst Name	%of Inst Count	% of Execution Time IBM	Amdahl
1	L	28.17	17.68	19.56
2	MVI	15.86	23.23	12.61
3	AR	14.84	6.21	10.31
	Totals	58.86	47.12	42.48

LINSY2

	Inst Name	%of Inst Count	% of Execution Time IBM	Amdahl
1	LR	17.96	8.55	10.11
2	AR	13.10	6.24	7.39
3	BC	12.46	21.70	12.35
4	SR	7.28	3.46	4.10
	Totals	50.80	39.94	33.94

COBOLGO

	Inst Name	%of Inst Count	% of Execution Time IBM	Amdahl
1	L	16.58	1.65	1.57
2	AP	10.72	15.45	10.63
3	ZAP	8.96	16.03	10.70
4	BCR	9.92	2.20	1.75
5	MVC	7.31	8.48	8.85
	Totals	52.49	43.82	33.49

FORTC

	Inst Name	%of Inst Count	% of Execution Time IBM	Amdahl
1	L	27.47	15.22	16.22
2	BC	13.01	18.76	14.65
3	ST	12.16	13.47	7.60
	Totals	52.64	47.45	38.47

PL1C

	Inst Name	%of Inst Count	% of Execution Time IBM	Amdahl
1	BC	24.40	24.78	19.49
2	LA	7.77	3.34	3.20
3	CLI	6.76	2.68	2.78
4	L	5.26	2.08	2.16
5	MVC	4.31	16.35	19.73
6	BCR	3.96	4.07	3.90
	Totals	52.47	53.30	51.26

The most commonly executed instructions are often not the ones which account for most of the execution time. Table 5 shows the instructions which, for each of the programs, represent at least 50% of the execution time. Some of the more exotic and many of the variable-length instructions of the 370 architecture now demonstrate their influence; Divide

****** TABLE 5 -- OPCODE TIME DISTRIBUTIONS

COBOLC

	IBM Name	IBM %Inst Time	IBM %Exec Count	Amdahl Name	Amdahl %Inst Time	Amdahl %Exec Count
1	BC	18.81	22.31	MVC	16.47	3.92
2	MVC	14.97	3.92	BC	13.83	22.32
3	STM	11.47	2.19	XC	9.65	0.49
4	LM	8.38	2.77	EX	8.31	2.08
5	CLC	6.07	2.72	LM	7.70	2.77
	Totals	59.70	33.92		55.97	31.58

FORTGO

	IBM Name	IBM %Inst Time	IBM %Exec Count	Amdahl Name	Amdahl %Inst Time	Amdahl %Exec Count
1	STE	9.80	10.54	BXLE	11.22	5.33
2	BXLE	7.41	5.33	DR	11.13	0.94
3	LM	7.41	1.98	L	6.64	14.05
4	ST	7.27	7.81	LM	6.14	1.98
5	DR	7.16	0.94	AR	5.70	12.06
6	STM	6.66	0.67	DER	5.58	0.87
7	L	6.54	14.05	STE	5.33	10.54
	Totals	52.24	41.32		51.74	45.77

PL1GO

	IBM Name	IBM %Inst Time	IBM %Exec Count	Amdahl Name	Amdahl %Inst Time	Amdahl %Exec Count
1	MVI	23.23	15.86	L	19.56	28.17
2	L	17.68	28.17	MVI	12.61	15.86
3	BC	9.53	5.37	AR	10.31	14.84
4	ST	8.99	7.16	BC	8.36	5.37
	Totals	59.43	56.55		50.84	64.23

LINSY2

	IBM Name	IBM %Inst Time	IBM %Exec Count	Amdahl Name	Amdahl %Inst Time	Amdahl %Exec Count
1	BC	21.70	12.46	MDR	17.48	3.10
2	MDR	11.27	3.10	BC	12.35	12.46
3	LR	8.55	17.96	LR	10.11	17.96
4	STD	8.17	5.72	STD	10.02	5.72
5	AR	6.24	13.11	AR	7.38	13.11
	Totals	55.92	52.35		57.34	52.35

COBOLGO

	IBM Name	IBM %Inst Time	IBM %Exec Count	Amdahl Name	Amdahl %Inst Time	Amdahl %Exec Count
1	DP	18.65	1.47	DP	32.76	1.47
2	ZAP	16.03	8.96	ZAP	10.70	8.96
3	AP	15.45	10.72	AP	10.63	10.72
	Totals	50.14	34.00		54.09	21.15

FORTC

	IBM Name	IBM %Inst Time	IBM %Exec Count	Amdahl Name	Amdahl %Inst Time	Amdahl %Exec Count
1	BC	18.76	13.01	L	16.22	27.47
2	L	15.22	27.47	BC	14.65	13.01
3	ST	13.47	12.16	ST	7.60	12.16
4	STM	7.64	0.79	LM	5.69	1.21
5	BCR	6.37	4.67	BCR	5.64	4.67
6	LM	6.02	1.21	STM	5.52	0.79
	Totals	67.48	59.31		55.32	59.31

PL1C

	IBM Name	IBM %Inst Time	IBM %Exec Count	Amdahl Name	Amdahl %Inst Time	Amdahl %Exec Count
1	BC	24.78	24.40	MVC	19.73	4.31
2	MVC	16.35	4.31	BC	19.49	24.40
3	TRT	5.38	1.00	EX	5.42	1.10
4	STM	4.41	0.68	BAL	5.06	3.08
5	BCR	4.07	3.96	TRT	4.13	1.00
	Totals	54.98	34.35		53.82	33.89

Decimal (DP) accounts for 18.65% of the Amdahl time for COBOLGO, and Translate and Test (TRT) accounts for 5.38% of the IBM time for PL1C. The particular strengths and weaknesses of the implementations are apparent; the Amdahl implementation of DR suffers in comparison to IBM (FORTGO), whereas IBM fares rather poorly on STM. Certain dips in performance are clearly evident, and two such examples appear in COBOLC. The Execute (EX) instruction, which the Amdahl designers expected not to be important, is a particularly obvious problem, and has been noted before [EME]. The Exclusive Or Character (XC) instruction, which accounts for 8.31% of the execution time, is almost always a case of overlap discussed later, which IBM optimized but Amdahl did not.

Instruction Length

The 370 architecture has three instruction lengths: 2, 4, and 6 bytes, which loosely correspond to register to register, register to memory, and memory to memory instructions. Table 6 gives the fraction of each type encountered and the average instruction length. The average instruction length does not vary considerably from program to program; the range is 2.92 to 4.49, with most programs around 3.6 bytes. The only exceptions are the COBOL programs, for which 6-byte storage to storage instructions predominate, and the LINSY2 program, for which 2-byte register to register instructions predominate. Although the average does not vary considerably, the proportion of 4-byte instructions varies from 46% to 81%, and similarly 2-byte instructions vary from 15% to 60%. The high fraction of 2-byte instructions for LINSY2 results from the fact that most of the instructions executed are part of a short (26 byte) inner loop that was highly optimized by the compiler.

****** TABLE 6 -- INSTRUCTION LENGTHS

Program	%2-byte	%4-byte	%6-byte	Average
COBOLC	16.15	75.91	7.94	3.836
FORTGO	29.02	70.69	0.29	3.425
PL1GO	16.99	82.37	0.64	3.673
LINSY2	53.96	46.04	0.00	2.920
COBOLGO	14.74	45.77	39.49	4.495
FORTC	18.52	80.86	0.62	3.642
PL1C	17.20	75.45	7.35	3.803

Branch Opcode Analysis

For most programs studied, branch instructions represent a considerable fraction of all instuctions executed (usually 15% to 30%). In five of the seven programs traced, at least one of the branch instructions (usually the simple conditional branch BC) appears in the 50% group.

In Table 7, the column marked '% Count' indicates the fraction of all instructions executed that were potential branch instructions. The column marked '% Success' which follows, shows the fraction of those potential branches that were successful. In the 370 architecture there are two classes of branches: unconditional branches, and conditional branches whose success depends on values at execution time. Each class contains both successful and unsuccessful branches. The only unusual subclass is the unconditionally unsuccessful branch, which is a no-op instruction. The second part of Table 7 shows the fraction of branches in each of these four subclasses as a fraction of all potential branches encountered.

Branch instructions can create difficulties for pipelined implementations of computer architectures. The instruction fetch mechanism is often a stage in the pipeline which is independent of the instruction decoder, and therefore does not recognize branch instructions. A naive implementation results in a large number of unnecessary instruction fetches following a branch instruction, since the recognition of the need to fetch instuctions from the branch target comes too late.

****** TABLE 7 -- ANALYSIS OF BRANCH INSTRUCTIONS

Program	%Brnchs	%Success	Unconditional %Succ	Unconditional %Unsucc	Conditional %Succ	Conditional %Unsucc
COBOLC	31.26	61.75	35.01	6.22	26.74	32.03
FORTGO	13.49	81.81	31.89	6.62	49.92	12.57
PL1GO	6.65	76.04	11.80	9.17	64.25	14.78
LINSY2	14.13	49.34	0.29	0.05	49.64	50.01
COBOLGO	15.78	71.23	35.87	2.75	35.36	26.02
FORTC	21.60	64.41	24.59	3.22	39.82	32.37
PL1C	35.27	67.65	33.50	4.03	34.15	28.32

To address this problem the 168 has a rather sophisticated mechanism by which both the instructions following the potential branch and the instructions at the branch target are fetched into two separate sets of instruction buffers. Although the fraction of success for potential branches seems to be a fairly consistent 60-80%, table 8 demonstrates that it depends heavily on the particular type of branch instruction. The designers of the 168 accounted for this fact by having the instruction fetch mechanism use the specific opcode of the branch to estimate the likelihood of success.

****** TABLE 8

INSTRUCTIONS WHICH CAUSED BRANCHES, SORTED BY FREQUENCY

OPCODE		COUNT	% OF BRANCHES	% SUCCESS FOR THIS OPCODE
47	BC	1343374	56.365%	60.260% OF 2229306
07	BCR	555745	23.318%	69.504% OF 799591
87	BXLE	272120	11.418%	92.208% OF 295116
05	BALR	97030	4.071%	53.303% OF 182036
46	BCT	81041	3.400%	96.562% OF 83926
45	BAL	19646	0.824%	100.000% OF 19646
86	BXH	14387	0.604%	25.434% OF 56565
06	BCTR	3	0.000%	0.009% OF 34229
0A	SVC	1	0.000%	0.420% OF 238
		2383347	100.00%	

In contrast, the 470 simply treats branch instructions as if they had memory operands, and uses the normal memory operand fetch mechanism to fetch the first two words at the branch target location. Pipeline complexity is minimized by having the execution unit determine the results required for conditional branches as early as possible. This is consistent with the very successful philosophy of the Amdahl designers to keep the pipeline as simple as possible. Since we generally find that branch instructions represent a smaller percentage of the execution time for the 470 than the 168, it appears as though the decision to use a simpler mechanism was a good one.

Branch and Execution Distances

One of the common criticisms of the 370 architecture involves the absence of program-counter-relative branch instructions. Table 9 is a typical branch distance distribution which supports this attack, since 75-85% of the branch distances are within 2048 bytes of the program counter. The displacement of 12 bits used in RX branch instructions could therefore have been used for most

branches so that base registers would have been unnecessary for most program references. The fact that 50-60% of the branch distances are within 128 bytes of the program counter indicates that even an 8-bit displacement could be used to considerable advantage.

Although 95-99% of the longer branch distances are within 32K bytes, there are still a substantial number of longer branches (8M bytes and above) representing calls to supervisor routines far from the user's program area.

Most programs show a few important peaks in the branch distance distribution corresponding to the important program loops. Note that the asymmetry around the program counter is not sufficient to justify other than a symmetric signed displacement for relative branch instructions.

****** TABLE 9

BRANCH DISTANCES FOR SUCCESSFUL BRANCHES

(RELATIVE TO THE ADDRESS OF THE INSTRUCTION FOLLOWING THE BRANCH INSTRUCTION.)

INTERVAL	COUNT	CUM % FROM -2
-8388608 TO -16777214	438	36.51%
-4194304 TO -8388606	0	36.47%
-2097152 TO -4194302	0	36.47%
-1048576 TO -2097150	0	36.47%
-524288 TO -1048574	0	36.47%
-262144 TO -524286	11	36.47%
-131072 TO -262142	0	36.47%
-65536 TO -131070	0	36.47%
-32768 TO -65534	5797	36.47%
-16384 TO -32766	36522	35.97%
-8192 TO -16382	36939	32.85%
-4096 TO -8190	22100	29.68%
-2048 TO -4094	23397	27.79%
-1024 TO -2046	16830	25.79%
-512 TO -1022	24076	24.35%
-256 TO -510	36941	22.28%
-128 TO -254	31159	19.12%
-64 TO -126	65120	16.45%
-32 TO -62	69591	10.87%
-16 TO -30	46926	4.91%
-8 TO -14	10160	0.90%
-4 TO -6	292	0.03%
-2 TO 2	43204	3.70%
4 TO 6	109920	13.11%
8 TO 14	119401	23.34%
16 TO 30	97510	31.69%
32 TO 62	38946	35.03%
64 TO 126	44641	38.85%
128 TO 254	36311	41.96%
256 TO 510	45227	45.83%
512 TO 1022	38096	49.10%
1024 TO 2046	41504	52.65%
2048 TO 4094	28314	55.08%
4096 TO 8190	23357	57.08%
8192 TO 16382	30697	59.70%
16384 TO 32766	37796	62.94%
32768 TO 65534	5956	63.45%
65536 TO 131070	0	63.45%
131072 TO 262142	0	63.45%
262144 TO 524286	10	63.45%
524288 TO 1048574	0	63.45%
1048576 TO 2097150	0	63.45%
2097152 TO 4194302	0	63.45%
4194304 TO 8388606	0	63.45%
8388608 TO 16777214	438	63.49%

TOTAL 1167627

Table 10 shows information related to execution distances, which is defined to be the number of bytes of instructions executed between successful branch instructions. The last column gives the equivalent distance in number of instructions, obtained by dividing the average execution distance by the average instruction length for that program. It would seem to be a reasonable estimate of the true average number of instructions between successful branches.

****** TABLE 10 — EXECUTION DISTANCE

Program	Average	Std. Dev.	Avg. # Inst
COBOLC	19.86	17.25	5.18
FORTGO	28.52	31.03	8.33
PL1GO	69.40	34.11	18.89
LINSY2	41.40	25.92	14.17
COBOLGO	33.96	48.07	7.56
FORTC	26.05	25.08	7.15
PL1C	15.94	13.51	4.19

For most programs, the average execution distance is surprisingly small (less than 32 bytes, which is the cache line size) but the standard deviation is large. There are often isolated peaks for relatively large execution distances (see Table 11). With the exception of the PL1GO program, which has the highest average execution distance, 77% to 85% of execution distances are less than 32 bytes. Distances less than 16 bytes account for 40-60% of the execution distances. This tends to justify the choice of 32 bytes for the linesize of the cache on both machines, at least as far as instruction fetch is concerned. This is also consistent with older designs for instruction fetch buffers, such as the IBM 360/91 which has a 64 byte instruction stack.

****** TABLE 11

EXECUTION DISTANCES

AVERAGE LENGTH 33.964 BYTES
(7.556 INSTRUCTIONS OF AVG LENGTH 4.495 BYTES)

LENGTH (BYTES)	COUNT	CUM %
0	0	0.0 %
2	0	0.0 %
4	12830	3.21%
6	61386	18.55%
8	24800	24.75%
10	18364	29.34%
12	44346	40.43%
14	26190	46.97%
16	12370	50.07%
18	55437	63.92%
20	12826	67.13%
22	12717	70.31%
24	8272	72.38%
26	2931	73.11%
28	15868	77.08%
30	5058	78.34%
32	114	78.37%
34	1926	78.85%
36	3552	79.74%
38	2	79.74%
40	1574	80.13%
42	2886	80.85%
44	1	80.85%
46	8049	82.87%
48	100	82.89%
50	5601	84.29%
52	0	84.29%
54	228	84.35%

Opcode Pairs

The measurement of opcode pair frequencies confirms that the overall frequency of an opcode is not independent of the surrounding instructions. Pair occurrences are also important in performance analysis because of pipeline interlocks and other miscellaneous issues such as memory store-through. Table 12 gives the five most frequent opcode pairs for each program. It is not uncommon for the measured frequency of those pairs to be 4 to 9 times greater than the product of the individual opcode frequencies.

An examination of the frequent opcode pairs fails to discover any pair which occurs frequently enough to suggest creating additional instructions to replace it. Many of the instruction pairs which do occur frequently are those that when combined would save only one opcode field since the other instruction fields would still be required. Examples of this nature are test or compare instructions followed by conditional branches (TM/BC, C/BC). Many other frequent pairs are artifacts of the program structure; a simple example is the pair which consists of a loop branch and its target instruction. Alexander [ALE75] mentions the load-branch pair as an extremly frequent one for the XPL compiler (L-BC is 12.4% of the count). We find no pairs with such high frequencies, and in particular find the load-branch combination to be significant only in two of the seven programs. Frequent pairs often result from peculiarities of software conventions; the subroutine-call instruction (BALR) is often followed by the unconditional branch (BC) because the first instruction in almost all subroutines is a branch around the name of the program. For the FORTGO program, the extra branches (which could be easily eliminated by putting the name before the first instruction of the subroutine) cost 0.70% of the execution time of the entire program. Many of the programs have a similar extra cost of between 0.5% and 1.0% due to the same convention.

****** TABLE 12 — OPCODE PAIR DISTRIBUTIONS

COBOLC	First Instr	Second Instr	% Pair Count	% Freq. Product	RATIO
1	TM	BC	4.74	1.09	4.36
2	CLI	BC	4.08	0.93	4.36
3	CLC	BC	2.67	0.61	4.40
4	BC	CLI	2.57	0.93	2.75
5	BC	TM	2.00	1.09	1.84

FORTGO	First Instr	Second Instr	% Pair Count	% Freq. Product	RATIO
1	LE	ST	7.37	1.72	6.29
2	ST	AR	5.34	1.27	4.20
3	AR	AR	5.29	1.45	3.64
4	AR	BXLE	5.28	0.64	8.21
5	BXLE	LE	5.13	0.59	8.66

PL1GO	First Instr	Second Instr	% Pair Count	% Freq. Product	RATIO
1	MVI	MVI	7.65	2.51	3.05
2	AR	AR	7.65	2.20	3.47
3	AR	L	7.16	4.18	1.71
4	L	AR	6.67	4.18	1.60
5	L	A	6.00	1.71	3.50

LINSY2	First Instr	Second Instr	% Pair Count	% Freq. Product	RATIO
1	LR	SR	7.26	1.31	5.55
2	BC	LR	6.65	2.24	2.97
3	SLL	LD	5.39	0.40	13.54
4	LR	SLL	5.22	1.01	5.19
5	LR	AR	4.72	2.35	2.00

COBOLGO	First Instr	Second Instr	% Pair Count	% Freq. Product	RATIO
1	L	BCR	5.79	1.48	3.92
2	AP	NI	5.20	0.72	7.28
3	L	CVD	4.21	0.79	5.31
4	NI	L	3.96	1.11	3.58
5	BCR	L	3.73	1.48	2.52

FORTC	First Instr	Second Instr	% Pair Count	% Freq. Product	RATIO
1	BC	L	6.29	3.57	1.76
2	L	L	6.19	7.54	0.82
3	ST	L	4.03	3.34	1.21
4	L	BCR	3.76	1.28	2.94
5	L	ST	3.66	3.34	1.09

PL1C	First Instr	Second Instr	% Pair Count	% Freq. Product	RATIO
1	CLI	BC	6.54	1.65	3.96
2	BC	LA	4.20	1.90	2.22
3	BC	CLI	3.76	1.65	2.28
4	TM	BC	2.93	0.79	3.71
5	CR	BC	2.26	0.58	3.89

The distinction between the distribution of instruction pairs executed and the static distibution of instruction pairs in the program text should be carefully made. Our results do not contradict findings based on static analysis [FOS71a, HEH] that certain pairs of instructions might be frequent enough to justify replacement by a single instruction to improve code density.

Registers and Address Calculation

The 370 architecture expresses addresses as the sum of a 24 bit base value in a register with a 12 bit displacement in the instruction. Some instructions allow an additional 24 bit quantity in another register to be used as an index. In all cases specification of register 0 for the base or index indicates that a value of zero is to be used in lieu of the contents of the register. The hardware does not distinguish between registers which contain addresses and registers which contain index values, so the interpretation of statistics about base and index register utilization are difficult to relate to the program organization. Nevertheless information about the occurrence of zero in the register fields can be easily interpreted. Table 13 shows that it is very infrequent for instructions to specify the use of both index and base registers. Except for the program LINSY2, which is known to have many array references, 80% to 95% of the indexed instructions do not use both base and index registers. A reorganization of the 370 addressing modes could profitably include a non-indexed mode in which the space saved is used for a longer displacement.

****** TABLE 13

REGISTER USE FOR RX-INSTRUCTION
EFFECTIVE ADDRESS CALCULATION

Program	%No Regs	%1 Reg	%2 Reg
COBOLC	0.39	95.51	4.09
FORTGO	0.96	77.25	21.79
PL1GO	0.09	82.05	17.86
LINSY2	0.24	65.04	34.72
COBOLGO	0.01	98.93	1.06
FORTC	4.08	87.95	7.97
PL1C	1.93	92.48	5.59

The distribution of register utilization for address calculation shows that no more than 3 registers account for most of the use. The others are used for address calculation less frequently, or are used for program accumulators.

Operand Lengths

The TRACE program accumulates the distribution of the lengths of all the operands for instructions for which the operand lengths are not implied by the opcode. These operand lengths are either fixed and defined in other fields of the instructions (like the number of registers specified in the Load Multiple instruction), or are data dependent (like the number of bytes which must be referenced before an inequality is detected in a Compare Character instruction). These variables are required to calculate the instruction execution times.

For the purposes of exposition we have divided the variable operand length instructions into three classes: (1) the multiple register load and store instructions (LM and STM), (2) the character manipulation instructions, like Move Character (MVC), and Compare Character (CLC), and (3) the decimal arithmetic instructions like Add Decimal (AP).

LM/STM. The STM and LM instructions save and load a contiguous set of registers designated by a starting and ending register. From one to sixteen registers may be moved by a single instruction. Table 14 shows a typical distribution (from FORTGO) of the number of registers stored and loaded. It is common for there to be two peaks, one for a low value of about 2 to 3 registers for accessing data stored in consecutive words, and another at a high value of 11 to 15 registers for saving and restoring registers across procedure calls. The LM and STM are not used symmetrically: for a given number of registers loaded or stored the frequency counts are often quite different. For the FORTGO program, the average number of registers used for STM is 13.23, and for LM is 5.99. For both machines, the marginal cost of storing one more register is smaller than the execution time of a load or store instruction, but there is a higher overhead for starting each instruction for IBM than for Amdahl. In both cases it is faster to use several store or load instructions when 3 or fewer registers are involved. Despite the fact that these instructions are never among the most frequent, they contribute much more to the CPU time than their frequency would suggest because of their long execution time. For the FORTGO program for example, the 0.67% of instructions which are STM account for 6.66% of the IBM execution time and 4.59% of the Amdahl execution time.

Character Instructions. The second group of storage-to-storage (SS) instructions are those which specify a source and destination location for a character string and a single length for both operands in the range 1 to 256. One of the characteristics of these instructions that makes their implementation very difficult is that overlapped operands are allowed and must be treated a byte at a time. This allows, for example, a single byte to be propagated throughout a string by a move instruction whose destination address is one greater than the source address, since the fields are processed left to right. Lower performance machines in the 370 family implement these instructions in all cases by processing each byte individually, but for high performance machines this would be too slow. Therefore both computers exhibit execution speeds for the non-overlapped cases which are much higher than that for overlapped. For the IBM Move Character instruction, for example, the non-overlapped case takes 40 nsec per byte moved, but 240 nsec per byte of overlapped move.

On jobs for which MVC is a frequent instruction (PL1C and COBOLC) we find that the nonoverlapped case occurs about 50 times more frequently than the overlapped case. However, the average number of bytes moved is less than 8 for the nonoverlapped move, and greater than 50 for the overlapped move. The result is that the 2% of the MVCs which are overlapped are responsible for 20% of the total MVC time.

****** TABLE 14

LENGTH DISTRIBUTION FOR STM

#REGS	#TIMES	PERCENT
2	17982	11.223
3	521	0.325
5	1082	0.675
6	839	0.524
8	1	0.001
9	4	0.002
10	3471	2.166
11	77	0.048
12	3741	2.335
15	128589	80.259
16	3911	2.441
AVG:	13.231 REGS	

LENGTH DISTRIBUTION FOR LM

#REGS	#TIMES	PERCENT
2	151704	35.174
3	19726	4.574
4	25302	5.866
5	63802	14.793
6	897	0.208
7	10	0.002
8	30146	6.990
9	1105	0.256
10	3392	0.786
11	127559	29.576
12	3741	0.867
13	1	0.000
14	519	0.120
15	1	0.000
16	3392	0.786
AVG:	5.989 REGS	

The overlapped MVC instructions are used primarily to fill a work area with a specific character, and are probably most used to initialize I/O buffers. This is confirmed by the peaks near 80 and 133 which correspond to card and line printer buffers. For programs which don't otherwise use MVC but still do I/O, the overlapped case is an even higher fraction of all occurrences of MVC. For FORTC, for example, the 6% overlapped MVCs account for 52% of the MVC time.

Table 15 is the distribution of operand length for MVC instruction in FORTC. It is representative of the other distributions in the presence of large peaks for small values, and an overall average of 10.06 bytes. Since the startup overhead for these instructions is large, there is almost always a less expensive way to do the equivalent operation for a small number of bytes. For one byte, a IC/STC combination takes less than half the time of a one-byte MVC on both machines.

Most of the other instructions in this variable operand class are much less frequent than MVC. Among them are the instructions for which the number of bytes processed may be much smaller than indicated in the instruction, such as Compare Character (CLC) and Translate and Test (TRT). For these instructions, the distribution of the length specified in the instructions is a poor indicator of the length actually used. A typical examples is COBOLC, where the average CLC instruction specifies 4.53 bytes, but an average of

****** TABLE 15

LENGTH DISTRIBUTION FOR MVC

BYTES	#TIMES	PERCENT
1	24263	52.518
2	2809	6.080
3	957	2.071
4	12871	27.860
5	898	1.944
6	64	0.139
7	10	0.022
8	34	0.074
9	4	0.009
10	3	0.006
11	3	0.006
12	2	0.004
13	1	0.002
14	10	0.022
15	5	0.011
16	5	0.011
17	2	0.004
18	1	0.002
19	1	0.002
20	11	0.024
21	8	0.017
22	9	0.019
23	2	0.004
24	9	0.019
25	14	0.030
26	1	0.002
27	2	0.004
28	9	0.019
29	1	0.002
30	2	0.004
32	6	0.013
33	8	0.017
43	2	0.004
46	1	0.002
48	3	0.006
54	1	0.002
55	447	0.968
70	7	0.015
79	495	1.071
80	1367	2.959
81	872	1.887
89	2	0.004
90	14	0.030
120	21	0.045
132	942	2.039
TOTAL:		46199.
AVG:	10.062 BYTES	

only 1.744 bytes are examined by the hardware.

Another instruction of note is the Exclusive Or Character (XC) which is predominately used in total overlap mode in order to zero fields. This fact was used to advantage in the 168, where the total overlap case is specially optimized to be 15 times faster than the other overlap cases. This was not done for the 470, which explains that XC accounts for 9.6% of the COBOLC program for the 470, but only 3.0% for the 168.

Decimal Instructions. The third group of storage-to-storage instructions consist primarily of those for decimal arithmetic. They appear in significant numbers only in the COBOLGO program. For that program, however, they account for 26.29% of the count, and represent 66.39% of the IBM execution time and 64.30% of the Amdahl execution time. These instructions can vary in execution time by as much as 16 to 1 depending on the operand lengths, but the large execution time arises despite the fact that relatively short operands are common. Most operands are 2 to 6 bytes long even though the maximum possible is 16. The average execution time of the Divide Decimal (DP) instruction is about 15 usec for both machines. Not suprisingly, the average instruction execution rate for the COBOLGO program (.810 MIPS for IBM, 1.353 MIPS for Amdahl) is drastically smaller than the average for all the programs (3.519 MIPS for IBM, 5.518 MIPS for Amdahl). Considering the popularity of COBOL as a programming language, these instructions, which require slow serial byte processing, represent a major degradation of the speed of the machines.

In view of the poor performance of many of the variable operand length instructions, their inclusion in the the architecture of a high-performance computer is questionable. The absence of such instructions in machines like the CDC 7600 and the CRAY-1 is indicative of their emphasis on high speed. The arithmetic which must occur before these instructions begin their data transfer suggests that it is quite difficult to optimize them for short operands. A compromise, if the execution of these instructions cannot be optimized, may be to supply simpler instructions from which the more complex character and decimal instructions can be composed, as illustrated by the byte instructions of the PDP-10. An immediate improvement could be obtained if compilers were to replace these instructions by faster equivalents when they are available, but this would require tailoring the compilers to specific models of the computer series.

Cache Effects

The correction due to cache misses ranges from 1% to 5% for IBM, but from 3% to 19% for Amdahl, indicating that the memory subsystem is a major bottleneck for the Amdahl machine. In some sense the memory architecture forces the 470 to lose some of the raw speed advantage of the CPU. There are two factors which contribute to the problem. The cache organization of the Amdahl machine produces from 1.7 to 3 times the number of cache misses, and the penalty for each miss is 1.56 times that for IBM. Thus the overall cache penalty for Amdahl is 2.5 to 4 times more than IBM, whereas the raw execution speed, defined as Tins (the time required to execute the instructions with no cache misses) is 1.9 times faster than IBM. The loss due to the cache organization could have been eliminated, but to maintain the raw speed advantage would have required a cache miss penalty of 250 nsec, which would not have been economically feasible at the time. The dilemma of Amdahl may result from a mismatch between the MOS memory chips available commercially and its proprietary ECL LSI technology which is far more advanced.

Pipeline Effects for the 470

Because the timing formulas for the Amdahl machine include specific pipeline variables, we can assess their effect on the execution. The pipeline is optimized for 4-byte instructions which have single word operands, and any deviation causes potential conflicts with subsequent instructions.

The seven pipeline variables depend upon local instruction sequences (for exammple S1 and DWD described earlier), and therefore cannot be computed from global averages. The exact evaluation of these variables would require a complete and complex simulation of the pipeline at the time the program is traced. As a compromise, we use the pair and triple frequency data collected while tracing to reconstruct instruction sequences and average the variable value for each sequence.

In general, the speed degradation due to pipeline conflicts seems to be quite small. For most programs, each of the variables contributes less than 0.5% to the total execution time. The only cases of a larger contribution are when the variables affect specific instructions which occur frequently. For the COBOLGO job, an average additional 1.1 cycles (35.75 nsec) is added to each decimal instruction. This represents a 1.35% increase in execution time. For PL1GO, the doubleword store instructions result in an additional 1.17%. For LINSY2, the delay caused by late setting of the condition code needed for conditional branches adds 0.3%. Although there are wide variations, these worst case examples demonstrate the overall good design of the pipeline.

Summary

A verifiable model of CPU performance using simple and reusable tools shows that basic CPU speed as seen by the user is significantly degraded by memory and operating system effects. This performance analysis, based on instruction timing rather than frequency data, shows also that a few instructions can be disproportionately costly. Many traditional problem areas for high performance computers seem to be under control. The instruction pipeline functions well and branching has little deliterious effect. Memory can be a bottleneck, but the effects of cache store-through policies are negligible. No popular instruction pairs cause particular difficulties, and they are often program-specific artifacts.

Program usage seems to be inconsistent with high-performance implementations in some areas. Decimal arithmetic may be convenient for some applications but is disastrously slow. Storage to storage instruction operands are almost always short and those instructions have high startup costs. Some special cases allowed by the architecture (such as totally overlapped Exclusive-Or) must be individually optimized or performance will suffer. Interaction with the operating system is not only visible because of the time charged for its services, but also because it seriously affects the program miss ratio by disturbing cache memory contents.

These conclusions suggest that designers of high-performance computers should consider the following items to be important: (1) faster memory, (2) more efficient cache, (3) simple pipelines, (4) avoidance of instructions which require serial processing of small data elements, and (5) high-speed decimal arithmetic if it must be included at all.

Conclusion

The performance evaluation techniques described in this paper allow us to draw conclusions about the architecture and the implementation of two high-perfomance computers with the same architecture. The time spent by an executing program can be apportioned among the various system components. The confidence in the results derives from the verification of the model with actual performance. The accuracy exhibited by these techniques and the ability to change the timing formulas to reflect changes in an implementation allow the designer to predict the performance effects of those changes on future machines.

ACKNOWLEDGEMENTS

The considerable assistance and advice of Forest Baskett was essential to this work. John Banning was very helpful in criticizing an early version of the paper. We thank Amdahl Corporation, and specifically Kornel Spiro, Manager of Computer Architecture, for their cooperation and for the generous use of an early version of the instruction statistics program originally developed at Amdahl. We are indebted to Chuck Gray at the University of Michigan for running benchmark jobs on their Amdahl 470. The original incentive for the analysis of machine traces is due to Harry Saal. It should be emphasized that the results and discussions are strictly unrelated to any current or future architectural efforts of the manufacturers involved.

REFERENCES

[AGA73] Agarwal, D.P., "Design of an Efficient Instruction Set", Carnegie-Mellon, 12/72, 11/73

[AGA75] Agajanian, A.H., "A Bibliography on System Performance Evaluation", Computer, November 1975, pps 63-74.

[ALE72] Alexander, W.G., "How a Programming Language is Used", Computer Systems Research Group, University of Toronto, Report CSRG-10, February 1972.

[ALE75] Alexander, W.G., Wortman, D.B., "Static and Dynamic Characteristics of XPL Programs", Computer, November 1975, Vol 8, 11, pps 41-46.

[AMD] Amdahl 470V/6 Machine Reference Manual, Amdahl Corporation, Form No. MrM 1000-1, 2nd Ed., 1976, Sunnyvale, Calif.

[ANA] Anagnostopoulos, P.C., Michel, M.J., Sockut, G.H., Stabler, G.M., VanDam, "Computer Architecture and Instruction Set Design", NCC 1973, pps 519-527.

[ARB] Arbuckle, R.A., "Computer Analysis and Throughput Evaluation", Computers and Automation, January 1966, pps 12-15.

[BEN] Bencher, D., "OS/VS2 Release 1 Functional Description", SHARE XL Proceedings, March 1973, pps 320-324.

[CON] Connors, W.D., Mercer, V.S., Sorlini, T.A., "S/360 Instruction Usage Distribution", IBM Systems Development Division, Report TR 00.2025, Poughkeepsie, N.Y., May 1970.

[EME] Emery, A.R., Alexander, M.T., "A Performance Evaluation of the Amdahl 470V/6 and the IBM 370/168", CMG IV, October 1975, San Francisco.

[FLY] Flynn, M.J., "Trends and Problems in Computer Organizations", Information Processing 74, North Holland Pub. Co., pps 3-10, 1974.

[FOS71a] Foster, C.C., Gonter, R., "Conditional Interpretation of Operation Codes", IEEE Trans. on Computers, January 1971, pps 108-111.

[FOS71b] Foster, C.C., Gonter, R.H., Riseman, E.M., "Measures of Opcode Utilization", IEEE Transactions on Computers, May 1971, pps 582-584.

[GIB] Gibson, J.C., "The Gibson Mix", IBM System Development Division, Report TR 00.2043, Poughkeepsie, N.Y., 1970. Research done in 1959.

[HAN] Haney, F.M., "Using a Computer to Design Computer Instruction Sets", Carnegie-Mellon, May 1968 PhD Thesis

[HEH] Hehner, E.C.R., "Matching Program and Data Representations to a Computing Environment", Computer Systems Research Group, University of Toronto, Report CSRG-44, November 1974.

[HUG] Hughes, J.H., "A Functional Instruction Mix and Some Related Topics", International Symposium on Computer Performance Modeling Measurement and Evaluation, Cambridge, Mass., March 1976.

[IBM] IBM System/370 Model 168 Theory of Operation / Diagrams Manual, Form No. SY22-6931-6936, Volumes 1-6, IBM Corporation, Poughkeepsie, N.Y., 1974.

[JAY] Jay, R.M., National CSS Inc, Distribution at SHARE, New York, August 1975.

[KAP] Kaplan, K.R, Winder, R.O, "Cache-Based Computer Systems", Computer, March 1973, pps 30-36.

[LIP] Lipps, H., "Instruction Timing for the CDC 7600 Computer", European Organization for Nuclear Research, CERN 75-19, Geneva, December 1975.

[LUN] Lunde, A., "Evaluation of Instruction Set Processor Architecture by Program Tracing", Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa., July 1974.

[MER] Merrill, B., "370/168 Cache Memory Performance", SHARE Computer Measurement and Evaluation Newsletter, July 1974, pps 98-101.

[MUR] Murphey, J.D., and Wade, R.M., "The IBM 360/195 in a world of Mixed Job Streams", Datamation, April 1970, pps 72-79.

[ROS] Rossmann, G.E., Palyn Associates, unpublished communication.

[SNI] Snider, D.R., et al, "Comparison of the Amdahl 470 V/6 and the IBM 370/195 Using Benchmarks", Argonne National Laboratory Report ANL-76-50, March 1976.

[VAN] VanTuyl, W.H., "An Engineering View of Performance, IBM System/370 Model 168", SHARE Computer Measurement and Evaluation Selected Papers, Volume II, p 816-829, August 1973

[WIN] Winder, R.O., "A Data Base for Computer Performance Evaluation", Computer, March 1973, pps 25-29.