Relating Static and Dynamic Machine Code Measurements

Jack W. Davidson, MEMBER, IEEE*, John R. Rabung, and David B. Whalley,* MEMBER, IEEE

Abstract—**In an effort to relate static measurements of machine code instructions and addressing modes to their dynamic counterparts, both types of measurements were made on nine different machines using a large and varied suite of programs. Using classical regression analysis techniques, the relationship between static architecture measurements and dynamic architecture measurements was explored. The statistical analysis showed that many static and dynamic measurements are strongly correlated and that it is possible to use the more easily obtained static measurements to predict dynamic usage of instructions and addressing modes. With few exceptions, the predictions are accurate for most architectural features.**

Index Terms—**Instruction sets, performance evaluation, computer architecture, dynamic measurements, static measurements, machine design.**

I. Introduction

Static measurements of program code at machine level are generally thought to be useful for determining textual space needs while dynamic measurements can be used in performance evaluation [10, 18]. Making either type of measurement is conceptually straightforward with access to assembly code, but the implementation of dynamic measuring techniques is more difficult and time-consuming since it may involve simulation, tracing, or compiler modification. This difficulty often results in using a small set of programs [14], a large set of small programs [9], or small test data sets [20]. Furthermore, in architectural design settings where a machine may not even exist except on paper and where preliminary information on dynamic behavior could improve early design decisions, direct dynamic measurement is impossible.

In an effort to simplify the collection of dynamic measurements of an architecture's use of instructions and addressing modes, the relationship between static measurements of the usage of instructions and addressing modes and the corresponding dynamic measurements were analyzed using classical regression analysis techniques. The results of the analyses revealed that many static and dynamic measurements are

This work was supported in part by the National Science Foundation Grant CCR-8611653.

J. W. Davidson is with the Department of Computer Science, University of Virginia, Charlottesville, VA 22903. J. R. Rabung is with the Department of Computer Science, Randolph-Macon College, Ashland, VA 23005. D. B. Whalley is with the Department of Computer Science, Florida State University, Tallahassee, FL 32306.

strongly correlated and that it is possible to predict dynamic behavior from the more easily obtained static measurements for given workloads. The predictions are accurate for most architectural features, but deviations can occur because of the lack of common architecture features and the use of different code generation strategies. The ability to predict dynamic behavior from static data can significantly reduce the design time necessary to implement a new architecture.

In the following section, the techniques that are commonly used to gather static and dynamic measurements are reviewed. Section III describes how the measurements for the experiments were obtained and the statistical analyses performed. The results and interpretation of the analyses are presented in Section IV. Section V contains concluding remarks.

II. Methods of Obtaining Dynamic Measurements

There have been several different methods or combination of methods used to collect dynamic measurements. The factors that may influence the choice of a method include:

1. implementation effort
2. accuracy of measurements
3. types of measurements required
4. computational requirements for obtaining measurements

Several techniques for collecting program measurements are described below.

Machine simulation imitates a machine by interpreting the machine instructions [1, 2]. Dynamic architectural-level measurements are updated as the simulator interprets each instruction. The efforts to construct a simulator and a compiler are comparable. The total execution time of a simulated program is typically hundreds of times slower than if the program is executed directly [9]. One advantage of machine simulation is the ability to try out new machine designs very early in the design process (i.e., before hardware exists). Another advantage is that if the simulator is carefully designed with respect to portability, simulations can be run on any machine that supports the language used to implement the simulator.

Program tracing records a sequence of events which occur during the execution of a program. These events may include the sequence of instructions executed and the memory addresses referenced. A common method of implementing program tracing is to force a ‘trap’ after the execution of each instruction. The trap handler is responsible for gathering and recording the relevant information. Some architectures provide support for this approach via a bit in the processor status register. When the bit is set, an interrupt occurs after the execution of each instruction. The execution of a program with tracing enabled may run 1000 times slower than one without tracing [9]. An advantage of using tracing is the ability to collect dynamic architectural-level measurements that require knowledge of the order of the execution of instructions [12, 17, 20].

Program monitoring collects dynamic architectural-level program measurements without perturbation of a machine. This method is only possible with a device known as a hardware monitor which allows the frequency and time taken by each instruction to be measured [5]. The advantage of monitoring is that program measurements can be collected with no overhead in execution time. Furthermore, measurements that include the effect of the operating system can be captured. The disadvantages include limited measurements and the requirement of a specialized, expensive, hardware device.

Program instrumentation modifies a program by inserting code to increment counters during program execution. Static information, collected by program analysis, is associated with each counter. After the termination of the execution of the program, each set of static information is weighted by the value of its associated counter to produce the dynamic measurements [9]. Instrumentation captures program measurements with little overhead in execution time. One must ensure, however, that the modifications made to collect measurements do not change the execution behavior of the program.

Microcode instrumentation can be used when the machine being measured has been implemented using microcode. With this approach, the microcode is modified to record dynamic architectural-level measurements after the execution of each macroinstruction. Relative to tracing and simulation, this method does not impose a large run-time penalty. For example, modification to collect instruction frequency information slowed a Mesa machine by about a factor of six [14]. A difficulty with microcode instrumentation is that the microcode of a machine is often not accessible to the typical user, and when it is accessible it often requires great expertise to modify without adversely affecting the operation of the machine.

III. Analysis of the Relationship of Static and Dynamic Machine Code Measurements

A. Hypothesis

The hypothesis tested was that a linear relationship exists between the static and dynamic measurements of an architecture's use of instructions and addressing modes. Static measurements are those that are obtained by simply counting occurrences of instructions and addressing modes in the generated code. Dynamic measurements are those that are obtained by executing the programs and counting how many times each instruction and addressing mode is used. All measurements in this paper are reported as percentages relative to the other measurements in the corresponding static or dynamic category.

B. Experiment Setting

To test the hypothesis, a set of nineteen C programs were collected. These nineteen programs distributed over forty-five files (more than 22,000 lines of source code) is intended to be representative of the program mix on a typical, general-use Unix system. An execution of the test set on a test machine resulted in the execution of over 100,000,000 machine instructions. Clearly it is not possible to claim that this test set fits all work environments, but it is believed that they represent a reasonable UNIX work load. The set of test programs is summarized in Table I.

The approach to gathering the static and dynamic data was to retarget a C compiler to nine different architectures. The test architectures were:

- Digital Equipment VAX-11
- Motorola 68020/68881
- National Semiconductor 32016
- Intel 80386/80387
- Harris HCX-9
- Concurrent 3230
- IBM PC/RT
- AT&T 3B15
- Intergraph Clipper

The compiler technology, called *vpo* (Very Portable Optimizer), is easy to retarget and generates very good code [3]. Benchmark comparisons of *vpo*-based C compilers and production-quality optimizing compilers on several machines reveal that *vpo* generates comparable code. For example, on the Sun-3 (Motorola 68020-based machine), the SPEC benchmark suite, a widely used set of benchmark programs, was

Table I: Test Set

Class	Name	Description or Emphasis
Unix System Utilities	cal	Calendar Generator
	cb	C Program Beautifier
	compact	Huffman Coding File Compression
	diff	Differences between Files
	grep	Search for Pattern
	nroff	Text Formatting Utility
	od	Octal Dump
	sed	Stream Editor
	sort	Sort or Merge Files
	spline	Interpolate Smooth Curve
	tr	Translate Characters
	wc	Word Count
Benchmark Programs	dhrystone	Typical Set of C Statements and Operands
	matmult	Multidimensional Arrays and Simple Arithmetic
	puzzle	Recursion and Array Indexing
	sieve	Simple Iteration and Boolean Arrays
	whetstone	Arithmetic Operations
User Code	mincost	VLSI Circuit Partitioning
	vpcc	Very Portable C Compiler

compiled using a *vpo*-based C compiler and the production compiler supplied by the manufacturer. The *vpo*-based compiler scored a SPEC mark of 4.3 while the native compiler scored 4.0 (a higher SPEC mark means better performance). Furthermore, this technology is being used commercially in compilers for several different machines, languages, and applications. It is being used in an Ada compiler, in a C compiler for a high-performance signal processor, and in a C compiler for a microprocessor designed to support embedded applications.

vpo was modified to record instruction information before it produced assembly code and to insert instructions to update frequency counters in the assembly code [6]. Since instruction information was recorded before instructions were inserted to update counters, the inserted instructions did not contribute to the information they were meant to collect. For efficiency, the inserted instructions were placed so that they recorded the frequency of use of ‘‘execution classes’’ of basic blocks, rather than that of the basic blocks themselves. Two basic blocks are considered to be in the same execution class if each was executed every time the other was. Typically, the instrumented code ran 10 to 15 percent slower than code that was not instrumented.

The nineteen programs listed in Table I were compiled and executed on each of the machines used in the study. The data for the statistical analysis were the static and dynamic measurements that were captured and accumulated over the entire test set for each architecture. The percentage of each program's contribution to the total static count of instructions and to the total dynamic count of instructions executed for each architecture was also calculated. Table II shows the minimum contribution of the program across architectures, the contribution of the program averaged over all architectures, and the maximum contribution of the program across architectures. As one would expect, the number of static instructions is roughly related to the size of the source program. For example, the two largest programs in this study, *nroff* and *vpcc*, contributed over 60 percent to the static measurements.

Table II: Static and Dynamic Instruction Percentages by Program

Name	Percent Contribution to Static Instruction Counts			Percent Contribution to Dynamic Instruction Counts		
	Minimum Contribution	Average Contribution	Maximum Contribution	Minimum Contribution	Average Contribution	Maximum Contribution
cal	0.66%	0.77%	0.82%	0.02%	0.03%	0.04%
cb	1.82%	2.06%	2.56%	0.35%	0.40%	0.45%
compact	2.07%	2.24%	2.59%	13.84%	15.35%	17.59%
dhrystone	0.83%	0.96%	1.03%	11.06%	13.18%	15.54%
diff	5.49%	6.02%	6.45%	3.30%	4.33%	5.01%
grep	1.85%	2.06%	2.43%	0.12%	0.17%	0.22%
matmult	0.34%	0.38%	0.47%	4.57%	6.67%	12.16%
mincost	2.17%	2.50%	2.90%	18.79%	21.50%	24.64%
nroff	22.59%	23.88%	25.63%	9.18%	10.48%	11.38%
od	2.70%	2.89%	3.15%	1.05%	1.18%	1.41%
puzzle	1.06%	1.28%	1.56%	4.73%	5.77%	7.10%
sed	7.91%	8.73%	9.44%	8.94%	10.79%	12.82%
sieve	0.11%	0.13%	0.15%	1.19%	1.46%	1.62%
sort	3.76%	3.99%	4.30%	0.33%	0.61%	0.92%
spline	1.58%	2.08%	2.87%	0.16%	0.20%	0.23%
tr	0.67%	0.78%	0.89%	0.05%	0.07%	0.08%
wc	0.41%	0.44%	0.48%	0.02%	0.02%	0.03%
whetstone	1.62%	2.08%	3.53%	4.76%	5.81%	8.27%
vpcc	32.61%	36.73%	39.24%	1.39%	1.98%	2.92%

C. Statistical Analysis

Using well-known regression analysis techniques [11], the relationship between static and dynamic measurements was analyzed. To illustrate the process, the relationship between the static and dynamic measurements of call instructions is examined. On obtaining counts of static and dynamic calls from the test set on a particular machine, conversion to percentage was done prior to statistical analysis. For example, results from the VAX-11 showed that 3261 of the 32,117 static machine instructions (10.15%) were `call` instructions while 2,852,847 of the 90,344,853 machine instructions executed (3.16%) were `call` instructions. Consequently, the static-dynamic pair used for representing `call` instructions on the VAX-11 was (10.15, 3.16). Corresponding pairs for other machines are shown in Table III.

Table III. Static and Dynamic Measurements of Call Instructions

Machine	Static Percentage	Dynamic Percentage
VAX-11	10.15	3.16
Motorola 68020	8.14	2.31
Nat. Semi. 32016	8.35	2.43
Intel 80386	7.35	2.13
HCX-9	9.81	2.89
Concurrent 3230	5.96	1.72
IBM RT	5.79	1.46
AT&T 3B15	9.43	2.44
Clipper	7.42	1.78

Given these pairs of numbers for the nine test machines, SPSS/PC+[16] was used to compute the Pearson correlation coefficient, an indicator of linear relationship. Having a possible range of -1 to 1, this coefficient suggests such a relationship when its value is near either extreme of the range. The static-dynamic pairs for the `call` instructions over the sampling of machines yield a Pearson correlation coefficient, r, of 0.9506, which strongly suggests a linear relationship. One interpretation of r is that when a straight line with equation

$$dynamic = b_0 static + b_1$$

is least-squares fitted to the sample static-dynamic pairs, the percentage of the deviation of sample dynamic figures from the dynamic mean ‘‘explained’’ by the linear fit is $100r^2$. Thus, in the case of `call` instructions $100(0.9506)^2 = 90.36$ percent of such deviation in the sample data is subsumed by the line $dynamic = 0.33\ static - 0.43$, where b_0=0.33 and b_1=–0.43 are the linear coefficients from least-squares fitting. This line is shown along with the data points in Figure 1. Appendix I contains regression analysis plots for other categories of frequently used instructions and addressing modes.

Figure 1. Regression of Dynamic Percentage of Call Instructions on Static Percentage of Call Instructions

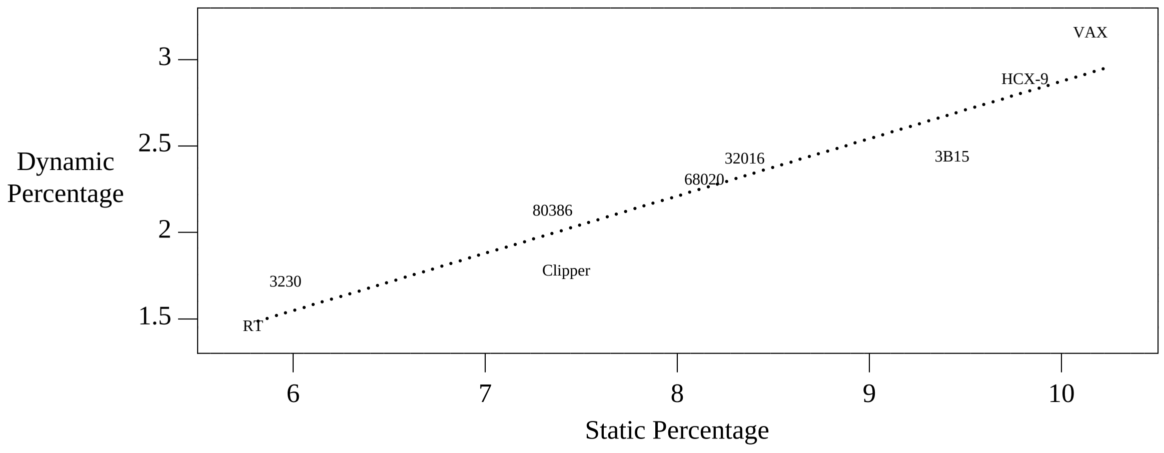


Table IV contains results of the regression analysis for other categories of instructions and addressing modes—categories chosen, in part, for their commonality to most machines. Correlation coefficients shown in the table with one or two asterisks mark cases where sample data gave significant evidence of linearity at a 0.01 or 0.001 significance level, respectively, in a one-tailed hypothesis-testing setting. Thus, in the event that the corresponding static and dynamic figures were actually *not* positively correlated, the chance of a random sample turning up static-dynamic pairs yielding a correlation coefficient as high as that shown would be less than 0.01 and 0.001, respectively.

Some of the measurements for which results are shown in Table IV were tabulated over subgroupings of the architectures because they were heavily influenced by either architectural features or code-generation strategies. The IBM PC/RT was not included in the measurements for register-deferred, displacement, or direct addressing modes since 32-bit addresses could not be encoded directly, but had to be implemented as register-deferred and displacement addresses. Neither was the RT included in the measuring of integer multiply or divide instructions because it does not have these instructions. Instead, it has an

Table IV. Statistical Results from Sample Static-Dynamic Pairs
Except as noted below, all sample sizes are 9.

Measurement	Correlation Coeff	Coeff b_0	Coeff b_1	Static Range Low	Static Range High	Static Mean	Static Std Dev	Std Error
Instructions								
call	.9506**	0.33	-0.43	5.79%	10.15%	8.04%	1.58	0.18
compare	.9743**	1.46	-2.04	4.73%	11.88%	9.53%	2.42	0.88
cond jmp	.9537**	1.42	-1.94	7.38%	13.53%	10.74%	2.15	1.03
uncond jmp (A)	.8775	0.47	-0.17	3.24%	5.65%	4.82%	1.08	0.34
uncond jmp (B)	.9336	0.65	-1.50	3.95%	6.03%	5.38%	0.84	0.24
return (A)	.9445	0.72	-0.32	2.39%	4.08%	3.58%	0.80	0.24
return (B)	.9478*	1.84	-0.90	1.13%	1.65%	1.46%	0.20	0.15
move	.9036**	1.91	-42.87	37.87%	54.82%	45.01%	6.01	5.81
int add	.9865**	1.96	0.76	2.30%	8.90%	5.14%	2.59	0.90
int subtract	.8828**	0.97	0.60	0.47%	4.07%	1.88%	1.15	0.63
int multiply	.9564**	2.91	0.28	0.37%	0.77%	0.55%	0.14	0.14
int divide	.9229*	0.69	-0.04	0.31%	0.52%	0.42%	0.07	0.02
float add	.9347**	3.68	-0.14	0.09%	0.15%	0.12%	0.02	0.03
float subtract	.9503**	0.85	-0.04	0.06%	0.11%	0.09%	0.02	0.01
float multiply	.9673**	2.15	-0.11	0.11%	0.18%	0.15%	0.02	0.01
float divide	.9871*	1.55	-0.02	0.05%	0.09%	0.07%	0.01	0.003
shift	.9969**	3.58	0.21	0.25%	2.90%	1.44%	0.94	0.28
bitwise and	.9547**	1.29	-0.29	0.79%	2.02%	1.47%	0.46	0.20
bitwise or	.9435**	0.28	-0.02	0.24%	0.45%	0.36%	0.07	0.01
Address Modes								
register	.9233**	0.86	16.06	29.64%	58.67%	43.10%	10.40	3.99
immediate	.9706**	0.75	1.10	13.53%	24.17%	18.83%	4.37	0.87
reg-deferred	.8790*	1.74	-1.05	2.49%	6.68%	3.21%	0.60	0.61
displacement	.9830**	1.45	-1.85	5.36%	10.37%	7.56%	1.68	0.62
direct	.9402**	0.46	-0.22	7.95%	20.26%	13.62%	3.74	0.68

instruction that produces a partial product. Sequences of these instructions were used to implement most integer multiplication operations. Division operations were accomplished by invoking a library function.

The Concurrent 3230 was not included in the tabulation of displacement address modes since it uses a caller-save calling sequence. This distorted the static measurement because saves and restores (store multiple and load multiple with displacement operands) were placed around what might be many calls to the same function rather than being placed once inside the called function as on other machines. Similarly, the Intergraph Clipper measurements were excluded in the displacement category since it uses displacement addressing rather than an auto-decrement mode to push arguments onto the run-time stack.

The measurements for integer add operations on the 68020 and 80386 were adjusted to exclude instructions used to adjust the stack pointer after a call. This operation on other machines was either accomplished implicitly by the return instruction or was not required due to only adjusting the stack pointer at the begining of the function rather than each time an argument was evaluated.

The architectures were also partitioned into two groups for the reporting of results concerning unconditional jumps and returns. The first group (A in Table IV) consisted of those machines (four of them) on which the compiler might generate several return instructions in a single function. The second group (B in the table) included those machines (five of them) where a sequence of instructions was required to effect a return from a function. For these machines, the code generation strategy used was to generate this sequence once, at the end of the function. All return instructions (except possibly the last) would jump to this sequence. This, of course, increased the number of unconditional jumps.

D. Experimental Validation

The above results show that there is a strong linear relationship between a machine's static and dynamic usage of many instructions and addressing modes. The data in Table IV can be used to derive a confidence interval estimate of the population mean dynamic figure for a machine *X* from a given static measurement for machine *X*. The formula [4] for such an estimate is:

$$b_0 static_{given} + b_1 \pm t_{1-\alpha/2;n-2}\, S_e \sqrt{\frac{1}{n} + \frac{(static_{given} - static_{mean})^2}{(n-1)S_{static}^{\,2}}}$$

where

$b_0,\ b_1$	are the linear coefficients for the least-squares fit;
$static_{given}$	is the given static level at which prediction is desired;
$1-\alpha$	is the degree of confidence associated with the interval estimate;
$t_{\gamma;\,df}$	is the abscissa value associated with the standard Student-t distribution with df degrees of freedom below which lies 100γ percent of the area under the distribution curve;
S_e	is the standard error of the estimate;
n	is the sample size;
$static_{mean}$	is the mean of the static sample figures; and
S_{static}	is the static sample standard deviation.

So, for instance, to estimate the true mean dynamic `call` statement percentage given a static `call` percentage of 7.50, one might use the results shown in Table IV and the formula above to get a 95 percent confidence interval of

$$0.33\,(7.50)\ -\ 0.43\ \pm\ (2.37)(0.18)\sqrt{\frac{1}{9} + \frac{(7.50-8.04)^2}{8(1.58)^2}}$$

or about

$$2.04\ \pm\ 0.15.$$

Note that this is an estimate of the population mean rather than an estimate of an individual outcome. An individual prediction interval would be considerably wider and is found by the following formula [4] similar to that given for the mean above.

$$b_0 static_{given}\ +\ b_1\ \pm\ t_{1-\alpha/2;n-2}\ S_e \sqrt{1+\frac{1}{n}+\frac{(static_{given}-static_{mean})^2}{(n-1)S_{static}{}^2}}$$

Calculation of an individual 95% prediction interval of the `call` instruction results in the interval

$$2.04\ \pm\ 0.45.$$

When a static measure that is not in the range of static observations is used to predict the corresponding dynamic measure, such a prediction is called an extrapolation [15]. When extrapolating, as a static observation moves further and further away from the range of observations, the prediction interval becomes increasingly wide. Furthermore, the prediction is based on the assumption that the true regression is linear even if we go beyond the range of observations. Consequently, predictions outside the interval should always be viewed carefully.

To determine if the equations defined by the coefficients in Table IV are effective at predicting the dynamic values for a machine, *vpo* was retargeted to the Hewlett-Packard Precision Architecture (PA-RISC) [8]. The PA-RISC, as the acronym implies, is a reduced instruction set computer. Two other machines in this study, the Intergraph Clipper and the IBM PC/RT, are also regarded as RISC machines. Using the test suite of programs, both static and dynamic measures of the PA-RISC were collected. The information in Table IV was used to compute the predicted dynamic behavior from the observed static measures.

The results of this experiment are displayed in Table V. Of the nineteen applicable categories, all but three of the dynamic measures fell within the predicted ranges for an individual outcome. The three categories not within range were the call instruction, the integer add instruction, and the displacement addressing mode. The PA-RISC does not have an integer multiply instruction. An integer multiplication operation is accomplished by calling a library routine (referred to as a millicode routine). The calls to the multiply millicode routines increased the measured dynamic percentage of call instructions.

The measured dynamic usages of integer add instructions and displacement addressing mode were also not within the range predicted by the analysis. First, note that the static measures for these categories were not within the static ranges reported in Table IV. As was pointed out earlier, while it is possible to extrapolate from a measure that is outside the range of observations, the result of such extrapolations should be examined carefully. In the case of the integer add and displacement addressing mode, the explanation is similar to that for the call instruction. These categories are being used to synthesize the direct addressing mode which is not available on the PA-RISC. One important use of the direct addressing mode is to generate addresses of global variables. Thus, the use of the integer add instruction in combination with the displacement addressing mode to generate global addresses skewed the results.

IV. Results and Observations

The result of the regression analyses along with the experimental results using the PA-RISC demonstrate that the linear relationship between static and dynamic machine measurements is robust. The strong relationships between the static and dynamic measurements occurred despite differences in instruction set complexity, such as the ability (or inability) to reference memory in each operand of each instruction.

Thus it appears that it is indeed possible to predict accurately the dynamic usage of instructions and addressing modes by obtaining static measurements. Clearly, code generation strategies and lack of common architectural features can influence these relationships. For instance, the decision to statically generate a single (or several) returns in a function or the ability (or inability) to encode a 32-bit address in a single operand influenced the results. Also a different test set and/or different optimizing compiler technology could influence these relationships [7].

While one would not expect the dynamic percentages to be the same as the static percentages, it is interesting to observe that dynamic percentages for some categories are substantially larger than their static counterparts while dynamic percentages for other categories are substantially smaller. When b_1 is small,

Table V. Prediction Intervals and Results for PA-RISC

Measurement	Static	Dynamic	Prediction Interval		In Range
			Low	High	
Instructions					
call	6.38	2.41	1.20	2.16	
compare	7.97	10.50	7.34	11.86	✓
cond jmp	7.95	10.45	6.54	12.16	✓
uncond jmp (B)	4.31	2.03	0.32	2.24	✓
return (B)	1.17	1.57	0.62	1.88	✓
move	49.24	49.96	36.31	66.05	✓
int add	27.45	25.55	47.70	61.42	
int subtract	0.62	2.33	0.00	2.87	✓
int multiply	N/A	N/A	N/A	N/A	
int divide	N/A	N/A	N/A	N/A	
float add	0.09	0.24	0.11	0.27	✓
float subtract	0.08	0.04	0.01	0.05	✓
float multiply	0.10	0.15	0.07	0.15	✓
float divide	0.05	0.06	0.03	0.09	✓
shift	1.31	4.51	4.20	5.60	✓
bitwise and	0.92	0.60	0.37	1.43	✓
bitwise or	0.27	0.06	0.04	0.08	✓
Address Modes					
register	58.50	62.40	55.24	77.50	✓
immediate	21.33	18.77	14.89	19.31	✓
reg-deferred	4.24	5.94	4.47	8.19	✓
displacement	10.67	6.87	11.28	15.96	
direct	N/A	N/A	N/A	N/A	

this relationship is reflected in the linear coefficient, b_0, being greater than or less than 1. Relying on static percentages during the design phase without understanding these shifts could result in a machine whose performance is less than expected.

Table VI shows the static and dynamic percentages for the Motorola 68020/68881. This machine is fairly representative of the machines CISC machines studied. For this machine, the dynamic percentage of call instructions is less than ⅓ of the static percentage. Tanenbaum observed similar behavior [19]. In his study, this behavior was attributed to error handling procedures that contribute to the static measurement, but in normal operation are not called and therefore do not increase the dynamic measurement.

Table VI. Static and Dynamic Percentages on the Motorola 68020

Measurement	Static Percentage	Dynamic Percentage
Instructions		
call	8.14	2.31
compare	9.28	12.03
cond jmp	10.93	13.86
uncond jmp	5.97	2.54
return	1.61	2.06
move	43.79	42.90
int add	3.13	7.49
int subtract	1.57	2.10
int multiply	0.50	1.84
int divide	0.43	0.24
float add	0.12	0.32
float subtract	0.09	0.04
float multiply	0.14	0.20
float divide	0.07	0.09
shift	0.44	1.48
bitwise and	1.94	2.12
bitwise or	0.37	0.08
Address Modes		
register	41.42	50.81
immediate	17.22	14.14
reg-deferred	2.49	3.27
displacement	6.85	7.24
direct	14.21	7.14

Analysis of the test suite used for this study revealed a more dominant factor. Functions are often written to abstract some common set of operations that are used repeatedly in a program. Consequently, in computing the static percentage of call instructions, the instructions comprising the body are only counted once. When computing the dynamic percentage, however, on each call the instructions in the function that are executed are counted. This greatly increases the total number of instructions executed while only slightly increasing the number of call instructions executed. The result is a significantly lower dynamic percentage of call instructions. This effect is further magnified because statically 16.9% of the functions in the test suite used for this study were leaves (functions that make no calls to other functions), while a full 55 percent of the executed functions were leaf functions. It was observed that the static measurements for the 68020 indicated that instructions in leaf functions comprised 7.0% of the total number of instructions, while dynamic measurements showed they accounted for 30.3%. These measurements indicate that while leaf functions were on average smaller than nonleaf functions, leaf functions were also used more frequently.

The difference in the measurements for conditional and unconditional jumps reveals an interesting behavior. For all machines, the dynamic percentage of conditional jumps was higher than the static percentage, while for unconditional jumps the opposite was true. For the 68020, the dynamic percentage of unconditional jumps was less than ½ the static percentage. Detailed inspection of the code generated for the test suite programs showed that the dominant factor that raised the dynamic percentage of conditional jumps was loops. The code for the termination condition of a loop generally involves a conditional jump. This jump is executed each time through the loop. Statically on the 68020, the conditional jumps associated with the termination condition of loops were 14.6% of the total number of conditional jumps. The dynamic measurements indicated that these conditional jumps accounted for 30.0% of the total. Consequently, it is not surprising that the dynamic percentage is higher than the static.

With respect to unconditional jumps, there were two situations that seemed to account for the difference in the static and dynamic percentages. In `switch` statements, each case within the switch usually ends with an unconditional jump instruction to the end of the `switch` statement. With a `switch` statement that has many cases, there will be many instances of unconditional jumps, but when the `switch` is executed only one case is selected and therefore only one unconditional branch is executed.

Correspondingly, with `if-then-else` constructs, a conditional jump is generated to evaluate the `if` portion of the statement, while an unconditional jump is generated at the end of the `then` block. At run-time, if the `if-then-else` construct is executed, the conditional jump is always executed (the branch may or may not be taken), while the unconditional jump may not be executed.

Studies of instruction usage of specific architectures have been performed. MacGregor and Rubenstein measured the dynamic usage of instructions of a 68000-based machine [13]. The top four categories by dynamic percentage were:

1. move instructions (32.85%)
2. branch instructions (23.95%)
3. arithmetic instructions (16.31%)
4. compare instruction (10.95%)

These categories account for over 80% of all instruction executions.

Inspection of Table VI reveals similar rankings. The top groups by dynamic percentage were:

1. move instructions (42.90%)
2. branch instructions (conditional jump + unconditional jump = 16.40%)
3. arithmetic instructions (int add + int sub + int mul + int divide + shift = 13.15%)
4. compare instructions (12.03%)

Again, the top four instruction categories account for over 80% of all instruction executions.

Similar measurements of the use of instructions on the VAX-11/780 were reported by Clark and Levy [5]. They also observed that a small number of instructions account for a large percentage of instruction executions. They also note that applications use instruction sets quite differently. This points out the importance of picking a representative suite of test programs.

V. Conclusion

This study has shown that there is a strong linear relationship between static and dynamic machine code measurements. This result can be used by designers of new machines to estimate data on dynamic instruction and addressing mode usage very early in the design process—before either a simulator or machine is available. This preliminary information can be used to direct the implementation effort to the areas that will be most crucial to achieving high performance. The estimated dynamic usages can also be used to determine whether a potential design meets performance goals and objectives. This is particularly important for architectures used in real-time systems. Furthermore, even when a simulator is available, the overhead associated with many commonly used machine simulation techniques limit the size of programs that can be measured. The strong relationship between static and dynamic measurements allows dynamic information to be obtained for very large programs without having to simulate their execution.

VI. Acknowledgements

The authors would like to thank Robert P. Cook for suggesting a statistical approach to analyzing the data we had collected. The support of the National Science Foundation under Grant CCR-8611653 is gratefully acknowledged. Anne Holler and the referees provided many helpful suggestions and improvements. We would also like to thank Intel Corporation and Concurrent Computer Corporation for donating equipment to support this research. We also wish to express our appreciation to Hewlett-Packard, Inc, Intergraph Corporation, and International Business Machines for making their machines available for this study.

VII. Appendix I: Plots of Static and Dynamic Measurements

Figure 2. Regression of Dynamic Percentage of Compare Instructions on Static Percentage of Compare Instructions

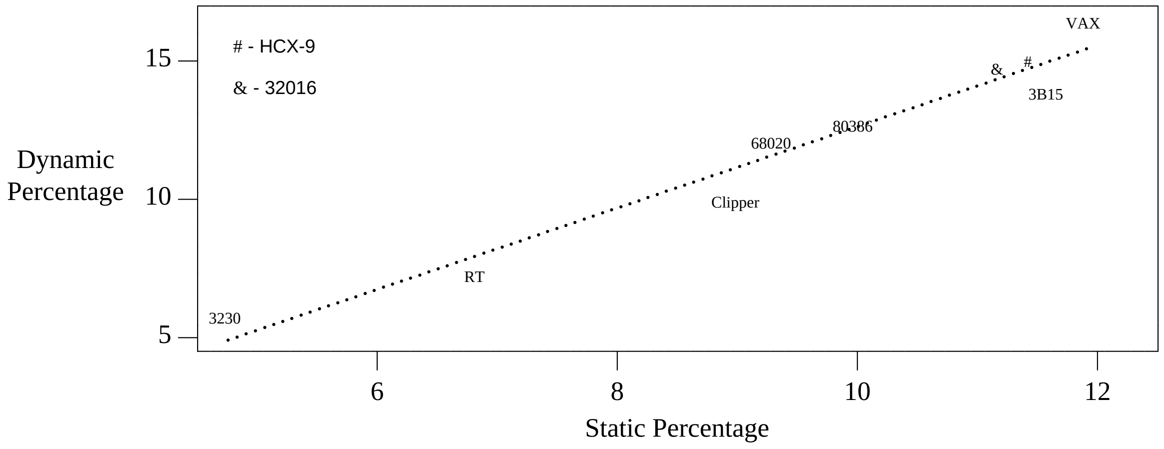


Figure 3. Regression of Dynamic Percentage of Conditional Jump Instructions on Static Percentage of Conditional Jump Instructions

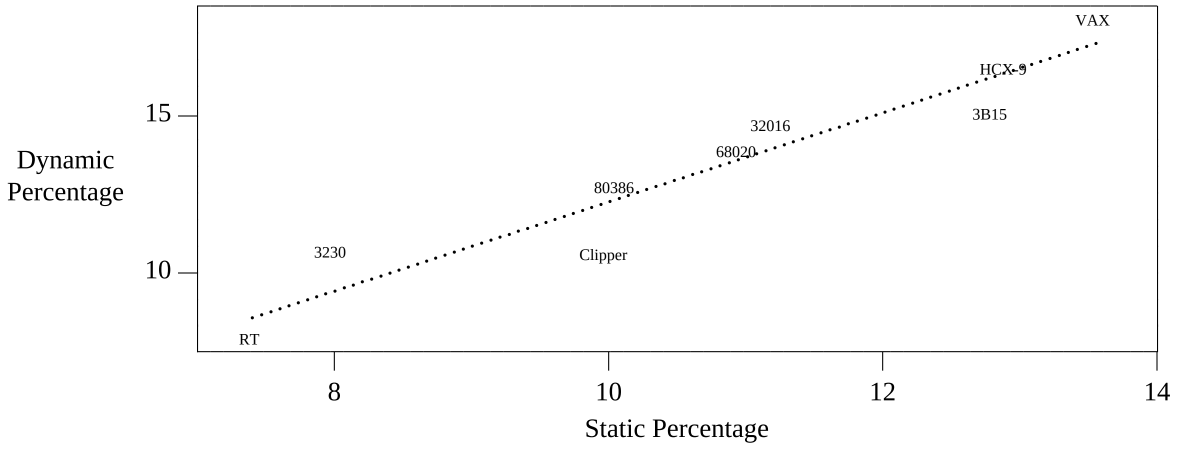


Figure 4. Regression of Dynamic Percentage of Unconditional Jump Instructions on Static Percentage of Unconditional Jump Instructions

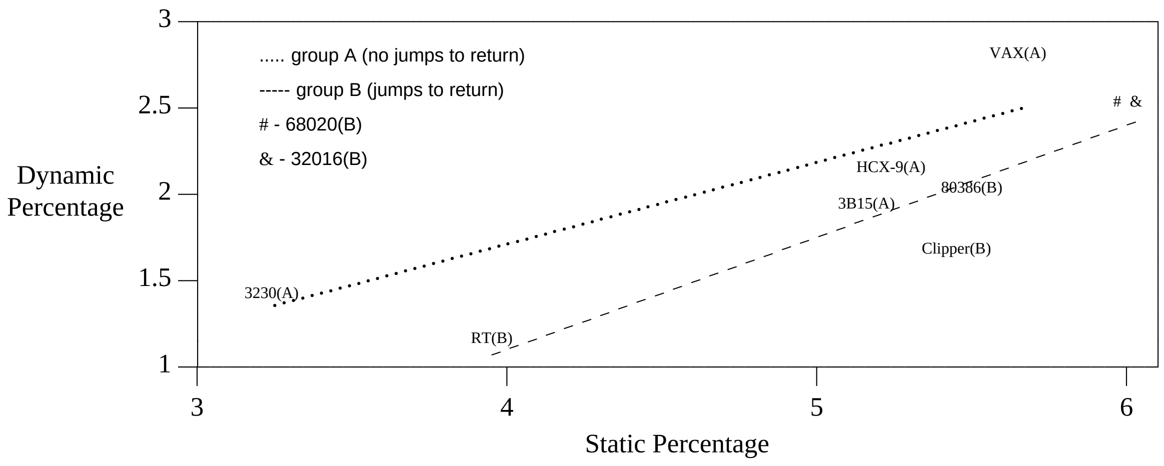


Figure 5. Regression of Dynamic Percentage of Return Instructions on Static Percentage of Return Instructions

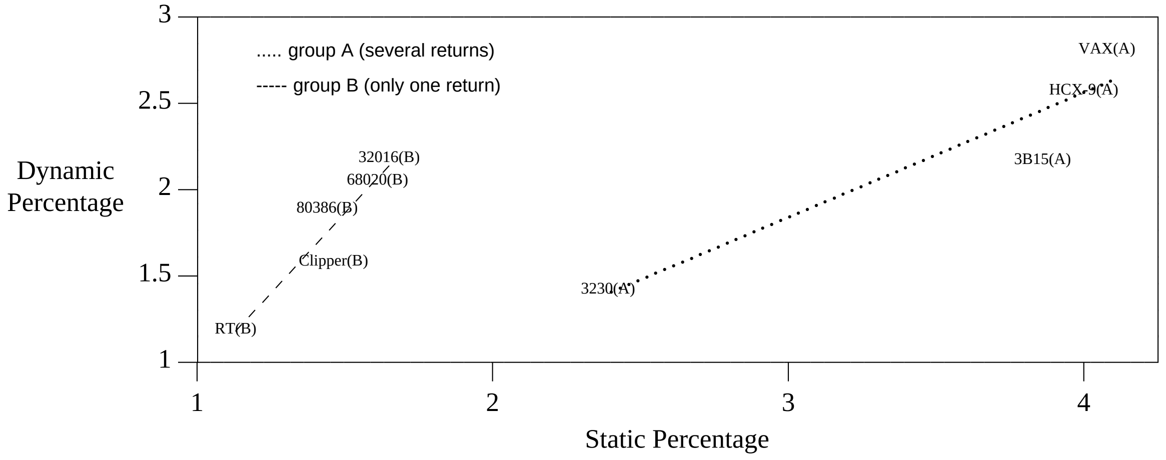


Figure 6. Regression of Dynamic Percentage of Move Instructions on Static Percentage of Move Instructions

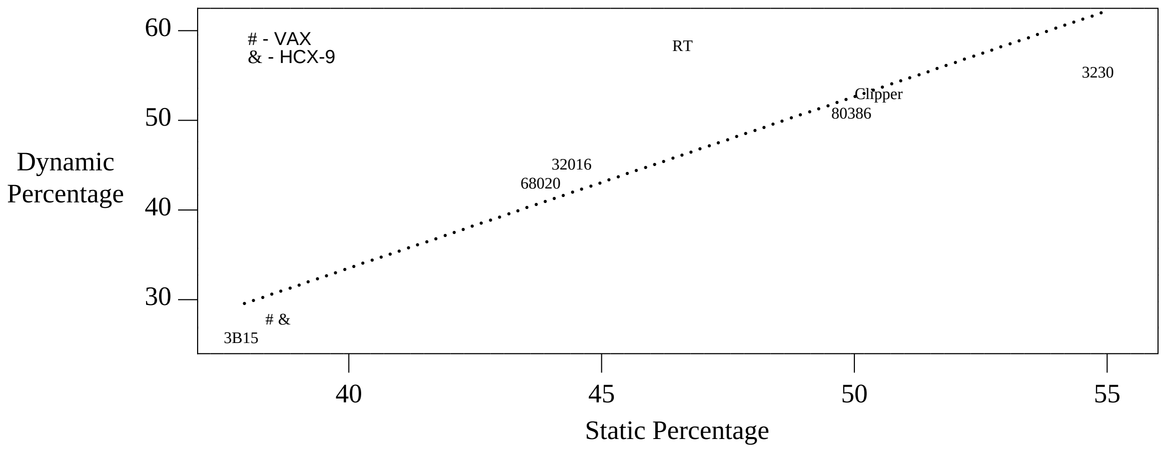


Figure 7. Regression of Dynamic Percentage of Integer Add Instructions on Static Percentage of Integer Add Instructions

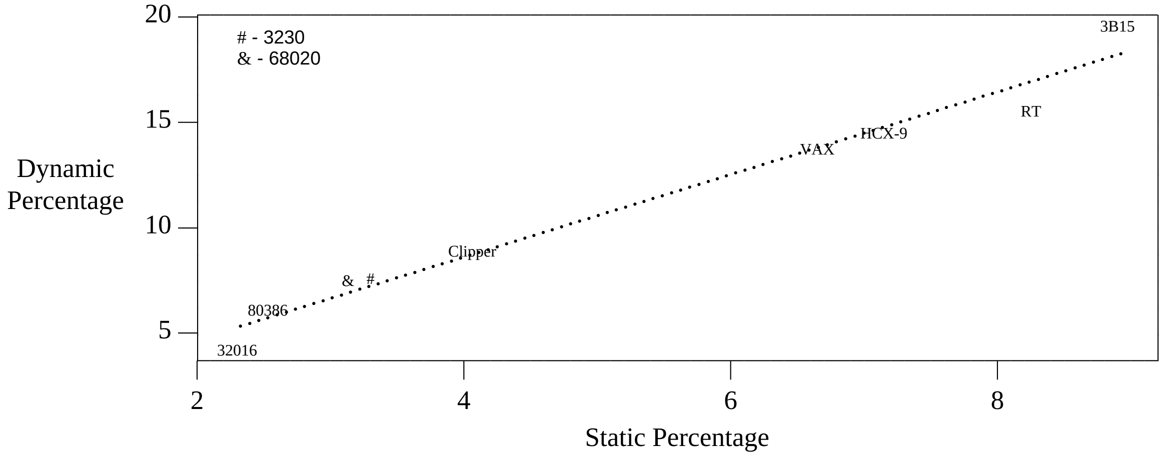


Figure 8. Regression of Dynamic Percentage of Integer Subtract Instructions on Static Percentage of Integer Subtract Instructions

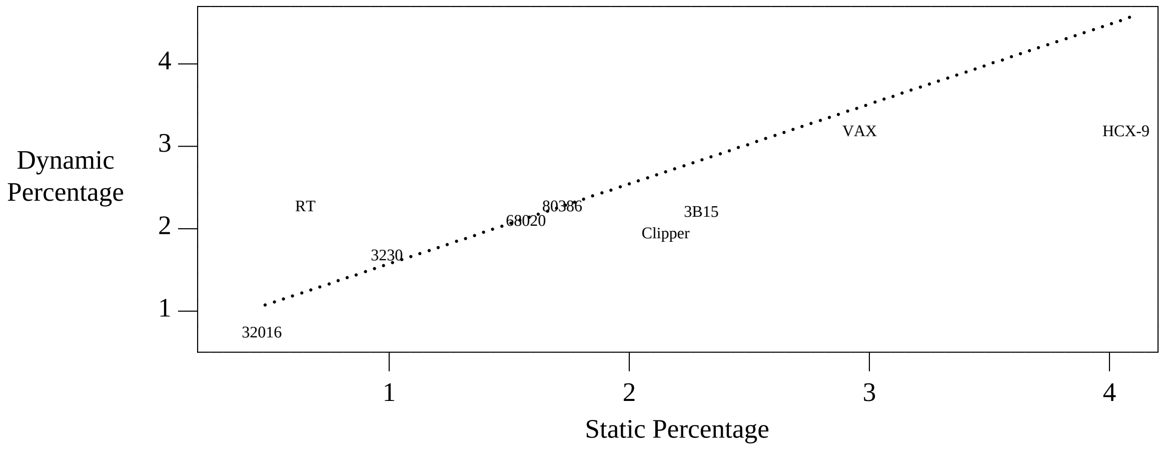


Figure 9. Regression of Dynamic Percentage of Shift Instructions on Static Percentage of Shift Instructions

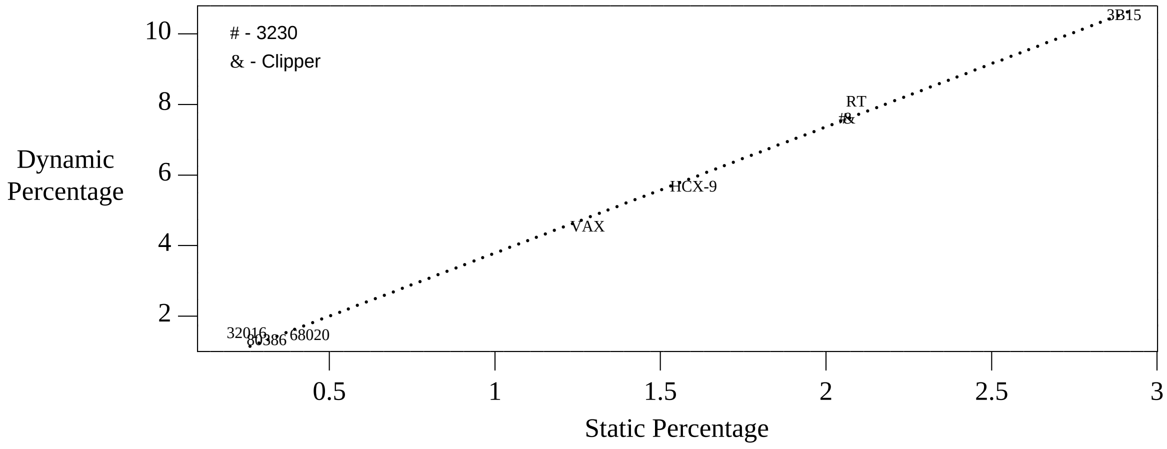


Figure 10. Regression of Dynamic Percentage of Register Addressing Mode on Static Percentage of Register Addressing Mode

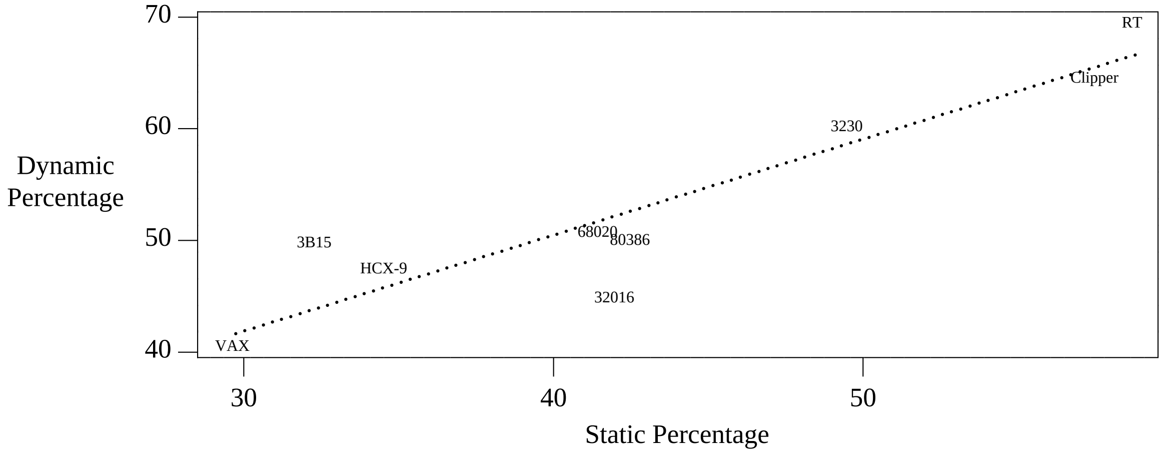


Figure 11. Regression of Dynamic Percentage of Immediate Addressing Mode on Static Percentage of Immediate Addressing Mode

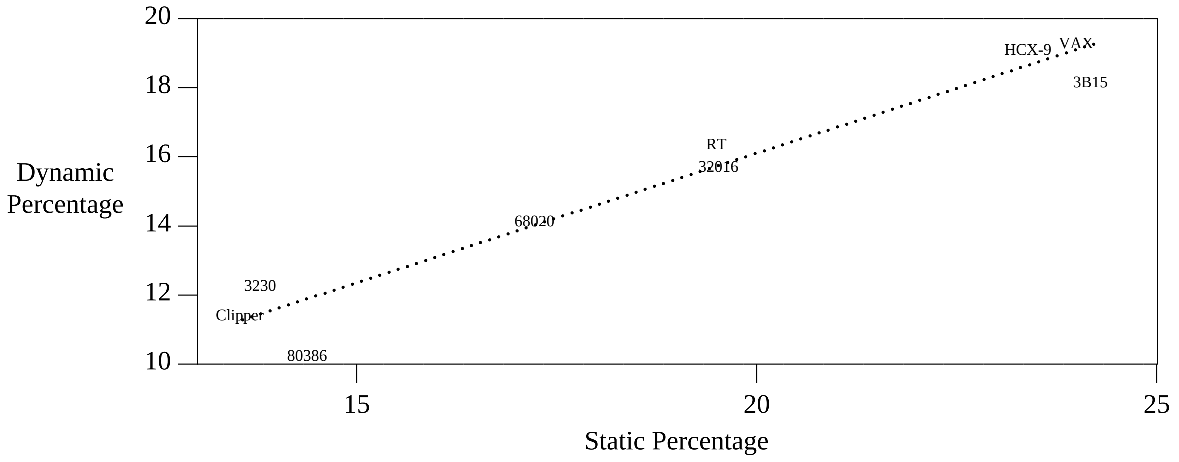
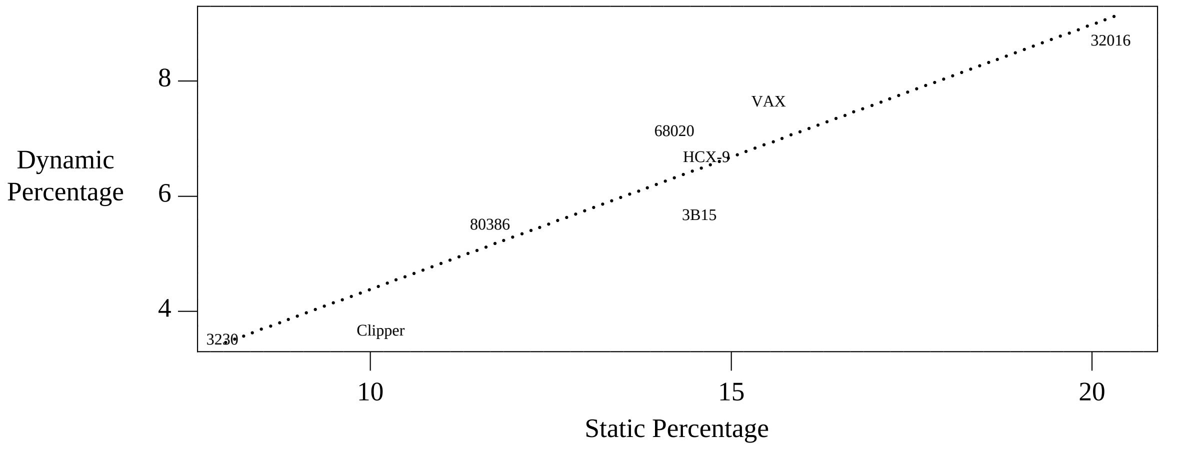


Figure 12. Regression of Dynamic Percentage of Direct Addressing Mode on Static Percentage of Direct Addressing Mode



VIII. References

1. W. G. Alexander and D. B. Wortman, Static and Dynamic Characteristics of XPL Programs, *Computer 8*,11 (November 1975), 41-46.
2. M. R. Barbacci, D. Siewiorek, R. Gordon, R. Howbrigg and S. Zuckerman, An Architectural Research Facility—ISP Descriptions, Simulation, Data Collection, *Proceedings of the AFIPS Conference*, Dallas, TX, June 1977, 161-173.
3. M. E. Benitez and J. W. Davidson, A Portable Global Optimizer and Linker, *Proceedings of the SIGPLAN Notices '88 Symposium on Programming Language Design and Implementation*, Atlanta, GA, June 1988, 329-338.
4. M. L. Berenson, D. M. Levine and M. Goldstein, *Intermediate Statistical Methods and Applications*, Prentice-Hall, Inc, 1983.
5. D. W. Clark and H. M. Levy, Measurement and Analysis of Instruction Use in the VAX-11/780, *Proceedings of the 9th Annual Symposium on Computer Architecture*, Austin, Texas, April 1982, 9-17.
6. J. W. Davidson and D. B. Whalley, Ease: An Environment for Architecture Study and Experimentation, *Proceedings SIGMETRICS '90 Conference on Measurement and Modeling of Computer Systems*, Boulder, CO, May 1990.
7. J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, San Mateo, CA, 1990.
8. *Precision Architecture and Instruction Reference Manual*, Hewlett-Packard, Inc., 1986.
9. M. Huguet, T. Lang and Y. Tamir, A Block-and-Actions Generator as an Alternative to a Simulator for Collecting Architecture Measurements, *Proceedings of the SIGPLAN Notices '87 Symposium on Interpreters and Interpretive Techniques*, St. Paul, Minnesota, June 1987, 14-25.
10. M. G. H. Katevenis, *Reduced Instruction Set Computer Architectures for VLSI*, PhD Dissertation, University of California, Berkeley, CA, 1983.
11. D. G. Kleinbaum and L. L. Kupper, *Applied Regression Analysis and Other Multivariable Methods*, Duxbury Press, North Scituate, MA, 1978.
12. A. Lunde, Empirical Evaluation of Some Features of Instruction Set Processor Architectures, *Communications of the ACM 20*,3 (March 1977), 143-153.
13. D. MacGregor and J. Rubinstein, A Performance Analysis of MC68020-based Systems, *IEEE Micro 5*,6 (December 1985), 50-70.
14. G. McDaniel, An Analysis of a Mesa Instruction Set Using Dynamic Instruction Frequencies, *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, CA, March 1982, 167-176.
15. I. Miller and J. E. Freund, *Probability and Statistics for Engineers*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1965.
16. M. J. Norusis, *SPSS/PC+*, McGraw-Hill, New York, NY, 1986.
17. B. L. Peuto and L. J. Shustek, An Instruction Timing Model of CPU Performance, *Proceedings of the 4th Annual Symposium on Computer Architecture*, Silver Spring, Maryland, March 1977, 165-178.
18. R. E. Sweet and J. G. Sandman, Jr., Empirical Analysis of the Mesa Instruction Set, *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, CA, March 1982, 158-166.
19. A. S. Tanenbaum, Implications of Structured Programming for Machine Architecture, *Communications of the ACM 21*,3 (March 1978), 237-246.
20. C. A. Wiecek, A Case Study of VAX-11 Instruction Set Usage for Compiler Execution, *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, California, March, 1982, 177-184.