Benchmark Characterization for Experimental System Evaluation

Thomas M. Conte Wen-mei W. Hwu

Center for Reliable and High-Performance Computing
University of Illinois
conte@csg.uiuc.edu

Abstract

Benchmarking in its various forms has become a popular approach to evaluating system performance and design decisions. There are at least three levels of behavior of a computer system measured by the benchmarking technique: the level of benchmark program and input set selection, the level of benchmark characteristics, and the level of system behavior in response to the benchmarks. Traditionally, only the system behavior has been measured. In order to make strong conclusion about benchmarking results, however, the nature of the benchmark programs must be characterized. This paper addresses this issue by presenting ways of measuring benchmark characteristics independent of system design. These benchmark characteristics include memory access behavior, control transfer behavior, and data dependencies. Measuring benchmark characteristics independent of the design parameters provides for cross-design and cross-architecture comparisons using the same benchmark set. They also serve as the basis for interpreting benchmarking results. Instruction memory access behavior and control transfer behavior are extracted from real programs and presented in this paper to illustrate the usefulness of benchmark characterization.

1 Introduction

The evaluation of the performance of a computer system typically uses the technique of *benchmarking*. In this technique, the performance of the system is measured for a set of programs, or *benchmarks*, that are executed on the actual system or a simulation of the system. Various forms of benchmarking, such as trace driven simulation, detailed simulation based on executable files, microcode-assisted measurement, hardware monitoring, and software probing, have been used extensively in the evaluation of experimental computer architectures, such as in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]

Traces are records of the dynamic system behavior in response to benchmarks. These traces are used as the input to simulators that generate system-level performance metrics [2, 3]. The amount of the collected information in these traces often makes measurement of long-term or global system behavior intractable. For example, on a contemporary 5 MIPS machine that accesses memory approximately twice per instruction, a second of real time results in a reference string of length 2 references/inst. $\times$ 5 MIPS $\times$ 4 bytes/reference $=\sim$ 40Mbytes long. This paper presents behavior measurement techniques based on information that is proportional to the static rather than dynamic program size. Instead of collecting traces, graph and density-function representations of the program are annotated with information from execution. This reduces substantially the amount of storage needed and increases the space of tractable system behavior measurement.

Benchmarking usually involves collecting a set of programs that are believed to be representative of the workload the system will encounter. Since some of these programs might process data, a set of representative inputs for each program is also collected. As each benchmark program is executed, it makes requests of the system that exercise various system features. These requests in turn force the system into certain behavioral patterns. This behavior of the system is then measured and interpreted as its performance. Hence, there are at least three levels of performance of a computer system measured by the benchmarking technique: the level of benchmark input set selection, the level of benchmark characteristics and compiler decisions, and the level of system behavior in response to the benchmarks. Traditionally, only the end-product performance of the system has been measured and this performance has been taken as the performance of the system in general.

0073-1129/90/0000/0006$01.00 © 1990 IEEE

This paper addresses this issue by presenting ways of measuring benchmark characteristics separate from system performance.

The performance of the system under benchmarks is often measured in terms of system-level, observable performance metrics, such as throughput and turnaround time [15, 16]. These metrics are functions of the characteristics of the benchmarks used for the performance study. Since they measure the system's performance, the metrics are also a function of the system design parameters. Measuring the characteristics of the benchmarks independent of the design parameters provides for cross-architecture comparisons using the same benchmark set. Additionally, the system-level performance metrics can potentially be approximated using analytical formulas involving the benchmark characteristics and the design parameters. These analytical formulas can provide insight into how benchmark performance and system design parameters impact the system's performance.

The remainder of this paper is divided into three parts. The following section discusses the methodology selected to measure benchmark characteristics. The third section presents implementation issues. An example of the approach is presented in section four. Finally, conclusions and future directions are presented.

2 A Methodology for Benchmark Characterization

This section presents the methodology of benchmark characterization. The benchmark characteristics proposed in this paper are selected to be general, highly architecture-parameter independent rulers by which the system's performance can be estimated. These characteristics, or *General Ruler Independent of Parameters, (GRIPs)*, are defined below.

2.1 Locality measures

An abstract reference stream of items is a time sequence, $w(t) = r_i$, over a set of possible item values, $r_i \in R$. The items (r_i's) may be the addresses of instructions or data items generated during the execution of a benchmark program, for example. Modern computers exploit the temporal and spatial locality behavior of reference streams by using special fast buffers to achieve high performance (e.g., cache memories) [1]. For this reason, many of the GRIPs presented below are based on locality measures.

Some definitions concerning reference streams will be required below:

DEFINITION 2.1: Define $\text{next}(w(t)) = k$, if k is the smallest integer such that $w(t) = w(t+k)$. ∎

DEFINITION 2.2: The number of unique references between $w(t)$ and $\text{next}(w(t))$, is defined as, $u(w(t)) = \|\{\, w(t+k) \mid i \leq k < \text{next}(w(t)) \,\}\|$. ∎

DEFINITION 2.3: Define $f^T(x)$, the *interreference temporal density function,* $f^T(x)$,, to be the probability of there being x unique references between successive references to the same item,

$$f^T(x) = \sum_t P\Big[\, u(w(t)) = x \,\Big].$$

∎

The interreference temporal density function is a measure of temporal locality of a reference stream. The performance of buffers managed under stacking replacement policies (e.g., LRU) depends directly on this measure of temporal locality. The hit ratio for a fully associative buffer of size N is $h(N) = \sum_{y \leq N} f^T(y)$ (see [17]).

DEFINITION 2.4: The *interreference spatial density function,* $f^S(x)$, is defined as,

$$f^S(x) = \sum_t \sum_{k=1}^{\text{next}(w(t))} P\Big[\, |w(t) - w(t+k)| = x \,\Big].$$

∎

The interreference spatial density function is a measure of the probability that between references to the same item, a reference to an item x units away occurs. Hence, the function captures the intrinsic interference between items in direct-mapped caches.

Another useful representation of a reference stream of items is as a. How to construct such a graph is illustrated in the following two definitions.

DEFINITION 2.5: The (directed) reference graph, $G = (V, E)$, of a reference stream is defined as $V = R$ and,

$$E = \{\, (r_i, r_j) \mid w(t) = r_i \text{ and } w(t+1) = r_j \,\}.$$

∎

DEFINITION 2.6: Let $n_i(r_i)$ be the number of occurrences $w(t) = r_i$, for $0 \leq t \leq T$. Furthermore, let $n_{ij}(r_i, r_j)$ be the number of occurrences of $w(t+1) = r_j$, if $w(t) = r_i$. Then, the weighted reference graph, $G' = (V, E)$, is defined such that each node, $r_i \in V$, is weighted with $P[r_i] = n_i/T$, and each edge, $(r_i, r_j) \in E$ is weighted with $P[r_j|r_i] = n_{ij}/n_i$. ∎

Based on these graph definitions, groups of items that are referenced together can be defined. The strongly-connected components of the reference graph, called the *phases,* are such partitions of the reference stream, outlined in the following definition.

DEFINITION 2.7: The set of phases for a reference stream is defined as $\Phi = \{\phi_1, \phi_2, \ldots \phi_i \ldots \phi_p\}$, where

$$\phi_i = \{ r_i \mid \{(r_i, r_{i+1}), (r_{i+1}, r_{i+2}), \ldots, (r_{k-1}, r_k), (r_k, r_i)\} \subseteq E \},$$

and, $\phi_1 \cap \phi_2 \cap \cdots \cap \phi_p = \emptyset$.

■

In a phase, any node can be reached from any other node through a sequence of edge traversals. During program execution, the items in a newly-encountered phase are guaranteed to not have been referenced before. Intrinsic cold-start buffer behavior can therefore be predicted using phase transitions, since the previous contents of a buffer are useless when a new phase is encountered. The *interphase density function* defined below is intended to capture this phase behavior of benchmarks.

DEFINITION 2.8: The interphase density function, $f^{\phi}(x)$, is the probability that a phase of size x is encountered in the reference stream,

$$f^{\phi}(x) = \sum_{\|\phi\|=x} \sum_{r_i \in \phi} P[r_i], \quad \text{for all } \phi \in \Phi.$$

■

2.2 Control flow GRIPs

The control flow behavior of a benchmark program can be characterized in terms of a reference stream of instructions, $w(t) = i_j, i_j \in I$, and its corresponding weighted reference graph, $G_I = (V_I, E_I)$. The instruction reference stream can be grouped into sets of instructions that must execute sequentially. These sets are called *basic blocks,* and the instruction reference stream can be redefined in terms of them, $w'(t) = B_i, B_i \in \mathcal{B}$. [18]. Some of the GRIPs for control flow are defined below in terms of the benchmark program's basic block weighted reference graph, $G_{BB} = (V_{BB}, E_{BB})$, also called the *weighted control graph* [19].

When the program is mapped into the linear memory space of a computer, the graph nature of the program is preserved using branch instructions. The graph nature still affects the performance of the system, especially for pipelined processors. Methods to reduce the penalty of this mapping have used both hardware and software approaches [10, 11, 12, 6, 13, 14, 8]. Software branch prediction schemes use the weights of the control flow graph to predict a branch's behavior to be either taken or not-taken for the duration of the program's execution [6, 8, 20]. It has been shown that these schemes perform as well as hardware schemes [8], yet the calculation of their performance is architecture-independent. The control flow GRIP *branch prediction accuracy, A,* is a variant of the accuracy of these software schemes.

DEFINITION 2.9: The prediction probability of B_i, $P_p(B_i)$ is defined as,

$$P_p(B_i) = \max\{ P[B_j | B_i] \mid (B_i, B_j) \in E_{BB} \}.$$

■

DEFINITION 2.10: The branch prediction accuracy, A, is defined as,

$$A = \sum_{i=1}^{N} P(B_i) P_p(B_i).$$

■

Hence, the branch prediction accuracy is the probability that a prediction based on the most likely behavior of a branch instruction is correct. Since some architectures have separate penalties for incorrectly predicting conditional- and unconditional branches, another GRIP, F_{CB}, is defined as the fraction of dynamic branches that are conditional branches.

The fetching of instructions in modern computers and the hardware-based prediction of branches often involve buffering [11, 6, 8]. The performance of instruction buffering techniques, such as instruction caches and branch target buffers, can estimated using the above locality measures for the instruction reference stream. Hence, $f_I^T(x)$ is defined to be the temporal locality GRIP and $f_I^S(x)$, the spatial locality GRIP for the instruction stream. Also, $f_I^{\phi}(x)$, the interphase density function, is included as a GRIP. It is important to mention an architecture-specific parameter that maps basic blocks into actual machine instructions. This parameter is the average length of basic blocks, $\overline{L}_{BB}$, and it is measured in terms of machine instructions instructions.

The GRIPs for control flow are summarized in Table 1. These GRIPs will be measured for benchmarks in an example characterization presented in Section 4.

2.3 Data flow GRIPs

The characterization of the data flow behavior of a benchmark program involves the concept of *variables.* A variable is a dynamic instance of a data item. The

Table 1: Control flow GRIPs

GRIP	Benchmark characteristic measured
A	Predictability of branches
F_{CB}	Fraction of conditional branches
$f_I^T(x)$	Instruction stream temporal locality
$f_I^S(x)$	Instruction stream spatial locality
$f_I^\phi(x)$	Instruction stream phase behavior

lifetime of variables, their locality, and the data dependencies that exist between them are the subject of this section.

Variables go through a life cycle in which they are created, used, and then discarded or written out. Register allocation is often performed using the technique of *graph coloring* [21, 22]. In this technique, a register is assigned to two different variables if the two variables are not live (i.e., active) at the same time. In essence, the number of registers required can be estimated by the *variable life density function.*

DEFINITION 2.11: Define the variable life density function, $f^{VL}(n_V)$, as the probability that n_V variables are live at any time during execution of the benchmark program.

∎

Hence, if there are m registers available for allocation by the compiler, then the register utilization will be $\sum_{i \leq m} f^{VL}(i)$, and the amount of spill code required will be $\sum_{i>m} f^{VL}(i)$. (This is similar to an approach described in [23].) The number of live variables can be measured using techniques described in Section 3.

Since buffering is used for data accesses, a set of GRIPs is defined for the locality of data references. Define $f_D^T(x)$ to be the interreference temporal density function, and $f_D^S(x)$ as the interreference spatial density function for the data reference stream. Note that unlike the instruction stream, the variable life density function must be used in conjunction with the locality density functions to predict the performance of buffers after register allocation. Also, phase behavior will be measured with, $f_D^\phi(x)$, the interphase density function for the data reference stream.

The data dependence behavior of a benchmark program can be captured using a *instruction dependence graph.* In this graph, the nodes are the (compiler-intermediate) instructions, $i_j \in I$, and the edges are due to flow dependencies. The following definition states this more formally.

DEFINITION 2.12: If $\mathcal{R}(i_j)$ is the set of variables read by instruction $w(t_1) = i_j$, and $\mathcal{W}(i_k)$ is the set of variables written by instruction $w(t_2) = i_k$, for $i_j, i_k \in I$, and $t_1 < t_2$, then, the instruction dependence graph is a graph, $G_{ID} = (V_I, E_{ID})$, such that $V_I = I$ and

$$E_{ID} = \{ (i_k, i_j) \mid \mathcal{W}(i_j) \cap \mathcal{R}(i_k) \neq \emptyset \}$$

(see [18]).

∎

The dynamic scheduling of instructions using an algorithm such as Scoreboarding or the Tomasulo algorithm is dictated by the structure of the dynamic data dependencies [24, 25, 4, 5]. A possible GRIP to capture the schedulability of a benchmark would be the probability of there being dependencies of distance i intermediate instructions. However, the overlap of dependencies and the branch behavior of the instruction stream is not captured by this GRIP. Emma and Davidson present a set of reductions that can be performed on the instruction dependence graph to eliminate overlap. After these reductions, the probability of a dependence spanning j taken branches while having a distance of i intermediate instructions is sufficient to characterize the performance of out-of-order execution schemes [26, 27]. This statistic is the probability of such a dependence occurring, $p_{i,j}^{DD}$, and can be calculated from the instruction dependence graph after a set of graph reductions are performed [27]. This then will serve as the scheduling GRIP for data flow.

The GRIPs for data flow are summarized in Table 2.

2.4 Other GRIPs

Three areas of benchmark characterization that are not covered in detail in this paper deserve some discussion. These three areas are operating system performance, I/O system performance, and large-grain parallelism.

The performance of a benchmark program under the interruptions and scheduling policies of an operating system is different from that of the program running alone. There are two ways of viewing this

Table 2: Data flow GRIPs

GRIP	Benchmark characteristic measured
$f^{VL}(n_v)$	Live variables/register use
$f_D^T(x)$	Data stream temporal locality
$f_D^S(x)$	Data stream spatial locality
$f_D^\phi(x)$	Data stream phase behavior
$p_{i,j}^{DD}$	Data dependence schedulability

interaction. From the operating system viewpoint, the benchmark program is actually an input. Hence, the operating system may be thought of as a meta benchmark program with the characteristics of the benchmarks running under it as its input set. From the benchmark program's viewpoint, the operating system satisfies requests and disturbs buffer usage. These two effects can be characterized using selective flushing of buffers via simulated multitasking, as in [1, 28]. Therefore, an operating system can be viewed as a benchmark and characterized using the same GRIPs. Additionally, the multitasking quantum can be used as a system parameter to modify the role the locality measures play in the approximations of system-level performance parameters.

The I/O system's performance is similar to the operating system in the sense that it views the entire set of benchmark programs as its input set. Since the sequence of references to a peripheral determines its performance, the performance of peripherals has to be modeled for each peripheral architecture. This is analogous to the modeling of cache behavior by locality measures, since cache behavior also depends on reference stream sequencing. Hence, the characterizations are tractable but beyond the scope of this paper.

Large-grain parallelism is usually expressed explicitly by the programmer as a conscious decision. Measuring this parallelism can be done by intercepting the synchronization primitives and then constructing the expressed dynamic parallelism. Again, these characterizations are tractable but beyond the scope of this paper.

3 Implementation issues

Several ways of measuring GRIPs are available. For example, microcode-based measurement techniques exist that modify the microcode of a machine to monitor the instruction stream [2, 3]. In essence, the traces generated by these techniques are reference streams of items. These streams can be analyzed to produce reference graphs and identify phases. However, the length of the traces are excessive.

This paper uses techniques where the reference graph is constructed on-the-fly without the intermediate step of recording the reference string. Previous work has implemented the construction of the control flow graph on-the-fly using the compiler to insert probe instructions at the entrance of each of the program's basic block. As the program executes, the weighted control graph is constructed and stored for later analysis [19, 8, 29].

This paper proposes an extension to the compiler-based profiling technique to measure the instruction dependence graph and the locality measures. Static analysis of dependence information can be done at compile time to produce a list of variables that are born and killed for each basic block [18]. However, static dependence analysis cannot deal efficiently with variables that span function call invocations and aliased pointer references [30]. These instances cause unknowns to appear in the static dependence information. The compiler-based dependence profiler inserts probe instructions at the site of these unknowns to measure their variables. As each basic block is executed, a script describing the birth and death of variables is executed by the profile analyzer. The unknowns are represented as 'wait for variable identity' commands in this script that instruct the analyzer to insert the identity of the variable in its dynamic copy of the script. As each basic block completes execution, the analyzer uses the dynamic copy of the script to update the instruction dependence graph that it builds. Also measured using this technique is the variable life density function, f^{VL}.

After the weighted control graph has been constructed, branch behavior can easily be measured. The branch prediction accuracy, A, can be calculated directly from the weighted control graph using the equation from Section 2.2. The dynamic fraction of conditional branches, F_{CB}, can be calculated by summing the weights of the basic blocks in the weighed control graph that end with conditional branches.

Phases and cycles in the control flow and dependence graphs can be detected during the execution

of the program using a stack of recently-seen items. A separate stack maintained using the least-recently-used replacement policy (see [17]) can be employed to find the interreference temporal and spatial density functions. The algorithm for locality measurement

```
Calc_loc_measures(r_i):
begin
    if not first time r_i encountered then
    begin
        d ← depth(r_i)
        remove r_i from the stack
        for all r_j with depth(r_j) < d
        begin
            dist ← |α(r_j) − α(r_i)|
            f^S(dist) ← f̂^S(dist) + 1
        end
        f̂^T(d) ← f̂^T(d) + 1
    end
    push(r_i)
end
```

Figure 1: The algorithm for calculating the locality distributions.

is outlined in Figure 1, where $\alpha(\cdot)$ is the address of a node, and **depth**$(\cdot)$ is the stack depth of a node. The approximate distributions, $\hat{f}^S(x)$ and $\hat{f}^T(x)$ are normalized after execution terminates.

4 Example benchmark characterization

As an illustration of the benchmark characterization idea, this section presents the control flow GRIPs for several benchmarks. The benchmark programs were selected to be highly data-driven so as to make their control flow behavior very diverse. The benchmarks are presented in Table 3. A description of the inputs that were used for the programs is also presented.

The GRIPs presented in Table 1 were measured using the techniques outlined in Section 3. The scalar GRIPs, A and F_{CB}, are presented in Table 4. The locality measures, $f_I^T(x)$, $f_I^S(x)$ and $f_I^\phi(x)$ are presented in graph form in Figures 2, 3, 4, and 5.

Table 4: Scalar control flow GRIPs

Benchmark	A	F_{CB}
grep-c	99.0%	30.5%
grep-words	94%	37%
yacc-awk	95%	47%
yacc-make	98%	37%

The value of A shows that all these benchmarks have highly predictable branches. For example, a typical static branch in *yacc-awk* can be correctly predicted 95% of the time by always choosing the most preferred direction. Hardware or software branch prediction mechanisms should be able to achieve this prediction accuracy. Any result significantly less than this value indicates the existence of system performance problems. In the case of hardware prediction schemes, the problem may be due to either insufficient branch target buffer entries or frequent context switches. The measured instruction-stream locality can then be used to estimate by how much the branch target buffer size should be increased. As for software prediction schemes, the problem may be due to the use of inaccurate profile information. With the A values, one knows what to expect from the measured branch prediction performance. Also, with such a high predictability, software and hardware prediction schemes can be expected to exhibit the same behavior. This is confirmed by the measurements presented in [8].

The spatial locality measure of *yacc-awk* (Figure 4) indicates that it is highly sequential. The likelihood of a basic block reference being within 30 basic blocks of any other reference is very high. This is of course dependent on the code layout decision made by the compiler. The code layout used in this measurement is intelligently done based on profile information. Therefore, one can expect the spatial locality to be lower for the same benchmark when compiled by a less intelligent compiler. With such a high spatial locality, one can expect the instruction buffers and caches with large blocks to perform well.

The temporal locality of *yacc-make* indicates that a cache which accommodates approximately 15 basic blocks will accommodate its working set. The entire program consists of almost 1300 basic blocks. With only 1% of the program active at a time, *yacc-make* has very high locality. With five instructions per basic block and four bytes per instruction, a 0.5KB cache will be adequate for accommodating the working set. Therefore, one expects to find the performance of instruction cache to saturate when the cache size increases beyond 0.5K bytes. Again, one knows what to expect before performing any architecture-parameter-specific measurement.

It is interesting to correlate the spatial and temporal localities. Although the results for *grep* are not strongly correlated, there is a strong correlation between the spatial and temporal locality measures for *yacc-awk* and *yacc-make*. This phenomenon is due in part to the intelligent compiler code layout scheme, *trace layout*. This scheme emits instructions in the

Table 3: The programs studied

Benchmark	Description	Input description	# Basic blocks
grep-c	*grep*: A general regular-	`grep -c '++' grep.c`	116
grep-words	expression parser	`grep -i '[aeiou]{2,4}' /usr/dict/words`	125
yacc-awk	*yacc*: a LALR(1)	The grammar for *awk*	1307
yacc-make	parser generator	The grammar for *make*	1293

order of their execution based on the program's performance for a large input set [31, 19, 7]. Hence, references a certain distance apart in time tend to be the same distance apart in space.

The comparison of the temporal locality for the same benchmark program using different inputs is also interesting. For example, *grep-c* (Figure 2) has 33% of its references separated by five unique references, whereas *grep-words* has only 11%. On the other hand, *grep-c* has $f_I^T(22) \approx 0$, whereas *grep-words* has $f_I^T(22) \approx 18\%$. Though the temporal localities are input dependent at these two points, buffers of size greater than 50 basic block lengths will perform equally well for both benchmarks. Hence, if a design decision is made to be robust, it will be insensitive to the benchmark program input selection.

Finally, the interphase density functions show a preference for large phases. For example, $f_I^\phi(79) = 99\%$ for *grep-c*. A similar execution probability occurs for a phase of size 513 for *yacc-awk*, and at a size of 74 for *grep-words*. This indicates that the penalty of intrinsic cold-start misses would be very high, but infrequent. However, this also suggests that context switching would have a large impact on buffer performance. Perhaps further subdividing phases by using the weights of the weighted control graph might provide more insight the effect of context switching.

5 Conclusions

This paper presents a method for characterizing benchmark programs. Two key features distinguish the ideas presented in this paper from those presented in the past. One is that all the characteristics are stored as data structures whose sizes are proportional to that of the static size of the benchmark program. In contrast, trace driven simulation is based on the analysis of dynamic execution traces whose size is proportional to the dynamic instruction count of the benchmark program. This makes it possible to characterize each benchmark with many realistic inputs. The other key feature is the separation of benchmark characteristics from architecture-specific parameters. Benchmark characterization is presented as a technique that can provide some insight into the performance of a system without actually having to simulate the system. This provides a uniform ground for comparing different architectures. It is also a tool to help interpret the results of simulation and measurements.

Several areas of benchmark characterization were presented. Control flow and data flow characterizations were explained in detail. To illustrate the ideas, the results of control flow characterization were presented and discussed. The control flow characteristics presented include branch predictability, instruction stream spatial-, and temporal locality locality. The implication of these characterization results on the evaluation of architecture design decisions were also presented.

A profiler and its supporting software tools are being constructed to implement the ideas presented in this paper. The extraction of control flow GRIPs has been completed and was used to derive the results presented in Section 4. The extraction of data flow GRIPs is under development. Once completed, this profiler will be distributed to the architecture research community.

The authors would like to thank Sadun Anik, David Griffith and all members of the IMPACT research group for their support, comments and suggestions. This research has been supported by the National Science Foundation (NSF) under Grant MIP-8809478, a donation from NCR, the National Aeronautics and Space Administration (NASA) under Contract NASA NAG 1-613 in cooperation with the Illinois Computer laboratory for Aerospace Systems and Software (ICLASS), and the Office of Naval Research under Contract N00014-88-K-0656.

References

[1] A. J. Smith, "Cache memories," *ACM Computing Surveys*, vol. 14, no. 3, pp. 473–530, 1982.

[2] S. W. Melvin and Y. N. Patt, "The use of microcode instrumentation for development, debugging and tuning of operating system kernels,"

in *Proc. ACM SIGMETRIC' '88 Conf. on Measurement and Modeling of (>mput. Sys.*, (Santa Fe, NM), pp. 207–214, May 988.

[3] A. Agarwal, R. L. Sites, and M. Horowitz, "ATUM: a new technique for capturing address traces using microcode," in *Proc. 13th Annu. Symp. on Comput. Arch.*, pp. 119–127, June 1986.

[4] N. P. Jouppi and D. W. Wall, "Available instruction-level parallelism for superscalar and superpipelined machines," in *Proc. Third Int'l Conf. on Architectural Support for Prog. Lang. and Operating Systems.*, pp. 272–282, Apr. 1989.

[5] M. D. Smith, M. Johnson, and M. A. Horowitz, "Limits on multiple instruction issue," in *Proc. Third Int'l Conf. on Architectural Support for Prog. Lang. and Operating Systems.*, pp. 290–302, Apr. 1989.

[6] S. McFarling and J. L. Hennessy, "Reducing the cost of branches," in *Proc. 13th Annu. Symp. on Comput. Arch.*, (Tokyo, Japan), pp. 396–403, June 1986.

[7] W. W. Hwu and P. P. Chang, "Achieving high instruction cache performance with an optimizing compiler," in *Proc. 16th Annu. Symp. on Comput. Arch.*, (Jerusalem, Israel), pp. 242–251, June 1989.

[8] W. W. Hwu, T. M. Conte, and P. P. Chang, "Comparing software and hardware schemes for reducing the cost of branches," in *Proc. 16th Annu. Symp. on Comput. Arch.*, (Jerusalem, Israel), pp. 1–1, June 1989.

[9] D. Ferrari, G. Serazzi, and A. Zeigner, *Measurement and tuning of computer systems.* Prentice hall, 1989.

[10] J. E. Smith, "A study of branch predition strategies," in *Proc. 8th Annu. Symp. on Comput. Arch.*, pp. 135–148, June 1981.

[11] J. K. F. Lee and A. J. Smith, "Branch prediction strategies and branch target buffer design," *IEEE Computer*, Jan. 1984.

[12] D. R. Ditzel and H. R. McLellan, "Branch folding in the CRISP microprocessor: reducing branch delay to zero," in *Proc. 14th Annu. Symp. on Comput. Arch.*, pp. 2–9, June 1987.

[13] J. A. DeRosa and H. M. Levy, "An evaluation of branch architectures," in *Proc. 14th. Annu. Symp. on Comput. Arch.*, pp. 10–16, June 1987.

[14] D. J. Lilja, "Reducing the branch penalty in pipelined processors," *IEEE Computer*, July 1988.

[15] J. P. Buzen, "Fundimental operational laws of computer system performance," *Acta Informatica*, vol. 7, pp. 167–182, 1977.

[16] H. M. Levy and D. W. Clark, "On the use of benchmarks for measuring system performance," *Computer architecture news*, Dec. 1982.

[17] R. L. Mattson, J. Gercsei, D. R. Slutz, and I. L. Traiger, "Evalutation techniques for storage hierarchies," *IBM Systems J.*, vol. 9, no. 2, pp. 78–117, 1970.

[18] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers, principles, techniques, and tools.* Reading, MA: Addison-Wesley, 1986.

[19] W. W. Hwu and P. P. Chang, "Trace selection for compiling large C application programs to microcode," in *Proc. 21st Annu. Workshop on Microprogramming and Microarchitectures*, (San Diego, CA.), Nov. 1988.

[20] W. W. Hwu and P. P. Chang, "Forward Semantic: a compiler-assisted instruction fetch method for heavily pipelined processors," in *Proc. 22st Annu. Workshop on Microprogramming and Microarchitectures*, (Ireland), Nov. 1989.

[21] G. J. Chaitin, M. A. Auslander, A. K. Chandra, and J. Cocke, "Register allocation via coloring," *Computer languages*, vol. 6, pp. 47–57, 1981.

[22] G. J. Chaitin, "Register allocation & spilling via graph coloring," in *Proc. ACM SIGPLAN '82 Symp. on Compiler Construction*, pp. 98–105, 1982.

[23] G. D. McNiven and E. S. Davidson, "Analysis of memory reference behavior for design of local memories," in *Proc. 15th. Annu. Symp. on Comput. Arch.*, pp. 56–63, June 1988.

[24] J. E. Thornton, "Parallel operation in the Control Data 6600," in *Proc. AFIPS FJCC*, pp. 33–40, 1964.

[25] R. M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM Journal of Research and Development*, vol. 11, pp. 25–33, Jan. 1967.

[26] P. G. Emma, *Discrete-time modeling of pipeline computers under flow perturbations.* PhD thesis,

Department of Electrical and Computer Engineering, University of Illinois, Urbana, Illinois, 1984. Coordinated Science Laboratory Computer Systems Group Report No. CSG-27.

[27] P. G. Emma and E. S. Davidson, "Characterization of branch and data dependencies in programs for evaluating pipeline performance," RC 11733 52535, Computer Science Dept., IBM T. J. Watson Research Center, Yorktown Heights, NY, 1986.

[28] A. Agarwal, M. Horowitz, and J. Hennessy, "An analytical cache model," *ACM Trans. Computer Systems*, vol. 7, pp. 184–215, May 1989.

[29] C. B. Stunkel and W. K. Fuchs, "TRAPEDS: producing traces for multicomputers via execution driven simulation," in *Proc. ACM SIGMETRICS '89 and PERFORMANCE '89 Int'l Conf. on Measurement and Modeling of Comput. Sys.*, (Berkeley, CA), pp. 70–78, May 1989.

[30] J. M. Barth, "A practical interprocedural data flow analysis algorithm," *J. ACM*, vol. 21, pp. 724–736, Sept. 1978.

[31] J. R. Ellis, *Bulldog: a compiler for VLIW architectures*. Combridge, MA: The MIT Press, 1986.

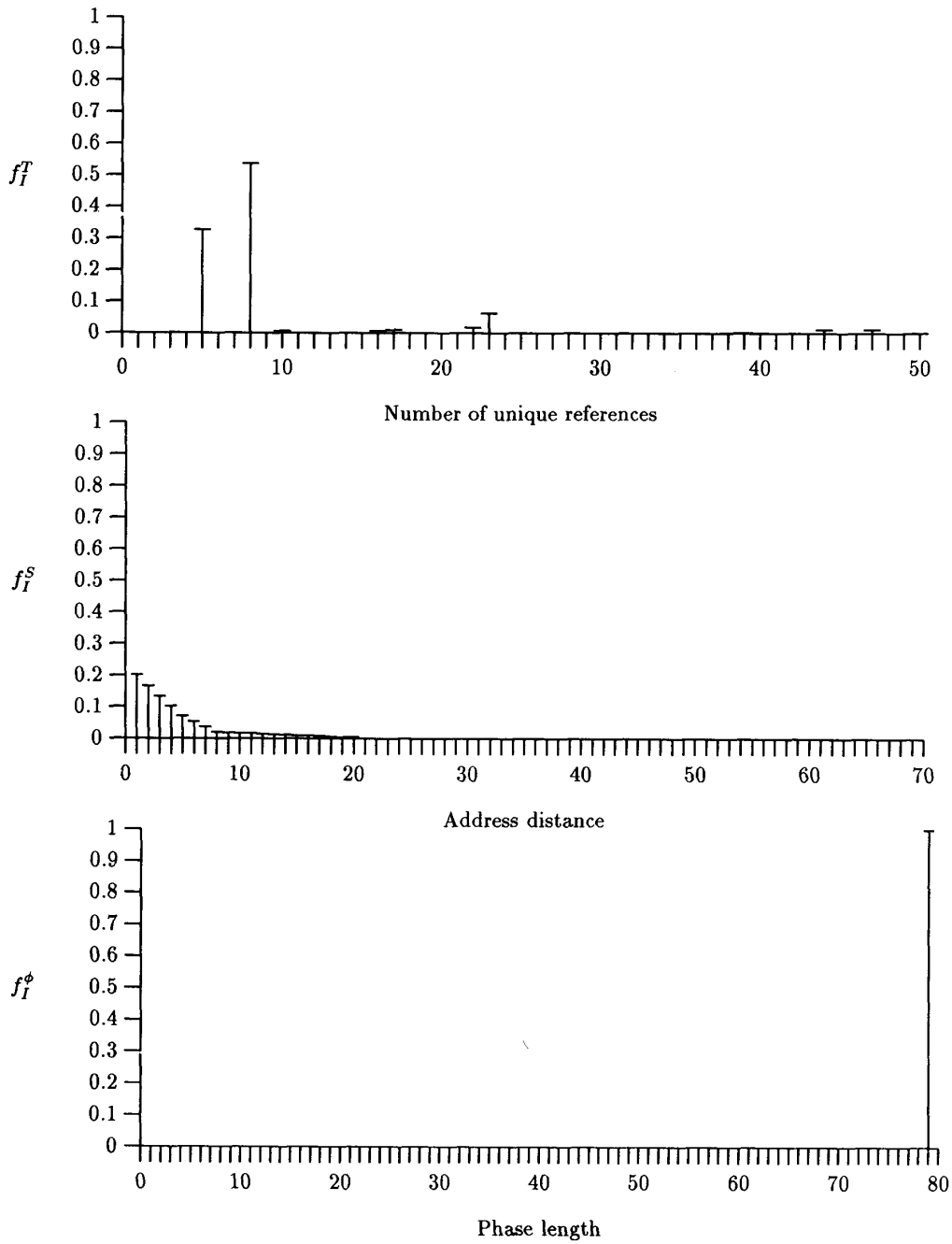

Figure 2: The locality measures, $f_I^T(x)$, $f_I^S(x)$, and $f_I^\phi(x)$, for *grep-c*

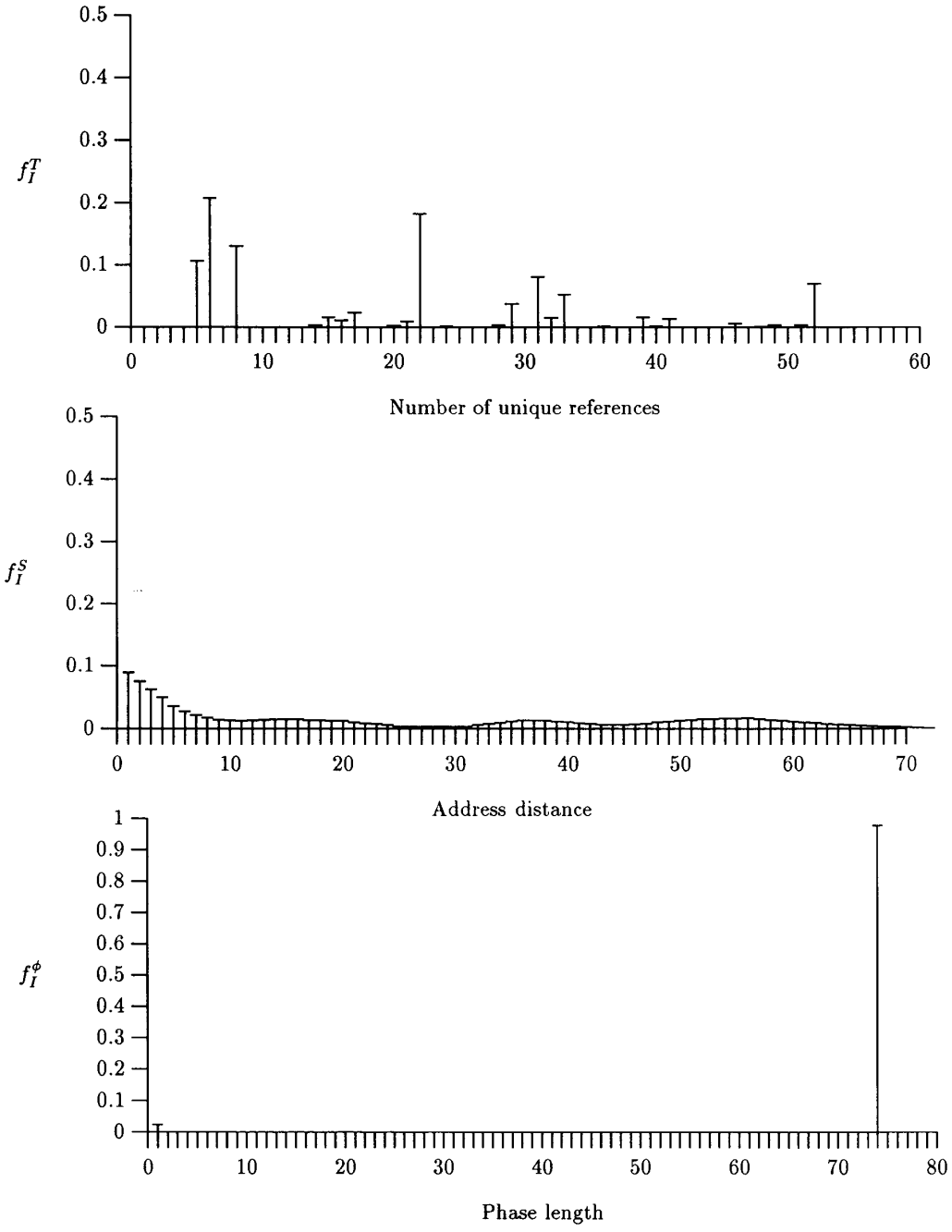

Figure 3: The locality measures, $f_I^T(x)$, $f_I^S(x)$, and $f_I^\phi(x)$, for *grep-words*

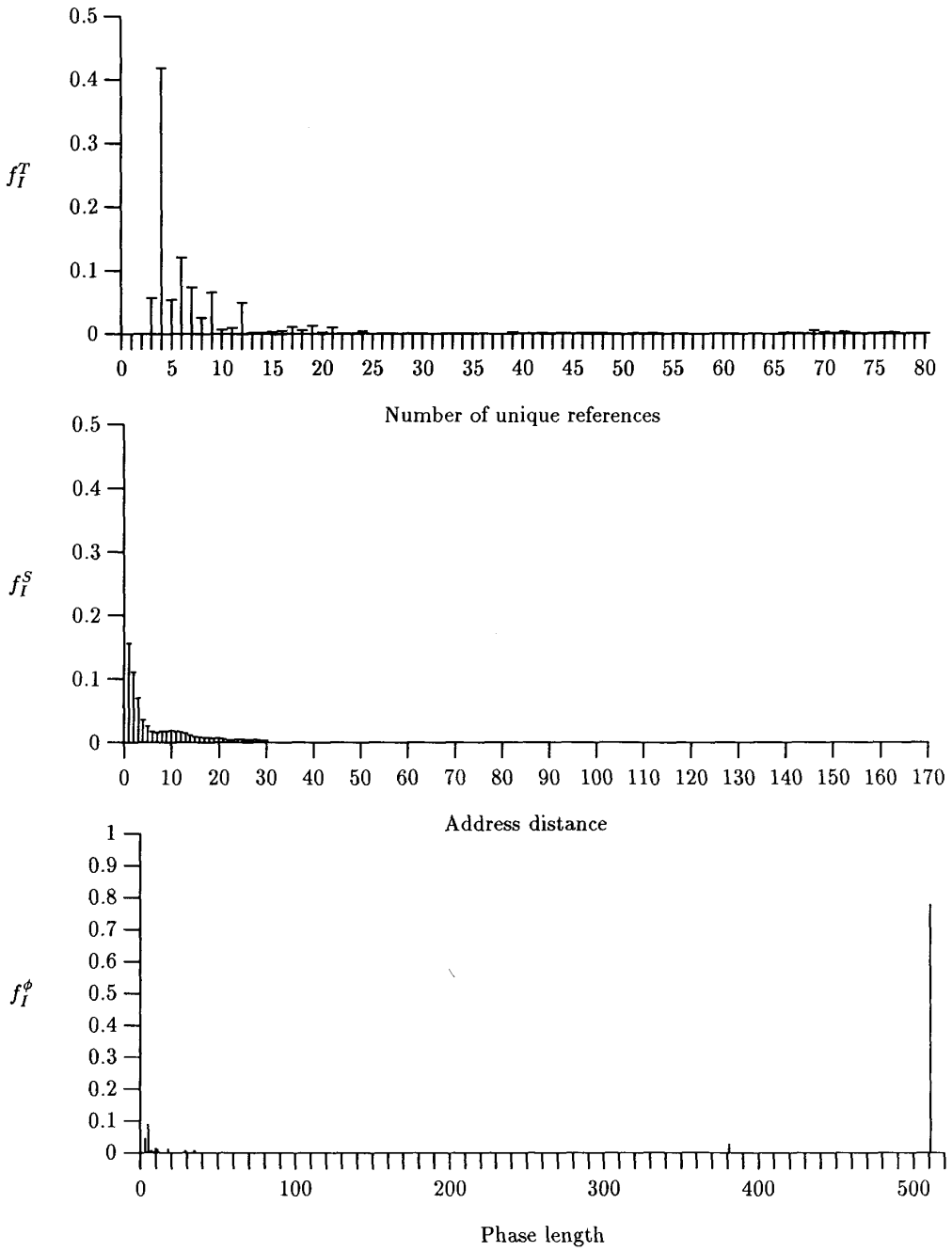

Figure 4: The locality measures, $f_I^T(x)$, $f_I^S(x)$, and $f_I^\phi(x)$, for *yacc-awk*

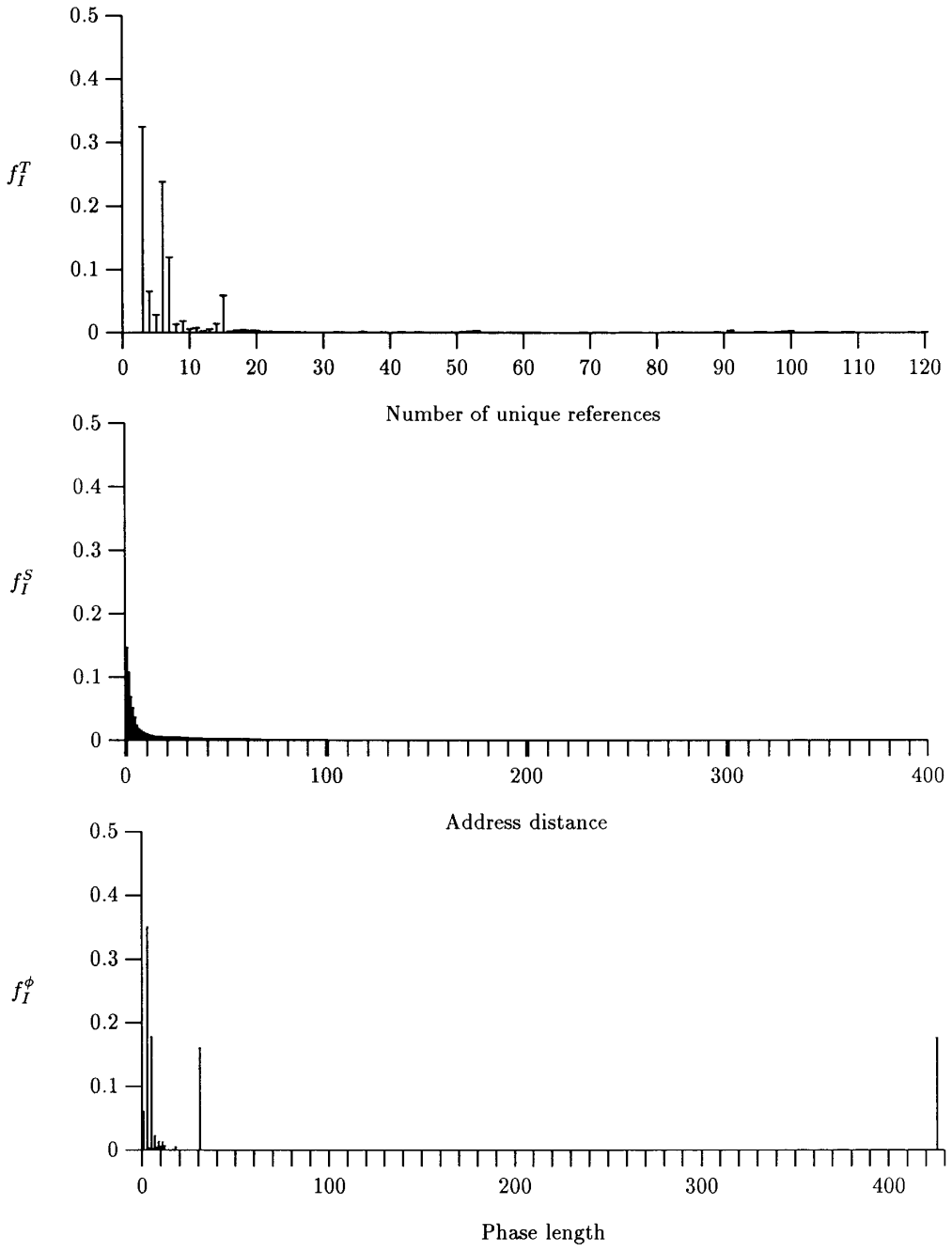

Figure 5: The locality measures, $f_I^T(x)$, $f_I^S(x)$, and $f_I^\phi(x)$, for *yack-make*