U.S. DEPARTMENT OF COMMERCE
National Institute of Standards and Technology

NISTIR 90-4275

National Computer Systems Laboratory

Workloads, Observables, Benchmarks and Instrumentation

G.E. Lyon
R.D. Snelick

U. S. DEPARTMENT OF COMMERCE
National Institute of Standards and Technology
National Computer Systems Laboratory
Advanced Systems Division
Gaithersburg, MD 20899

April 1990

CMRF

COMPUTER MEASUREMENT
RESEARCH FACILITY
FOR HIGH PERFORMANCE
PARALLEL COMPUTATION

Partially sponsored by
- Defense Advanced Research Projects Agency
- Department of Energy

Workloads, Observables, Benchmarks and Instrumentation

G.E. Lyon
R.D. Snelick

Advanced Systems Division
National Computer Systems Laboratory
National Institute of Standards and Technology
Gaithersburg, MD 20899

Partially sponsored by
- Defense Advanced Research Projects Agency
- Department of Energy

To appear in the proceedings of the Joint Conference on Vector and Parallel Processing, Zurich, September, 1990.

U.S. Department of Commerce, Robert A. Mosbacher, Secretary

National Institute of Standards and Technology
John W. Lyons, Director

April 1990

TABLE OF CONTENTS

Page

Static Characterizations 2

The System as Tree-Structured Filter 4

A Paradigm 4
Paths=Dependencies 5
Fanouts=Competitions 5
Use-Tree with Global Clock 6
Use-Tree with Local Clocks 6
Workload, Partitions, Weights 7

Skeletons of Real Systems 8

An Averaged Model for Vector Pipeline Systems 8

Application Examples 9

I--Altering Tree Admittances 9

II--Answering to the Tree 10

Local time 11
A Ring Benchmark 11

Summary and Prospects 13

Future Directions 14
Acknowledgment 14

Citations 14

Workloads, Observables, Benchmarks and Instrumentation

G.E. Lyon
R.D. Snelick

Main emphasis is upon a compact user-level summary that captures the performance variabilities of a system. A dependency tree provides a clear, static declaration of the relationships among a very limited number of major system resources that explain most performance variance. This approach is especially effective with hardware instrumentation that decouples workload from observation, for otherwise a compact set of observables may be unavailable. The tree supports simple predictions and promotes more meaningful comparisons of workloads. Accounting for the sources of performance variation shown in the tree can inspire new methods of assessment; a "time dilation" technique illustrates this for loosely-coupled systems with local clocks.

Key words: application; architecture; benchmarks; components; models; performance.

Measurement and modeling are as intertwined in computer performance characterization as they are for other experimental fields. Without good measurements, confidence in a model remains unsubstantiated. And without some model of features, measurements are disembodied values of uncertain worth. Performance comparisons of benchmarks on a newer architecture should reveal in simple terms the major reasons for any differences, but this is not always achieved. Good quality measurements are not always available, for many systems have very little to aid performance analysis. Other complications arise from characterization. Intricate performance models evade the issue of generality amid a screen of detail; they have no economy of description. The approach that follows emphasizes statistical formulations for quick, useful comparisons at the application level.

Main emphasis is upon a compact user-level format for expressing variable elements in a system's performance. The format, a tree, expresses performance alternatives that arise from underlying system mechanisms. However, while details such as scheduling, processor interference and the like are mentioned in the text, these should be seen primarily as illustrative sources of *performance variance* and its attendant properties. The mechanisms as described may lack fidelity to real cases, having been simplified for discussion.

 Partially sponsored by the Defense Advanced Research Projects Agency, ARPA Task No. 7066, and the Department of Energy, DoE Order No. DE-AI05-87ER25046. A mild abridgment of this report has been accepted for the Joint Conference on Vector and Parallel Processing, **CONPAR 90 - VAPP IV**, Swiss Federal Institute of Technology, Zurich, September, 1990.

Modern computers herald a profusion of new designs, many with parallel processing. But a broad understanding of system capabilities, *vis-a-vis* applications, has lagged behind *ad hoc* opportunities to build hardware. Consequently, there arise wide disparities between expected (or promised) performances and those actually realized. Common benchmark sets show no strong correlation among each other, nor do they generally represent a specific application of interest [REE89b]. The complexity and extreme variety of algorithms (and whole benchmark programs) weighs against their use as atomic units of measure, yet often benchmark program X is run and measurement numbers are in terms of X as an undifferentiated unit of workload. The relationship of X to another test program Y is not immediately (if ever) evident. As a measurement methodology, this coarse level of characterization is unsatisfactory.

P. Denning and G. Adams III have catalogued a number of important open research issues in performance characterization [DEN88]. Within the spirit of their list, an examination of stand-alone benchmarks might include simplified but reasonably accurate methods of: analyzing average execution rates, comparing distinct benchmarks, predicting gains from system upgrades and identifying likely application-system compatibilities. These objectives involve independent variables (algorithmic aspects), dependent variables (observed events), and systems. The perspective on these aspects throughout this examination assumes special instrumentation to:

1. Avoid making predictions from algorithmic analyses.
2. Apply an empirical approach in which distinct, constrained effects are measured.
3. Limit the number of effects (observables) to a manageable, easily interpreted set.

Item 1 is important because analyses could become an endless burden. Furthermore, prediction rules, especially on newer parallel systems, are not always available. Fortunately, benchmark programs fix their parameter settings at standard reference values. The importance of predicting from independent variables is thus much diminished. Item 2 is not quite as simple as it seems. Observables should be separable, *i.e.*, free of overlap, and should obey some rule(s) of consistency. In this manner, structure is imposed upon experiments, and checks can be made. The third point involves discovering which system effects are important, and measuring them.

Static Characterizations

Typical workload on a multiprogrammed computer is a melange that is modeled stochastically [HAR83]. The resultant dynamic description can be quite complex. A static view of workload has attractions because it greatly simplifies, but relying as it does upon aggregations and averages, it can fail to reflect accurately even one phase from a heterogeneous workload mix. However, a benchmark program provides possibilities for simplification. It is for practical purposes, and by definition, free of influences. This is particularly true on higher performance machines, where benchmarks are simplified versions of jobs that occupy the whole machine. The benchmark run is also closed end, that is, it has definite start and termination conditions expressed in its standard

parameters. Answers, phases of computation, durations--all of these are more predictable. Benchmark conditions, similar to isolated phases of interactive workloads (*e.g.*, [HAR83], p. 148), admit a *static* description of resource demands.

Attempts to define experiments for specially built instrumentation hardware (attached to both shared-memory and distributed-memory machines) show that many benchmark tests return less than satisfactory general knowledge [LYO89]. Yet benchmarks are quite useful, *e.g.*, in system procurement. Their disappointment lies more in narrow expectations than in any inherent nature. The *workload* that a benchmark presents to a system is distinct from *observation* of the system with this workload. Sometimes workload and observation become tied together through measurement limitations. Many systems have only a high-perturbation coarse clock, so that benchmarks are simply timed, start-to-finish. But since it is likely that performance is determined by two or more system components, a finer resolution--more observables with less overhead--is needed to explain performance variances.

Resolving workload components with benchmarks and elapsed time *alone* can lead to small specialized kernels that execute in unrealistic ways [UNI89]. Workload from a whole application represents a system's computation task well, but unfortunately a single observable (elapsed time) resolves system capabilities best with specialized workloads. The small, homogeneous kernel benchmark is really a secondary instrumentation technique; unfortunately, it must have narrow workload characteristics to achieve its purpose of measuring specialized performance capabilities. Adequate instrumentation can help immensely, separating workload from observation. A low overhead, high resolution clock on each processor is often sufficient. The challenge is then to discover what nature all workloads share. Certainly distinct jobs on the same machine exhibit some common dimensions of hardware and system use [HAR83]. *Resource demands* can be compared. The chart below contrasts typical benchmarking practices (first column) against system instrumentation (second column).

TYPICAL OBSERVABLES IN PRACTICE

WORKLOAD	One, of high overhead (software benchmarking)	Many, of low overhead (hardware instrumentation)
Large "code"	*typical test for machine procurement at installation*	*quantify in detail resource demands of application*
Small "kernel"	*assumes workload conditionings have little influence*	*attempt calibration of secondary-standard kernel metric*

The System as Tree-Structured Filter

Abstractly, computing can be viewed as system components responding to demands of an application, or alternately, as classes of system service partitioning the application workload. The latter provides a static perspective for performance measurement. Furthermore, the "partitioning" side of the duality is a good anchoring point; although hypothetically as varied as applications, only a tiny fraction of all possible systems is ever manufactured. In contrast, actual applications have a richness that is taken for granted. Thus, actual systems define a clear starting point for a static modeling paradigm:

1. Decompose each system into observable, homogeneous service components that determine general performance. Components are important statistical effects (observerables) that identify more than just hardware pieces.
2. Build workload signatures from application demands upon dominant service components.
3. Structure 1 and 2 (above) to reflect performance dependencies in the system.

General aspects of a machine's capabilities include processor bandwidths, i/o, memory-to-processor bandwidths, processor-to-processor communication, and memory-to-memory bandwidth, as well as memory sizes. While ascertaining specific service components may be a straightforward interpretation of the hardware and architecture, this is hardly guaranteed. Machines hold surprises in capabilities established not through obvious architectural features, but through synergistic strengths and weaknesses of functional groups, including compilers, loaders and schedulers. Features with latencies (delays) are often important bottlenecks. Heavily used resources are important, but so are lightly used resources that are slow. Some characteristics will have importance only in context. Statistical screening methods help select the meaningful from among the many possibilities [BHH78].

A Paradigm. The modeling paradigm will be a dependency tree that is decorated with measurement values [LYO89b]. For distributed systems with local clocks, the values will be times or percent of total service times. Shared memory with global time has another convention that uses service demand and service rate. Since demand divided by rate gives time (duration), this difference, imposed by clock differences, is not immediately obvious, although certainly global time can distinguish distributed occurrences that local time cannot. Discussion begins in terms of service demand (global time), with changes as noted. An application's total service demand decomposes linearly throughout the tree. Demand is 100% (the whole workload) at the root and eventually distributes as smaller weights on leaves, where interpretations apply. Leaf weights sum to 100%. The tree reduces a complex performance surface to a two-dimensional display that shows origins and relationships of performance variations. It does this through paths and fanouts.

Paths=Dependencies. Processing rates may be conditional, since modern machines perform operations with operands from various sources: cache, memory, or, with page-fault, disk. Rates for operations (effects) are thus expressed in the tree as paths. A full and unique effect name is the path to it from the root. Paths in a tree are unique, which disambiguates duplicate appearances of labels on the tree.

Fanouts=Competitions. A second source of performance variation concerns identical results arising in disparate ways. The rate of obtaining a value depends upon which of competing operations actually does the work. Tree fanouts represent different choices by programmer, compiler, or scheduler. Competitions provide a large, important source of performance variance. See Table I [after LYO89b], below.

	Competitors	Architectural Focus	Performance Differences	Speedup Realizations
1	scalar vs. vector	*vector processors*	*peak vector is 4x to 25x faster*	*Monte Carlo trial--none linear algebra--maximum*
2	GATHER-SCATTER vs. unit-stride	*memory system, vector operations*	*unit-stride can be 2.5x faster*	*3x to 7x estimated, if can avoid GATHER-SCATTER's sparse vectors*
3	by-row vs. by-column FORTRAN	*virt. memory subsystem, scalar operations*	*page faults slower by 10^4; source: row refs.*	*by-column 30% faster for linear eq. solver w/FORTRAN*
4	serial vs. parallel execution	*shared-memory parallel processors*	*for N processors, at most N faster*	*gen. unification--none linear algebra--O(N)*
5	messages vs. memory refs.	*distributed-memory nodes*	*memory refs--50x to 10^3x faster generally*	*array processor w/mesh: memory 10x faster*
6	NEWS messages vs. routed mgs.	*distributed-memory nodes*	*NEWS much faster when applicable*	*SIMD 3-D mesh sweep: NEWS 5-9x faster overall*

Table I. Sample Competitions

The finite tree holds decorations from empirical measurements or detailed dynamic modeling. To fit the tree, a continuous parameter must be separated into discrete subranges whose number is determined by resolution of the model or measurements. Suppose there are system resources A and B that at any moment may be concurrently producing the same thing. The unit interval of their mix ratio, A:B, must be made discrete. The resultant tree (see Fig. 1) succinctly displays this source of performance variation.

Shorn of numerical values, the tree somewhat resembles descriptive dependency graphs (a.k.a., "fishbone" or Ishikawa graphs) employed in statistical design-of-experiments; in these, minor (independent) factors successively contribute to more dominant but less differentiated factors. The root is the main, dependent effect. A decorated performance tree is different-- independent variables (the realm of algorithms, etc.) are ignored. The main effect (again, a dependent variable) decomposes into other

dependent effects that must reconstitute the original. Whatever is being observed is conserved at each fanout in the tree.

Conservation of effects allows a tree to support overall quantitative estimates and predictions from changes to system resource parameters. A tabular format might also work, but in general it is not flexible enough for deep or imbalanced structures. Nor is a table easily decorated, since its contiguous structure lacks arcs. The tree, designated *UT* for *use-tree*, displays crucial measurement assumptions in a compact, quickly surveyed format.

Use-Tree with Global Clock. A UT is a doubly-weighted tree-graph. The unadorned tree displays a system's dominant performance effects in a structure that explains performance variances. Nodes and arcs each have weights of capacity (an admittance) and use (a frequency). Arc capacity admittances c_i describe a system's component strengths. Capacities are admittances because these can be obtained from benchmarks without correcting constantly for code size. Arc frequency weights f_i define a relative amount of demand upon architectural features. The weight is a relative fraction of overall program demand, and is *not* any instantaneous overlap or mixing ratio (as per A and B, Figure 1).

UT arc weights are intrinsic to the stage that an arc represents, whereas node weights are cumulative from the tree root. Arcs from a node represent alternatives, e.g., operations on scalars or vectors, operands via inter- or intra-node communications. *Interpretation* assumes that these alternatives never proceed concurrently--the UT must be built accordingly. Let C_w and F_w be capacity and frequency weights of tree node W. Distinguished root node R is such that $C_R=1$ and $F_R=1$: this reflects 100% workload at peak performance. Suppose that a directed arc *wx* from W to X has weights $0<c_{wx}\leq 1$ and $0<f_{wx}\leq 1$, subject to $\Sigma_i f_{wi}=1$. Then

$$C_X = C_W c_{wx}$$

$$F_X = F_W f_{wx}$$

A node with no fanout is a leaf. Each leaf *i* has a frequency weight F_i and a capacity C_i. Leaf weights are used to estimate performance.

Use-Tree with Local Clocks. Without a global sense of time, it is difficult to capture certain collective rates, *e.g.,* level of parallelism at some instant. Events must be recorded separately at each node (clock). If an algorithm is sufficiently homogeneous (across processors), individual node times can be averaged for a compact characterization. Synchronization and communication latencies that were implicit in shared-memory admittances become explicit effects, such as receive-message delay. Interpretation of the tree changes. No longer does it represent a flow of work, but rather, specific time *consumptions*. The lack of an absolute sense of time sometimes can be remedied indirectly, as will be demonstrated later with "time dilation." Furthermore, while the issue of consistency seems trivial, in this case it can be of great experimental help. For instance, when a set of variables is constrained to equal elapsed time (or some other known entity), there is a check on logical completeness of the set and measurement soundness of its associated observations. This is very important when predictive methods

(based upon independent variables) are not available.

Workload, Partitions, Weights. Whenever possible, applications (and the independent variables) are defined logically at a language level, as in FORTRAN, Pascal or Ada®. While identical textual repetitions of language expressions may incur distinct execution costs, it is assumed that any machines to be compared run essentially the same logical programs. A workload of *unordered*, but not necessarily unqualified, operations is partitioned into (disjoint) subsets determined by service classes (the UT identifies these). This is much weaker than stipulating stochastic generations. Since an interpretation is applied to the (static) subsets, partition consistency and level are of paramount importance. With global time, executions within a subset proceed in some undefined manner, serial or parallel, but at *one designated rate*. Any variation collapsed into the subset will be lost in the model. The actual description of an application workload comprises fractions for classes of service, the weights constituting a *signature* whose terms sum to unity. Total (elapsed or consumed) time is then the sum of times of all subsets. Because workload partitionings are usually restricted (*e.g.*, only vectors worry about *length*), a tree paradigm works well.

Higher levels of designated system resource, *e.g.*, subroutines, may through hierarchical dependencies of implementation and system service multiply the needed number of observables. A software architecture has not only its own structural sources of performance variance--it is built upon system pieces that themselves show substantial variation. The "FORTRAN virtual machine" model in [SSM89] requires 102 parameters. Characterizations herein stress machine and system resources that limit the number of parameters, even if this requires special instrumentation hardware. This is also consonant with real-time monitoring, where instrumentation cost and collection time can limit observables to perhaps 10, and in any case certainly less than 100. Alternately, an observable resource can be cast at too low a level, *e.g.*, clock ticks. Since everything uses ticks, 100% of application performance can be ascribed to this featureless "resource"-- this is, in fact, the common problem of having only elapsed time available.

A benchmark program must reproduce its results adequately from run to run. A better-designed benchmark will do this, whereas a poorer example will have considerable variance in its effects. Most examples herein ignore the benchmark as an additional source of performance variance--focus is upon the system's contribution *across* benchmarks. Secondly, a benchmark must be stable. Minor perturbations in parameter settings must not yield completely distinct behavior. Should this not be true, issues of internal representation, timing uncertainties, rounding and truncation may achieve disproportionate roles that confound any intended test results. An experimental communication benchmark set illustrates the probem of stability [LSN89]. Of three tests--Ring, Mesh, Random--the Random test was refractory. It displayed a very narrow range of parameters over which it would run. A simple prediction model was difficult to fit; see below. The primary issue was load imbalance from random assignments of work. Later versions correct this. While codes like Random do appear in real life, a single-trial benchmark must behave better.

Predicting "Random" from Observed Timings					
Confidence Level	50%	70%	80%	90%	95%
Tolerance	*±14%*	*±21%*	*±25%*	*±33*	*±40%*

Skeletons of Real Systems

Figure 2 has five skeletal trees of real computing systems. These trees reflect effects important to each system's performance. For the 205 and Cray-1, treating scalar-vector overlap with only four categories may seem coarse, but in general it is too fine a distinction (as the next paragraph shows). While a skeleton is devoid of weights, it does show if a proposed set of measurements is adequate. For instance, given only peak vector performance, much of the tree is undecorated for the 205, Cray-1, or AP-120B (the vector machines in Figure 2). Such sparsity is acceptable only if an application assigns negligible weights to the unspecified branches. Aside from decoration completeness, the skeletal shape reveals sources of variation. Virtual memory for the CDC 205 is readily apparent on its tree's pure scalar branches. Such details highlight system dissimilarities that dictate programming, performance and measurement distinctions.

An Averaged Model for Vector Pipeline Systems. Individual UT skeletons lose details within a class characterization. Consider a general tree for the architecture class of vector pipeline systems. The sources for this tree are reports [WAN88] and [BAI88], whose conclusions reinforce from different perspectives. The first applies statistics to benchmark results, whereas the second uses typical benchmarking design. Wang, Gary, and Iyer subject data from the 24 Livermore FORTRAN Kernels (LFK or "loops") to rigorous statistical analyses. Their first analysis shows that benchmark values from but one dimension, such as Linpack's peak vector measurements, cannot alone explain performance variance. Cluster analysis then separates the "loops" data into groups distinguished as (i) scalar, (ii) peak vector, or (iii) moderately vectorized. Combining analyses, the important statistical aspects of the LFK are: (1) scalar rate, (2) peak vector rate, (3) rate for intermediate-length vectors, and (4) compiler vectorization capability. These categories cover 98% of variation in LFK (loops) data. A NIST advisory committee [BAI88] had earlier recommended benchmarks for aspects 1-3. Since point 4, degree-of-vectorization, is actually determined by program and compiler, the corresponding UT (Figure 3) subsumes this aspect within its arc weights. Hence, the two approaches--statistical and architectural--dovetail. Note, however, that smaller details of individual systems are lost in the aggregate class description.

Application Examples

Having obtained a UT, it can be put to use. A shared-memory system may have a tree whose admittances can be varied to estimate results of proposed upgrades. This explores system sensitivities *for a chosen application*, the example below being a sort. Prediction limitations arise through redistribution of workload as admittances change. A system with fixed scheduling (*e.g.*, a vector processor) may avoid the immediate problem, but multiprocessors characteristically balance loads dynamically. The uncertainty between best and worst schedulings imposes a prediction tolerance. Even when a multicomputer does not rebalance workload, as with the second example of a simple hypercube, changing system capacities causes problems. For example, message latencies, which are like slack variables, expand or contract. As shown below, an emulation technique called "time dilation" can explore such changes of compute/communicate balance.

I--Altering Tree Admittances

This example involves a 16-processor shared-memory multiprocessor. A parallel Quicksort application illustrates some resource demand consequences of its divide-and-conquer paradigm. Because data, 31,000 values per trial, are in memory, the chosen model is a primitive thing, a bushy tree with 16 branches from its root. The branches correspond to 16 levels of parallelism. Quicksort demand coefficients α_i are typically small except for the 16-processor mode, α_{16} (column 2, below). They have been measured via special low-perturbation methods [CAR88, esp. Figure 4].

Active Processors, **Mode i**	Resource Demand, α_i
	20 trial ave.
01	.026
02	.023
04	.036
08	.057
10	.001
12	.001
14	.002
16	.854

Use Profile--Parallel Quicksort

Imagine the system modified with some (but not many) processors of an improved type. Estimates are made for substitution of one, two, and four of the processors with performances of 3x, 5x or 9x the original. Originally p=16, but with one substitution of 3x, p=16-1+3=18. Other substitutions are treated similarly, so $18 \le p \le 48$. The machine is modeled such that each unit of processor capacity (p) beyond p=1 (monoprocessor) diminishes overall system effectiveness by 1.233%. With p=48, the net available capacity is only 27. A ceiling of 30 units from interconnect limitations is never invoked. This is very simple, but not unrealistic for less expensive parallel systems. The UT's role is to reveal trends quickly, with coarse distinctions. It is not for final engineering determinations. Best and worst performance cases ($\pm x$ in table below) are established via optimistic and pessimistic schedulings. The tree does not reflect methods of scheduling, which must be supplementary calculations. However, detailed scheduling formulations can be tedious, which limits their applicability [LYO89c]. The scheduling here is simple: the best case assumes that the new capacity is always used when it can benefit, and that load balancing is fine-grained and very good. The worst case simply assumes no improvement whatsoever.

Once obtained, scheduling tolerances must be assessed relative to other sources of variation, which are also high. For instance, demand fluctuates from trial to trial. The greatest change would be whenever workload is exchanged between α_{16} and α_1 (max. and min. processing rates). Since $\alpha_1+\alpha_{16} = 0.88$ and rate $R_{original} = 1$, the *relative* change is $(\partial R/\partial\alpha_{16})(\alpha_{16}/R) = (8.4)(0.854/1) = 7.2$. Consequently, even minor demand fluctuations among trials pose large performance uncertainties. Actual fluctuations of α_{16} over 20 trials were +3%, -2% about its mean, or potentially +21.6%, -14.4% if amplified in a worst case. Looking at another source of variance, compiler upgrades can cause a 30% change in performance from one release to another. Given these broad ranges, tolerances in the table are mostly acceptable, if wide. The principal diagonal illustrates three configurations of equal processing power (p=24) but differing concentrations of improvement (the latter being 4@3x, 2@5x and 1@9x, reading down, left-to-right). Configuration 1@9x improves more, but suffers greater uncertainty.

16 Processors	Speed of Faster Processors		
# Fast: # Original	*3x*	*5x*	*9x*
4 fast: 12 slow	1.38 ± 26%	1.63 ± 39%	1.97 ± 49%
2 fast: 14 slow	1.27 ± 21%	1.44 ± 30%	1.70 ± 41%
1 fast: 15 slow	1.19 ± 16%	1.30 ± 23%	1.48 ± 32%

Predicted Relative Improvement, Sort Workload

II--Answering to the Tree

Distinct applications on the same system are displayed with identical UT skeletons but differing weights; application *versus* application comparisons are immediate and

informative. And, since all algorithmic detail is removed, dissimilar applications can have similar signatures. These programs should have similar performance (although independent algorithmic variables will not correspond). If such a pair exhibit performance differences that interpretation of the tree does not predict, the tree lacks detail. Sources of performance variation are missing. On the other hand, a UT may indicate which effects have been collapsed into less revealing performance measurements. This knowledge may inspire further investigations to resolve the finer details. A distributed application with local clocks will illustrate this.

Tree 2-(i) depicts an iPSC-1® hypercube system in which clocks are local, even though our NIST iPSC-1® instrumentation can supply a global time. Local clocks are more realistic for processors separated widely over some computing network: this layout, increasingly more prevalent, raises interesting measurement questions. iPSC-1® computation and communication form a natural dichotomy, since each node has but one processor to handle both duties. Computation is relatively simple (there are no accelerators or virtual memory). Communication (send, receive) is asynchronous or synchronous. The synchronous form has two latency components, *logical delay* and *physical transport.* Program logic delay causes waitings for unsent messages; it indicates algorithmic bottlenecks. Physical transport is the time a message takes traveling from sender node to recipient. Together, logical and transport latencies indicate whether a poorly performing application needs a new algorithm (less logical delay) or faster interconnection hardware (less transport delay).

Local time. A problem for loosely-coupled systems with local clocks is that *observations cannot directly resolve logical and transport contributions.* Yet these are important sources of performance variation. While modeling and simulation might work, accuracies are always questionable; such techniques cannot usually account for deep system details without straining the budget of modest projects. Direct global timings are similarly expensive. However, indirect instrumentation will work. Local clocks can be used to modify node behavior; physical transport *appears* faster through *slowed computation.* This emulation technique, *time dilation* [LYO88], provides a range of relative speeds. All real aspects of the iPSC-1® remain and application programs are unchanged. The latter point is especially significant for sprawling distributed systems. Each node changes its computation rate first by timing each calculation segment between communications; it then introduces an on-the-fly delay that dilates the segment by a multiplicative constant, *D*. Physical transport appears D-times faster as a consequence.

A Ring Benchmark. The test algorithm is a synthetic ring benchmark that models applications with global process dependencies, such as molecular dynamics [LSN89, LYO89a]. It is from the 3-program communications test set mentioned earlier. Computational workload for these iPSC-1® experiments is first partitioned by *communication interrupt time* and *user time* (the latter comprising application computation, operating system services, and residual interrupts). Communication workload is given two components: *receive time* and *send time*, each of which can be synchronous (WRECV, SENDW) or asynchronous (RECV, SEND). The UT for a homogeneous hypercube program appears in Figure 2-(i). (Homogeneous programs have all nodes running the same tasks.) Synchronous SENDWs and asynchronous RECVs are not used. The measured execution times of the components sum to an overall service consumption. This provides a basis for analysis. Receive time (WRECV) is a

measurement from initiation of a receive request until completion of that request. Send time (SEND) covers message dispatch, which may be only the time to get a local buffer. Since clocks are local, logical and physical delays for the synchronous WRECV are indistinguishable--they merge into one observation.

A ring with nearest neighbor connections is allocated with each physical processor node supporting one (logical node) process. Each node (process) will originate a given number of messages, and additionally, process all other messages passing by. Each message makes a complete circuit around the ring while being processed by each node. Once a message returns to its origin, it is removed. A new message is sent unless a node has exhausted all of its quota. When all nodes have processed all messages, the program terminates. The ring parameters have a wide range of settings. Parameters include the number of nodes, message length, number of messages, and computation per datum. Communication in the ring is semi-synchronous, with messages being acknowledged within the program on a one-to-one basis. This flow control keeps messages from exhausting buffer space. All communications (*i.e.*, messages) are between neighboring hypercube nodes. A normalized time, T_D, is defined as the average node service time, T, for a component (*e.g.*, WRECV service time = average WRECV time for n=16 nodes) divided by dilation factor, D, the latter signifying the emulated speedup for physical transport. E is overall elapsed time of the dilated program and E_D is a normalized elapsed time.

$$T_D = \frac{T}{D} \quad \text{and} \quad E_D = \frac{E}{D}$$

Normalization corrects "dilated" measurements back to those for the base machine with a D-faster physical transport. Another useful measure is the speedup of an application for a given D. This is obtained by dividing E_1 (the elapsed time of an undilated program) by E_D .

$$Speedup = \frac{E_1}{E_D} \quad (where\ D > 1)$$

Ring parameter set XA investigates a ring application that is mostly computational. XB provides another extreme: the application is burdened with frequent communications. Table II shows application signatures (*i.e.*, percent of time spent in a given mode) for XA and XB.

APPLICATION	condensed SIGNATURE COMPONENTS			
	User	Comm. Intr.	Wait-Receive	Async. Send
XA: Computation	93.7%	0.5%	5.5%	0.3%
XB: Communication	19.9%	5.1%	73.5%	1.5%

Table II. E_1 Signatures of Two Ring Settings, n=16

As expected for XA, neither its signature nor E_D change significantly under dilation. *Transport delay is not a performance bottleneck with XA.*

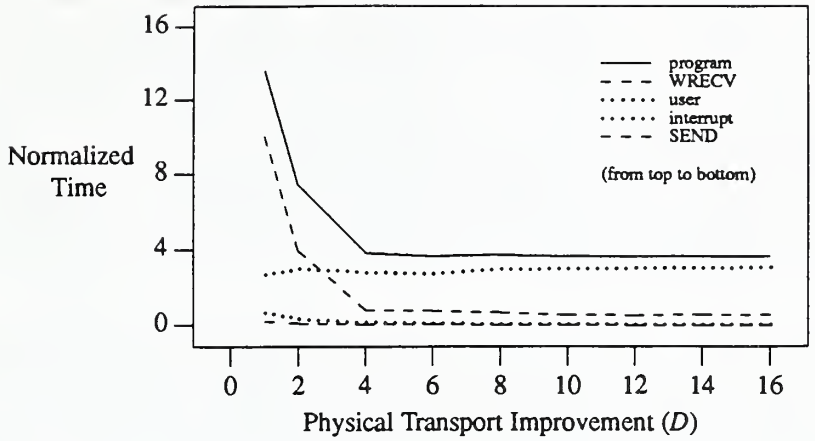


Figure 4. Normalized Time for Program and Factors, Communication-Bound Application.

Figure 4 shows that XB's time for WRECV is significantly reduced as physical transport accelerates. Improvement directly affects the program. With a transport system twice as fast, normalized receive time is reduced from 10.0 seconds to 3.85 seconds. Normalized receive time continues to decrease (down to 0.8 seconds) with a transport system speedup of four, at which point the computation-communication balance of the program has shifted towards the computation end. Communication constitutes but 22.1% (previously 73.5%) of overall service demand. If the modified architecture has a communication system twice as fast originally, XB improves by 1.8. This figure climbs to 3.5 when transport is four times faster. Speedup peaks at 3.7 with a dilation factor of D=6. No further benefits accrue beyond this. *Application XB clearly benefits from enhanced transport capacity*. However, if another communication-bound application, XB2, shows little speedup with dilation, then logical delays are to blame; XB2 needs another algorithm. Time dilation provides a method of investigating the pivotal transport delay with neither application recoding nor globally synchronized clocks often called for by other techniques [REE89a]. Dilation winnows possibilities without expensive architectural or software improvements.

Summary and Prospects

Discussion has explored a new form of concise, coherent performance summary. A need does exist for a simple, statistical viewpoint. For example, users certainly benefit from any characterizations that share common aspects; such characterizations encourage direct comparisons among applications and promote insights on performance. The principal argument in the text has been that stand-alone benchmarks are fixed, specialized forms of workload, which in conjunction with a system, are amenable to static characterizations as passive filters.

The new structure, the use-tree (UT), provides a clear declaration of relationships among a very limited number of major system resources. These explain much of the overall response characteristics of the system to application demands. The set of system resources induces a partition of workload that is shared by all applications that use the system. This is a common thread for benchmark comparisons. Observables (effects) related to the workload partition must, however, be treated in a systematic way. Effects should be separable, that is, free of inconsistent overlap. Any structure imposed by constraining observables provides both a check on experimental soundness and a foundation for studying system sensitivities. And last, limiting concern to dominant resources of a system provides an economy of description.

A use-tree captures critical dependencies and substitutabilities that determine gross performance variations. However, UT effectiveness may hinge upon hardware instrumentation that decouples workload from observation. Otherwise, a compact set of observables may be unavailable. Furthermore, the workload must have stable properties that are expected for experiments. Given the foregoing, an available UT shapes expectations. Accounting for sources of performance variation shown in a UT can motivate new methods of assessment. The "time dilation" technique has illustrated this.

Future Directions. The characterization of distributed, *heterogeneous* computations is an open question. A static (absolute) allocation of tasks to nodes may demand UT structures highly dependent upon the spatial layout of the application. This will naturally impair comparisons with other applications. Fortunately, trends in architecture are away from primitive machines, which are often difficult to program well.

There are plans to calibrate E_D (normalized elapsed time) against direct measurements of logical and transport delays obtained with a central, shared clock. True measurements will give the indirect dilation method a sharper, quantitative interpretation. Another project is to take a larger benchmark set and develop signatures of its members on our shared-memory and loosely-coupled systems.

Acknowledgment. J. Antonishek implemented dilation features on our hypercube, and D. Dimmick provided values for Quicksort resource demands.

Citations

[PERF] Perfect Club Benchmarks, CSRD, University of Illinois at Urbana-Champaign, 104 S. Wright St. Urbana, IL 61801 (217-244-0042). Examples: Structural dynamics, seismic migration. Fourteen codes as of March, 1989.

[SPEC] System Performance Evaluation Cooperative, 65 Washington St. #138, Santa Clara, CA. 95050 (415-792-2901). Currently ten "real-world" applications chosen from among about fifty submissions. Examples: Compilation time of 19 preprocessed source files; a Lisp interpreter run for a fixed problem.

[ANN89] Annaratone, M., Pommerel, C. and Ruehl, R. "Interprocessor communications speed and performance in distributed-memory parallel processors." Proceedings, 16th Annual Int. Sym. on Computer Architecture, *Computer Architecture News* 17, 3(June 1989), 315-324.

[AS90] Antonishek, J. and Snelick, Robert D. "Emulation through time dilation." (to appear in) Proceedings, Fifth Distributed Memory Computing Conference, DMCC5, Charleston, S.C., April, 1990, 8 pp.

[BAI88] Bailey, D., E. Brooks, J. Dongarra, A. Hayes, M. Heath, and G. Lyon. "Benchmarks to supplant export

[BHH78] Box, G.E.P., Hunter, W.G. and J.S. Hunter. **Statistics for Experimenters.** John Wiley & Sons, Inc., New York, 1978.

[CAR88] Carpenter, R.J., "Performance measurement instrumentation for multiprocessor computers." (in) *High Performance Computer Systems*, E. Gelenbe (ed.), North-Holland, 1988, 81-92.

[D'H88] D'Hollander, E.H. "Performance modelling of supercomputer architectures and algorithms." (in) **Scientific Computing on Supercomputers**, J.T. Devreese and P.E. Van Camp (eds.), Plenum Press 1988, 3-28.

[DEN88] Denning, P.J. and G.B. Adams III. "Research questions for performance analysis of a supercomputer." (in) **Performance Evaluation of Supercomputers**, J.L. Martin (ed.), Elsevier Science Publishers B.V. 1988, 403-419.

[GRE72] Grenander, U. and R.F. Tsao. "Quantitative methods for evaluating computer system performance: A review and proposals," (in) **Statistical Computer Performance Evaluation**, W. Freiberger (ed.), Academic Press 1972, 3-24.

[HAR83] Haring, G. "On stochastic models of interactive workloads." (in) **Performance '83**, A.K. Agrawala and S.K. Tripathi (eds.), North Holland 1983, 133-152.

[HOC84] Hockney, R.W., and C.R. Jesshope. **Parallel Computers.** Bristol, England: Adam Hilger Ltd., 1984.

[LSN89] Lyon, G. and Snelick, R. Architecturally-Focused Benchmarks with a Communication Example. NISTIR 89-4053, March, 1989, 38pp.

[LYO88] Lyon, G. Notes within NIST: "A simple emulator for variable hardware balance on a hypercube," May, 1988; "Time dilation (logical vs. physical delay)," September, 1989.

[LYO89a] Lyon, G.E. "Design factors for parallel processing benchmarks," *Theoretical Computer Science* 64, (1989), 175-189.

[LYO89b] Lyon, G.E. "Hybrid structures for simple computer performance estimates." NISTIR 89-4063, March, 1989, 24pp. *Also in abridged text as* "Capacity-and-use trees for estimating computer performance variations" at the International Conf. on Computing and Information, ICCI'89, May, 1989, Toronto.

[LYO89c] Lyon, G.E. "Processing rate sensitivities of a heterogeneous multiprocessor." NISTIR 89-4128, August, 1989, 11pp.

[POT86] Potier, D. "Modelling techniques and tools," (in) **Modelling Techiques and Tools for Performance Analysis '85**, N. Abu El Ata (ed.), El;sevier Science Publishers B.V., 1986, 3-4.

[REE89a] Reed, D.A. "Distributed memory working group summary." (in) *Instrumentation for Future Parallel Computing Systems*, M. Simmons, *et.al.*, (eds.), ACM Press, 1989, 239-250.

[REE89b] Reed, D.A. "Performance analysis of parallel systems: A look at alternatives." notes from the Third ISR Supercomputing Workshop, August 30-September 1, 1989, Oahu, Hawaii.

[SER85] Serazzi, G. "Workload modeling techniques." (in) **Modeling Techniques and Tools for Performance Anaysis '85**, N. Abu El Ata (ed.), Elsevier Science Publishers B.V., 1986, 13-27.

[SSM89] Saavedra-Barrera, R.H., A.J. Smith, and E. Miya. "Machine characterization based on an abstract high-level language machine." *IEEE Trans. on Computers 38*, 12(December 1989), 1659-1679.

[UNI89] Uniejewski, J. "Characterizing RISC systems using application benchmarks." *Computer Design RISC Supplement*,(13 November 1989), 45-48.

[WAN88] Wang, J.C., J.M. Gary, and H.K. Iyer. "On the analysis of computer performance data," Draft report, NIST, Dec. 1988, 33pp., and revision, "A technique to evaluate benchmarks: A case study using the Livermore Loops," Jan. 1990, 26pp.

[vanW87] van Waveren, G.M. "Application of sparse vector techniques on a molecular dynamics program." (in) **Algorithms and Applications on Vector and Parallel Computers**, H.J.J. te Riele, T.J. Dekker, and H.A. van der Vorst (eds.), Elsevier Science Publishers B.V. 1987, 405-428.

[WYB88] Wybranietz, D. and D. Haban. "Monitoring and performance measuring distributed systems during operation." Proc., 1987 ACM SIGMETRICS Conf. on Meas. and Modeling of Computer Systems, (in) *Performance Evaluation Review,* Special issue V. 15, 1(May 1987), 197-206.

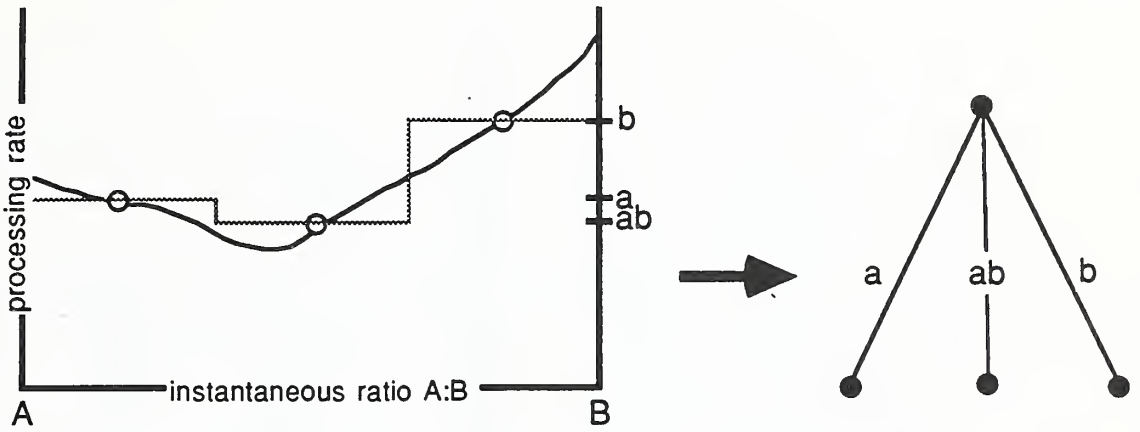
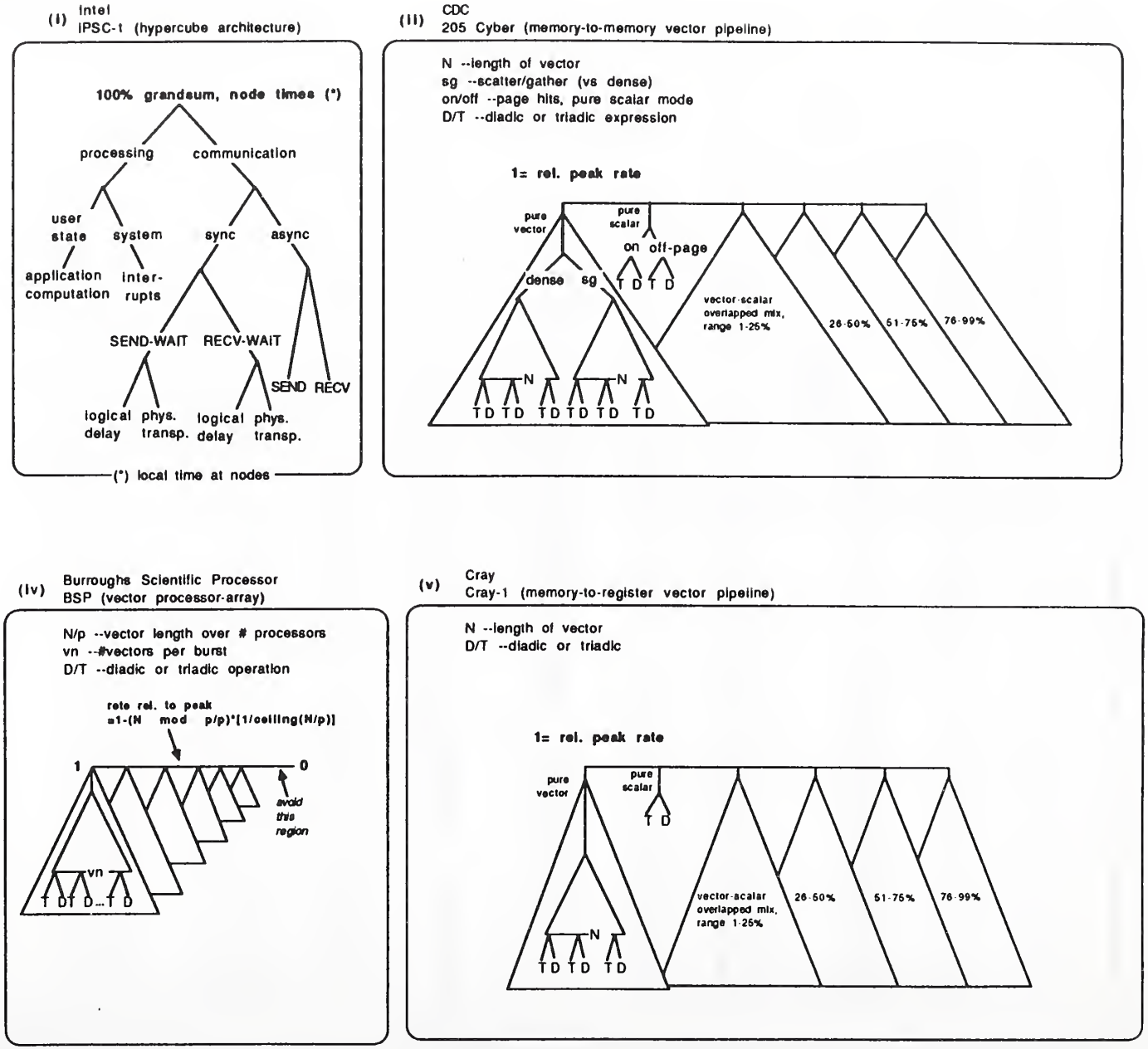
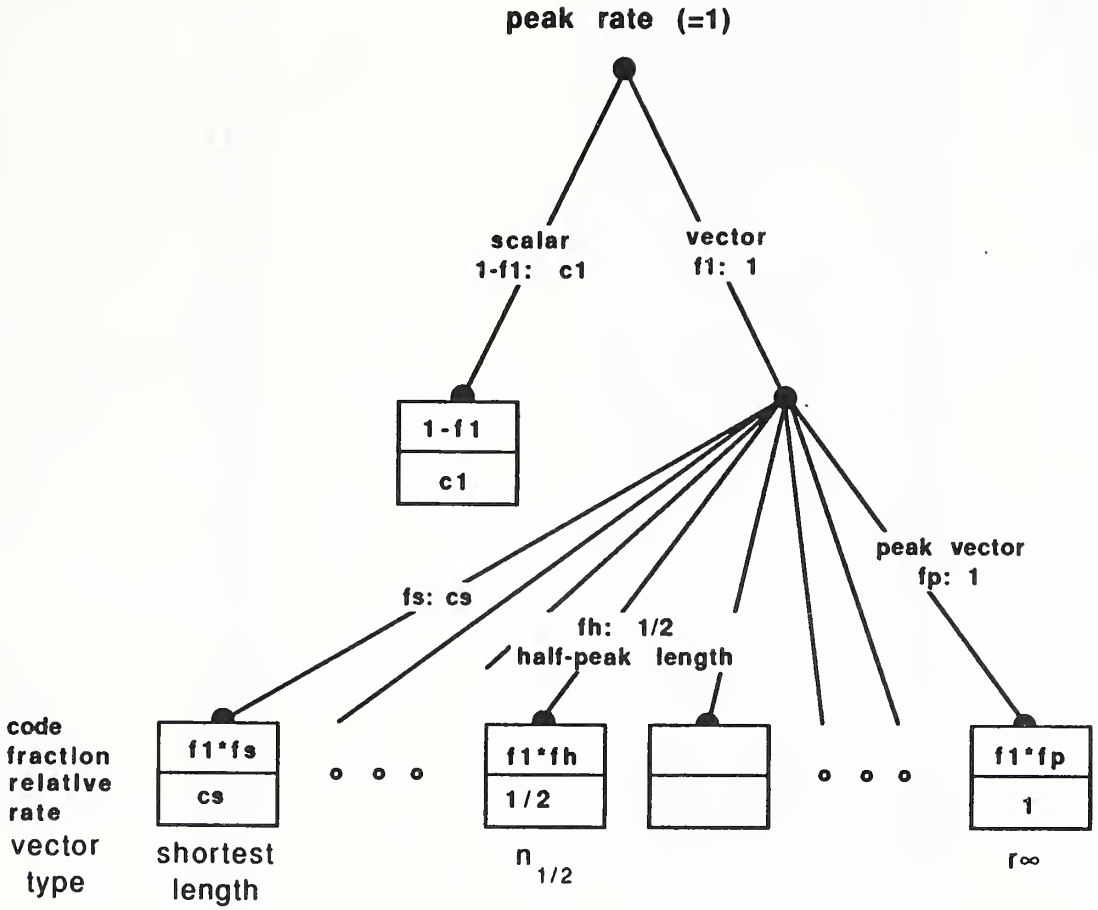


Figure 1. Conversion to Discrete Tree

Figure 2. Use-Trees for Several Architectures





Notes:
1) format of weights is fraction: capacity.

2) capacities are from benchmark measurements

3) each application code has its own signature of frequencies

4) "Degree of vectorization" shows in a code's signature

Figure 3. UT Diagram for Vector Pipeline System

NIST-114A
(REV. 3-89)

U.S. DEPARTMENT OF COMMERCE
NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY

BIBLIOGRAPHIC DATA SHEET

1. PUBLICATION OR REPORT NUMBER
NISTIR 90-4275

2. PERFORMING ORGANIZATION REPORT NUMBER

3. PUBLICATION DATE
APRIL 1990

4. TITLE AND SUBTITLE

Workloads, Observables, Benchmarks and Instrumentation

5. AUTHOR(S)

G. E. Lyon and R. D. Snelick

6. PERFORMING ORGANIZATION (IF JOINT OR OTHER THAN NIST, SEE INSTRUCTIONS)

U.S. DEPARTMENT OF COMMERCE
NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY
GAITHERSBURG, MD 20899

7. CONTRACT/GRANT NUMBER

8. TYPE OF REPORT AND PERIOD COVERED

9. SPONSORING ORGANIZATION NAME AND COMPLETE ADDRESS (STREET, CITY, STATE, ZIP)

Defense Advanced Research Projects Agency, Arlington, VA 22209

Department of Energy, Washington, DC 20545

10. SUPPLEMENTARY NOTES

[ ] DOCUMENT DESCRIBES A COMPUTER PROGRAM; SF-185, FIPS SOFTWARE SUMMARY, IS ATTACHED.

11. ABSTRACT (A 200-WORD OR LESS FACTUAL SUMMARY OF MOST SIGNIFICANT INFORMATION. IF DOCUMENT INCLUDES A SIGNIFICANT BIBLIOGRAPHY OR LITERATURE SURVEY, MENTION IT HERE.)

Standalone benchmarks are fixed, specialized forms of workload amenable to static characterizations. A tree provides a clear declaration of the relationships among a very limited number of major system resources that explain most of the benchmark performance. This approach is especially effective with hardware instrumentation that decouples workload from observation, for otherwise a compact set of observables may be unavailable. The tree supports simple predictions and promotes more meaningful comparisons. Accounting for the sources of performance variation shown in the tree can motivate new methods of assessment; a "time dilation" technique illustrates this for loosely-coupled systems with local clocks.

12. KEY WORDS (6 TO 12 ENTRIES; ALPHABETICAL ORDER; CAPITALIZE ONLY PROPER NAMES; AND SEPARATE KEY WORDS BY SEMICOLONS)

application; architecture; benchmarks; components; models; performance

13. AVAILABILITY

[X] UNLIMITED

[ ] FOR OFFICIAL DISTRIBUTION. DO NOT RELEASE TO NATIONAL TECHNICAL INFORMATION SERVICE (NTIS).

[ ] ORDER FROM SUPERINTENDENT OF DOCUMENTS, U.S. GOVERNMENT PRINTING OFFICE, WASHINGTON, DC 20402.

[X] ORDER FROM NATIONAL TECHNICAL INFORMATION SERVICE (NTIS), SPRINGFIELD, VA 22161.

14. NUMBER OF PRINTED PAGES
23

15. PRICE
A02

ELECTRONIC FORM