Tracing Application Program Execution on the Cray X-MP and Cray 2

Allen D. Malony* John L. Larson† Daniel A. Reed‡

Center for Supercomputer Research and Development
University of Illinois
Urbana, Illinois 61801

Abstract

Important insights into program operation can be gained by observing dynamic execution behavior. Unfortunately, many high-performance machines provide execution profile summaries as the only tool for performance investigation. We have developed a tracing library for the Cray X-MP and Cray 2 supercomputers that supports the low-overhead capture of execution events for sequential and multitasked programs. This library has been extended to use the automatic instrumentation facilities on these machines, allowing trace data from routine entry and exit, and other program segments, to be captured. To assess the utility of the trace-based tools, three of the Perfect Benchmark codes have been tested in scalar and vector modes with the tracing instrumentation. In addition to computing summary execution statistics from the traces, interesting execution dynamics appear when studying the trace histories. It is also possible to compare codes across the two architectures by correlating the event traces. Our conclusion is that adding tracing support in Cray supercomputers can have significant returns in improved performance characterization and evaluation.

1 Introduction

Typically, the performance of an application can vary greatly during execution. For supercomputers, this situation is even more acute as, in general, the performance range is greater and performance variations among program segments can be more pronounced. The complexity in the performance space surrounding the use and interactions of advanced architectural and software features of supercomputers often implies that minor alterations in execution behavior are manifest as large changes in achieved performance. Simply put, application performance, especially for supercomputers, can be highly variable and depends significantly on the dynamic interaction of the code with the high-performance features of the machine.

The problem facing the performance analyst is how to characterize an application's operation both in terms of its overall performance and its dynamic execution behavior — what code is executed when and how the machine resources are used. Profiling tools typically report application code performance as summaries of execution time across program code blocks [4]. Whereas summary performance statistics directly identify code segments that consume large fractions of total execution time, they do not provide insight into how the application executes over time nor its dynamic use of machine resources.

In part, the goal of this paper is to present techniques for capturing and analyzing dynamic program execution and to show that performance measurements of execution behavior can provide greater insight than summary statistics. But this conclusion is not unexpected — other research efforts have reported similar findings [13, 14]. There has been a reluctance, however, to capture dynamic execution state for fear that the measurement system will corrupt the execution behavior being observed, particularly in the case of high-performance systems. In many cases, this fear is largely unfounded and simple perturbation models can be applied to recover true execution performance [10]. To the extent that it does occur, the increased insight into applications operation offered by the trace data must be weighed against the perturbations, and therefore performance inaccuracies, in observed execution behavior.

In this paper, we describe both a trace-based measurement system implemented for Cray supercomput-

*Supported in part by the National Science Foundation under Grants No. NSF MIP-88-07775 and No. NSF ASC-84-04556, and the NASA Ames Research Center Grant No. NCC-2-559.

†Supported in part by the National Science Foundation under grant NSF ASC-84-04556. Author's current address is CSRD.

‡Supported in part by the National Science Foundation under grants NSF CCR-86-57696, NSF CCR-87-06653 and NSF CDA-87-22836 and by the National Aeronautics and Space Administration under NASA Contract Number NAG-1-613.

CH2916-5/90/0000/0060/$01.00 © IEEE

ers and its use in characterizing the performance dynamics of full application codes. The Cray supercomputers provide a practical environment to test the merits of trace-based performance characterization. The Cray compilers support automatic instrumentation at the routine level to capture entry and exit events [2, 8]. Moreover, the existence of a fast-access, high-resolution system clock allows fine-grained timing measurements. Furthermore, the Cray X-MP includes a hardware performance monitor for run-time capture of several hardware metrics.

The remainder of the paper is organized as follows. In §2, we briefly review the standard profiling tools available on the Cray systems. In §3, we describe the tracing system developed for the Cray X-MP and Cray 2. We introduce the applications codes from the Perfect Benchmark set used for testing purposes in §4. In §5, we compare the profiling results from the standard tools to those calculated from the application code traces. We use the comparison of these results as a measure of the reliability of the trace data. In §6, we analyze the execution dynamics of the Perfect codes, primarily FLO52, on the Cray X-MP and Cray 2. Finally, we present some conclusions and suggest directions for future research.

2 Standard Cray Tools

Two profiling tools are commonly used on Cray systems: *Flowtrace* and *Perftrace*. The *Flowtrace* tool [2] is available on the Cray X-MP, Cray Y-MP, and Cray 2. It purpose is to measure where time is spent in a program's execution and to generate a time profile based on program routines. Unlike sample-based, interrupt-driven profilers [5], *Flowtrace* calculates the profile dynamically by inserting profiling code at the beginning and end of each routine. This instrumentation is provided automatically by the Cray compilers. At the end of the program's execution, the time profile is formatted and written to a file. The profile includes the number of calls to the routine, the time spent in each routine, and the average time per routine call.

The *Perftrace* tool [8] is available only on Cray X-MP and Cray Y-MP systems. *Perftrace* computes all the *Flowtrace* statistics and generates a profile of hardware performance. Similar to the time profile measurement, *Perftrace* samples the counters of the hardware performance monitor (HPM) at routine entry and exit to determine the distribution of hardware performance across the program routines. Because the HPM allows only one of four hardware counter groups (each group contains eight counters) to be monitored at a time, *Perftrace* reports only the statistics for the selected counter group. Multiple runs (up to four) must be made if hardware profiles spanning counter groups are desired.

In addition to the time profile statistics, *Perftrace* reports the counter value of each hardware metric accumulated for each routine (e.g., floating point operations or memory references), this value shown in millions per second (e.g., millions of floating point operations per seconds), the percentage of the hardware metric total the routine counter values represent, and different derived statistics depending on the counter group (e.g., memory references per floating point operation).

Both *Flowtrace* and *Perftrace* produce summary performance statistics — the statistics reflect performance totals accumulated for the entire program execution. Summary statistics reflect dynamic execution, but only in an averaged form.

3 Tracing Environment

We have developed a tracing system that can capture the history of a program's performance behavior. We were able to use the existing Cray compiler support for *Flowtrace* and *Perftrace* to provide automatic instrumentation for tracing measurement. The tracing environment included a low-level library for trace recording plus libraries that replaced the standard *Flowtrace* and *Perftrace* routines.

3.1 Software Event Tracing

The basic function of a software event tracing facility is to record the occurrence of an event by writing a timestamped event identifier, with optional data, to a trace buffer. For concurrent event tracing, a buffering scheme is needed that allows multiple tasks concurrent access without conflicts. In our tracing facility, we allocate trace buffers statically, one for each possible task.

The routines in our Cray tracing library are described in Table 1. The event identifier and optional data are supplied by the user, and the high-resolution system clock is retrieved from the Cray library routine *irtc()*.[1] Because the high-resolution system clock is a register shared by every processor in the machine, all tasks see a common, global time value. The side-effect of using the high-resolution cycle counter is that it is real-time and, therefore, timing measurements are susceptible to multiprogramming influences. Thus, our tracing library can generate accurate timing measurements only in dedicated mode.

The routine *texit()* is called at the end of program execution to save the trace data in trace files. If, during execution, one or more trace buffers overflow, the trace data for the offending tasks are written out to the associated trace file before task execution continues.

Although the tracing routines support the concurrent capture of trace events from multiple tasks, all the re-

[1] In Cray's Fortran compiler,CFT77, *irtc()* is compiled as a single machine instruction.

Routine	Description
tevent(*tid*, *eid*)	writes a timestamp and the event *eid* to the trace buffer of task *tid*
tdatas(*tid*, *eid*, *data*)	writes a timestamp, the event *eid*, and the integer data item *data* to the trace buffer of task *tid*
tdata(*tid*, *eid*, *n*, *ptr*)	writes a timestamp, the event *eid*, and the first *n* integer data items pointed to by *ptr* to the trace buffer of the task *tid*; *tdatas()* is faster for a single data item
texit()	writes the data currently in the trace buffers for each task to files task.*i* where *i* is the task id

Table 1: Cray Tracing Library Routines

sults reported in the paper are from scalar and vector executions. Analysis of concurrent traces is complicated by the inability to efficiently determine the processing resource where the event was recorded. We currently are pursuing solutions to this problem.

In general, tracing routines should be implemented to minimize execution time overhead. The current implementation of the tracing routines is in Fortran to enhance portability across the Cray family.[2] The execution time overheads (mean, variance, and standard deviation) for the tracing routines on the Cray X-MP and Cray 2 are shown in Table 2. The *tdata()* numbers are for ten data items saved with the event identifier. In all cases, the subroutine invocation times constitute a significant portion of the total. The execution time overheads of the tracing routines are used during trace analysis to remove the intrusion due to instrumentation; see §5.

3.2 TraceFlow and TracePerf

We developed code that captures routine entry and exit events using the tracing library to record the trace data. The goal was to extend the existing, automatic *Flowtrace* and *Perftrace* instrumentation provided by the Cray compilers [2, 8]. Our approach replaced the *Flowtrace* and *Perftrace* routines with versions that generate traces.

We refer to the traced *Flowtrace* routines as the *TraceFlow* library. The traced *Perftrace* routines we refer to as the *TracePerf* library. The routines in each library are described in Table 3. The principal difference between the *TracePerf* and *TraceFlow* libraries is that the *TracePerf* routines record current HPM counter values.[3] Note that for both *TraceFlow* and *TracePerf*, the routine name is saved in the trace only for "entry" events. We assume that entry/exit pairs can be determined from the trace during analysis. Because a frequently called routine can cause the trace to grow large, the *flowoff()* and *flowon()* routines have been implemented to control when trace data should be saved.

Routine	TraceFlow	
	Minimum	*Average*
flowentr	720	733.7
flowexit	476	491.96
flowin	514	535.12
flowout	480	495.04

Table 5: Cray 2 TraceFlow Execution Time Overheads

There is a significant practical difference between the overheads for the *TraceFlow* and *TracePerf* routines. Table 4 gives the mean execution time overheads for calling the *TraceFlow* and *TracePerf* routines for the Cray X-MP.[4] Table 5 shows the mean execution time overheads for *TraceFlow* on the Cray 2.[5] Although *TracePerf* records approximately five times the amount of data as *TraceFlow*, the execution time difference is largely because *TracePerf* routines require an I/O operation to retrieve HPM counter values.

In addition to the execution time overheads for *TracePerf*, there are also perturbations in the HPM counters; calling the *TracePerf* routines will affect the HPM counters. To accurately recover the HPM counts during trace analysis, we measured the counter "overheads" for the *TracePerf* routines for each of the counter groups. These overheads for group zero are reported in Table 6. The overheads for the other groups can be found in [9].

[2] In fact, there are no differences in the tracing library for the Cray X-MP and the Cray 2, and the library should port without change to the Cray Y-MP system.

[3] We used the hardware performance monitor of the CRAY X-MP/48 running UNICOS at the National Center for Supercomputing Applications in Urbana, Illinois. The HPM is a standard feature of all CRAY X-MP's and CRAY Y-MP's. For further information on the HPM, see [7, 8].

[4] As in *Perftrace*, only one group of eight counters out of 32 are accessible during any execution. Thus, execution time overheads are listed for each of the four groups.

[5] Because no hardware performance monitor exists on the Cray 2, no *TracePerf* library can be constructed.

Routine	Cray X-MP (cp = 8.5 ns)			Cray 2 (cp = 4.1 ns)		
	Mean	*Variance*	*Standard Deviation*	*Mean*	*Variance*	*Standard Deviation*
tevent()						
clock periods	142	33	6	335	728	27
nanoseconds	1207	281	51	1374	2985	111
tdatas()						
clock periods	155	64	8	351	923	30
nanoseconds	1318	544	68	1439	3784	123
tdata()						
clock periods	1102	428	20	2662	12427	111
nanoseconds	9367	3638	170	10914	50951	455

Table 2: Cray X-MP and 2 Tracing Execution Time Overheads

Routine	Description	
	TraceFlow	**TracePerf**
flowentr()	records the routine entry event in the trace for the calling task; the routine identifier is saved	records the routine entry event in the trace for the calling task; the routine identifier and HPM counters are saved
flowexit()	records the routine exit event in the trace for the calling task	records the routine exit event in the trace for the calling task; the HPM counters are saved
flowin(*blockid*)	records the block entry event in the trace for the calling task; the block identifier *blockid* is saved	records the block entry event in the trace for the calling task; the block identifier *blockid* and HPM counters are saved
flowout()	records the block exit event in the trace for the calling task	records the block exit event in the trace for the calling task; the HPM counters are saved
perfon()	not defined	same as flowin()
perfoff()	not defined	same as flowout()
flowstop()	writes the trace files	write the trace files
flowoff()	disable tracing	disable tracing
flowon()	enable tracing	enable tracing

Table 3: Cray TraceFlow and TracePerf Routines

Routine	TraceFlow	TracePerf			
		Group 0	*Group 1*	*Group 2*	*Group 3*
flowentr	307	1493	1492	1492	1492
flowexit	204	1481	1481	1481	1481
flowin	230	1495	1495	1497	1496
flowout	204	1409	1409	1409	1409
perfenable	—	1534	1535	1534	1534
perfdisable	—	1460	1460	1460	1460

Table 4: Cray X-MP TraceFlow and TracePerf Execution Time Overheads

Routine	Counter							
	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*
flowentr	391	590	13	0	153	0	0	0
flowexit	385	572	14	0	149	0	0	0
flowin	397	589	13	0	158	0	0	0
flowout	378	543	12	0	146	0	0	0
perfenable	401	609	14	0	159	0	0	0
perfdisable	385	569	13	0	149	0	0	0

Table 6: Cray X-MP TracePerf HPM Counter Overheads - Group 0

4 Perfect Benchmarks

The applications codes used in our study (FLO52, OCEAN, and DYFESM) are benchmark programs from the Perfect Benchmark suite [1]. The baseline versions of the codes were compiled with default CFT77 compiler options and executed on one processor. As such, our results differ slightly from those presented in [12]. In particular, no preprocessors were used, all codes were run on a single processor, and no hand optimizations were performed.

The benchmark codes were chosen from different application areas. FLO52 and OCEAN are fluid dynamics programs. FLO52 is a two-dimensional analysis of the transonic inviscid flow past an airfoil and solves the unsteady Euler equations [6]. OCEAN solves the dynamical equations of a two-dimensional Boussinesq fluid layer to study the chaotic behavior of free-slip Rayleigh-Benard convection [3]. DYFESM is a two-dimensional finite element program for the analysis of symmetric anisotropic structures [11]. An explicit leap-frog temporal method with substructuring is used to solve for the displacements and stresses, along with the velocities and accelerations at each time step.

5 Trace Verification

The primary motivation behind tracing application program execution is to capture dynamic performance behavior. The *TraceFlow* library can be used to understand the timing properties of routine execution. The *TracePerf* library can be used to obtain additional data on machine performance. However, with the use of these libraries comes perturbations in the program's behavior, both in execution time and in the HPM counter data. The severity of these perturbations and our ability to remove them will determine the reliability of the trace information.

We have shown that perturbations due to trace-based monitoring can, in many cases, be modeled and, subsequently, removed during trace analysis [10]. We applied the perturbation analysis techniques from our previous work during the development of our trace analysis programs. In particular, we developed trace analysis programs that compute the same set of profile statistics generated by *Flowtrace* and *Perftrace*. We used the results obtained from these programs as a measure of the reliability of the trace data after perturbation analysis has been applied.

5.1 Flowtrace Profiling versus TraceFlow Profiling

The *Flowtrace* statistics summarize the distribution of execution time across an application's routines. The total number of calls to a routine and the total execution time within a routine are reported. Information is also provided about the execution call graph. Our program to analyze an application trace produces similar statistics, plus additional execution time profiles, for the parents and children of each routine.

Table 7 shows the execution time in seconds for some of the routines in FLO52 when executed in scalar mode, as calculated by *Flowtrace* and from a *TraceFlow* trace. As shown, the trace-generated statistics are very close to those produced by *Flowtrace*. The statistics from the vector execution of FLO52, plus those from the scalar and vector execution of the other two Perfect codes, also match extremely well.[6]

Two general conclusions can be drawn from this analysis. First, we can accurately measure gross execution performance characteristics, in the form of execution time profiles, from traces of routine entry and exit by including perturbations introduced by the trace instrumentation. Second, we believe the dynamic performance behavior, as represented by the full event trace, should also be credible. Although this second conclusion is difficult to prove in practice, other analysis [10] supports this conclusion.

5.2 Perftrace Profiling versus TracePerf Profiling

An analysis similar to that used with *Flowtrace* and *TraceFlow* can be applied to *Perftrace* and *TracePerf*. The results from the vector execution of FLO52 are given in Table 8. In addition to the execution time measurements, some of the operation counts from counter data in group zero are shown; Madds represents millions of floating point additions, Mmult is millions of floating point multiplies, Mrecip is millions of floating point reciprocals, and Mflop is millions of total floating point operations.

The statistics generated from the *TracePerf* trace are very close to those reported by *Perftrace*. The importance of the *TracePerf* data lies in its ability to associate

[6] These statistics can be found in [9].

Routine	Cray X-MP				Cray 2			
	Flowtrace		TraceFlow		Flowtrace		TraceFlow	
	Time	Percent	Time	Percent	Time	Percent	Time	Percent
ADDX	1.002	1.94	1.010	1.94	1.175	1.98	1.177	2.00
BCFAR	0.311	0.60	0.315	0.60	0.295	0.50	0.301	0.51
BCWALL	0.918	1.78	0.925	1.77	0.815	1.38	0.809	1.37
COLLC	0.611	1.18	0.615	1.18	0.702	1.19	0.694	1.18
CPLOT	0.128	0.25	0.129	0.25	0.297	0.50	0.282	0.49
DFLUX	10.898	21.13	10.986	21.05	11.684	19.74	11.67	19.84
DFLUXC	1.566	3.04	1.584	3.04	1.573	2.66	1.578	2.68
EFLUX	15.125	29.33	15.171	29.07	17.218	29.09	17.11	29.08
EULER	7.726	14.98	7.800	14.95	8.603	14.54	8.534	14.51
MESH	0.100	0.19	0.100	0.19	0.108	0.18	0.108	0.18
PRNTFF	0.113	0.22	0.119	0.23	0.223	0.38	0.266	0.45
PSMOO	10.370	20.11	10.627	20.36	14.583	24.64	14.314	24.33
STEP	2.593	5.03	2.617	5.01	1.749	2.96	1.756	2.98
TOTAL	51.570	99.78	52.184	99.64	59.186	99.74	58.826	99.60

Table 7: Flowtrace and TraceFlow Statistics for Scalar FLO52

Routine	Measurement	Time	Percent	Madds	Mmult.	Mrecip.	Mflop.
EFLUX	Flowtrace	1.76	26.40	101.0	106.0	10.60	218.0
	TraceFlow	1.79	26.90	101.0	106.2	10.59	217.8
PSMOO	Flowtrace	1.29	19.38	69.5	35.4	0.02	105.0
	TraceFlow	1.33	20.03	69.5	35.4	0.03	104.9
DFLUX	Flowtrace	1.23	18.43	70.3	54.8	6.07	131.0
	TraceFlow	1.24	18.61	70.3	54.8	6.07	131.1
EULER	Flowtrace	1.07	16.08	35.4	46.3	6.59	88.3
	TraceFlow	1.00	15.06	35.4	46.2	6.59	88.2
STEP	Flowtrace	0.25	3.77	12.8	19.0	4.42	36.3
	TraceFlow	0.26	3.85	12.8	19.0	4.42	36.3
DFLUXC	Flowtrace	0.21	3.07	9.42	8.98	1.29	19.7
	TraceFlow	0.21	3.22	9.42	8.99	1.29	19.8
ADDX	Flowtrace	0.15	2.26	8.92	8.81	0.43	18.2
	TraceFlow	0.15	2.28	8.92	8.81	0.43	18.2
CPLOT	Flowtrace	0.13	1.92	0.00	0.07	0.00	0.07
	TraceFlow	0.13	1.92	0.00	0.07	0.00	0.07
PRNTFF	Flowtrace	0.11	1.70	0.04	0.13	0.00	0.17
	TraceFlow	0.11	1.70	0.04	0.13	0.00	0.17
MESH	Flowtrace	0.10	1.50	0.59	0.69	0.08	1.36
	TraceFlow	0.10	1.51	0.59	0.69	0.08	1.36

Table 8: Perftrace and TracePerf Statistics for Vector FLO52 - Cray X-MP, Group 0

dynamic machine operation with the sequence of application routine executions. The high reliability of this data implies that a finer degree of performance characterization can be obtained than with summary results of hardware performance.

5.3 Caveats

It should be mentioned that *Flowtrace*, *Perftrace*, *TraceFlow*, and *TracePerf* measures can be in error for routines with small execution times. This is mainly the result of instrumentation overhead and not timer resolution. Although the analysis programs attempt to remove the overhead, a few percent deviation in the overhead relative to the routine execution time could be significant.[7] From the viewpoint of performance analysis, however, achieving high measurement accuracy for routines representing a small fraction of total program execution time is not of primary importance.

6 Dynamic Execution Analysis

As noted earlier, detailed event traces hold promise for characterizing dynamic execution performance if perturbation effects can be understood and compensated. Earlier, we used the accuracy of execution profile statistics, calculated from an event trace, as a measure of the reliability of the trace data and the validity of the perturbation analysis techniques. Below, we demonstrate some of the dynamic execution behavior analyses possible by applying the *TraceFlow* and *TracePerf* tools. Although we show results for the three Perfect codes, for brevity's sake, we concentrate primarily on the FLO52 code.

6.1 Subroutine Events

Using *TraceFlow* trace data, one can can analyze the dynamic characteristics of subroutine calls, beginning with a simple *procedure event graph* that shows the subroutine and function transitions during application program execution. In such graphs, the axes represent, respectively, the currently executing application procedure and the procedure execution time.

When the dynamic pattern of calls is correlated with application source code locations, these procedure event graphs can show the spatial and temporal patterns of control flow. More abstractly, the procedure event graphs show application execution stages (e.g., the functional combination of multiple application algorithms or the processing of multiple input data sets).

[7]The Cray tools do not report routine timings if the average time per call is less than 0.001 seconds because the results are considered untrustworthy.

Routine	Event	Routine	Event
FLO52Q	1	STEP	9
COORD	2	EULER	10
GEOM	3	COLLC	11
MESH	4	ADDX	12
GRID	5	PRNTFF	13
XPAND	6	CPLOT	14
METRIC	7	GRAPH	15
INIT	8	RPLOT	16

Table 9: FLO52 Routine Names to Event Numbers

Generally, the dynamic pattern of procedure calls is identical for both scalar and vector execution.[8] Clearly, vectorization compresses the event time scale and may change the relative execution times of individual procedure invocations. These are seen as changes in the "shape" of the procedure event graph.

Figures 1 and 2 show the procedure event graphs for FLO52 executions on the Cray X-MP in both scalar and vector modes, respectively. To provide sufficient resolution to display all relevant calls, the procedure event graphs are drawn in eight sections. In each section, time increases from top to bottom and continues at the top of the next section. In the figures, each procedure invocation is marked by a horizontal line (i.e., a state transition). Vertical lines denote the amount of time spent in the currently executing procedure. To present an entire execution history, while avoiding event clutter, some of the more frequent procedure events have been elided. Finally, Table 9 shows the association of procedure names and event numbers used in Figures 1 and 2.

From Figures 1 and 2, one can identify five major execution phases:

- Initialization
- Grid One
- Grid Two
- Grid Three
- Termination

In the three grid phases, event analysis reveals a periodic sequence of procedure calls, where each phase contains forty-eight periods of the procedure invocation sequence. Obviously, the application phases, the repetition of procedure event sequences, and the procedure sequence itself are manifestations of the application program's call graph and the input data. Each grid phase reflects a dynamic instantiation of the static procedure call graph for a particular grid. Here, each

[8]Inline function substitution to increase vectorization is one possible exception.

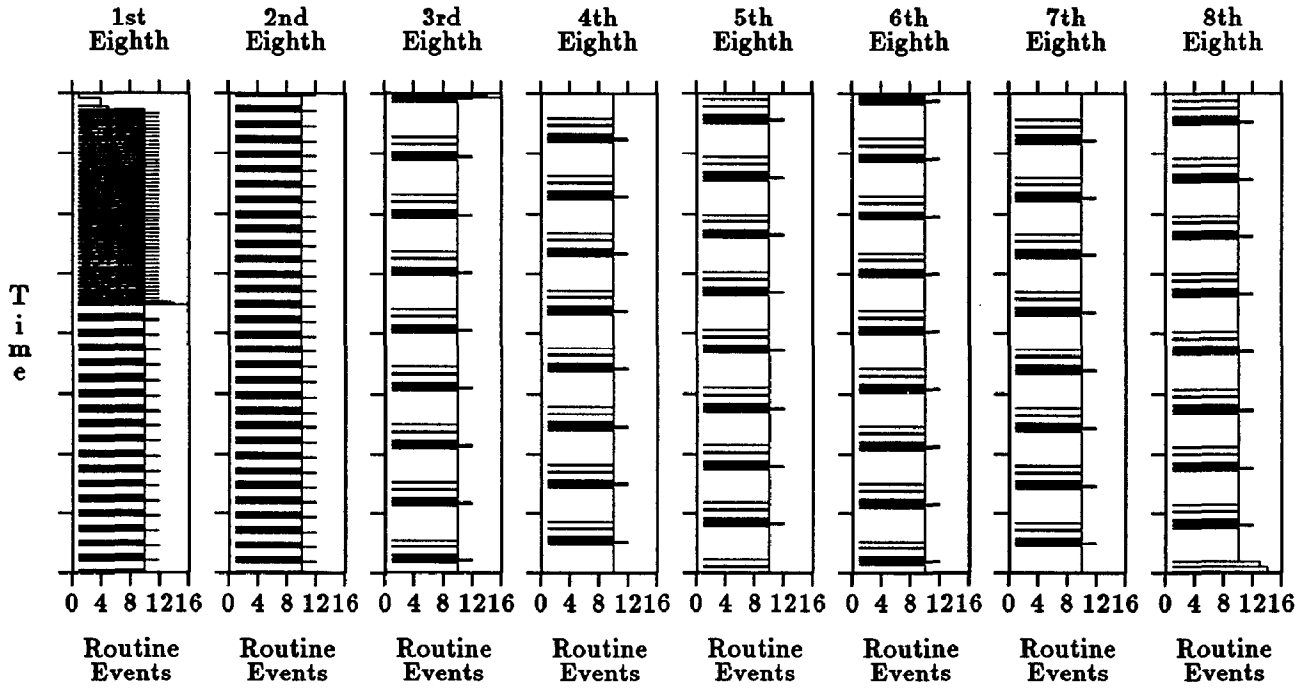

Figure 1: Event Graph for Scalar Execution of FLO52 on Cray X-MP (52.188527 seconds)

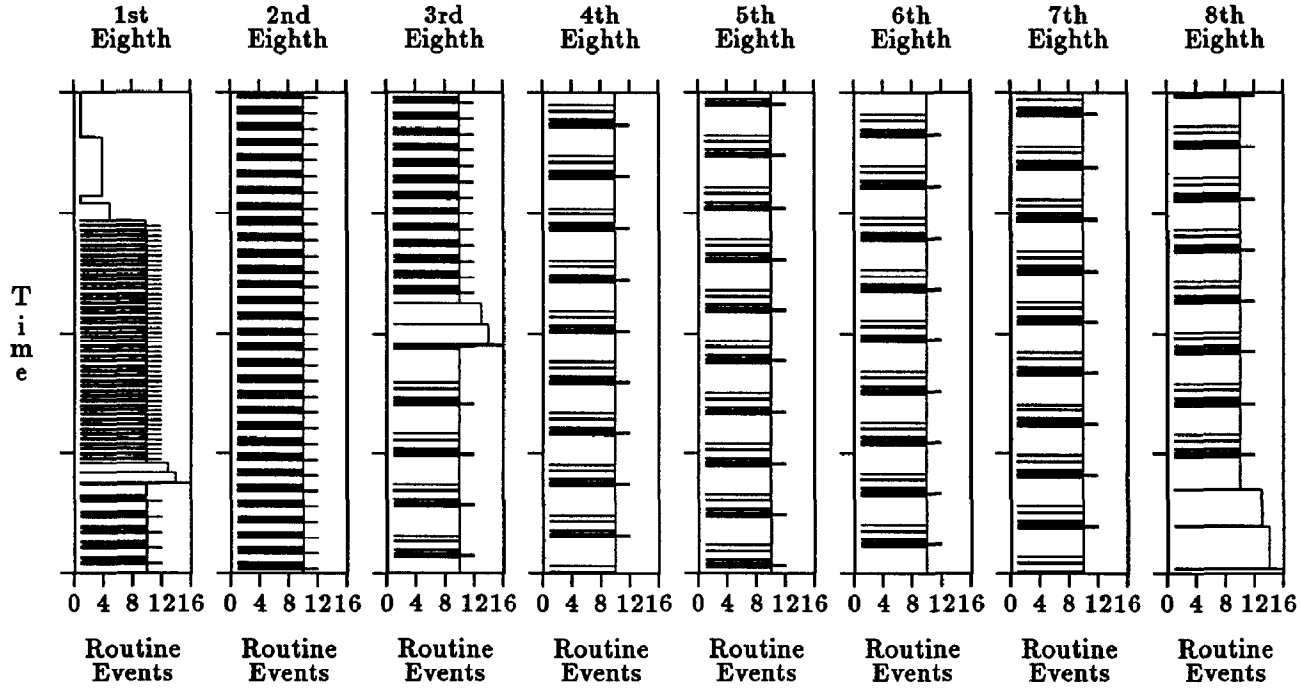

Figure 2: Event Graph for Vector Execution of FLO52 on Cray X-MP (6.749813 seconds)

"grid" phase represents a grid refinement of the FLO52 application's multigrid algorithm. In addition to the procedure calling behavior, Figures 1 and 2 show the changes in procedure execution times across application algorithm phases (i.e., grid refinements).

By comparing procedure event graphs for scalar and vector execution, one can quantify the interactions of vectorization, vector length, and grid size. For FLO52, the lack of vectorization in the initialization and termination phases increases their relative contribution to the vectorized version's total execution time. Moreover, grids two and three, with longer vectors, attain a substantially higher fraction of the Cray X-MP's peak performance in vector mode. This illustrates the value of time-dependent information. The average megaflop rate for this application is the weighted average of the megaflop rates for the three grid computations and depends on both the degree of application code vectorization and the range of vector lengths. These interactions are best understood by direct examination of the time varying application behavior, rather than attempting to infer interactions from first and second moments.

Although not shown, the procedure event graphs for the OCEAN and DYFESM Perfect applications show equally interesting, though different, procedure invocation behavior. As with the FLO52 code, these patterns of procedure calls can be correlated with the applications' algorithms and computation stages.[9] Comparing procedure event graphs from the Cray 2 with those from the X-MP shows non-linear compressions of the event graph time scale. Although the general shapes of the graphs are similar, relative timing differences reflect architectural features, compiler optimizations, and the match of application code to the machines.

6.2 Hardware Performance

In addition to generation and analysis of procedure call data, the *TracePerf* traces permit analysis of dynamic machine performance, as captured by the HPM counters, during a application program execution. Below, we discuss a series of plots that display the time-varying values of hardware performance metrics (in millions of events per second) for the entire execution of our example applications.

To simplify analysis and presentation, we divided each application program execution into five hundred, fixed size intervals of time. In each interval, we computed the average value for each hardware performance metric. Thus, applications that generate larger event traces likely will show less variation across adjacent time intervals (e.g., a trace for OCEAN contains ten times as many events as FLO52, and DYFESM contains twice as many as OCEAN). Also, because only one quarter of the HPM counters can be captured during a single program execution, we ran each application program eight times, once for each of the HPM counter groups in scalar and vector mode, to capture a complete set of hardware performance data. In the interests of space, we analyze only a small subset of the captured data.

Figures 3 and 4 show the data from HPM group zero for the scalar and vector execution of FLO52, respectively. Similarly, Figures 5 and 6 show the vector execution of OCEAN and DYFESM, respectively. In each figure's graphs, the horizontal axis is program execution time, and the vertical axis is the rate (in millions of events per second) of the associated hardware performance metric; except for the last graph, which shows the number of memory references per floating point operation. Although the scalar and vector executions of FLO52 show similar behavior, the differences among the FLO52, OCEAN, and DYFESM codes are striking. The behavior of the vector DYFESM code is very regular; in contrast, the vector version of OCEAN code shows substantial variations in the floating point execution rate and the number of issued instructions. On a vector architecture, instruction issue rate and vector operations are inversely related (i.e., as the number of vector operations increases, the total number of issued instructions decreases – each instructions represents more useful computation). Thus, the behavior of the OCEAN code likely is attributable to frequent transitions to and from vector mode.

Although the application source code embodies a set of application algorithms and an associated number of arithmetic operations, a program compilation potentially can produce many different executable versions that are not work conservation equivalent. For example, a vectorized version might introduce redundant arithmetic operations to increase the vectorization level and execution performance. Thus, when comparing the performance of an application program across scalar and vector modes, one must verify that work is conserved. For the FLO52 code, the total number of floating point operations (6.43×10^8) is the same for both the scalar and vectorized versions.[10] Below, we compare and contrast the two versions of this code based on data obtained from the HPM counters.

In Figures 3 and 4, the three phases of grid evaluation are clear. In the vector case, the successively larger grids have longer vectors. This is reflected in higher megaflop rates (the fifth graph) and a lower instruction issue rate (the first graph). In both figures, "clock periods holding issue" measures conflicts for access to both registers and functional units that prevent release of an instruction to a functional unit. As vector length in-

[9] Space constraints preclude a complete discussion of the procedure event graphs for the Cray 2 and for the other two Perfect codes.

[10] The fifth graph of Figures 3 and 4 shows the time varying rate of these floating point operations in millions per second (i.e., megaflops).

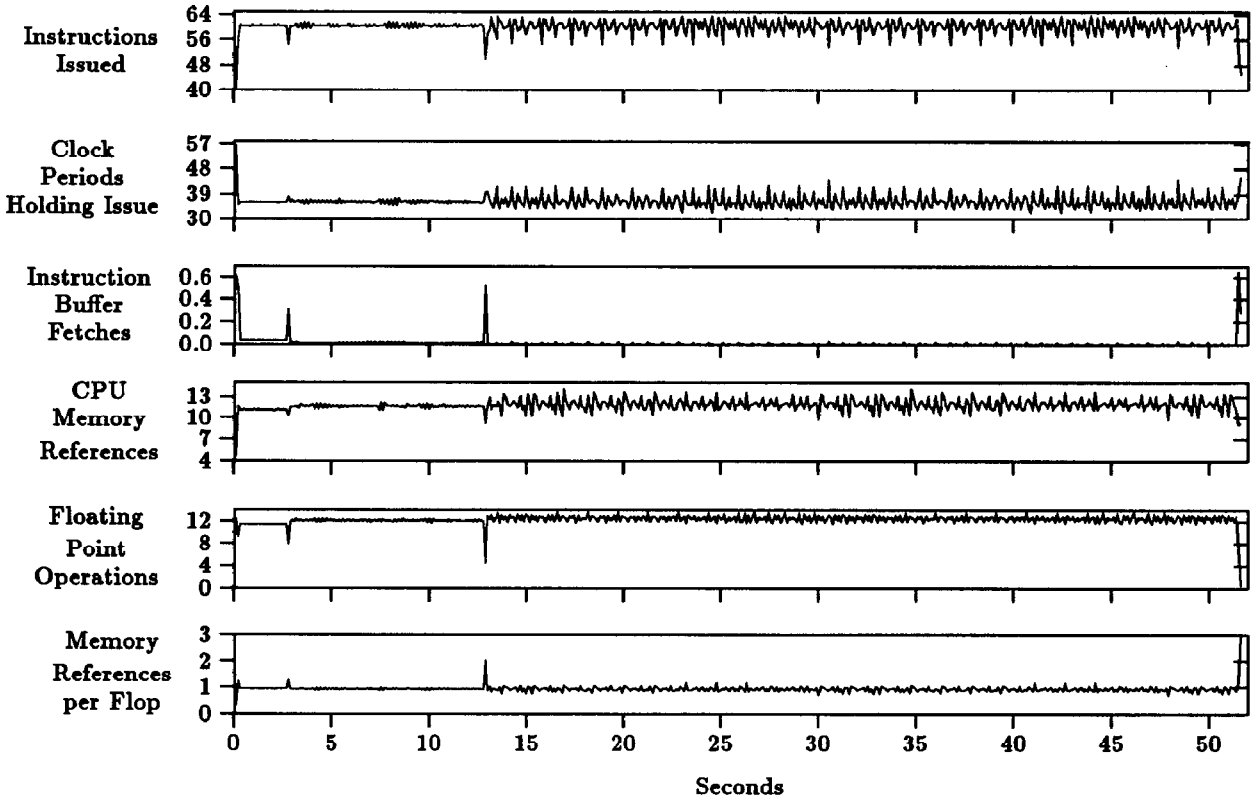

Figure 3: TracePerf Data for FLO52 Scalar Execution - Counter Group 0

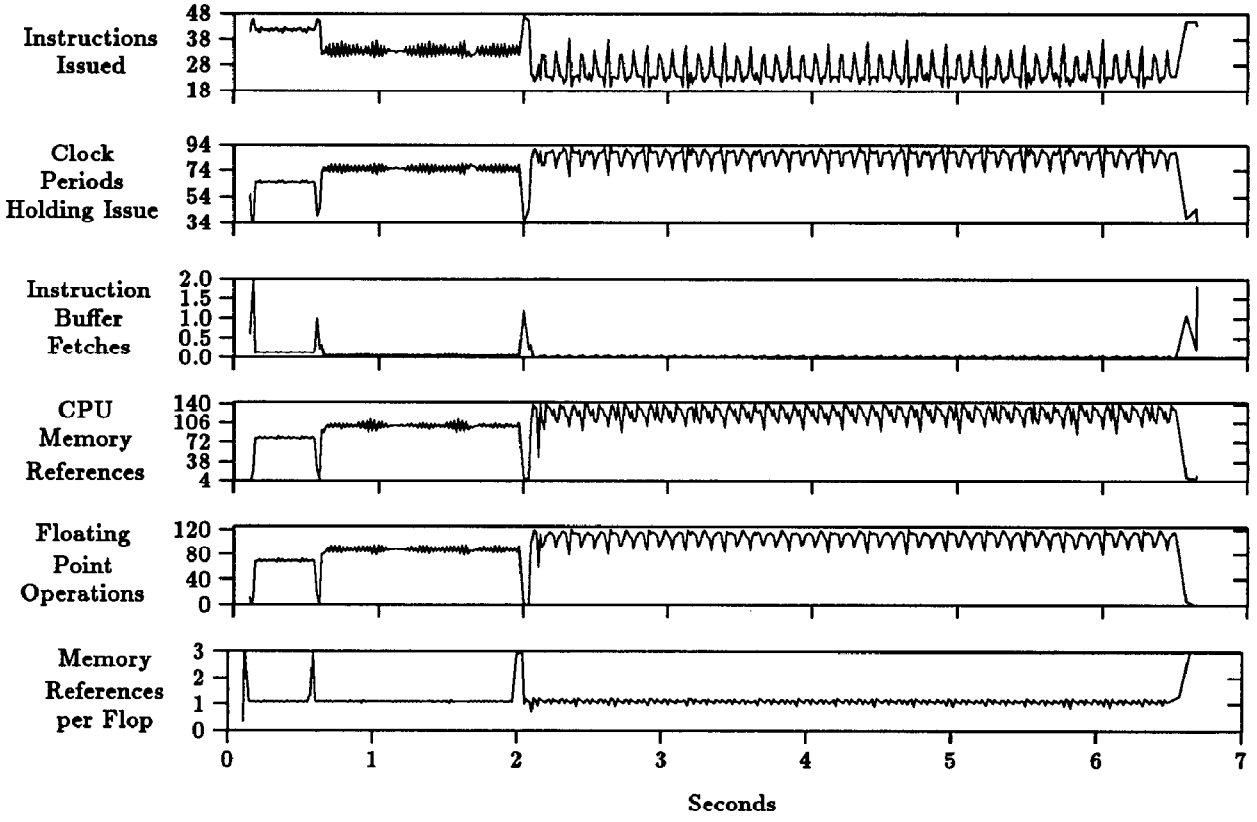

Figure 4: TracePerf Data for FLO52 Vector Execution - Counter Group 0

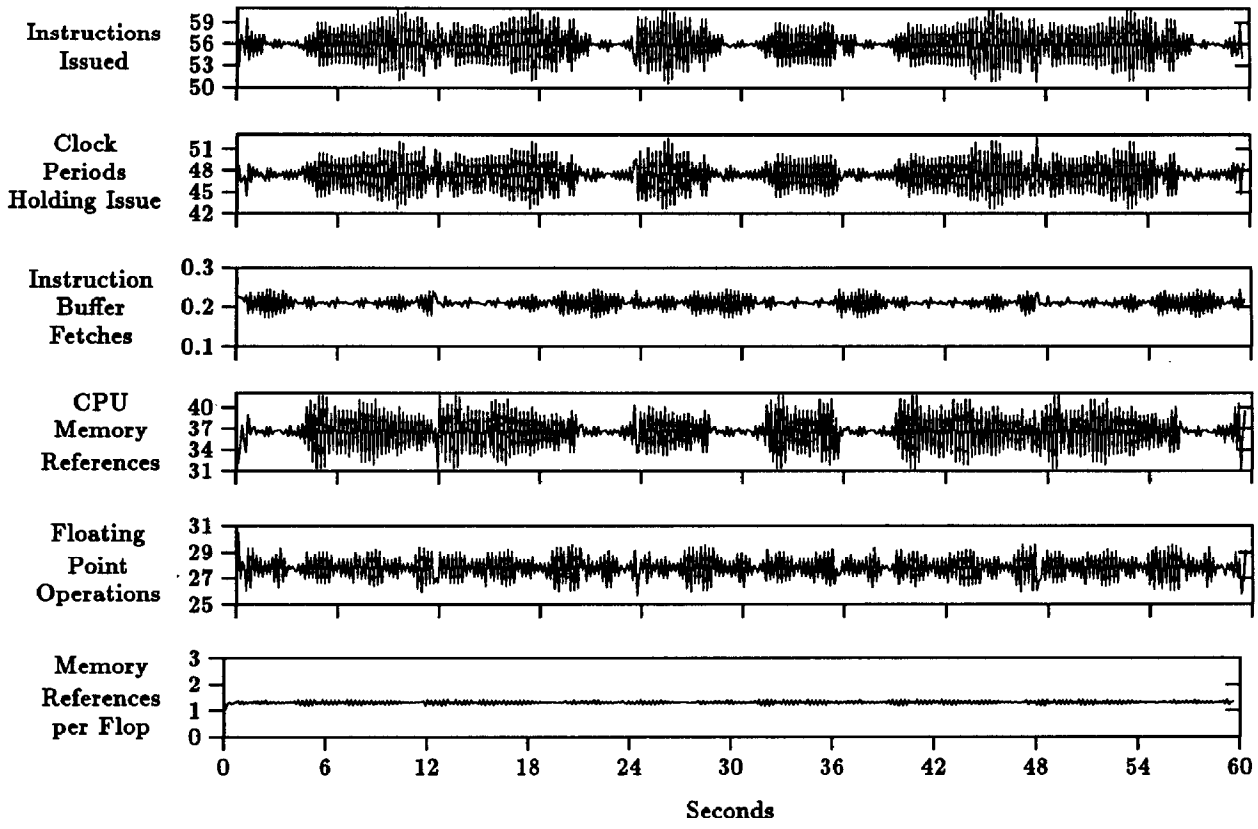

Figure 5: TracePerf Data for OCEAN Vector Execution - Counter Group 0

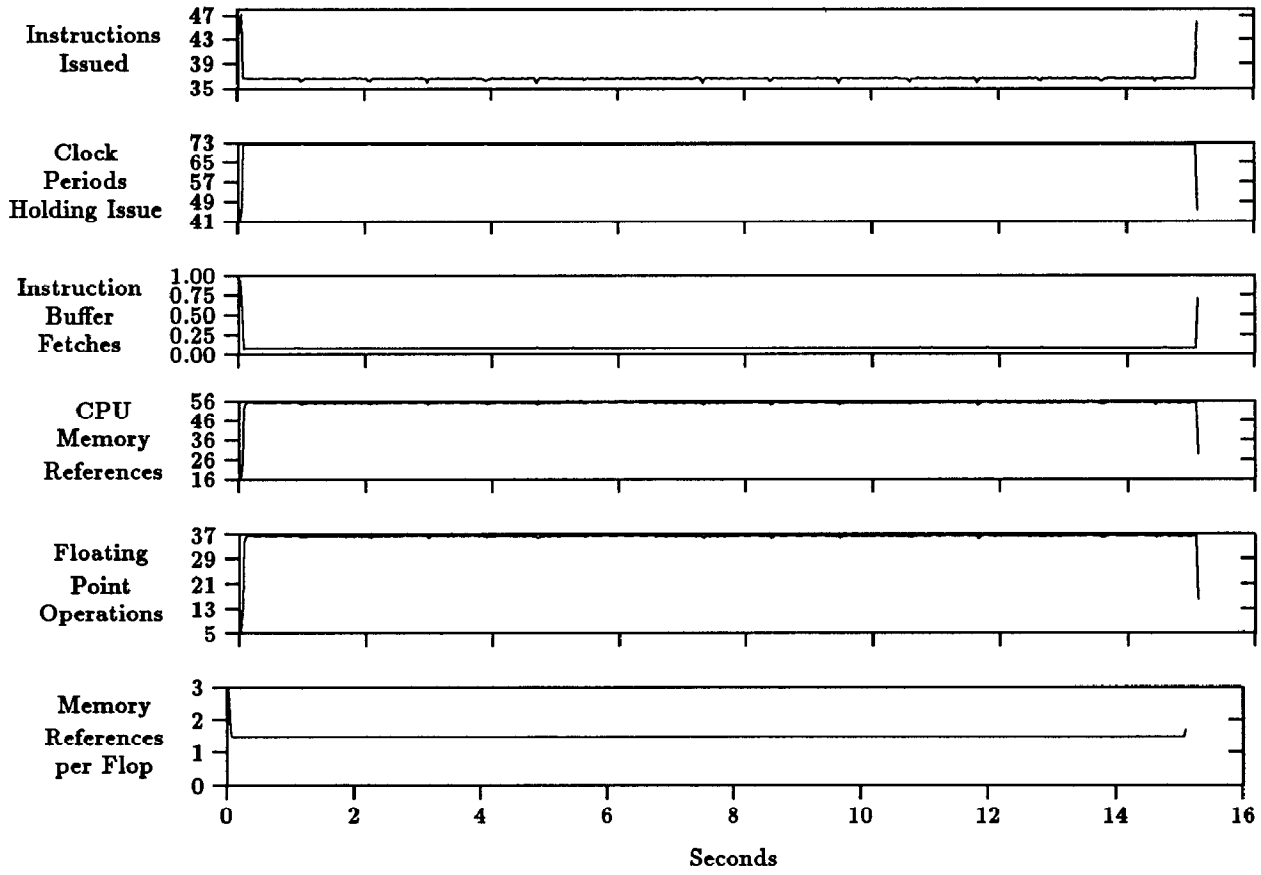

Figure 6: TracePerf Data for DYFESM Vector Execution - Counter Group 0

creases, more time is spent waiting for vector registers and vector functional units.

Because the Cray X-MP contains an instruction buffer, not every instruction fetch generates a memory reference. For example, if all code for a loop resides in the instruction buffer, instruction fetches from memory will cease during loop execution. As the third graph of Figures 3 and 4 suggests, the rate of instruction buffer fetches from memory is low, and the interference with operand memory fetches is small. However, examining both the scalar and vector executions of FLO52 shows that the number of instruction buffer fetches increases sharply during the transitions between grid sizes. In addition, the instruction buffer fetch rate is inversely proportional to megaflop rate for vector execution. Simply put, as the vector length increases, fewer instructions are needed to realize the same number of floating point operations.

Coupling our analysis of instruction buffer fetches with the graphs of CPU memory references confirms that most memory operations are for data access. In Figure 4, the memory reference rate is directly proportional to the megaflop rate.

Although space permitted showing only the total floating point operation graph, we found that the floating point reciprocal, addition, and multiplication were directly proportional for the FLO52 computation.

Finally, the average number of memory references for each floating point operation is near one. Although this might suggest that a single memory port, as found in the Cray-1, might suffice, the small-scale variance is large.[11] Without the Cray X-MP's three memory ports, performance would degrade substantially.

6.3 Megaflop Distributions

Although hardware performance graphs show the time-varying behavior of critical hardware metrics, it is important to correlate this behavior with the execution of particular application procedures. The *Perftrace* summary statistics show hardware performance data for each procedure, averaged over the the computation lifetime. However, variations in these metrics exist, and they depend both on each procedure's computation and its input data. Fortunately, from the *TracePerf* traces, one can calculate these per procedure statistics (e.g., see Table 8), and one also can determine performance variations.

To understand the variations in measured megaflops for each procedure invocation, we generated *megaflop distribution graphs* that show the percentage of all calls made to a procedure that executed at a given megaflop rate. As an example, Figure 7 shows the megaflop distribution graph for the FLO52 code in both scalar and vector execution modes. Although the figures do not show the fraction of application execution time attributable to each procedure call instance, the megaflop variations are clearly visible. For some procedures, the megaflop range is small. For others, such as PSMOO and EULER, the range is quite large. As before, much of this behavior can be explained based on knowledge of the application program. The PSM00 and EULER procedures are primary components of the FLO52 grid computation, and the variations in megaflop rates reflect the changes in vector lengths across the three grids.

In addition to the producing megaflop distributions, using the *TracePerf* traces, it is possible to compare scalar and vector hardware performance for all instances of a procedure. For megaflops, this permits calculation of speedups for each procedure instantiation. In general, knowing the range of procedure's vector performance, rather than just the mean, provides greater insight into optimization potential.

7 Conclusions

It is becoming increasingly apparent that to achieve high performance on supercomputers, the application programmer must have a better understanding of the dynamic execution characteristics of the computation and its interactions with the high-performance features of the machine. However, tools for capturing such data are not commonplace, and currently, users must instead rely on execution summary statistics provided mainly by routine-based profilers. We have developed a tracing facility for the Cray X-MP and Cray 2 supercomputers that has been demonstrated to provide important insight into time-dependent execution behavior while maintaining a high degree of accuracy. The tracing tools provided will port directly to the Cray Y-MP without change. The existence of a common tracing environment across the Cray family will permit performance experiments across machines and will provide a basis for cross-architecture performance studies.

In general, there are several potential advantages provided by dynamic tracing over conventional execution profiling. As a framework for future applications of our tracing facilities, and for potential tracing implementations on other systems, we enumerate some of the more important advantages below.

- A dynamic calling trace can be analyzed to show time-dependent calling relationships between program units.
- Program execution phases can be identified, enhancing the understanding of program function.

[11] Recall that each graph point represents the average over 1/500th of the computation. The size of these intervals hides most of the small-scale variance.

- Different modes of execution (sequential, vector, or multitasking) can be compared at a finer level to show the effects of different source code optimizations.
- Parameters for execution models of certain program structures, such as DO loops, can be derived from the trace data.
- A realistic evaluation of performance improvement potential can be made from a study of performance variability of routine execution.
- Given access to hardware performance data, the high-performance features of the machine can be more accurately correlated with time-varying performance behavior.
- In theory, decisions regarding future parallelism choices (vectorization and multitasking) can be made at run-time based on previous trace-based performance knowledge of a program and its current performance behavior. This allows the system to reconfigure the parallelism of the program to better match the parallelism of the hardware.
- Comparisons between architectures based on specific, local information can be made. Timing and performance characteristics deemed intrinsic to the program may be used to directly evaluate the interaction of program constructs with architecture features.
- More appropriate selection of code characteristics for use in performance prediction model design can be made.

References

[1] M. Berry. The Perfect Club Benchmarks: Effective performance evaluation of supercomputers. *The International Journal of Supercomputer Applications*, 3(3):5–40, Fall 1989.

[2] Cray Research Inc. *UNICOS Performance Utilities Reference Manual*, May 1989.

[3] J. Curry, J. Herring, J. Loncaric, and S. Orszag. Order and Disorder in Two- and Three-Dimensional Benard Convection. *Journal of Fluid Mechanics*, 174:1, 1984.

[4] S. L. Graham, P. B. Kessler, and M. K. McKusik. An Execution Profiler for Modular Programs. *Software — Practice and Experience*, 13:671–685, 1983.

[5] S.L. Graham, P.B. Kessler, and M.K. McKusick. gprof: A Call Graph Execution Profiler. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, pages 120–126, Boston, MA, June 1982. Association for Computing Machinery.

[6] A. Jameson. Solution of the Euler Equations for a Two-Dimensional Transonic Flow by a Multigrid Method. *Applied Mathematics and Computation*, 13:327, 1983.

[7] J. Larson. CRAY X-MP Hardware Performance Monitor. *Cray Channels*, 1985.

[8] J. Larson and R. Lutz. Perftrace User Guide. Technical report, Cray Research Inc., August 1985.

[9] A.D. Malony, J.L. Larson, and D.A. Reed. Tracing Application Program Execution on the Cray X-MP and Cray 2. Technical Report CSRD No. 985, University of Illinois at Urbana-Champaign, Center for Supercomputing Research and Development, May 1990.

[10] A.D. Malony, D.A. Reed, and H. Wijshoff. Performance Measurement Intrusion and Perturbation Analysis. Technical Report CSRD No. 923, University of Illinois at Urbana-Champaign, Center for Supercomputing Research and Development, October 1989. submitted to IEEE Transactions on Parallel and Distributed Systems.

[11] A. Noor and J. Peters. Model-Size Reduction Techniques for the Analysis of Symmetric Anisotropic Structures. *Engineering Computations*, 2(4):285, 1985.

[12] L. Pointer. Perfect: Performance Evaluation for Cost-Effective Transformations - report 2. Technical Report CSRD No. 964, University of Illinois at Urbana-Champaign, Center for Supercomputing Research and Development, 1990.

[13] R. Snodgrass. A Relational Approach to Monitoring Complex Systems. *ACM Transactions on Computer Systems*, 6(2):157–196, May 1988.

[14] C.-Q. Yang and B. P. Miller. Performance Measurement for Parallel and Distributed Programs: A Structured and Automatic Approach. *IEEE Transactions on Software Engineering*, 15(12):1615–1629, December 1989.

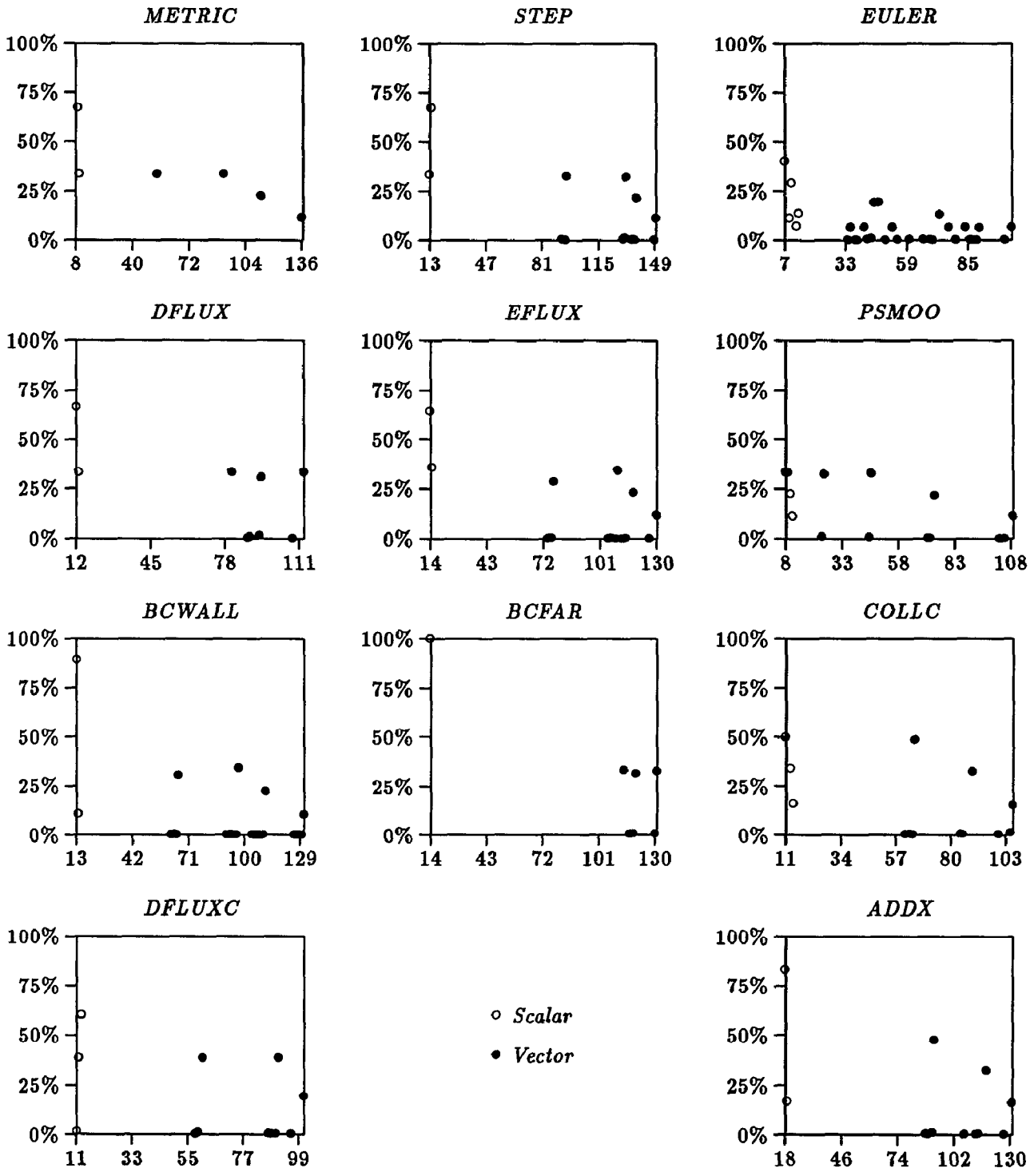

Figure 7: Megaflop Distribution for FLO52 Subroutines on the Cray X-MP