--

--

The Integration of Application and System Based Metrics in a Parallel Program Performance Tool

Jeffrey K. Hollingsworth
hollings@cs.wisc.edu

R. Bruce Irvin
rbi@cs.wisc.edu

Barton P. Miller
bart@cs.wisc.edu

Computer Sciences Department
University of Wisconsin−Madison
1210 W. Dayton Street
Madison, Wisconsin 53706

Abstract

The IPS-2 parallel program measurement tools provide performance data from application programs, the operating system, hardware, network, and other sources. Previous versions of IPS-2 allowed programmers to collect information about an application based only on what could be collected by software instrumentation inserted into the program (and system call libraries). We have developed an open interface, called the ‘‘external time histogram’’, providing a graceful way to include external data from many sources. The user can tell IPS-2 of new sources of performance data through an extensible metric description language. The data from these external sources is automatically collected when the application program is run. IPS-2 provides a library to simplify constructing the external data collectors.

The new version of IPS-2 can measure shared-memory and message-passing parallel programs running on a heterogeneous collection of machines. Data from C or Fortran programs, and data from simulations can be processed by the same tool. As a result of including the new external performance data, IPS-2 now can report on a whole new set of performance problems.

We describe the results of using IPS-2 on two real applications: a shared-memory database join utility, and a multi-processor interconnection network simulator. Even though these applications previously went through careful tuning, we were able to precisely identify performance problems and extract additional performance improvements of about 30%.

Research supported in part by National Science Foundation grant CCR-8815928, Office of Naval Research grant N00014-89-J-1222, a grant from Sequent Computer Systems Inc., and a Digital Equipment Corporation External Research Grant.

To appear in Proceedings of the 1991 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.

1. Introduction

A performance tool should provide as complete a picture as possible of a program's execution. The IPS-2 parallel program performance tools[1, 2] allow a programmer to monitor and analyze the performance of application programs running in both shared-memory and message-passing environments. The performance of an application program can be studied at varying levels of detail, from the whole program down to an individual procedure or synchronization variable. IPS-2 provides a powerful, interactive, graphic user interface for both specification of the program to be studied and for display of the performance results. These tools use software instrumentation to collect the necessary trace data. The standard data collected by IPS-2 relates only to the performance of the application program, as opposed to the system on which the application is running. We have extended IPS-2 with a new facility that allows us to include performance information from sources other than standard instrumentation. Data relating to operating system, hardware, network, or user-defined performance metrics is now integrated into the IPS-2 data analyses and displays.

The original implementation of IPS-2 provided a single interface between the performance data that was collected and the tools that analyzed the data. This interface was the *trace log*. As the application program runs, traces are generated for interesting events; these events include procedure entry and exit, synchronization operations, process operations, and I/O operations. The analysis tools would process this log and allow the user to interactively study the performance of the program execution. The separation of the trace generation and analysis allows the programmer to (optionally) save the traces, with the possibility of archiving and future study. In addition, other tools, such as simulations, also generate traces using the IPS-2 format, providing these users with the full power of IPS-2's analysis and display facilities.

We have added a new external interface to IPS-2, based on the *time histogram*. Each time histogram is a data structure that describes the value of a single performance metric over time. IPS-2 uses this data structure to summarize the information contained in the trace log. The summarized information allows for the efficient display of graphs and tables. We now allow a wide variety of external sources for these time histograms. These external sources can generate performance data for metrics beyond the standard ones supported by IPS-2. When the application programmer describes a program to be run, they also can describe *external data collector* processes that will produce time histograms. These external processes may obtain their data from the operating system (such as disk I/O or context switch rates), special hardware facilities (such as bus traffic or cache hit rates), the network (such as Ethernet collisions), or high level, user defined metrics (such as transaction commit/abort rates).

The inclusion of the external metrics adds a new dimension to the type of information that can be used to study a program's performance. The cause of a performance problem may not lie directly in the application program; it might be caused by resource contention in the operating system (with other programs) or by hardware limitations (such as bus contention). The external metrics allow us to include these factors and present them to the programmer. Constructing these external data collectors is a simple activity. We provide a library of routines that insulates the programmer from all of the details of the data structures.

The usefulness of a performance tool is determined by how much it can improve real programs. To illustrate how our tool works, we present two case studies in tuning parallel programs. These examples show step by step the process of using IPS-2, and include screen dumps showing the output produced by the tool.

The next section presents an overview of the basic IPS-2 system. In this section we describe the program and measurement model of the IPS-2 system. The design of the external histogram interface is described in Section 3. Section 4 describes the implementation of the external histograms and metrics, and describes three External Data Collectors that we have written. Section 5 describes a mechanism for defining new metrics derived from a combination of existing ones. Section 6 presents two case studies that use IPS-2. Section 7 presents our conclusions and ongoing research.

2. Overview of IPS-2

The system currently runs on versions of the UNIX operating system, and handles programs written in both C and Fortran. Programs can run on VAX, DECstation, and Sequent Symmetry machines (or a combination of these machines). The power of the IPS-2 user interface comes from the abstraction of a parallel program that is presented to the application programmer. The next section describes this abstraction, and Section 2.2 outlines how the system implements this abstraction.

2.1. User's View

IPS-2 presents information about programs to the user in a hierarchical form. This presentation permits the user to start at the top of the hierarchy and see the global view of the program, and work down to lower levels getting more detailed information at each lower level. The tool provides a large amount of performance information, and using this approach makes getting information easier and more intuitive. By working down the program hierarchy, the user can refine their view of the program as they search for performance bottlenecks.

There are four levels in the hierarchy. *Program level* is the highest level, and includes information for the entire

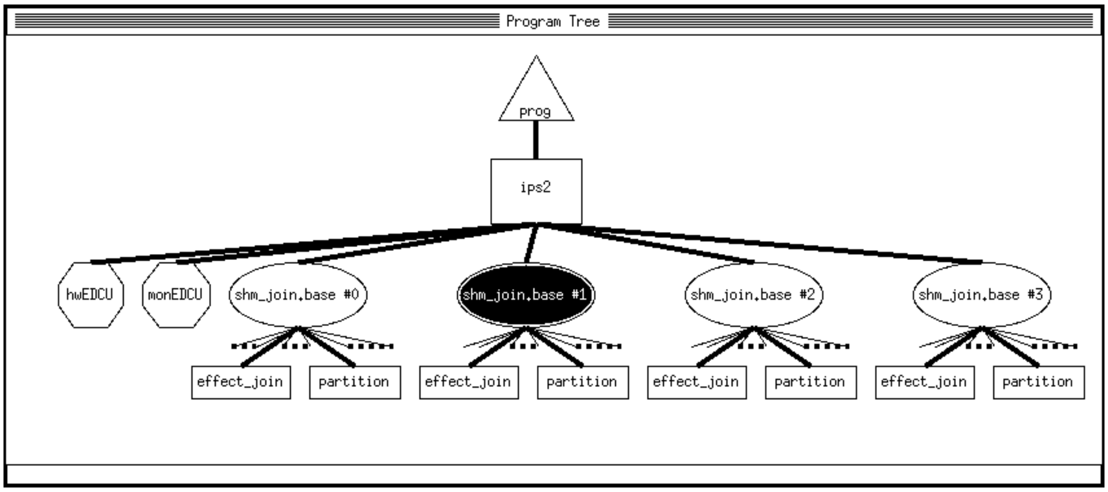


Figure 1.
A Sample Program Tree Hierarchy.

parallel program. *Machine Level* level includes information about each machine. The *Process Level* includes information about specific processes that ran on each machine. The lowest level in the hierarchy is the *Procedure Level*. This level includes information about each procedure that was called during the program's execution. A sample program tree appears in Figure 1.

For each level in the hierarchy there are many performance metrics that can be displayed. For example, the amount of CPU time used can be displayed for the whole program or for individual machines, processes and functions. The performance metrics are available in several types of displays: tabular summary, time histograms, Critical Path profiles, and sorted profile tables. In addition, an experimental version of IPS-2 exists that provides an implementation of the Quartz Normalized Processor Time metric[3].

The simplest kind of metric in IPS-2 is a table metric. A table metric has a single value for the entire length of the program's execution. For example, tabular CPU time at the procedure level is how long a particular procedure ran, and at the process level it is the total CPU time for that process. At the higher levels, it is the sum of the CPU time of the lower levels.

Time histograms show how a metric changes during a program's execution. Histograms are displayed in a window that can be panned and zoomed to permit both global and detailed views of the information. Figure 4 shows a sample histogram. Several histograms can be plotted on the same axis. This permits correlation of different metrics. Like tables, histograms are available at any level in the program tree hierarchy.

Critical Path analysis is based on identifying the path through the program's execution that consumed the most time[4]. This technique identifies the part of the program that was responsible for its length of execution. To calculate a program's Critical Path, we build a graph of the program's execution. This graph consists of the events in each process and the synchronization dependencies (e.g. interprocess communication, locks, or semaphores) between the processes. By finding the longest time-weighted path through this graph we are able to determine which components of the program's execution were responsible for the program's runtime. Critical Path data is presented in tabular form sorted by contribution to the longest path though the graph. This information is available at any level in the program tree hierarchy. For example, at the procedure level each procedure is listed along with its contribution to the Critical Path. Critical Path analysis also provides a what-if mechanism that permits users to judge the effect of improving a component on the Critical Path. We permit users to change the weight of a component on the Critical Path to zero and then recalculate the Critical Path. This only provides an approximation of the new run time of the program, but it does help to assess the impact of improving sections of code.

Profile tables are similar to the type of information presented by the standard UNIX profiling tool Gprof[5]. At the procedure level, profile tables list procedures sorted by the amount of CPU time used. In addition, profile tables list the time each procedure spent in synchronization waits. At higher levels, profile tables summarize the information from the lower levels. Information from all of the processes in the application program are integrated into a single table for each level in the hierarchy.

2.2. Implementation

IPS-2 consists of four parts: the User Interface, Master Analyst, Slave Analyst, and instrumentation probes. The Master and User Interface are a single process. The User Interface, built using the X Window system, provides a graphical representation of the program hierarchy and generates the various performance displays. It supports both monochrome and color screens. The Master Analyst aggregates the information from Slave Analysts and provides it to the user interface.

The instrumentation probes are implemented as a replacement for the standard C library. They produce trace records for each interesting event in a program's execution. Interesting events include procedure calls and returns, message passing, synchronization events (e.g. semaphores and spin locks), I/O, and process creation. Trace records include a time-stamp for the event (both process and wall clock time) and event specific information.

There is one Slave Analyst on each machine running the application. Slave Analysts start the application processes, wait for the application to finish, and then read the trace logs produced by the instrumentation library. The Slave Analysts also build time histograms based on information in the trace logs.

3. Design of External Histograms

The time histogram is a major internal format within IPS-2. It provides detailed information about how a metric varies over time. In addition, each time histogram can be summarized into a single value, to provide a cumulative view of a particular metric. For example, a time histogram for the number of file operations can be summed over all intervals to produce the total operations during the program. A time histogram of virtual memory usage can be averaged to show the average virtual memory in use during the program's execution. Time histograms also provide a natural interface for importing data from external sources.

There are three steps in creating a time histogram for an external performance metric. First, the metric is specified, including the type and source of data. Second, the data is collected (into histogram data structures) during the application program's execution. Third, the Slave Analysts process the data for display.

To smoothly integrate external information into IPS-2, each external metric needs to be attached at an appropriate node in the program tree hierarchy. For example, system memory bus contention only makes sense at the machine level, since it is based on the total activity running on the machine. The level at which an external metric is placed in the tree depends on the type of the metric and granularity of the data. Once the data is collected, IPS-2 aggregates metrics from their initial level in the hierarchy up to the higher levels. The type of aggregation required (summation or averaging) depends on the metric.

External histogram data is collected by processes called *external data collector units* (EDCU). The EDCUs are the interface between the source of the data (operating system, hardware, network, or application) and IPS-2. An EDCU consists of two parts: the metric specific part and the histogram generation part. The metric specific part must know how to collect samples of the data; this might be via calls to the operating system, reading special device registers, or reading special memory locations. The histogram generation part, storing and summarizing the data, is made simple by use of the EDCU library. The EDCU collects the metric (or set of metrics), and periodically calls a function in the EDCU library to indicate the current value of the metric. From these sample values, the EDCU library creates the histograms, and at the termination of the application it stores them in files.

External metrics can also be generated directly by the application program. The programmer may want to monitor an activity whose semantics are understood by the program. For example, to record the number (and frequency) of commits and aborts in a transaction system, the programmer would insert EDCU library calls into the appropriate place in their program†. The interface to the EDCU library is the same as for a standard EDCU, except that the data is coming from the application process and not from a separate collector process.

An IPS-2 user who wishes to collect external data simply creates an EDCU node while editing the IPS-2 program tree (see nodes "monEDCU" and "hwEDCU" in Figure 1). An EDCU node is similar to a process node in that it appears at the process level of the tree and contains a command line to execute.

When the application program terminates, the Slave Analyst collects the various histogram files. Based on the data in each histogram, the Master Analyst places the metric in the appropriate node in the program tree. Once placed in the tree, the fact a metric is external becomes transparent to the tool user who is free to display histograms and tables consisting of both external and internal data.

4. Implementation of External Histograms

We have extended IPS-2 in several ways to handle externally generated performance data. The EDCU library provides the interface between EDCUs and IPS-2. Metric description files allow EDCU programmers to describe external metrics to IPS-2. The control structure of IPS-2 handles the execution and termination of EDCUs. Finally, IPS-2 allows external data to be inserted at any node in the program tree and allows users to select and display these metrics with the same menus, tables and graphs as built-in metrics.

4.1. Writing an External Data Collector

A typical EDCU contains an initialization section and a main body. During initialization, an EDCU allocates histograms for each metric that it will collect. An EDCU consists of a main loop which collects data samples and deposits them into histograms. Since the duration of an EDCU's execution is generally not known ahead of time, the main loop iterates indefinitely until the process receives a termination signal from IPS-2. The EDCU library catches the signal, writes the histograms to a file and terminates the EDCU. The EDCU library simplifies writing an EDCU by providing data abstractions for histograms and timestamps, and by handling all communication with the Slave Analyst. Therefore, the programmer need only understand the interface presented by the library to write an EDCU, and the Slave is insured that the data generated is stored in a compatible format.

Each data sample must include a timestamp. The user may furnish timestamps or ask the EDCU library to collect them. If a data sample's timestamp exceeds the pre-allocated time limit of the histogram, the library automatically compacts the histogram and expands its time limit to accommodate the new sample. The histograms can therefore expand dynamically in time without allocating more memory.

In addition to providing a means for collecting external data, an EDCU programmer must also describe each metric to the Master Analyst. These descriptions are contained in a metric description file that is read when IPS-2 starts a measurement session. A metric description includes a metric's name, the tree level of the metric, how the metric is to be labeled in menus, tables, and graphs, how the metric is to be calculated for tables, and how higher levels should aggregate the metric from lower levels (see Figure 2). When creating a histogram during execution, an EDCU must specify a name for the metric and the node in the IPS-2 program tree to which it will be attached. The name given for the histogram's metric must be defined in the EDCU's metric description file.

```
metric prod_bus_util {
    class external;
    level machine;
    aggregation sum;
    tablecalc average;
    title "Productive Bus Utilization";
    xlabel "(Prod. Bus Util.)";
    units percent_bus;
    tabletitle "Average Bus Utilization";
}
```

Figure 2.
A Sample Description of an External Metric.

† This is the only type of data collection in IPS-2 that requires the application program to be modified, and this approach is **not** required for normal use of IPS-2.

4.2. Current External Data Collectors

To date we have written three external data collectors: one that filters information from the UNIX utility ‘‘vmstat’’, one that collects information from hardware counters in the Sequent Symmetry, and one that collects kernel level information in the Dynix operating system (Sequent’s parallel version of the UNIX). We anticipate a core set of data collectors that will become standard, and will be provided with the IPS-2 tools. Additional collectors will be created by users for more specialized uses.

The Sequent Symmetry[6] is a shared memory multi-processor. Each processor has a set-associative cache, and connects to main memory via the system bus. The hardware data collector (‘‘hwEDCU’’) collects metrics from special hardware monitors in the Sequent Symmetry, including utilization of the system bus, and cache miss information for each processor.

The kernel data EDCU samples kernel tables to gather information about the state of the system. It collects data about the utilization of the processors, virtual memory system, network traffic, and disk system. The collector is implemented as a user process. To efficiently collect these metrics, it maps sections of the kernel’s memory into its address space. Samples are collected at only 10 millisecond intervals, which provides sufficiently accurate information without causing much bus traffic.

4.3. Executing External Data Collectors

When the user runs their application, the IPS-2 Master sends EDCU and process command lines to the Slaves. Each Slave Analyst starts the EDCU processes before starting any application processes. When all application processes under its control are finished, the Slave Analyst terminates each EDCU and then reads the histogram files.

Upon reading an external histogram file a Slave attempts to insert it into a node in the program tree. If a histogram belongs to a tree node that is not under the Slave’s control, it sends the histogram to the Master. In this way, a Slave can handle external data for other machines or for the program level of the tree (e.g. network statistics). Once a Slave has finished collecting and placing external histograms, it forms aggregate histograms at the process and machine levels using the aggregation attributes found in the metric descriptions. Finally, the Slave reports to the Master, listing the metrics that are defined at each node. The Master uses this information to generate menus for the user interface. An EDCU is only one method of collecting external histograms. An IPS-2 user may collect data that is pertinent to a particular application by modifying the application to create and fill histograms. The Slaves read user defined histograms when they process program trace logs.

5. Derived Metrics

IPS-2 provides direct access to many different metrics − those that are built-in and those collected from external sources. IPS-2 also provides an added degree of flexibility. The tool user can specify new metrics, *derived metrics*, that are combinations of existing metrics. For example, IPS-2 provides a metric for file operations per second and a metric for message operations per second. The user could specify a new metric that is the sum of these two built-in metrics. Like any other metric, a derived metric is associated with a node in the program tree hierarchy. It is defined by simple expressions composed of arithmetic operators and previously defined metrics (from any of the three sources).

Derived metrics are defined in a metric description file. A derived metric description is identical to an external metric description except that it includes a formula for deriving the metric. For example, the metric *share* shown in Figure 3 divides the process time of the application (useful CPU time plus busy waiting) by the total time the machine spent in user mode to compute the share of the machine’s user time used by the application. *Cpu_time* and *busy_waits* are built-in metrics and *sys_user_time* is an external metric.

```
metric share {
    class derived;
    level machine;
    aggregation sum;
    tablecalc average;
    title "Share of User Time";
    xlabel "(Share)";
    units percent_cpu;
    tabletitle "Average Share";
    formula (cpu_time + busy_waits)
     / sys_user_time;
}
```

Figure 3.
A Sample Description of a Derived Metric.

When a user wishes to view data for a particular node in the IPS-2 program tree, the user interface presents a menu that contains built-in, external, and derived metrics. All classes of metrics can be displayed simultaneously in graphs and tables by simply selecting them from the menus. If the user selects a built-in or external metric for display, the Master checks to see whether it has already received or aggregated the data, and if necessary, the Master requests the data from the appropriate Slave. When the user selects a derived metric, the Master evaluates the expression given in the metric description, requesting the operands of the expression from the Slaves as needed.

6. Case Studies

To test the ability of our system to improve programs, we picked two real programs (not written by us) to see how much we could improve their running times. Both of these programs had previously been tuned by their authors and they had reached the point where they were unable to identify further opportunities for improvement. The first program is an implementation of a join function for a relational database. The second program simulates a memory-to-processor interconnection network in a multiprocessor computer.

For each of these programs, we iterated through a cycle of running the programs using our tools and improving a component of the program. Because our instrumentation of a program could perturb its performance, we also ran each iteration of the program without instrumentation to confirm the validity of the guidance that our tool was providing. This exercise showed that while the absolute numbers might change between instrumented and uninstrumented versions, trends and the percent improvement in the program were similar. All histograms and Critical Path Tables presented in this section are screen dumps of actual displays produced by our tool. The values presented in the percent improvement tables are based on programs compiled without IPS-2 instrumentation and with the compiler's optimizer enabled.

6.1. Shared Memory Join Program

The shared memory join application (*shm_join*) is an implementation of the join function for a relational database. It implements a hash-join algorithm[7] using shared memory for inter-process communication. The program was written to study shared-memory and shared-nothing join algorithms and, as a result, contains a large number of tunable parameters that effect the algorithm's performance. For the purpose of our performance study, we used a fixed set of parameters and attempted to improve the algorithm's implementation. IPS-2 can also be used to tune the performance of the program by adjusting these parameters. The test data consisted of the Wisconsin Benchmark Join ABPrime[8] query with a 50,000 tuple inner and a 50,000 tuple outer relation.

Figure 4 shows a histogram for the initial version of the *shm_join* application running on a four processor Sequent Symmetry. The three curves shown in the figure are CPU Time†, Productive Bus Utilization, and Page Faults. Productive Bus Utilization was collected by the hardware data collector, and Page Faults was collected by the kernel monitor EDCU. We have divided the program's execution into four phases (labeled ''A'' through ''D'' at the top of the histogram). Each phase represents a different part of the program's execution, and the characteristics of the metrics is different in each.

Phase ''A'' is the creation of the shared memory heap and synchronization variables. This phase shows little CPU utilization and initially low bus utilization and page faults. This phase is totally sequential, and contains a large amount of system CPU time. This was discovered by plotting a histogram of system time (not shown). At the end of this phase, the page fault and bus utilization rates increase as the application brings in the code and data required for execution of the second phase.

The second phase (phase ''B'') involves initialization of the private data in each process. There is almost no sharing of information. This can be seen by the relatively low bus utilization. The end of this phase is marked by a stair step decline in useful CPU time. This is caused by processors finishing their part of the data, and waiting at a barrier before proceeding to the next phase.

Phase ''C'', the partitioning of tuples by processor, is characterized by high rates of page faults and bus utilization. Peak CPU rates are less than in phase ''B''. In this phase, new data pages are being used by the processes resulting in overhead (as seen in the page fault rate) that limits the useful amount of work that can be done. Appli-

† CPU time is useful time only and does not include time spent busy waiting.

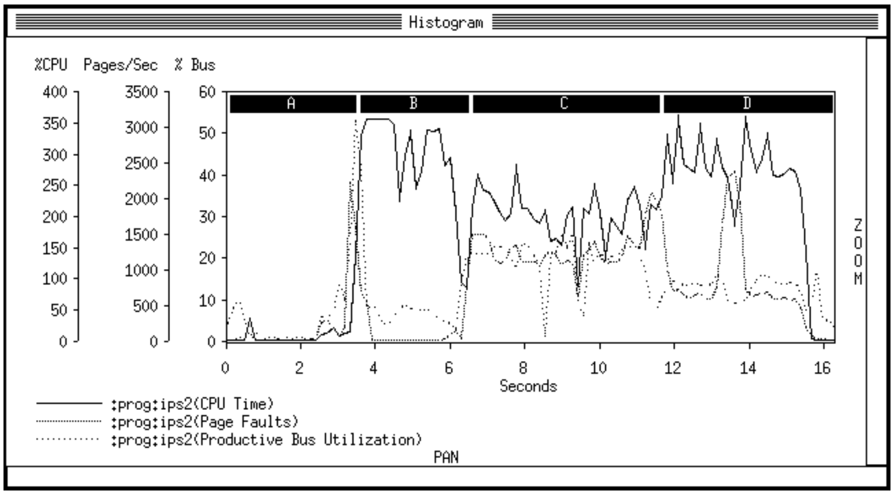


Figure 4.
Histogram of the initial version of the shared memory join application showing built-in (CPU time) and external metrics.

cation metrics alone were not enough to show why the CPU utilization is less than the number of processors working on the problem. During this phase sustained bus utilization reaches its highest level. The application creates large amounts of bus traffic by creating, zeroing and then transferring pages between processors.

The final phase of the program's execution, phase ''D'', shows a moderate page fault rate (between the levels in phases ''B'' and ''C''). The initial spike in the page fault rate is most likely due to data (and possibly code) being brought into memory for use during this phase. The peak in page faults in the middle of phase ''D'' is caused by the join algorithm moving to a new bucket of tuples, and the resulting change in the pages being used. The bus utilization remains about the same (even after the page fault rate drops) due to sharing of pages between processes. The spike in the Bus Utilization at the end of the phase is caused by an increase in disk writes (curve not shown) at the end of the program's execution.

To improve the performance of this program, three major changes were made. The first two changes improved the implementation of the major procedures of phases ''B'' and ''D''. The third change improved the running time of phase ''C'' by reducing the page faulting rate. This was accomplished by making changes to phases ''A'' and ''B''. Each of these changes is described in detail below.

Procedure-level Critical Path

machine:process	func name	time	%time
TOTAL LENGTH		10.67	
ips2:shm_join.base #2	effect_join	2.35	22.02
ips2:shm_join.base #2	random_shuffle	1.44	13.50
ips2:shm_join.base #3	partition	0.96	9.00
ips2:shm_join.base #0	partition	0.81	7.59
ips2:shm_join.base #1	partition	0.78	7.31
ips2:shm_join.base #2	partition	0.74	6.94
ips2:shm_join.base #2	init_relation	0.43	4.03
ips2:shm_join.base #2	writeblk	0.37	3.47
ips2:shm_join.base #0	exchange_buffers	0.30	2.81
ips2:shm_join.base #1	exchange_buffers	0.28	2.62
ips2:shm_join.base #2	get_output_buffers	0.26	2.44
ips2:shm_join.base #3	exchange_buffers	0.24	2.25

Figure 5.
Critical Path table for initial version of shared memory join application.

Figure 5 shows the Critical Path for the initial version of the program. The top function on that list is *effect_join* which accounts for 22% of the Critical Path. When this procedure was examined, we discovered that its major activity was copying data. The copying was done using the standard C library function *bcopy*. *Bcopy* is a general purpose routine that permits arbitrary bytes of memory to be copied, and is implemented as a function, not inline code. By replacing the call to *bcopy* with a hand-coded assembly macro that copies in long-word (four bytes) increments, we reduced the running time of the program by about 15%. Further investigation revealed that two additional procedures, *writeblk* and *readblk*, also did a large number of long-word aligned copies. Inserting our macro into these routines further reduced the program's running time by 4%. The cumulative effect of these changes was to decrease the running time of the program by about 19%. In addition, as Figure 6 shows, *effect_join* is no longer at the top of the Critical Path table. Notice that the total Critical Path length was reduced 31.8% (from 10.67 to 7.27), but the running time was only reduced by 19%. This is due to the large amount of the running time that is due to system time in Phase ''A''.

Procedure-level Critical Path

machine:process	func name	time	%time
TOTAL LENGTH		7.27	
ips2:shm_join.bcopy.ips #3	random_shuffle	1.50	20.63
ips2:shm_join.bcopy.ips #0	effect_join	0.92	12.65
ips2:shm_join.bcopy.ips #1	partition	0.45	6.19
ips2:shm_join.bcopy.ips #3	init_relation	0.44	6.05
ips2:shm_join.bcopy.ips #3	partition	0.44	6.05

Figure 6.
Critical Path table showing improvement in effect_join function.

The next step was to try to improve the new top of the Critical Path, the procedure *random_shuffle*. This procedure is part of the initialization phase ''B'' (see Figure 4). Investigation of this function showed that it is composed of a single **for** loop consisting of a call to a random number generator, and an exchange of elements in an array. Since the exchange was implemented as a swap of pointers, we decided to try to improve the random number generation. The easiest thing to do was to make code for the function in-line. This reduced the running time of the program by 3.5%. It also reduced the Critical Path to 7.02 seconds (table not shown).

Metric Table - Phase C

	prog ips2
CPU Time	9.81
Elapsed Time	5.09
Pages Faulted	6358.22
Speedup	1.93
Zero Pages Created	5599.96

Figure 7.
Table showing parallelism (speedup) and page fault activity.

After the first two steps, we decided to move onto the third item on the Critical Path. This procedure, *partition*, is called during the third phase of the program's execution. During this phase, the effective parallelism (shown in Figure 7 as speedup) is only 1.93. This is due to the large number of zero page creates (also summarized in Figure 7). Zero page creates occur on the first reference to a page in the shared heap. These pages were previously allocated (but not created) by calls to *shmalloc* (the shared memory allocation routine). To improve this phase, we tried to force the operating system to create these pages during otherwise wasted CPU time. Recall from Figure 4 that phase ‘‘A’’ consists of only sequential processing. It is desirable to use this time to create the pages in the shared heap. However, this phase allocates these pages using *shmalloc*. Further investigation showed that phase ‘‘B’’, local data initialization, did not depend on the memory being set up in phase ‘‘A’’. So, we decided to have three processes do their local initialization while the fourth process allocated the shared memory. When the memory had been allocated, the fourth process would start its local initialization. This left time for the other three process, after they finished their local initialization, to create the pages just allocated by the first process. To force the operating system to create the pages, we wrote a procedure that reads the first word of each page. The effect of this reorganization, and additional step was to reduce the running time of the program by over 10%. This type of analysis would not have been possible without the combination of system and program level statistics.

Figure 8 shows metrics similar to those in Figure 4, but is for an execution of the program after we made our changes. The shape of the CPU time curve has changed substantially from the initial version of the program. Phases ‘‘A’’ and ‘‘B’’ now blend together, phase ‘‘C’’ is now achieving better parallelism, and phase ‘‘D’’ has been reduced in length considerably. Figure 9 summarizes the effect of the changes made to the program. The total percent improvement is not as large as the sum of the individual improvements due to interactions between the different changes.

Version	Time	% Change
base	10.3	
improve bcopy	8.3	19.8
inline random	9.9	3.5
force page creates	9.3	10.3
All Changes	7.3	29.4†

Figure 9.
Table of improvements to shared memory join application (time is in seconds).

6.2. Network Simulator

The network simulator application (*psim*) simulates an indirect *k*-ary *n*-cube processor-memory interconnection network. The program has been used previously to evaluate the performance of such networks [9] and to study cache coherence protocols for shared memory multi-processors [10]. *Psim* is an integer intensive application that parallelizes well, exhibiting nearly linear speedup for 20 processors.

The network simulated by *psim* consists of a request sub-network and a response sub-network. The program allocates half of the available processors to each sub-network. All of the processors synchronize at a barrier after each logical clock cycle. For our study we analyzed *psim* running on relatively small test cases with IPS-2. After making improvements, we collected timings by running the program with larger test cases.

† See text for an explanation of why this does not equal the sum of the individual changes.

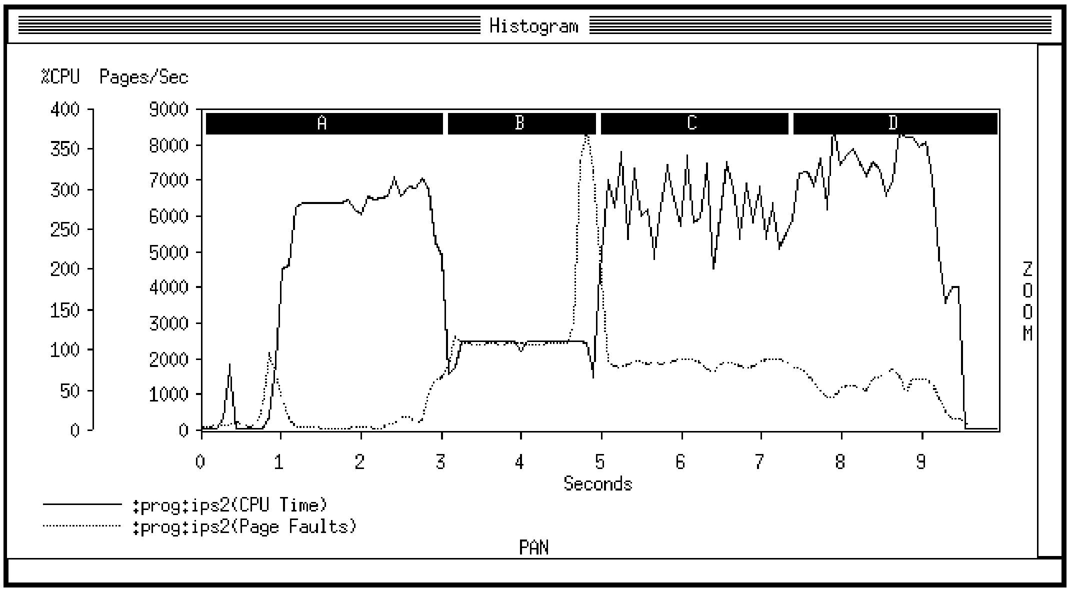


Figure 8.
Histogram of shared memory join application after performance tuning.

The histogram in Figure 10 shows the CPU time curve of the original program simulating a 32 processor/memory network. The program has two primary phases, a relatively constant initialization phase (labeled ‘‘A’’ in the figure), and the main processing phase (labeled ‘‘B’’) whose length varies with the size of the network and the length of the memory request vectors.

Procedure-level Critical Path - Phase A

machine:process	func name	time	%time
TOTAL LENGTH		2.01	
topaz:psim.ips #0	prandi	1.22	60.70
topaz:psim.ips #0	psrandi	0.72	35.82
topaz:psim.ips #0	Process Creation (main)	0.04	1.99

Figure 11.
Critical Path table for phase A of initial version of network simulator.

Figure 11 shows a procedure-level Critical Path table for phase ‘‘A’’. The table shows that the procedures *prandi* and *psrandi* account for over 95% of the Critical Path during this phase. The CPU time curves for these procedures are shown in Figure 10. Because these procedures are called during a sequential section of the program, the other processes must wait at a barrier before proceeding to the next phase. We examined the code and found that *psrandi* and *prandi* initialized a random number array that was never used with most of *psim’s* parameter settings. By running these procedures only when parameter settings required them, we were able to considerably shorten phase ‘‘A’’. However, when we ran the program on large test cases (for which initialization time was negligible), the improvement was less than 1% of total running time.

Procedure-level Critical Path - Phase B

machine:process	func name	time	%time
TOTAL LENGTH		4.48	
topaz:psim.ips #4	putpacket	0.50	11.16
topaz:psim.ips #4	getpacket	0.40	8.93
topaz:psim.ips #4	net_set	0.35	7.81
topaz:psim.ips #3	getpacket	0.33	7.37
topaz:psim.ips #3	putpacket	0.27	6.03
topaz:psim.ips #0	getpacket	0.27	6.03
topaz:psim.ips #3	net_set	0.25	5.58
topaz:psim.ips #3	pmalloc	0.24	5.36
topaz:psim.ips #0	putpacket	0.23	5.13
topaz:psim.ips #2	getpacket	0.22	4.91
topaz:psim.ips #1	getpacket	0.20	4.46
topaz:psim.ips #1	putpacket	0.16	3.57

Figure 12.
Critical Path table for phase B of initial version of network simulator.

Figure 12 shows the procedure-level Critical Path table for phase ‘‘B’’. The table indicates that the procedures *getpacket* and *putpacket* account for more than 50% of the Critical Path during the phase. When we examined the code, we discovered that *getpacket* and *putpacket* were small routines that moved packets from buffer to buffer in the network. To improve this phase we converted both *getpacket* and *putpacket* to inline macros. This

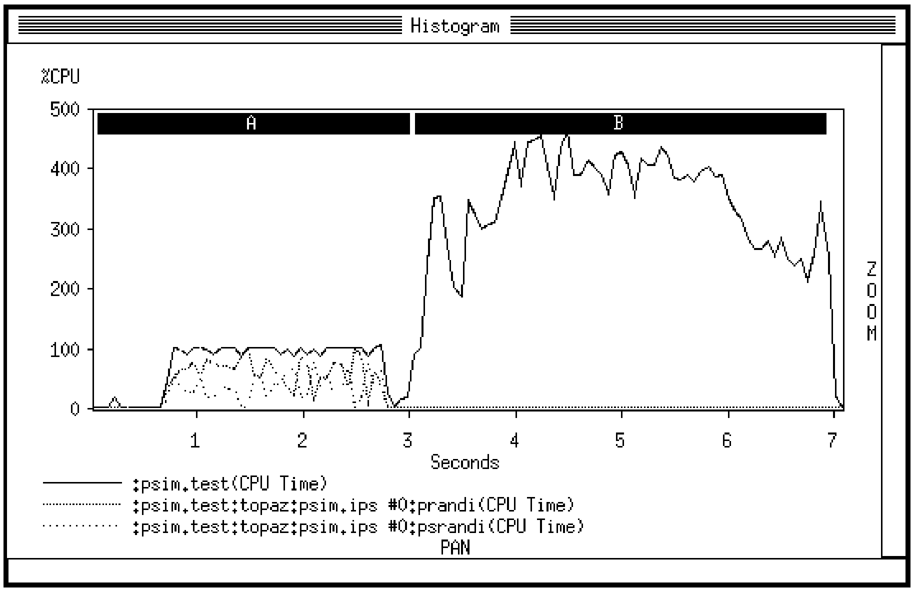
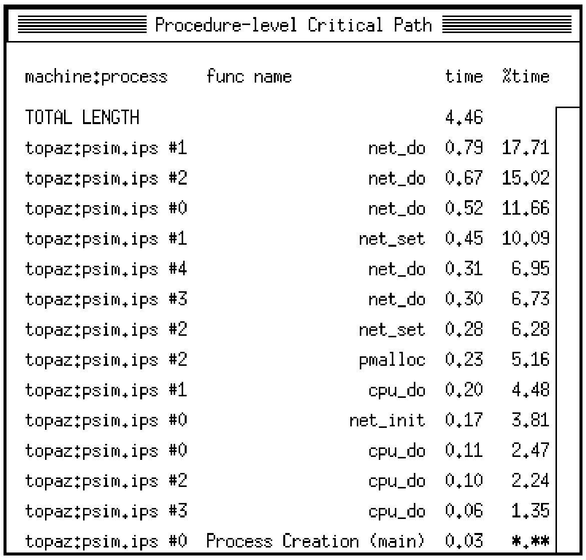


Figure 10.
Histogram of the initial version of the network simulator.

change improved the running time of all the test cases by approximately 11% (shown as the intermediate version in Figure 13).



Procedure-level Critical Path

machine:process	func name	time	%time
TOTAL LENGTH		4.46	
topaz:psim.ips #1	net_do	0.79	17.71
topaz:psim.ips #2	net_do	0.67	15.02
topaz:psim.ips #0	net_do	0.52	11.66
topaz:psim.ips #1	net_set	0.45	10.09
topaz:psim.ips #4	net_do	0.31	6.95
topaz:psim.ips #3	net_do	0.30	6.73
topaz:psim.ips #2	net_set	0.28	6.28
topaz:psim.ips #2	pmalloc	0.23	5.16
topaz:psim.ips #1	cpu_do	0.20	4.48
topaz:psim.ips #0	net_init	0.17	3.81
topaz:psim.ips #0	cpu_do	0.11	2.47
topaz:psim.ips #2	cpu_do	0.10	2.24
topaz:psim.ips #3	cpu_do	0.06	1.35
topaz:psim.ips #0	Process Creation (main)	0.03	*.**

Figure 14.
Critical Path table for improved version of network simulator.

Figure 14 presents the procedure-level Critical Path table for the improved *psim*. The table indicates that the procedures *net_do* and *net_set* are now responsible for most of the Critical Path. We discovered that these procedures did a lot of array indexing, and contained some redundant tests in conditional statements. We improved the procedures by adding pointer variables and by rearranging the evaluation order in the conditional statements. The improvements accounted for an additional 12% improvement in execution time for our test cases.

Version	Test 1	% Change	Test 2	% Change
base	74.7		375.2	
intermed.	65.8	11.9	335.1	10.7
final	57.7	22.7	287.4	23.4

Figure 13.
Table of improvements to network simulator (time is in seconds).

Figure 13 shows the results of running the three versions of *psim* on large test cases. Test Case #1 is a 128 processor/memory network, and Test Case #2 is a 512 processor/memory network. Both test cases simulated 4096 memory requests from each processor with stride of 1. We ran all of the tests on an unloaded, 20 processor Sequent Symmetry S81.

Figure 15 shows a histogram of the final version of *psim* running on a small (32 processor/memory) test case. The display shows that both phase ‘‘A’’ and phase ‘‘B’’ are now considerably shorter than in the original version of the program (Figure 10). Figure 15 also shows the program’s Barrier Time and Productive Bus Utilization curves. These curves reveal that the program spends a lot of time waiting at barriers and as a result causes additional bus traffic (though bus traffic does not in any way limit performance). The barrier waiting is especially noticeable at

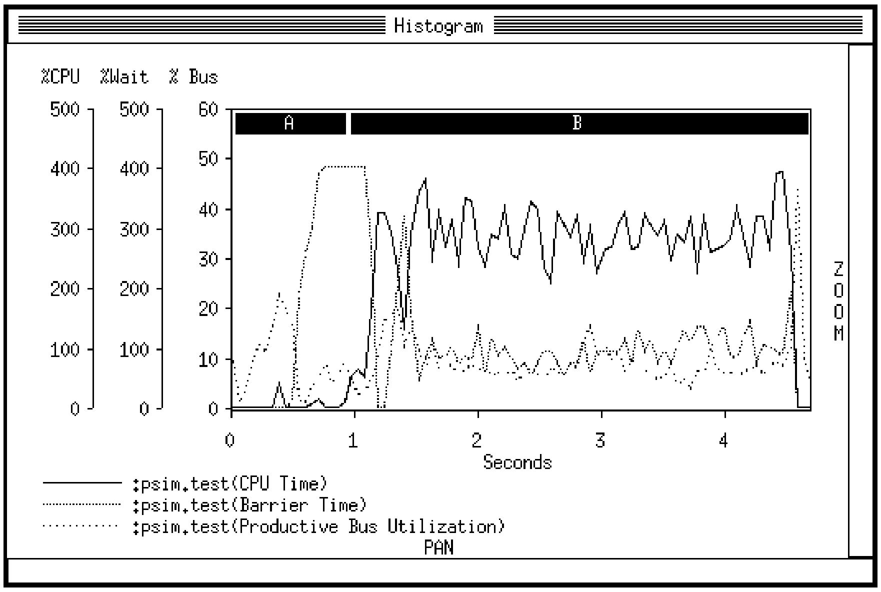


Figure 15.
Histogram of network simulator after performance tuning.

the beginning and end of large network simulations when one of the sub-networks has no memory requests to simulate. We have tried to reduce the barrier waiting time by dynamically allocating processors to the network, but the increased synchronization and bus traffic necessary for the dynamic allocation outweighed the benefit of load balancing.

6.3. Commentary

As a result of using our own tools on real applications, we learned when the tools excel, and equally important discovered a few weaknesses in the current version.

The ability to combine metrics collected from the hardware and kernel was useful in explaining variations in the CPU utilization of applications. Without the external information, it would have been difficult to determine where to look for these performance problems. Prior to the integration of external data, we could not determine if CPU time variations were due to system resource utilization (e.g. page faulting) or interference with other processes on the system. Even when we knew that a problem was caused by our application, it was almost impossible to identify which system resource was being taxed.

These interactions between the operating system and an application program can have significant effects on its performance. For example, we discovered that certain data sets for the shared-memory join application cause the program to run longer than expected because some of the processes get swapped out. The author of the program had also discovered this fact, but because IPS-2 was not available to her, she was forced to hand instrument her program with calls to the *getrusage()* system call. What took her several day to discover, IPS-2 was able to show in a matter of minutes.

Derived metrics provided a new and helpful addition to the system. However, many times during our measurement sessions we wanted to define new metrics to get a better idea about how a group of procedures or metrics related. We discovered that our approach to derived metric definition is not satisfactory. Currently derived metrics must be defined before the start of the measurement session. However, since the desired metric is often discovered during a measurement session, we had to iterate through metric definition and measurement sessions. Derived metrics would be more useful if they could be defined interactively during a measurement session rather than statically before it starts. Extending our tool to permit this is not a difficult problem, and we are currently implementing this feature.

Adding external histograms to IPS-2 caused a few problems. The user can be overwhelmed by the large number of metrics to choose. As a first step in solving this problem, the metric selection menus are sorted alphabetically. A more general solution would be to have IPS-2 help the user to select relevant metrics.

7. Conclusions

To provide a complete picture of a program's execution, we must include performance data from the program itself, the operating system and hardware on which it executes, and other sources, such as the network. By integrating performance data from all of these sources into a single tool, we simplify the programmer's task of understanding their program. To easily integrate performance data from all of these sources, we must provide an open interface using a well-defined, common data format. The time histogram appears to be a suitable format that is simple to generate, reasonably compact, and supportive of a wide variety of data types. IPS-2's existing facilities combined with the new external and derived metrics allow us to perform analyses that were not previously possible.

The external histogram interface allows IPS-2 to easily adapt to new operating systems and architectures. By keeping the system-specific collection facilities external, we are able to support the tools on radically different systems without major changes to the core of IPS-2.

We presented two examples of applying IPS-2 to improve programs. Both programs were real and written by other people to solve problems, not to demonstrate our tool. For the first program, a database join utility, we reduced the running time by 29%. For a second program, an interconnection network simulator, we reduced the running time by 23%.

Our current work includes using external histograms to monitor specific constructs within the operating system kernel, as a move toward full kernel tracing. We are also implementing interactive, user-defined metric definitions.

8. Acknowledgments

We are grateful to Neal Wyse of Sequent Computer Systems for his help with the initial design of the external data interface and for providing us with the Hardware EDCU. We wish to thank Joann Ordille for her shared-memory parallel join program used in Section 6.1, to Shreekant Thakkar for providing the network simulator program used in Section 6.2, and to the people at Sequent Computer Systems for helping with our many questions.

9. References

1. B. P. Miller, M. Clark, J. Hollingsworth, S. Kierstead, S. Lim and T. Torzewski, "IPS-2: The Second Generation of a Parallel Program Measurement System", *IEEE Transactions on Parallel and Distributed Systems 1*, 2 (April 1990), pp. 206-217.

2. J. Hollingsworth, B. P. Miller and R. B. Irvin, "IPS User's Guide", *Computer Sciences Technical Report*, December 1989.

3. T. E. Anderson and E. D. Lazowska, "Quartz: A Tool for Tuning Parallel Program Performance", *Proc. of the 1990 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Boston, May 1990, pp. 115-125.

4. C. Yang and B. P. Miller, "Critical Path Analysis for the Execution of Parallel and Distributed Programs", *8th Int'l Conf. on Distributed Computing Systems*, San Jose, Calif., June 1988, pp. 366-375.

5. S. L. Graham, P. B. Kessler and M. K. McKusick, "gprof: a Call Graph Execution Profiler", *Proc. of the 1982* SIGPLAN *Symposium on Compiler Construction*, Boston, June 1982, pp. 120-126.

6. *Symmetry Technical Summary*, Sequent Computer Systems, Inc., 1988.

7. D. DeWitt and R. Gerber, "Multiprocessor Hash-Based Join Algorithms", *Proc. of the 1985 VLDB Conference*, Stockholm, Sweden, August 1985, pp. 151-164.

8. B. Bitton, D. DeWitt and C. Turbyfill, "Benchmarking Database Systems- A Systematic Approach", *Proc. of the 1983 VLDB Conference*, Florance, Italy, October 1983, pp. 8-19.

9. E. D. Brooks III, "The indirect *k*-ary *n*-cube for a vector processing environment", *Parallel Computing 6*, 3 (1988), pp. 339-348.

10. S. S. Thakkar, Performance of Parallel Applications on a Shared-Memory Multiprocessor System, in *Performance Instrumentation and Visualization*, M. Simmons and R. Koskela (ed.), ACM Press, 1990, 233-256.