Performance Data Gathering
for the Origin2000

Uroš Prestor
April 19, 1999

Abstract

There is no performance visualization tool for CC-NUMA architectures in general and the Origin2000 system in particular. This paper briefly discusses the desired features of such a visualization system, such as low-overhead trace gathering, top-down approach to performance data presentation, scalability and extensibility. Designing and implementing a complete system consists of two parts: a low-overhead trace based data gathering infrastructure and visualization/analysis back end. The scope of this thesis is limited to developing the data gathering infrastructure. A design of a distributed data gathering system is presented, followed by a discussion of each component. The thesis work will focus on minimizing the impact of data gathering on the application performance while scaling with system size. Some general features of the CC-NUMA architecture will be investigated; however, the work will be focused on Origin2000 systems.

1 Introduction

In recent years, parallel computer architectures have become ubiquitous. On the desktop, small SMP systems are becoming commonplace; larger SMP systems are a staple for departmental servers while yet larger MPP systems do the number crunching for grand challenge applications.

While the hardware is there, programming such systems to deliver even a fraction of the theoretical peak performance still remains an elusive goal. The increasing complexity of high performance systems requires the use of tools which help optimize program performance. Simple program profilers [12, 6] have been around for a long time, and the advent of massively parallel distributed memory systems in the eighties brought about first forays into performance visualization for large-scale parallel systems. [3] By the early nineties, parallel performance tool developers recognized a need for a general top-down presentation of performance data which replaced a mind-boggling selection of different visualization displays, and a need for scalability in large-scale visualization systems. [13] At the same time, parallel computer architecture designs started to converge towards a distributed system consisting of a number of processing elements (a
node with one or more CPUs, some local memory and a communication assist),
linked together by a fast interconnect medium. [4]

Silicon Graphics' Origin2000 system is an example of a successful realization
of the cache-coherent non-uniform memory access (CC-NUMA) architecture.
Currently, it scales from a two processor Origin200 desktop system up to a 128-
processor Origin2000 supercomputer. Future systems will expand the number
of processors in a single system up to 512 processors, while even today it's possible
to link several standalone systems together with a fast interconnect. In such
clustered environment, shared memory programming is possible within a single
system; message passing is needed to harness the power of multiple standalone
systems.

Despite the popularity of Origin systems in the supercomputing community,
there is a lack of tools designed to help develop large-scale parallel programs.
SGI's development environment includes a few rudimentary tools which use
hardware performance data to help detect bottlenecks in the program [17, 16],
but they require intimate knowledge both of the system architecture and its
implications for the design of parallel programs. In addition, there are other
sources of performance data which aren't used at all. Similar projects exist for
other processor architectures [2]. On the other hand, currently active research
projects in academia [1, 5, 9] strive for portability and support for heterogeneous
systems. This approach tends to discard the wealth of performance data sources
available on modern systems, or uses them in a primitive manner.

This proposal presents a case for a new performance visualization system
designed specifically for the Origin architecture. By limiting the scope to a single
architecture, it should be possible not only to capture all low-level performance
data but also combine data from multiple sources with detailed knowledge about
the architecture to detect performance bottlenecks. Such a system should be
able to capture data from all relevant sources while incurring minimal overhead
on the application. It should be flexible enough to support advanced data
capturing and data analysis techniques [10, 8]. Finally, it should be scalable up
to the maximum size Origin system.

The goal of this work is to design and implement a low-overhead, trace-
based scalable performance data gathering system. The evaluation work will
investigate the total overhead and impact on the application and the trade-
offs in a CC-NUMA environment. A rudimentary visualization system will be
developed to provide some insight into the data collected and to provide a basis
for future development of the system.

The rest of the proposal is organized as follows: Section 2 presents native
performance data sources on the Origin system and discusses vendor-supplied
tools. Section 3 presents major software modules in thesis implementation work
and Section 4 presents approximate work schedule. Finally, Section 5 discusses
several related research projects.
2 Background

Various building blocks of the Origin system contain performance monitoring features: the R10000 processor can count various processor events using two hardware counters [15]; the Hub ASIC defines six sets of performance monitoring classes where each class consists of six different events counted by 20-bit counters; and the Spider chip maintains four link utilization counters for each of its six communication ports.

The IRIX operating system has extensive support for R10K performance counters (virtual 64-bit counters and counter multiplexing across 32 events, where the counters are preserved across a context switch). There is an interface to get to the Hub counters (revised and updated in IRIX 6.5.2) and a set of ioctl commands which retrieve Spider link utilization counts together with other link statistics (error counters etc.).

There is also extensive support for NUMA memory reference counters: hardware counters for each 4K local memory block combined with the OS support which—besides providing access to raw hardware counters—virtualizes 12 bit counters for each block to 32 bits.

2.1 Existing SGI Tools

Despite the abundance of performance data, existing SGI tools all but ignore it; processor counters remain the only well supported source of performance data. The perfex [17] program is used to run an application and print aggregate event counts upon program termination. Besides the display of raw numbers of events, perfex can also compute some performance metrics, derived from basic event counts (e.g. CPI and MFLOPS rate, cache and memory bandwidth, cache line reuse etc.) SpeedShop [16] can be used to profile an application and then use PC samples based on performance counter overflows to detect which source lines or instructions are causing stalls. This analysis is inaccurate because of the out-of-order nature of instruction execution in the R10K processor. Both tools are thread-aware; they distinguish between different threads in a share group and print data for each thread separately. However, they are not parallel tools per se.

Another tool which can be used to monitor system performance is the Performance CoPilot (PCP). This tool defines a large set of performance metrics, which are gathered mostly from kernel-maintained statistic counters. PCP includes support for R10K counters in system mode and has access to Hub and Spider counters. However, PCP is a system analysis tool—it has no notion of individual application performance; rather, its view reflects the performance of the system as a whole. Besides this, the useful sampling granularity is quite low: due to its design, it’s not possible to do sampling with a fine-grained intervals (e.g. 100 milliseconds) without imposing a severe burden on the system. PCP cannot be used to track application performance; however, its suite of visualization tools is quite impressive.
Last year, SGI's efforts to create a new performance analysis tool (code-named Caribou) were abandoned. The existing work was to be integrated with WorkShop, SGI's code development environment. However, the release of WorkShop shipping with IRIX 6.5.1 has a very limited support for performance evaluation; it's possible to instrument code with pixie and display static function profiles, trace system calls and I/O activity and look for page faults and TLB misses. As with perfex and SpeedShop, the support for massively parallel codes is severely limited.

3 Proposed Design

Performance visualization systems can be decoupled into two independent parts: the data collector and a suite of visualization and data analysis tools. The objective of this thesis is to design and implement a scalable trace-based data collector for Origin systems. Data visualization and analysis is not the focus of this work; however, a rudimentary visualization back-end will be provided, serving both to verify data collection and as a basis for future work.

3.1 Data Collector

The data collection module gathers application traces from various sources and either stores them on a file or sends them over a socket to the receiving application. Traces stored in a file can be used in a post mortem analysis. Applications receiving traces from a socket can provide interactive feedback to the user, perform an automated bottleneck search, etc. No special instrumentation should be necessary to gather application traces. The only requirement the application should satisfy is the presence of debugging information which makes translation from instruction addresses to source line numbers possible; even this can be avoided if the user is willing to give up source code browsing.

Figure 1 presents the building blocks of the data collector. The main components are: a loadable kernel module, an application wrapper and a data sink process. The kernel module collects data from the Origin hardware counters; the application wrapper tracks sproc and fork calls, which is necessary to support a generic distributed programming model (multiple multithreaded processes running MPI, for instance). The data sink process collects application traces, controls the kernel module, drains its buffers and gathers other kernel-generated data (e.g. scheduler activity available through /dev/par interface). User-level communication between the data sink process and the application/library wrappers goes through a shared memory segment, the shmem arena, which holds configuration data, event queues and other data structures needed for user-level trace gathering. One of the challenges of this work is to find algorithms which minimize communication overhead in the CC-NUMA environment.

The data collector should support a means of interactive control. For example, a user interactively tracing application performance would like to focus on a limited set of events which interest him—instead of the full trace, the collector
should be able to dynamically reconfigure itself while the application is running. Such dynamic reconfiguration is useful for other purposes, such as for automatic bottleneck search.

The data collector should impose low overhead on the system and exhibit as little interference with the application as possible. The collector should scale to the maximum size of the Origin system: it should support cases where most of the machine is occupied by a single application being profiled. It should be noted that even without profiling, it is not reasonable to oversubscribe the machine, i.e., spawn more threads than there are processors in the system or allocate more memory than is physically present; even when there is a single computing task in the system, a portion of the machine needs to be set aside for interactive users and system daemons. To assess the impact of profiling on the system, it is necessary to evaluate two cases of overhead: first, the overhead of data gathering on compute nodes, which comes from application/library wrappers and from R10K interrupts; second, an estimate is needed of how many nodes need to be dedicated for the data sink process, given different amounts of trace data gathered from the application.

3.1.1 Application Wrapper

The application wrapper’s function is twofold: first, it needs to set up a communication path with the data sink; second, it needs to set up hooks which let us trace the application as it creates new threads and child processes. In order to access data and control the application being profiled, it is necessary to map a dynamic shared object (DSO) into the process’ address space.

The mapping of an arbitrary DSO into a user process is possible by changing the behavior of rld, the IRIX dynamic linker. Whenever a file is executed via the exec system call, the operating system first invokes rld. The dynamic
linker then maps the executable code into user space and determines the shared libraries the code depends on. Before mapping the required dynamic libraries, \texttt{ld} first checks for the existence of the \_RLD\_LIST environment variable; if it exists, \texttt{ld} maps all DSOs from \_RLD\_LIST in addition to the list of required shared libraries. Therefore, in order to insert the application wrapper when a file is executed, all that’s needed is a simple manipulation of \_RLD\_LIST.

A useful feature of the data collector is to be able to dynamically attach to a program in the middle of the execution. Dynamic linker trickery doesn’t work in this case; however, it’s possible to temporarily suspend process execution and modify its address space by means of the /proc (debug) interface. This is sufficient to insert bootstrap code which manually \texttt{dlopen}s the application wrapper DSO and executes it.

The primary function of the application wrapper is to set up a communication path with the data sink process. Trace events generated by the application are stored in a section of shared memory accessible to both processes. This communication path will initially be used to transfer generic application events (e.g., start and stop times) as well as event traces generated by library wrappers (see Section 3.1.4). However, the transport mechanism should be flexible enough to support high-volume traffic generated by fine-grain trace mechanisms (e.g., source line statistics generated by Portland Group’s HPF compiler [11]).

The application wrapper needs to detect when the application creates a new thread or a child process. When such an event occurs, the application wrapper needs to attach itself to the thread or child process and set up tracing for it. IRIX provides two interception mechanisms to detect such cases—\texttt{atfork} and \texttt{atsproc} function families. The \texttt{ld} mechanism described above is used to trace application execution across \texttt{exec} calls.

It should be possible to dynamically reconfigure the data collector while the application is running. This can be accomplished by registering a signal handler for one of the user-defined IRIX signals (\texttt{SIGUSR1} or \texttt{SIGUSR2}). With this method, dynamic reconfiguration would be accomplished by the data sink process: first, the parameters in shared memory arena are modified; then, signals are sent to all threads which need to be reconfigured.

### 3.1.2 Loadable Kernel Module

The primary function of the kernel module is to periodically collect thread execution traces and present them to the data sink process. Thread execution trace consists of a sequence of samples, each sample recording the values of processor, Hub and Spider counters in addition to other execution attributes such as CPU number on which the thread executes or I/O activity on the node.

There are various user level interfaces in IRIX which let us capture all the information described above; however, collecting data in this manner has several drawbacks. Each source of data is accessed via a different system call, adding unwanted overhead. These system calls don’t directly access the hardware counters; rather, they return an in-core cached copy which is occasionally updated. The Hub counters are updated every clock tick (10ms) and Spider port
utilization counters are updated every two seconds; such long update intervals render this data useless. Therefore, the kernel module will bypass the existing kernel infrastructure to add direct support for the Hub and Spider counters; it will extend the existing support for the R10K counters and provide services to the data sink process which would be impossible or too time consuming to implement in user space.

Implementing a new infrastructure for accessing Hub and Spider counters is trivial—instead of relying on cached copies counter values, each read of counter values translates into a direct hardware read; the in-core copy of the counter is updated immediately afterwards. (Memory copies are needed because hardware counters have limited precision—20 bits for the Hub counters and 16 bits for link utilization counters, making overflow intervals very short.)

In contrast to the Hub and Spider counters, IRIX provides extensive support for the R10K performance counters. The kernel can multiplex 32 events on two physical counters; it keeps all counter values in a per-thread data structure, preserving them across a context switch; the kernel can also deliver a signal to a process when one of the counters overflows. The programming effort needed to duplicate the R10K counter infrastructure is not trivial; in addition to the time needed to write the code, due to tight integration with other parts of the kernel, this effort would require access to IRIX source code. Therefore, instead of developing a new R10K performance counter infrastructure it seems better to extend it. Fortunately, there are hooks in the existing interface which allow us to insert extra functionality into the kernel. The net result is that the kernel module will be able to leverage off the existing infrastructure and extend it with new functionality. The most important capability implemented here will be the ability to collect R10K counter traces in kernel as opposed to bouncing off into user space, which minimizes the overhead.

Last but not least, a presence in the kernel space allows for simple, low-overhead implementation of basic services such as timestamping. Each Hub ASIC has a free-running counter with 800ns resolution. These counters are kept in sync across an entire Origin system by means of a scheme resembling hardware phase-locked loop. This scheme makes sure that the counters tick at the same rate; however, actual counter values differ between Hubs. The kernel module is a convenient place to implement a counter synchronization scheme.

3.1.3 Scheduler Activity

The performance metrics collected during an application run can be roughly divided into two groups, thread- and system-oriented. The first group includes R10K counters and the library wrapper data; Hub and Spider counters fall into the second group. In order to find correlation between metrics in both groups, it is necessary to obtain the scheduling information. Given a thread, we need to know on which processor it was executing at some point in time; and, given a node, we need to know which threads were executing on the node CPUs.

One way of tracking this information is to use performance counter overflow interrupts—each interrupt is handled by the processor on which the thread was
executing; the interrupt handler stores processor id together with other data. Another way is to use the IRIX pseudo device /dev/par. Pseudo devices are kernel drivers without any associated hardware; for this reason, they are sometimes called software drivers. In essence, a pseudo device is a communication channel between a section of the kernel and a user-level process. Pseudo device /dev/par can be used to obtain scheduler activity for a set of related processes or for an entire system.

The most useful feature of /dev/par is the possibility to trace all scheduler activity. An event is generated whenever a thread state changes from blocked to ready, when the thread is scheduled on a processor and when a thread is descheduled. This event trace makes it possible to track thread migration and scheduling. The ability to get this kind of information is important because it’s not easy to predict the behavior of the scheduler.

A potential problem with this source of information is the lack of documentation. IRIX man pages give information only for the command-line interface; the API is not discussed. It would be possible to reverse-engineer the API from the header files but this approach would take some time; the preferred way would be to obtain source code for par, padc and rmon commands from Silicon Graphics.

### 3.1.4 Library Wrappers

An essential feature of the performance visualization system is to be able to determine how much time was spent doing synchronization or communication. Library wrappers can be used to generate trace events whenever a thread calls a library function and/or when the call is complete. In this way, wrapping synchronization primitives in system libraries and MPI calls provides the necessary information.

The idea of library wrappers is based on an old hacker trick which was used to exploit the telnetd environment vulnerability. The trick used the Telnet protocol to change the environment variables which control the dynamic linker, in the same way as the application wrapper does it. Once the rogue library is loaded, it catches references to a widely used library function (e.g., read). Before calling the real function in the standard library, rogue code checks whether the caller has superuser privileges; if this is the case the system is compromised.

The same idea can be used to generate library function call traces. For each function to be “wrapped,” a call to the real library function is replaced by a call to a stub which generates an event before and after calling the library function. For example, a wrapped call to read would be replaced by the following stub:

```c
size_t read(int fd, char* buf, size_t bufsiz)
{
    event(READ_START);
    (*real_read)(fd, buf, bufsiz);
    event(READ_FINISH);
}
```
In the code fragment above, `real_read` is a pointer to the `read` function in the standard C library, obtained at initialization time by consulting `rld`.

It should be obvious that a simple wrapper shown above is just the first step. More elaborate stubs could include call-specific data in its trace output or limit the output to objects of interest (e.g., a wrapper for barriers would generate trace data just for selected barrier instances). Such library wrappers can be made arbitrarily flexible and configurable.

Together with the ability to trace system calls through `/dev/par`, library wrappers can effectively determine the behavior of user-level libraries. For example, a barrier implementation in system library might want to spin for a certain amount of time before blocking the thread and letting another thread run; library wrapper would generate an event when the barrier function was called while an event on `/dev/par` would be generated when barrier function decides to block the thread (and when the thread is resumed later on).

### 3.1.5 Data Sink Process

The data sink process serves as a focal point in the data gathering infrastructure. It collects data from various input sources, does some initial postprocessing (such as collating samples according to timestamps) and stores data stream on a file or a socket. On the interactive side, the data sink process is the sole entity which controls the rest of data gathering infrastructure. That is, a command stream for interactive control needs to attach to data sink process only.

The alternative to a separate data sink process is to have data stored using signal handlers in the application process; having a separate process has distinct advantages. Additional postprocessing can be performed with minimal interference with the application. A process can be tied for execution on a particular node/CPU. In a space-shared system, one could have the application execute on a dedicated collection of batch nodes while the data collector would run on CPUs dedicated to interactive processes; the impact on caches would be minimized. New data sources can be incorporated easily. On systems linked in a cluster, data sink processes can be coupled together to provide cluster-wide scaling. Last but not least, a separate process simplifies modular programming.

The data sink process needs to scale from a two-node Origin200 to 64-node Origin2000. In order to accomplish this, and in order to implement other functionality (e.g., interactive control), the data sink needs to be multithreaded. It also needs to be flexible: one should be able to limit the execution to a set of nodes/CPSUs, to specify which data sources to monitor etc. Providing a flexible interface to the data sink is the key to making the entire infrastructure flexible.

### 3.2 Visualization

Collecting all the performance data in a system is a closed problem; there are only so many sources that one needs to tap into. The big challenge is presenting the data in a meaningful way. Visualizing the interactions between various parts of a parallel program is challenging; however, this is not the end of the story.
Having performance data, one can imagine numerous ways to use it: parallel efficiency evaluation and automated bottleneck search, feedback for optimizing compilers, not to mention the evaluation of the architecture on a scale not possible with hardware simulators. With the addition of interactivity, a whole new area of program steering and interactive performance monitoring opens up.

As already mentioned, the goal of this thesis is the development of a solid performance data gathering framework. However, performance data by itself is useless and its simple ASCII dumps bewildering. Therefore, a means of presenting the data needs to be provided, both to gain some insight and enable validation of the data gathered, and as a basis for future development. The goal of my thesis work is to develop basic infrastructure for visualization and analysis.

A graphical user interface, written in Tcl/Tk, will consist of two widgets presenting two basic functions required by an analysis tool: a timeline display and a source code browser. They should not be viewed as complete applications; rather, they should serve as points of departure, enabling the development of more sophisticated functionality by providing low-level infrastructure.

The TIMELINE widget will serve as the main window of the graphical user interface. It will allow the user to zoom in the execution time intervals and to inspect individual sample points. There are at least two views the TIMELINE widget should support: synchronized thread- and node-oriented views of the execution. The former groups together performance metrics pertaining to a single thread of execution, such as processor performance counters (and its derived metrics such as CPI), whereas the latter presents performance data from the standpoint of hardware resources comprising a node (memory references from both CPUs, I/O activity, link utilization etc.) Additional views can be imagined, such as display of the entire system in the manner of PCP's overview. This widget embodies the concept of navigation in time.

The SOURCE BROWSER widget provides mapping from individual sample points to source code where the sample was taken. It should be viewed as an extension of the TIMELINE widget: when a sample is selected in a thread-oriented view, the PC address of the interrupted instruction is translated back to the source line number and presented in a read-only window. The implementation of this widget will serve as an investigation into problems of reverse-mapping PC address; it is quite possible that this mapping will be rendered less useful because of the transformations performed by the optimizing compiler.

The widgets described above are just a beginning. Additional features in the visualization system include support for interactivity, a VCR-style replay controls and interactive displays in the manner of ParaGraph.

4 Plan of Attack

Except for implementation details, the design of the performance data gathering system is finished. The implementation work has already begun. I propose three major milestones in the implementation schedule, breaking the implementation
work into three phases. The first two phases shall complete by the end of May; by that time, I should have a working data gathering system with basic visualization capabilities. The third phase shall be completed during the summer internship at SGI; the work in this phase will focus on system performance and overhead measurement.

**Phase 1** The goal in first phase is to “close the loop.” At the end of this phase, I should have a working code consisting of the application wrapper, a skeleton loadable kernel module (LKM) and data sink process storing traces on a file.

The work on the application wrapper and on the LKM is completed. The data sink process is halfway implemented; I expect to complete this phase by mid-May.

**Phase 2** The primary focus of the second phase is the work on the graphical user interface. In addition to GUI work, in this phase the LKM will be extended to support R10K counters and an infrastructure for the automatic generation of library wrappers will be implemented.

The work on GUI will be done in parallel with phase 3, the evaluation of data gathering overhead.

**Phase 3** Phase three will be focused on system evaluation and miscellaneous enhancements. In particular, I plan on implementing the `/dev/par` interface while at SGI because I will have access to internal documents. The work will be done while interning at Silicon Graphics; the tentative dates for the internship are from June 7 to August 27.

I plan on writing the thesis in parallel with the implementation work. I expect to have a draft version of the thesis ready by the end of August. The tentative date for thesis defense is early October.

5 **Related Work**

The tools for performance optimization have been around for a long time. On Unix systems, `prof` and its successor, `gprof`, [12, 6] used statistical PC sampling, combined with a compiler-generated profiling code to obtain flat program profiles. However, these tools aren’t useful when optimizing parallel programs.

SeeCube [3] was one of the first systems to take advantage of cheap graphics on workstations to show interactions in a parallel program. It was able to display dynamic representations of a hypercube topology and it had a flexible method of mapping performance metrics to color codes. It was, however, tightly coupled with a specific operating system and machine topology.

The second generation of performance visualization tools attempted to go beyond one platform and one interconnect topology. ParaGraph [7] used program traces, obtained with Portable Instrumented Communication Library (PICL), to show processor utilization, message traffic and user-defined task interactions
in a variety of displays. Its methods to visualize performance data varied from simple utilization profiles and histograms to Gantt charts and Kiviat diagrams. Sheer variety of diagrams and topologies in ParaGraph was a great help for the expert; it confused the typical programmer. This drawback was identified by researchers working on the current generation of performance visualization system. Besides the attempt at multilevel semantic correlation, bridging the gap between hardware and system software on the one side and the application on another, third generation systems identified three important goals: portability, scalability and extensibility. First attempts were made at automating the search for performance bottlenecks.

The Pablo toolkit [13] decoupled performance data semantics from its structural representation via a performance data metaformat. This enabled a construction of a coarse-grained dataflow model for graphical programming of performance analyses and visualizations. Its follow-on work, SvPablo [1], closed the circle by integrating source code browser with manual instrumentation and the display of performance metrics. Requested instrumentation points are passed to a preprocessor for C and C++ programs, which instruments source code with calls to data capture library; instrumented sources are passed to the compiler, traces are gathered while the application is running, then combined and presented in the source browser window. SvPablo supports Portland Group HPF compiler trace mechanism as well.

Paradyn performance instrumentation and visualization tool supports real-time insertion and removal of instrumentation code during program execution. A technique called W3 search [8] is used to automatically search for performance bottlenecks. The system uses real-time instrumentation to test hierarchical tree of progressively refined hypotheses describing possible causes for program bottlenecks.

Digital continuous profiling infrastructure [2] developed a system for low-overhead statistical gathering of PC samples across the entire system. These samples are used by a variety of postprocessing tools, from simple program profilers to sophisticated machine models which attempt to explain possible causes for instruction stalls. DCPI is not a parallel tool per se; it’s intended for computer architects and OS designers. However, the methods used to collect performance data are applicable in my thesis work.

References


