Automating the Large-Scale Collection and Analysis of Performance Data on Linux Clusters[1]

Philip Mucci, Jack Dongarra, Shirley Moore, Fengguang Song, and Felix Wolf

University of Tennessee-Knoxville
{mucci,dongarra,shirley,song,fwolf}@cs.utk.edu

Rick Kufrin

National Center for Supercomputing Applications
rkufrin@ncsa.uiuc.edu

Introduction

Many factors contribute to overall application performance in today's high-performance cluster computing environments. These factors include the memory sub-system, network hardware and software stack, compilers and libraries, and I/O sub-system. The large variability in hardware and software configurations present in clusters can cause application performance to also exhibit large variability on different platforms or on the same platform over time. Compute-intensive applications may perform well on an architecture with efficient utilization of CPU and single-processor memory, such as the Intel Xeon, while memory-intensive applications may perform well on an architecture with good scalability of the memory subsystem, such as the AMD Opteron node [6]. Even with a fixed hardware configuration, software factors can cause large variations in performance. Compilers that produce acceptable code on some platform configurations may produce suboptimal code on other platform variants. Some math libraries require hand tuning of various compiled-in parameters, and a library that is hand-tuned for one platform may perform poorly on a different variant of the same platform. Some libraries (e.g., BLAS, LAPACK) have standardized APIs that are shared across different implementations that can have considerable variations in performance. It can be difficult to predict which library variant will perform best on a particular platform without testing each variant on that platform. If an application is updated and/or ported to a platform originally not supported, the optimization flags in the application Makefile may be anachronistic or otherwise inappropriate and may need to be altered to achieve acceptable performance on new target platforms and platform variants.

[1] This work is supported by DOE SciDAC under grants CE-FC02-01ER25490 and CE-FG02-01ER25510 and by NSF PACI under grant NSF-ACI-9619019 subaward #790.

Since the first steps in performance tuning of an application are to use the best compiler flags and the best implementations of library code available for that platform, automated collection of performance data for benchmark and application codes that test these factors can help system administrators and application developers in making the correct default and application-specific selections, respectively.

Application developers may have a choice of cluster systems on which to run and may need performance data that characterizes their application, for example as compute, memory, or I/O intensive, in order to make the best choice. If performance data for benchmark codes with similar characteristics have already been collected, application developers can make an appropriate choice without having to optimize and run their code on all the available systems.

Once a particular system has been selected and the best compile flags and libraries determined, further performance improvement will involve hand tuning. Routine-level and/or loop-level profile data based on both timing and hardware counter metrics, including derived metrics such as instructions per cycle (IPC) and the floating point to memory operations ratio (F:M), can help point to areas of the code that will benefit from particular hand-tuning techniques (e.g., outer loop unrolling for nested loops with low F:M ratio). However, the majority of application scientists do not have the time or inclination to make extensive changes to their source code in order to collect performance data. Furthermore, analyzing large amounts of profile data can be a daunting task, and pinpointing specific performance problems that will benefit most from hand tuning can be like looking for a needle in a haystack. Determining the cause of a performance problem and how to fix it often requires specialized knowledge of the architecture and its interaction with the compiler and runtime system. Automated analysis of performance data can help reduce the dimensionality of the performance metric space, identify points in the space that indicate performance problems, and map those points onto locations in the source code.

The remainder of this paper describes the following tools that address the above issues:

?? the PerfSuite collection of easy-to-use tools, utilities, and libraries for performance analysis on Linux clusters

?? the Dynaprof tool for inserting performance measurement instrumentation directly into a running application's address space at run time, and

?? the CUBE display tool for interactive exploration of a multidimensional performance space based on a processor-node-cluster hierarchy

PerfSuite

PerfSuite is a collection of tools, utilities, and libraries for software performance analysis where the primary design goals are ease of use, comprehensibility, interoperability, robustness, and simplicity. PerfSuite development was motivated primarily by the lack of available, reliable, low- or no-cost software suitable for use on Linux clusters in a production environment with the widest possible variety of application software. The user community of the NSF supercomputer center program conducts research across a wide variety of scientific and engineering domains and the production codes used on these resources are no longer limited to the traditional Fortran-77 style shared-memory-parallel application code. It is common to see languages such as

C, C++ and Java as the primary languages used in computationally-intensive simulations, so performance analysis support for applications written in these languages is essential for general-purpose use.

At the National Center for Supercomputing Applications (NCSA), a transition from shared-memory multiprocessors such as the SGI Origin 2000 array, which had been the mainstay of production computing resources in the second half of the 1990's, to Intel IA32-based and (later) IA64-based Linux clusters equipped with Myrinet interconnects represented a major shift in the computing paradigm. This shift affected not only the way that researchers and application developers designed and implemented their software, but in addition had substantial implications with respect to the supporting software available for sophisticated performance analysis. Like many vendors of HPC systems, the IRIX operating system used on the Origin offers a wealth of excellent GUI-based performance analysis tools to the user provided in the SpeedShop package. Additionally, a simple yet powerful command line utility "perfex" and a high-level API are provided for users who wish to collect detailed performance-related data with minimal effort [28]. In practice, it was noted that a substantial majority of users tended to prefer the simpler interfaces to performance analysis in their development efforts and frequently found that the basic yet accurate data provided by the simpler tools were sufficient for their analysis needs.

With the arrival of the IA32 and IA64 Linux clusters at NCSA in 2002, the lack of any similar capabilities was immediately apparent. Fortunately, the then-current release of the PAPI library [19] and the underlying performance counter drivers that it uses provided the low-level access to the data and functionality required to support similar tools on both architectures and so design and development of what became PerfSuite began. It is important to note that the motivation for PerfSuite was and is not research-oriented, but rather was to address an immediate need of the general scientific and engineering communities - as a result, design decisions are heavily influenced by an assessment of the anticipated tradeoff between sophisticated capabilities that may be of long-term interest only to the computer science and tool developer communities versus simpler capabilities more likely to be of benefit in day-to-day use in the field by end users. We feel that the bias towards simplicity pays off in stability, ease-of-use, maintainability, comprehensibility, and learnability for the broadest possible audience. Prototype releases of PerfSuite software were made internally in late 2002 and began to see initial adoption by technical staff working directly with the NCSA user community shortly thereafter [11,12].

Figure 1 is a block diagram that shows the organization of a subset of PerfSuite software that addresses hardware performance event data. PerfSuite also provides additional support for other common tasks in performance analysis such as MPI communication statistics and compiler optimization interpretation that are not addressed in this paper. The software is designed to be independent of the underlying mechanism for accessing performance data and this is reflected in the insulation of the user-accessible components (the command-line utility psrun and supporting libraries) from external software support. While the current release supports PAPI (versions 2 and 3) software releases, it is possible to use an entirely different supporting library to access performance data and indeed an instance of this type of replacement already exists for access to statistical profiling support using the standard profil() routine in the GNU C (and similar) libraries. The existence of this alternate interface allows the

installation and use of PerfSuite on platforms where PAPI is not available and/or supported, albeit with a restricted set of functionality.

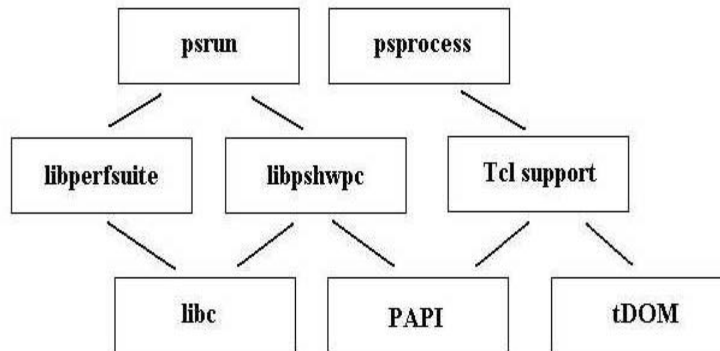

Fig. 1. PerfSuite structure

PerfSuite includes the following command-line utilities:

?? **psinv**: a utility that provides access to information about the characteristics of a machine (e.g., processor type, cache information, available performance counters)

?? **psprocess**: a utility that assists with a number of common tasks related to pre- and post-processing of performance measuremenets

?? **psrun**: a utility for hardware performance event counting and profiling of single-threaded, POSIX threads-based, and MPI applications. Performance counter multiplexing is supported. psrun requires no source code changes or relinking of the application.

psrun operates in one of two modes: "counting" mode or "profiling" mode. In counting mode, psrun reports overall performance information for the monitored program, while in profiling mode, psrun relates hardware performance event occurrences to the program's source code in much the same way as time-based profilers like gprof. Optionally, psrun can also monitor other resource usage of an application (e.g, maximum memory usage, faults, swaps, user/system time, exit information, etc). psrun makes use of the XML standard for data representation to enhance the flexibility and cross-platform compatibility of the data collected. By default, psrun uses the PAPI library for access to hardware performance counters [1]. After psrun has generated performance counter measurement file(s), the psprocess command-line utility can be used to convert the data into a text-based format that lists the hardware performance data as well as a number of metrics that can be derived from the raw counter values. The events that psrun monitors are specified by providing an event configuration file in the form of an XML document. Default configuration files are provided which select appropriate hardware events depending on the architecture.

In keeping with the philosophy of favoring simplicity over full-featured environments, each component of PerfSuite addresses one simple, well-defined task. For example, the psrun tool is the gateway to the most essential aspect of the performance analysis process: it is responsible for configuring the performance experiment as directed by the user, acquiring performance data for the application, and depositing the raw data in XML format to an output stream. It does not attempt to further process the data nor to present it in a form more suitable for human consumption. The psprocess tool's responsibility is to transform the raw performance data generated by psrun into a report that can be used by the performance analyst or application developer. The user need not explicitly specify the precise type of performance data contained in the XML documents presented to psprocess - the tool adapts itself based on the XML document type encountered. Currently supported XML document types include:

?? single process hardware performance counting measurements
?? multiple hardware performance counting measurements combined into a single
?? "multi-experiment" XML document
?? single-process statistical profiling measurements using either hardware
?? performance based data or time-based data through profil()

psprocess limits itself to directly generating text-based output, as we have found in practice that this is sufficient for the day-to-day needs of most users incorporating performance analysis in their development cycle. Another important influence on the choice of maintaining simplicity by limiting psprocess output to text-based reports is that there have been substantial advances in the development of easily accessible GUI libraries over the past decade. While in the past, the development of a graphical user interface may have required a sustained effort by experienced programmers fluent in toolkits such as Motif or Swing, today it is possible to rapidly develop a prototype GUI that is tailored precisely to the user's needs. Indeed, many production-quality GUIs are now developed entirely within the context of high-level scripting languages such as Perl, Python, and Tcl/Tk (psprocess itself is written in Tcl). We believe that by providing essential data collection functionality presented in a standard format (XML) for which a wide variety of parsers are readily available that we "open up" the data for the broadest possible audience of potential users and do not limit the use or presentation of the data in any pre-determined format or style. This approach also allows us to leverage existing work in GUI development; for example, psprocess can be directed through a command-line option to deposit the transformed XML statistical profiling data generated by psrun in a format compatible with the VProf toolkit developed by Curtis Janssen of Sandia National Labs [9]. VProf's graphical interface (based on the Trolltech Qt library) has shown itself to be a convenient and accessible tool for exploring performance data within communities such as those using NCSA resources. Figure 2 shows a screenshot of a VProf display of profiling data collected using psrun.

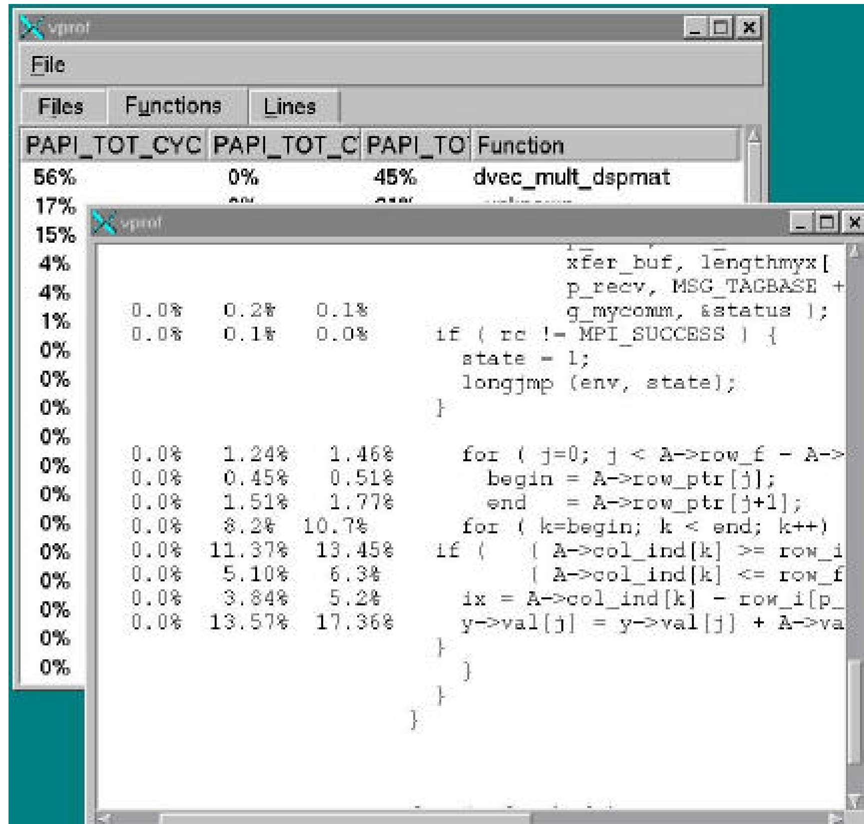

Fig. 2. Vprof display of profile data collected using psrun

Here are the steps required for a user to conduct a performance experiment of an MPI-based application using PerfSuite. If the goal is to perform aggregate performance measurement from application start to completion, it's sufficient to enter at the command-line or in a batch script:

```
$ mpirun -np 8 psrun pcg
$ psprocess psrun.PID.xml
```

This example uses an application named "pcg" using 8 MPI tasks. Each of the tasks will write out an XML document named (by default) "psrun.PID.xml", where PID is replaced by the process ID of the application instance. For an 8-processor run, this will result in eight separate XML files. The user may then examine each of these files individually by post-processing with psprocess as shown above. In this case, performance information will be displayed as shown in Figure 3:

```
PerfSuite Hardware Performance Summary Report

Version                        :  1.0
Created                        :  Thu Apr 01 10:43:50 AM CST 2004
Generator                      :  psprocess 0.2
XML Source                     :  pcg-8p-2ppn.20035.xml

Execution Information
==============================================================
Date                           :  Wed Mar 31 18:28:52 2004
Host                           :  cn003
User                           :  rkufrin

Processor and System Information
==============================================================
Node CPUs                      :  2
Vendor                         :  Intel
Family                         :  Pentium Pro (P6)
CPU Revision                   :  6
Clock (MHz)                    :  997.001
Memory (MB)                    :  1510.82
Pagesize (KB)                  :  4

Cache Information
==============================================================
Cache levels                   :  2
--------------------------------
Level 1
Type                           :  instruction
Size (KB)                      :  16
Linesize (B)                   :  32
Assoc                          :  4
Type                           :  data
Size (KB)                      :  16
Linesize (B)                   :  32
Assoc                          :  4
--------------------------------
Level 2
Type                           :  unified
Size (KB)                      :  256
Linesize (B)                   :  32
Assoc                          :  8
```

Fig. 3. Text-based performance output from a single process (counting mode)

```
Index Description                                                              Counter Value
=========================================================================
  1 Conditional branch instructions.................................       17353508735
  2 Branch instructions.............................................       17344906363
  3 Conditional branch instructions mispredicted....................         322161562
  4 Conditional branch instructions taken...........................       12481985055
  5 Branch target address cache misses..............................         888816252
  6 Requests for exclusive access to clean cache line................         25498507
  7 Requests for cache line invalidation............................          28142160
  8 Requests for cache line intervention............................          17579637
  9 Requests for exclusive access to shared cache line..............          30047790
 10 Floating point multiply instructions............................         284143164
 11 Floating point divide instructions..............................           9123114
 12 Floating point instructions.....................................         573805313
 13 Hardware interrupts.............................................             71188
 14 Total cycles....................................................      136942513622
 15 Instructions issued.............................................      114255067874
 16 Instructions completed..........................................      102258408172
 17 Vector/SIMD instructions........................................                 0
 18 Level 1 data cache accesses.....................................       78079180353
 19 Level 1 data cache misses.......................................         510888983
 20 Level 1 instruction cache accesses..............................      134868207990
 21 Level 1 instruction cache misses................................          45070810
 22 Level 1 instruction cache reads.................................      134809529833
 23 Level 1 load misses.............................................         422414635
 24 Level 1 store misses............................................          28906112
 25 Level 1 cache misses............................................         503221275
 26 Level 2 data cache reads........................................         424398044
 27 Level 2 data cache writes.......................................          28811783
 28 Level 2 instruction cache accesses..............................          47087108
 29 Level 2 instruction cache reads.................................          44898190
 30 Level 2 cache misses............................................         358556969
 31 Cycles stalled on any resource..................................       62025866772
 32 Instruction translation lookaside buffer misses.................           2311907

Event Index
=========================================================================
  1: PAPI_BR_CN      2: PAPI_BR_INS     3: PAPI_BR_MSP     4: PAPI_BR_TKN
  5: PAPI_BTAC_M     6: PAPI_CA_CLN     7: PAPI_CA_INV     8: PAPI_CA_ITV
  9: PAPI_CA_SHR    10: PAPI_FML_INS   11: PAPI_FDV_INS   12: PAPI_FP_INS
 13: PAPI_HW_INT    14: PAPI_TOT_CY    15: PAPI_TOT_IIS   16: PAPI_TOT_INS
 17: PAPI_VEC_INS   18: PAPI_L1_DCA    19: PAPI_L1_DCM    20: PAPI_L1_ICA
 21: PAPI_L1_ICM    22: PAPI_L1_ICR    23: PAPI_L1_LDM    24: PAPI_L1_STM
 25: PAPI_L1_TCM    26: PAPI_L2_DCR    27: PAPI_L2_DCW    28: PAPI_L2_ICA
 29: PAPI_L2_ICR    30: PAPI_L2_TCM    31: PAPI_RES_STL   32: PAPI_TLB_IM
```

Fig. 3. (cont.)

```
Statistics
================================================================
Counting domain.......................................................  user
Multiplexed...........................................................  yes
Graduated floating point instructions per cycle........................  0.004
Vector instructions per cycle..........................................  0.000
Floating point instructions per graduated instruction.................  0.006
Vector instructions per graduated instruction..........................  0.000
Floating point instructions per level 1 data cache access..............  0.007
Graduated instructions per cycle.......................................  0.747
Issued instructions per cycle..........................................  0.834
Graduated instructions per issued instruction..........................  0.895
Issued instructions per level 1 instruction cache miss.................  2535.013
Graduated instructions per level 1 instruction cache miss..............  2268.839
Level 1 instruction cache miss ratio...................................  0.000
Level 1 data cache accesses per graduated instruction..................  0.764
% floating point instructions of all graduated instructions............  0.561
% cycles stalled on any resource.......................................  45.293
Level 1 instruction cache misses per issued instruction................  0.000
Level 1 cache read miss ratio (instruction)............................  0.000
Level 1 cache miss ratio (data)........................................  0.007
Level 1 cache miss ratio (instruction).................................  0.000
Bandwidth used to level 1 cache (MB/s).................................  117.237
Bandwidth used to level 2 cache (MB/s).................................  83.534
MFLIPS (cycles)........................................................  4.178
MFLIPS (wall clock)....................................................  3.943
MVOPS (cycles).........................................................  0.000
MVOPS (wall clock).....................................................  0.000
MIPS (cycles)..........................................................  744.486
MIPS (wall clock)......................................................  702.628
CPU time (seconds).....................................................  137.354
Wall clock time (seconds)..............................................  145.537
% CPU utilization......................................................  94.378
```

Fig. 3. (cont.)

The user may extend the scope of the report to include aggregrate information collected from all MPI tasks by first combining the individual XML documents into a single "multi-document" using psprocess, and then repeating the post-processing step with the new multi-document, as follows:

```
$ psprocess -c psrun.*.xml > combined.xml
$ psprocess combined.xml
```

```
PerfSuite Hardware Performance Summary Report

Version                  :  1.0
Created                  :  Thu Apr 01 10:54:05 AM CST 2004
Generator                :  psprocess 0.2
XML Source               :  combined.xml

Execution Information
======================================================================
Date                     :  Wed Mar 31 06:28:52 PM CST 2004
Hosts                    :  cn003 cn004 cn005 cn006
Users                    :  rkufrin

Aggregate Statistics                    Min      Max   Median     Mean   StdDev      Sum
======================================================================
% CPU utilization..............        92.68    94.86    94.24    94.00    0.72   751.99
% cycles stalled on any resour          7.10    75.42    58.03    53.86   21.70   430.90
% floating point instructions of all graduated instructions
                                        0.00     1.26     0.81     0.77    0.39     6.14
Bandwidth used to level 1 cache (MB/s)
                                        0.75   161.66   136.20   122.36   51.00   978.89
Bandwidth used to level 2 cache (MB/s)
                                        0.38   140.33   112.68   100.70   44.18   805.62
CPU time (seconds).............       136.21   137.40   137.23   136.92    0.53  1095.40
Floating point instructions per graduated instruction
                                        0.00     0.01     0.01     0.01    0.00     0.06
Floating point instructions per level 1 data cache access
                                        0.00     0.01     0.01     0.01    0.00     0.08
Graduated floating point instructions per cycle
                                        0.00     0.01     0.01     0.00    0.00     0.04
Graduated instructions                  0.50     1.11     0.63     0.69    0.20     5.54
Graduated instructions per issued instruction   0.43     0.90     0.66     0.67    0.18     5.38
Graduated instructions per level 1 instruction cache miss
                                     1331.06 71290.13  1995.55 10647.95 24506.02 85183.61
Issued instructions per cycle...        0.79     1.28     1.03     1.05    0.20     8.39
Issued instructions per level 1 instruction cache miss
                                     2437.11 82205.88  3131.90 13081.17 27941.92 104649.37
Level 1 cache miss ratio (data).        0.00     0.01     0.01     0.01    0.00     0.06
Level 1 cache miss ratio (instruction)    0.00     0.00     0.00     0.00    0.00     0.00
Level 1 cache read miss ratio (instruction)    0.00     0.00     0.00     0.00    0.00     0.00
Level 1 data cache accesses per graduated instruction
                                        0.69     0.85     0.79     0.79    0.05     6.31
Level 1 instruction cache miss          0.00     0.00     0.00     0.00    0.00     0.00
Level 1 instruction cache misses per issued instruction
                                        0.00     0.00     0.00     0.00    0.00     0.00
MFLIPS (cycles)................         0.00     6.31     5.14     4.65    2.00    37.21
MFLIPS (wall clock)............         0.00     5.85     4.83     4.37    1.87    34.93
MIPS (cycles)..................       498.73  1107.23   632.90   690.91  194.84  5527.31
MIPS (wall clock)...........          462.20  1044.74   594.26   650.07  185.79  5200.53
MVOPS (cycles).........                 0.00     0.00     0.00     0.00    0.00     0.00
MVOPS (wall clock).............         0.00     0.00     0.00     0.00    0.00     0.00
Vector instructions per cycle...        0.00     0.00     0.00     0.00    0.00     0.00
Vector instructions per graduated insdtruction   0.00     0.00     0.00     0.00    0.00     0.00
Wall clock time (seconds).......      144.72   147.05   145.46   145.67    0.89  1165.38
```

Fig. 4. Text-based performance output from a parallel run (counting mode)

Figure 4 shows an example of this type of performance reporting. To allow for more scalable handling of potentially large processor-count runs, information is displayed using standard descriptive statistics (e.g., mean/max/min, deviation). The intent is to allow the user to quickly isolate outliers with respect to performance that can be then examined more closely to determine specific causes for performance degradation. In the case of this particular example (pcg), the algorithm dedicates one processor to handling I/O activities and therefore there is a single task that displays significantly different behavior than the remainder. By isolating and removing this extraneous task from the aggregate report (by simply not including it in the combining step above), the user can focus on those tasks that are performing the bulk of the computational work.

With minimal effort, the performance analysis with PerfSuite can be adjusted to work in profiling mode. There is no need for the user to modify their build process for their application in any way, with the exception of retaining symbol table information to allow for mapping of program addresses to specific source code locations. Using PerfSuite in profiling mode is accomplished by specifying an alternate XML configuration document, as follows:

```
$ mpirun -np 8 psrun -c /usr/share/perfsuite/xml/pshwpc/papi_profile_cycles.xml pcg
$ psprocess -e pcg psrun.PID.xml
```

This minor change to the command-line used to invoke psrun results in an XML document being created that records the results of a statistical profiling experiment using any PAPI event as a trigger (this is similar to functionality provided in SGI's SpeedShop toolset). An example of the output of psprocess (edited to reduce space requirements) when applied to such an experiment is shown in Figure 5.

As previously mentioned, the primary motivation for PerfSuite development was to enable easy-to-use and general techniques for performance analysis on IA-32 and IA-64 Linux clusters at NCSA. However, the software was soon adopted for center-wide use with the express purpose of automating the performance analysis of the entire workload of jobs running on NCSA's large-scale Pentium and Itanium clusters. Because user intervention is not required and measurements can be obtained with arbitrary existing applications, PerfSuite was incorporated within the software stack at NCSA to be used on all parallel applications submitted to these clusters. Within the span of eight months, nearly five million records of performance data were gathered this way and stored in a relational database for use in later workload characterization analysis. This automatic collection continues (now expanded to include the largest single cluster deployed to date at NCSA, a 2500+ processor Intel Xeon cluster, currently #4 on the Top500 supercomputer list).

```
PerfSuite Hardware Performance Summary Report

Profile Information
======================================================================
Class                          :  PAPI
Event                          :  PAPI_TOT_CYC (Total cycles)
Period                         :  10000000
Samples                        :  10422
Domain                         :  user
Run Time                       :  126.26 (seconds)
Min Self %                     :  (all)

Module Summary
--------------------------------------------------------------------------------
Samples   Self %   Total %   Module
   10030   96.24%   96.24%  /u/ncsa/rkufrin/apps/pcg/pcg
     374    3.59%   99.83%  /lib/libc-2.2.4.so
      17    0.16%   99.99%  /lib/libpthread-0.9.so
       1    0.01%  100.00%  /lib/libm-2.2.4.so

File Summary
--------------------------------------------------------------------------------
Samples   Self %   Total %   File
    8177   78.46%   78.46%  /u/ncsa/rkufrin/apps/pcg/matvect.c
     980    9.40%   87.86%  /u/ncsa/rkufrin/apps/pcg/main.c
     624    5.99%   93.85%
/usr/src/build/85131-i386/BUILD/glibc-2.2.4/csu/init.c
     244    2.34%   96.19%  /u/ncsa/rkufrin/apps/pcg/vector.c
     230    2.21%   98.40%
/usr/src/build/85131-i386/BUILD/glibc-2.2.4/string/../sysdeps/generic/memcpy.c
      72    0.69%   99.09%
/usr/src/build/85131-i386/BUILD/glibc-2.2.4/malloc/malloc.c
      28    0.27%   99.36%
/usr/src/build/85131-i386/BUILD/glibc-2.2.4/stdlib/strtod.c
      11    0.11%   99.46%
/usr/src/build/85131-i386/BUILD/glibc-2.2.4/stdio-common/vfscanf.c
       8    0.08%   99.54%
/usr/src/build/85131-i386/BUILD/glibc-2.2.4/linuxthreads/mutex.c
       7    0.07%   99.61%
/usr/src/build/85131-i386/BUILD/glibc-2.2.4/libio/../sysdeps/i386/bits/string.h
       7    0.07%   99.67%
/usr/src/build/85131-i386/BUILD/glibc-2.2.4/string/../sysdeps/generic/strncpy.c
       4    0.04%   99.71%  /u/ncsa/rkufrin/apps/pcg/dmio.c
Function Summary
--------------------------------------------------------------------------------
Samples   Self %   Total %   Function
    8177   78.46%   78.46%  dvec_mult_dspmat
     980    9.40%   87.86%  preconditioning
     624    5.99%   93.85%  ?
     230    2.21%   96.06%  memcpy
     125    1.20%   97.26%  dvec_all_dotprod
     105    1.01%   98.26%  saxpy
      67    0.64%   98.91%  chunk_free
      29    0.28%   99.18%  __strtod_internal
```

Fig. 5. Abbreviated psprocess output from a profiling experiment

The initial performance collection yielded extremely interesting and useful results and provided concrete information regarding the effective utilization of Linux-based clusters for state-of-the-art high-performance computing resources. For example, it was learned that the fraction of user applications achieving ten percent of the peak theoretical floating point performance was approximately 12% on NCSA's Pentium III cluster and approximately 7% on first-generation Itanium hardware. High-level performance characterizations such as enabled by these studies make it possible for center management to easily assess the effectiveness of the resources delivered to the user community and also provides awareness of specific applications that might be good candidates for more focused efforts in optimization by the developers and external performance experts. Figure 6 shows an example of a graphical breakdown of the NCSA workload characterization obtained using PerfSuite during a portion of 2003.

DynaProf

DynaProf is a performance analysis tool designed to insert performance measurement instrumentation directly into a running application's address space at run time []. The instrumentation included with the current release of DynaProf can measure real-time as well as any hardware performance metrics available through the PAPI hardware performance counter library. Run-time instrumentation of the object code has numerous advantages over traditional source-based performance profiling systems, most significant of which is the elimination of the interference of calls to the instrumentation with the compiler's optimization passes. For aggressively scheduled processors, significant code reorganization and subroutine inlining are often required for maximal utilization of the processors functional units and can interfere with source-code based performance instrumentation. An additional benefit is the removal of the instrumentation's dependency on the compilation process. The type and format of the instrumentation can be changed without recompiling the application, and instrumentation can be both inserted and removed dynamically while the application is running. On Linux systems, DynaProf is based on the freely available Dyninst dynamic instrumentation library from University of Maryland [2].

DynaProf provides a simple easy-to-use command line interface. Commands are provided for loading or attaching to an executable, listing the modules and functions and instrumentation points, and inserting instrumentation in the form of probes. For threaded codes, DynaProf detects a threaded executable and loads a special version of the probe library that detects thread creation and termination and instruments all threads. For MPI programs, DynaProf provides a special load command that enables instrumentation of all the MPI processes.

The current release of Dynaprof includes several measurement probes. The following three probes provide the ability to instrument specific regions of code:

?? **papiprobe** for measuring PAPI preset and native events
?? **papiclock** for measuring PAPI real-time and virtual-time cycles
?? **wallclock** for measuring real-time

These probes generate inclusive, exclusive, and 1-level call tree profile data for each instrumented function. Post-processing scripts are provided that display the profile data in human readable form.

papiprobe gathers measurements using PAPI [1,19]. PAPI uses the processor's hardware performance counters to measure specific hardware events like cache misses, branch mis-predictions and floating point instructions. By default, if no argument is specified, papiprobe defaults to counting with PAPI_FP_INS, or floating poin*t* instructions. Currently, Dynaprof uses PAPI in the *user domain*. This means that only events that occur in user *context* will be counted. Other activity on the system will not appreciably affect the counts of most operations except resources that must be flushed and reloaded upon context switches, like caches and TLBs. Note that the papiprobe also supports multiplexing of counters. That is, if you pass more events than your processor can count at any one given time, papiprobe will timeshare the counting hardware to give the illusion that there are far more counters available than actually exist on the hardware.

The wallclock probe takes no arguments. It very simply measures elapsed real-time which is sometimes referred to as wallclock time. It does this using the highest resolution and lowest latency real time clock available on the host architecture. The output units are in microseconds.

DynaProf inserts instrumentation directly into the application's address space. This is accomplished through a run-time code generation and patching mechanism based upon either Dyninst or DPCL, IBM's derivative effort. Whenever a function is instrumented, all its children are instrumented as well. This is to enable the probe to generate both inclusive and exclusive metrics.

Dynaprof does not enforce the manner in which each probe is to generate its output. By not placing these restrictions on the probe modules, the probe designer is free to determine whatever output format is most appropriate, be that a real time binary data feed to a visualization engine or a static data file dumped to disk at the end of the run. The probes included with Dynaprof write the collected data to disk either when the application finishes or the user explicitly sends the application a SIGHUP signal. This signal causes the probe module to flush the data to disk. Note that this data will be overwritten at the end of the run, so it is recommended that the user copy this data to a new file as soon as the flush has been performed. Currently, both the PAPI probe and the Wallclock probe produce a compact file consisting of encoded ASCII data. The data files are created in the directory where the application exists. Each probe prints a message to this effect when the probe is first initialized. The files are named <*executable.pid*>, where *pid* is the process identifier. For multithreaded applications, each thread generates a data file of the form <*executable.pid.tid*> where *tid* is the thread identifier.

Figure 7 shows an example of loading the swim application (a popular shallow water benchmark), enabling use of papiprobe, instrumenting selected functions, running the application, and generating a report. The instrumentation measures Level 1 Instruction and Level 1 Data Cache Misses.

```
(dynaprof) load tests/swim
(dynaprof) use probes/papiprobe PAPI_L1_DCM, PAPI_L1_ICM
(dynaprof) instr function swim.F calc*
swim.F, inserted 8 instrumentation points
(dynaprof) run
papiprobe: output goes to /home/mucci/work/dynaprof/tests/swim.7366
 SPEC benchmark 102.swim

 NUMBER OF POINTS IN THE X DIRECTION     512
 NUMBER OF POINTS IN THE Y DIRECTION     512
 GRID SPACING IN THE X DIRECTION      25000.
 GRID SPACING IN THE Y DIRECTION      25000.
 TIME STEP                             20.
 TIME FILTER PARAMETER               0.001
 NUMBER OF ITERATIONS                  120

 CYCLE NUMBER   60 MODEL TIME IN  HOURS  0.33

 Pcheck =   0.1314E+11
 Ucheck =   0.5215E+05
 Vcheck =   0.5215E+05


 CYCLE NUMBER  120 MODEL TIME IN  HOURS  0.67


Pcheck =   0.1314E+11
 Ucheck =   0.5215E+05
 Vcheck =   0.5215E+05

Program exited normally.
Now let's display the data.

[mucci@nebula]$ probes/papiproberpt /home/mucci/work/dynaprof/tests/swim.7366 > out
Output file            : /home/mucci/work/dynaprof/tests/swim.7366
Option string          : PAPI_L1_DCM,PAPI_L1_ICM
Processor              : 1198 Mhz GenuineIntel Intel Pentium III rev 0x1 (1-way)
Total metrics measured  : 2
Metric 1:              : PAPI_L1_DCM, Level 1 data cache misses (Native 0x45,0x45)
Metric 2:              : PAPI_L1_ICM, Level 1 instruction cache misses (Native 0xf28,0xf28)
Total functions        : 4

Exclusive Profile of Metric PAPI_L1_DCM.

Name          Percent        Total          Calls
-------------   -------        -----          --------
TOTAL          100            5.155e+08      1
calc3_        52.73          2.718e+08      118
calc2_        38.52          1.986e+08      120
calc1_        8.086          4.168e+07      120
unknown        0.3937         2.03e+06       1
calc3z_       0.2722         1.403e+06      1
```

Fig. 6. Using DynaProf to instrument the swim application and display performance data

Inclusive Profile of Metric PAPI_L1_DCM.

Name	Percent	Total	SubCalls
TOTAL	100	5.155e+08	0
calc3_	52.73	2.718e+08	0
calc2_	38.52	1.986e+08	0
calc1_	8.086	4.168e+07	0
calc3z_	0.2722	1.403e+06	0

1-Level Inclusive Call Tree of Metric PAPI_L1_DCM.

Parent/-Child	Percent	Total	Calls
TOTAL	100	5.155e+08	1
calc1_	100	4.168e+07	120
calc2_	100	1.986e+08	120
calc3z_	100	1.403e+06	1
calc3_	100	2.718e+08	118

Exclusive Profile of Metric PAPI_L1_ICM.

Name	Percent	Total	Calls
TOTAL	100	9.916e+04	1
unknown	29.52	2.927e+04	1
calc2_	24.01	2.381e+04	120
calc1_	23.5	2.331e+04	120
calc3_	22.87	2.268e+04	118
calc3z_	0.09378	93	1

Inclusive Profile of Metric PAPI_L1_ICM.

Name	Percent	Total	SubCalls
TOTAL	100	9.916e+04	0
calc2_	24.01	2.381e+04	0
calc1_	23.5	2.331e+04	0
calc3_	22.87	2.268e+04	0
calc3z_	0.09378	93	0

1-Level Inclusive Call Tree of Metric PAPI_L1_ICM.

Parent/-Child	Percent	Total	Calls
TOTAL	100	9.916e+04	1
calc1_	100	2.331e+04	120
calc2_	100	2.381e+04	120
calc3z_	100	93	1
calc3_	100	2.268e+04	118

Fig. 6. (cont.)

CUBE

Acceptance of performance tools among program developers is often limited by their complexity [18], which is usually perceived through a tool's user interface. Unfortunately, the development of user interfaces is a costly issue, so we designed and implemented a generic GUI named CUBE [24] explicitly emphasizing simplicity by combining only a small number of orthogonal
features.

CUBE (CUBE Uniform Behavioral Encoding) is a generic viewer that provides the ability to interactively browse through a multidimensional performance space. The performance space is essentially a mapping of metrics onto program resources, such as the call tree, and system resources, such as nodes and processes. The input of CUBE is a performance experiment stored in an XML-based file format. Similar to Paradyn [16], CUBE displays the different dimensions of the performance space consistently using tree browsers. However, since we are interested in interactively exploring a mapping of all metric/resources combinations onto numbers as opposed to highlighting a limited set of resource foci, we provide a flexible mechanism to select a particular section of the performance space that contains a subset of the values provided by the data set as well as to aggregate values within and across the different dimensions. The display is divided into three parts. The left pane contains various metrics organized in a specialization hierarchy (i.e., the metric tree), while the middle pane shows the call tree with its nodes representing call paths and the the left pane shows the system hierarchy consisting of nodes and processes or threads running on them. Alternatively, the user can switch from the call-tree view to a flat-profile representation giving the performance for functions as opposed to call paths. Figure 7 shows various metrics calculated from event traces for the SWEEP3D benchmark running on a Linux cluster.

Every tree-node in the display is labeled with a metric value, which is displayed simultaneously using a number as well as a colored icon. Colors enable the easy identification of nodes of interest even in a large tree, whereas the numerical values enable the precise comparison of individual values. The color is taken from a typical spectrum, as it is usually used for temperature scales. The idea is to assign small values a ``cold" color and big values a ``hot" color. A value shown in the metric tree represents the sum of a particular metric for the entire program, that is, across all call paths and the entire system. A value shown in the call tree represents the sum of the selected metric across the entire system for a particular call path. A value shown in the system tree represents the selected metric for the selected call path and a particular system entity. Briefly, a tree is always an aggregation of all of its neighbor trees to the right. For example, the user can click on a particular metric and see its distribution across the call tree. Other than that, the user can specify a certain level of detail by collapsing and expanding nodes. In collapsed state, a node aggregates across the entire subtree rooted at itself, in expanded state it represents only itself without any child nodes. The user can switch between absolute values and percentages relative to the overall execution time. To facilitate the comparison between different experiments the user can select another experiment as the basis for computing the percentages. Also, the GUI includes a source-code display that shows the position of a function or call

site when clicking on it in the call tree. CUBE is implemented in C++ using the wxWidgets GUI toolkit [26] and libxml2 [14] to parse the XML format.

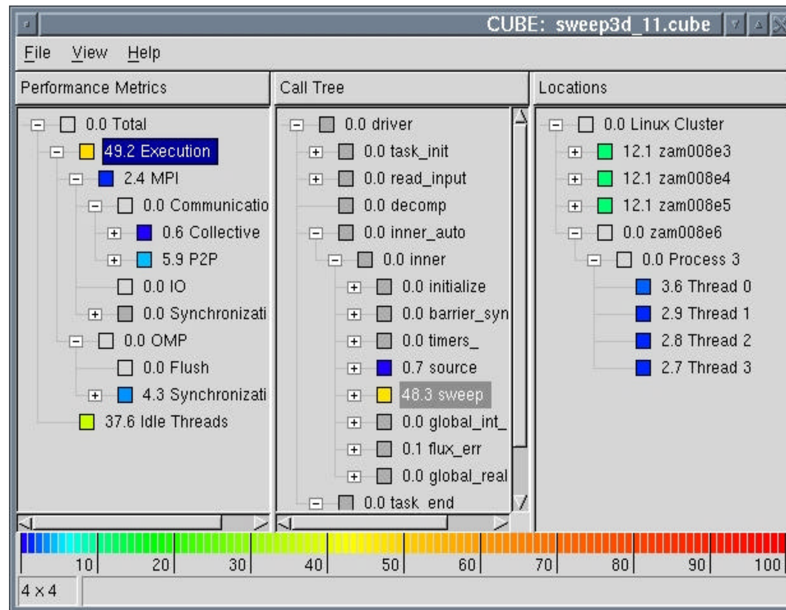

Fig. 7. CUBE display of performance data for SWEED3D benchmark

CUBE is generic in that the underlying data model is independent of specific metrics. It includes an API to write data files and can be used by any tool mapping metrics onto program and system resources in a profile-like fashion. Since flat profiles can be represented as a single-level call tree, it is not limited to tree profiles. A distinct feature of CUBE is its ability to support cross-experiment analysis by allowing the user to compute difference experiments from two experiments with different execution parameters or different code versions. The difference experiment is displayed just like an ordinary one only with the metric values replaced by difference values, so that the user can browse through all the program and system resources and see where and how much the performance differs. Since difference values happen to be negative and positive for different program or system resources, mathematical sign is graphically represented by giving the color icon a relief. A raised relief symbolizes a positive sign, whereas a sunken relief symbolizes a negative sign.

To demonstrate CUBE's usefulness in practice, we have combined it with KOJAK [25,8], an integrated performance evaluation environment for OpenMP and/or MPI applications, which is available for a large number of UNIX platforms including Linux clusters. KOJAK generates event traces from running applications and automatically searches them offline for execution patterns indicating inefficient performance behavior. In this way, KOJAK transform of the huge amount of low-level data

into a compact representation of performance behavior to be consumed by the end user.

KOJAK includes both tools for event-trace generation and post-mortem event-trace analysis. Recording of events related to OpenMP parallel execution is supported through the POMP [17] profiling interface for OpenMP. The OPARI preprocessor included in KOJAK allows users to automatically instrument the OpenMP constructs in their codes. On Linux systems, automatic instrumentations of user functions can be accomplished using third-party software, such as TAU [23] or the PGI compiler [20]. The PGI compiler provides a profiling interface which notifies the tracing library of function entry and exit events when the function was compiled using a specific compile flag. Besides analyzing inefficient use of the parallel programming model, KOJAK provides the ability to assess CPU and memory performance by analyzing counts of low-level hardware events collected with PAPI [1], such as cache misses or floating point instructions.

Changing execution parameters for a program can alter its performance behavior. Altering the performance behavior means that different results are achieved for different metrics. Some might increase while others might decrease. Some might rise in certain parts of the program only, while they drop off in other parts. Finding the reason for a gain or loss in overall performance often requires considering the performance change as a multidimensional structure. With CUBE's difference operator, a user can view this structure by computing the difference between two experiments and rendering the virtual result experiment like a real one.

We demonstrate this feature by comparing two different domain-decomposition strategies for CX3D [15]. CX3D is an MPI application used to simulate Czochralski crystal growth, a method applied in the silicon-wafer production. The simulation covers the convection processes occurring in a rotating cylindrical crucible filled with liquid melt. The convection, which strongly influences the chemical and physical properties of the growing crystal, is described by a system of partial differential equations. The crucible is modeled as a three-dimensional cubic mesh with its round shape expressed by cyclic border conditions. The mesh is distributed across the available processes using a two-dimensional spatial decomposition. The application was executed on an Intel Pentium III Xeon 550 MHz cluster with eight 4-way SMP nodes connected through Myrinet. We ran the application with a total of 16 processes on four nodes.

We used a 16 x 1 decomposition for the first and a 4 x 4 decomposition for the second experiment. The application was compiled and instrumented using the PGI compiler. We generated an event trace for each configuration and processed both of them using KOJAK, obtaining one CUBE experiment per trace. After that, we applied the difference operator to the two experiments (4 x 4) - (16 x 1) yielding a virtual difference experiment, which is shown in Figures 8 and 9.

The left pane with label ``Metrics'' provides the aggregated difference for all metrics. In Figure 8, the root metric is collapsed and shows the difference in total execution time. The difference is negative, as indicated by a minus sign and a sunken relief of the color square, so the 4 x 4 version performed better. Expanding the metric tree, as in Figure 9, reveals that in spite of the better overall performance, the late-sender problem (i.e., a receiver waiting for a message that has not been sent yet) became more severe, as indicated by a missing minus sign and a raised relief of the color

square. The middle pane contains the distribution of this waiting time across the call tree. It turns out that the aggravated late-sender problem came from a call path that was not taken in the 16 x 1 version. The conclusion is that the positive effect of the 4 x 4 configuration is to some extent overridden by the increased late-sender problem, which points to an opportunity for further improvement.

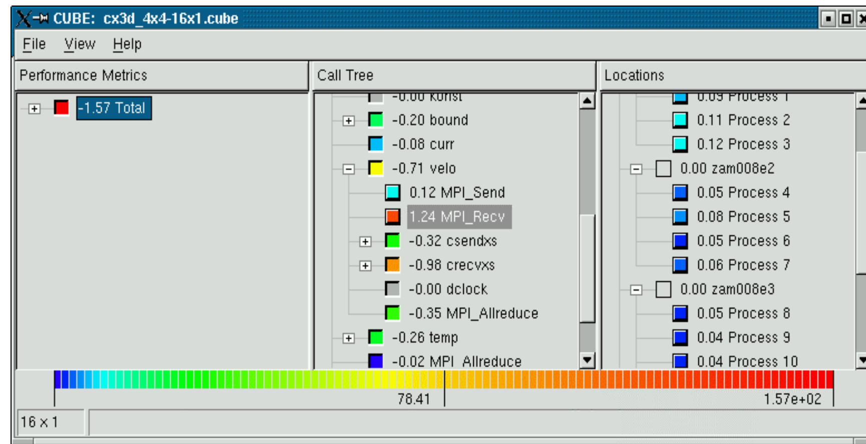

Fig. 8. CUBE analysis of CX3D executions showing difference in total execution time

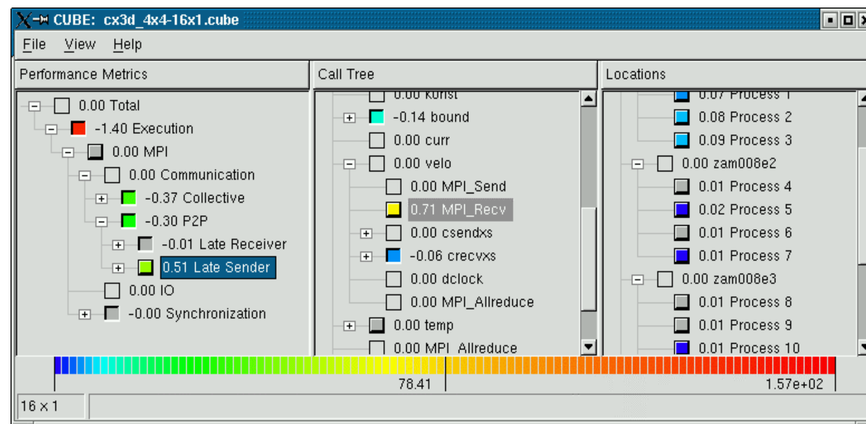

Fig. 9. CUBE analysis of CX3D executions showing differences in performance metrics

Related Work

There are a multitude of interactive browsers, such as AKSUM [22], HPCview [13], and SvPablo [4], that correlate the program structure with different performance metrics. From a technical viewpoint, also the coloring of nodes in the tree to symbolize a numeric value has been previously applied, for example, in the xlcb corefile [27] browser. The CUBE display's distinctive feature is the combination of a generic API that makes it available for third-party tools together with its ability to calculate and display difference experiments. Historically, CUBE emerged from the KOJAK [25] project, where a similar browser was used to present the results of event-trace analysis.

The logic used to calculate difference experiments extends the framework for multi-execution performance tuning by Karavanic et al. [7], which was used in the Paradyn project [16] for an optimization strategy based on using historical performance data to guide the search for performance bottlenecks. The most significant difference between CUBE and their framework is that the difference experiment can be processed and viewed like an ordinary one, that is, the difference operation is closed.

The concept of combining multiple experiments in a single (albeit virtual) experiment can be of great benefit when used together with experiment generators, such as ZENTURIO [21]. Likewise, the CUBE difference operation could as well work on data stored in a performance database, such as PerfDBF [3].

Conclusions and Future Work

Used collectively at different stages of the application development, testing, and tuning cycle, the above suite of tools can help reduce the time and effort involved in achieving good application performance on Linux clusters.

Further work is needed to integrate these tools. For example, we plan to write DynaProf probes to produce XML formatted data for PerfSuite and CUBE.

Acknowledgments

We would like to thank Zizhong Chen from University of Tennessee and Klaus Wingerath from Forschungszentrum Jülich for giving us access to their applications.

References

1. Browne, S., J. Dongarra, N. Garner, G. Ho, and P. Mucci. A Portable Programming Interface for Performance Evaluation on Modern Processors. *International Journal of High-Performance Computing Applications* 14(3): 189-204, 2000.

2. Buck, B. and J. Hollingsworth. An API for Runtime Code Patchin. *International Journal of High Performance Computing Applications* 14(4): 317-329, 2000.

3. Dongarra, J., A. Malony, S. Moore, P. Mucci, and S. Shende. Scalable Performance Analysis Infrastructure for Terascale Systems. *Future Generation Computer Systems*, 2004 (to appear).

4. DeRose, L.A., Y. Zhang, and D.A. Reed. SvPablo: A Multi-language Architecture-independent Performance Analysis System. *Proc. International Conference on Parallel Processing (ICPP'99)*, Fukishima, Japan, September 1999.

5. DynaProf, http://www.cs.utk.edu/~mucci/dynaprof/

6. Guiang, C.S., K.F. Milfeld, A. Purkayastha, and J. Boisseau. Memory Performance of Dual Processor Nodes: Comparison of Intel, Xeon, and AMD Opterson Memory Sybsystem Architectures. in *4th LCI International Conference on Linux Clusters: The HPC Revolution*, June 2003. San Jose, CA.

7. Karavanic, K.L. and B.P. Miller. A Framework for Multi-executoin Performance Tuning. *Parallel and Distributed Computing Practices* 4(3): 275-299, Special Issue on Monitoring Systems and Tool Interoperability, September 2001.

8. The KOJAK Project, http://icl.cs.utk.edu/kojak/

9. Janssen, C. VProf web site. http://aros.ca.sandia.gov/~cljanss/perf/vprof/

10. Kufrin, R. PerfSuite web sites. http://perfsuite.sourceforge.net and http://perfsuite.ncsa.uiuc.edu/

11. Kwok, W. Performance Analysis of PHASTA on the NCSA Intel IA-64 Linux Cluster. *Terascale Performance Analysis Workshop (ICCS 2003)*, Melbourne, Australia, June 2003.

12. Kwok, W., P. Li, F. Saied, and M. Norman. Performance analysis and tuning of ZEUS-MP on the NCSA IA-64 Linux cluster. Unpublished technical report, 2003.

13. Mellor-Crummey, J., R. Fowler, and G. Marin. HPCView: A Tool for Top-down Analysis of Node Performance. *The Journal of Supercomputing* 23:81-101, 2002.

14. The XML C Parser and toolkit of Gnome, http://www.xmlsoft.org/

15. Mihelcic, M., H. Wenzl, and H. Wingerath. Flow in Czochralski Crystal Growth Melts. Technical Report Jül-2697, Forschungszentrum Jülich, 1992.

16. Miller, B.P., M.D. Callaghan, J.M. Cargille, J.K. Hollingsworth, R.B. Irvine, K.L. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn Parallel Performance Measurement Tool. *IEEE Computer* 28(11): 37-46, 1995.

17. Mohr, B., A. Malony, S. Shende, and F. Wolf. Design and Prototype of a Performance Tool Interface for OpenMP. *The Journal of Supercomputing* 23:105-128, 2002.

18. Pancake, C.M. Applying Human Factors to the Design of Performance Tools. *Proc. 5th International Euro-Par Conference, LNCS 1685*, pp. 44-60, August/September 1999.

19. PAPI web site, http://icl.cs.utk.edu/PAPI/

20. The Portland Group Compiler Technology: Product Documentation, http://www.pgroup.com/doc/index.htm

21. Prodan, R., and T. Fahringer. On Using ZENTURIO for Performance Parameter Studies on Cluster and Grid Architectures. *Proc. 11th Euromicro Conference on Parallel Distributed Network-based Processing (PDP2003)*, Genua, Italy, February 2003.

22. Seragiotto Jr., C., M. Geissler, G. Madsen, and H. Moritsch. On Using Aksum for Semi-automatic Searching of Performance Problems in Parallel, Distributed Programs. *Proc. 11th Euromicro Conference on Parallel Distributed Network-based Processing (PDP 2003)*, Genua, Italy, February 2003.

23. Shende, S.S. The Role of Instrumentation and Mapping in Performance Measurement. Ph.D. Thesis, University of Oregon, August 2001.

24. Song, F. and F. Wolf.

25. Wolf, F. and B. Mohr. Automatic performance analysis of hybrid MPI/OpenMP applications. *Journal of Systems Architecture* 49(10-11): 421-439, Special Issue on Evolutions in parallel distributed, network-based processing, 2003.

26. wxWidgets open source C++ GUI framework, http://www.wxWidgets.org/

27. xlcb Graphical Browser: User Manual, Parallel Tools Consortium. http://web.engr.oregonstate.edu/~pancake/ptools/lcb/xlcb.html

28. Zagha, M., B. Larson, S. Turner, and M. Itzkowitz. Performance Analysis Using the MIPS R10000 Performance Counters. *Proc. Supercomputing '96*, IEEE/ACM, 1996.

.