Lawrence Berkeley National Laboratory

Lawrence Berkeley National Laboratory

Title

Performance Modeling: Understanding the Present and Predicting the Future

Permalink

https://escholarship.org/uc/item/1jp3949m

Authors

Bailey, David H.
Snavely, Allan

Publication Date

2005-11-30

eScholarship.org Powered by the California Digital Library
University of California

Performance Modeling: Understanding the Present and Predicting the Future

David H. Bailey[1] and Allan Snavely[2]

[1] Lawrence Berkeley Laboratory
Mail Stop 50B-2239, Berkeley, CA, 94720
dhbailey@lbl.gov
http://crd.lbl.gov/~dhbailey

[2] University of California, San Diego, 9500 Gilman Drive
La Jolla, California, 92093-0505
asnavely@cs.ucsd.edu
http://www.sdsc.edu/~allans

Abstract. We present an overview of current research in performance modeling, focusing on efforts underway in the Performance Evaluation Research Center (PERC). Using some new techniques, we are able to construct performance models that can be used to project the sustained performance of large-scale scientific programs on different systems, over a range of job and system sizes. Such models can be used by vendors in system designs, by computing centers in system acquisitions, and by application scientists to improve the performance of their codes.

1 Introduction

The goal of performance modeling is to gain understanding of a computer system's performance on various applications, by means of measurement and analysis, and then to encapsulate these characteristics in a compact formula. The resulting model can be used to gain greater understanding of the performance phenomena involved and to project performance to other system/application combinations.

We will focus here on large-scale scientific computation, although many of the techniques we describe below apply equally well to single-processor systems and to business-type applications. Also, this paper focuses on some work being done within the Performance Evaluation Research Center (PERC) [1], a research collaboration funded through the U.S. Department of Energy's Scientific Discovery through Advanced Computation (SciDAC) program [2]. A number of important performance modeling activities are also being done by other groups, for example at Los Alamos National Laboratory [3].

The performance profile of a given system/application combination depends on numerous factors, including: (1) system size; (2) system architecture; (3) processor speed; (4) multi-level cache latency and bandwidth; (5) interprocessor network latency and bandwidth; (6) system software efficiency; (7) type of application; (8) algorithms used; (9) programming language used; (10) problem size; (11) amount of I/O; and others. Indeed, a comprehensive model must incorporate most if not all of the above factors. Because of the difficulty in producing a truly comprehensive model, present-day performance modeling researchers generally limit the scope of their models to a single system and application, allowing only the system size and job size to vary. Nonetheless, as we shall see below, some recent efforts appear to be effective over a broader range of system/application choices.

Performance models can be used to improve architecture design, inform procurement, and guide application tuning. Unfortunately, the process of producing performance models historically has been rather expensive, requiring large amounts of computer time and highly expert human effort. This has severely limited the number of high-end applications that can be modeled and studied. Someone has observed that, due to the difficulty of developing performance models for new applications, as well as the increasing complexity of new systems, our supercomputers have become better at predicting and explaining natural phenomena (such as the weather) than at predicting and explaining the performance of themselves or other computers.

2 Applications of Performance Modeling

Performance modeling can be used in numerous ways. Here is a brief summary of these usages, both present-day and future possibilities:

Runtime estimation. The most common application for a performance model is to enable a scientist to estimate the runtime of a job when the input parameters for the job are changed, or when a different number of processors is used in a parallel computer system. One can also estimate the largest size of system that can be used to run a given problem before the parallel efficiency drops to an unacceptable area.

System design. Performance models are frequently employed by computer vendors in their design of future systems. Typically engineers construct a performance model for one or two key applications, and then compare future technology options based on performance model projections. Once performance modeling techniques are better developed, it may be possible to target many more applications and technology options in the design process. As an example of such "what-if" investigations, application parameters can be used to predict how performance rates would change with a larger or more highly associative cache. In a similar way, the performance impact of various network designs can be explored. We can even imagine that vendors could provide a variety of system customizations, depending on the nature of the user's anticipated applications.

System tuning. One example of using performance modeling for system tuning is given in [4]. Here a performance model was used to diagnose and rectify a misconfigured MPI channel buffer, which yielded a doubling of network performance for programs sending short messages. Along this line, Adolfy Hoisie of LANL recalls that when a recent system was installed, its performance fell below model predictions by almost a factor of two. However, further analysis uncovered some system difficulties, which, when rectified, improved performance to almost the same level the model predicted [3]. When observed performance of a system falls short of that predicted by a performance model, it may be the system that is wrong not the model!

Application tuning. If a memory performance model is combined with application parameters, one can predict how cache hit-rates would change if a different cache-blocking factor were used in the application. Once the optimal cache blocking has been identified, then the code can be permanently changed. Simple performance models can even be incorporated into an application code, permitting on-the-fly selection of different program options.

Performance models, by providing performance expectations based on the fundamental computational characteristics of algorithms, can also enable algorithmic choice before going to the trouble to implement all the possible choices. For example, in some recent work one of the present authors employed a performance model to estimate the benefit of employing an "inspector" scheme to reorder data-structures before being accessed by a sparse-matrix solver, as part of software being developed by the SciDAC Terascale Optimal PDE Simulations (TOPS) project [5]. It turned out that the overhead of these "inspector" schemes is more than repaid provided the sparse-matrices are large and/or highly randomized.

System procurement. Arguably the most compelling application of performance modeling, but one that heretofore has not been used much, is to simplify the selection process of a new computing facility for a university or laboratory. At the present time, most large system procurements involve a comparative test of several systems, using a set of application benchmarks chosen to be typical of the expected usage. In one case that the authors are aware of, 25 separate application benchmarks were specified, and numerous other system-level benchmark tests were required as well. Preparing a set of performance benchmarks for a large laboratory acquisition is a labor-intensive process, typically involving several highly skilled staff members. Analyzing and comparing the benchmark results also requires additional effort. These steps involved are summarized in the recent HECRTF report [6].

What is often overlooked in this regard is that each of the prospective vendors must also expend a comparable (or even greater) effort to implement and tune the benchmarks on their systems. Partly due to the high personnel costs of benchmark work, computer vendors often can afford only a minimal effort to implement the benchmarks, leaving little or no resources to tune or customize the implementations for a given system, even though such tuning and/or customization would greatly benefit the customer. In any event, vendors must factor the cost of implementing

and/or tuning benchmarks into the price that they must charge to the customer if successful. These costs are further multiplied because for every successful proposal, they must prepare several unsuccessful proposals.

Once a reasonably easy-to-use performance modeling facility is available, it may be possible to greatly reduce, if not eliminate, the benchmark tests that are specified in a procurement, replacing them by a measurement of certain performance model parameters for the target systems and applications. These parameters can then be used by the computer center staff to project performance rates for numerous system options. It may well be that a given center will decide not to rely completely on performance model results. But if even part of the normal application suite can be replaced, this will save considerable resources on both sides.

3 Basic Methodology

Our framework is based upon *application signatures*, *machine profiles* and *convolutions*. An application signature is a detailed but compact representation of the fundamental operations performed an application, independent of the target system. A machine profile is a representation of the capability of a system to carry out fundamental operations, independent of the particular application. A convolution is a means to rapidly combine application signatures with machine profiles in order to predict performance. In a nutshell, our methodology is to

- Summarize the requirements of applications in ways that are not too expensive in terms of time/space required to gather them but still contain sufficient detail to enable modeling.
- Obtain the application signatures automatically.
- Generalize the signatures to represent how the application would stress arbitrary (including future) machines.
- Extrapolate the signatures to larger problem sizes than what can be actually run at the present time.

With regards to application signatures, note that the source code of an application can be considered a high-level description, or application signature, of its computational resource requirements. However, depending on the language it may not be very compact (Matlab is compact, while Fortran is not). Also, determining the resource requirements the application from the source code may not be very easy (especially if the target machine does not exist!). Hence we need cheaper, faster, more flexible ways to obtain representations suitable for performance modeling work. A minimal goal is to combine the results of several compilation, execution, performance data analysis cycles into a signature, so these steps do not have to be repeated each time a new performance question is asked.

A dynamic instruction trace, such as a record of each memory address accessed (using a tool such as Dynist [7], of the Alpha processor tool ATOM) can also be considered to be an application signature. But it is not compact—address traces alone can run to several Gbytes even for short-running applications—and it is not machine independent.

A general approach that we have developed to analyze applications, which has resulted in considerable space reduction and a measure of machine independence, is the following: (1) statically analyze, then instrument and trace an application on some set of existing machines; (2) summarize, on-the-fly, the operations performed by the application; (3) tally operations indexed to the source code structures that generated them; and (4) perform a merge operation on the summaries from each machine [4,8-10]. From this data, one can obtain information on memory access patterns (namely, summaries of the stride and range of memory accesses generated by individual memory operations) and communications patterns (namely, summaries of sizes and type of communications performed).

The specific scheme to acquire an application signature is as follows: (1) conduct a series of experiments tracing a program, using the techniques described above; (2) analyze the trace by pattern detection to identify recurring sequences of messages and loads/store operations; and (3) select the most important sequences of patterns. With regards to (3), infrequent paths through the program are ignored, and sequences that map to insignificant performance contributions are dropped.

As a simple example, the performance behavior of CG (the Conjugate Gradient benchmark from the NAS Parallel Benchmarks [11]), which is more 1000 lines long, can be represented *from a performance standpoint* by one random memory access pattern. This is because 99% of execution is spent in the following loop:

```
do k = rowstr(j), rowstr(j+1)-1
  sum = sum + a(k)*p(colidx(k))
enddo
```

This loop has two floating-point operations, two stride-1 memory access patterns, and one random memory access pattern (the indirect index of p). On almost all of today's deep memory hierarchy machines the performance cost of the random memory access pattern dominates the other patterns and the floating-point work. As a practical matter, all that is required to predict the performance of CG on a machine is the size of the problem (which level of the memory hierarchy it fits in) and the rate at which the machine can do random loads from that level of the memory. Thus a random memory access pattern succinctly represents the most important demand that CG puts on any machine.

Obviously, many full applications spend a significant amount of time in more than one loop or function, and so the several patterns must be combined and weighted. Simple addition is often not the right combining operator for these patterns, because

different types of work may be involved (say memory accesses and communication). Also, our framework considers the impact of different compilers or different compiler flags in producing better code (so trace results are not machine independent). Finally, we develop models that include scaling and not just ones that work with a single problem size. For this, we use statistical methods applied to series of traces of different input sizes and/or CPU counts to derive a scaling model.

The second component of this performance modeling approach is to represent the resource capabilities of current and proposed machines, with emphasis on memory and communications capabilities, in an application-independent form suitable for parameterized modeling. In particular, we use low-level benchmarks to gather machine profiles, which are high-level representations of the rates at which machines can carry out basic operations (such as memory loads and stores and message passing), including the capabilities of memory units at each level of the memory hierarchy and the ability of machines to overlap memory operations with other kinds of operations (e.g., floating-point or communications operations). We then extend machine profiles to account for reduction in capability due to sharing (for example, to express how much the memory subsystem's or communication fabric's capability is diminished by sharing these with competing processors). Finally, we extrapolate to larger systems from validated machine profiles of similar but smaller systems.

To enable time tractable modeling we employ a range of simulation techniques [4; 12] to combine applications signatures with machine profiles:

- Convolution methods for mapping application signatures to machine profiles to enable time tractable statistical simulation.
- Techniques for modeling interactions between different memory access patterns within the same loop. For example, if a loop is 50% stride-1 and 50% random stride, we determine whether the performance is some composable function of the these two separate performance rates.
- Techniques for modeling the effect of competition between different applications (or task parallel programs) for shared resources. For example, if program A is thrashing L3 cache with a large working set and a random memory access pattern, we determine how that impacts the performance of program B with a stride-1 access pattern and a small working set that would otherwise fits in L3.
- Techniques for defining "performance similarity" in a meaningful way. For example, we determine whether loops that "look" the same in terms of application signatures and memory access patterns actually perform the same. If so, we define a set of loops that span the performance space.

In one sense, cycle-accurate simulation is the performance modeling baseline. Given enough time, and enough details about a machine, we can always explain and predict performance by stepping through the code instruction by instruction. However, simulation at this detail is exceedingly expensive. So we have developed fast-

to-evaluate machine models for current and proposed machines, which closely approximate cycle-accurate predictions by accounting for fewer details.

Our convolution method allows for relatively rapid development of performance models (full application models take 1 or 2 months now). Performance predictions are very fast to evaluate once the models are constructed (few minutes per prediction). The results are fairly accurate. Figure 1 show qualitatively the accuracy results across a set of machines and problem sizes and CPU counts for POP, the Parallel Ocean Program.

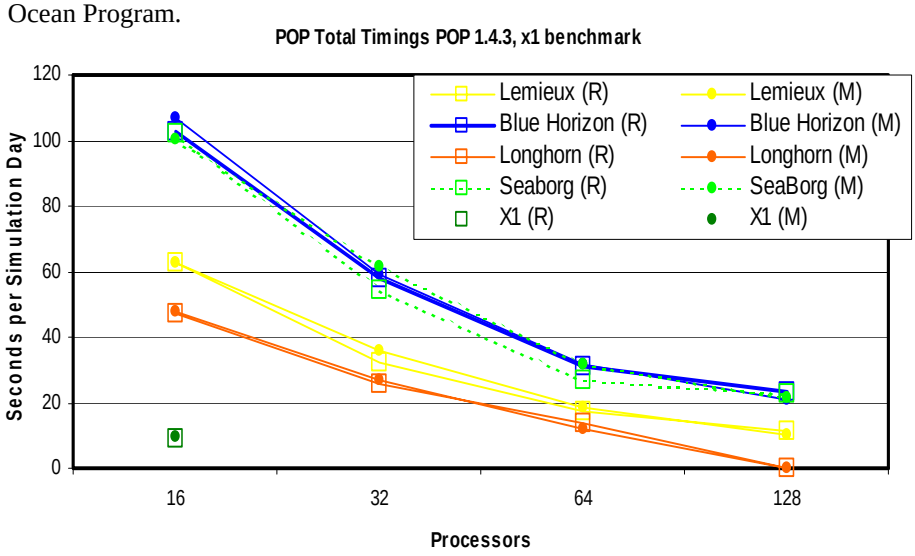


Fig. 1. Results for Parallel Ocean Program (POP). (R) is real runtime (M) is modeled (predicted) runtime

We have carried out similar exercise for several sizes and inputs of POP problems. And we have also modeled several applications from the DOD HPCMO [13] workload, including AVUS a CFD code, GAMESS a computational chemistry code, HYCOM a weather code, and OOCORE an out-of-core solver. In a stern test of the methods we were allowed access to DOD machines only to gather machine profiles via low-level benchmarks. We then modeled these large parallel applications at several CPU counts ranging from 16 to 384, on Power3, Power4 in two different flavors, Alpha, Xeon, and R16000 processor based supercomputers. We then predicated application performance on these machines; an d only after the predictions were issued were the application true runtimes independently ascertained by DOD personnel.

Table 1. Results of 'blind' predictions of DoD HPCMO Workload

Category	Average Absolute Error	Standard Deviation

Overall	**20.5%**	**18.2%**
AVUS std. input	15.0%	14.2%
AVUS large input	16.5%	16.2%
GAMESS std. input	45.1%	24.2%
HYCOM std. input	21.8%	16.7%
HYCOM large input	21.4%	16.9%
OOCORE std. input	32.1%	27.5%
Power3	17.4%	17.0%
Power4 p690	12.9%	9.6%
Power4 p655	15.7%	19.9%
Alpha	29%	17.6%
R16000	41.0%	18.5%
Xeon	28.2%	12.3%

Table 1 above gives the overall average absolute error and standard deviation of absolute average error as well as breakdowns by application/input and architecture. We conducted this 'blind' test (without knowing the performance of the applications in advance) in order to subject our modeling methods to the sternest possible test and because we think it is important to report successes *and failures* in modeling in order to advance the science. The conditions of independent application runtime assessment led to some of the error above. For example, we modeled the MPI version of GAMESS but in several cases it was the shmem version that was run (a case of predicting an apple and getting an orange). In the case of the Power 3, the predictions were consistently too high which was later traced to a mis-configured system parameter that allowed paging (another case of the machine being broken rather than the model). However some weaknesses in the models were also identified; the models do not do a good job of modeling I/O at present, which contributed to high application error for OOCORE (an I/O intensive code) and high machine error in the case of the Alpha system (which has a weak I/O subsystem). Xeons were consistently over predicted for reasons that appear to have to do with weak architectural support for floating-point (few, shallow, pipelines). Augmentation of the models to address systematic errors and add additional terms for I/O and enhanced accuracy of floating-point scheduling is work in progress.

4 Performance Sensitivity Studies

Reporting the accuracy of performance models in terms of model-predicted time vs. observed time (as in the previous section) is mostly just a validating step for obtaining confidence in the model. A more interesting and useful exercise is to explain and quantify performance differences and to play "what if" using the model. For example, it is clear from Figure 1 above that Lemeiux, the Alpha-based system, is faster across-the-board on POP x1 than is Blue Horizon, the Power3 system. The question is why? Lemeuix has faster processors (1GHz vs. 375 MHz), and a lower-latency network (a measured ping-pong latency of about 5 ms vs. about 19 ms), but Blue Horizon's network has the higher bandwidth (a measured ping-pong bandwidth

of about 350 MB/s vs. 269 MB/s). Without a model, one is left to conjecture "I guess POP performance is more sensitive to processor performance and network latency than network bandwidth," but without solid evidence.

With a model that can accurately predict application performance based on properties of the code and the machine, we can carry out precise modeling experiments such as that represented in Figure 2. Here we model perturbing the Blue Horizon (BH) system (withPower3 processors and a Colony switch) into the TCS system (with Alpha ES640 processors and the Quadrics switch) by replacing components one by one. Figure 2 represents a series of cases modeling the perturbing from BH to TCS, going from left to right. The four bars for each case represent the performance of POP x1 on 16 processors, the processor and memory subsystem performance, the network bandwidth, and the network latency, all normalized to that of BH.

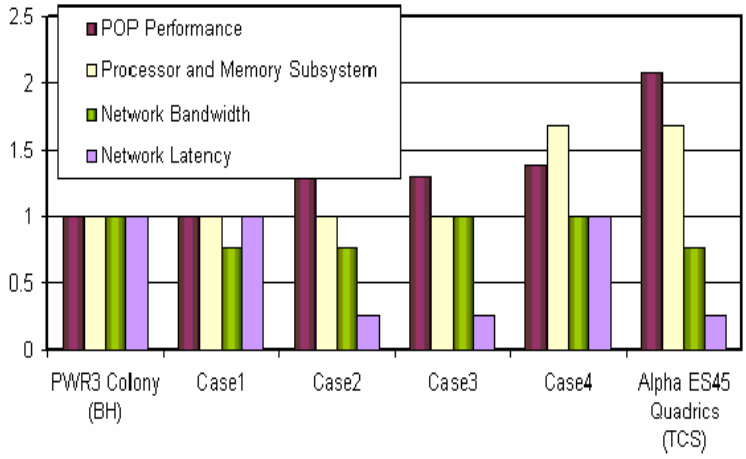


Fig. 2. Performance Sensitivity study of POP applied to proposed Lemieux upgrade

In Case 1, we model the effect of reducing the bandwidth of BH's network to that of a single rail of the Quadrics switch. There is no observable performance effect, as the POP x1 problem at this size is not sensitive to a change in peak network bandwidth from 350MB/s to 269MB/s. In Case 2, we model the effect of replacing the Colony switch with the Quadrics switch. Here there is a significant performance improvement, due to the 5 ms latency of the Quadrics switch versus the 20 ms latency of the Colony switch. This is evidence that the barotropic calculations in POP x1 at this size are latency sensitive. In Case 3, we use Quadrics latency but the Colony bandwidth just for completeness. In Case 4, we model keeping the Colony switch latency and bandwidth figures, but replacing the Power3 processors and local memory subsystem with Alpha ES640 processors and their memory subsystem. There is a substantial improvement in performance, due mainly to the faster memory subsystem

of the Alpha. The Alpha can load stride-1 data from its L2 cache at about twice the rate of the Power3, and this benefits POP x1 significantly. The last set of bars show the TCS values of performance, processor and memory subsystem speed, network bandwidth and latency, as a ratio of the BH values.

The principal observation from the above exercise is that the model can quantify the performance impact of each machine hardware component.

In these studies we find that larger CPU count POP x1 problems become more network latency sensitive and remain not-very bandwidth sensitive.

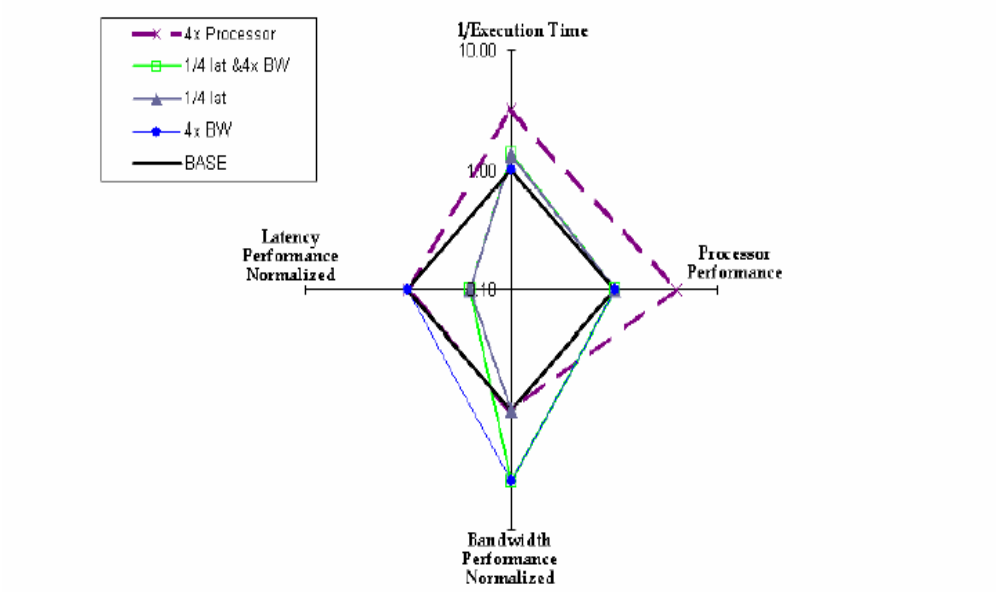


Fig. 3. A generalized performance sensitivity study

We can generalize a specific architecture comparison study such as the above, by using the model to generate a machine-independent performance sensitivity study. As an example, Figure 3 indicates the performance impact on the 128-CPU POP x1 program of quadrupling the speed of the CPU-memory subsystem (lumped together we call this the processor), quadrupling the network bandwidth, reducing network latency by four, and various combinations of these four-fold hardware improvements. The data values are plotted in a logarithmic scale and normalized to one, so that the solid black quadrilateral represents the execution time, network bandwidth, network latency, CPU and memory subsystem speed of Blue Horizon. At this size, POP x1 is quite sensitive to processor speed (a faster CPU and memory subsystem), somewhat sensitive to latency (because of the barotropic portion of the code is communications-bound, with small-messages), and fairly insensitive to bandwidth. In a similar way we can "zoom in" on the processor performance factor. In the above results for POP,

the processor axis shows modeled execution time decreasing from a four-times faster CPU with respect to clock rate (implying a 4X floating-point issue rate), but also quadruple bandwidth and one-quarter latency to all levels of the memory hierarchy (unfortunately this may be hard or expensive to achieve architecturally!).

Conclusion

We have seen that performance models enable "what-if" analyses of the implications of improving the target machine in various dimensions. Such analyses obviously are useful to system designers, helping them optimize system architectures for the highest sustained performance on a target set of applications. They are potentially quite useful in helping computing centers select the best system in an acquisition. But these methods can also be used by application scientists to improve performance in their codes, by better understanding which tuning measures yield the most improvement in sustained performance.

With further improvements in this methodology, we can envision a future wherein these techniques are embedded in application code, or even in system software, thus enabling self-tuning applications for user codes. For example, we can conceive of an application that performs the first of many iterations using numerous cache blocking parameters, a separate combination on each processor, and then uses a simple performance model to select the most favorable combination. This combination would then be used for all remaining iterations.

Our methods have reduced the time required for performance modeling, but much work needs to be done here. Also, running an application to obtain the necessary trace information multiplies the run time by a large factor (roughly 1000). The future work in this arena will need to focus on further reducing the both the human and computer costs.

References

1. The Performance Evaluation Research Center (PERC), see http://www.perc.nersc.gov
2. Scientific Discovery through Advanced Computing (SciDAC), see http://www.science.doe.gov/scidac .
3. Hoisie, A., O. Lubeck, H. Wasserman, "Performance and Scalability Analysis of Teraflop-Scale Parallel Architectures Using Multidimensional Wavefront Applications," The International Journal of High Performance Computing Applications, vol. 14 (2000), no. 4, pg 330-346.
4. Carrington, L., A. Snavely, X. Gao, and N. Wolter, "A Performance Prediction Framework for Scientific Applications," ICCS Workshop on Performance Modeling and Analysis (PMA03), June 2003, Melbourne, Australia.
5. Terascale Optimal PDE Simulations (TOPS) project, see

http://www-unix.mcs.anl.gov/scidac-tops .
6. Report of the High-End Computing Revitalization Task Force (HECRTF), see http://www.sc.doe.gov/ascr/hecrtfrpt.pdf .
7. Buck, B. and J. K. Hollingsworth, “An API for Runtime Code Patching,” Journal of Supercomputing Applications, 14(4), 2000, pp. 317-329.
8. Snavely, A., L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha, “A Framework for Application Performance Modeling and Prediction,” Proceedings of SC2002, Nov. 2002, Baltimore, MD.
9. Snavely, A., X. Gao, C. Lee, N. Wolter, J. Labarta, J. Gimenez, and P. Jones, “Performance Modeling of HPC Applications,” Proceedings of the Parallel Computing Conference 2003, Oct. 2003, Dresden, Germany.
10. Carrington, L., N. Wolter, A. Snavely, and C. B. Lee, “Applying an Automated Framework to Produce Accurate Blind Performance Predictions of Full-Scale HPC Applications”, UGC 2004, Williamsburgh, June 2004.
11. Bailey, D., et. al, “The NAS Parallel Benchmarks,” International Journal of Supercomputer Applications, vol. 5 (1991), no. 3, pg. 66-73.
12. Perelman, E., G. Hamerly, M. V. Biesbrouck, T. Sherwood, and B. Calder, “Using SimPoint for accurate and efficient simulation,” ACM SIGMETRICS Performance Evaluation Review, vol. 31 (2003), no. 1, pg. 318-319.
13. Department of Defense High Performance Computing Modernization Office (HPCMO), see http://www.hpcmo.hpc.mil .