Performance Prediction and Ranking of Supercomputers

Tzu-Yi Chen
Department of Computer Science
Pomona College
Claremont, CA 91711
tzuyi@cs.pomona.edu

Omid Khalili
Department of Computer Science and Engineering
University of California, San Diego
9500 Gilman Drive, Mail Code 0404
La Jolla, CA 92093-0404
okhalili@cs.ucsd.edu

Roy L. Campbell, Jr.
Army Research Laboratory
Major Shared Resource Center
Aberdeen Proving Ground, MD 21005
rcampbell@arl.army.mil

Laura Carrington, Mustafa M. Tikir, and Allan Snavely
Performance Modeling and Characterization (PMaC) Lab, UCSD
9500 Gilman Dr
La Jolla, CA 92093-0505
{lcarring,mtikir,allans}@sdsc.edu

Contents

Abstract

Performance prediction asks how much time executing an application is likely to take on a particular machine. Machine ranking asks which of a set of machines is likely to execute an application most quickly. These two questions are discussed within the context of large parallel applications run on supercomputers. Different techniques are surveyed, including a framework for a general approach that weights the results of machine benchmarks run on all systems of interest. Variations within the framework are described and tested on data from large-scale applications run

on modern supercomputers, helping to illustrate the trade-offs in accuracy and effort that are inherent in any method for answering these two questions.

1 Introduction

Given a parallel application, consider answering the following two questions: how much time is executing the application likely to take on a particular machine, and which of a set of machines is likely to execute the application most quickly? Answers to these questions could enable users to tune their applications for specific machines, or to choose a machine on which to run their applications. Answers could help a supercomputing center schedule applications across resources more effectively, or provide them data for decisions regarding machine acquisitions. More subtly, answers might enable computer architects to design machines on which particular applications are likely to run quickly.

But answering these questions is not easy. Since the performance of a parallel application is a function of both the application and the machine it runs on, accurate performance prediction has become increasingly difficult as both applications and computer architectures become more complex. Consider Figure 1, which plots the normalized relative runtimes of 8 large-scale applications on 6 supercomputers, both described in Section 4.[1] The plot shows that across these applications, no single machine is always the fastest, suggesting that there is no trivial way to predict even relative performance across a set of machines.

Since different machines can be best for different applications, this chapter discusses techniques for answering the original two questions:

- How can one accurately predict the running time of a specific application, on a given input, on a particular number of processors, on a given machine? (performance prediction)
- How can one accurately predict which of a set of machines is likely to execute an application fastest? (machine ranking)

Note that while the ability to do the former gives us a way to do the latter, the reverse is not true. In practice, however, while any method for predicting performance (including the many that are described in Section 2) could also be used to rank machines, users may consider the work required for the initial prediction to be excessive. Since sometimes the only information desired is that of expected relative performance across a set of machines, this scenario is also addressed. In addition, Section 7 considers an even further generalization where the goal is to find an application *independent* machine ranking that is sufficiently accurate to provide useful information.

[1] Since the machines shown are only a subset of those on which each application was run, the highest bar is not at 1 for every application. In addition, not all applications were run on all machines with the chosen number of processors.

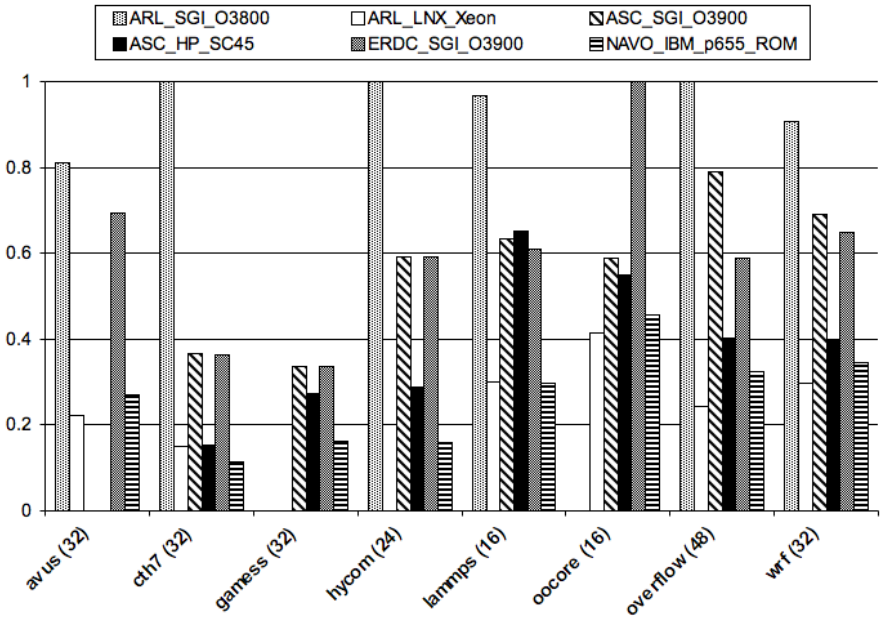

Figure 1: This graph shows the relative runtimes of 8 applications on 6 supercomputers. The x-axis gives the name of the application, with the number of processors on which it was run in parentheses. The y-axis gives the runtime divided by the maximum time taken by any of a larger set of 14 machines to run the particular application.

The rest of this chapter is laid out as follows. Section 2 surveys some general approaches for both performance prediction and machine ranking, organized by the type of information and level of expertise required for each. The trade-offs inherent in choosing a particular performance prediction technique or a particular ranking technique are studied in the context of a methodology that attempts to achieve a balance between accuracy and effort in Sections 3 through 6. The basic framework for the methodology is covered in Section 3, details are addressed in Section 4, and examples of how different techniques fit into this framework are discussed in Sections 5 and 6. An exploration of how to apply the techniques to generate application-independent machine rankings is presented in Section 7. Results are given and analyzed in each section.

2 Methods for predicting performance

Methods for predicting the performance of an application on a parallel machine begin with the assumption that the running time is a function of application and system characteristics. Currently the most accurate predictions are made by

creating detailed models of individual applications which describe the running time as a function of system characteristics. These characteristics can then be carefully measured on all systems of interest. However, in many situations the time and/or expertise required to build a detailed model of an individual application are not available, and sometimes even the system characteristics can not be measured (for example, when performance predictions are used to help decide which of a number of proposed supercomputers to build). And even when data are available, there are situations in which precise performance prediction is unnecessary: for example, when the question is which of a set of machines is expected to run a particular application most quickly.

In this section we give an overview of some of the approaches to performance prediction, noting connections to ranking as appropriate. The techniques are distinguished by the amount of information they use about a system and an application, as well as by the sophistication of the ways in which they use that information to predict the performance of an application.

While the focus here is on predicting the performance of large-scale applications on parallel machines, there has also been considerable work on predicting the performance of single processor applications (see, for example, [30]).

2.1 Benchmarks

At one extreme are benchmarks, which can be used to predict performance using only information about the machine. Consider that low-level performance metrics such as processor speed and peak floating-point issue rate are commonly reported, even in mass-market computer advertisements. The implication is that these numbers can be used to predict how fast applications will run on different machines, hence faster is better. Of course, manufacturer specifications such as theoretical peak floating-point issue rates are rarely achieved in practice, so simple benchmarks may more accurately predict relative application performance on different machines.

A particularly well-known parallel benchmark is Linpack [14], which has been used since 1993 to rank supercomputers for inclusion on the Top 500 list [45]. The Top 500 list is popular partly because it is easy to read, is based on a simple metric that is easy to measure (essentially peak FLOPS), and is easy to update. Unfortunately, simple benchmarks such as Linpack may not be sufficient for accurately predicting runtimes of real applications [7]. This is not surprising, since Linpack gives a single number for a machine which, at best, allows the execution time to be modelled as some application-specific number divided by that particular system's performance on Linpack.

To better predict the performance of individual applications, two approaches have been taken. One is to provide benchmarks which more closely mimic actual applications. The best known of these is perhaps the NAS Parallel Benchmark suite [3], which consists of scaled down versions of real applications. The other is to provide benchmarks which take into consideration the performance of multiple system components. An early example of the latter considered the ratio of FLOPS to memory bandwidth [31], which has the advantage of allowing sim-

ple comparisons between machines since it also gives a single number for each machine.

More recently benchmark suites that give multiple performance numbers measuring assorted system characteristics have been proposed. These include the IDC Balanced Rating [20], which has been used to rank machines based on measurements in the broad areas of processor performance, memory system capability, and scaling capabilities; and the HPC Challenge (HPCC) benchmark [29], which consists of 7 tests measuring performance on tasks such as dense matrix multiplication and the Fast Fourier Transform. Of course, with such benchmark suites it becomes incumbent on the user to decide which measurements are most relevant for predicting the performance of a particular application.

Note that a benchmark such as Linpack, which generates a single number for each machine, produces an application independent ranking of machines. While their usefulness is limited, such rankings are still of significant interest, as the use of Linpack in generating the popular Top 500 list [45] demonstrates. Alternative methods for generating application independent rankings are explored in Section 7. In contrast, a benchmark suite that generates multiple numbers for each machine has the potential to produce more useful application specific rankings, but requires a user to interpret the benchmark numbers meaningfully.

2.2 Weighted benchmarks

If several benchmarks are run on a machine, a user must determine how to interpret the collection of results in light of an individual application. With a benchmark such as the NAS Parallel Benchmarks [3], a user can choose the benchmark that most closely resembles their particular application. For lower level benchmark suites such as HPCC [19], users can turn to research on performance prediction techniques that consist of weighting the results of simple benchmarks. The amount of information assumed to be available about the application in generating the weights can range from end-to-end runtimes on a set of machines, to more detailed information.

For example, Gustafson and Todi [16] used the term *convolution* to describe work relating "mini-application" performance to that of full applications. McCalpin [31] showed improved correlation between simple benchmarks and application performance, though the focus was on sequential applications rather than the parallel applications of interest here. Other work focussing on sequential applications includes that of Marin and Mellor-Crummey [30], who described a clever scheme for combining and weighting the attributes of applications by the results of simple probes. Using a full run of an application on a reference system, along with partial application runtimes on the reference and a target system, Yang et. al. [48] describe a technique for predicting the full application performance using the relative performance of the short runs. While the reported accuracy is quite good, this type of approach could miss computational behavior that changes over the runtime of the application; in addition, the accuracy was reduced when using the partial runtime to predict the application's

performance on a different problem size or number of processors.

Section 3 in this chapter discusses another general method for weighting benchmark measurements. Sections 4 and 5 discuss the use of a least squares regression to calculate weights for any set of machine benchmarks, and demonstrate their use for both performance prediction and machine ranking.

2.3 Building detailed performance models

For the most accurate performance predictions, users must employ time and expertise to build detailed models of individual applications of interest.

With this approach the user begins with an in-depth understanding of how the application works, including details about the computational requirements, the communication patterns, and so on. This understanding is then used to build a performance model consisting of a potentially complex equation that describes the running time in terms of variables that specify, for example, the size of the input, processor characteristics, and network characteristics. While this approach can generate highly accurate predictions, building the model is generally acknowledged to be a time-consuming and complex task [43]. Nevertheless, if there is significant interest in a critical application, the investment may be deemed worthwhile.

Other research focusses on highly accurate modelling of specific applications [17, 18, 23, 28]. The very detailed performance models built as a result have been used both to compare advanced architectures [22, 24], and to guide the performance optimizations of applications on specific machines [35].

Due to the difficulty of constructing detailed models, an assortment of general techniques for helping users build useful performance models has also been proposed.

Many of these methods are based on a hierarchical framework that is described in [1]. First the application is described at a high level as a set of tasks that communicate with one another in some order determined by the program. The dependencies are represented as a graph, which is assumed to expose all the parallelism in the application. This task graph is then used to predict the overall performance of the application, using low level information about how efficiently each task can be executed.

Examples that can be fit into this framework include work on modelling applications as collections of independent abstract Fortran tasks [36, 37, 38], as well as using graphs that represent the dependencies between processes to create accurate models [32, 33, 39]. Other work describes tools for classifying overhead costs and methods for building performance models based on an analysis of those overheads [11]. Another technique that also begins by building graphs that reveal all possible communication continues by measuring the potential costs on the target machine and uses those partial measurements for predicting the performance of the overall application [47].

2.4 Simulation

One way to try and approach the accuracy of detailed performance models, but without the same need for human expertise, is through simulation.

For example, one could use cycle accurate simulations of an application [4, 5, 6, 27, 34, 44]. Of course, the main drawback of this approach is the time required. Due to the level of detail in the simulations, it could take several orders of magnitude more time to simulate an application than to run it. Again, if there is significant interest in a single application, this expense may be considered acceptable.

A related technique described in detail in [42] and briefly referred to in Section 6 starts by profiling the application to get memory operation counts and information on network messages. This model is coarser than the detailed models described previously in that there is no attempt to capture the structure of the application; rather, the data collected provides a higher level model of what the application does. This information can later be convolved with the rates of memory operations — possibly by modeling the cache hierarchy of the target architecture on the application trace — and combined with a simulation of the network messages in order to predict the performance of the given application on the specified machine.

Other methods that attempt to avoid the overhead of cycle accurate simulations include those that instrument the application at a higher level [12, 15] in order to predict performance.

2.5 Other approaches

It is also worth noting other approaches that have been proposed for predicting the performance of large scale applications.

For example, attempts have been made to employ machine learning techniques. In [10] the authors examine statistical methods for estimating machine parameters and then describe how to use these random variables in a performance model. Neural networks are used in [21, 40] to make performance predictions for an application as a function of its input parameter space, without building performance models. This methodology can find nonlinear patterns in the training input in order to make accurate performance predictions; however, it requires first running the target application numerous times (over $10,000$ in the example in [21]) with a range of input parameters, which may not always be practical.

3 A method for weighting benchmarks

The rest of this chapter explores a few methods for predicting performance and for ranking machines. These methods are unified by their assumption that all knowledge regarding machine characteristics come from results of simple benchmark probes. That this is enough to distinguish the machines is demonstrated in Figure 2, which shows that different machines are best at different types of

operations. For example, when compared to the other machines, the machine labelled ARL_Xeon_36 has a very high FLOPS rate (as one would expect from its clock speed, shown in Table 1 in Section 4.1), but poor network bandwidth.

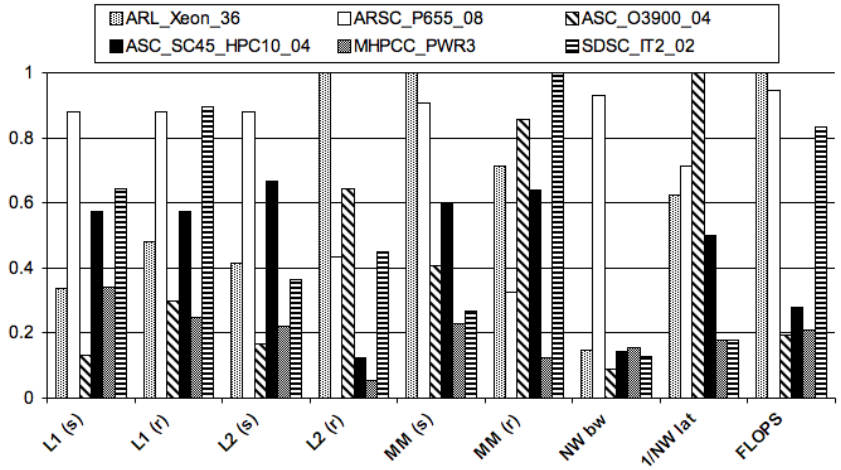

Figure 2: A plot of machine characteristics for a set of supercomputers. The characteristics along the x-axis are described in Table 2 in Section 4.1. The highest count for each characteristic is normalized to 1.

Almost all of the methods discussed also assume that information about the application is limited to end-to-end runtimes (although Sections 6 and 7 consider what can be done through incorporating the results of lightweight traces). Just as Figure 2 shows that different machines are best at different operations, Figure 3 demonstrates that different applications stress different types of machine operations. As a result, changing the behavior of a single system component can affect the overall performance of two applications in very different ways.

3.1 Machine and application characteristics

As noted previously, methods for predicting performance and generating machine rankings typically assume that performance is a function of machine and application characteristics. The question is then how to get the most accurate predictions and rankings using data about the machines and applications that is as cheap as possible to gather.

The examples explored in the rest of this chapter assume that basic benchmark measurements can be taken on all machines of interest (or, at a minimum, accurately estimated, as in the case where the system of interest has yet to be built). The same benchmarks must be run across all the machines, although no further assumptions are made. In particular, these benchmarks could consist of microbenchmarks (e.g., a benchmark measuring the network latency between

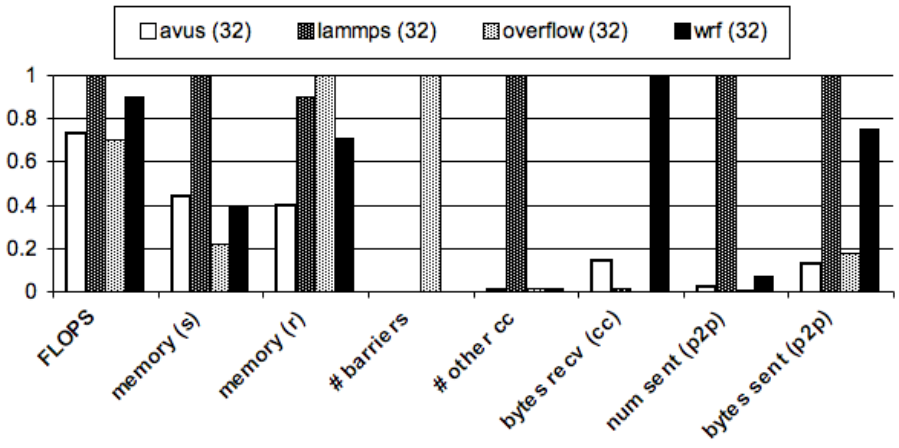

Figure 3: A plot of application characteristics for a set of parallel applications. From left to right, the x-axis refers to the average count over all processors of floating point operations, strided memory accesses, random memory accesses, barriers, other collective communications, the total number of bytes received as a result of those collective communications, the number of point to point messages sent, and the number of bytes sent as a result of those point to point communications. The highest count for each application characteristic is normalized to 1.

two nodes), or computational kernels (e.g., the FFT component of the HPC Challenge suite [19]), or something even closer to full-scale applications (e.g., the NAS Parallel Benchmarks [3]). This assumption is reasonable since, by their nature, these benchmarks tend to be easy to acquire and to run.

The examples in this chapter are of two types when it comes to the data needed regarding the applications. Those discussed in Section 5 require only end-to-end runtimes on some small number of machines; those discussed in Sections 6 and 7 use simple trace information about the application, including the number of floating point operations and/or the number of memory references. Memory and network trace information can be collected using tools such as the PMaC MetaSim Tracer [8] and MPIDTrace [2] tools, respectively. All the techniques also assume that the input parameters to the application during tracing and measurement of runtimes are the same across all machines. Note that while these techniques could be used even if the above assumption was not true, the resulting accuracy of the predictions could be arbitrarily poor.

3.2 General performance model

The other, more fundamental, assumption made by the techniques described in this chapter is that the performance of an application can be modelled to an

acceptable accuracy as a linear combination of the benchmark measurements. As a small example, say three benchmarks are run on some machine and that the benchmarks take m_1, m_2, and m_3 seconds, respectively. Then the assumption is that the running time P of any application (with some specified input data and run on some specific number of processors) on that machine can be approximated by the following equation:

$$P \approx m_1 w_1 + m_2 w_2 + m_3 w_3 = \mathbf{m} \cdot \mathbf{w} \quad (1)$$

Here w_1, w_2, and w_3 are constants that may depend on the application, the machine, the input, and the number of processors with which the application was run.

This model helps illustrate the trade-off between expertise/time and accuracy. While the linear model is appealingly simple, it could have difficulty capturing, say, the benefits of overlapping communication and computation in an application.

Given application runtimes on a set of machines, and benchmarks measurements on those machines, Section 5 describes how to use a least squares regression to obtain weights w that are optimal in the sense that they minimize the sum of the squares of the errors in the predicted times over a set of machines.

A less restrictive approach to the performance model can also be taken; an example is the method briefly summarized in Section 6. Instead of the dot-product in Equation 1, this method combines machine and application characteristics using a more complex convolution function.

3.3 Evaluating performance predictions

To test the methods that are based on linear regression, cross-validation is used. In other words, each machine in the data set is considered individually as the target machine for performance prediction. That machine is not included when calculating the weights w in Equation 1. Then, after the weights are calculated using only the other machines, those weights are used to predict the performance on the target machine. The absolute value of the relative error in the predicted time (given by Equation 2) is then calculated.

$$\left| \frac{predictedRT - actualRT}{actualRT} \right| \quad (2)$$

After repeating this process for each machine in the data set, the average of all the relative errors is reported. This process becomes clearer when looking at the examples in Section 5.

Note that cross-validation simulates the following real world usage of the method: a researcher has run their application on different systems, has access to the benchmark measurements on both those systems and a target system, and would like to predict the running time of the application on the target system without having to actually run the application on that machine.

Sections 5.3 and 6 briefly describe two other methods for performance prediction. There, again, the absolute value of the relative error is calculated, and the average over all machines is reported.

3.4 Evaluating rankings

The performance prediction models can also be used to rank a set of machines in order of the predicted runtimes of an application on those systems. Those rankings, or rankings generated in any other way, can be evaluated using the metric of *threshold inversions*, proposed in [9] and summarized here.

3.4.1 Predicting runtimes for ranking

Testing a predicted ranking requires predicting the performance on more than one machine at a time. So, instead of removing a single target machine as is done with the cross-validation procedure described previously, now a set of machines is randomly selected and the performance on all of those machines is predicted using some performance prediction methodology. Once the predicted runtimes are calculated, the number of threshold inversions between the predictions and the true runtimes can be determined. This is repeated 5000 times, each time choosing a random set of machines to rank and to count thresholded inversions for, to get a strong mix of randomly selected validation machines. While some sets may be repeated in the 5000 trials, because they are chosen at random, this should not greatly affect the average accuracies reported.

3.4.2 Thresholded inversions

A simple inversion occurs when a machine ranking predicts machine A will be faster than machine B on some application, but actual runtimes on the two machines show the opposite is true. For example, if machine A has larger network bandwidth than machine B, then the ranking based on network bandwidth would contain an inversion if, in practice, some application runs faster on machine B. The number of inversions in a ranking, then, is the number of pairs of machines that are inverted. In the above example this is the number of pairs of machines for which the inter-processor network bandwidth incorrectly predicts which machine should execute a given application faster. Note that if there are n machines, the number of inversions is at least 0 and is no larger than $n(n-1)/2$.

A threshold is added to the concept of an inversion in order to account for variations in collected application runtimes and/or benchmark measurements. These variations can be caused by a variety of reasons including, for example, system architectural and design decisions [26].

For evaluating the ranking methods presented here, two thresholds are used. One (α) accounts for variance in the measured runtimes, while the other (β) accounts for variance in the benchmark measurements. Both α and β are required to have values between 0 and 1, inclusive. For example, let the predicted

runtimes of A and B be $\hat{RT}_A$ and $\hat{RT}_B$ and the measured runtimes be RT_A and RT_B. If the predicted runtimes $\hat{RT}_A < \hat{RT}_B$, then A would be ranked better than B, and, if the measured runtimes $RT_A > RT_B$, then there is an inversion. Yet, when using threshold inversions with α, that would only count as an inversion if $RT_A > (1+\alpha) \times RT_B$. β is used in a similar fashion to allow for variance in benchmark measurements, and is usually set to be less than α. The different values for α and β are because one generally expects less variance in benchmark times than in full application runtimes due to the fact that benchmarks are typically simpler than large scale real applications and so their execution times are more consistent.

The examples in this chapter use values of $\alpha = .01$ (which means a difference of up to 1% in the application runtimes is considered insignificant) and $\beta = .001$ (which means a difference of up to .1% in the benchmark times is considered insignificant). In addition, Table 7 in Section 5 demonstrates the effect of changing the threshold values on the number of inversions for a particular scenario.

This metric based on thresholded inversions is particularly appealing because it is monotonic in the sense that adding another machine and its associated runtime cannot decrease the number of inversions in a ranking. Within our context of large parallel applications, this feature is highly desirable because often only partial runtime data is available: in other words, rarely have all applications of interest been run with the same inputs and on the same number of processors on all machines of interest.

4 Examples

While the techniques described in the following sections could be used for any set of benchmarks in order to study any parallel application, the examples in this chapter use the following machines, benchmarks, and applications. To see these techniques applied to other combinations of benchmarks and applications, see [25].

4.1 Machines

Table 1 summarizes the set of machines on which benchmark timings were collected and applications were run; the locations are abbreviations for the sites noted in the Acknowledgments at the end of the chapter. Regarding the benchmarks, information was collected about the FLOPS, the bandwidth to different levels of the memory hierarchy, and network bandwidth and latency.

To determine the FLOPS, results from the HPC Challenge benchmarks [19] were used. Although the HPC Challenge benchmarks were not run directly on the machines in Table 1, results on similar machines (determined based on their processor type, the processor frequency, and the number of processors) were always available and so were used instead. In practice, memory accesses always take significantly longer than floating point operations and so getting the exact

Location	Vendor	Processor	Frequency	# Processors
ASC	SGI	Altix	1.600GHz	2000
SDSC	IBM	IA64	1.500GHz	512
ARL	IBM	Opteron	2.200GHz	2304
ARL	IBM	P3	0.375GHz	1024
MHPCC	IBM	P3	0.375GHz	736
NAVO	IBM	P3	0.375GHz	928
NAVO	IBM	p655	1.700GHz	2832
NAVO	IBM	p655	1.700GHz	464
ARSC	IBM	p655	1.500GHz	784
MHPCC	IBM	p690	1.300GHz	320
NAVO	IBM	p690	1.300GHz	1328
ARL	IBM	p690	1.700GHz	128
ERDC	HP	SC40	0.833GHz	488
ASC	HP	SC45	1.000GHz	768
ERDC	HP	SC45	1.000GHz	488
ARSC	Cray	X1	0.800GHz	504
ERDC	Cray	X1	0.800GHz	240
AHPCRC	Cray	X1E	1.130GHz	960
ARL	LNX	Xeon	3.060GHz	256
ARL	LNX	Xeon	3.600GHz	2048

Table 1: Systems used for the examples in this chapter.

FLOPS measurement on each machine would be unlikely to make a significant difference in the results presented in this chapter.

Because increases in clock speed have far outpaced increases in the bandwidth between processors and memory, the bottleneck for today's applications is as, if not more, likely to be memory bandwidth than FLOPS [46]. As a result, the FLOPS measurement was augmented by the results of the MAPS benchmark in Membench [7], which measures the bandwidth to different levels of the memory hierarchy for both strided and random accesses. Note that the fundamental difference between strided and random memory references is that the former are predictable, and thus prefetchable. Because random memory references are not predictable, the bandwidth of random accesses actually reflects the latency of an access to some particular level of the memory hierarchy. As an example, Figure 4 demonstrates the result of running MAPS on an IBM p690 node to measure the bandwidth of strided accesses. As the size of the array increases, eventually it will no longer fit into smaller, faster caches — as a result, the effective bandwidth drops. A region for a memory level is defined as a range of data array sizes where the array fits into the level and achievable bandwidth from the level is fairly stable (each plateau in the MAPS curve). Once the regions for L1, L2, L3 caches, and main memory have been identified by a human expert, a single point in each region is used as the bandwidth metric for that level of the memory hierarchy.

Finally, Netbench [7] was used to measure network bandwidth and latency.

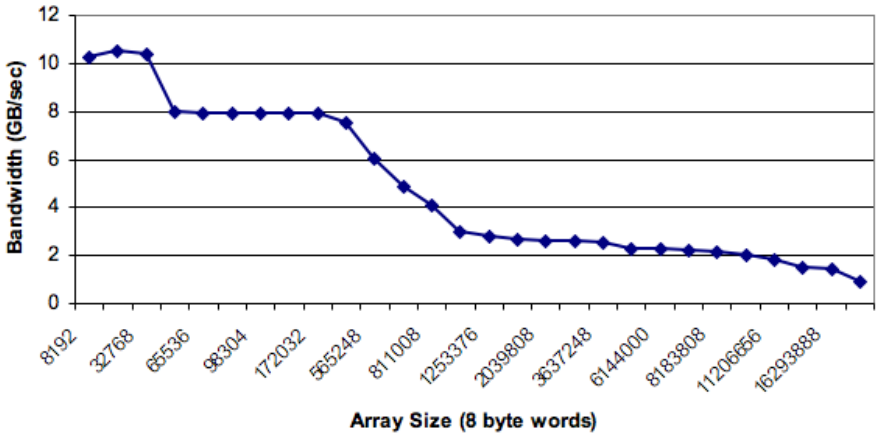

Figure 4: MAPS bandwidth measurements (in Gigabytes/second) for an IBM p690 node as a function of array size.

The benchmark metrics used for the examples in this chapter are summarized in Table 2.

Abbreviation	Description	Benchmark Suite
L1 (s)	Bandwidth of strided accesses to L1 cache	MAPS
L1 (r)	Bandwidth of random accesses to L1 cache	MAPS
L2 (s)	Bandwidth of strided accesses to L2 cache	MAPS
L2 (r)	Bandwidth of random accesses to L2 cache	MAPS
L3 (s)	Bandwidth of strided accesses to L3 cache	MAPS
L3 (r)	Bandwidth of random accesses to L3 cache	MAPS
MM (s)	Bandwidth of strided accesses to main memory	MAPS
MM (r)	Bandwidth of random accesses to main memory	MAPS
NW bw	Bandwidth across interprocessor network	Netbench
NW lat	Latency for interprocessor network	Netbench
FLOPS	Floating point operations per second	HPCC

Table 2: Benchmark metrics used for the examples in this chapter.

4.2 Applications

The applications used in this chapter are from the Department of Defense's Technical Insertion 2006 (TI-06) program [13]. The following are short descriptions of the eight applications:

AVUS: Developed by the Air Force Research Laboratory, AVUS is used to determine the fluid flow and turbulence of projectiles and air vehicles. The parameters used calculates 100 time-steps of fluid flow and turbulence for a wing, flap, and end plates using 7 million cells.

CTH: The CTH application measures effects of multi-material, large deformation, strong shock wake, solid mechanics and was developed by the Sandia national Laboratories. CTH models multi-phase, elastic viscoplastic, porous and explosive materials on 3D and 2D rectangular grids, as well as 1D rectilinear, cylindrical, and spherical meshes.

GAMESS: Developed by the Gordon research group at Iowa State University, GAMESS computes *ab initio* molecular quantum chemistry.

HYCOM: HYCOM models all of the world's oceans as one global body of water at a resolution of one-fourth of a degree measured at the Equator. It was developed by the Naval Research Laboratory, Los Alamos National Laboratory and the University of Miami.

LAMMPS: Developed by the Sandia National Laboratories, LAMMPS is generally used as a parallel particle simulator for particles at the mesoscale or continuum levels.

OOCORE: An out-of-core matrix solver, OOCORE was developed by the SCALAPACK group at the University of Tennessee at Knoxville. OOCORE has been included in past benchmark suites and is typically I/O bound.

OVERFLOW: NASA Langley and NASA Ames developed OVERFLOW to solve CFD equations on a set of overlapped, adaptive grids, so that the resolution near an obstacle is higher than other portions of the scene. With this approach, computations of both laminar and turbulent fluid flows over geometrically complex non-stationary boundaries can be solved.

WRF: A weather forecasting model that uses multiple dynamical cores and a 3D variational data assimilation system with the ability to scale to many processors. WRF was developed by a partnership between the National Center for Atmospheric Research, the National Oceanic and Atmospheric Administration, the Air Force Weather Agency, the Naval Research Laboratory, Oklahoma University, and the Federal Aviation Administration.

These eight applications were each run multiple times with a variety of processor counts ranging from 16 to 384 on the HPC systems summarized in Table 1. Each application was run using the DoD "standard" input set. Each application was run on no fewer than 10, and no more than 19, of the machines. Sometimes applications were not run on particular machines with particular processor counts either because those systems lacked the required number of processors or because the amount of main memory was insufficient. But, more generally, the examples in this chapter were meant to reflect real world conditions and, in the real world, it is not unusual for timings that have been collected at different times on different machines by different people to be incomplete in this way.

At a minimum, for each run, the end-to-end runtime was collected. This is the cheapest data to collect, and is the only information used by the methods discussed in Section 5. However, in some cases trace information was also collected and used, in varying levels of detail, for the methods described in Sections 6 and 7.

In addition to counting the number of FLOPS and memory accesses, some of the methods required partitioning the memory accesses between strided and random accesses. Since there is a standard understanding of what it means to count the total number of memory accesses in an application, but not of what it means to partition memory accesses into strided and random, a little more detail is presented on how this was done.

These examples categorize memory accesses by using the Metasim tracer [8], which partitions the code for an application into non-overlapping basic blocks. Each block is then categorized as exhibiting either primarily strided or primarily random behavior using both dynamic and static analysis techniques. For the examples in this chapter, if either method classifies the block as containing at least 10% random accesses, all memory accesses in that block are counted as random. While the number 10% is somewhat arbitrary, it is based on the observation that on many machines the sustainable bandwidth of random accesses is less than the sustainable bandwidth of strided accesses by an order of magnitude.

The dynamic method for determining if a block exhibits primarily random or strided behavior uses a trace of the memory accesses in each basic block and

considers each access to be strided if there has been an access to a sufficiently nearby memory location within some small number of immediately preceding memory accesses. The advantage of a dynamic approach is that every memory access is evaluated, so nothing is overlooked. The disadvantage is that the number of preceding accesses considered must be chosen carefully. If the size is too small, some strided accesses may be misclassified as random. If the size is too large, the process becomes too expensive computationally. In contrast, the static analysis method searches for strided references based on an analysis of dependencies in the assembly code. Static analysis is less expensive than dynamic analysis and also avoids the potential for misclassifying accesses due to a window size that is too small. On the other hand, static analysis may miss some strided accesses because of the difficulty of analyzing some types of indirect accesses. Since the two types of analysis are predisposed to misclassify different types of strided accesses as random, both methods are applied and an access is considered to be strided if either method classifies it as such.

5 Using end-to-end runtimes

If the only information available for the applications are end-to-end runtimes on some set of machines, then the weights in Equation 1 can be estimated by finding $\boldsymbol{w}$ in the equation $M \times \boldsymbol{w} = \boldsymbol{P}$, where M is a matrix containing the benchmark measurements for a set of machines. Written out in matrix form, the equation looks as follows:

$$\begin{pmatrix} m_{1,1} & ... & m_{1,b} \\ m_{2,1} & ... & m_{2,b} \\ m_{3,1} & ... & m_{3,b} \\ . & & . \\ . & ... & . \\ . & & . \\ m_{n,1} & ... & m_{n,b} \end{pmatrix} \times \begin{pmatrix} w_1 \\ w_2 \\ w_3 \\ ... \\ w_b \end{pmatrix} = \begin{pmatrix} p_1 \\ p_2 \\ p_3 \\ . \\ . \\ . \\ p_n \end{pmatrix} \tag{3}$$

Again, the matrix M contains all the benchmark measurements on all the machines. Each row represents a single machine and each column a particular benchmark. Hence, for the M in Equation 3, there are benchmark measurements for b resources on n machines. The vector $\boldsymbol{P}$ contains the end-to-end, measured runtimes for the target application on the n machines in M. (Note the assumption that each of these times was gathered running on the same number of processors, on the same input.) Furthermore, in addition to assuming that the linear model described in Section 3.2 holds, the assumption is also made that the weight vector $\boldsymbol{w}$ varies relatively little across machines. Obviously this is a simplifying assumption which affects the accuracy of the predictions; the techniques in Section 6 relax this assumption.

Some basic preprocessing is done before proceeding with any of the techniques described here. First, since all measurements must be in the same units,

the inverse of all non-latency measurements (e.g., FLOPS, memory and network bandwidths, etc.) must be used. Next, the measurements in M are normalized by scaling the columns of M so that the largest entry in each column is 1. This allows us to weight different operations similarly, despite the fact that the cost of different types of operations can vary by orders of magnitude (e.g., network latency versus time to access the L1 cache).

After the preprocessing step, the weights for the benchmark measurements can be estimated by finding the "best" $\boldsymbol{w}$ in $M \times \boldsymbol{w} = \boldsymbol{P}$ (Equation 3). If $n < b$, then the system is underdetermined and there is not enough data to choose a single "best" value for $\boldsymbol{w}$. As a result, this technique assumes that $n \geq b$. Since this means that equality in Equation 3 may not be achievable, the metric for evaluating the quality of any given value of $\boldsymbol{w}$ must be specified.

Finally, having obtained $\boldsymbol{w}$, it can be used to make predictions for a new set of machines which have benchmark measurements captured in a matrix M_{new}. The runtime predictions for the application on the new set of machines is then $\boldsymbol{P_{new}} = M_{new} \times \boldsymbol{w}$. Machine ranking can be done on the new machines by sorting their respective application runtimes in $\boldsymbol{P_{new}}$.

5.1 Basic least squares

Perhaps the most natural way to find $\boldsymbol{w}$ is by using a least squares regression, also known as solving the least squares problem, which computes the $\boldsymbol{w}$ that minimizes the 2-norm of the residual $r = \boldsymbol{P} - M \times \boldsymbol{w}$.

With this technique the entire set of machine benchmarks given in Table 2, also referred to here in this chapter as "the full basis set", is used. This set consists of:

$$< [FLOPS], L1(s), L2(s), L3(s), MM(s), L1(r), L2(r), L3(r), MM(r), NWbw, NWlat >$$

FLOPS is in brackets because sometimes it is included in the full basis set and sometimes not. The distinction should always be clear from the context.

5.1.1 Results for performance prediction

The results shown in Table 3 were computed using a tool that reads the benchmark data and application runtimes for a set of machines and performs the above analysis [25]. Because of the requirement that the number of machines be no less than the number of machine benchmarks, some entries in the table could not be computed.

Using just the MAPS and Netbench measurements, the performance predictions are worse than FLOPS alone. Using a combination of the FLOPS, MAPS and Netbench measurements, however, provides generally better performance predictions than FLOPS alone. But regardless of the set of benchmarks used, the performance predictions are poor, with average errors ranging from 51.6% to 72.4%.

	FLOPS	MAPS + Netbench	FLOPS + MAPS + Netbench
avus	73.9	213.4	–
cth	61.2	64.8	59.2
gamess	72.6	–	–
hycomm	67.6	70.8	–
lammps	58.4	54.3	84.8
oocore	62.9	33.3	40.2
overflow	74.8	23.2	28.6
wrf	58.2	69.6	45.2
Average	66.2	72.4	51.6

Table 3: Average absolute relative error for TI-06 applications using FLOPS, the full MAPS+Netbench set and the full FLOP+MAPS+Netbench set.

5.1.2 Results for ranking

The least squares performance prediction method can be extended to make predictions for several machines at a time, then to rank the machines based on the predicted performance using the methodology described in Section 3.4.

In Table 4 the average number of thresholded inversions is reported. For each entry 5 machines are chosen at random for ranking, hence the maximum number of inversions is 10.

	FLOPS	MAPS + Netbench	FLOPS + MAPS + Netbench
cth	2.9	–	–
lammps	2.9	3.6	–
oocore	3.1	3.1	2.9
overflow	3.3	2.2	2.2
wrf	3	3.6	–
Average	3.0	3.2	2.6

Table 4: Average number of thresholded inversions for TI-06 applications using FLOPS, the full MAPS+Netbench set and the reduced MAPS+Netbench set. ($\alpha = .01$ and $\beta = .001$)

Using the MAPS and Netbench measurements provides more thresholded inversions in these tests than FLOPS alone, similar to how its performance prediction were worse. However when including FLOPS with the MAPS and Netbench measurements, not only did the accuracy of the performance predictions generally improve, but the rankings also became more accurate. Taken together, this suggests that FLOPS cannot be completely ignored for accurate performance predictions.

5.2 Least squares with basis reduction

One drawback of applying least squares in such a straightforward way to solve Equation 3 is that the benchmark measurements may not be orthogonal. In other words, if M contains several benchmarks whose measurements are highly correlated, the redundant information may have an unexpected effect on the accuracy of predicted runtimes on new systems.

This redundant information can be removed using the method of *basis reduction* [25]. After computing the correlations of all pairs of benchmark measurements across all the machines in M, highly correlated pairs are identified and one of each pair is dropped. For the purposes of this chapter, pairs are considered to be highly correlated if the correlation coefficient between them is greater than 0.8 or less than -0.8.

In the cross-validation tests, basis reduction is run on M after the inverse of non-latency measurements is taken, but before the columns are normalized. M is reduced to M_r, where M_r has equal or fewer columns than M, depending on the correlation coefficients of the columns in M. From here, M_r is normalized and the cross-validation tests are conducted in the same way as before, using M_r instead of M.

5.2.1 Results for prediction

Since the applications were run on subsets of the single set of 20 machines, basis reduction was run on all machines, instead of for each application's set of machines. Recall that the full basis set consisted of the following:

$$< [FLOPS], L1(s), L2(s), L3(s), MM(s), L1(r), L2(r), L3(r), MM(r), NWbw, NWlat >$$

On the machines in Table 1, measurements for strided access to L1 and L2 caches were highly correlated, and only L1-strided bandwidths were kept. In addition, both strided access to L3 cache and main memory along with random access to L3 cache and main memory were highly correlated, and only main memory measurements were kept. The correlation between L3 cache and main memory is not surprising since not all of the systems have L3 caches. As a result, when the human expert looked at the MAPS plot (as described in Section 4.1) and had to identify a particular region as representing the L3 cache, the region chosen tended to have very similar behavior to that of the region identified for the main memory.

While different combinations of eliminated measurements were tested, in practice different combinations led to only minor differences in the results. As a result, the first predictor in each highly correlated pair was always dropped. This led to a reduced basis set consisting of:

$$< [FLOPS], L1(s), MM(s), L1(r), L2(r), MM(r), NWbw, NWlat >$$

Note that, in addition to using just the MAPS and Netbench measurements, the FLOPS measurement was also included in the full basis set. FLOPS was

not correlated with any of the other measurements, and so its inclusion had no effect on whether other metrics were included in the reduced basis set.

Table 5 presents the prediction results using the least squares solver (results in bold are the most accurate predictions in their row). Reducing the full set of measurements helps provide more accurate performance predictions for all applications with and without including FLOPS in the basis set.

	FLOPS	MAPS + Netbench		FLOPS + MAPS + Netbench	
		Full	Reduced	Full	Reduced
avus	73.9	213.4	**43.5**	–	60.4
cth	61.2	64.8	36.0	59.2	**33.4**
gamess	72.6	–	55.0	–	**35.2**
hycomm	67.6	70.8	60.4	–	**58.4**
lammps	58.4	54.3	**32.4**	84.8	34.6
oocore	62.9	33.3	**24.0**	40.2	27.4
overflow	74.8	**23.2**	27.9	28.6	31.9
wrf	58.2	69.6	17.4	45.2	**16.6**
Average	66.2	72.4	**37.1**	51.6	42.4

Table 5: Average absolute relative prediction error for TI-06 applications using FLOPS, the full MAPS+Netbench set with and without FLOPS and the reduced MAPS+Netbench set with and without FLOPS.

5.2.2 Results for ranking

The average number of inversions for each application is presented in Table 6. The table shows that using FLOPS alone provides the best ranking only for CTH. Although using the reduced FLOPS, MAPS and Netbench basis set did not provide the better performance predictions, it provides the best rankings, on average, for the systems. Moreover, whether or not FLOPS is included in the full basis set, after basis reduction the reduced basis set always provides more accurate rankings than the full one.

Table 7 shows the effects on the number of inversions as α and β are varied. This is only shown for the case where the MAPS and Netbench measurements, not including FLOPS, are used. As α and β get larger, larger variances in the collected runtimes and benchmark measurements are considered insignificant. As a result, the number of thresholded inversions declines. Nonetheless, when averaged over all applications, using the reduced basis set always provides more accurate rankings compared to FLOPS alone or the full basis set.

5.3 Linear Programming

In [41] a method that uses linear programming to solve Equation 1 is described. This method uses no more benchmark or application information than the least

	FLOPS	MAPS + Netbench		FLOPS + MAPS + Netbench	
		Full	Reduced	Full	Reduced
cth	**2.9**	–	3.2	–	3.4
lammps	2.9	3.6	3.1	–	**2.8**
oocore	3.1	3.1	2.4	2.9	**2.4**
overflow	3.3	2.2	2.1	2.2	**2.0**
wrf	3.0	3.6	1.9	–	**1.8**
Average	3.0	3.2	2.6	2.6	**2.5**

Table 6: Average number of thresholded inversions for TI-06 applications using FLOPS, the full MAPS+Netbench set and the reduced MAPS+Netbench set. ($\alpha = .01$ and $\beta = .001$)

(α, β)	$(.01, .001)$	$(.1, .01)$	$(.2, .02)$	$(.5, .05)$
FLOPS	3.0	2.5	2.2	1.7
Full Basis Set	3.2	2.9	2.8	2.3
Reduced Basis Set	2.6	2.3	2.1	1.6

Table 7: The number of thresholded inversions, averaged over all applications, with different values of α and β for the TI-06 applications using the MAPS and Netbench benchmarks.

squares methods described previously, but it adds the ability to incorporate human judgement and so demonstrates the impact that expert input can have.

The basic idea is to add a parameter γ to Equation 1 so that it is changed from:

$$P \approx m_1 w_1 + m_2 w_2 + m_3 w_3, \tag{4}$$

to:

$$P(1-\gamma) \geq m_1 w_1 + m_2 w_2 + m_3 w_3 \tag{5}$$

$$P(1+\gamma) \leq m_1 w_1 + m_2 w_2 + m_3 w_3. \tag{6}$$

The goal is to find non-negative weights w that satisfy the above constraints, while keeping γ small. When some constraints are determined to be difficult to satisfy, a human expert can decide that either the end-to-end application runtime measurement, or the benchmark measurements, are suspect and simply eliminate that machine and the corresponding two constraints. Alternatively, the decision could be made to rerun the application and/or benchmarks in the hopes of getting more accurate measurements.

5.3.1 Results

The linear programming method tries to find the weights that best fit the entries in M and P under the same assumptions as with the least squares methods.

However, with human intervention, it can also identify entries in M and P that seem suspect and either ignore or correct them.

On the set of test data used here, a few runtimes and benchmark measurements were found to be suspect. After correcting those errors, the linear programming method was run again, giving the overall results presented in Table 8. The fact that these predictions are so much more accurate than those in Table 5 using the least squares method reflects the power of allowing a human expert to examine the results and to eliminate (or to rerun and recollect) suspicious benchmark measurements and application runtimes. Significantly more detail and analysis can be found in [41].

System	Average Error
ASC_SGI_Altix	8%
SDSC_IBM_IA64	—
ARL_IBM_Opteron	8%
ARL_IBM_P3	4%
MHPCC_IBM_P3	6%
NAVO_IBM_P3	6%
NAVO_IBM_p655 (Big)	6%
NAVO_IBM_p655 (Sml)	5%
ARSC_IBM_p655	2%
MHPCC_IBM_p690	7%
NAVO_IBM_p690	9%
ARL_IBM_p690	6%
ERDC_HP_SC40	8%
ASC_HP_SC45	4%
ERDC_HP_SC45	6%
ARSC_Cray_X1	5%
ERDC_Cray_X1	3%
AHPCRC_Cray_X1E	—
ARL_LNX_Xeon (3.06)	8%
ARL_LNX_Xeon (3.6)	8%
Overall Average Error	**6%**

Table 8: Average absolute error over all applications using linear programming for performance prediction. The systems are identified by a combination of the Department of Defense computer center, the computer manufacturer, and the processor type.

5.4 Discussion

A simple approach to obtaining benchmark weights is to use least squares. Using this method is quick and simple, assuming that end to end runtimes of

the application on different machines, along with the results of simple machine benchmarks, are available. Although the the accuracy of the performance prediction (at best a relative error of 37% averaged over all TI-06 applications) may be insufficient for scenarios such as a queue scheduler, they are accurate relative to each other and so can be useful for ranking a set of machines.

Simply using the full set of benchmark measurements is not the best approach for the least squares method. For example, when using the full set of MAPS and Netbench measurements, the average relative error for predictions was as high as 72.4%, and there were an average of 3.2 thresholded inversions when ranking a set of 5 systems. But, once the full set of measurements were reduced using basis reduction to an orthogonal set, the performance predictions improved to an average relative error of 37% and the thresholded inversions reduced to an average of 2.6. In all cases but one, using the reduced set of benchmark measurements for making performance predictions and ranking systems is better than using FLOPS alone. The only exception to this is ranking systems for CTH.

The combination of MAPS and Netbench measurements with and without FLOPS perform similarly: the performance predictions are better by 5% when FLOPS are not included in the set. Although the average number of thresholded inversions are lower when FLOPS is included in the set, the difference between the two is quite small.

The linear programming method used exactly the same information about the machines and the applications as the least squares methods, but added the ability to factor in human expertise. This improved the results significantly, indicating the power of having access to human expertise (in this case, the ability to throw out results judged to be due to errors in either the benchmark measurements or the measured application runtimes).

6 Using basic trace data

As noted previously, different applications with the same execution time on a given machine may stress different system components. As a result, applications may derive varying levels of benefit from improvements to any single system component. The techniques described in Section 5 do not take this into consideration.

In contrast, this section discusses some techniques that incorporate more application-specific information, in particular the lightweight trace data described in Section 4.2. Note that the types of traces used are considerably cheaper than those used for the cycle-accurate simulations described in Section 2.4.

6.1 Predicting performance

In [41] the authors describe methods for performance prediction that assume both M and w in Equation 3 are known, but that allow the operation for

combining them to be significantly more complex than matrix multiplication. In addition, whereas M is determined as before, w is determined by tracing the application. Most notably, w is allowed to vary depending on the system. To keep costs down, the application is only traced on a single system, but the data collected (summarized in Figure 5) is then simulated on other systems of interest in order to generate w for those other machines.

A detailed description of the technique can be found in [41]; here it suffices to note simply that the results they attained (presented in Table 9) are quite accurate, with an average absolute error of under 10%.

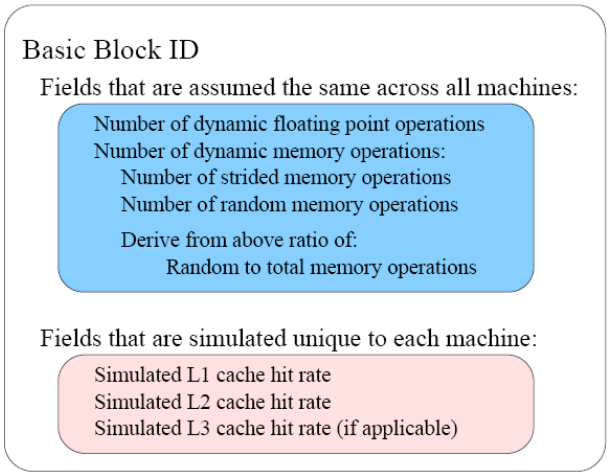

Figure 5: Data collected on each basic block using the MetaSim Tracer.

Systems	average error
ARL_IBM_Opteron	11%
NAVO_IBM_P3	7%
NAVO_IBM_p655 (Big)	6%
ARSC_IBM_p655	4%
MHPCC_IBM_p690	8%
NAVO_IBM_p690	18%
ASC_HP_SC45	7%
ERDC_HP_SC45	9%
ARL_LNX_Xeon (3.6)	5%
Overall Average Error	**8%**

Table 9: Absolute error averaged over all applications.

6.2 Ranking

As in Section 5, once performance predictions have been made for each system, the times can be used to generate a machine ranking. In [7] the quality of performance predictions using 9 methodologies of different sophistication are evaluated. Using the methodologies and the data from [7], Table 10 gives the summed number of thresholded inversions in the predicted runtimes.

The first 3 cases should produce rankings that are equivalent to rankings based on only FLOPS, only the bandwidth of strided accesses to main memory, and only the bandwidth of random accesses to main memory, respectively. The ranking based on the bandwidth of strided access to main memory is the best of these three. As expected from the description in [7], the first and fourth cases produce equivalent rankings.

Case 5 is similar to a ranking based on a combination of FLOPS and the bandwidth of strided and random accesses to memory. Case 6 is similar to Case 5, but with a different method for partitioning between strided and random accesses. In Table 10, both of these rankings are significantly better than those produced by the first four cases. As the rankings in these cases are application dependent, it is not surprising that they outperform the application independent rankings discussed in Section 7.

Cases 7 through 9 use more sophisticated performance prediction techniques. These calculations consider combinations of FLOPS, MAPS memory bandwidths (case 7), Netbench network measurements (case 8) and loop and control flow dependencies for memory operations (case 9). As expected, they result in more accurate rankings.

Methodology (case #)	1	2	3	4	5	6	7	8	9
# thresh. inversions	165	86	115	165	76	53	55	44	44

Table 10: Sum of the number of thresholded inversions for all applications and numbers of processors, for each of the nine performance prediction strategies described in [7].

6.3 Discussion

Although the methods in this section do not use any information about the actual end-to-end runtimes of an application across a set of machines, the trace information that these methods employ instead allows them to achieve high accuracy. This represents another point on the trade-off line between accuracy and expense/complexity. As with the other, more detailed, model-based methods summarized in Section 2, this approach constructs an overall application performance model from many small models of each basic block and communications event. This model can then be used to understand where most of the time is spent and where tuning efforts should be directed. The methods described in Section 5 do not provide such detailed guidance.

Furthermore, generating more accurate predictions using these methods also gives improved (application dependent) machine rankings.

7 Application-independent rankings

Thus far the techniques discussed have focussed on application-specific performance prediction and rankings. When it comes to ranking machines, this means one is given a set of machines and a specific application, and the goal is to predict which of those machines will execute the application fastest. However, there is also interest in application-independent rankings (e.g., the Top 500 list [45]), in which a set of machines is ranked and the general expectation is that the machine ranked, say, third, will execute most applications faster than the machine ranked, say, tenth.

The Top 500 list ranks supercomputers based solely on their performance on the Linpack benchmark which essentially measures FLOPS. This section studies whether it is possible to improve on that ranking, and what the cost is of doing so.

7.1 Rankings using only machine metrics

With the metric described in Section 3.4 for evaluating the quality of a ranking, it is possible to objectively evaluate how FLOPS compares to other machine benchmarks as a way for generating machine rankings. All rankings in this section are tested on the set of applications described in Section 4.2, run on subsets of the machines described in Section 4.1.

The first experiment considers the quality of rankings generated by the machine benchmarks summarized in Table 2: bandwidth of strided and random accesses to L1 cache, bandwidth of strided and random accesses to L2 cache, bandwidth of strided and random accesses to main memory, interprocessor network bandwidth, interprocessor network latency, and peak FLOPS.

Table 11 sums the number of thresholded inversions over all the applications and all the processor counts on which each was run. Because each application is run on a different set of processor counts and not every application has been run on every machine, the numbers in Table 11 should not be compared across applications, but only on an application by application basis, across the rankings by different machine characteristics.

The last row of Table 11 shows that the bandwidth of strided accesses to main memory provides the single best overall ranking, with 309 total thresholded inversions (in contrast, there is also a machine ranking that generates over 2000 thresholded inversions on this data). The ranking generated by the bandwidth of random accesses to L1 cache is a close second; however, it is also evident that there is no single ranking that is optimal for all applications. Although the bandwidth of strided accesses to main memory is nearly perfect for avus, and does very well on hycom, wrf, and cth7, it is outperformed by the bandwidth of both strided and random accesses to L1 cache for ranking performance on

Metric	L1(s)	L1(r)	L2(s)	L2(r)	MM(s)	MM(r)	1/NW lat	NW bw	FLOPS
avus	51	26	44	42	**1**	61	19	30	22
cth	32	**18**	30	82	21	117	63	37	35
gamess	25	**16**	40	55	48	76	65	35	25
hycom	26	**10**	26	83	17	126	65	28	35
lammps	136	107	133	93	80	157	95	116	**68**
oocore	44	**31**	56	71	61	91	75	50	52
overflow	71	**39**	79	91	47	104	108	81	44
wrf	99	63	92	134	**34**	203	103	83	60
overall sum	484	310	500	651	**309**	935	593	460	341

Table 11: Sum of the number of thresholded inversions ($\alpha = .01, \beta = .001$) for all processor counts for each application. The smallest number (representing the best metric) for each application is in **bold**. The last row is a sum of each column and gives a single number representing the overall quality of the ranking produced using that machine characteristic.

gamess. One interpretation of the data is that these applications fall into three categories:

- codes dominated by time to perform floating-point operations,
- codes dominated by time to access main memory,
- and codes dominated by time to access L1 cache.

With 20 machines, there are 20! possible distinct rankings. Searching through the subspace of "feasible" rankings reveals one that gives only 195 inversions (although this number cannot be directly compared to those in Table 11 since the parameter values used were $\alpha = .01$ and $\beta = 0$). In this optimal ranking the SDSC Itanium cluster TeraGrid was predicted to be the fastest machine. However, across all of the benchmarks, the Itanium is only the fastest for the metric of bandwidth of random access to main memory — and Table 11 shows using random access to main memory alone to be the poorest of the single-characteristic ranking metrics examined. This conundrum suggests trying more sophisticated ranking heuristics.

Testing various simple combinations of machine metrics — for example, the ratio of flops to the bandwidth of both strided and random accesses to different levels of the memory hierarchy — gave rankings that were generally significantly worse than the ranking based solely on the bandwidth of strided accesses to main memory. This suggests a need to incorporate more information.

7.2 Rankings incorporating application characteristics

As in Section 6, one might try improving the rankings by incorporating application characteristics and using those characteristics to weight the measured

machine characteristics. However, in order to generate a single application-independent ranking, this requires either choosing a single representative application, or using values that represent an "average" application. This section considers the former approach.

Since the goal is to use as little information as possible, the first example presented uses only the number of memory accesses m and the number of floating point operations f. Recall that the goal is an application-independent ranking, so while we evaluate the rankings generated by each of the applications, only the best result over all the applications is reported. In other words, this is the result of taking the characteristics of a single application and using it to generate a ranking of the machines that is evaluated for all the applications in the test suite.

If a memory reference consists of 8 bytes, m and f can be used in a natural way by computing the following number for each machine m_i:

$$r_i = \frac{8m}{\text{bw\_mem}(i)} + \frac{f}{\text{flops}(i)}. \tag{7}$$

Within Equation 7, m (and f) could be either the average number of memory accesses over all the processors, or the maximum number of memory accesses over all the processors. In addition, bw_mem can be the strided or random bandwidth of accesses to any level of the memory hierarchy.

Using the bandwidth of strided accesses to main memory for bw_mem, regardless of whether the average or the maximum counts for m and f are used, leads to a ranking that is identical to a ranking based only on the bandwidth of strided accesses to main memory. Since the increase in memory bandwidth has not kept pace with the increase in processor speed, it is not surprising that the memory term in Equation 7 overwhelms the processor term. The effect would be even greater if the bandwidth of random accesses to main memory for bw_mem(i) was used, since the disparity between the magnitude of the two terms would be even greater. Moreover, using the measurement that has the fastest access time of all levels of the memory hierarchy, bandwidth of strided accesses to L1, for bw_mem in Equation 7 results in a ranking that is independent of f.

This suggests a model that accommodates more detail about the application, perhaps by partitioning the memory accesses. One possibility would be to partition m into $m = m_{l1} + m_{l2} + m_{l3} + m_{mm}$, where m_{l1} is the number of accesses that hit in the L1 cache, and so on. Another is to partition m into $m = m_1 + m_r$, where m_1 is the number of strided accesses and m_r is the number of random accesses to memory. Partitioning into levels of the hierarchy is dependent on the architecture chosen, which suggests trying the latter strategy (using the technique described in Section 4 to partition the memory accesses).

Once the m memory accesses into random (m_r) and strided (m_1), Equation 8 can be used to compute the numbers $r_1, r_2, \ldots, r_n$ from which the ranking is generated:

$$r_i = \frac{8m_1}{\text{bw\_mem}_1(i)} + \frac{8m_r}{\text{bw\_mem}_r(i)} + \frac{f}{\text{flops}(i)}. \tag{8}$$

Metric	l1(1,r)	mm(1,r)	mm(1), l1(r)
avus	12	21	**9**
cth7	14	80	**9**
gamess	**16**	77	26
hycom	**2**	44	**2**
lammps	107	148	**81**
oocore	**31**	100	44
overflow2	**34**	78	**34**
wrf	63	158	**44**
overall sum	279	706	**249**

Table 12: Sum of the number of thresholded inversions for all numbers of processors for each application, with $\alpha = .01$ and $\beta = .001$. The smallest number (representing the best metric) for each application is in **bold**.

Notice that there is again a choice to be made regarding what to use for bw_mem_1 and bw_mem_r. There are several options including: using the bandwidths of strided and random accesses to main memory; the bandwidths of strided and random accesses to L1 cache; or, considering the data in Table 11, the bandwidth of strided access to main memory and of random access to L1 cache. Furthermore, since the goal is a single, fixed ranking that can be applied to all applications, a choice also has to made about which application's m_1, m_r, and f to use for generating the ranking. In theory one could also ask what processor count of which application to use for the ranking; in practice, these traces take time to perform, and so m_1 and m_r counts were only gathered for one processor count per application.

Table 12 shows the results of these experiments. Each column shows the number of thresholded inversions for each of the 8 applications using the specified choice of strided and random access bandwidths. In each column the results use the application whose m_1, m_r, and f led to the smallest number of inversions for all other applications. When using the random and strided bandwidths to L1 cache, the most accurate ranking was generated using overflow2; when using the bandwidths to main memory, the best application was oocore; and when using a combination of L1 and main memory bandwidths, avus and hycom generated equally good rankings.

Comparing the results in Table 12 to those in Table 11 reveals that partitioning the memory accesses is useful as long as the random accesses are considered to hit in L1 cache. Using the bandwidth of random access to L1 cache alone did fairly well, but the ranking is improved by incorporating the bandwidth of strided accesses to L1 cache, and is improved even more by incorporating the bandwidth of strided accesses to main memory. When we use the bandwidth of accesses to main memory only, the quality of the resulting order is between those of rankings based on the bandwidth of random accesses and based on the bandwidth of strided accesses to main memory.

In [9] the authors discuss possible reasons why the combined metric based on mm(1) and l1(r) works so well. One observation is that this may be representative of a more general fact: applications with a large memory footprint that have many strided accesses benefit from high bandwidth to main memory because the whole cache line is used and prefetching further utilizes the full main memory bandwidth. For many of these codes main memory bandwidth is thus the limiting performance factor. On the other hand, applications with many random accesses are wasting most of the cache line and these accesses do not benefit from prefetching. The performance of these codes is limited by the latency hiding capabilities of the machine's cache, which is captured by measuring the bandwidth of random accesses to L1 cache.

7.3 Discussion

Two things that might further improve on the ranking would be partitioning memory accesses between the different levels of the memory hierarchy and allowing different rankings based on the processor count. The two possibilities are not entirely independent since running the same size problem on a larger number of processors means a smaller working set on each processor and therefore different cache behavior. However, allowing different rankings for different processor counts takes us away from the original goal of finding a single fixed ranking that can be used as a general guideline.

This leaves partitioning memory accesses between the different levels of the memory hierarchy. As noted previously, this requires either choosing a representative system or moving towards a more complex model that allows for predictions that are specific to individual machines, as is done in [7, 30]. Therefore, given the level of complexity needed for a ranking method that incorporates so much detail, we simply observe that we achieved a ranking with about 28% more thresholded inversions than the brute-force obtainable optimal ranking on our data set without resorting to anything more complex than partitioning each application's memory accesses into strided and random accesses. This represents a significant improvement over the ranking based on FLOPS, which was about 75% worse than the optimal ranking.

8 Conclusion

This chapter addressed two related issues of interest to various parties in the world of supercomputing: performance prediction, and machine ranking. The first is a long-standing problem that has been studied extensively, reflected in part by the survey of work in Section 2. The second, while not as well studied, is still of interest both when the goal is a machine ranking for a particular application, and when the goal is a more general application independent ranking.

To illustrate the trade-offs between accuracy and effort that are inherent in any approach, one framework for both prediction and ranking is presented. The main assumption in this framework is that simple benchmarks can be run

(or accurately estimated) on all systems of interest. Then several variations within the framework are examined: ones that use only end-to-end runtimes for an application on any set of machines (Section 5), and those that also employ basic trace data about an application (Section 6). Using trace data is more expensive and, not surprisingly, gives significantly more accurate predictions than a completely automatic method that is based on least squares and uses only end-to-end runtimes. However, a linear programming method that also only uses end-to-end runtimes can partially compensate by allowing human expert intervention. Finally, in Section 7 the question of application independent machine rankings is addressed, again within the same framework. Once again, reasonable results can be obtained using only the results of simple benchmark measurements, but the results can be improved by incorporating limited application trace information.

9 Acknowledgments

We would like to thank Michael Laurenzano and Raffy Kaloustian for helping to collect trace data; and Xiaofeng Gao for writing the dynamic analysis tool used in Section 4.2. The applications benchmarking data used in this study was obtained by the following members of the Engineering Research and Development Center (ERDC), Computational Science and Engineering Group: Mr. Robert W. Alter, Dr. Paul M. Bennett, Dr. Sam B. Cable, Dr. Alvaro A. Fernandez, Ms. Carrie L. Leach, Dr. Mahin Mahmoodi, Dr. Thomas C. Oppe, and Dr. William A. Ward, Jr. This work was supported in part by a grant from the DoD High Performance Computing Modernization Program (HPCMP) along with HPCMP-sponsored computer time at the Army Research Laboratory (ARL), the Aeronautical Systems Center (ASC), the Engineering Research and Development Center (ERDC), and the Naval Oceanographic Office (NAVO) Major Shared Resource Centers (MSRCs) and the Army High Performance Computing Research Center (AHPCRC), the Artic Region Supercomputing Center (ARSC), and the Maui High Performance Computing Center (MHPCC). Computer time was also provided by SDSC. Additional computer time was graciously provided by the Pittsburgh Supercomputer Center via an NRAC award. This work was supported in part by a grant from the National Science Foundation entitled "The Cyberinfrastructure Evaluation Center", and by NSF grant #CCF-0446604. This work was sponsored in part by the Department of Energy Office of Science through SciDAC award "High-End Computer System Performance: Science and Engineering", and through the award entitled "HPCS Execution Time Evaluation".

References

[1] V. Adve. *Analyzing the behavior and performance of parallel programs.* PhD thesis, University of Wisconsin, Madison, 1993.

[2] R. Badia, G. Rodriguez, and J. Labarta. Deriving analytical models from a limited number of runs. In *Parallel Computing: Software Technology, Algorithms, Architectures, and Applications (PARCO 2003)*, pages 769–776, Dresden, Germany, 2003.

[3] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS parallel benchmarks. *Intl. J. Supercomp. Appl.*, 5(3):63–73, Fall 1991.

[4] R. S. Ballansc, J. A. Cocke, and H. G. Kolsky. *The lookahead unit, planning a computer system.* McGraw-Hill, New York, NY, 1962.

[5] L. T. Boland, G. D. Granito, A. V. Marcotte, B. V. Messina, and J. W. Smith. The IBM system 360/model9: storage system. *IBM J. Res. and Dev.*, 11:54–79, 1967.

[6] D. Burger, T. M. Austin, and S. Bennett. Evaluating future microprocessors: the simplescalar tool set. Technical Report CS-TR-1996-1308, University of Wisconsin-Madison, 1996.

[7] L. Carrington, M. Laurenzano, A. Snavely, R. L. Campbell, Jr., and L. Davis. How well can simple metrics predict the performance of real applications? In *Proceedings of Supercomputing (SC|05)*, November 2005.

[8] L. Carrington, A. Snavely, N. Wolter, and X. Gao. A performance prediction framework for scientific applications. In *Proceedings of the International Conference on Computational Science (ICCS 2003)*, Melbourne, Australia, June 2003.

[9] T.-Y. Chen, M. Gunn, B. Simon, L. Carrington, and A. Snavely. Metrics for ranking the performance of supercomputers. *CTWatch Quarterly*, 2(4B), November 2006.

[10] M. J. Clement and M. J. Quinn. Multivariate statistical techniques for parallel performance prediction. In *HICSS '95: Proceedings of the 28th Hawaii International Conference on System Sciences*, pages 446–455, 1995.

[11] M. E. Crovella and T. J. LeBlanc. Parallel performance prediction using lost cycles analysis. In *Supercomputing '94: Proceedings of the 1994 ACM/IEEE conference on Supercomputing*, pages 600–609, Washington, D.C., 1994.

[12] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, San Diego, CA, May 1993.

[13] Department of Defense High Performance Computing Modernization Program. Technology Insertion - 06 (TI-06). http://www.hpcmo.hpc.mil/Htdocs/TI/TI06, May 2005.

[14] J. Dongarra, P. Luszczek, and A. Petitet. The LINPACK benchmark: past, present and future. *Concurr. Comput. : Pract. Exper.*, 15:1–18, 2003.

[15] M. Faerman, A. Su, R. Wolski, and F. Berman. Adaptive performance prediction for distributed data-intensive applications. In *Supercomputing '99: Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, page 36, Portland, OR, 1999.

[16] J. L. Gustafson and R. Todi. Conventional benchmarks as a sample of the performance spectrum. *J. Supercomp.*, 13(3):321–342, 1999.

[17] A. Hoisie, O. M. Lubeck, and H. J. Wasserman. Performance analysis of wavefront algorithms on very-large scale distributed systems. In *Workshop on Wide Area Networks and High Performance Computing*, pages 171–187, London, UK, 1999. Springer-Verlag. Also Lecture Notes in Control and Information Sciences, Vol. 249.

[18] A. Hoisie, O. M. Lubeck, and H. J. Wasserman. Scalability analysis of multidimensional wavefront algorithms on large-scale SMP clusters. In *FRONTIERS '99: Proceedings of the The 7th Symposium on the Frontiers of Massively Parallel Computation*, pages 4–15, Annapolis, MD, February 1999.

[19] HPC Challenge Benchmarks. http://icl.cs.utk.edu/hpcc/.

[20] IDC reports latest supercomputer rankings based on the IDC Balanced Rating test. In EDP Weekly's IT Monitor, December 2002.

[21] E. Ipek, B. R. de Supinski, M.Schulz, and S. A. McKee. *Euro-Par 2005 Parallel Processing*, volume 3648, chapter An approach to performance prediction for parallel applications, pages 196–205. Springer Berlin / Heidelberg, 2005.

[22] D. J. Kerbyson, A. Hoisie, and H. J.Wasserman. A performance comparison between the Earth Simulator and other terascale systems on a characteristic ASCI workload. *Concurr. Comput. : Pract. Exper.*, 17(10):1219–1238, 2005.

[23] D. J. Kerbyson and P. W. Jones. A performance model of the parallel ocean program. *Intl. J. High Perf. Comput. Appl.*, 19(3):261–276, Summer 2005.

[24] D. J. Kerbyson, H. J. Wasserman, and A. Hoisie. Exploring advanced architectures using performance prediction. In *IWIA '02: Proceedings of the International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems*, pages 27–40, 2002.

[25] O. Khalili. Performance prediction and ordering of supercomputers using a linear combination of benchmark measurements. Master's thesis, University of California at San Diego, La Jolla, CA, June 2007.

[26] W. T. C. Kramer and C. Ryan. Performance variability of highly parallel architectures. In *Proceedings of the International Conference on Computational Science (ICCS 2003)*, Melbourne, Australia, June 2003.

[27] J. Lo, S. Egger, J. Emer, H. Levy, R. Stamm, and D. Tullsen. Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading. *ACM Trans. Comput. Sys.*, August 1997.

[28] Y. Luo, O. M. Lubeck, H. Wasserman, F. Bassetti, and K. W. Cameron. Development and validation of a hierarchical memory model incorporating CPU- and memory-operation overlap model. In *WOSP '98: Proceedings of the 1st International Workshop on Software and Performance*, pages 152–163, Santa Fe, NM, 1998. ACM Press.

[29] P. Luszczek, J. Dongarra, D. Koester, R. Rabenseifner, B. Lucas, J. Kepner, J. McCalpin, D. Bailey, and D. Takahashi. Introduction to the HPC challenge benchmark suite. Available at http://www.hpcchallenge.org/pubs/, March 2005.

[30] G. Marin and J. Mellor-Crummey. Cross-architecture performance predictions for scientific applications using parameterized models. In *Proceedings of SIGMETRICS/Performance'04*, New York, NY, June 2004.

[31] J. D. McCalpin. Memory bandwidth and machine balance in current high performance computers. IEEE Technical Committee on Computer Architecture Newsletter, December 1995.

[32] C. L. Mendes and D. A. Reed. Performance stability and prediction. In *Proceedings of the IEEE/USP International Workshop on High Performance Computing*, 1994.

[33] C. L. Mendes and D. A. Reed. Integrated compilation and scalability analysis for parallel systems. In *PACT '98: Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, pages 385–392, 1998.

[34] J. O. Murphey and R. M. Wade. The IBM 360/195. *Datamation*, 16(4):72–79, 1970.

[35] F. Petrini, D. J. Kerbyson, and S. Pakin. The case of the missing supercomputer performance: achieving optimal performance on the 8,192 processors of ASCI Q. In *Proceedings of Supercomputing (SC'03)*, Phoeniz, AZ, November 2003.

[36] R. H. Saavedra and A. J. Smith. Measuring cache and TLB performance and their effect on benchmark runtimes. *IEEE Trans. Comput.*, 44(10):1223–1235, 1995.

[37] R. H. Saavedra and A. J. Smith. Performance characterization of optimizing compilers. *IEEE Trans. Softw. Eng.*, 21(7):615–628, 1995.

[38] R. H. Saavedra and A. J. Smith. Analysis of benchmark characteristics and benchmark performance prediction. *ACM Trans. Comput. Sys.*, 14(4):344–384, 1996.

[39] J. Simon and J. Wierun. Accurate performance prediction for massively parallel systems and its applications. In *Proceedings of the 2nd International Euro-Par Conference*, Lyon, France, August 1996.

[40] K. Singh, E. Ipek, S. A. McKee, B. R. de Supinski, M. Schulz, and R. Caruana. Predicting parallel application performance via machine learning approaches. 2007.

[41] A. Snavely, L. Carrington, M. M. Tikir, R. L. Campbell Jr., and T.-Y. Chen. Solving the convolution problem in performance modeling. 2006.

[42] A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha. A framework for application performance modeling and prediction. In *Proceedings of Supercomputing (SC2002)*, Baltimore, MD, November 2002.

[43] D. Spooner and D. Kerbyson. Identification of performance characteristics from multi-view trace analysis. In *Proceedings of the International Conference on Computational Science (ICCS 2003)*, Melbourne, Australia, June 2003.

[44] G. S. Tjaden and M. J. Flynn. Detection and parallel execution of independent instruction. *IEEE Trans. Comput.*, C-19:889–895, 1970.

[45] Top500 supercomputer sites. http://www.top500.org.

[46] W. A. Wulf and S. A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Arch. News*, 23(1):20–24, 1995.

[47] Z. Xu, X. Zhang, and L. Sun. Semi-empirical multiprocessor performance predictions. *J. Parallel Distrib. Comput.*, 39(1):14–28, 1996.

[48] L. T. Yang, X. Ma, and F. Mueller. Cross-platform performance prediction of parallel applications using partial execution. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 40, Seattle, WA, 2005.