In Proceedings of Supercomputing '94, *pp. 600-610.*

Parallel Performance Prediction Using Lost Cycles Analysis*

Mark E. Crovella and Thomas J. LeBlanc
Department of Computer Science
University of Rochester
Rochester, New York 14627
(716) 275-5671
{crovella,leblanc}@cs.rochester.edu

Abstract

Most performance debugging and tuning of parallel programs is based on the "measure-modify" approach, which is heavily dependent on detailed measurements of programs during execution. This approach is extremely time-consuming and does not lend itself to predicting performance under varying conditions. Analytic modeling and scalability analysis provide predictive power, but are not widely used in practice, due primarily to their emphasis on asymptotic behavior and the difficulty of developing accurate models that work for real-world programs. In this paper we describe a set of tools for performance tuning of parallel programs that bridges this gap between measurement and modeling.

Our approach is based on lost cycles analysis, *which involves measurement and modeling of* **all** *sources of overhead in a parallel program. We first describe a tool for measuring overheads in parallel programs that we have incorporated into the runtime environment for Fortran programs on the Kendall Square KSR1. We then describe a tool that fits these overhead measurements to analytic forms. We illustrate the use of these tools by analyzing the performance tradeoffs among parallel implementations of 2D FFT. These examples show how our tools enable programmers to develop accurate performance models of parallel applications without requiring extensive performance modeling expertise.*

*This research was supported under NSF CISE Institutional Infrastructure Program Grant No. CDA-8822724, and ONR Contract No. N00014-92-J-1801 (in conjunction with the DARPA HPCC program, ARPA Order No. 8930). Mark Crovella is supported by an ARPA Research Assistantship in High Performance Computing administered by the Institute for Advanced Computer Studies, University of Maryland.

1 Introduction

Traditional performance debugging and tuning of parallel programs is based on detailed measurement of program executions. Programmers typically implement a program, measure its performance in detail, modify the program, and iterate. This "measure-modify" approach to performance tuning [19] results in detailed knowledge about a series of executions of a parallel program. However, programmers would often like to know about other, potential, executions of the program: with varying inputs, on a different number of processors, or on a different machine. Using the measure-modify approach, those executions must be measured also. Unfortunately it may be expensive or impossible to perform those measurements: measurements take time, and machines, data, or processors may be scarce or unavailable.

Performance tuning over a range of machines, data, or processors is important because the best parallelization of an application is often not fixed. Many researchers have noted that the best parallelization for a given application can vary depending on the input, machine, or problem definition [13; 25; 27]. In fact, the best parallelization for a given application can depend on the size of the input dataset, the structure of the input dataset, the specific problem definition, the number of processors used, and the particular machine used [10]. Exploring each of these environmental factors fully requires the predictive power of modeling; it is simply impractical to measure the effects of all these factors after each modification of the application.

Analytic models, such as provided by scalability analysis, provide predictive power but have not been widely used in performance tuning. We feel that modeling is not common in practice because: 1) analytic models emphasize asymptotic performance, minimizing the importance of constants that determine true performance; 2) analytic models often assume in advance that a particular overhead or overheads will dominate over the entire range of study, which may not be the case in practice; and 3) programmers writing parallel applications expect that analytic models are too difficult to develop in practice.

In this paper we describe a set of tools and associated techniques that address these three impediments to using performance prediction for parallel programs. Our goals in developing our performance prediction tools and techniques are:

- sufficient accuracy to allow the user to select among alternative implementations;
- applicability to applications written in any language or programming style, and for a wide range of machines; and
- utility to programmers who are familiar with their applications, but are not experts in the area of performance modeling.

To address these goals, we are developing a performance prediction method called *lost cycles analysis*, along with a supporting tool set. Lost cycles analysis is based on the observation that the distinction between *productive computation* and *parallel overhead* is useful both for performance diagnosis and for performance prediction. Furthermore, decomposing overhead into *categories* is similarly useful for both diagnosis and prediction. The core of lost cycles analysis is a careful breakdown of overheads into categories that can be separately modelled.

We describe two tools to support lost cycles analysis: `pp` and `lca`. The role of `pp` is to accurately measure parallel overheads and attribute them to the categories we have defined. Although we show that the output of `pp` can provide insight to parallel programmers in its own right, our main use of `pp`'s output is as input to the tool `lca`. The role of `lca` is to guide the user in fitting performance models to the data output from `pp`. The output from `lca` forms the basis for the performance model of the application.

2 Lost Cycles Analysis

To predict performance, we must be able to predict two quantities: total parallel overhead T_o, and pure computation T_c, as functions of all environment variables. Given T_o and T_c, we can predict running time as $T_p = (T_o + T_c)/p$. As long as the parallel algorithm is a parallelization of the best serial algorithm, pure computation is equal to serial running time.

Once we have an accurate measurement for T_o we can obtain $T_c = pT_p - T_o$. We measure T_o by decomposing it into categories. In order for measurements of overhead categories to be useful, all categories must be measured using the same metric. We call this metric *lost cycles.* Lost cycles are simply aggregate seconds of parallel overhead, attributed to various categories. Lost cycles is an important notion because it allows us to quantitatively study tradeoffs among effects such as serial fraction, synchronization, communication, and contention that are often measured and modeled in incompatible ways. The portion of the execution time not consumed in lost cycles we refer to as *pure computation.*

Measuring lost cycles directly for the entire environment space is still impractical, but if the categories are chosen properly, modeling them as a separate function of each environment variable is feasible. A small number of measurements for each environment variable will then suffice to parameterize the models, leading to an aggregate model of performance prediction spanning the entire environment space.

Given performance models for each implementation that span the entire environment space, a number of performance tuning problems can be addressed. When an application is to be ported, or run on a different kind of data set, or run on a different number of processors, the performance models can be quickly reduced to functions of the environment variable(s) of interest. The crossover boundaries at which one implementation outperforms another are then obtained by directly solving the performance functions as simultaneous equations. In the context of a parallel programming environment these performance models would be associated with their implementations, for ready use as implementation-selection decisions arise.

The core of our approach is the proper selection of categories. To be successful, lost cycles must be allocated to a set of categories that together meet three criteria:

1. **Completeness.** The categories must capture *all* sources of overhead.
2. **Orthogonality.** The categories must be mutually exclusive.
3. **Meaning.** The categories must correspond to states of the p execution that are meaningful for analysis.

Although often overlooked, completeness is a crucial criterion. Completeness ensures that we do not ignore any overheads as we vary environment variables, regardless of whether we expect them to be dominant. Completeness is not often achieved in performance measurement tools; many tools concentrate on specific performance metrics such as cache miss rates, message traffic, and execution profiles. Each of these tools is useful as long as the tool's metric corresponds to a dominant source of overhead. However, predicting which category of overhead is dominant in all cases is a very difficult task — it is not uncommon that performance is dominated by unexpected effects.

Completeness is also rarely achieved because it requires measurement of effects that occur at different levels — application (e.g., load imbalance) and hardware (e.g., resource contention). A system that attempts to measure lost cycles therefore must be able to instrument the application, as well as have access to hardware performance data.

Completeness and orthogonality together ensure that we can correctly measure lost cycles, and indirectly, pure computation. Completeness ensures that measurements of pure computation are accurate. Orthogonality ensures that we can subtract overheads from running time to calculate pure computation in the natural way.

Meaningfulness of categories serves a rather different purpose, and makes the choice of categories somewhat more difficult. Categories must be meaningful so that they are likely to be amenable to simple analysis. It is certainly possible to define a set of overhead categories that are complete and orthogonal, but without meaning for analytic purposes — the simplest such set would contain one category for all lost cycles. Thus the challenge in defining a category set is in dividing overheads finely enough that they can be analyzed simply, but not so finely as to present problems in measurement, or in verifying completeness and orthogonality.

Meaningfulness of categories also allows the programmer to relate the measurements made to the program being studied. Categories that are amenable to analysis tend to correspond in simple ways to the structure of the program. As a result, measurements of meaningful categories can provide significant performance tuning assistance even apart from their use in analytic models.

The category set we use in this paper is:

Load Imbalance (LI): processor cycles spent idling, while unfinished parallel work exists.

Insufficient Parallelism (IP): processor cycles spent idling, while no unfinished parallel work exists.

Synchronization Loss (SL): processor cycles spent acquiring a lock, or waiting in a barrier.

Communication Loss (CL): processor cycles spent waiting while data moves through the system.

Resource Contention (RC): processor cycles spent waiting for access to a shared hardware resource.

This category set has satisfied the three criteria (completeness, orthogonality, meaning) for the applications we have studied. Of course, it will need to be expanded to handle a wider range of overheads as it is used in more varied situations. In particular, it does not currently distinguish between synchronization types, measure contention for software resources, or measure operating system and runtime library effects. Each of these extensions appears to be straightforward within the existing framework however.

3 Tools for Lost Cycles Analysis

3.1 Measuring Lost Cycles: `pp`

In previous work [8] we showed that basing measurement on logical expressions that recognize lost cycles is a particularly useful approach. We call these expressions *performance predicates*. The use of performance predicates to specify categories of lost cycles makes program instrumentation straightforward, and allows *predicate profiles* to be constructed based on user demands. For example, using predicate profiling, the user can ask for a breakdown of lost cycles by processor number, task, or procedure. The current implementation uses predicate profiling of event logs, rather than the runtime profiling used in our earlier work.

Our current implementation of the predicate profiler, `pp`, measures Fortran programs running on the Kendall Square KSR1, and consists of 1) a library linked into the executable code and 2) a post-processor of event logs that outputs the profile. The KSR Fortran runtime system can log events such as the start and end of individual loop iterations, which we use for calculating load imbalance. Additional calls to our library routines are inserted at the start and end of parallel loops, parallel tasks, and synchronization operations. The inserted library calls are quite simple and could easily be added by a source-to-source preprocessor.

The KSR1 [17] is a two-level ring architecture in which all memory is managed as a cache, which is organized in two levels on each node. Thus, inter-node communication occurs only as the result of misses in the secondary cache. Dedicated hardware monitors the state of buses between the processor and the second-level cache. This performance monitor counts the number of secondary cache misses, the time taken to service secondary cache misses, and the number of cache lines that passed through the higher-level ring before arrival. Based on this data, we can calculate the amount of communication performed in an execution and the amount of resource contention that occurred.

Communication loss is measured as a product of the number of cache misses and the ideal time to perform the cache line transfers. Resource contention (contention for the ring interconnect and for remote memories) is measured as in [28] — that is, the ideal time to perform the communication operations is compared to the actual elapsed time. Since the KSR1 hardware monitors both the number of cache lines transferred and the elapsed time waiting for cache lines, this calculation is straightforward. Although the performance monitoring hardware on the KSR is rather unique, something comparable may be required for other cache-coherent architectures. On simpler architectures, such as a message-passing system, the performance monitoring capabilities of the DEC Alpha [12] should be sufficient to gather the same information.

pp is currently installed for use by the user community at the Cornell Theory Center on their 128-node KSR1. Example output from the current version of pp is shown in Figure 1. Lost cycles for each category are presented in seconds, aggregated over all processors. Actual execution time of the application was 9.86 seconds, which is equal to total time (49.29 seconds) divided by the number of processors (5). In this execution, pp has identified that the primary bottleneck is a serial section of code. Although this implementation does not support it, a simple modification to the profiler would periodically sample the value of the program counter, allowing the bottleneck to be associated with a particular section of code.

```
% f77 -o runfast myprog.f -lctc -lpmon
% runfast
% pp
  PP version 4.0
  ** processors: 5
  Load Imbalance              0.180677
  Insuff Parallelism         16.813304
  Synchronization Loss        0.003779
  Communication Loss          2.274899
  Resource Contention         1.351362
  Total Time                 49.293890
  Remaining Time             28.669868
```

Figure 1: Example Output of pp

3.2 Analyzing Lost Cycles: lca

lca is a tool that manages performance data and focuses the user on a selection of appropriate models for each category of lost cycles. It assists the user in two ways:

1. It guides the user's selection of models for each category and environment variable, using defaults based on our experience in modeling the categories of lost cycles. For example, when varying data set size, the model for communication loss defaults to a linear function of data size. Likewise, when varying the number of processors, the model for insufficient parallelism loss defaults to a linear function of number of processors.

2. It provides error estimates for the goodness-of-fit for each of the default models, and for any models explicitly requested by the user. This gives the user the opportunity to compare the quality of each of the default models and to additionally compare any models the user feels might be better than the defaults. To assist in selecting the best fit, lca can also output the raw data and the fitted models in graphical format, using GNUPLOT.

An example output from lca is shown in Figure 2. This example first shows the collection of a number of predicate profiles for the program runfast, in which the number of processors used (p) is varied (2, 4, 6, and 8). The resulting datafile (in a form output by pp especially for use by lca) is then processed by lca. The arguments given to lca specify that we are interested in selecting a model for Load Imbalance, while varying p. The tool has 3 default models that describe how Load Imbalance often varies with p: $p\sqrt{p}$, p, and null (independent of p). The R^2 column shows that $p\sqrt{p}$ is the best-fitting model, and the form column indicates that the coefficient based on a least squares fit of these data is 0.000298.

```
% runfast -p 2
% pp -l >> datafile
% runfast -p 4
% pp -l >> datafile
% runfast -p 6
% pp -l >> datafile
% runfast -p 8
% pp -l >> datafile

% lca datafile -var p -cat li
(0.000298 +/- .000017) *p*sqrt(p)  R2: .975
(0.149540 +/- .007899) *p          R2: .567
(12.9491 +/- 10.7811)              R2: .002
```

Figure 2: Example Output of `lca`

The R^2 value provided by `lca` is the fraction of the total variation in the measurements that is explained by each model. Generally, the user can select a model if it explains a large fraction of the total variation (*e.g.*, more than 95%). Each parameter estimated by `lca` is also given a 90% confidence interval. The confidence interval is provided so that the user can distinguish terms in each model that do not contribute to goodness-of-fit; if any parameter's confidence interval contains zero, then that parameter cannot be statistically differentiated from zero, and the associated term should be eliminated. Eliminating terms from a model is useful because it narrows the confidence intervals on the model's predictions.

While `lca` assists in selecting a model for each category as a function of a single variable, the user's goal is often to create performance models that are functions of more than one variable. To use `lca` to construct multivariate models, the user must employ an appropriate experimental design. In many cases, the environmental factors (*e.g.*, number of processors or data set size) *interact,* that is, they are not simply additive. In addition, overhead models for many factors are nonlinear. For these two reasons, the user must use factorial or reduced-factorial experimental designs, and must measure more than two values for each factor [2; 4]. In practice, these criteria are met if the user samples the "edges" of the parameter space, for 3 or more points on each edge.

In some cases it will be necessary to perform simple analysis on the program to ascertain the models (which would still be tested by the user using `lca`). The examples in Section 4 exhibit cases where this occurs. We expect that in most cases this analysis will be straightforward.

4 Using Lost Cycles Analysis

This section presents a detailed case study of the use of lost cycles analysis on parallel implementations of the two-dimensional discrete Fourier transform program (2D FFT). Additional case studies showing the use of lost cycles analysis on less regularly-structured applications are presented in [9].

The serial implementation of 2D FFT consists of a number of iterations which consist of 1D FFTs on columns of the input matrix, followed by 1D FFTs on the rows of the matrix. The environmental factors of interest are the number of processors p (varied from 2 to 26), and the size of one side of the input matrix n (varied from 32 to 1024).

We consider two parallel implementations of the program. The first parallelization is purely data parallel (DP). In the DP version, each iteration of the program consists of 5 parallel loops: one to initialize the matrix, one to perform the columnwise FFTs, two to transpose the matrix (using an intermediate matrix), and one to perform the rowwise FFTs.

The second parallel implementation uses task parallelism as well as data parallelism. In this implementation (TP), processors are segregated into two groups using tasking directives. One group initializes the matrix and performs data-parallel rowwise 1D FFTs, while the other group transposes the matrix and performs data-parallel column-wise 1D FFTs. The two tasks are pipelined so that each one is kept busy working on separate matrices.

4.1 Modeling the Performance of 2D FFT

The ability to capture the expected performance of a program based on a small number of measurements is critical to managing the problem of understanding and selecting among differing implementations. Measuring and debugging program performance without gathering large amounts of data is an important capability in its own right, and is the subject of much current effort [3; 21; 23]. The results in this section show that lost cycles modeling is a convenient way of capturing large amounts of performance data, requiring minimal

Category	Model	
	Varying n	Varying p
PC	$n^2 \log(n)$	1
LI	$n \log(n)$	$p\sqrt{p}$
IP	1	p
SL	0	0
CL	n^2	p
RC	n^2	$p, p > \theta$

Table 1: Models of Overhead, 2D FFT, as a Function of n and p

measurement effort and little storage.

In this example we will:

1. Measure the program's lost cycles for the "edges" of the parameter space, *i.e.*, for $p = 2$, $p = 26$, $n = 32$, and $n = 1024$. This is a full factorial design.

2. Select appropriate simple models for each category of lost cycles and for pure computation, as separate functions of varying data size and varying number of processors.

3. Use the measurements made in step 1 to combine and parameterize the models selected in step 2, yielding predictions for running time over the entire range of data sizes and numbers of processors.

The process of combining models in step 3 uses the data collected from step 1. In some cases, models will be additive; in other cases they will be multiplicative. The data collected in step 1 can be used to distinguish these cases by using standard allocation of variation techniques for factorial designs [16]. Allocation of variation uses least-squares fits to assess the main (additive) and interaction (multiplicative) terms of a model. To perform allocation of variation, we use `lca` in a slightly different role — we use it to find the least squares fit of additive and multiplicative terms to the experimental data. If interactions account for the majority of variation, we combine the models for separate factors by multiplying them together; otherwise we add the separate-factor models to form the combined model.

The simple models we chose to describe each overhead are listed in Table 1, as separate functions of n (the length of a side of the input matrix) and p (number of processors). Each model has an implicitly associated constant; the purpose of our lost cycles measurements in step 1 is to discover the constants. Each of these models is a simple, initial approximation to reality. Better models for each are possible, but not necessary in this context since they trade increasing accuracy for increasing measurement cost, and decreasing analytic tractability.

Considering first the models for varying n, the model for pure computation is based on simple algorithmic analysis of 2D FFT. The model for load imbalance is based on the length of an iteration of the program's parallel loops. There are no synchronization operations in the program, so we expect no synchronization loss. The model for insufficient parallelism is based on the the portion of the code that runs serially, which has no data size dependencies. The model for communication loss is based on the total amount of data used. Finally, the model for resource contention is based on the expectation that resource contention will be proportional to data size.

In choosing models of overhead as we vary the number of processors, we can rely on the large body of work in the literature to provide likely candidate models. Most of the models we use are straightforward: pure computation does not vary as we vary processors, insufficient parallelism obeys Amdahl's Law, and synchronization loss is zero.

Load imbalance can arise in two ways: variation in the running time of each loop iteration, and unequal numbers of loop iterations handled by different processors. If variation in running time of iterations is random, the time taken by the longest iteration can be modeled using order statistics (e.g., [11]) and predicted to grow proportionally to $\sqrt{p}$. Communication loss can be difficult to model, but for this simple application is proportional to p. Finally, resource contention can be expected to grow linearly once the number of processors passes a threshold value.

Using the lost cycles measurements we then parameterize the six models (*PC, LI, IP, SL, CL,* and *RC*). For example, the final form for Load Imbalance (in which the effects of varying p and n are multiplicative) is:

$$LI(n,p) = \frac{n \log(n) p \sqrt{p}}{36200}.$$

Using the basic identity $T_p = (T_c + T_o)/p$, we construct the performance model for the implementation by adding the performance models for each category and dividing by p. The results for the 2D FFT program are shown in Figure 3. These plots show the performance of the application measured in Mflops, as a function of both number of processors and of dataset size. The left hand plot shows

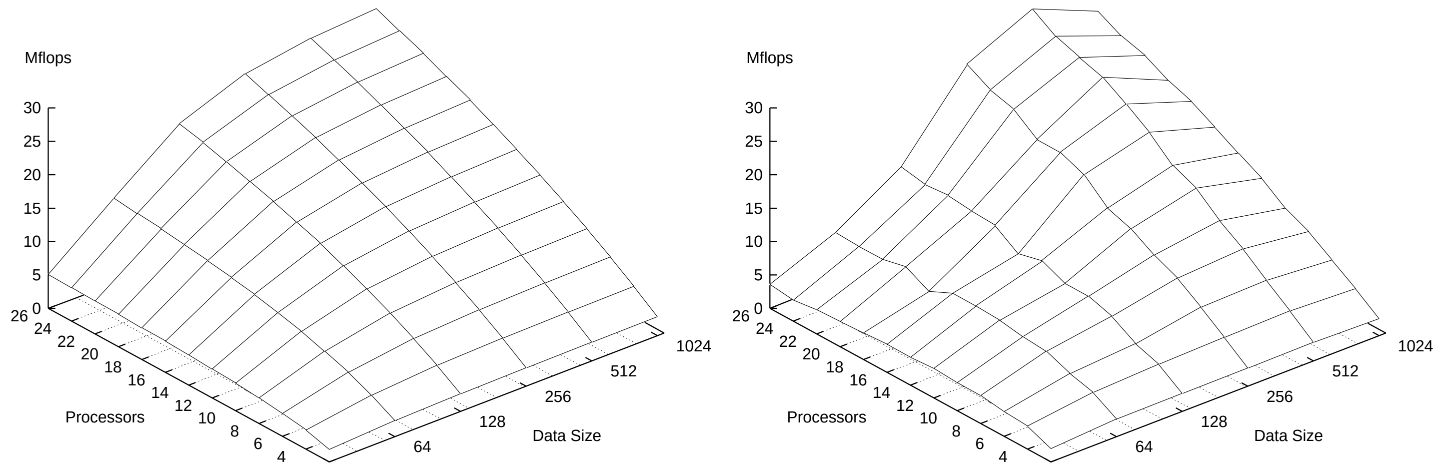

Figure 3: Predicted and Actual Performance of 2D FFT

the predictions of our model for 78 data points, that is, all points within the range of processors and data set sizes we set out to model. The right hand plot shows the actual measured performance of the application on the KSR1 for those same 78 data points.

As can be seen, the model is an idealized but reasonably accurate approximation to actual performance. In fact, the average relative error of the model with respect to the actual performance, over all 78 points, is only 12.5%. For comparison, the average relative error of a simple linear interpolation (without interactions) based on a least squares fit of the four "corner" points is over 750%. Thus both the overall shape of the predicted performance curve and its actual values are sufficiently accurate to allow it to be used in studying tradeoffs against an alternative implementation, which we will do in the next section.

4.2 Task Parallel vs. Data Parallel 2D FFT

A comparison of the task parallel and data parallel implementations of 2D FFT on the iWarp was presented in [27]. On that machine, the authors discovered that as data set sizes are varied past a certain threshold, the choice of which implementation is best changes. For small data sets ($n \leq 128$) the parallel tasking implementation outperformed the pure data parallel implementation. For large data set sizes ($n \geq 256$), the purely data parallel implementation outperformed the parallel tasking implementation. The principal reason for this effect is that in the parallel task version, communication between tasks must pass through a single channel of the iWarp network, while purely data parallel communication can take place along multiple channels. For small data sizes, the larger problem granularity of parallel tasking leads to better performance, but as problem sizes increase, intertask communication becomes a bottleneck.

It is interesting to ask whether a similar effect would be observed when this application is run on the KSR1, a machine with a significantly different architecture. Unfortunately, the data from the iWarp cannot help us decide which executions to measure, since the machines are so different. Thus we immediately run into a problem: perhaps there is a crossover between implementations in some section of the environment space (here, n and p), but finding the crossover would require measurements over the entire space.

To answer this question using the lost cycles approach, we only need to construct a model for the application. The simplest approach in this case is to 1) decide whether the category models used for the pure data parallel implementation follow the same functions; and 2) determine new constants for the overhead functions. To do this we measured 6 points varying the data set size, (to explore the functions of n) and 6 points varying the number of processors (to explore the functions of p).

The results are shown in Table 2. The table

Category	Data Parallel	Task Parallel
PC	$\frac{n^2 \log(n)}{3550}$	$\frac{n^2 \log(n)}{3350}$
LI	$\frac{n \log(n)}{63.0}$	$\frac{n \log(n)}{81.9}$
IP	3.36	$\frac{n^2}{992}$
SL	0	$\frac{n^2}{31600}$
CL	$\frac{n^2}{14900}$	$\frac{n^2}{12200}$
RC	$\frac{n^2}{20100}$	$\frac{n^2}{35600}$

Table 2: Performance Models for Data Parallel and Task Parallel 2D FFT

shows the functions and the associated constants for the n variable, since the models did not differ significantly in the p dimension.[1] These functions immediately answer our questions about these two implementations. First of all, resource contention in the task parallel implementation is significantly less than in the data parallel implementation, indicating that the channel bottleneck effects observed on the iWarp will not be present on the KSR1. This conclusion is reasonable, since intra-ring communication costs are insensitive to source and destination on the KSR1. In fact, we see that resource contention is only about half as great in the task parallel version, since in the pure data parallel version, all processors are simultaneously requesting *and* providing data during the matrix transpose, while in the parallel task version, half the processors request data and the other half provide it.

The second observation is that on this machine, the task parallel implementation will always perform more poorly than the pure data parallel implementation. Synchronization loss and insufficient parallelism are functions of n^2 in the task parallel implementation. The reason for this change from constant values to functions of n^2 when the implementation is changed can be seen in observing that synchronization loss is now equal to about a third of the communication loss. In fact, in this implementation, the two tasks do *not* incur equal overhead. The task that transposes the matrix incurs more overhead because it must traverse the source matrix across cache lines, destroying locality. Thus each loop iteration for the transposing task takes slightly longer than an iteration of the initializing task; a pair of spin locks prevents either task from overtaking the other. As a result of this synchronization loss in the main thread of the initializing task, the other threads in its group must wait without work, incurring lost cycles due to insufficient parallelism. This insufficient parallelism has a particularly small constant in the denominator and hence dominates the small improvements in resource contention and load imbalance generated by task parallelism.[2]

Thus we have quickly answered the question of whether two implementations, known to have a performance tradeoff on at least one architecture, have a similar performance tradeoff on the architecture of interest to us. To do this, we only needed to measure a small number of data points in each of the 2 environmental dimensions, and compare the resulting lost cycles models.

5 Related Work

The majority of the tools and metrics devised for performance evaluation and tuning reflect their orientation on the measure-modify paradigm (*e.g.*, [6; 14; 15]). Such tools are very useful in application fine-tuning, but usually do not provide completeness (i.e., they don't measure *all* sources of overhead in the execution). As a result, they are limited to cases in which the principal overheads are known in advance.

A number of researchers in the parallel performance evaluation and tuning community have focused on measurement of multiple parallel overheads. In particular the PEM system has developed a taxonomy of parallel overheads similar to ours [5], and Quartz and MemSpy together can measure the overhead categories we use [1; 20]. However, since these (and similar tool sets) are not oriented toward performance prediction of alternative implementations, the specific overhead categories they use are not always amenable to easy analysis. In addition, the completeness criterion has not been emphasized in most previous overhead measurement work [5; 22; 28].

A common method of modeling of overheads in parallel programs is scalability analysis [18]. Scalability analysis develops analytic, asymptotic models of computation and selected overhead categories as a function of the size of the problem n and the number of processors p. These analyses provide insight

[1] We hold p constant at its maximum value (26) in these formulae.

[2] Presumably these effects were not present on the iWarp because of message passing optimizations.

into the inherent scalability of a particular application and machine combination. Such analyses are an important part of our approach, but they don't provide enough mechanism on their own to solve the best implementation problem. Most importantly, they are subject to the problem of deciding beforehand which overheads will dominate, which can be error-prone. In addition, they cannot be used directly for performance prediction or for a comparison of alternative implementations in general because of their reliance on asymptotic analysis, and on constants which must be determined experimentally.

Another method of performance prediction is based on the notion of parallel program *templates* [24; 29]. These methods require the programmer to explicitly select a program template, which defines the mapping from application-level constructs onto the parallel hardware. Templates free the programmer from developing complex performance models, since by selecting a template for the application, the programmer also implicitly selects a performance model. Our approach, while not providing as much accuracy as is possible using the template approach, does not restrict the programmer to any set of preconstructed templates.

Our work bears similar goals to [7], which takes a static approach to performance prediction, rather than the dynamic approach we use.

Finally, the notion of selecting alternative implementations based on the environment is present in the ISSOS system [26]. The emphasis in that work is on selecting alternative implementations dynamically based on ongoing system monitoring. As a result the dynamic adaptations do not include drastic restructuring of the application, and no guidelines for how the programmer should select among alternatives were developed.

6 Conclusion

We have presented a technique that combines performance measurement with analytic modeling. Our technique is based on measuring overhead categories that meet the three criteria of completeness, orthogonality, and meaning; we have shown how each of these criteria is necessary in order to produce reliable performance predictions. We have shown that lost cycles analysis yields a performance model of the application that is accurate enough to distinguish among implementations, can be applied to programs using a wide range of languages and platforms, and can be supported by tools that automate most of the modelling process.

Our examples show that lost cycles analysis has significant flexibility. Our first example showed how modeling lost cycles can capture a great deal of performance data using only a small number of measurements. Our second example studied two implementations and used lost cycles modeling to expose differences in the predicted and actual behavior of the two applications.

A limitation of our work lies in the application of analytic models to effects that may not have analytic behavior (such a synchronization loss). We are addressing this in two ways: we believe that some subcategories may have analytic behavior (such as contention for software data structures, currently measured as sychronization loss); and we are considering nonanalytic extensions to `lca` (such as critical-path predictions) to handle the remaining cases.

We are continuing to develop the lost cycles approach in two directions. We intend to extend the set of categories measurable so that lost cycles analysis will be complete for a wider range of applications. We are also exporing the use of lost cycles analysis on program fragments, to allow users to predict the effects of program restructuring in advance of implementation.

Acknowledgments

Bart Miller provided helpful comments on an earlier draft of this paper. Jaspal Subhlok provided the iWarp Fortran code for the 2D FFT problem. Our thanks to the Cornell Theory Center and staff, for their help and the use of their KSR1.

References

[1] Thomas E. Anderson and Edward D. Lazowska. Quartz: A tool for tuning parallel program performance. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 115–125, May 1990.

[2] A. C. Atkinson and A. N. Donev. *Optimum Experimental Design*. Oxford Statistical Science Series. Oxford Science Publications, 1992.

[3] Thomas Ball and James R. Larus. Optimally profiling and tracing programs. In *Conference Record of the Nineteenth POPL*, Albuquerque, NM, 19-22 January 1992.

[4] George E. P. Box, William G. Hunter, and J. Stuart Hunter. *Statistics for Experimenters: An Introduction to Design, Data Analysis, and Model*

Building. Wiley Series in Probability and Mathematical Statistics. John Wiley and Sons, Inc., 1978.

[5] Helmar Burkhart and Roland Millen. Performance measurement tools in a multiprocessor environment. *IEEE Transactions on Computers*, 38(5):725–737, May 1989.

[6] David Callahan, Ken Kennedy, and Allan Porterfield. Analyzing and visualizing performance of memory hierarchies. In *Performance Instrumentation and Visualization*, pages 1–26. ACM Press, 1990.

[7] Mark J. Clement and Michael J. Quinn. Analytical performance prediction on multicomputers. In *Proceedings of Supercomputing '93*, pages 886–894, November 1993.

[8] Mark E. Crovella and Thomas J. LeBlanc. Performance debugging using parallel performance predicates. In *Proceedings of the 3rd ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 140–150, May 1993.

[9] Mark E. Crovella and Thomas J. LeBlanc. The search for lost cycles: A new approach to parallel program performance evaluation. Technical Report 479, Computer Science Department, University of Rochester, December 1993.

[10] Lawrence A. Crowl, Mark Crovella, Thomas J. LeBlanc, and Michael L. Scott. The advantages of multiple parallelizations in combinatorial search. *Journal of Parallel and Distributed Computing*, 21(1):110–123, April 1994.

[11] H. A. David. *Order Statistics*. John Wiley and Sons, Inc., 1981.

[12] Digital Equipment Corporation. DECChip 21064-AA RISC microprocessor preliminary data sheet. Digital Equipment Corporation, Maynard, MA, 1992.

[13] Derek L. Eager and John Zahorjan. Chores: Enhanced run-time support for shared-memory parallel computing. *ACM Transactions on Computer Systems*, 11:1–32, February 1993.

[14] Aaron J. Goldberg and John L. Hennessy. Mtool: An integrated system for performance debugging shared memory multiprocessor applications. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):28–40, January 1993.

[15] Michael T. Heath and Jennifer A. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, 8(5):29–39, September 1991.

[16] Raj Jain. *The Art of Computer Systems Performance Analysis*. Wiley and Sons, Inc., 1991.

[17] Kendall Square Research. KSR1 principles of operation. Kendall Square Research, 170 Tracer Lane, Waltham MA, 15 October 1991.

[18] Vipin Kumar and Anshul Gupta. Analyzing scalability of parallel algorithms and architectures. Technical report, TR-91-18, Computer Science Department, University of Minnesota, June 1991.

[19] Ted Lehr, Zary Segall, Dalibor Vrsalovic, Eddie Caplan, Alan Chung, and Charles Fineman. Visualizing performance debugging. *IEEE Computer*, pages 38–51, October 1989.

[20] Margaret Martonosi, Anoop Gupta, and Thomas Anderson. MemSpy: Analyzing memory system bottlenecks in programs. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 1–12, June 1992.

[21] Barton P. Miller and Jong-Deok Choi. A mechanism for efficient debugging of parallel programs. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 135–144, June 1988.

[22] Peter Møller-Nielsen and Jørgen Staunstrup. Problem-heap: A paradigm for multiprocessor algorithms. *Parallel Computing*, 4:64–74, 1987.

[23] Robert H. B. Netzer and Barton P. Miller. Optimal tracing and replay for debugging message-passing parallel programs. In *Proceedings Supercomputing '92*, pages 502–511, Minn., MN, November 1992. IEEE.

[24] G. R. Nudd, E. Papaefstathiou, Y. Papay, T. J. Atherton, C. T. Clarke, D. J. Kerbyson, A. F. Stratton, R. Ziani, and M. J. Zemerly. A layered approach to the characterisation of parallel systems for performance prediction. In *Performance Evaluation of Parallel Systems (PEPS) '93*, pages 26–34, U. Warwick, U.K., 29-30 November 1993.

[25] V. Nageshwara Rao and Vipin Kumar. Parallel depth-first search. *International Journal of Parallel Processing*, 16(6), 1989.

[26] Karsten Schwan, Rajiv Amnath, Sridhar Vasudevan, and David Ogle. A language and system for the construction and tuning of parallel programs. *IEEE Transactions on Software Engineering*, 14(4):455–471, April 1988.

[27] J. Subhlok, J. M. Stichnoth, D. R. O'Hallaron, and T. Gross. Programming task and data parallelism on a multicomputer. In *Proceedings of the Fourth PPOPP*, San Diego, CA, 20-22 May 1993.

[28] Thin-Fong Tsuei and Mary K. Vernon. Diagnosing parallel program speedup limitations using resource contention models. In *Proceedings of the 1990 International Conference on Parallel Processing*, pages I–185 – I–189. The Pennsylvania State University Press, August 1990.

[29] Dalibor Vrsalovic, Daniel P. Siewiorek, Zary Z. Segal, and Edward F. Gehringer. Performance prediction and calibration for a class of multiprocessor systems. *IEEE Transactions on Computers*, 37(11):1353–1365, November 1988.