IEEE Transactions on Parallel and Distributed Systems, 4(1):28–40, January 1993.

[7] Jeffrey K. Hollingsworth and Barton P. Miller. Dynamic control of performance monitoring on large scale parallel systems. In *7th ACM International Conference on Supercomputing*, July 20-22 1993.

[8] Michael J. Kaelbling and David M. Ogle. Minimizing monitoring costs: Choosing between tracing and sampling. In *Proceedings of the 23rd Hawaii International Conference on Systems Sciences*, pages 314 – 320, January 1990.

[9] James Kohn and Winifred Williams. ATExpert. *Journal of Parallel and Distributed Computing*, 14, May 1993.

[10] Ten-Hwang Lai and Sartaj Sahni. Anomalies in parallel branch and bound search. *Communications of the ACM*, 27(6), June 1984.

[11] T. J. LeBlanc, J. M. Mellor-Crummey, and R. J. Fowler. Analyzing parallel program executions using multiple views. *Journal of Parallel and Distributed Computing*, 9:203–217, June 1990.

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

[13] Margaret Martonosi, Anoop Gupta, and Thomas Anderson. MemSpy: Analyzing memory system bottlenecks in programs. *Performance Evaluation Review*, 20(1):1–12, June 1992.

[14] Barton P. Miller, Morgan Clark, Jeff Hollingsworth, Steven Kierstead, Sek-See Lim, and Timothy Torzewski. IPS-2: The second generation of a parallel program measurement system. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):206–217, April 1990.

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

Processor ID	0	1	2	3	4	5	6	7
Load Imbalance	4.5397	5.1334	5.0505	4.9637	4.3170	4.0277	4.0857	3.5376

Table 2: Per-processor Load Imbalance (in Seconds); Multiple Solutions, Sparse Input, KSR1

5 Conclusion

This paper introduced the notion of parallel performance predicates in order to form a language for discussing the performance of parallel programs. These predicates can be used to describe a small but surprisingly useful set of categories for the performance debugging of parallel programs: they formally define poor performance, they are relatively easy to use, and they apply even to programs for which common metrics like speedup aren't well defined.

An important feature of performance predicates is that, since they are formally defined, they allow precise measurements of program states that are often only informally defined, such as load imbalance. In this role, performance predicates form an unambiguous language for discussing the performance of parallel programs. Although a number of other tools share some goals with ours ([7] defines sources of poor performance in advance of execution, and [9] reports on various sources of overhead in a parallel program), the use of a uniform user-definable language for performance tuning is unique to our approach.

In addition, the definition of poor performance based on states rather than events is a useful one. In particular, state-based definition means that performance evaluation can be performed on any subset of an execution, and can be performed online as well as post-mortem. This flexibility is possible because performance predicates in their definition specify all program state necessary for their evaluation. In contrast, event-based performance evaluation is often forced into a post-mortem style because events may require interpretation in the context of arbitrarily complex program history.

In conclusion, we showed that by using performance predicates we can: 1) characterize aspects of parallel programs in ways that were previously informally, or only intuitively, defined; 2) verify that parallel programs are efficient when other methods are ineffective; 3) easily organize the instrumentation and measurement of complex parallel programs; 4) provide significant high-level insight into the causes of poor performance; and 5) perform detailed performance debugging via predicate refinement. We demonstrated these claims by using the technique of predicate profiling on a complex application. We showed that predicate profiling is a natural outgrowth of the performance predicate approach; that its implementation is straightforward even in complex programs; and that it gives significant insight into program behavior with very little programmer effort or run-time overhead.

Acknowledgements

We thank Donna Bergmark and the Cornell National Supercomputing Facility for their assistance and the use of their KSR1.

References

[1] Ziya Aral and Ilya Gertner. High-level debugging in Parasight. In *Proceedings ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 151–162, May 1988.

[2] Kendall Square Research Corporation. *KSR1 Principles of Operation.* Kendall Square Research Corporation, 170 Tracer Lane, Waltham, MA, 1992.

[3] Lawrence A. Crowl, Mark E. Crovella, Thomas J. LeBlanc, and Michael L. Scott. Beyond data parallelism: The advantages of multiple parallelizations in combinatorial search. Technical Report 451, Department of Computer Science, University of Rochester, April 1993.

[4] G. Cybenko, J. Bruner, S. Ho, and S. Sharma. Parallel computing and the perfect benchmarks. In *Intl. Symposium on Supercomputing*, Fukwoka, Japan, November 1991.

[5] Helen Davis and John Hennessy. Characterizing the synchronization behavior of parallel programs. In *Proceedings of the First PPEALS*, pages 198–211, July 1988.

[6] A.J. Goldberg and J.L. Hennessy. Mtool: An integrated system for performance debugging shared memory multiprocessor applications.

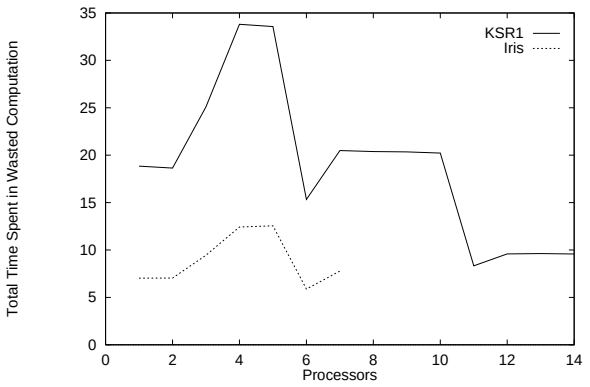

Figure 8: Decreasing Wasted Computation in Tree Parallelism

tween these two sources of LC indicates that tree parallelism is favored on machines such as the KSR which can provide many processors, while loop parallelism is favored on machines such as the Iris which have a smaller number of faster processors.

4.3 Tuning One Parallelization of Subgraph Isomorphism

While the previous section concentrated on the use of predicate profiling in the design or application-level tuning of a program, in this section we show an example of how predicate profiling can help in tuning an application whose parallel structure has been determined.

We have seen that load imbalance is one of the principal contributors to poor performance in subgraph isomorphism. We studied the last example in the previous section in more detail, to see if predicate profiling could offer insight that could lead to removing some load imbalance. To do so, we separated out the load imbalance measurements on a per-processor basis. The resulting measurements for 8 processors on the KSR1 (the program showed minimum running time in this case) is given in Table 2.

Table 2 indicates that there is a systematic source of load imbalance present in the parallel loops, which is indicated by the steadily decreasing values for load imbalance with processor number (after excluding processor 0, which is a special case). In this program, loop iterations are statically scheduled in a blocked fashion, with processor 1 getting the first I/N iterations, processor 2 getting the next I/N, and so on (where I is the total number of iterations in the loop, and N is the number of processors participating). The load imbalance values indicate that loop iterations with lower-numbered indices tend to have less work to do. The reason this occurs is that each iteration of a filter loop corresponds to eliminating search nodes at one particular tree level. Later iterations correspond to search nodes closer to the leaf level. When the program searched downward to some level n, all levels closer to the root than n contain only one search node, representing the path taken to the current node. Tree levels closer to the leaves than n will in general contain many search nodes. Since the filters operate by trying to eliminate search nodes one by one, the filters do more processing in later loop iterations (in the average case).

The presence of systematic load imbalance suggests that a simple modification of the program, to a round-robin scheduling of loop iterations, might alleviate some load imbalance. In such a schedule, processor 1 is assigned iterations numbered 1, $1+N$, etc. In fact, this simple modification, suggested by the predicate profile, results in a performance improvement of 12%, decreasing running time from 10.22 seconds down to 9.04 seconds on 8 processors.

This sort of extension of predicate profiling can be thought of as an instance of the general notion of *predicate refinement.* Predicate refinement is the way in which predicates are made to apply to a more restricted domain, so as to narrow the scope of investigation and pinpoint performance problems.

For example, as written, the `Load_Imbalance` predicate does not distinguish between the two common kinds of load imbalance: unequal task sizes, and unequal numbers of tasks on per-processor work queues. It would be a simple and consistent extension of these predicates to make that distinction, by adding a per-processor state that indicates an empty work queue. We would then refine the `Load_Imbalance` predicate into two disjoint predicates, based on the state of the per-processor work queue, providing better insight into the kinds of overhead being measured. Other predicate refinements include restricting predicate evaluation to certain segments of code, or differentiating among many code segments. Highly detailed predicate refinement would consist of sampling the program counter as well as the performance predicates, and relating overheads to code segments at the source line level.

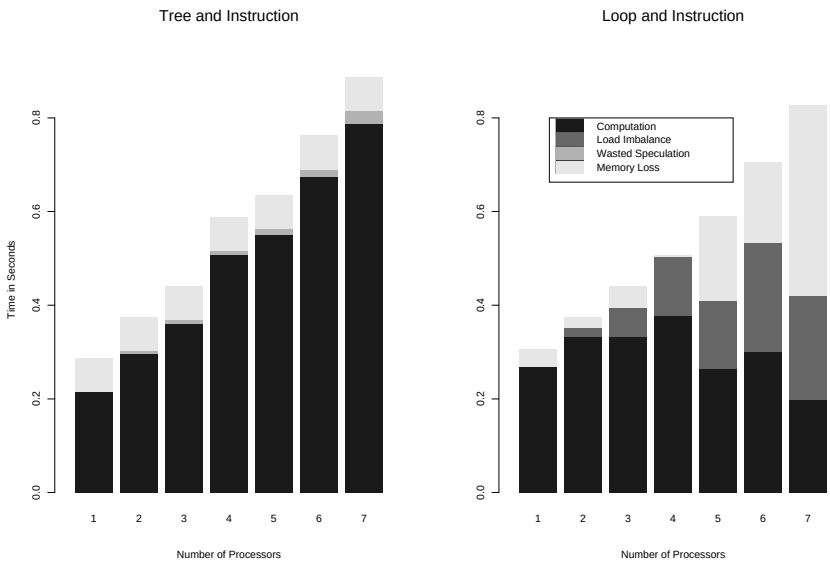

Figure 6: Predicate Profiles of Two Parallelizations on Iris; Many Solutions, Dense Input

solving this problem on a single processor, the Iris only has 8 processors, while our KSR configuration has 32 processors (much larger machines are available). Our measurements of load imbalance show that for this problem, on these machines, the degree of load imbalance under loop parallelism grows quite large with an increase in the number of processors. Figure 7 shows the fraction of total processor cycles lost due to load imbalance for loop parallelism on this problem on both machines. The figure indicates that beyond about 8 processors, the fraction of cycles lost due to load imbalance grows very large. In fact, the benefit of adding additional processors beyond this point is completely counteracted by the increase in load imbalance, precluding the KSR from benefiting from its larger supply of processors.

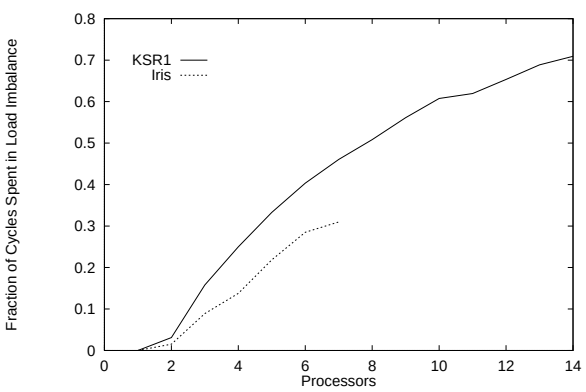

Figure 7: Increasing Load Imbalance in Loop Parallelism

On the other hand, the KSR outperforms the Iris under tree parallelism. As before, the single processor case favors the Iris (7.03 seconds on the Iris, 18.86 on the KSR1). However, there is no load imbalance under tree parallelism on this problem; the dominant source of LC is wasted computation due to speculation. Figure 8 shows the total time spent on wasted computation for this problem on both machines, in seconds. As the number of processors increases, each time the line does not rise, the program has benefited from an increase in processing power: when the line stays flat, a constant amount of work has been divided among a larger number of processors, and when the line drops, the program has found a cheaper set of solutions via speculative parallelism. The figure shows that increasing processors for this problem continues to yield significant benefits beyond 8 processors; as a result, the KSR is able to exploit its larger number of processors to advantage and outperform the Iris. Using the common metric of LC in both cases allows us to confirm that the KSR should outperform the Iris for tree parallelism after about 11 processors, since at that point the KSR execution contains the same or less LC than does the 8-processor Iris execution.

In this case we have seen that the LC due to speculation decreases with an increase in processors, while the LC due to load imbalance increases with an increase in processors. The tradeoff be-

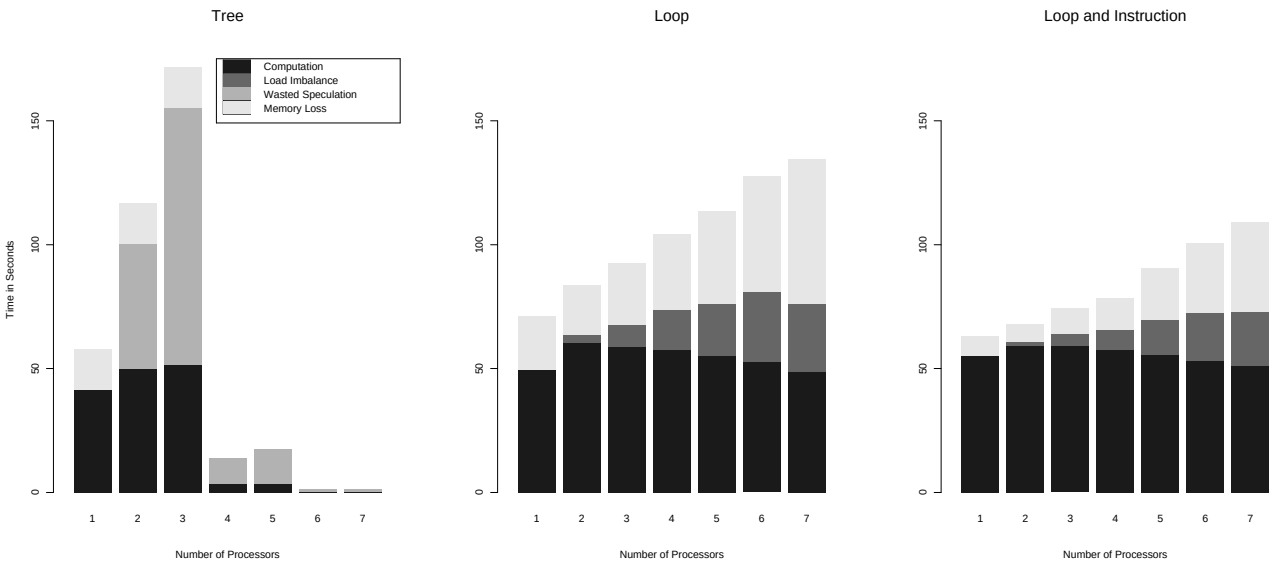

Figure 4: Predicate Profiles of Three Parallelizations on Iris; One Solution, Sparse Input

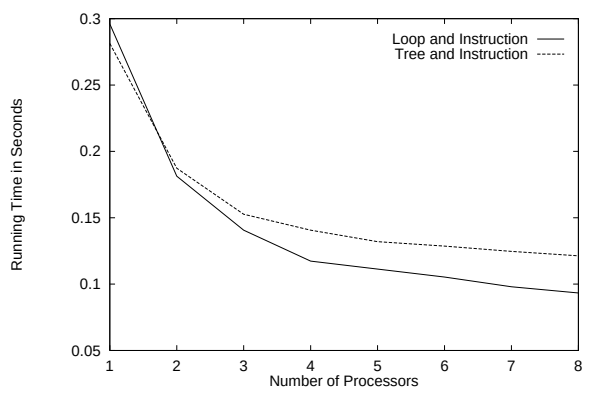

Figure 5: Running Time of Two Parallelizations on Iris; Many Solutions, Dense Input

The similar performance of these two versions occurs for different reasons, as the profiles in Figure 6 show. Both parallelizations start out finding the same set of solutions, as can be seen by their comparable profiles in the one processor case. However, as we add processors, tree parallelism (on the left) is benefiting from finding some solutions faster in each additional subtree. This benefit is shown by the fact that each additional processor adds some pure computation, but not as much as the 1 processor case. On the other hand, loop parallelism (on the right) is still finding the same solutions, only faster. It is in contrast being limited by load imbalance and memory loss (communication) as the number of processors increases.

	Iris	KSR1
loop	2.05	10.68
tree	2.59	2.24

Table 1: Running Time of Loop and Tree (in seconds); Multiple Solutions, Sparse Input

By using LC as common metric, these data show that the two parallelizations are exploiting entirely different sources of performance improvement on the same problem. Thus, predicate profiling suggests that there might be an opportunity for a hybrid algorithm that exploits both tree and loop parallelism, a fact which is confirmed in [3].

In our third example, we show how predicate profiling can be used to understand a case in which loop parallelism outperforms tree on the Iris, while tree parallelism outperforms loop on the KSR1. When searching for multiple solutions in a sparse input space, we find that the two parallelizations perform as shown in Table 1. This table shows that the proper choice of parallelization depends on the underlying machine.

To understand why the Iris outperforms the KSR under loop parallelism, we first note that the uniprocessor (sequential) running time of the program is 21.88 seconds on the KSR, while it is 8.66 seconds on the Iris. Although the Iris is faster at

3. different parallelizations of a program may perform very differently when ported to a new machine.

Our first example examines the reasons for the widely differing performance of three parallelizations of subgraph isomorphism. We are concerned with the best parallelization when searching for a single isomorphism, given a sparse input space. The running times of three parallelizations: tree, loop, and loop plus instruction, are shown in Figure 3 for the Iris. This figure shows that there are a number of issues involved in understanding the performance of these parallelizations: why is tree parallelism the worst performer on 2 and 3 processors, but the best performer on 4 or more processors? Why is loop plus instruction parallelism better than loop alone? What is preventing loop parallelism from performing well? Will tree parallelism always outperform loop plus instruction on more than seven processors?

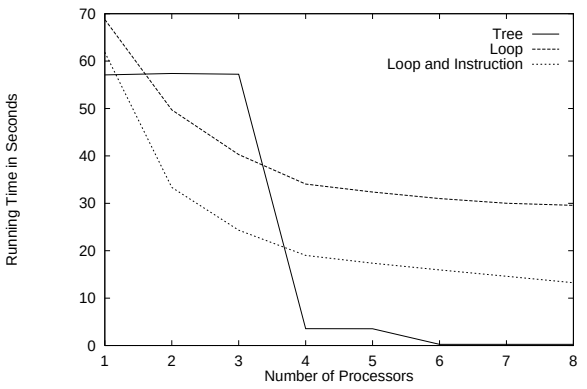

Figure 3: Running Time of Three Parallelizations on Iris; One Solution, Sparse Input

Figure 4 shows predicate profiles for the three executions shown in Figure 3. In these charts, each bar shows the total processing time, summed over all processors. Each bar is further broken down to show computation time (lowest segment) and the three kinds of LC that are significant in this case: Load Imbalance, Wasted Speculation, and Memory Loss.

The leftmost profile shows the overheads in tree parallelism. On 1 through 3 processors, we see that LC due to wasted speculation entirely accounts for the lack of any speedup. The increases in wasted speculation imply that, in this sparse solution space, the branches searched by processors 2 and 3 do not yield a solution before the branch searched by processor 1. However, the fourth processor, exploring its own branch, finds a solution much sooner than does processor 1; likewise, processor 6 also improves on processor 4's solution time. Thus this figure gives us good insight into the distribution and relative cost of solutions in the search tree, and explains why tree parallelism outperforms loop and loop plus instruction for processors greater than 3.

Next, we examine the other two profiles. The loop profile (in the center) shows that the LC constraining this parallelization is both communication (memory) and load imbalance. Comparing it to the loop plus instruction profile, we see that computation cost when using instruction parallelism is actually higher — instruction parallelism itself is not outweighing the costs of packing and unpacking data, which must occasionally take place under instruction parallelism. However, the profiles show a less-expected benefit of this parallelization: the smaller dataset size created by packing data leads to lower communication costs when using instruction parallelism. The advantage of the common use of LC for comparing overheads in this example is that we can directly observe how instruction parallelism improves execution time: the increase in computation costs caused by data manipulations are more than offset by the decrease in LC attributable to communication. Since there is little communication-caused LC in the tree parallel execution, we expect instruction parallelism to have little benefit in combination with tree parallelism. Our data confirm this conclusion: instruction parallelism for this problem *increases* the running time of tree parallelism anywhere from 5 to 25%.

The effects of speculative parallelism seen in the tree parallel version demonstrate that for some programs, it can be difficult to determine when they are performing well. Speedup is not an effective metric for this program because the multiprocessor execution does not compute the same result as the uniprocessor execution. However, LC can be used to assess whether the program is performing well, since the absence of LC, along with an efficient uniprocessor algorithm, indicates an efficient program.

Our second example shows two parallelizations that exhibit complementary reasons for good performance. Figure 5 shows the running time of instruction plus loop, and instruction plus tree parallelism when searching for many solutions in a dense solution space.

4 An Example

We have found predicate profiling to be useful in application-level performance tuning (e.g., selecting the best parallelization of an algorithm) and in program-level tuning (e.g., improving one particular parallelization of a program). The next section describes the example problem we use to demonstrate predicate profiling. Section 4.2 then gives three examples of application-level tuning, and section 4.3 gives an example of program-level performance tuning.

4.1 The Example Problem: Subgraph Isomorphism

We employed predicate profiling on a program that solves the *subgraph isomorphism* problem. Given two graphs, one small and one large, the problem is to find one or more isomorphisms from the small graph to arbitrary subgraphs of the large graph. An isomorphism is a mapping from each vertex in the small graph to a unique vertex in the large graph, such that if two vertices are connected by an edge in the small graph, then their corresponding vertices in the large graph are also connected by an edge. A complete description of the problem and the algorithm we used is given in [3]

In our algorithm, the search for isomorphisms takes the form of a tree, where nodes at level i correspond to a single postulated mapping for vertices 1 through i in the small graph. Since the search space is very large, it is prudent to eliminate tree nodes early, before they are visited in the search. We do this by applying a set of *filters* as each tree node is visited. These filters typically loop over the vertices in the two graphs, looking for inconsistencies between the current mapping and search nodes not yet visited. For the problem sizes we consider, these filters prune the search space enough to make the problem tractable.

In the experiments presented here, input graphs are randomly generated; by varying the likelihood of edges between nodes in each graph, we can vary the likelihood that a leaf node in the search tree represents an isomorphism. We use this probability, which we refer to as the *density* of the problem space, to characterize the input data in our examples.

This algorithm can be parallelized in a number of different ways, three of which we will discuss:

Tree In tree parallelism, we assign different processors to different subtrees of the root node in the search tree. This parallelization exploits *speculative* computation; adding a processor can cause a new section of the search space to be explored.

Loop In loop parallelism, we parallelize the loops within the filters. This accelerates the depth-first search of the solution space.

Instruction In instruction parallelism, we pack boolean data into words and use logical operations to implement set operations. This form of parallelism is of course also available on uniprocessors.

We present data for implementations running on two multiprocessors: an 8 processor Silicon Graphics 4D/480GTX multiprocessor workstation, and a 32 processor Kendall Square Research KSR1. Although these are both shared-memory multiprocessors, we show that in some cases different parallelizations are preferable on different machines.

We show data for only three parallelizations of this problem, but there are more opportunities for parallelism available — for example, vector operations and functional parallelism. Additionally, we might choose to parallelize only certain loops, or certain subtrees. As a result, there are a very large number of possible parallelizations of this problem. Without an understanding of the reasons why one parallelization outperforms another for a particular machine, choosing the best parallelization for this algorithm would be very difficult. Most performance evaluation tools provide too much low-level detail to quickly assess each of these parallelizations; in contrast, we show in the next section how predicate profiling can quickly and simply explain the important factors determining the best parallelization for subgraph isomorphism on a given machine.

4.2 Choosing the Best Parallelization of Subgraph Isomorphism

Predicate profiling seems especially well suited to the design and application-level tuning of parallel programs. In this section we present examples that show how predicate profiling can explain why:

1. different parallelizations of a program may show widely differing performance;

2. different parallelizations of a program may show similar performance for different reasons; and

$$\texttt{Load_Imbalance}(x) \equiv \texttt{Work_Exists} \wedge \texttt{Processors_Idle}(x)$$
$$\texttt{Starvation_Loss}(x) \equiv \neg\ \texttt{Work_Exists} \wedge \texttt{Processors_Idle}(x)$$
$$\texttt{Synchronization_Loss}(x) \equiv \texttt{Work_Exists} \wedge \texttt{Processors_Spinning}(x)$$
$$\texttt{Braking_Loss}(x) \equiv \texttt{Solution_Found} \wedge \texttt{Processors_Busy}(x)$$
$$\texttt{Memory_Loss}(x) \equiv \texttt{Processors_Stalled}(x)$$
$$\texttt{Wasted_Computation}(x) \equiv f(\texttt{Events, States})$$

Figure 1: Candidate Set of Performance Predicates

the cost is minimal. Measuring this predicate requires that we (automatically) modify a program's assembly code so that each basic block stores the current instruction count to local memory — this makes the stall behavior of each processor visible at run time.

Measuring the `Wasted_Computation` category is naturally application-dependent, and so it requires programmer definition; however, in our application it is easily identified. We define a state corresponding to each independently-schedulable piece of work (task), and if the processor, after executing the task, has not contributed to the solution it signals so via an event. The event causes the time measured for that task to be charged to `Wasted_Computation`. Since wasted computation may occur in many forms, this approach is not completely general, but the underlying mechanisms are sufficient for a wide variety of cases.

The cost of predicate profiling in a shared-memory multiprocessor amounts to the additional communication generated by the sampling process. This cost can be reduced by sampling at less than the maximum possible rate; however, lowering the sampling granularity increases the possibility of error in the results. Luckily, neither of these effects are very severe. Figure 2 shows both the percent error introduced in a profile and the percent increase in running time for a typical run of our example application. The figure shows that for a wide range of sampling intervals (between about 50 and 250 μs), both running time overhead and measurement error are close to about 5%.

The effort involved in predicate profiling is relatively small. The sampling task is structured as 50 to 100 lines of code (depending on the number of predicates) which periodically evaluate each predicate. The flag-setting code amounted to adding only 18 lines of code to our application. The code for the sampling task is reusable across implementations, and the flag setting code can be made invisible by embedding it in the runtime system.

The results of applying predicate profiling to an

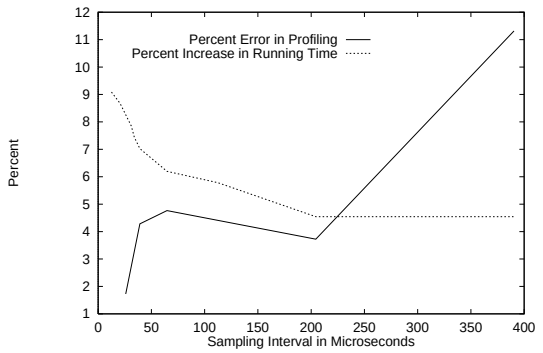

Figure 2: Measurement Error and Running Time Increase For a Range of Sampling Intervals

application can be presented in a number of ways. We have found that the simplest presentation is often the most useful: a summary table for each execution of the program. The table shows the total time, summed over all processors, that the program spent in each category. It provides an immediate assessment for the programmer of how the program performed, and what sort of tuning would be profitable. It allows the programmer to make an informed tradeoff between different kinds of tuning (e.g., memory system tuning vs. load balance tuning) and the relative effort that each kind of tuning requires.

Other useful presentations expand the profiling data along one of a number of dimensions: processors, time, or code. In the next section we show an example where the per-processor profiles give insight to performance problems. We have also expanded the profiling data in the time dimension, and can present the results using the display tool *Upshot*, available from Argonne National Labs. Finally, we believe that by combining predicate profiling with traditional program-counter profiling, that overhead values could be associated with procedures or loops, allowing more detailed exploration of code segments during performance tuning.

use in this space is the metric LC. Although it may not be obvious that such a set exists, the next section presents a set that meets these criteria; the remainder of the paper discusses the use of this set and demonstrates its utility.

2.2 Defining a Set of Performance Predicates

The set of performance predicates we use in our example are defined in Figure 1.[1] Most predicates are defined based on global or per-processor states, which are the interpretations of the expressions on the right hand side of the equations in the figure. For example, the first predicate is read as "A load imbalance on x processors exists if `Work_Exists` is true and `Idle` is true for exactly x processors."

If processors are idle, their processing power is being wasted; the first two predicates express this, and distinguish between load imbalance and insufficient parallelism as causes of idling. If processors are spinning (defined to be true while waiting for synchronization), then their processing power is being wasted only if work exists to be done; the third predicate captures this case. If processors are busy, their work is being wasted only if the solution has already been found; the fourth predicate expresses this case. The fifth predicate (`Memory_Loss`) recognizes inaction while waiting for service from memory. This predicate thus includes communication costs in a shared-memory multiprocessor as well as cold cache and replacement misses.

The last predicate (`Wasted_Computation`) is defined by the user; it expresses algorithmically wasted computation. Wasted computation occurs in some programs because adding processors changes the actual work done by the algorithm [10]. The work added may not contribute to solving the problem; providing this category allows the programmer to interpret and treat this wasted work as a form of overhead.

3 Predicate Profiling

The precise definitions of the performance predicates in the last section provide a basis for quantitative evaluation via predicate profiling. The profiling could be done by system software on each processor, but for our prototype implementation we allocated one of the system's processors to the sampling task, as in Parasight [1].

[1] The names of some of these predicates are taken from [15].

An advantage of using predicate definitions that are primarily posed in terms of global and per-processor states is that they can be implemented by setting flag-variables in a shared-memory environment. This allows profiling to be implemented by a process that periodically inspects shared memory.

Our global states are `Work_Exists` and `Solution_Found`. `Work_Exists` is true whenever parallel work has been created, but not yet completed; for example, between a **fork** and its corresponding **join**. `Solution_Found` is true when the program has completed its necessary computation; unnecessary computation may still be occurring. That is, `Solution_Found` is true as soon as the program *could* print out the answer it is intended to compute.

Our per-processor states are `Busy`, `Spinning`, and `Idle`. `Busy` is true for a processor when it is executing code that is logically a part of the program. For example, `Busy` is true when a processor begins executing its iterations of a parallel loop, and if the processor does not execute the subsequent serial code, `Busy` is false when its loop iterations complete. `Spinning` is true for a processor after it has requested a lock, but before it has acquired the lock (`Spinning` overrides `Busy`). `Idle` is true for a processor whenever it is not `Busy` or `Spinning`.

These flag definitions are intuitive, which is an essential feature. Because of of their unambiguous definitions, decisions on where to set and clear such flags in the source are very straightforward, which allows accurate implementation in a wide variety of applications. In fact, we believe that these flag manipulations can be incorporated in self-instrumenting macros, such as the Argonne P4 macros, allowing predicate profiling to be totally hidden from the programmer.

Profiling the `Memory_Loss` predicate is more difficult. On most machines, processors stall while waiting for memory, so this predicate cannot be defined based on an observable processor state, since the processor cannot itself determine when it is stalled. The most common way this state is measured is via simulation, as in [6]; we used that method in our implementation on the SGI Iris multiprocessor workstation. We have been able to profile the `Memory_Loss` predicate at run-time, however, using the on-line instruction counting implemented by compilers on the KSR1 [2]. KSR compilers update a dedicated register with the current instruction count for each basic block (at minimum). Usually this can be done by replacing no-ops, so

wrong with a parallel program.

In this paper we argue that there is significant benefit to a rapid, complete, assessment of a parallel program's performance in terms of categories that are semantically significant to programmers. This assessment either directly identifies opportunities for performance tuning, or it serves as a focusing mechanism prior to the use of more detailed performance tuning tools. In order to accomplish this assessment, we suggest a candidate set of categories, and provide precise definitions for each category. We show that these definitions can be expressed in terms of functions that identify instantaneous program states; we call these functions *performance predicates*. Via examples we demonstrate how the use of performance predicates imposes an easily-understood and easily-implemented structure on the performance debugging and tuning process. In addition, our examples show that performance predicates provide a common basis for the comparison of different kinds of overhead in a parallel program. This common basis is especially useful in program design and application-level tuning, and can be used to decide when a program is performing well when other metrics (e.g., speedup) are ineffective.

2 Parallel Performance Predicates

A performance predicate is a recognizer; it recognizes a state of the program that the programmer wishes to measure. In general, it can be very expensive to identify the instant when an arbitrary predicate changes value, especially in a parallel or distributed system [8]. Therefore, exact measurement of program states based on measuring each duration during which a predicate has a certain value is unattractive. Luckily, however, highly precise measurements are not required for performance debugging; we only need to know about any state that, in the aggregate, consumes more than some small fraction of running time. As a result, we can use a simple, sampling-based method that periodically evaluates each predicate. We call this approach *predicate profiling.*

We use the term *performance* predicate to distinguish performance predicates from correctness predicates. A correctness predicate must be evaluated atomically, which presents problems in parallel and distributed systems. A performance predicate on the other hand, need not *always* have a consistent interpretation when it is evaluated. Simple performance predicates will be inconsistent only very rarely; with sufficiently frequent sampling, any errors introduced during evaluation will have negligible effect on the end results.

2.1 Properties of Performance Predicates

The proper choice of a set of performance predicates (and the states they recognize) is constrained in three ways. First, the states should represent categories of poor performance that are intuitively meaningful. The advantage in using meaningful states is that the resulting performance measurements are easily related to design decisions at the application level.

Second, the states should be complete. That is, they should include every source of performance loss. Completeness can be verified empirically: if, after measuring all overhead in a multiprocessor execution, the remaining computation equals that of the uniprocessor case, then the set is complete for that execution.

Third, the states should be mutually exclusive. Mutual exclusion of states makes their measurements orthogonal — no wasted time is ever charged to two categories. This ensures that we can add and subtract overheads accurately, and makes it possible to reason about overheads independently.

Predicates chosen according to these constraints have the property that they express overheads in common units. That is, we can directly compare the overhead measured using one predicate to that measured using another, since they both recognize states in which no useful work is done. We will call time spent in these states in general *lost cycles* (LC). We usually express LC as its sum over all processors in an execution. Although overheads are not independent (so we cannot in general vary one without changing another) the effect of changing LC in an execution is the same no matter what specific kind of overhead is actually changing. If the useful computation done by a program is C, then the running time of the program on P processors is always

$$\frac{C + LC}{P}.$$

This equation is useful because in most cases C doesn't vary as we change P. Thus, LC forms a single uniform metric for comparing different overheads that are defined by performance predicates.

In summary, the set of performance predicates should form a meaningful, orthogonal basis set for the space of performance overheads; the norm we

Performance Debugging using Parallel Performance Predicates*

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

Abstract

Parallel programs incur overhead in many different ways, such as synchronization, load imbalance, communication, and insufficient parallelism. We have found that all of these categories are important in understanding the performance of parallel programs, and that a rapid assessment of how processing time is spent in each of these categories is extremely helpful in the performance tuning of parallel programs. As a result we have developed the notion of performance predicates, which are expressions that define these categories and can be used to recognize and classify inefficient states during a program's execution. Formal definition allows us to discuss the categories quantitatively; we present a method for measuring time spent in each category, based on the common metric of *lost cycles*. The method we describe, called *predicate profiling*, is shown to be quite useful for both application-level and program-level performance tuning. We show that predicate profiling is relatively easy to implement, and has very low run-time cost. We also show that the lost cycles metric is applicable to programs for which other metrics, like speedup, aren't well defined.

1 Introduction

The use of parallelism in a program presents many new opportunities for performance degradation. Most parallel programmers are aware of these new sources of poor performance, and group them into general categories, such as load imbalance, communication overhead, and synchronization loss. The use of these categories allows programmers to reason about and discuss the performance of specific programs. Programmers tend to think about the performance of parallel programs in terms of these categories; in fact, in our experience, understanding which category is a primary cause of poor performance is often sufficient to guide initial program tuning and design changes.

Unfortunately, current parallel performance tuning and analysis tools do not directly support programmers in assessing the amount of performance degradation attributable to each of these categories. Many performance analysis tools assume the programmer already knows which general category is the root cause of poor performance in the case at hand. These tools focus the programmer's attention on the code or data that is most to blame for poor performance in a given category (e.g., for memory system effects [6; 13], synchronization costs [5], or insufficient parallelism [4]). Other performance analysis tools are more general, but do not present data specifically in terms of categories of poor performance [11; 12; 14]. These more general tools capture and present large amounts of data, which can make it difficult to form high-level assessments of program performance.

Our work is based on a number of observations regarding the performance tuning process: programmers can identify categories of poor performance; poor performance is the result of the program or machine spending time in particular, inefficient, states; those states can be recognized, and time spent in them can be measured; and those states can be associated with categories of poor performance. We believe these observations suggest a new approach to performance debugging: using pre-defined predicates that recognize categories of poor performance and report specifically on *what is*

*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 a DARPA Research Assistantship in Parallel Processing administered by the Institute for Advanced Computer Studies, University of Maryland.