Improving Online Performance Diagnosis by the Use of Historical Performance Data[1]

Karen L. Karavanic and Barton P. Miller
{karavan,bart}@cs.wisc.edu

Computer Sciences Department
University of Wisconsin
Madison, WI 53706-1685

Abstract

Accurate performance diagnosis of parallel and distributed programs is a difficult and time-consuming task. We describe a new technique that uses historical performance data, gathered in previous executions of an application, to increase the effectiveness of automated performance diagnosis. We incorporate several different types of historical knowledge about the application's performance into an existing profiling tool, the Paradyn Parallel Performance Tool. We gather performance and structural data from previous executions of the same program, extract knowledge useful for diagnosis from this collection of data in the form of search directives, then input the directives to an enhanced version of Paradyn, which conducts a directed online diagnosis. Compared to existing approaches, incorporating historical data shortens the time required to identify bottlenecks, decreases the amount of unhelpful instrumentation, and improves the usefulness of the information obtained from a diagnostic session.

1 INTRODUCTION

Accurate performance diagnosis of parallel and distributed programs is a difficult and time-consuming task. Recent research [1, 2, 14, 3, 4] examines possible approaches for automating, and thereby simplifying, the process of diagnosing a single program run. This paper describes how historical performance data, i.e., data gathered in one or more previous executions of an application, can be used to increase the effectiveness of automated performance diagnosis. To test our ideas we incorporate several different types of historical knowledge about an application's performance into an existing diagnostic research tool, the Paradyn Parallel Performance Tool [5].

Paradyn's Performance Consultant performs online, automated bottleneck detection in a single execution of a parallel or serial code. The general search strategy used in the Performance Consultant works well for studying new and unfamiliar applications. It provides systematic investigation of an application that does not depend on any assumptions about the application or the runtime environment, so it yields useful information for a wide range of programs. In practice, we noticed that the second time we sat down with the same application, it would miss data for interesting events and possibly stop before completion due to inherent instrumentation cost limits. There is a natural tension between a generally useful, single button approach to performance diagnosis and a more application-specific, knowledge-dependent approach. Our goal is not to replace the Performance Consultant's single button model, rather, to augment the search strategy in cases where prior knowledge of the program being studied is available.

The goals for our research are:

1. Shorten the time required to identify important bottlenecks. We evaluate this strategy by measuring and comparing the total time to find bot-

1. This work is supported in part by Department of Energy Grant DE-FG02-93ER25176, NASA GSRP grant NGT-51368, NSF grants CDA-9623632 and EIA-9870684, and DARPA contract N66001-97-C-8532. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

tlenecks with and without historical information.

2. Decrease the amount of unhelpful instrumentation. There is a practical limit to the total amount of instrumentation in place at one time, to minimize inaccuracy of results due to perturbation. Decreasing unhelpful instrumentation in some cases will allow the search to continue where it might otherwise reach a limit and halt. We evaluate this strategy by measuring the total amount of instrumentation and the time to find bottlenecks.
3. Determine the precise location of all significant bottlenecks. Results most useful for performance tuning are obtained when testing identifies a reasonably small number of well-defined potential problem areas. Practical limits on the total amount of instrumentation can result in important bottlenecks not being fully explored because of the "noise" of less useful bottlenecks being tested. We measure this by identifying a set of "important" bottlenecks for a particular execution, then evaluating the effect of historical information on finding the bottlenecks in that set.

We save performance and structural data from successive executions of an application, then extract knowledge useful for diagnosis from this collection of data, in the form of search directives. There are three types of directives: *prunes*, which cause the Performance Consultant to ignore some bottleneck tests completely; *priorities*, which provide an ordering for the tests; and *thresholds*, which provide a level against which to test the application's performance. Last, we perform online performance diagnosis with an enhanced version of Paradyn, using the directives to guide the search. We evaluated our technique by testing an MPI application on the IBM SP/2, with reductions of 31% to 98% in the time needed to locate performance bottlenecks.

2 PARADYN'S PERFORMANCE CONSULTANT

Our testbed for the studies is an enhanced version of Paradyn. Paradyn is an application profiler that uses *dynamic instrumentation* to insert and delete measurement instrumentation as a program runs. This approach results in a relatively small amount of data, in contrast to most tracing methods that may result in (possibly unusably) large data files. Paradyn's Performance Consultant (PC) [2] capitalizes on this dynamic instrumentation to automate bottleneck detection during a program execution. The PC starts searching for bottlenecks by issuing instrumentation requests to collect data for a set of pre-defined performance hypotheses for the whole program. Each hypothesis is based on a continuously measured value computed by one or more Paradyn metrics, and a fixed threshold. For example, the PC starts its search by measuring total time spent in computation, synchronization, and I/O waiting, and compares these values to predefined thresholds. Instances where the measured value for the hypothesis exceeds the threshold are defined as *bottlenecks*. The full collection of hypotheses is organized as a tree, where hypotheses lower in the tree identify more specific problems than those higher up.

We represent a program as a collection of discrete program resources. Possible resources include the program code (i.e. modules and functions), application processes, machine nodes, synchronization points, data structures, and data files. Each group of resources provides a distinct view of the application. We organize the program resources into trees called *resource hierarchies*. The root node of each resource hierarchy is labeled with the hierarchy's name. As we move down from the root node, each level of the hierarchy represents a finer-grained description of the program. A *resource name* is formed by concatenating the labels along the unique path within the resource hierarchy from the root to the node representing the resource. For example, the resource name that represents function verifyA (shaded) in Figure 1 is <Code/testutil.C/verifyA>.

For a particular performance measurement, we may wish to specify certain parts of a program. For example, we may be interested in measuring CPU time as the total for one entire execution, or as the total for a single function. The *focus* constrains our view of the program to a selected part. Selecting the root node of a resource hierarchy represents the unconstrained view, the whole program. Selecting any other node narrows the view to include only those leaf nodes that are descendents of the selected node. For example, the shaded nodes in Figure 1 represent the constraint: function verifyA of process Tester:2 running on any CPU, which is labeled with the focus: < /Code/testutil.C/verifyA, /Machine, /Process/Tester:2 >.

Each node in a PC search represents instrumentation and data collection for a (hypothesis:focus) pair. If a node tests true, meaning a bottleneck is found, the Performance Consultant tries to determine more specific

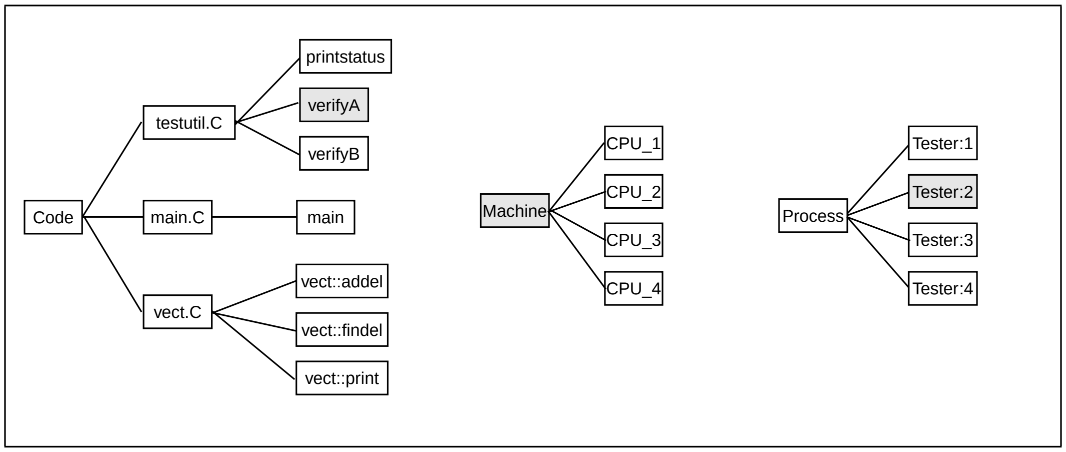
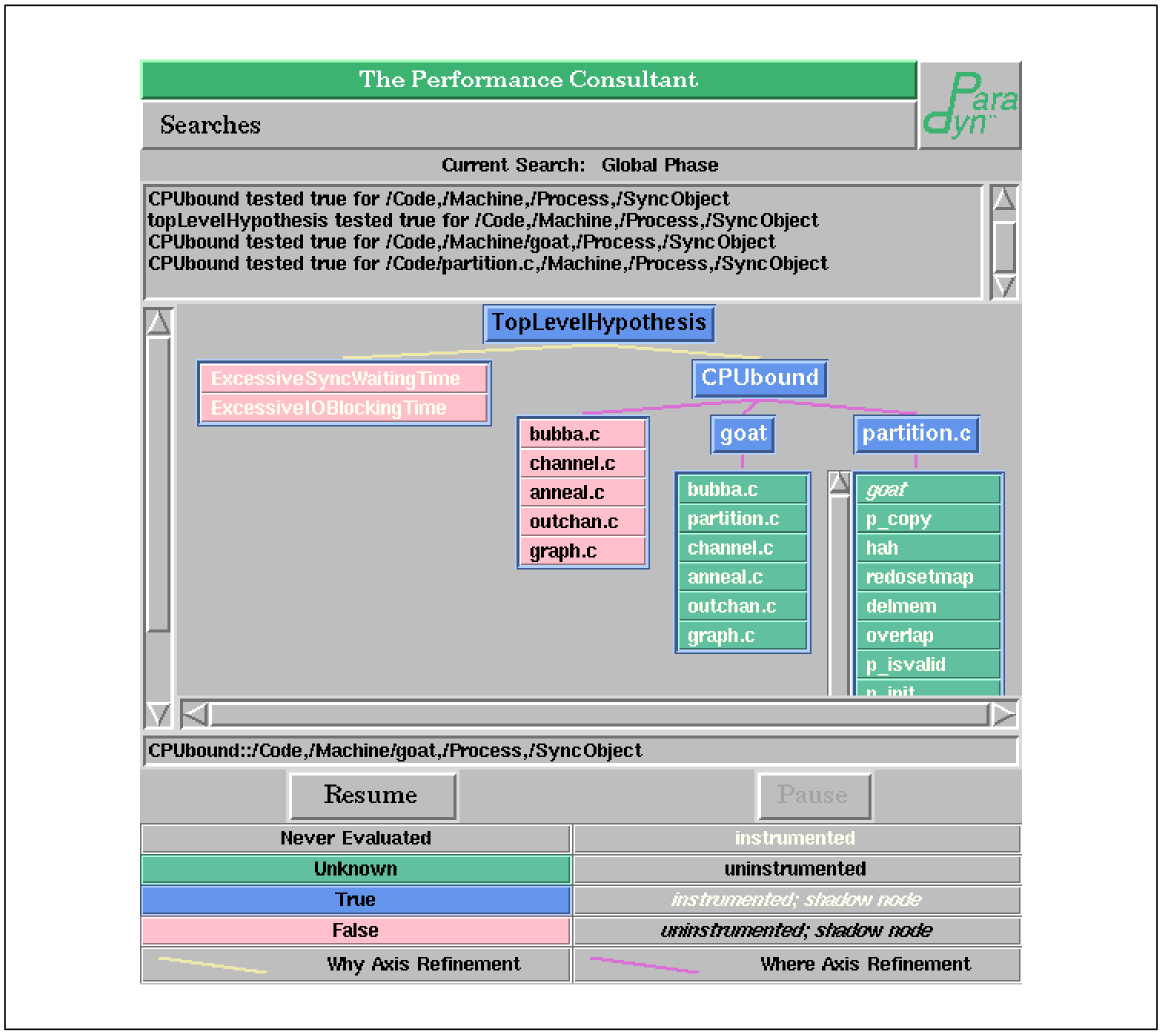


Figure 1: Representing program *Tester.* There are three resource hierarchies: Code, Machine, and Process.



CPUbound tested true for /Code,/Machine,/Process,/SyncObject
topLevelHypothesis tested true for /Code,/Machine,/Process,/SyncObject
CPUbound tested true for /Code,/Machine/goat,/Process,/SyncObject
CPUbound tested true for /Code/partition.c,/Machine,/Process,/SyncObject

CPUbound::/Code,/Machine/goat,/Process,/SyncObject

Never Evaluated	instrumented
Unknown	uninstrumented
True	*instrumented; shadow node*
False	*uninstrumented; shadow node*
Why Axis Refinement	Where Axis Refinement

Figure 2: A Performance Consultant search in progress. The three items below TopLevelHypothesis have been added as a result of refining the hypothesis. Nodes ExcessiveSyncWaitingTime and ExcessiveIOBlockingTime have tested false, as indicated by node color (light grey in this figure), and node CPUbound (dark grey) has tested true and has been expanded by refinement. The nodes bubba.c, channel.c, anneal.c, outchan.c, and graph.c all tested false, whereas the nodes goat and partition.c tested true and were refined.

information about the bottleneck. It considers two types of expansion: a more specific hypothesis, and a more specific focus. A child focus is defined as any focus obtained by moving down along a single edge in one of the resource hierarchies. Determining the children of a focus by this method is referred to as *refinement*. If a pair (h : f) tests false, testing stops and the node is not refined. The PC refines all true nodes to as specific a focus as possible.

Each (hypothesis : focus) pair is represented as a node of a directed acyclic graph called the Search History Graph (SHG). The root node of the SHG represents the pair (TopLevelHypothesis : WholeProgram), and its child nodes represent the refinements chosen as described above. Paradyn displays the SHG in list box form; we show an example in Figure 2.

Depending on the number of resources needed to represent an application, the number of hypothesis/focus pairs to be explored might be quite large. To prevent the PC data requests from overwhelming the system capacity or perturbing the application to a point where reliable results cannot be determined, the cost of instrumentation enabled by the PC is continually monitored. Search expansion, which generates new instrumentation requests, is halted when the cost reaches a critical threshold, and restarted once instrumentation deletion (initiated when nodes test false) causes the cost to return to an acceptable level.

3 HARVESTING HISTORICAL DATA

We investigated three mechanisms for including historical data in a diagnostic tool: *pruning directives* that tell the tool to ignore some resources entirely; *priorities* that tell the tool which aspects of the application and runtime environment to look at first; and *thresholds* that tell the tool specific values against which to measure the application's actual performance. These directives are described in Section 3.1. In order to use search directives extracted from one run in a new diagnosis session, it is necessary to perform a mapping on the resource names. We describe this mapping in Section 3.2.

3.1 Types of Search Directives

Pruning directives instruct the diagnostic tool to ignore a subtree of a resource hierarchy in its evaluation of a specific hypothesis. They are a mechanism for conveying information about insignificant parts of an application. The total number of hypothesis/focus pairs tested by the Performance Consultant may become large if the total number of resources is large. In practice, this is frequently true. The top-down approach taken by the PC has the effect of excluding part of the potentially huge search space, since false nodes are never refined. Prunes further shrink the size of the search space. For example, we can avoid the overhead of instrumenting small, infrequently executed functions by pruning them from the search. Pruning directives can also be used to customize the search strategy for a particular environment. For example, the static process model of MPI version 1 leads to a one-to-one correspondence between process and machine node. It is not necessary to investigate relative performance by both process and machine, so we can prune out the machine hierarchy. Pruning does not dictate the overall search strategy employed – what to examine first or next – rather it reduces the size of the total search space. One possible side effect of pruning is incorrectly eliminating something important. For this reason we also investigated other methods with better robustness. We investigated pruning based on historical data, such as functions with short execution time and redundant hierarchies (e.g. machine hierarchy if processes and machines map one-to-one) or sections of hierarchies. We also investigated pruning based on general rules, such as pruning the /SyncObject hierarchy from all but synchronization-related hypotheses.

Priorities assign a relative level of importance to specified focus-hypothesis pairs. This allows resources more likely to be responsible for behaviors of interest to be studied first, allowing data to be collected for a longer time interval. Unlike prunes, priorities do not exclude any foci from consideration; they instruct the diagnostic tool to consider certain hypothesis-focus pairs first. Each hypothesis-focus pair is given priority: High if it tested true in at least one previous execution; Low if it tested false in all previous executions; otherwise, Medium. High priority pairs are instrumented at search start and are persistent (i.e., testing continues throughout the entire program run, regardless of whether a true or false conclusion is reached). Starting up high priority pairs immediately, rather than waiting for the default top down search order to refine down to them, results in more control over the overall search order. By comparison, setting priority to medium or low only ensures an ordering between the node and its siblings.

Thresholds are the values used to determine if a hypothesis is true or false for a given focus. In the standard version of Paradyn, there is a threshold value for each hypothesis that can be set by the user. The goal is to keep the number of bottlenecks reported in a practically useful range. Reporting a large number of different bottlenecks yields inadequate guidance to the tuning effort, i.e., what to look at first, and also drives up the cost of instrumentation. Reporting only one or two bottlenecks, or failing to refine the bottlenecks to a detailed level, provides less information than might reasonably be obtained through simple visualization. We investigated automatically setting the thresholds based on historical data.

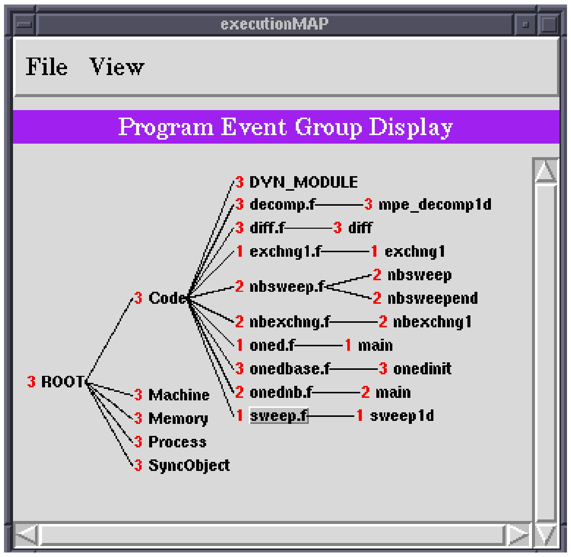
3.2 Mapping Resource Names Between Different Executions

Resources can change from one run of a program to the next. For example, an 8-node application might run on nodes 0-7 during one run and on nodes 123-130 on the next run. Similarly, process ID’s are likely to be different for each run. If we are to relate performance results from a previous run to the current run, we must be able to establish an equivalency between (*map*) the differently named resources.

The issue of mapping can also appear for code (module or function) resources. In Section 4.3, we present results from multiple implementations of a Poisson function decomposition program. The different version have different names for their main function and kernel function

After each run of the Performance Consultant, we have the search history graph and the program’s resource hierarchies. These results are used to generate search directives to be used in subsequent runs. We added new functionality to the Performance Consultant to map focus names found in these directives onto names valid in the current environment. Mapping allows us to link resources from different executions with different names, so Paradyn treats them as equivalent. One example to motivate the need for mapping is the common case of executing on differently named nodes of a machine in different runs. Mapping is implemented as a set of directives of the form “map *resourceName1 resourceName2*” specified by the user in an input file. After starting Paradyn, we apply the specified mappings to the list of extracted search directives, then read the directives into the Performance Consultant. For increased efficiency, we apply specified pruning directives, if any, to the resulting list of search directives before we read it into the Performance Consultant

Figure 3 shows combined resource hierarchies for two versions of an MPI application, Versions A and B. Each resource is tagged with execution identifier 1, 2, or 3 if the resource is found in Version A,Version B, or



Mappings Used

```
map /Code/exchng1.f /Code/nbexchng.f
map /Code/exchng1.f/exchng1 /Code/nbexchng.f/nbexchng1
map /Code/oned.f /Code/onednb.f
map /Code/sweep.f /Code/nbsweep.f
map /Code/sweep.f/sweep1d /Code/nbsweep.f/nbsweep
```

Figure 3: Mappings for Versions A and B. On the left we show the execution map for Versions A and B of the Poisson decomposition application, with the Code hierarchy expanded. Each resource is tagged with an execution identifier: resources unique to version A are labeled with “1,” those unique to version B are labeled with “2,” and those common to both are labeled with “3.” We map unique nodes which refer to code that was modified between versions, including a change of name. The mapping directives we used are shown on the right.

both, respectively.Resources unique to one execution are candidates for mapping. For example, the module containing function main is named "oned.f" in Version A, and "onednb.f" in Version B. We map these two resources, /Code/oned.f and Code/onednb.f, so that search directives extracted from runs of A may be used in diagnosing runs of B. The full set of mappings we used in this example is shown to the right of the resource hierarchies.

4 RESULTS

We performed a set of experiments to evaluate the use of prior knowledge in the form of pruning, prioritization, and threshold directives. The first section reports on the effectiveness of adding pruning and priority directives to the Performance Consultant. The second section explores the advantages of using application-specific thresholds formulated using historical data. The final section studies the use of pruning, prioritization, and generated thresholds with different versions of the same application, to simulate the common practice of performance tuning successive versions of an implementation.

4.1 Using Pruning and Priority Directives

We ran our enhanced version of the Performance Consultant on an MPI application that solves the 2-D Poisson problem[6], running on four nodes of an IBM SP/2. First we ran the PC on the application with no modifications, and saved the resource hierarchies, search history graph, and performance results. This run forms our base case and was allowed to run to completion to identify the complete (100%) set of possible bottlenecks. Then we tested three variations of directed searching: first we generated only pruning directives, second only priorities, and third a combined version with both prunes and priorities. Identical search thresholds were used in all runs. In each experiment, we recorded the time each bottleneck was reported by the tool. The times we recorded are the timestamps assigned by Paradyn to the data, and reflect application execution time. Since Paradyn performs dynamic instrumentation, the starting timestamp is determined by the instant of the instrumentation request, plus the time required to actually insert the instrumentation into the application code. Each conclusion about a performance hypothesis is determined once a set time interval of data has been received from the running application. The results are reported in Table 1.

The first experiment investigated the performance advantages obtained using pruning directives. We used data from previous runs to generate a list of pruning directives. Then we ran Paradyn, providing the list of pruning directives as input to the modified Performance Consultant. The combined search pruning directives result in a reduction of 93.5% in time to locate all true bottlenecks. We ran further tests to evaluate the effects of each of the two types of pruning: general prunes, such as pruning the /SyncObject hierarchy from all but synchronization-related hypotheses, are not specific to a particular application or environment; historic prunes, such as pruning a specific function with low execution time, are formulated based on data gathered in one or more previous executions of the same application. We see a substantial improvement with either type of pruning, and also see that the combination yields the best results: adding historic prunes resulted in execution times 28% shorter than using only general prunes.

In the second experiment, we studied the effects of ordering the search for bottlenecks using priorities. We used historical data to generate priorities for each hypothesis/focus pair as outlined in Section 3. We expected that, compared to using the PC with no historical data, we would reduce the time required to find the

% B'necks Found	No Directives	Prunes Only	General Prunes Only	Historic Prunes Only	Priorities Only	Priorities & All Prunes
25%	115.2	80.0 (-30.6%)	102.4	108.8	80.0 (-30.6%)	51.2 (-55.6%)
50%	182.4	83.2 (-54.4%)	121.6	204.8	124.8 (-31.6%)	57.6 (-68.4%)
75%	1011.2	140.8 (-86.1%)	169.6	281.6	211.2 (-79.1%)	86.4 (-91.4%)
100%	2611.2	169.4 (-93.5%)	236.8	470.4	560.0 (-78.6%)	147.2 (-94.4%)

Table 1: Time (in seconds) to Find all True Bottlenecks with Search Directives

major (true) bottlenecks, and see no change in the amount of instrumentation. We obtained a reduction of 79% in time to locate all true bottlenecks. The improvement is more modest than the reduction of 93.5% we obtained using pruning directives. However, reordering the search does not introduce the possibility of missing bottlenecks, which is an important advantage to the method.

In the final experiment, we tested a combination of prunes and priorities. Our goal was to improve on the time reduction obtained using only priorities, yet avoid the possibility of pruning important tests from the search. We included pruning of redundant and irrelevant hierarchies, but did not include prunes for previously false hypothesis/focus pairs. This combined approach may result in some retesting of false nodes, however, it will never miss new behaviors due to pruning. We obtained a reduction of 94.4% for finding 100% of the true bottlenecks, which is a reduction of 20 seconds from pruning without priorities.

4.2 Using Thresholds Determined from Historical Data

We studied the behavior of the Performance Consultant with varying threshold values for the 2-D decomposition application of Section 4.1 run across four nodes of an IBM SP/2. This sample application is strongly dominated by synchronization waiting time, which accounts for approximately 75% of the total execution time. 45% of the total execution time for all four processors is spent waiting in function exchng2, and 20% in function main. This wait is split between three message tags, 3/0, 3/1, and 3/-1 (27%, 19%, and 20% respectively). Individual processes 3 and 4 are dominated by wait time (81% and 86%) and significant waiting also occurred in processes 1 and 2 (46% and 47%).

We investigated the quality of the PC's diagnosis by checking for the number of these areas reported as bottlenecks, either individually (e.g., function main) or in combination (e.g., message tag 3/0 for function main). Full results are shown in Table 2. When a threshold setting greater than 10% was used, bottlenecks we previously determined to be significant were not reported by the PC. When the threshold was set to 12% the tool reported close to the full set of bottlenecks; the default Paradyn setting of 20%, in contrast, resulted in 7 of the 26 bottlenecks being missed. The third column shows how much instrumentation was used to diagnose the program run. Setting the threshold to 12% (shaded) yields good results and also uses noticeably less instrumentation than a setting of 10% or 5%. The final column shows an efficiency metric determined by dividing the number of bottlenecks found by the number of hypothesis/pairs tested. Efficiency decreases with thresholds below 12%, an indication that lowering the threshold below 12% increases the amount of instrumentation but does not improve the result.

In earlier studies we found similar results for an ocean circulation modeling code using PVM, running on SUN SPARCstations. We found an optimal synchronization threshold at 20%, from a starting point of 30% (which yielded an incomplete diagnosis). Efficiency decreased below 20%, for example the number of metric-focus pairs instrumented was 326 for 20%

Synchronization Bottleneck Threshold Setting (% of total execution time)	Number of Bottlenecks Reported by the Performance Consultant	Total Number of Hypothesis/Focus Pairs Tested	Efficiency (Bottlenecks Found Per Pair Tested)
30%	9	30	0.3
20%	19	66	0.29
14%	22	76	0.29
12%	25	85	0.29
10%	26	107	0.24
5%	26	105	0.25

Table 2: Bottlenecks Found with Varying Threshold Values. Number of bottlenecks reported are rounded, averaged values calculated from repeated tests.

and jumped to 373 for 10%. The useful threshold in this case differs from that found for the MPI application, showing the advantage of application-specific historical performance data.

4.3 Using Historical Data with Different Code Versions

We studied the use of historical application knowledge where the application has been revised over time. While tuning an application, a developer repeats through a cycle of profile-analyze-change. We performed a series of performance diagnoses using different versions of an MPI application on the IBM SP/2. The application implements an iterative Poisson function decomposition. We used several of the different versions of the implementation presented by Gropp *et al*[6]. In each step of the study, we used results from previous runs of the Performance Consultant to direct subsequent PC runs. There were four versions of the application: Version A is a 1-dimensional version that uses blocking send and receive operators; Version B is a non-blocking 1-dimensional version; Version C performs a 2-dimensional decomposition; and Version D runs the same code as Version C across 8 nodes (all others run on 4 nodes). We changed all versions to compute a fixed number of iterations, rather than stopping as soon as a solution is reached.

We started by running the Performance Consultant on Version A without search directives, resulting in a time to locate true bottlenecks of 2272 seconds. Next, we repeated the same diagnosis on the same version, this time including search directives generated from the previous execution, and decreased the diagnosis time by approximately 92%.

Next we examined Version B using search directives extracted from runs of Version A, and found a 98% improvement in diagnosis time. We continued for Versions C and D, each time running the Performance Consultant with search directives extracted from each individual prior run. We mapped each pair of machine resources so that search directives generated in one run could be meaningfully used to refer to machine resources discovered in a subsequent run. We also mapped functions and modules between the different code versions, as described in Section 3.2. The full results are shown in Table 3. In every case, adding historical knowledge to the Performance Consultant greatly improved its ability to quickly diagnose performance bottlenecks: diagnosis time was reduced a minimum of 75% in all executions using historical knowledge. In Table 3, each row represents the version of the application currently being diagnosed. Each column represents the source from which we extracted the search directives used. The first column contains the time to reach a diagnosis using no search directives, and subsequent columns contain the time to reach a diagnosis using search directives from different sources. We used dedicated machine time and therefore saw relatively low variability in run time for repeated executions of the same version.

After completing the test runs, we analyzed the Performance Consultant behavior to determine how it was affected by the search directives we added. First we examined the effects of using search directives from the base run of A, *a1*, to diagnose a second run of A,

		Source of Search Directives				
		None	A	B	C	D
Application Version	A	2272	183 (-92%)			
	B	4454	96 (-98%)	135 (-97%)		
	C	1021	186 (-82%)	173 (-83%)	256 (-75%)	
	D	3411	554 (-84%)	810 (-76%)	438 (-87%)	429 (-87%)

Table 3: Time (in seconds) to find all bottlenecks with search directives from different application versions. Times reported are median values for several runs, reported in seconds. Standard Deviations range from 3 to 17 seconds. Each row contains the data for a particular application version, A through D. Each column contains the data for a particular source of the search directives used with the Performance Consultant. For example, the cell found at row “C” and column “B” contains the time to diagnose C using directives from a previous run of B. Time relative to the base version (column “None”) is shown in parentheses.

Priority Setting	A only	B only	C only	A, B only	A, C only	B, C only	A, B, C	TOTAL
High	16	13	3	10	10	9	46	107
Low	32	72	24	28	20	13	92	281
Both	48	85	27	38	30	22	138	388

Table 4: Similarity of Extracted Priorities Across Code Versions. Each column represents the source(s) of the priority directives: a run of one or more of versions A, B, and C. The rows contain data for high priority, low priority, and the complete set of both. The values are the number of priority directives for the particular category. For example, of the total 107 different high priority directives, 16 were unique to version A and 46 were common to versions A, B, and C.

a2. 81 hypothesis/focus pairs tested true in *a1*, resulting in 81 search directives that set priority to high. In *a2*, a total of 103 hypothesis/focus pairs tested true. 78 were pairs that tested true in *a1* (and were included in the 81 search directives); of the remaining 25, 3 had been set to low priority, 6 were intermediate level nodes not tested in *a1*, and the remaining 16 were more detailed/refined answers not tested in *a1* due to cost limits. In this case, using search directives resulted in a more detailed diagnosis than could be performed without the directives.

Although we had anticipated search directives from different versions would not be as effective as search directives from the same version, the results showed only small differences in most cases. We examined the different runs of Version C, noting the differences in the sets of search directives extracted from the base runs of Versions A, B, and C. As shown in Table 4, 36% of the priorities were common across all three sets of directives, 41% were unique to a single set, and the remaining 23% occurred in two of the three sets. High priority settings have a bigger impact; for this category, 43% were common to all three, 30% were unique to one, and the remaining 27% were common to two.

The list of bottlenecks found did not vary between the runs of C that used search directives extracted from Versions A, B, or C. Of 115 total bottlenecks diagnosed as true by the Performance Consultant in any of these runs, 113 were common across all three, and the remaining 3 were common to two of the three.

We conclude that, for this example, despite modifications to the communications primitives (blocking or non-), and modifications to the algorithm (1-d or 2-d decomposition), the bottleneck locations remained the same. So although total synchronization time and total execution time varied between versions, the set of resources responsible for the time was similar.

We also investigated using results from multiple previous runs to guide the current run. We looked at two different approaches to combining search directives from different versions: $A \cap B$ sets to a high/low priority only those hypothesis/focus pairs that tested true/false in *both* Versions A and B. $A \cup B$ sets to a high priority those hypothesis/focus pairs that tested true in either A or B, and sets to low priority those hypothesis/focus pairs which tested false in either version and did not test true in A or B. We used the resulting set of directives to diagnose Version C. In this particular example, the lists of priorities that result from the two methods of combination have 59 common directives, with 38 additional directives unique to $A \cup B$. The resulting diagnosis times were 176 for $A \cup B$ and 179 for $A \cap B$. This difference is too small for us to conclude the superiority of one combination method over the other. Which performs better is related to the similarity of the sets of directives generated using data from runs A and B, not the similarity in code or platform of the versions.

5 RELATED WORK

We know of no other existing tool for automated performance diagnosis that adapts its testing strategy using historical performance data. Chitra[7] generates a parameterized empirical model fitting all observed data from one or more program runs to predict future program performance. The CMon and PSpec tools [8] gather data from multiple executions and produce a single summary of application behavior, checking for particular metric values at predetermined execution points. There is no widely used set of benchmarks to measure effectiveness or correctness of parallel and distributed performance diagnosis. Francioni [9] proposed a test suite for debugging/performance analysis tools called SWAMP. Malony [10] conducted a detailed study of performance perturbation due to instrumentation. Hondroudakis and Procter [11] classify the tasks involved in parallel performance tuning based on extensive user survey. Their results support the need for performance data storage across multiple executions and across different tuning studies. A recent study by Smith *et al* [12] investigates relevant parameters for predicting application run times from historical information. Our goal is not to predict run time, rather to harvest directives for a runtime tool to measure behavior at a more detailed level.

6 CONCLUSIONS AND FUTURE WORK

We have described a new approach to automated performance diagnosis that incorporates knowledge from previous runs of the same application. The result is a performance tool that learns from each diagnostic program run, adapting its search strategy to obtain more useful diagnoses more quickly. We show performance gains of up to 98% obtained by incorporating historical knowledge into the Performance Consultant's search strategy. The results presented demonstrate the utility of our approach for repeated performance diagnosis of similar program runs, a common scenario when tuning parallel applications.

Harvesting useful historical knowledge requires an available store of performance data gathered from one or more previous program runs. This work is part of an ongoing research effort in which we are designing and developing an infrastructure for storing, naming, and querying multi-execution performance data. Our representation for the space of executions, and techniques for quantitatively and automatically comparing two or more executions, are described in a previous paper [13].

We are currently extending this research in several directions. We are studying additional approaches for mapping resources from different executions. Our goal is to automate the mapping to the furthest extent possible, while continuing to allow user-specified mappings. We have demonstrated resource mapping performed at the start of each new execution, and we hope to extend this to cover cases in which new resources are discovered later in an application run. We are also extending the ability to extract search directives to the case where results in the form of a Search History Graph from a previous PC run are not available, but we do have the raw data needed to test hypotheses postmortem. This would allow us to study use of search directives extracted from results gathered with different monitoring tools.

7 REFERENCES

[1] W. Williams, T. Hoel, and D. Pase. The MPP Apprentice performance tool: Delivering the performance of the Cray T3D. In K.M. Decker and R.M. Rehmann, editors, **Programming Environments for Massively Parallel Distributed Systems**, Birkhäuser, 1994.

[2] J. K. Hollingsworth and B. P. Miller. Dynamic control of performance monitoring on large scale parallel systems. In *Proceedings of the International Conference on Supercomputing*, Tokyo, July 1993.

[3] B. R. Helm, A. D. Malony, and S. F. Fickas. Capturing and automating performance diagnosis: the Poirot approach. In *Proceedings, 1995 International Parallel Processing Symposium*, pages 606–613, April 1995.

[4] A. B. Sinha and L. V. Kale. Towards automatic performance analysis. In *Proceedings, 1996 International Conference on Parallel Processing*, pages III–53–60, 1996.

[5] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn parallel performance measurement tool. *IEEE Computer*, 28(11):37–46, November 1995.

[6] W. Gropp, E. Lusk, and A. Skjellum. **Using MPI: Portable Parallel Programming with the Message-Passing Interface**, chapter 4. The MIT Press, 1994.

[7] A. Mathur and M. Abrams. Toward a machine assisted software performance diagnosis methodology. Technical Report TR 93-12, Virginia Polytechnic Institute and State University Department of Computer Science, 1993.

[8] S. E. Perl, W. E. Weihl, and B. Noble. Continuous monitoring and performance specification. Technical Report 153, Compaq Digital Systems Research Cen-

ter, June 1998.

[9] J. M. Francioni. Determining the effectiveness of interfaces for debugging and performance analysis tools. In M. L. Simmons, A. H. Hayes, J. S. Brown, and D. A. Reed, editors, **Debugging and Performance Tuning for Parallel Computing Systems**, pages 127–141. IEEE Computer Society Press, 1996.

[10] A. D. Malony. *Performance Observability*. PhD thesis, University of Illinois at Urbana-Champaign, 1990.

[11] A. Hondroudakis and R. Procter. An empirically derived framework for classifying parallel program performance tuning problems. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 112–121. ACM Press, August 1998.

[12] W. Smith, I. Foster, and V. Taylor. Predicting application run times using historical information. In *Proceedings of the IPPS/SPDP’98 Workshop on Job Scheduling Strategies for Parallel Processing*, March 1998.

[13] K. L. Karavanic and B. P. Miller. Experiment Management Support for Performance Tuning. In *SC97*, San Jose, CA, November 1997.

[14] The APART Working Group on Automatic Performance Analysis: Resources and Tools, http://www.fz-juelich.de/apart/.