Deep Start: A Hybrid Strategy for Automated Performance Problem Searches[1]

Philip C. Roth and Barton P. Miller

Computer Sciences Department, University of Wisconsin–Madison,
1210 W. Dayton Street, Madison, WI 53706–1685 USA
{pcroth,bart}@cs.wisc.edu

Abstract. We present Deep Start, a new algorithm for automated performance diagnosis that uses stack sampling to augment our search-based automated performance diagnosis strategy. Our hybrid approach locates performance problems more quickly and finds problems hidden from a more straightforward search strategy. Deep Start uses stack samples collected as a by-product of normal search instrumentation to find *deep starters*, functions that are likely to be application bottlenecks. Deep starters are examined early during a search to improve the likelihood of finding performance problems quickly. We implemented the Deep Start algorithm in the Performance Consultant, Paradyn's automated bottleneck detection component. Deep Start found half of our test applications' known bottlenecks 32% to 59% faster than the Performance Consultant's current call graph-based search strategy, and finished finding bottlenecks 10% to 61% faster. In addition to improving search time, Deep Start often found more bottlenecks than the call graph search strategy.

1 Introduction

Automated search is an effective strategy for finding application performance problems [7,10,13,14]. With an automated search tool, the user need not be a performance analysis expert to find application performance problems because the expertise is embodied in the tool. Automated search tools benefit from the use of structural information about the application under study such as its call graph [4] and by pruning and prioritizing the search space based on the application's behavior during previous runs [12]. To attack the problem of scalability with respect to application code size, we have developed *Deep Start*, a new algorithm that uses sampling [1,2,8] to augment automated search. Our hybrid approach substantially improves search effectiveness by locating performance problems more quickly and by locating performance problems hidden from a more straightforward search strategy. We have implemented the Deep Start algorithm in the Performance Consultant, the automated bottleneck detection component of the Paradyn performance tool [13].

To search for application performance problems, the Performance Consultant (hereafter called the PC) performs experiments that test the application's behavior.

1. This work is supported in part by Department of Energy Grant DE-FG02-93ER25176, Lawrence Livermore National Lab grant B504964, and NSF grants CDA-9623632 and EIA-9870684. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

Each experiment is based on a *hypothesis* about a potential performance problem. For example, an experiment might use a hypothesis like "the application is spending too much time on blocking message passing operations." Each experiment also reflects a *focus*. A focus names a set of application resources such as a collection of functions, processes, or semaphores. For each of its experiments, the PC uses dynamic instrumentation [11] to collect the performance data it needs to evaluate whether the experiment's hypothesis is true for its focus. The PC compares the performance data it collects against user-configurable thresholds to decide whether an experiment's hypothesis is true. At the start of its search, the PC creates experiments that test the application's overall behavior. If an experiment is true (i.e., its hypothesis is true at its focus), the PC *refines* its search by creating one or more new experiments that are more specific than the original experiment. The new experiments may have a more specific hypothesis or a more specific focus than the original experiment. The PC monitors the cost of the instrumentation generated by its experiments, and respects a user-configurable cost threshold to avoid excessive intrusion on the application. Thus, as the PC refines its search, it puts new experiments on a queue of pending experiments. It *activates* (inserts instrumentation for) as many experiments from the queue as it can without exceeding the cost threshold. Also, each experiment is assigned a priority that influences the order which experiments are removed from the queue.

A *search path* is a sequence of experiments related by refinement. The PC prunes a search path when it cannot refine the newest experiment on the path, either because the experiment was false or because the PC cannot create a more specific hypothesis or focus. The PC uses a Search History Graph display (see Fig. 1) to record the cumulative refinements of a search. This display is dynamic—nodes are added as the PC refines its search. The display provides a mechanism for users to obtain information about the state of each experiment such as its hypothesis and focus, whether it is currently active (i.e., the PC is collecting data for the experiment), and whether the experiment's data has proven the experiment's hypothesis to be true, false, or not yet known.

The Deep Start search algorithm augments the PC's current call-graph-based search strategy with stack sampling. The PC's current search strategy [4] uses the application's call graph to guide refinement. For example, if it has found that an MPI application is spending too much time sending messages, the PC starts at the main function and tries to refine its search to form experiments that test the functions that main calls. If a function's experiment tests true, the search continues with its callees. Deep Start augments this strategy with stack samples collected as a by-product of normal search instrumentation. Deep Start uses its stack samples to guide the search quickly to performance problems. When Deep Start refines its search to examine individual functions, it directs the search to focus on functions that appear frequently in its stack samples. Because these functions are long-running or are called frequently, they are likely to be the application's performance bottlenecks.

Deep Start is more efficient than the current PC search strategy. Using stack samples, Deep Start can "skip ahead" through the search space early in the search. This ability allows Deep Start to detect performance problems more quickly than the current call graph-based strategy. Due to the statistical nature of sampling and because some types of performance problems such as excessive blocking for synchronization are not neces-

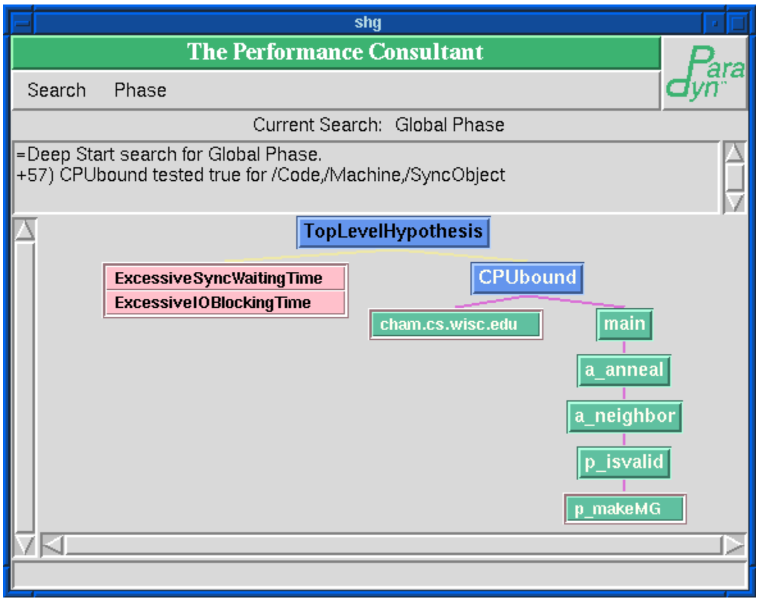


Fig. 1. The Performance Consultant's Search History Graph display

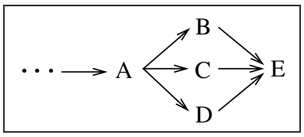


Fig. 2. A part of an application's call graph. Under the Performance Consultant's call graph-based search, if B, C, and D are not bottlenecks, E will not be examined. In contrast, the Deep Start algorithm will examine E if it appears frequently in the collected stack samples

sarily indicated by functions frequently on the call stack, Deep Start also incorporates a call-graph based search as a background task.

Deep Start is able to find performance problems hidden from the current strategy. For example, consider the portion of an application's call graph shown in Fig. 2. If A is a bottleneck but B, C, and D are not, the call graph strategy will not consider E even though E may be a significant bottleneck. Although the statistical nature of sampling does not guarantee that E will be considered by the Deep Start algorithm, if it occurs frequently in the stack samples Deep Start will examine it regardless of the behavior of B, C, and D.

2 The Deep Start Search Algorithm

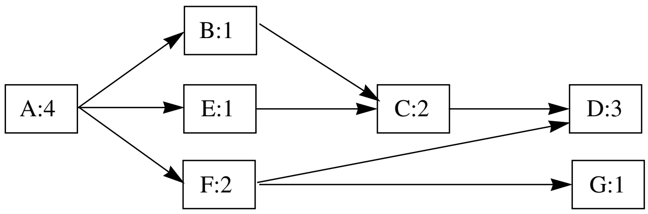
Deep Start uses stack samples collected as a by-product of dynamic instrumentation to guide its search. Paradyn daemons perform a stack walk when they insert instrumentation; this stack walk checks whether it is safe to insert instrumentation code into the application's processes. Under the Deep Start algorithm, the PC collects these stack samples and analyzes them to find *deep starters*—functions that appear frequently in the samples and thus are likely to be application bottlenecks. It creates experiments to examine the deep starters with high priority so that they will be given preference when the PC activates new experiments.

2.1 Selecting Deep Starters

If an experiment tests true and was examining a Code resource (i.e., an application or library function), the PC triggers its deep starter selection algorithm. The PC collects stack samples from each of the Paradyn daemons and uses the samples to update its *function count graph*. A function count graph records the number of times each function appears in the stack samples. It also reflects the call relationships between functions as indicated in the stack samples. Nodes in the graph represent functions of the application and edges represent a call relationship between two functions. Each node holds a count of the number of times the function was observed in the stack samples. For instance, assume the PC collects the stack samples shown in Fig. 3 (a) (where $x \longrightarrow y$ denotes that function x called function y). Fig. 3 (b) shows the function count graph resulting from these samples. In the figure, node labels indicate the function and its count. Once the PC has updated the function count graph with the latest stack sample information, it traverses the graph to find functions whose frequency is higher than a user-configurable *deep starter threshold*. This threshold is expressed as a percentage of the total number of stack samples collected.

A⟶B⟶C⟶D
A⟶E⟶C⟶D
A⟶F⟶D
A⟶F⟶G

(a)



(b)

Fig. 3. A set of stack samples (a) and the resulting function count graph (b)

In reality, the PC's function count graph is slightly more complicated than the graph shown in Fig. 3. One of the strengths of the PC is its ability to examine application behavior at per-host and per-process granularity. Deep Start keeps global, per-host, and per-process function counts to enable more fine-grained deep starter selections. For example, if the PC has refined the experiments in a search path to examine process 1342 on host `cham.cs.wisc.edu,` Deep Start will only use function counts from that process' stack samples when selecting deep starters to add to the search path. To enable fine-grained deep starter selections, each function count graph node maintains a tree of counts as shown in Fig. 4. The root of each node's *count-tree* indicates the number of times the node's function was seen in all stack samples. Subsequent levels of the count-tree indicate the number of times the function was observed in stack samples for specific hosts and specific processes. With count-trees in the function count graph, Deep Start can make per-host and per-process deep starter selections

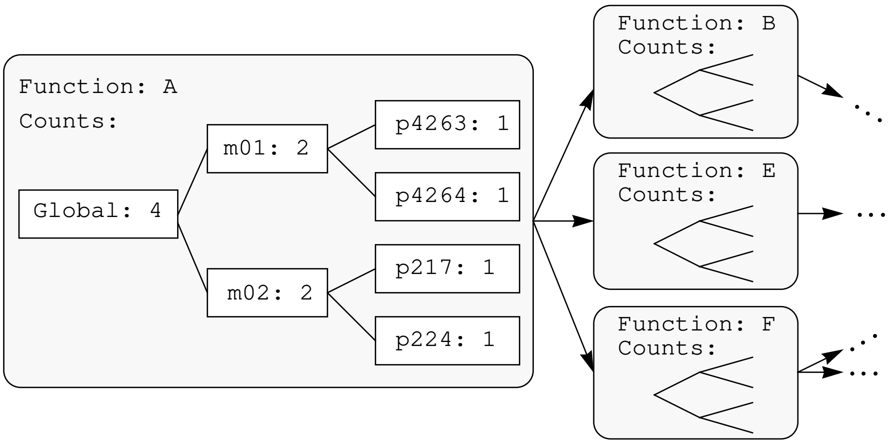


Fig. 4. A function count graph node with count-tree

As Deep Start traverses the function count graph, it may find connected subgraphs whose nodes' function counts are all above the deep starter threshold. In this case, Deep Start selects the function for the deepest node in the subgraph (i.e., the node furthest from the function count graph root) as the deep starter. Given the PC's call-graph-based refinement scheme when examining application code, the deepest node's function in an above-threshold subgraph is the most specific potential bottleneck for the subgraph and is thus the best choice as a deep starter.

2.2 Adding Deep Starters

Once a deep starter function is selected, the PC creates an experiment for the deep starter and adds it to its search. The experiment *E* whose refinement triggered the deep starter selection algorithm determines the nature of the deep starter's experiment. The deep starter experiment uses the same hypothesis and focus as *E*, except that the portion of *E*'s focus that specifies code resources is replaced with the deep starter function. For example, assume the experiment *E* is

hypothesis:	CPU bound
focus:	< /Code/om3.c/main, /Machine/c2-047/om3{1374} >

(that is, it examines whether the inclusive CPU utilization of the function `main` in process 1374 on host c2-047 is above the "CPU bound" threshold). If the PC selects `time_step` as a deep starter after refining *E*, the deep starter experiment will be

hypothesis:	CPU bound
focus:	< /Code/om3.c/time_step, /Machine/c2-047/om3{1374} >.

Also, Deep Start assigns a high priority to deep starter experiments so that they are given precedence when the PC activates experiments from its pending queue.

With the PC's current call-graph-based search strategy, the PC's search history graph reflects the application's call graph when the PC is searching through the application's code. Deep Start retains this behavior by creating as many *connecting experi-*

ments as necessary to connect the deep starter experiment to some other experiment already in the search history graph. For example, in the search history graph in Fig. 1 the PC chose `p_makeMG` as a deep starter and added connecting experiments for functions `a_anneal`, `a_neighbor`, and `p_isvalid`. Deep Start uses its function count graph to identify connecting experiments for a deep starter experiment. Deep Start gives medium priority to the connecting experiments so that they have preference over the background call-graph search but not over the deep starter experiments.

3 Evaluation

To evaluate the Deep Start search algorithm, we modified the PC to search using either the Deep Start or the current call graph-based search strategy. We investigated the sensitivity of Deep Start to the deep starter threshold, and chose a threshold for use in our remaining experiments. We then compared the behavior of both strategies while searching for performance problems in several scientific applications. Our results show that Deep Start finds bottlenecks more quickly and often finds more bottlenecks than the call-graph-based strategy.

During our experimentation, we wanted to determine whether one search strategy performed "better" than the other. To do this, we borrow the concept of *utility* from consumer choice theory in microeconomics [15] to reflect a user's preferences. We chose a utility function $U(t) = -t$ where t is the elapsed time since the beginning of a search. This function captures the idea that users prefer to obtain results earlier in a search. For a given search, we weight each bottleneck found by U and sum the weighted values to obtain a single value that quantifies the search. When comparing two searches with this utility function, the one with the smallest absolute value is better.

3.1 Experimental Environment

We performed our experiments on two sequential and two MPI-based scientific applications (see Table 1). The MPI applications were built using MPICH [9], version 1.2.2. Our PC modifications were made to Paradyn version 3.2. For all experiments, we ran the Paradyn front-end process on a lightly-loaded Sun Microsystems Ultra 10 system with a 440 MHz Ultra SPARC IIi processor and 256 MB RAM. We ran the sequential applications on another Sun Ultra 10 system on the same LAN. We ran the MPI applications as eight processes on four nodes of an Intel x86 cluster running Linux, kernel version 2.2.19. Each node contains two 933 MHz Pentium III processors and 1 GB RAM. The cluster nodes are connected by a 100 Mb Ethernet switch.

3.2 Deep Start Threshold Sensitivity

We began by investigating the sensitivity of Deep Start to changes in the deep starter threshold (see Sect. 2.1). For one sequential (ALARA) and one parallel application (om3), we observed the PC's behavior during searches with thresholds 0.2, 0.4, 0.6, and 0.8. We performed five searches per threshold with both applications. We observed that smaller thresholds gave better results for the parallel application. Although the 0.4 threshold gave slightly better results for the sequential application, the difference between Deep Start's behavior with thresholds of 0.2 and 0.4 was small. Therefore, we

Name	Version	Type	Language	Domain	Size
DRACO	6.0	Sequential	Fortran 90	Hydrodynamic simulation	68981 lines 18632 KB 398 functions
ALARA	2.4.4	Sequential	C++	Induced radioactivity analysis	19576 lines 2911 KB 720 functions
om3	1.5	Parallel (MPI)	C	Global ocean climate simulation	2674 lines 385 KB 36 functions
su3_rmd	6	Parallel (MPI)	C	Quantum chromodynamics pure gauge lattice theory simulation	35845 lines 511 KB 189 functions

Table 1. Characteristics of the applications used to evaluate Deep Start

decided to use 0.2 as the deep starter threshold for our experiments comparing the Deep Start and the call graph search strategy.

3.2.1 Comparison of Deep Start and Call Graph Strategy

Once we found a suitable deep starter threshold, we performed experiments to compare Deep Start with the PC's existing call graph search strategy. For each of our test applications, we observed the behavior of ten PC searches, five using Deep Start and five using the call graph strategy. Fig. 5 shows search profiles for both Deep Start and call graph search strategies for each of our test applications. These charts relate the bottlenecks found by a search strategy with the time they were found. The charts show the cumulative number of bottlenecks found as a percentage of the total number of known bottlenecks for the application. Each curve in the figure shows the average time over five runs to find a specific percentage of an application's known bottlenecks. Range bars are used to indicate the minimum and maximum time each search strategy needed to find a specific percentage across all five runs. In this type of chart, a steeper curve is better because it indicates that bottlenecks were found earlier and more rapidly in a search. Table 2 summarizes the results of these experiments for each of our test applications, showing the average number of experiments attempted, bottlenecks found, and weighted sum for comparison between the two search strategies.

For each application, Deep Start found bottlenecks more quickly than the current call graph search strategy as evidenced by the average weighted sums in Table 2 and the relative slopes of the curves in Fig. 5. Across all applications, Deep Start found half of the total known bottlenecks an average of 32% to 59% faster than the call graph startegy. Deep Start found all bottlenecks in its search an average of 10% to 61% faster than the call graph strategy. Although Table 2 shows that Deep Start tended to perform more experiments than the call graph search strategy, Deep Start found more bottlenecks when the call graph strategy found fewer than 100% of the known bottlenecks. Our results show that Deep Start finds bottlenecks more quickly and may find more bottlenecks than the current call graph search strategy.

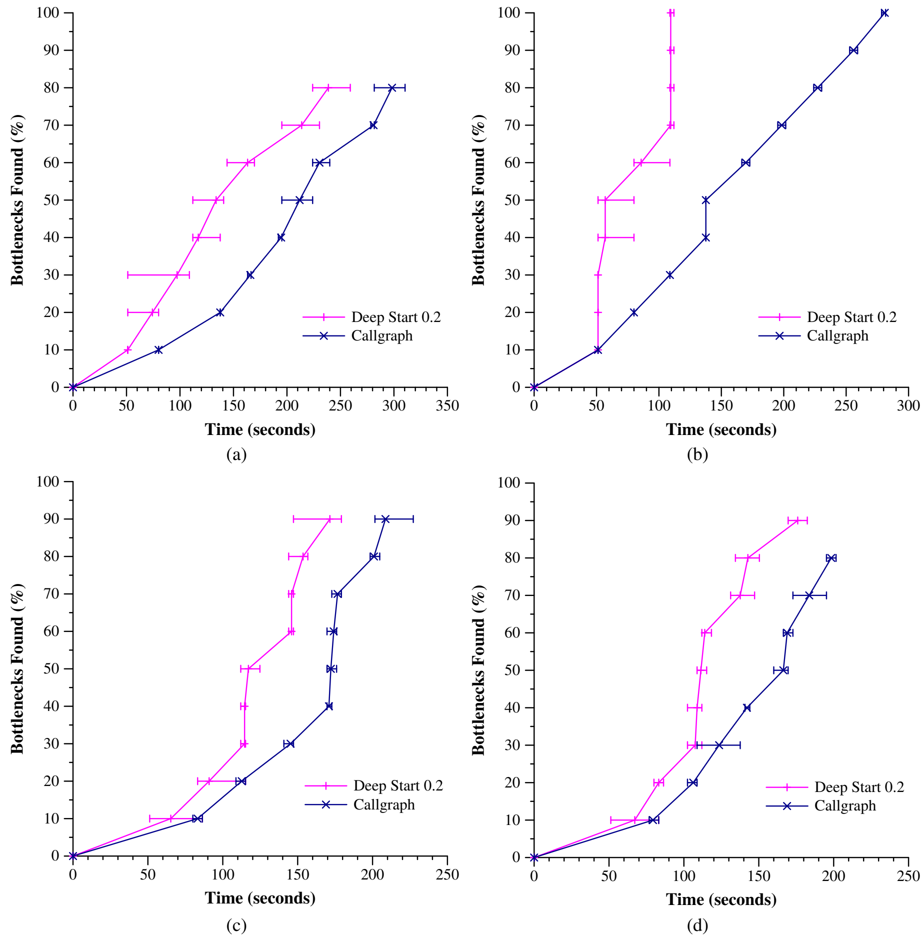


Fig. 5. Profiles for Deep Start and call graph searches on (a) ALARA, (b) DRACO, (c) om3, and (d) su3_rmd. Each curve represents the average time taken over five runs to find a specific percentage of the application's total known bottlenecks. The range bars indicate the best and worst time taken to find each percentage across the five runs

4 Related Work

Whereas Deep Start uses stack sampling to enhance its normal search behavior, several tools use sampling as their primary source of application performance data. Most UNIX distributions include the prof and gprof [8] profiling tools for performing flat and call graph-based profiles, respectively. Quartz [2] addressed the shortcomings of prof and gprof for parallel applications running on shared memory multiprocessors. ProfileMe [1] uses program counter sampling in DCPI to obtain low-level information about instructions executing on in-order Alpha [5] processors. Recognizing the limitations of the DCPI approach for out-of-order processors, Dean et al [6] designed hardware support for obtaining accurate instruction profile information from these types of

Application	Total Known Bottlenecks	Search Type	Average Experiments Attempted	Average Bottlenecks Found	Average Weighted Sum
ALARA	46	Call Graph	174.0	39.4 (86%)	-191.9
		Deep Start	173.4	39.8 (87%)	-134.1
DRACO	18	Call Graph	105.0	18.0 (100%)	-152.7
		Deep Start	105.0	18.0 (100%)	-75.2
om3	145	Call Graph	261.6	141.8 (98%)	-158.1
		Deep Start	269.0	142.2 (98%)	-124.5
su3_rmd	85	Call Graph	260.0	75.6 (89%)	-141.8
		Deep Start	261.4	82.2 (97%)	-114.3

Table 2. Summary of Deep Start/Call Graph comparison experiments. “Total Known Bottlenecks” is the number of unique bottlenecks observed during any search on the application, regardless of search type and deep starter threshold

processors. Each of these projects use program counter sampling as its primary technique for obtaining information about the application under study. In contrast, Deep Start collects samples of the entire execution stack. Sampling the entire stack instead of just the program counter allows Deep Start to observe the application’s call sequence at the time of the sample and to incorporate this information into its function count graph. Also, although our approach leverages the advantages of sampling to augment automated search, sampling is not sufficient for replacing the search. Sampling is inappropriate for obtaining certain types of performance information such as inclusive CPU utilization and wall clock time, limiting its attractiveness as the only source of performance data. Deep Start leverages the advantages of both sampling and search in the same automated performance diagnosis tool.

Most introductory artificial intelligence texts (e.g., [16]) describe heuristics for reducing the time required for a search through a problem state space. One heuristic involves starting the search as close as possible to a goal state. We adapted this idea for Deep Start, using stack sample data to select deep starters that are close to the goal states in our problem domain—the bottlenecks of the application under study. Like the usual situation for an artificial intelligence problem search, one of our goals for Deep Start is to reduce the time required to find solutions (i.e., application bottlenecks). In contrast to the usual artificial intelligence search that stops when the first solution is found, Deep Start should find as many “solutions” as possible.

The goal of our Deep Start work is to improve the behavior of search-based automated performance diagnosis tools. The APART working group [3] provides a forum for discussing tools that automate some or all of the performance analysis process, including some that search through a problem space like Paradyn’s Performance Consultant. For example, the Poirot approach [10] uses heuristic classification as a control strategy to guide an automated search for performance problems. Also, FINESSE [14] supports a form of search refinement across a sequence of application runs to provide performance diagnosis functionality. Search-based automated performance diagnosis tools like these should benefit from the Deep Start approach if they have low-cost access to information that allows them to “skip ahead” in their search space.

5 Conclusions

With the Deep Start search algorithm we have found that a hybrid approach combining stack sampling with automated search can outperform the Performance Consultant's current call graph-based search strategy. Our experiments show that Deep Start finds bottlenecks more quickly than the PC's call graph strategy. Also, our experiments show that Deep Start finds bottlenecks hidden from the simpler search strategy.

References

[1] Anderson, J.M., Berc, L.M., Dean, J., Ghemawat, S., Henzinger, M.R., Leung, S.-T.A., Sites, R.L., Vandevoorde, M.T., Waldspurger, C.A., Weihl, W.E.: Continuous Profiling: Where Have All the Cycles Gone? In Operating Systems Review **31**(5), Proc. 16th ACM Symp. on Operating Systems Principles, Saint Malo, France (1997) 1–14

[2] Anderson, T.E., Lazowska, E.D.: Quartz: A Tool For Tuning Parallel Program Performance. Performance Evaluation Review **18**(1) (1990) 115–125

[3] The APART Working Group on Automatic Performance Analysis: Resources and Tools. `http://www.gz-juelich.de/apart`

[4] Cain, H.W., Miller, B.P., Wylie, B.J.N.: A Callgraph-Based Search Strategy for Automated Performance Diagnosis. In Bode, A., Ludwig, T., Karl, W., Wismüller, R. (eds): LNCS 1900, Proc. 6th Intl. Euro-Par Conf., Munich, Germany (2000) 108–122

[5] Compaq Corporation: 21264/EV68A Microprocessor Hardware Reference Manual. Part Number DS-0038A-TE (2000)

[6] Dean, J., Hicks, J.E., Waldspurger, C.A., Weihl, W.E., Chrysos, G.: ProfileMe: Hardware Support for Instruction-Level Profiling on Out-of-Order Processors. In Proc. 30th Annual IEEE/ACM Intl. Symp. on Microarchitecture, Research Triangle Park, North Carolina, USA (1997) 292–302

[7] Gerndt, H.M., Krumme, A.: A Rule-Based Approach for Automatic Bottleneck Detection in Programs on Shared Virtual Memory Systems. In Proc. 2nd Intl. Workshop on High-Level Programming Models and Supportive Environments, Geneva, Switzerland (1997) 93–104

[8] Graham, S., Kessler, P., McKusick, M.: An Execution Profiler for Modular Programs. Software—Practice & Experience **13** (1983) 671-686

[9] Gropp, W., Lusk, E., Doss, N., Skjellum, A.: A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. Parallel Computing **22**(6) (1996) 789–828

[10] Helm, B.R., Malony, A.D., Fickas, S.F.: Capturing and Automating Performance Diagnosis: the Poirot Approach. In Proc. 1995 Intl. Parallel Processing Symposium, Santa Barbara, California, USA (1995) 606–613

[11] Hollingsworth, J.K., Miller, B.P., Cargille, J.: Dynamic Program Instrumentation for Scalable Performance Tools. In Proc. 1994 Scalable High Performance Computing Conf., Knoxville, Tennessee, USA (1994) 841–850

[12] Karavanic, K.L., Miller, B.P.: Improving Online Performance Diagnosis by the Use of Historical Performance Data. In Proc. SC'99, Portland, Oregon, USA (1999)

[13] Miller, B.P., Callaghan, M.D., Cargille, J.M., Hollingsworth, J.K., Irvin, R.B., Karavanic, K.L., Kunchithapadam, K., Newhall, T.: The Paradyn Parallel Performance Measurement Tool. IEEE Computer **28**(11) (1995) 37–46

[14] Mukerjee,N., Riley, G.D., Gurd, J.R.: FINESSE: A Prototype Feedback-Guided Performance Enhancement System. In Proc. 8th Euromicro Workshop on Parallel and Distributed. Processing, Rhodos, Greece (2000) 101–109

[15] Pindyck, R.S., Rubinfeld, D.L.: Microeconomics. Prentice Hall, Upper Saddle River, New Jersey (2000)

[16] Rich, E., Knight, K.: Artificial Intelligence. McGraw-Hill, New York (1991)