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

Philip C. Roth Barton P. Miller

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

Abstract

To attack the problem of scalability of performance diagnosis tools with respect to application code size, we have developed the Deep Start search strategy—a new technique that uses stack sampling to augment an automated search for application performance problems. Our hybrid approach locates performance problems more quickly and finds performance problems hidden from a more straightforward search strategy. The Deep Start strategy uses stack samples collected as a by-product of normal search instrumentation. Using these samples, our strategy selects *deep starters*—functions that are likely to be application bottlenecks and thus are good locations to consider early in the search. With priorities and careful control of the search refinement, our strategy gives preference to experiments on the deep starter functions and their callees. This approach enables the Deep Start strategy to find application bottlenecks more efficiently and more effectively than a more straightforward search strategy.

We implemented the Deep Start search strategy in the Performance Consultant, Paradyn's automated bottleneck detection component. In our tests, the Deep Start strategy found half of all known bottlenecks between 25% and 63% faster on average than the Performance Consultant's current search strategy. The Deep Start strategy found all bottlenecks in its search between 7% and 61% faster on average than the current strategy.

Keywords: Automated performance diagnosis, search, sampling.

1 Introduction

An automated search for performance problems is an effective strategy for performance diagnosis [7,10,14,15]. A user need not be an expert in performance analysis to be productive using an automated performance problem search tool; the analysis expertise is embodied in the tool's search strategy. The capability to automate searches for performance problems is enhanced by incorporating structural information about the application under study such as its call graph [4], or by pruning and prioritizing the space in which the search takes place based on the application's behavior during previous runs [13]. We have developed a new technique that uses sampling [1,2,8] to augment an automated search for performance problems to attack the problem of code size scalability. Our hybrid approach, called Deep Start, substantially improves a tool's search effectiveness by locating performance problems more quickly and by locating performance problems hidden from a more straightforward search strategy. To demonstrate its power, we have implemented our hybrid search strategy in the Performance Consultant, the automated search component of the Paradyn performance tool [14].

Under control of the Performance Consultant, Paradyn performs an automated search for performance problems using dynamic instrumentation technology [11]. To implement its search, the Performance Consultant forms *hypotheses* about potential performance problems in the application and inserts dynamic instrumentation to evaluate whether the hypotheses are true. The Performance Consultant begins its search by considering general performance problems

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.

at broad scope (*focus*, in Paradyn terminology), such as whether the entire application is spending too much time performing synchronization operations according to a user-configurable threshold. If the Performance Consultant finds a hypothesis to be true, it refines its search by testing more specific hypotheses. For example, if the Performance Consultant has found the application as a whole to be spending too much time doing synchronization, it refines its search to consider whether the application is spending too much time performing synchronization operations in the processes of the application.

The current Performance Consultant search strategy [4] uses the application's call graph structure to guide its refinement through the application code. For example, if the Performance Consultant has found that an MPI application as a whole is spending too much time sending messages, it evaluates whether the application is spending too much time sending messages in the application's `main` function (including all of its direct and indirect callees). Subsequently, if the Performance Consultant finds that the application is spending too much time sending messages in the `main` function and its callees, it refines its search to form individual hypotheses about the functions that `main` calls directly and all of their callees. Note that the search branches at this point if the `main` function has more than one callee. The Performance Consultant prunes its search on a particular refinement path when it encounters a function for which its hypothesis is found to be false.

The Deep Start search strategy augments this call-graph-based refinement strategy with stack sample information to guide the search to performance problems more quickly. This sampling information is gathered as a by-product of normal instrumentation. In particular, to ensure that dynamic instrumentation is inserted safely Paradyn performs a stack walk in the application processes to determine whether they are executing in code that will be overwritten by instrumentation. The Deep Start search strategy uses stack samples collected early in the search, when the Performance Consultant is testing for general performance problems in the entire application. When the Deep Start strategy refines its search to consider hypotheses for individual functions, it uses the stack sample information to begin searching not only at the application's `main` function, but also at functions which appear frequently in the collected stack samples. In general, functions that appear frequently in the stack samples are long-running or are called many times. In either case, they are likely to be closer in terms of the automated search's refinement strategy to the functions exhibiting the actual performance problems.

The use of stack sample information makes the Deep Start search strategy more efficient than the current Performance Consultant search strategy. By considering functions that appear frequently in its stack samples, the Deep Start search strategy can "skip ahead" in the search space so as to start looking for performance problems closer to where they actually exist in the application. This ability allows the Deep Start strategy to detect performance problems more quickly than the current Performance Consultant search strategy that follows the application's call graph structure. Nevertheless, the Deep Start strategy incorporates the standard call graph-based search strategy at a lower priority to find performance problems not indicated by the stack sample data alone.

The Deep Start search is more effective than the current Performance Consultant search strategy because it is able to find performance problems hidden from the current strategy. For example, consider the portion of an application's call graph shown in Figure 1. The call graph strategy will prune its search after finding A to be a bottleneck if B, C, and D are not bottlenecks, 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 search strategy, the likelihood that it will be considered is high if it occurs frequently in the stack samples.

It is not desirable to replace dynamic instrumentation entirely with sampling in a performance tool. Using sampling, it is possible to collect inclusive CPU performance data—performance data for a function and all of its direct

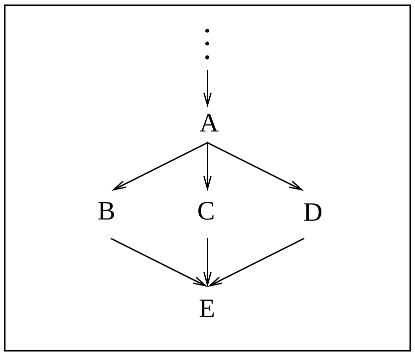


Figure 1: An application call graph with “hidden” performance bottleneck E.

and indirect callees—by sampling not only the current program counter but the program counters for all functions on the stack, and by correlating the stack samples with the application’s call graph. While this approach makes collection of inclusive CPU performance data possible using sampling, it is complicated and expensive. In contrast, collecting inclusive performance data using dynamic instrumentation is trivial, requiring only instrumentation to sample a timer at function entry and exit and to form the difference between these timer samples. Furthermore, sampling is inappropriate for collecting performance data other than CPU time spent executing in a function. For example, it is difficult to collect elapsed time (i.e., wall-clock time) data using sampling, and it is impossible to collect accurate counts of program events using a sampling approach. On the other hand, dynamic instrumentation is well suited for collecting elapsed time data as long as a wall timer (e.g., a system-wide tick counter) is available. Collecting accurate event counts using dynamic instrumentation is as simple as inserting instrumentation to increment a counter whenever an event occurs. Also troublesome is the use of event-driven sampling, in which a sample is collected in response to a low-level event other than the expiration of a timer. For example, a tool may collect samples in response to instruction cache misses. Without hardware support [6], event-driven sampling is problematic on modern out-of-order microprocessors such as the Compaq Alpha 21264 [5] and the Intel Pentium 4 [12] due to their imprecise interrupts. Imprecise interrupts cause an unpredictable delay between the occurrence of an event and the delivery of the interrupt triggering the sample, making it impossible for a sampling-based tool to associate an event with the instruction (and hence the function) that caused the event to occur. Finally, it is possible for application behavior to be missed using a sampling approach due to its statistical nature. In light of these limitations of the sampling approach, it is not desirable to replace dynamic instrumentation entirely with sampling. Instead, we leverage the strengths of both sampling and dynamic instrumentation in the Deep Start search strategy.

2 The Performance Consultant

The Performance Consultant is Paradyn’s automated bottleneck detection component. It searches for application bottlenecks by performing experiments that determine whether the application is experiencing performance problems and, if so, where the problems are occurring. Initially, the experiments test the behavior of the application as a whole and, if the Performance Consultant discovers bottlenecks in the application, it refines the experiments to be more and more specific about the nature and location of each bottleneck.

Each experiment tests a *hypothesis*—a reason why the application may be performing poorly. For example, one of the hypotheses tested by the Performance Consultant is that the application is spending too much time performing blocking I/O operations. When the Performance Consultant activates an experiment, it inserts instrumentation code into the application to collect performance data that allows it to evaluate whether the experiment’s hypothesis is true.

An experiment's *focus* helps determine where the experiment's instrumentation code is inserted. A focus denotes a set of application *resources*—locations where the application may be performing poorly. The functions that comprise the application's executable code are resources, as are the application's processes and the systems on which they are running. The synchronization objects used by the application such as barriers, spin locks, semaphores, and MPI communicators and message tags also are application resources. As shown in Figure 2, Paradyn organizes application resources into hierarchies by type. A given resource represents all resources organized beneath it within its hierarchy. For example, beneath the top-level Machine resource are resources for each system on which application processes are running, and beneath each system resource are resources for the application processes running on that system. As a concise name for a potentially large set of resources, a focus takes advantage of this hierarchical resource organization. A focus is a tuple containing one resource from each resource hierarchy. For example, the focus `</Code/setup.c,/Machine/lc01.cs.wisc.edu,/SyncObject>` denotes all functions from the source file `setup.c`, but only if they are executing in processes running on the host named `lc01.cs.wisc.edu`. This focus specifies the top level SyncObject resource, so it does not constrain the set of resources it names to any synchronization object or type of synchronization object.

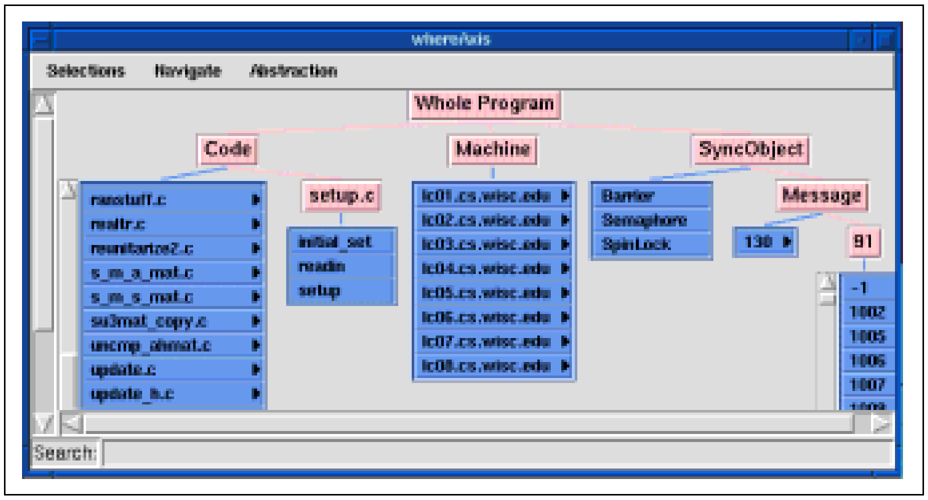


Figure 2: Application resources arranged in hierarchies.

Using a technique called *refinement*, the Performance Consultant continues its bottleneck search from a true experiment (i.e., an experiment whose hypothesis is true at its focus) by creating one or more new experiments that are more specific than the original experiment. The new experiments have either a more specific hypothesis or a more specific focus than the original experiment. Because the Performance Consultant uses a small number of predefined hypotheses, search refinement usually involves the refinement of an experiment's focus. To refine a focus, the Performance Consultant replaces one of the resources in the focus with another of the same type. If the resource being replaced is a Code resource, the Performance Consultant generates the new foci using the callees of the function represented by the Code resource. For example, if the Performance Consultant refines the focus `</Code/setup.c/setup,/Machine/lc01.cs.wisc.edu,/SyncObject>` by replacing its Code resource, and the `setup` function calls the functions `setup.c/initial_set`, `ranstuff.c/initialize_prn`, and `make_lattice.c/make_lattice`, the resulting foci are:

`</Code/setup.c/initial_set,/Machine/lc01.cs.wisc.edu,/SyncObject>`,
`</Code/ranstuff.c/initialze_prn,/Machine/lc01.cs.wisc.edu,/SyncObject>`, and
`</Code/make_lattice.c/make_lattice,/Machine/lc01.cs.wisc.edu,/SyncObject>`.

If the resource being replaced is not a Code resource, the Performance Consultant replaces it with its children from its resource hierarchy. For example, if the Performance Consultant refines the sample focus by replacing its Machine resource, and the application has two processes running on host `lc01.cs.wisc.edu` with IDs 1427 and 3265, the resulting foci are:

`</Code/setup.c/setup,/Machine/lc01.cs.wisc.edu/1427,/SyncObject>` and
`</Code/setup.c/setup,/Machine/lc01.cs.wisc.edu/3265,/SyncObject>`.

In all cases, focus refinement (and hence most of the Performance Consultant search) involves taking one step at a time through the application's call graph or its resource hierarchies.

As it builds a *search path*—a sequences of experiments related by refinement—the Performance Consultant searches entirely through one resource hierarchy (or the application's call graph) before starting to search another. It does this by attempting to refine along the same resource type for successive focus refinements within a search path. For example, if the Performance Consultant generates new experiments from a true experiment using a focus refinement in which it replaces the Code resource, it will also try to replace the Code resource in any focus refinements from the new experiments. If it cannot refine the focus by replacing the Code resource (i.e., the function represented by that resource has no callees), the Performance Consultant will start searching through another resource hierarchy by replacing another type of resource. The Performance Consultant also starts searching through a new resource hierarchy when the last experiment on a path is found to be false. The Performance Consultant terminates a search path when it cannot refine the last experiment on the path. Because the refinement of an experiment may result in the creation of more than one new experiment, the Performance Consultant may have many search paths that are simultaneously active (i.e., not terminated). To ensure that it does not overly perturb the application under study with experiments from a large number of active search paths, the Performance Consultant observes a user-configurable limit on the amount of instrumentation inserted into the application at any time.

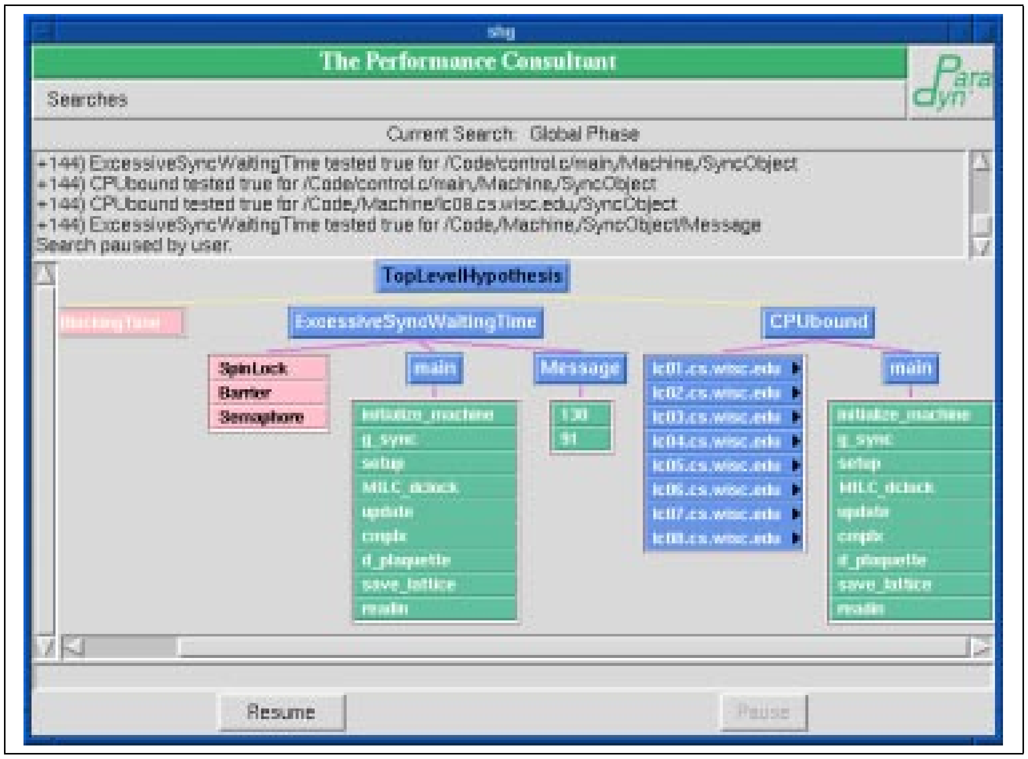
The graph that records the cumulative refinements of a search is called a *search history graph*. As shown in Figure 3, Paradyn provides a visual representation of this graph in its Search History Graph display. The display is dynamic—nodes are added as the Performance Consultant refines its search. Besides providing a visual record of the search performed by the Performance Consultant, this 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 Performance Consultant is collecting performance data for the experiment), and whether the performance data collected for an experiment indicates the experiment's hypothesis to be true, false, or not yet known.

3 The Deep Start Search Strategy

Under the Deep Start search strategy, the Performance Consultant uses stack samples collected as a by-product of dynamic instrumentation to guide its search. Paradyn incorporates a single front-end process with one or more daemons controlling application processes. Because the Paradyn daemons handle the insertion of dynamic instrumentation and the stack samples are performed during the insertion of the instrumentation, the daemons collect these stack samples and send them in batches to the Performance Consultant upon request. During its search, the Performance Consultant analyzes the stack samples to find *deep starters*—functions that are likely to be application performance problems in the application based on their frequency of occurrence in the stack samples. It creates experiments for the deep starter functions with high priority so that they will be given preference when the Performance Consultant chooses new experiments to perform.

3.1 Stack Sampling

The stack samples used by the Deep Start search strategy are collected as a side effect of the insertion of instrumenta-



```
+144) ExcessiveSyncWaitingTime tested true for /Code/control.c/main,/Machine,/SyncObject
+144) CPUbound tested true for /Code/control.c/main,/Machine,/SyncObject
+144) CPUbound tested true for /Code,/Machine/lc08.cs.wisc.edu,/SyncObject
+144) ExcessiveSyncWaitingTime tested true for /Code,/Machine,/SyncObject/Message
Search paused by user.
```

Figure 3: Performance Consultant Search History Graph display.

tion code into the application processes. Paradyn daemons insert instrumentation code when requested by some component of the Paradyn front-end process such as the Performance Consultant or the visualization manager. When inserting instrumentation code into a process, the Paradyn daemon controlling the process must ensure that the process is not executing in code that will be overwritten by the code patch. This requirement implies that the Paradyn daemon must consider not only the location where the process is executing, but also the locations to which control will return when the process completes any function calls that are in progress. That is, the Paradyn daemon must consider the current program counter and the return addresses of all frames on the call stack.[2] The Paradyn daemon performs this check by pausing the process and walking its call stack. To support the Deep Start search strategy, each time a Paradyn daemon performs a stack walk it saves the stack walk information. The daemon also records the ID of the process from which the sample was taken.

3.2 Choosing Deep Starters

When the Performance Consultant obtains a batch of stack samples from a Paradyn daemon, it adds the information from each sample to a data structure we call a *function count graph*. The stack samples determine the nodes and edges of this directed graph. Nodes in the graph represent functions of the application, while edges represent a call relationship between two functions as dictated in the stack samples. For each function represented in the graph, the graph keeps a count of the number of times the function was seen in the stack samples. For instance, assume the Performance Consultant collects the following stack samples (where $x \rightarrow y$ denotes that function *x* had called function *y* at the time of the stack sample):

2. Some compilers support a "frame pointer omission" optimization so that formal stack frames are not necessarily present on the stack for all function calls currently in progress. To ensure the safety of dynamic instrumentation in the presence of this compiler optimization, the Paradyn daemons must obtain the return addresses of function calls in progress despite the absence of a formal stack frame.

A→B→C→D
A→E→C→D
A→F→D
A→F→G

Figure 4 shows the function count graph resulting from these samples. In the figure, node labels indicate both the function and its count.

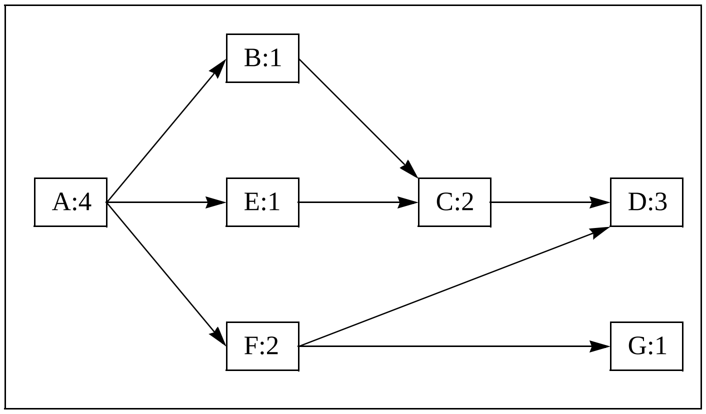


Figure 4: A function count graph.

One of the strengths of the Performance Consultant is its ability to examine not only the behavior of an application as a whole but also its behavior at per-host and per-process granularity. This capability is invaluable for finding performance problems related to workload imbalance or mismatched hardware. The Deep Start search strategy enhances these finer-granularity searches by supporting the selection of deep starters using global, per-host, and per-process data. To support choosing deep starters using function count information at varying granularities, our function count graph is slightly more complicated than was shown in Figure 4. Instead of a single count per function count graph node, we maintain a tree of counts at each node as shown in Figure 5. Each graph node still represents a single function from the application. The root of a *count-tree* contains an overall count for its associated function, counting the number of times the function was seen in all stack samples. The nodes at the first level of this tree count the number of times that the function was observed in the stack samples from specific hosts. The second level of this tree counts of the number of times that the function was observed in the stack samples of specific application processes. Using count-trees in the function count graph, the Performance Consultant can restrict the stack samples it considers when choosing deep starters to make per-host and per-process deep starter selections. For example, if the Performance Consultant is choosing deep starters restricted to a process *p* on a machine *m*, as the Performance Consultant traverses the function count graph it walks the count-tree of each graph node to find the count representing the number of times the node's function was seen in stack samples collected from process *p* on machine *m*. The Performance Consultant will consider adding a node's function as a deep starter only if this count is above its deep starter threshold.

When the Performance Consultant triggers its deep starter selection algorithm, it updates the function count graph by requesting the latest batch of stack samples from each Paradyn daemon controlling application processes. For each stack sample in a batch, the Performance Consultant processes the stack sample, starting at the function on the bottom of the call stack. As it considers each function in a sample, it walks the function count graph to increment the counts associated with the functions. If no node exists for a function in a stack sample, the Performance Consultant creates a new node for the function and initializes its count to one.

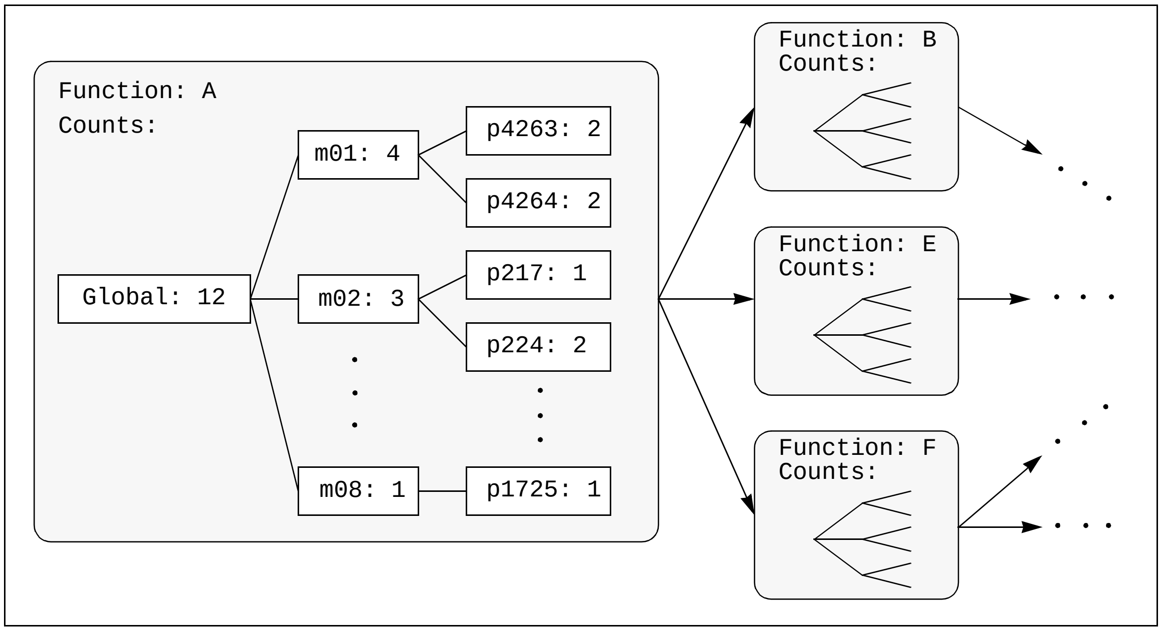


Figure 5: Function count graph nodes with count-tree.

When the Performance Consultant finishes updating the function count graph, it traverses the graph to find functions that are good choices to be deep starters. Functions that execute frequently or that take a long time to execute are good candidates. Fortunately, these functions are likely to be found from the stack sample data by looking for functions that occur frequently in the samples. Therefore, we base our decision for finding deep starter functions on the number of times a function occurs in the stack samples. We choose those functions whose counts are higher than a threshold, where the threshold is computed as a percentage of the total number of stack samples represented in the function count graph. Based on our experience with the initial implementation of the Deep Start search strategy, we default this threshold to be 60% of the stack samples represented in the function count graph. Furthermore, we chose as deep starters only the "deepest" functions from each above-threshold subgraph of the function count graph (i.e., those functions furthest from the root of the function count graph). By using only the deepest function in each above-threshold subgraph as a deep starter, the Performance Consultant focuses its attention on the most likely to be application performance bottlenecks.

The Performance Consultant uses a search path's *Code hook* experiment to determine which function count graph counts to consider when choosing deep starters for the path. A search path's Code hook is the experiment in the path at which the Performance Consultant first began refining the focus in the application's call graph (e.g., where the Performance Consultant refined the focus to the application's `main` function). When the Performance Consultant triggers its algorithm for choosing deep starters for a search path, it extracts the Machine resource from the focus of that search path's Code hook experiment. If the focus' Machine resource is the top-level Machine hierarchy resource, the Performance Consultant uses the global counts as it traverses the function count graph looking for functions whose counts are above the deep starter threshold. If the Machine resource represents a specific host or process, the Performance Consultant traverses the count-tree of each node it visits to find the sample count associated with that host or process.

The Performance Consultant also uses a search path's Code hook experiment to determine when to choose deep starters for the path. The Performance Consultant triggers its deep starter selection algorithm each time it refines an experiment on a search path that already has a Code hook. Because the Performance Consultant waits until a search path has a well-defined Code hook before choosing deep starters for the path, the Performance Consultant is guaranteed to have an appropriate place to add the deep starter experiments into the search as described in the next section.

3.3 Adding Deep Starters

Besides using a search path's Code hook experiment to determine how and when to select deep starters for a search path, the Performance Consultant uses the path's Code hook as a template for the experiments it creates for deep starter functions. It copies the Code hook's hypothesis and focus for deep starter experiments, replacing the Code resource from the Code hook's focus with the resources representing the deep starter functions.

Because the deep starter was chosen for its likelihood to be near the performance bottlenecks of the application, the Performance Consultant adds deep starter experiments at a higher priority than is used for the normal Performance Consultant search experiments. Since priorities are inherited as the Performance Consultant refines its search, the sub-search rooted at the deep starter experiment has precedence over any experiments generated by the normal operation of the Performance Consultant. For applications with a large number of functions, this behavior is highly advantageous. Since the Performance Consultant throttles the amount of active instrumentation to minimize perturbation to the application under study, giving the deep starter experiments high priority means that they will be undertaken before other experiments when the Performance Consultant is able to enable more instrumentation. Therefore, the use of high priority for deep starter experiments allows the Performance Consultant to focus its attention in the search space near the deep starters. Since these functions are likely to be near the actual bottlenecks of the application, the Performance Consultant is likely to find bottlenecks more quickly than with a strict top-down search of its search space.

One of the comforting properties of Paradyn's search history graph is that the search structure reflects the structure of the application's call graph when the Performance Consultant is searching through Code resources and reflects the resource hierarchies when it is searching through non-Code resources. To retain these properties in the presence of the addition of deep starter experiments, the Performance Consultant adds not only an experiment for the deep starter function but also as many experiments as necessary to connect the deep starter experiment to some other experiment that is already present in the search history graph. This behavior is shown in Figure 6, where the Performance Consultant added the experiment for function `p_makeMG` as the deep starter, and the experiments for functions `a_anneal`, `a_neighbor`, and `p_isvalid` as connecting experiments.

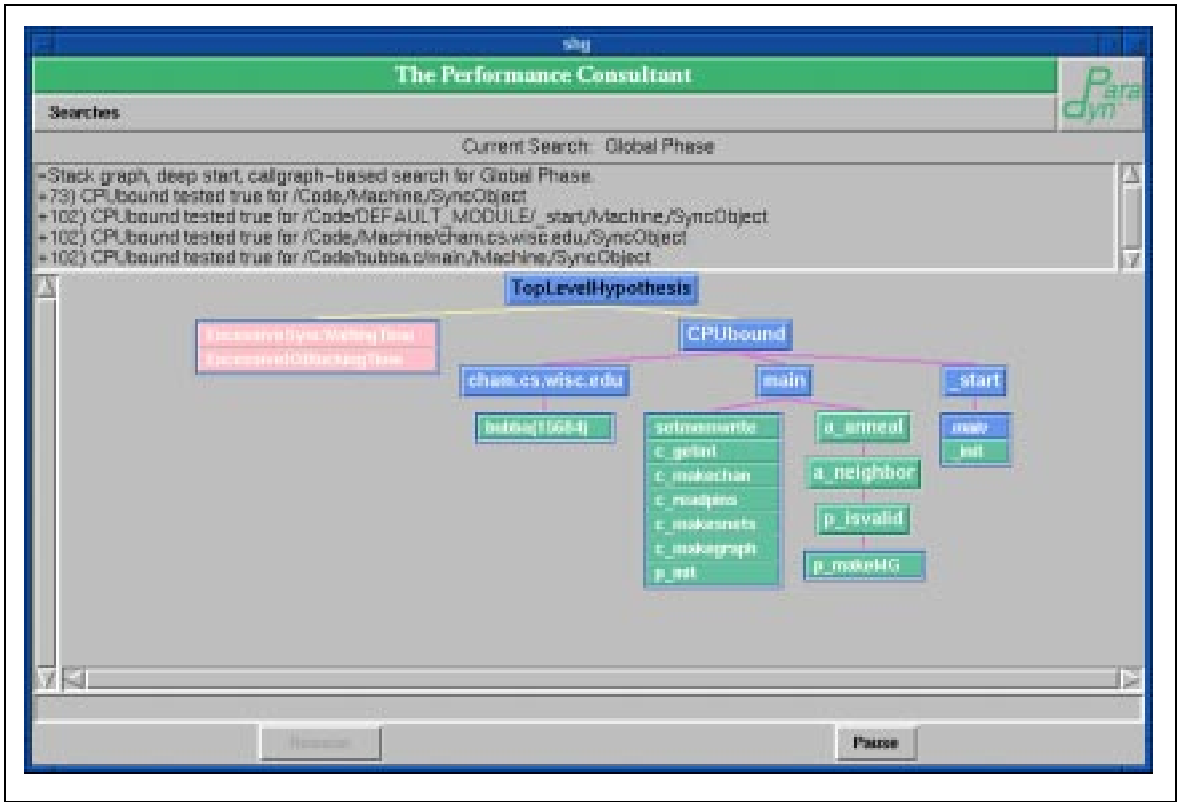
4 Evaluation

To evaluate the Deep Start search strategy, we modified the Performance Consultant to search using both the new strategy and its current call graph-based search strategy. We compared the behavior of the Deep Start strategy and the current strategy while searching for performance problems in several scientific applications.

4.1 Experimental Environment

We performed our experiments on two sequential and two MPI-based scientific applications (see Table 1). The MPI applications were built using version 1.2 of the MPICH [9] MPI implementation. We modified the Performance Consultant within the Paradyn version 3.2 software base, augmenting it to search using either the Deep Start search strategy or its current call graph-based search strategy. We also modified Paradyn's search history graph export facility to export enough information to allow us to reconstruct the search to compare the behavior of the two search strategies.

For all experiments, we ran the Paradyn front-end process on a lightly-loaded Sun Microsystems Ultra 10 system with a 440 MHz UltraSPARC IIi processor and 256 MB RAM. We ran the sequential applications on another Sun



```
-Stack graph, deep start, callgraph-based search for Global Phase.
+73) CPUbound tested true for /Code,/Machine,/SyncObject
+102) CPUbound tested true for /Code/DEFAULT_MODULE/_start/Machine,/SyncObject
+102) CPUbound tested true for /Code,/Machine/cham.cs.wisc.edu,/SyncObject
+102) CPUbound tested true for /Code/bubba.c/main,/Machine,/SyncObject
```

Figure 6: Deep starter and connecting experiments in the Search History Graph display.

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 radio-activity analysis	19576 lines 2911 KB 720 functions
om3	1.5	Parallel (MPI)	C	Global ocean climate simula-tion	2674 lines 385 KB 568 functions
su3_rmd	6	Parallel (MPI)	C	Pure gauge lat-tice theory sim-ulation	35845 lines 511 KB 688 functions

Table 1: Application characteristics.

Ultra 10 system on the same LAN. We ran the MPI applications on eight nodes of an Intel x86 cluster running Linux, kernel version 2.2.17. Each node contains 128 MB RAM and a 600 MHz Pentium III processor with 256 KB L2 cache. The cluster nodes are connected by a 100 Mb Ethernet switch.

4.2 Experimental Methodology

Our experiments consisted of several trials with each application. Each trial consisted of five runs of the application during which we used the Performance Consultant to search for application performance problems using one of the search strategies. During each run, we began the Performance Consultant search once the application reached its steady-state behavior. Once the Performance Consultant search reached a steady state (i.e., the Performance Consultant was not activating any new experiments), we exported the search history graph.

4.3 Experimental Results

We began by investigating the sensitivity of the Deep Start search strategy to changes in the deep starter threshold (see Section 3.2). We performed experiments using a range of deep starter thresholds on one sequential application and one parallel application. Based on the results of these experiments, we selected a single deep starter threshold for use in our remaining experiments. Then we compared the performance of the Deep Start and call graph search strategies for each of our test applications.

For our analysis, we often wanted to choose the best run from a set of runs. To determine whether one run is better than another, we borrow the concept of *utility* from the theory of consumer choice in microeconomics [16] to reflect a tool user's preferences. We postulate a utility function U that captures a user's preference for obtaining results from the tool and we weight the observed results by this utility function. We sum the weighted results to obtain a single value describing the behavior of a search. To capture the idea that users prefer results given earlier in the search, the appropriate utility function is one that is a decreasing function of time. For our analysis, we chose a linear function $U(t) = -t$. With this utility function, we look for the run with the largest weighted sum (i.e., the weighted sum with the smallest absolute value).

Because the `su3_rmd` application implements gather operations using point-to-point message passing functions and uses a unique message tag to label each gather operation, we used a prototype implementation of a resource discovery control mechanism in our `su3_rmd` experiments. For both Deep Start and call graph searches on the `su3_rmd` application, we ignored the discovery of new message tags used by the application so as to avoid swamping the search with experiments on these ephemeral resources. We are currently implementing a more sophisticated approach for handling ephemeral resources, such as message tags and threads, that ignores new resources if their creation rate becomes too high.

4.3.1 Deep Starter Threshold Sensitivity

To investigate the sensitivity of the Deep Start search strategy to changes in the deep starter threshold, we performed trials with the `ALARA` sequential application and the `om3` parallel application with thresholds of 0.2 (i.e., 20% of the collected stack samples), 0.4, 0.6, and 0.8. Table 2 contains a summary of these experiments. The *total known bottleneck count* for an application is the number of distinct bottlenecks found across all runs of the Performance Consultant on that application, regardless of the search strategy or the deep starter threshold used in the runs.

Figures 7 and 8 show profiles of the best searches with `ALARA` and `om3` using a range of deep starter thresholds from 0.2 to 0.8. The 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. In this type of chart, a steeper curve is better because it indicates that bottlenecks were found earlier and more rapidly in a search. Each curve shown in the figures represents the best run out of the five in a trial.

Based on the results from Table 2 and Figures 7 and 8, we chose to report results in the remainder of this paper from experiments using 0.4 as the deep starter threshold. The 0.4 threshold yielded searches for `om3` with the maximum weighted sum for both the best run and the average over all runs in a trial. Furthermore, using this threshold the Deep Start strategy found the most bottlenecks when compared with the other thresholds. For `ALARA`, no single threshold was superior for the number of bottlenecks found and best and average weighted sums. In most cases, the behavior with the 0.4 threshold was close to that with the other thresholds.

Application	Total Known Bottlenecks	Deep Starter Threshold	Average Number of Experiments Attempted	Average Number of Bottlenecks Found	Percentage of Known Bottlenecks Found	Average Weighted Sum	Best Weighted Sum
ALARA	46	0.8	174.0	39.8	86.5%	-158.7	-157.2
		0.6	172.6	40.2	87.4%	-140.6	-133.4
		0.4	173.4	39.8	86.5%	-134.1	-130.6
		0.2	171.4	39.6	86.1%	-133.6	-122.8
om3	163	0.8	419.2	152.4	93.5%	-174.4	-149.8
		0.6	444.8	152.6	93.6%	-189.1	-158.8
		0.4	465.8	152.6	93.6%	-166.6	-149.0
		0.2	496.2	154.4	94.7%	-226.7	-165.5

Table 2: Summary of deep starter threshold sensitivity 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.

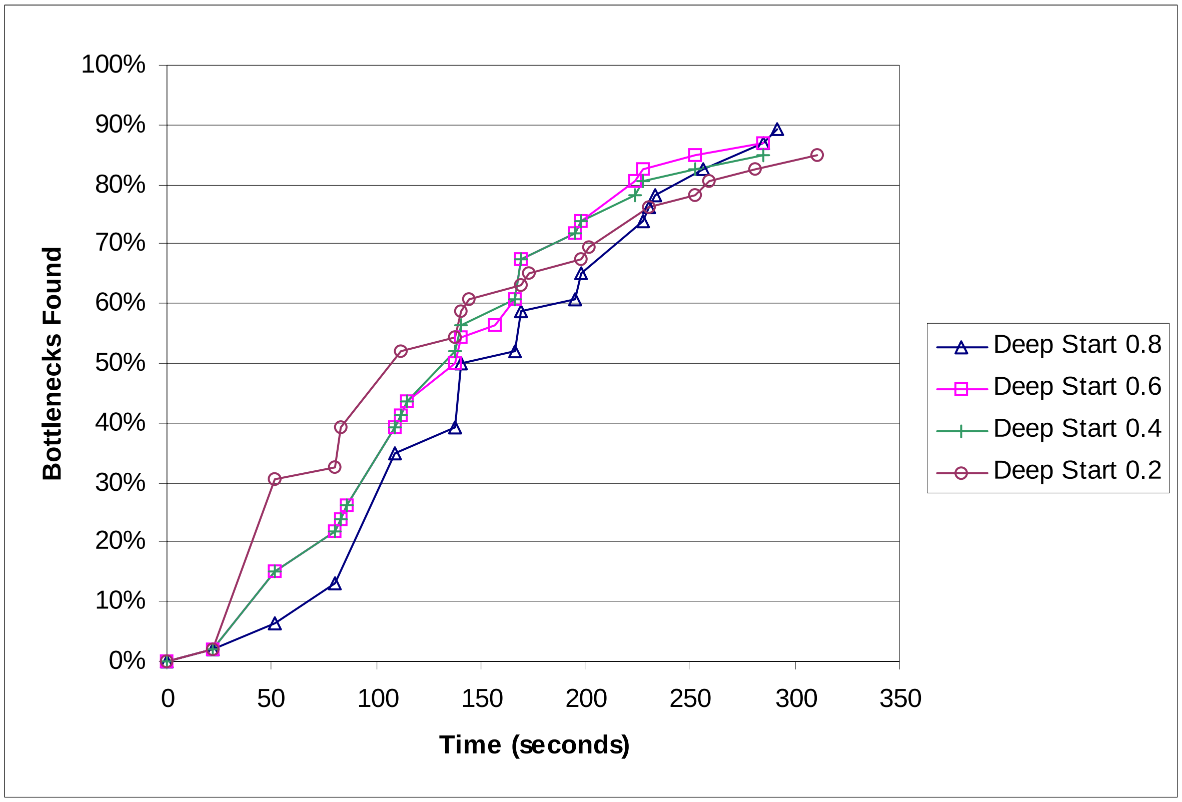


Figure 7: Search profiles for searches on the ALARA application using a range of deep starter thresholds.

4.3.2 Comparison of the Deep Start and Call Graph Searches

In general, the Deep Start search strategy found more bottlenecks than the Performance Consultant's call graph-based search strategy. InTable 3, we present summaries of the Deep Start and the call graph searches for each of our test applications. In cases where both the Deep Start and call graph strategies did not find all of an application's known bottlenecks, the Deep Start strategy found more bottlenecks than the call graph strategy.

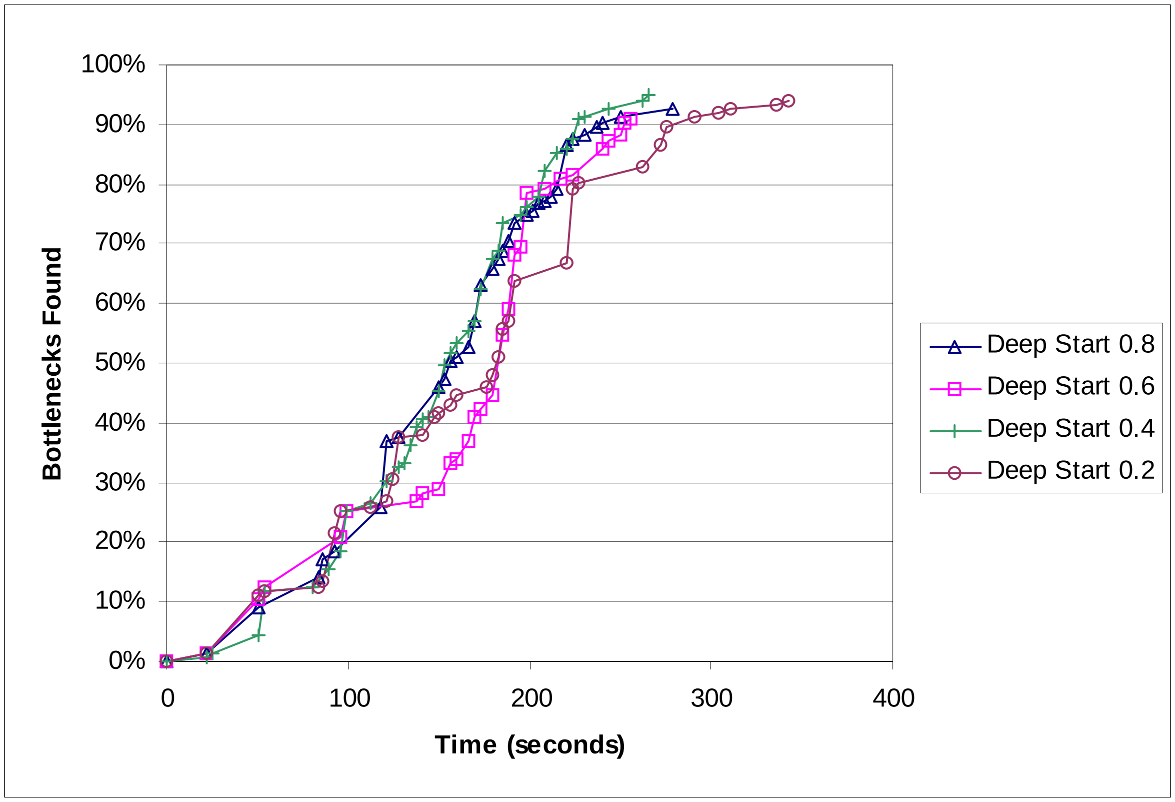


Figure 8: Search profiles for searches on the om3 application using a range of deep starter thresholds.

Application	Total Known Bottlenecks	Search Strategy	Average Experiments Attempted	Average Bottlenecks Found	Percentage of Bottlenecks Found	Average Weighted Sum	Best Weighted Sum
ALARA	46	Call Graph	174.0	39.4	86%	-191.9	-190.2
		Deep Start	173.4	39.8	87%	-134.1	-130.6
DRACO	18	Call Graph	105.0	18.0	100%	-152.7	-152.0
		Deep Start	105.0	18.0	100%	-75.2	-75.2
om3	163	Call Graph	363.0	151.0	93%	-226.3	-207.3
		Deep Start	465.8	152.6	94%	-166.6	-149.0
su3_rmd	114	Call Graph	354.4	102.2	90%	-166.2	-142.7
		Deep Start	369.0	104.2	91%	-141.8	-121.1

Table 3: 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.

As shown in Figures 9, 10, 11 and 12, the Deep Start search strategy produced results more quickly than the call graph strategy. Each figure shows the profile of a Deep Start search and a call graph search for one of our test applications. Each figure represents the best run out of the five in a trial. To quantify the results shown in Figures 9 through 12, we present the time taken to find a given percentage of the total known bottlenecks in Table 4. For our test applications, the Deep Start search strategy found half of the known bottlenecks for each application between

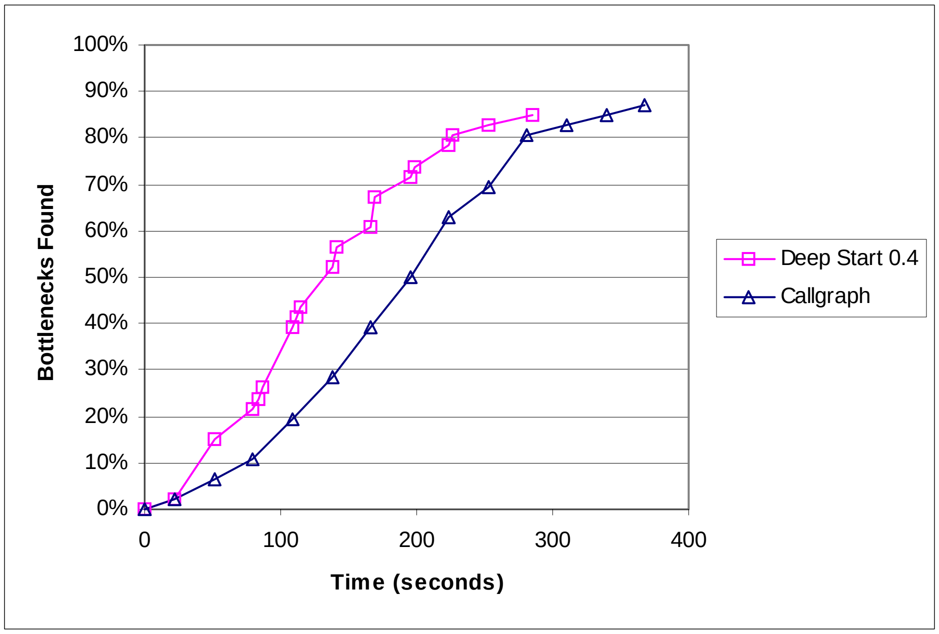


Figure 9: Search profiles for the best observed Deep Start and call graph search on ALARA.

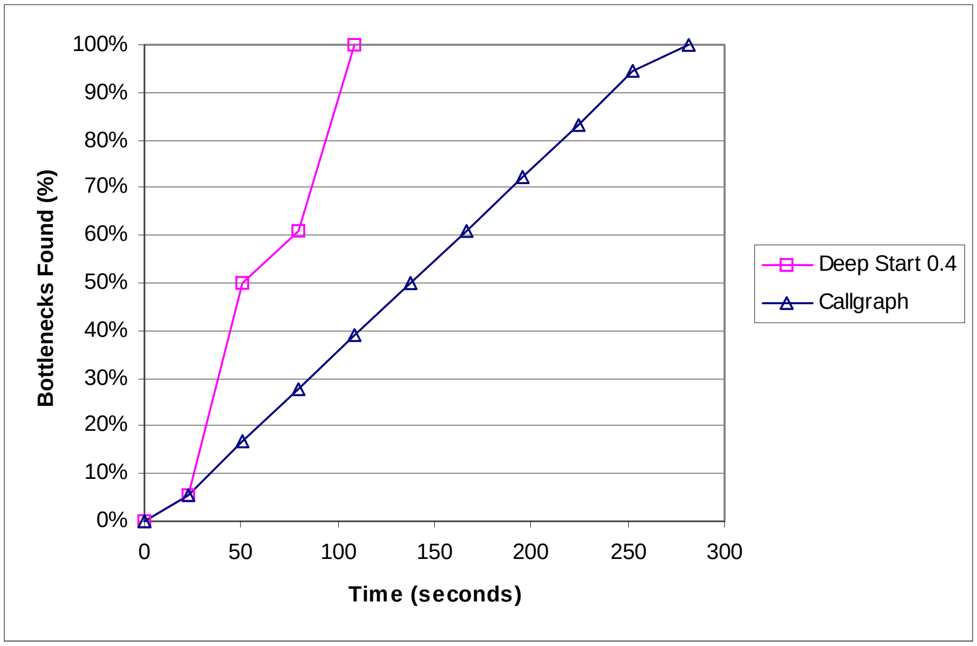


Figure 10: Search profiles for the best observed Deep Start and call graph search on DRACO.

21% and 63% faster on average than the call graph strategy. The Deep Start search strategy found all bottlenecks in its search between 7% and 61% faster on average than the call graph strategy.

5 Related Work

Whereas the Deep Start-enabled Performance Consultant 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

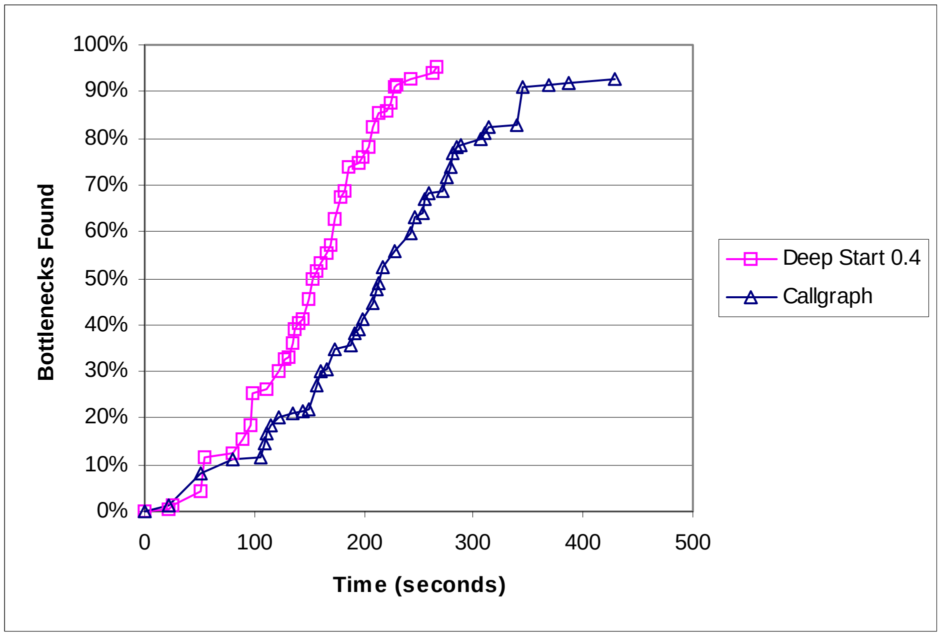


Figure 11: Search profiles for the best observed Deep Start and call graph search on om3.

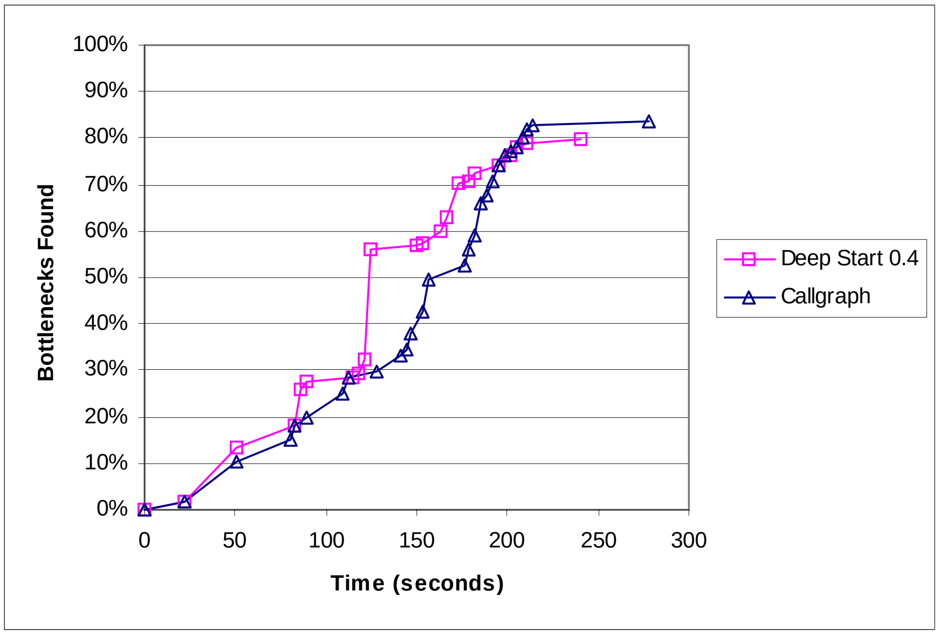


Figure 12: Search profiles for the best observed Deep Start and call graph search on su3_rmd.

the prof and gprof [8] profiling tools for performing flat and call graph-based profiles, respectively. Anderson and Lazowska [2] addressed the shortcomings of prof and gprof for parallel applications running on shared memory multiprocessors. Anderson *et al* [1] used program counter sampling in DCPI to obtain low-level information about instructions executing on in-order Alpha 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

Application	Percentage of Known Bottlenecks Found	Call Graph Average Time Required (sec)	Deep Start Average Time Required (sec)	Change From Call Graph (sec)	Percent Change From Call Graph
ALARA	10%	80.0	51.2	-28.8	-36%
	25%	137.6	85.1	-52.5	-38%
	50%	211.8	138.2	-73.6	-35%
	All Found	375.7	282.9	-92.8	-25%
DRACO	10%	51.2	51.2	0.0	0%
	25%	80.0	51.2	-28.8	-36%
	50%	137.6	51.2	-86.4	-63%
	All Found	281.0	108.8	-172.2	-61%
om3	10%	80.0	52.5	-27.5	-34%
	25%	169.0	113.9	-55.0	-33%
	50%	248.3	184.3	-64.0	-26%
	All Found	414.7	317.4	-97.3	-23%
su3_rmd	10%	51.8	52.5	0.6	1%
	25%	119.0	108.8	-10.2	-9%
	50%	208.6	165.1	-43.5	-21%
	All Found	278.4	256.6	-21.8	-7%

Table 4: Average time required to find specific percentages of the known bottlenecks for each test application. The All Found rows indicate the average time required to find all bottlenecks that were found during a run.

from these types of processors. A form of their ProfileMe design has been incorporated in modern Alpha processors [5]. Each of these projects use program counter sampling as its primary technique for obtaining information about the application under study. In contrast, the Deep Start search strategy collects samples of the entire execution stack.

Most introductory artificial intelligence texts (e.g., [17]) 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 the Deep Start search strategy, 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 the goals of the Deep Start search strategy is to minimize the time required to find solutions (i.e., application bottlenecks). However, in contrast to the usual artificial intelligence search that stops when the first solution is found, whereas our goal is to find as many solutions as possible.

The Deep Start search strategy is an augmentation of our call graph-based search strategy [4]. While both the call graph and the Deep Start search strategies seek to improve the search within one run of an application, Karavanic and Miller [13] showed the benefit of using information collected during previous runs of an application to improve the characteristics of a performance bottleneck search. Like Karavanic and Miller's approach, the Deep Start search strategy uses priorities to control the search behavior. Although the Deep Start search strategy is designed to improve a search using information from the current application run only rather than from several previous runs, the two techniques could be used together.

The goal of the Deep Start search strategy is to improve the behavior of a performance tool that implements an automated search for performance problems. The APART working group [3] provides a forum for discussing tools that also automate some or all of the performance analysis process, including some that search through a problem space similar to Paradyn's Performance Consultant. For example, Helm *et al* [10] describe the Poirot approach that uses heuristic classification as a control strategy to guide an automated search for performance problems. Also, the FINESSE project [15] supports a form of search refinement across a sequence of application runs to provide performance diagnosis functionality.

6 Conclusions

With the Deep Start search strategy we have found that a hybrid approach that augments search with stack sampling information can outperform a call graph-based search strategy. In general, our search strategy is more efficient than the call graph-based search strategy in locating application bottlenecks because it tends to find bottlenecks earlier in the search. Also, the strategy is more effective than the call graph-based search strategy because it can find bottlenecks that are hidden from the simpler strategy. In our experiments, the Deep Start strategy found half of all known bottlenecks for each application between 21% and 63% faster than the call graph search strategy on average, and found all bottlenecks in its search between 7% and 61% faster than the call graph strategy. Also, the Deep Start strategy is more effective than the call graph-based strategy because it can find bottlenecks that are hidden from the simpler strategy.

While our approach leverages the advantages of sampling to augment our automated search, sampling is not sufficient for replacing the search. The inability of sampling to obtain certain types of information, such as inclusive CPU metric information and wall clock time, limit its attractiveness as the only source of performance data for a diagnostic tool. With the Deep Start strategy, we leverage the advantages of both techniques in the same automated search tool.

6.1 Future Work

We are considering a number of enhancements to the Deep Start search strategy to improve its behavior or reduce its overhead. Adding a short, dedicated stack sampling period before the start of the search—a technique we call "priming the pump"—may improve the quality of deep starter selections made early in the search. Modifying the Performance Consultant to take advantage of semantic knowledge about the functions from its stack samples may allow it to tailor its deep starter selections for a search path to the hypothesis of that path's experiments. For example, using a separate function count graph for stack samples that include known communication library functions should improve deep starter selections made for paths with experiments on an "excessive synchronization waiting time" hypothesis. Finally, changing from a "pull model" to a "push model" and building the function count graphs in the Paradyn daemons instead of the front-end process should reduce the Deep Start-related traffic between daemons and the front-end process, improving the scalability of the tool on large machines.

We plan to continue exploring the use of microeconomics analysis techniques to capture user preferences for obtaining results from automated performance diagnosis tools. This work may include a survey of potential tool users to gauge their "willingness to pay" for one results production scheme over another. This information should improve our model of the user preference function; with a better model of user preference, we can look for other heuristics for controlling the bottleneck search so that results are produced according to that model.

Acknowledgments

This paper benefited from the hard work of the past and present members of the Paradyn research group. We wish to thank Victor Zandy, Paul Wilson, Professor Peter Norman of the University of Wisconsin-Madison Department of Economics, and Brian Wylie for fruitful discussions on our topic, and Victor Zandy, Erik Paulson, and David Thomp-

son for their help in enabling the collection of our MPI application results.

References

[1] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl, “Continuous Profiling: Where Have All the Cycles Gone?” *16th ACM Symposium on Operating Systems Principles*, Saint Malo, France, October 1997.

[2] T. E. Anderson and E. D. Lazowska, “Quartz: A Tool For Tuning Parallel Program Performance”, *ACM Conference on Measurement and Modeling of Computer Systems*, Boulder, Colorado, May 1990.

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

[4] H. W. Cain, B. P. Miller, and B. J. N. Wylie, “A Callgraph-Based Search Strategy for Automated Performance Diagnosis”, *Euro-Par 2000*, Munich, Germany, August 2000.

[5] Compaq Corporation, “21264/EV68A Microprocessor Hardware Reference Manual”, Part Number DS-0038A-TE, December 2000.

[6] J. Dean, J.E. Hicks, C.A. Waldspurger, W.E. Weihl, and G. Chrysos, “*ProfileMe*: Hardware Support for Instruction-Level Profiling on Out-of-Order Processors”, *Thirtieth Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-30)*, Research Triangle Park, North Carolina, December 1997.

[7] H.M. Gerndt and A. Krumme, “A Rule-Based Approach for Automatic Bottleneck Detection in Programs on Shared Virtual Memory Systems”, *2nd Int’l Workshop on High-Level Programming Models and Supportive Environments*, Genève, Switzerland, April 1997.

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

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

[10] B. R. Helm, A. D. Malony, and S. F. Fickas, “Capturing and Automating Performance Diagnosis: the Poirot Approach”, *Proceedings of the 1995 International Parallel Processing Symposium*, 606–613, April 1995.

[11] J.K. Hollingsworth, B.P. Miller, and J. Cargille, “Dynamic Program Instrumentation for Scalable Performance Tools”, *Scalable High Performance Computing Conf.,* Knoxville, Tennessee, May 1994.

[12] Intel Corporation, “Intel ® Pentium ® 4 Processor Optimization Reference Manual”, Order Number 248966, 2001.

[13] K.L. Karavanic and B.P. Miller, “Improving Online Performance Diagnosis by the Use of Historical Performance Data”, *SC’99*, Portland, Oregon, November 1999.

[14] 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, pp. 37-46, November 1995.

[15] N. Mukerjee, G.D. Riley, and J.R. Gurd, “FINESSE: A Prototype Feedback-Guided Performance Enhancement System”, *8th Euromicro Workshop on Parallel and Distributed Processing*, Rhodos, Greece, January 2000.

[16] R. S. Pindyck, D. L. Rubinfeld, **Microeconomics**, Prentice Hall, 2000.

[17] E. Rich and K. Knight, **Artificial Intelligence**, McGraw-Hill, Inc., 1991.