A Callgraph-Based Search Strategy for Automated Performance Diagnosis[1]

Harold W. Cain Barton P. Miller Brian J.N. Wylie
{cain,bart,wylie}@cs.wisc.edu
http://www.cs.wisc.edu/paradyn

Computer Sciences Department
University of Wisconsin
Madison, WI 53706-1685, U.S.A.

Abstract. We introduce a new technique for automated performance diagnosis, using the program's callgraph. We discuss our implementation of this diagnosis technique in the Paradyn Performance Consultant. Our implementation includes the new search strategy and new dynamic instrumentation to resolve pointer-based dynamic call sites at run-time. We compare the effectiveness of our new technique to the previous version of the Performance Consultant for several sequential and parallel applications. Our results show that the new search method performs its search while inserting dramatically less instrumentation into the application, resulting in reduced application perturbation and consequently a higher degree of diagnosis accuracy.

1 Introduction

Automating any part of the performance tuning cycle is a valuable activity, especially where intrinsically complex and non-deterministic distributed programs are concerned. Our previous research has developed techniques to automate the location of performance bottlenecks [4,9], and other tools can even make suggestions as to how to fix the program to improve its performance [3,8,10]. The Performance Consultant (PC) in the Paradyn Parallel Performance Tools has been used for several years to help automate the location of bottlenecks. The basic interface is a one-button approach to performance instrumentation and diagnosis. Novice programmers immediately get useful results that help them identify performance-critical activities in their program. Watching the Performance Consultant in operation also acts as a simple tutorial in strategies for locating bottlenecks. Expert programmers use the Performance Consultant as a head start in diagnosis. While it may not find some of the more obscure problems, it saves the programmer time in locating the many common ones.

An important attribute of the Performance Consultant is that it uses dynamic instrumentation [5,11] to only instrument the part of the program in which it is currently in-

[1] This work is supported in part by Department of Energy Grant DE-FG02-93ER25176, Lawrence Livermore National Lab grant B504964, 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.

Appears in Euro-Par 2000, Munich, Germany, August 2000

terested. When instrumentation is no longer needed, it is removed. Insertion and removal of instrumentation occur while the program is running unmodified executables. Instrumentation can include simple counts (such as function calls or bytes of I/O or communication), process and elapsed times, and blocking times (I/O and synchronization).

While the Performance Consultant has shown itself to be useful in practice, there are several limitations that can reduce its effectiveness when operating on complex application programs (with many functions and modules). These limitations manifest themselves when the PC is trying to isolate a bottleneck to a particular function in an application's code. The original PC organized code into a static hierarchy of modules and functions within modules. An automated search based on such a tree is a poor way to direct a search for bottlenecks, for several reasons: (1) when there is a large number of modules, it is difficult to know which ones to examine first, (2) instrumenting modules is expensive, and (3) once a bottleneck is isolated to a module, if there is a large number of functions within a module, it is difficult to know which ones to examine first.

In this paper, we describe how to avoid these limitations by basing the search on the application's callgraph. The contributions of this paper include an automated performance diagnostic strategy based on the application's callgraph, new instrumentation techniques to discover callgraph edges in the presence of function pointers, and a demonstration of the effectiveness of these new techniques. Along with the callgraph-based search, we are able to use a less expensive form of timing primitive, reducing run-time overhead.

The original PC was designed to automate the steps that an experienced programmer would naturally perform when trying to locate performance-critical parts of an application program. Our callgraph enhancements to the PC further this theme. The general idea is that isolating a problem to a part of the code starts with consideration of the main program function and if it is found to be critical, consideration passes to each of the functions it calls directly; for any of those found critical, investigation continues with the functions that they in turn call. Along with the consideration of called functions, the caller must also be further assessed to determine whether or not it is a bottleneck in isolation. This repeats down each exigent branch of the callgraph until all of the critical functions are found.

The callgraph-directed Performance Consultant is now the default for the Paradyn Parallel Performance tools (as of release 3.0). Our experience with this new PC has been uniformly positive; it is both faster and generates significantly less instrumentation overhead. As a result, applications that previously were not suitable for automated diagnosis can now be effectively diagnosed.

The next section describes Paradyn's Performance Consultant in its original form, and later Section 4 describes the new search strategy based on the application program's callgraph. The callgraph-based search needs to be able to instrument and resolve function pointers, and this mechanism is presented in Section 3. We have compared the effectiveness of the new callgraph-based PC to the original version on several serial and parallel applications, and the experiments and results are described in Section 5.

2 Some Paradyn Basics

Paradyn [9] is an application profiler that uses *dynamic instrumentation* [5,6,11] to insert and delete measurement instrumentation as a program runs. Run-time selection of instrumentation results in a relatively small amount of data compared to static (compile or link time) selection. In this section, we review some basics of Paradyn instrumentation, then discuss the Performance Consultant and its original limitations when trying to isolate a bottleneck to particular parts of a program's code.

2.1 Exclusive vs. Inclusive Timing Metrics

Paradyn supports two types of timing metrics, *exclusive* and *inclusive*. Exclusive metrics measure the performance of functions in isolation. For example, exclusive CPU time for function `foo` is only the time spent in that function itself, excluding its callees. Inclusive metrics measure the behavior of a function while it is active on the stack. For example, inclusive time for a function `foo` is the time spent in `foo`, including its callees. Timing metrics can measure process or elapsed (wall) time, and can be based on CPU time or I/O, synchronization, or memory blocking time.

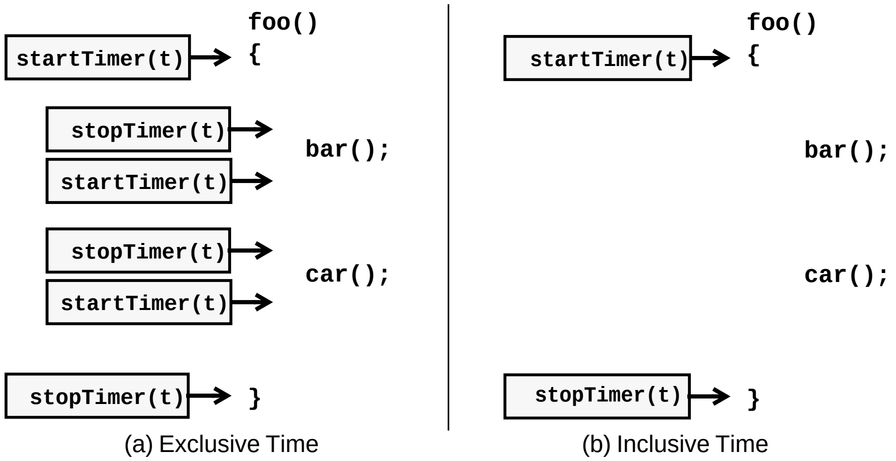


Figure 1 Timing instrumentation for function `foo`.

Paradyn inserts instrumentation into the application to make these measurements. For exclusive time (see Figure 1a), instrumentation is inserted to start the timer at the function's entry and stop it at the exit(s). To include only the time spent in this function, we also stop the timer before each function call site and restart it after returning from the call. Instrumentation for inclusive time is simpler; we only need to start and stop the timer at function entry and exit (see Figure 1b). This simpler instrumentation also generates less run-time overhead. A start/stop pair of timer calls takes 56.5 µs on a SGI Origin. The savings become more significant in functions that contain many call sites.

2.2 The Performance Consultant

Paradyn's Performance Consultant (PC) [4,7] dynamically instruments a program with timer start and stop primitives to automate bottleneck detection during 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. Resources include the program code (e.g., modules and functions), machine nodes, application processes and threads, synchronization objects, 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 hierarchy 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 2 is < /Code/testutil.C/verifyA>.

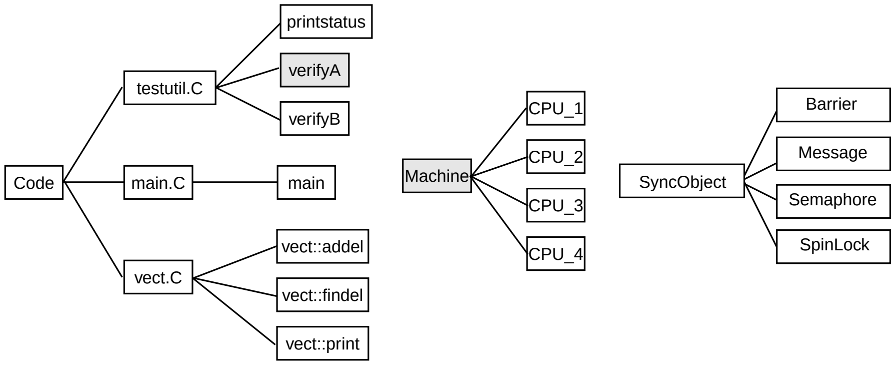


Figure 2 Three Sample Resource Hierarchies: Code, Machine, and SyncObject.

For a particular performance measurement, we may wish to isolate the measurement to specific parts of a program. For example, we may be interested in measuring I/O blocking time as the total for one entire execution, or as the total for a single function. A *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 immediate descendents of the selected node. For example, the shaded nodes in Figure 2 represent the constraint: code function `verifyA` running on any CPU in the machine, which is labeled with the focus: < /Code/testutil.C/verifyA, /Machine >.

Each node in a PC search represents instrumentation and data collection for a (hypothesis : focus) pair. If a node tests true, meaning a bottleneck has been found, the

Performance Consultant tries to determine more specific information about the bottleneck. It considers two types of search 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, and only these foci are used as roots for refinement to more specific hypothesis constructions (to avoid undesirable exponential search expansion).

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. An example Paradyn SHG display is shown in Figure 3.

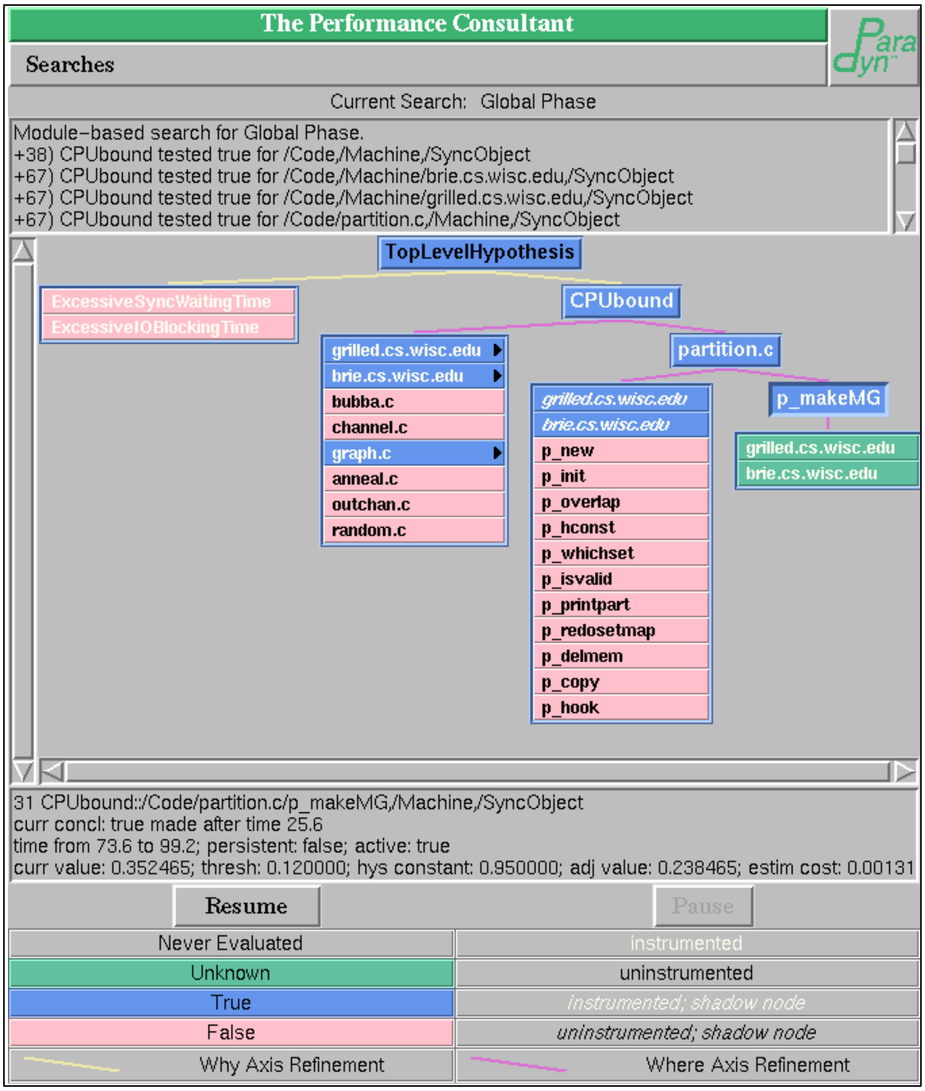
2.3 Original Paradyn: Searching the Code Hierarchy

The search strategy originally used by the Performance Consultant was based on the module/function structure of the application. When the PC wanted to refine a bottleneck to a particular part of the application code, it first tried to isolate the bottleneck to particular modules (`.o/.so/.obj/.dll` file). If the metric value for the module is above the threshold, then the PC tries to isolate the bottleneck to particular functions within the module.

This strategy has several drawbacks for large programs.

1. Programs often have many modules; and modules often have hundreds of functions. When the PC starts to instrument modules, it cannot instrument all of them efficiently at the same time and has no information on how to choose which ones to instrument first; the order of instrumentation essentially becomes random. As a result, many functions are needlessly instrumented. Many of the functions in each module may never be called, and therefore do not need to be instrumented. By using the callgraph, the new PC operates well for any size of program.
2. To isolate a bottleneck to a particular module or function, the PC uses exclusive metrics. As mentioned in Section 2.1, these metrics require extra instrumentation code at each call site in the instrumented functions. The new PC is able to use the cheaper inclusive metrics.
3. The original PC search strategy was based on the notion that coarse-grain instrumentation was less expensive than fine-grained instrumentation. For code hierarchy searches, this means that instrumentation to determine the total time in a module should be cheaper than determining the time in each individual function. Unfortunately, the cost of instrumenting a module is the same as instrumenting all the functions in that module. The only difference is that we have one timer for the entire module instead of one for each function.

   This effect could be reduced for module instrumentation by not stopping and starting timers at call sites that call functions inside the same module. While this technique is possible, it provides such a small benefit, it was not worth the complexity. Use of the callgraph in the new PC avoids using the code hierarchy.



```
Module–based search for Global Phase.
+38) CPUbound tested true for /Code,/Machine,/SyncObject
+67) CPUbound tested true for /Code,/Machine/brie.cs.wisc.edu,/SyncObject
+67) CPUbound tested true for /Code,/Machine/grilled.cs.wisc.edu,/SyncObject
+67) CPUbound tested true for /Code/partition.c,/Machine,/SyncObject
```

```
31 CPUbound::/Code/partition.c/p_makeMG,/Machine,/SyncObject
curr concl: true made after time 25.6
time from 73.6 to 99.2; persistent: false; active: true
curr value: 0.352465; thresh: 0.120000; hys constant: 0.950000; adj value: 0.238465; estim cost: 0.00131
```

Figure 3 The Original Performance Consultant with a Bottleneck Search in Progress. The three items immediately below **TopLevelHypothesis** have been added as a result of refining the hypothesis. **ExcessiveSyncWaitingTime** and **ExcessiveIOBlockingTime** have tested false, as indicated by node color (pink or light grey), and **CPUbound** (blue or dark grey) has tested true and been expanded by refinement. Code hierarchy module nodes **bubba.c**, **channel.c**, **anneal.c**, **outchan.c**, and **random.c** all tested false, whereas modules **graph.c** and **partition.c** and Machine nodes **grilled** and **brie** tested true and were refined. Only function **p_makeMG** in module **partition.c** was subsequently found to have surpassed the bottleneck hypothesis threshold, and the final stage of the search is considering whether this function is exigent on each Machine node individually. (Already evaluated nodes with names rendered in black no longer contain active instrumentation, while instrumented white-text nodes continue to be evaluated.)

3 Dynamic Function Call Instrumentation

Our search strategy is dependent on the completeness of the callgraph used to direct the search. If all caller-callee relationships are not included in this graph, then our search strategy will suffer from blind spots where this information is missing. Paradyn's standard start-up operation includes parsing the executable file in memory (dynamically linked libraries are parsed as they are loaded). Parsing the executable requires identifying the entry and exits of each function (which is trickier than it would appear [6]) and the location of each function call site. We classify call sites as *static* or *dynamic*. Static sites are those whose destination we can determine from inspection of the code. Dynamic call sites are those whose destination is calculated at run-time. While most call sites are static, there is still a non-trivial number of dynamic sites. Common sources of dynamic call sites are function pointers and C++ virtual functions.

Our new instrumentation resolves the address of the callee at dynamic call sites by inserting instrumentation at these sites. This instrumentation computes the appropriate callee address from the register contents and the offsets specified in the call instruction. New call destination addresses are reported to the Paradyn front-end, which then updates its callgraph and notifies the PC. When the PC learns of a new callee, it incorporates the callee in its search. We first discuss the instrumentation of the call site, then discuss how the information gathered from the call site is used.

3.1 Call Site Instrumentation Code

The Paradyn daemon includes a code generator to dynamically generate machine-specific instrumentation code. As illustrated in Figure 4, instrumentation code is inserted

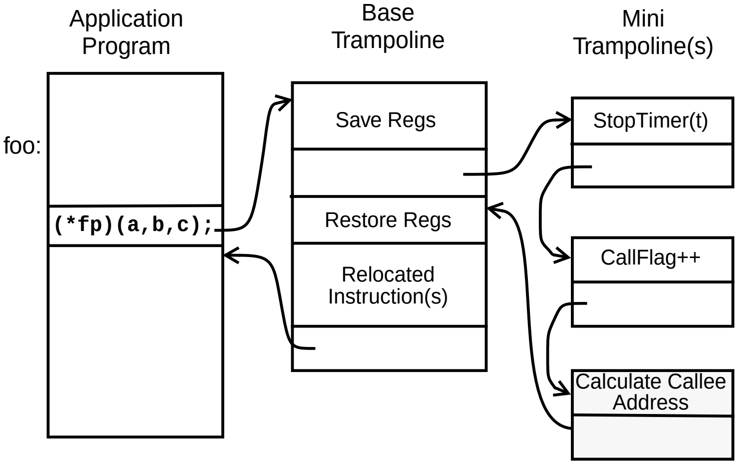


Figure 4 Simplified Control Flow from Application to Instrumentation Code. A dynamic call instruction in function `foo` is replaced with branch instructions to the base trampoline. The base trampoline saves the application's registers and branches to a series of mini trampolines that each contain different instrumentation primitives. The final mini trampoline returns to the base trampoline, which restores the application's registers, emulates the relocated dynamic call instruction, and returns control to the application.

into the application by replacing an instruction with a branch to a code snippet called the *base-trampoline*. The base-trampoline saves and restores the application's state before and after executing instrumentation code. The instrumentation code for a specific primitive (e.g. a timing primitive) is contained in a *mini-trampoline*.

Dynamic call instructions are characterized by the destination address residing in a register or (sometimes on the x86) a memory location. The dynamic call resolution instrumentation code duplicates the address calculation of these call instructions. This code usually reads the contents of a register. This reading is slightly complicated, since (because we are instrumenting a call instruction) there are two levels of registers saved: the caller-saved registers as well as those saved by the base trampoline. The original contents of the register have been stored on the stack and may have been overwritten. To access these saved registers, we added a new code generator primitive (abstract syntax tree operand type). We have currently implemented dynamic call site determination for the MIPS, SPARC, and x86, with Power2 and Alpha forthcoming.

A few examples of the address calculations are shown in Table 1. We show an example of the type of instruction that would be used at a dynamic call site, and mini-trampoline code that would retrieve the saved register or memory value and calculate the callee's address.

Table 1: Dynamic callee address calculation.

Instruction Set	**Call Instruction**	**Mini-Trampoline Address Calculation**	**Explanation**
MIPS	`jalr $t9`	`ld $t0, 48($sp)`	Load `$t9` from stack.

x86	`call [%edi]`	`mov %eax,-160(%ebp)`
		`mov %ecx,[%eax]`
			Load `%edi` from stack. Load function address from memory location pointed to by `%eax.`

SPARC	`jmpl %l0,%o7`	`ld [%fp+20],%l0`
		`add %l0,%i7,%l3`
			Load `%l0` from stack. `%o7` becomes `%i7` in new register window.

3.2 Control Flow for Dynamic Call Site Instrumentation

To instrument a dynamic call site, the application is paused, instrumentation inserted, and the application resumed. The call site instrumentation is inserted on demand, i.e., only when the caller function becomes relevant to the bottleneck search. For example, if function `foo` contains a dynamic call site, this site is not instrumented until the PC identifies `foo` as a bottleneck, at which point we need to know all of the callees of `foo`. By instrumenting these sites on demand, we minimize the amount of instrumentation in the application. The flow of control for these steps is shown as steps 1 and 2 in Figure 5.

The call site instrumentation detects the first time that a callee is called from that site. When a callee is first called from a site, the instrumentation code notifies the Paradyn daemon (step A in Figure 5), which notifies the PC in the Paradyn front-end (step

B). The new caller-callee edge is added to the callgraph and, if desired, instrumentation can be inserted in the newly discovered callee (steps C and D).

We do not want to incur this communication and instrumentation cost each time that a dynamic call site is executed. Fortunately, most call sites only call a few different functions, so we keep a small table of callee addresses for each dynamic call site. Each time that a dynamic call site is executed and a callee determined, we check this table to see if it is a previously known caller-callee pair. If the pair has been previously seen, we bypass the steps A-D.

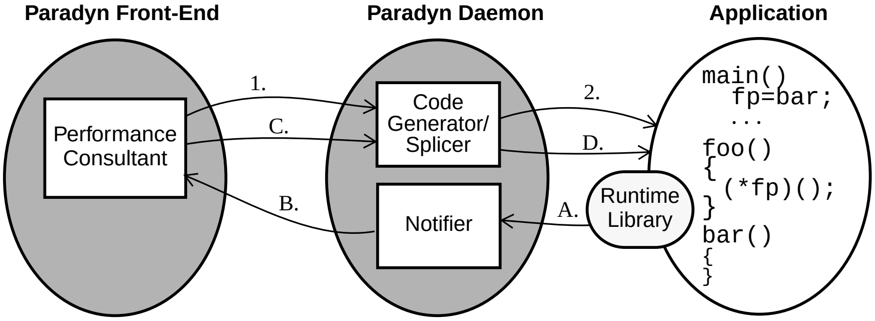


Figure 5 Control Flow between Performance Consultant and Application. (1) PC issues dynamic call-site instrumentation request for function `foo`. (2) Daemon instruments dynamic call sites in `foo`. (A) Application executes call instruction and when a new callee is found, runtime library notifies daemon. (B) Daemon notifies PC of new callee `bar` for function `foo`. (C) PC requests inclusive timing metric for function `bar`. (D) Daemon inserts timing instrumentation for `bar`.

Once the Paradyn daemon has notified the Performance Consultant of a new dynamic caller-callee relationship, the PC can take advantage of this information. If the dynamic caller has been previously determined a bottleneck, then the callee must be instrumented to determine if it is also a bottleneck. A possible optimization to this sequence is for the Paradyn daemon to instrument the dynamic callee as soon as it is discovered, thus reducing the added delay of conveying this information to the Paradyn front-end and waiting for the front-end to issue an instrumentation request for the callee. However, this optimization would require knowledge by the Paradyn daemon of the type of instrumentation timing primitive desired for this callee, and would also limit the generality of our technique for dynamic callee determination.

4 Callgraph-based Searching

We have modified the Performance Consultant's code hierarchy search strategy to direct its search using the application's callgraph. The remainder of the search hierarchies (such as Machine and SyncObject) are still searched using the structure of their hierarchy; the new technique is only used when isolating the search to a part of the Code hierarchy.

When the PC starts refining a potential bottleneck to a specific part of the code, it starts at the top of the program graph, at the program entry function for each distinct executable involved in the computation. These functions are instrumented to collect an inclusive metric. For example, if the current candidate bottleneck were CPUbound, the function would be instrumented with the CPU time inclusive metric. The timer associated with this metric runs whenever the program is running and this function was on the stack (presumably, the entire time that the application is running in this case).

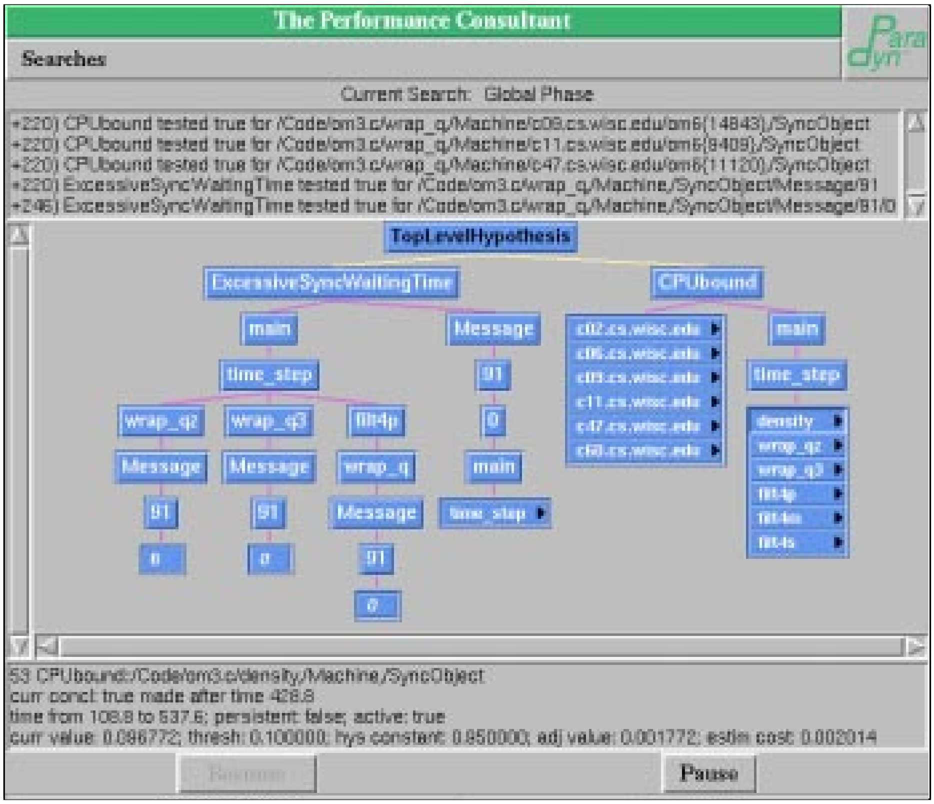
If the metric value for the main program function is above the threshold, the PC uses the callgraph to identify all the functions called from it, and each is similarly instrumented with the same inclusive metric. In the callgraph-based search, if a function's sustained metric value is found to be below the threshold, we stop the search for that branch of the callgraph (i.e., we do not expand the search to include the function's children). If the function's sustained metric value is above the threshold, the search continues by instrumenting the functions that it calls. The search in the callgraph continues in this manner until all possible branches have been exhausted either because the accumulated metric value was too small or we reached the leaves of the callgraph.

Activating instrumentation for functions currently executing, and therefore found on the callstack, requires careful handling to ensure that the program and instrumentation (e.g., active timers and flags) remain in a consistent state. Special retroactive instrumentation needs to be immediately executed to set the instrumentation context for any partially-executed and now instrumented function, prior to continuing program execution. Timers are started immediately for already executing functions, and consequently produce measurements earlier than waiting for the function to exit (and be removed from the callstack) before instrumenting it normally.

The callgraph forms a natural organizational structure for three reasons. First, a search strategy based on the callgraph better represents the process that an experienced programmer might use to find bottlenecks in a program. The callgraph describes the overall control flow of the program, following a path that is intuitive to the programmer. We do not instrument a function unless it is a reasonable candidate to be a bottleneck: its calling functions are currently be considered a bottleneck. Second, using the callgraph scales well to large programs. At each step of the search, we are addressing individual functions (and function sizes are typically not proportional to the overall code size). The total number of modules and functions do not effect the strategy. Third, the callgraph-based search naturally uses inclusive time metrics, which are (often significantly) less costly in a dynamic instrumentation system than their exclusive time counterparts.

An example of the callgraph-based Paradyn SHG display at the end of a comprehensive bottleneck search is shown in Figure 6.

While there are many advantages to using this callgraph-based search method, it has a few disadvantages. One drawback is that this search method has the potential to miss a bottleneck when a single resource-intensive function is called by numerous parent functions, yet none of its parents meet the threshold to be considered a bottleneck. For example, an application may spend 80% of its time executing a single function, but if that function has many parents, none of which are above the bottleneck threshold, our search strategy will fail to find the bottleneck function. To handle this situation, it is worth considering that the exigent functions are more than likely to be found on the



```
+220) CPUbound tested true for /Code/om3.c/wrap_q/Machine/c09.cs.wisc.edu/om6{14843}/SyncObject
+220) CPUbound tested true for /Code/om3.c/wrap_q/Machine/c11.cs.wisc.edu/om6{8409}/SyncObject
+220) CPUbound tested true for /Code/om3.c/wrap_q/Machine/c47.cs.wisc.edu/om6{11120}/SyncObject
+220) ExcessiveSyncWaitingTime tested true for /Code/om3.c/wrap_q/Machine/SyncObject/Message/91
+246) ExcessiveSyncWaitingTime tested true for /Code/om3.c/wrap_q/Machine/SyncObject/Message/91/0
```

```
53 CPUbound:/Code/om3.c/density/Machine/SyncObject
curr concl: true made after time 428.8
time from 108.8 to 537.6; persistent: false; active: true
curr value: 0.098772; thresh: 0.100000; hys constant: 0.850000; adj value: 0.001772; estim cost: 0.002014
```

Figure 6 The Callgraph-based Performance Consultant after Search Completion. This snapshot shows the Performance Consultant upon completion of a search with the OM3 application when run on 6 Pentium II Xeon nodes of a Linux cluster. For clarity, all hypothesis nodes which tested false are hidden, leaving only paths which led to the discovery of a bottleneck. This view of the search graph illustrates the path that the Performance Consultant followed through the callgraph to locate the bottleneck functions. Six functions, all called from the **time_step** routine, have been found to be above the specified threshold to be considered CPU bottlenecks, both in aggregation and on each of the 6 cluster nodes. The **wrap_q**, **wrap_qz** and **wrap_q3** functions have also been determined to be synchronization bottlenecks when using MPI communicator **91** and message tag **0**.

stack whenever Paradyn is activating or modifying instrumentation (or if it were to periodically ‘sample’ the state of the callstack). A record kept of these ‘‘callstack samples’’ therefore forms an appropriate basis of candidate functions for explicit consideration, if not previously encountered, during or on completion of the callgraph-based search.

5 Experimental Results

We performed several experiments to evaluate the effectiveness of our new search method relative to the original version of the Performance Consultant. We use three criteria for our evaluation: the accuracy, speed, and efficiency with which the PC performs its search. The accuracy of the search is determined by comparing those bottlenecks reported by the Performance Consultant to the set of bottlenecks considered true application bottlenecks. The speed of a search is measured by the amount of time required for each PC to perform its search. The efficiency of a search is measured by the amount of instrumentation used to conduct a bottleneck search; we favor a search strategy that inserts less instrumentation into the application. We describe our experimental set-up and then present results from our experiments.

5.1 Experimental Setup

We used three sequential applications, a multithreaded application and two parallel application for these experiments. The sequential applications include the SPEC95 benchmarks fpppp (a Fortran application that performs multi-electron derivatives) and go (a C program that plays the game of Go against itself), as well as Draco (a Fortran hydrodynamic simulation of inertial confinement fusion, written by the laser fusion groups at the University of Rochester and University of Wisconsin). The sequential applications were run on a dual-processor SGI Origin under IRIX 6.5. The matrix application is based on the Solaris threads package and was run on an UltraSPARC Solaris 2.6 uniprocessor. The parallel application ssTwod solves the 2-D Poisson problem using MPI on four nodes of an IBM SP/2 (this is the same application used in a previous PC study[7]). The parallel application OM3 is a free-surface, *z*-coordinate general circulation ocean model, written using MPI by members of the Space Science and Engineering Center at the University of Wisconsin. OM3 was run on eight nodes of a 24-node SGI Origin under IRIX 6.5. Some characteristics of these applications that affect the Performance Consultant search space are detailed in Table 2. (All system libraries are explicitly excluded from this accounting and the subsequent searches.)

Table 2: Application search space characteristics.

Application (Language)	Lines of code	Number of modules	Number of functions	Number of dynamic call sites
Draco (F90)	61,788	232	256	5
go (C)	26,139	18	376	1
fpppp (F77)	2,784	39	39	0
matrix (C/Sthreads)	194	1	5	0
ssTwod (F77/MPI)	767	7	9	0
OM3 (C/MPI)	2,673	1	28	3

We ran each application program under two conditions: first, with the original PC, and then with the new callgraph-based PC. In each case, we timed the run from the start of the search until the PC had no more alternatives to evaluate. For each run, we saved the complete history of the performance search (using the Paradyn export facility) and recorded the time at which the PC found each bottleneck.

A 5% threshold was used for CPU bottlenecks and 12% threshold for synchronization waiting time bottlenecks. For the sequential applications, we verified the set of application bottlenecks using the prof profiling tool. For the parallel applications, we used Paradyn manual profiling along with both versions of the Performance Consultant to determine their bottlenecks.

5.2 Results

We ran both the original and modified versions of the Performance Consultant for each of the applications, measuring the time required to locate all of the bottlenecks. Each experiment is a single execution and search. For OM3, the SGI Origin was not dedicated to our use, but also not heavily loaded during the experiments. In some cases, the original version of the PC was unable to locate all of an application's bottlenecks due to the perturbation caused by the larger amount of instrumentation it requires. Table 3 shows the number of bottlenecks found by each version of the PC, and the time required to complete each search. As we can see, the size of an application has a significant impact on the performance of the original PC. For the small fpppp benchmark and matrix application, the original version of the PC locates the application's bottlenecks a little faster than the callgraph-based PC. This is because they have few functions and no complex bottlenecks (being completely CPUbound programs, there are few types and combinations of bottlenecks). As a result, the original Performance Consultant can quickly instrument the entire application. The new Performance Consultant, however, always has to traverse some portion of the application's callgraph.

Table 3: Accuracy, overhead and speed of each search method.

	Bottlenecks found in complete search		Instrumentation mini-tramps used		Required search time (seconds)	
Application	**Original**	**Callgraph**	**Original**	**Callgraph**	**Original**	**Callgraph**
Draco	3	5	14,317	228	1,006	322
go	2	4	12,570	284	755	278
fpppp	3	3	474	96	141	186
matrix	4	5	439	43	200	226
ssTwod	9	9	43,230	11,496	461	326
OM3	13	16	184,382	60,670	2,515	957

For the larger applications, the new search strategy's advantages are apparent. The callgraph-based Performance Consultant performs its search significantly faster for each program other than fpppp and matrix. For Draco, go, and OM3, the original Performance Consultant's search not only requires more time, but due to the additional perturbation that it causes, it is unable to resolve some of the bottlenecks. It identifies only three of Draco's five bottleneck functions, two of go's four bottlenecks, and 13 of OM3's 16.

We also measured the efficiency with which each version of the Performance Consultant performs its search. An efficient performance tool will perform its search while inserting a minimum amount of instrumentation into the application. Table 3 also shows the number of mini-trampolines used by the two search methods, each of which corresponds to the insertion of a single instrumentation primitive. The new version of the Performance Consultant can be seen to provide a dramatic improvement in terms of efficiency. The number of mini-trampolines used by the previous version of the PC is more than an order of magnitude larger than used by the new PC for both go and Draco, and also significantly larger for the other applications studied. This improvement in efficiency results in less perturbation of the application and therefore a greater degree of accuracy in performance diagnosis.

Although the callgraph-based performance consultant identifies a greater number of bottlenecks than the original version of the of the performance consultant, it suffers one drawback that stems from the use of inclusive metrics. Inclusive timing metrics collect data specific to one function and all of its callees. Because the performance data collected is not restricted to a single function, it is difficult to evaluate a particular function in isolation and determine its exigency. For example, only 13% of those functions determined bottlenecks by the callgraph-based performance consultant are truly bottlenecks. The remainder are functions which have been classified bottlenecks en route to the discovery of true application bottlenecks. One solution to this inclusive bottleneck ambiguity is to re-evaluate all inclusive bottlenecks using exclusive metrics. Work is currently underway within the Paradyn group to implement this inclusive bottleneck verification.

6 Conclusions

We found the new callgraph-based bottleneck search in Paradyn's Performance Consultant, combined with dynamic call site instrumentation, to be much more efficient in identifying bottlenecks than its predecessor. It works faster and with less instrumentation, resulting in lower perturbation of the application and consequently greater accuracy in performance diagnosis.

Along with its advantages, the callgraph-based search has some disadvantages that remain to be addressed. Foremost among them are blind spots, where exigent functions are masked in the callgraph by their multiple parent functions, none of which themselves meet the threshold criteria to be found a bottleneck. This circumstance appears to be sufficiently rare that we have not encountered any instances yet in practice. It is also necessary to consider when and how it is most appropriate for function exigency consideration to progress from using the weak inclusive criteria to the strong exclusive criteria that determine true bottlenecks. The exclusive 'refinement' of a function found exigent using inclusive criteria can be considered as a follow-on equivalent to refinement to its children, or as a reconsideration of its own exigency using the stronger criteria. Additionally, it remains to be determined how the implicit equivalence of the main program routine and 'Code' (the root of the Code hierarchy) as resource specifiers can be exploited for the most efficient searches and insightful presentation.

Acknowledgements. Matthew Cheyney implemented the initial version of the static callgraph structure. This paper benefited from the hard work of the members of the Paradyn research group. While everyone in the group influenced and helped with the results in this paper, we would like to specially thank Andrew Bernat for his support of the AIX/SP2 measurements, and Chris Chambreau for his support of the IRIX/MIPS measurements. We are grateful to the Laboratory for Laser Energetics at the University of Rochester for the use of their SGI Origin system for some of our experiments, and the various authors of the codes made available to use. Ariel Tamches provided constructive comments on the early drafts of the paper.

References

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

[2] A. Beguelin, J. Dongarra, A. Geist, and V.S. Sunderam, "Visualization and Debugging in a Heterogeneous Environment", *IEEE Computer* **26**, 6, June 1993.

[3] 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.

[4] J.K. Hollingsworth and B.P. Miller, "Dynamic Control of Performance Monitoring on Large Scale Parallel Systems", 7th *Int'l Conf. on Supercomputing*, Tokyo, Japan, July 1993.

[5] 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.

[6] J.K Hollingsworth, B.P. Miller, M.J.R. Gonçalves, O. Naìm, Z. Xu, and L. Zheng, "MDL: A Language and Compiler for Dynamic Program Instrumentation", *6th Int'l Conf. on Parallel Architectures and Compilation Techniques*, San Francisco, California, Nov. 1997

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

[8] J. Kohn and W. Williams, "ATExpert", *Journal of Parallel and Distributed Computing* **18**, 205–222, June 1993.

[9] 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.

[10] N. Mukhopadhyay (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.

[11] Z. Xu, B.P. Miller, and O. Naìm, "Dynamic Instrumentation of Threaded Applications", *7th ACM Symp. on Principles and Practice of Parallel Programming*, Atlanta, Georgia, May 1999.