A Loop-aware Search Strategy for Automated Performance Analysis

Eli D. Collins and Barton P. Miller

Computer Sciences Department
University of Wisconsin
Madison, WI 53706-1685, USA
{eli,bart}@cs.wisc.edu

Abstract. Automated online search is a powerful technique for performance diagnosis. Such a search can change the types of experiments it performs while the program is running, making decisions based on live performance data. Previous research has addressed search speed and scaling searches to large codes and many nodes. This paper explores using a finer granularity for the bottlenecks that we locate in an automated online search, i.e., refining the search to bottlenecks localized to loops. The ability to insert and remove instrumentation on-the-fly means an online search can utilize fine-grain program structure in ways that are infeasible using other performance diagnosis techniques. We automatically detect loops in a program's binary control flow graph and use this information to efficiently instrument loops. We implemented our new strategy in an existing automated online performance tool, Paradyn. Results for several sequential and parallel applications show that a loop-aware search strategy can increase bottleneck precision without compromising search time or cost.

1 Introduction

Performance analysis tools aid in the understanding of application behavior. Automating the search for performance problems enables non-experts to use these tools and provides a fast diagnostic for experienced performance analysts [9, 4, 17, 7]. A performance tool that uses dynamic instrumentation [10] can search for and identify performance problems in an unmodified program while the program runs. Our previous research in online automated search has addressed scaling with large codes, lowering instrumentation cost, and locating bottlenecks quickly [2, 19, 11].

This paper describes an online automated search strategy that increases the precision (granularity at which bottlenecks are identified) of an automated performance search. To demonstrate the efficacy of our strategy, we have implemented it in an existing automated performance tool, the Paradyn Parallel Performance tool [15].

An online search strategy manages its search space by focusing on functions that are currently being executed or functions that are about to be executed. To this end the Paradyn performance tool's automated search component, the Performance Consultant, uses a program's callgraph to locate bottlenecks [2].

This topdown search of program behavior matches the process an experienced programmer might use. This helps limit the amount of instrumentation inserted into a program, which improves search scalability (the ability to operate efficiently on programs with large code sizes). Searching a program's call graph identifies performance problems at function granularity.

While identifying bottlenecks at function granularity is useful in practice, the user of a performance tool would often like to know where inside the function the bottleneck is located. A large function may contain multiple distinct bottlenecks. A small function may be a bottleneck because it is called repeatedly in a loop. To better localize a bottleneck, an automated search strategy must search inside a function. This requires introducing a new level in the callgraph that improves search precision. This level must not inhibit search scalability; just adding new levels to the callgraph increases the size of the search space. The level should represent a program structure that logically decomposes a function for the user, more precisely locates bottlenecks, and partitions functions for searching. Augmenting a program's callgraph with nested loops meets these requirements. When searching code, after functions, loops are the the next natural program decomposition. Loops increase precision in several ways: (1) they may be bottlenecks themselves, especially in long running programs and scientific applications, (2) they help identify which callsites in a function are bottlenecks, and (3) they logically decompose a function for a user.

Applications often contain loops that execute for many iterations, and loops are natural sources of parallelism that both compilers and hardware exploit. For example, the OpenMP `parallel do` directive [22] allows a compiler to automatically parallelize a loop. Loop-level performance data provides valuable feedback for these optimizations.

Even if a loop is not a bottleneck, it can help to more precisely locate a bottleneck. Suppose function A contains multiple calls to bottleneck function B. If the calls to B occur at different levels in A's loop hierarchy then we might infer which calls to B within A are responsible for the bottleneck from the performance data collected for A's loops. Figure 1 depicts the callgraph for these functions with and without loop information. Without loops, we can determine if A contains a bottleneck and if that bottleneck is B. With loops, we can determine if the bottleneck in A is due to a particular loop, or a particular call to B.

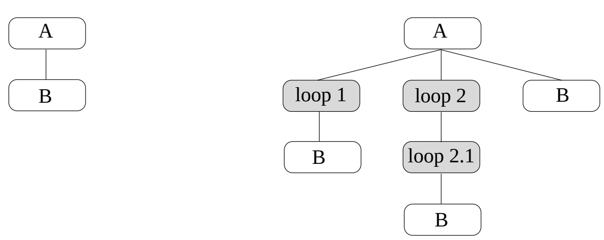


Fig. 1. A callgraph, and the same callgraph augmented with nested loops.

Loops naturally decompose functions for a user. A search that specifies bottlenecks at loop precision is appropriate because the user can easily examine and change the loop structure of a program. Loops also naturally decompose functions for searching. A loop may contain multiple nested loops and callsites. If a search uses an inclusive performance metric and determines that a loop is not a bottleneck then it does not have to instrument the loop's nested loops and callsites. Improving bottleneck precision can lower instrumentation cost, as we show in our experiments.

The contributions of this paper are (1) an automated search strategy that increases precision, (2) the definition of points in a function's control flow graph that correspond to loop execution, and (3) new dynamic instrumentation techniques that enable the efficient instrumentation of these points.

2 Related Work

Traditional profilers report performance data at function or statement granularity using sampling. The prof and gprof [6] profilers record flat function and callgraph profiles, respectively. Intel's VTune$^{\mathrm{TM}}$Performance Analyzer [21] provides callgraph, statement, and instruction level profiles. Sampling enables data collection at a fine granularity, but only provides CPU time. For example, it is difficult to collect inclusive CPU time, elapsed time, and synchronization or I/O blocking time using sampling [19].

Some tools automate the search through a problem space similar to Paradyn's Performance Consultant. For example, Helm *et al* [7] use heuristic classification to control an automated search for performance problems. Finesse [17] diagnoses performance problems using search refinement across multiple runs.

Several performance tools report performance data at loop granularity. SvPablo [18] allows users to instrument source code and browse runtime performance data. SvPablo correlates performance data with program source code at the level of statements, loops, and functions. Unlike Paradyn, which instruments a program's binary, SvPablo inserts instrumentation into the program source code using a preprocessor that can instrument functions and loops. Instrumenting the program's source reflects the code as the programmer wrote it, but may not fully reflect the code that was generated by the compiler.

MTOOL [5] is a tool for isolating memory performance bottlenecks. It uses a program's basic block count profile to identify frequently executed blocks to instrument. MTOOL instruments basic blocks with explicit timer calls, and aggregates basic block data to report loop, function, and whole program overheads.

HPCView [14] is a toolkit for combining multiple sets of profile data, and correlating this data with program source code. Aggregate performance data and derived metrics are presented at both function and loop levels. HPCView uses binary analysis to correlate performance data from program structures resulting from compiler transformations like loop fusion and distribution. Unlike Paradyn, which performs online automated performance analysis, HPCView is a post-mortem tool that combines the results of several program runs.

The DPOMP tool [3] uses dynamic instrumentation to collect performance data for OpenMP constructs, including `parallel do` loops. The compiler transforms OpenMP directives into function calls, which DPOMP instruments. Due to compiler optimizations, they can not always collect performance data for loop begin and end iteration events. Dynamic binary loop instrumentation enables the collection of these events by identifying loops through control flow rather than function calls inserted by the compiler.

The bursty tracing profiler [8] statically instruments a program to capture temporal profiles. To limit the overhead of their instrumentation they use counter-based sampling to switch between instrumented and un-instrumented copies of the binary. They can eliminate many checks (to switch between copies of the binary) by analyzing the program's callgraph and binary. They further limit overhead by not instrumenting "k-boring" loops (loops with no calls and at most k profiling events of interest). The bursty tracing profiler collects performance data online; performance analysis is handled offline. Our approach performs automated online performance analysis: we insert and remove instrumentation at run time based on the current performance data. We limit the overhead of fine-grain instrumentation by activating it only when necessary and using performance data to decide which parts of the program to instrument.

3 The Performance Consultant

The Performance Consultant is Paradyn's automated bottleneck detection component. It searches for application bottlenecks by performing experiments that measure application performance and try to locate where performance problems occur. Initially, the experiments measure the performance of the entire application. If the Performance Consultant discovers bottlenecks it refines the experiments to be more precise about the type and location of each bottleneck.

The experiments test *hypotheses* about why an application may suffer from performance problems. For example, one hypothesis tested by the Performance Consultant is whether the application is spending an excessive amount of time blocking on I/O operations. To activate an experiment the Performance Consultant inserts instrumentation code into the application to collect performance data. Instances when the measured value exceeds a predefined threshold for the experiment are termed *bottlenecks*.

Paradyn represents programs as collections of discrete resources. Resources include program code (modules, functions, and loops), processes, threads, machines, and synchronization objects. Paradyn organizes these program resources into trees called *resource hierarchies*. The root of each resource hierarchy is labeled with the hierarchy's name. As we move down from the root node, each level of the hierarchy represents a finer-grained description of the program. To form a *resource name* we concatenate the labels along the unique path from the root to the node representing the resource. For example, the resource name for function `fact` in module `math.C` is ⟨Code/math.C/fact⟩.

We may wish to constrain measurements to particular parts of a program. For example, we may want to measure CPU time for the entire execution of the program or for a single function or loop. An experiment's *focus* determines where the instrumentation code is inserted. Selecting a node in the resource hierarchy narrows the view to include only those nodes that descend from the selected node. For example, the focus ⟨/Code/math.C/fact, /Machine/toaster7.cs.wisc.edu, /SyncObject⟩ denotes function `fact` from module `math.C` executing on host `toaster7.cs.wisc.edu`. This focus specifies the top level SyncObject resource so it does not constrain the resources it names to any particular synchronization object or type of synchronization object.

The performance consultant *refines* its search for bottlenecks from a true experiment (an experiment whose hypothesis is true at its focus) by generating more experiments that have a more specific focus. To refine a focus, the Performance Consultant generates new foci. For example, the focus ⟨/Code/math.C/fact/loop_1, /Machine/toaster7.cs.wisc.edu, /SyncObject⟩ is refined to a particular loop in `fact` and the focus ⟨/Code/math.C/fact, /Machine/toaster7.cs.wisc.edu/8791, /SyncObject⟩ is refined to a particular process (with ID 8791) on host `toaster7.cs.wisc.edu`.

The *search history graph* records the cumulative refinements of a search. Each node represents a ⟨hypothesis, focus⟩ pair. Paradyn provides a visual representation of this graph that is dynamically updated as the Performance Consultant refines its search. This display provides both a visual history of the search and information about individual experiments such as its hypothesis, focus, whether it is active, its current measured value, and whether the experiment's hypothesis has yet to test true or false.

The cost of the instrumentation enabled by the Performance Consultant is continually monitored and limited to a user-selected threshold. New experiments generate new instrumentation requests and, when existing hypotheses test false, their instrumentation is removed.

4 Binary Instrumentation of Loops

To collect performance data at loop granularity, Paradyn must be able to instrument individual loops in a program binary. In this section we describe our implementation of loop instrumentation.

When Paradyn parses application binaries, it builds a flow graph for each function. Dominator information is calculated using the Lengauer-Tarjan algorithm [13] and is used to identify *natural loops*. We use standard definitions for basic blocks, dominators, back edges, and natural loops [1]. A *basic block* is a maximal sequence of instructions that can be entered only at the first instruction and exited only from the last instruction. A basic block M *dominates* block N, if every possible execution path in the flow graph from entry to N includes M. A *back edge* in a flow graph is an edge whose target dominates its source. The *natural loop* of a back edge M → N is the subgraph consisting of the set of nodes containing N and all the blocks from which M can be reached without passing through N. Block N is the *loop header*.

The decision to use natural loops (as opposed to any cycle in the flow graph) is reasonable given that irreducible loops are rare–you cannot create them in a structured language without using gotos. Certain compiler code replication techniques may transform reducible loops in the program's source code into irreducible loops in the program's binary, though this case is rare in our experience. The type of loop identified is not an issue of correctness; irreducible loops in a program binary are ignored, and do not hinder the instrumentation of natural loops.

If two natural loops share a header we cannot distinguish their nesting relationship, or if they are derived from a single loop in the source. In this case we combine the two loops into a single natural loop as is common in compilers [16].

To instrument loops in a flow graph, we define four *instrumentation points* in the flow graph that correspond to loop execution semantics:

1. *Loop entry* instrumentation executes when control enters a loop. We instrument the set of edges M → N such that N is the loop header and M is not a member of the loop. If M is the loop's *preheader* then one such edge exists and we may instrument M.
2. *Loop exit* instrumentation executes when control exits a loop. We instrument the set of edges M → N such that M is a member of the loop and N is not.
3. *Loop begin iteration* instrumentation executes at the beginning of each loop iteration. We instrument the loop header.
4. *Loop end iteration* instrumentation executes at the end of each loop iteration. We instrument the loop's back edge and loop exit instrumentation points.

Figure 2 (a) illustrates the location of these points for a simple loop. Loop entry and exit points are balanced. For example, when instrumenting loop entry with a start timer operation and loop exit with a stop timer operation, execution of the start timer will always be eventually followed by the execution of the stop timer. Loop begin and end iteration points are balanced as well.

Previous versions of our dynamic instrumentation [10] could instrument functions, basic blocks, and instructions. In this work, we add edge instrumentation. Edge instrumentation is not new, having been used in static binary editors, such as OM [20] and EEL [12]. Edges created by unconditional jumps can be instrumented simply by instrumenting the last instruction of the edge's source block. We do not instrument edges created by indirect jumps because they are not used for control transfer that create loops (they are typically used for jump tables and dynamic call sites). To instrument edges created by conditional jumps we create *edge trampolines*.

Figure 2 (b) illustrates how we instrument conditional jumps using edge trampolines. An edge trampoline is a code fragment that contains two new basic blocks, one that corresponds to execution of the fall-through edge and one for the taken edge (the shaded regions). We can instrument either edge by instrumenting its new basic block using our existing technique for creating instrumentation points. The conditional jump is overwritten with an unconditional jump to the edge trampoline. The conditional jump is relocated to the trampoline but is given a new target address. This simulates the execution of the original conditional jump but with our two new blocks as targets. These new blocks end with jumps to the original conditional jump targets.

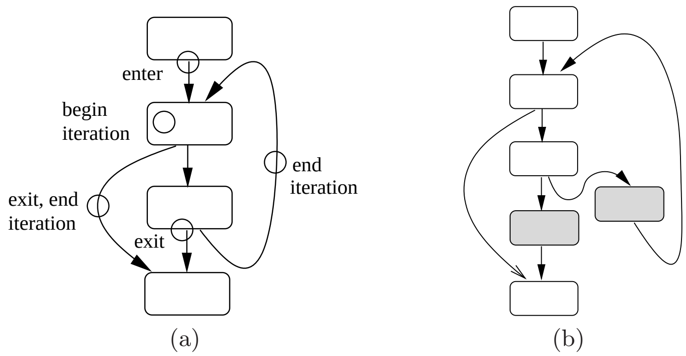


Fig. 2. Loop instrumentation points in a flow graph, indicated with circles, in (a). Instrumenting the conditional jump that creates the loop's back edge is shown in (b). An edge trampoline is used to create two new basic blocks (shaded) that correspond to the execution of the taken and fall-through edges of the conditional jump.

We use an absolute jump instruction to ensure that the edge trampoline can be reached from the application code we are instrumenting. On CISC architectures, such as the IA32, an absolute jump instruction may be larger than a PC relative jump instruction. This means the absolute jump we use to transfer control to the edge trampoline may be larger than the conditional jump that it replaces. In this case, we relocate enough instructions before the conditional jump to the head of the edge trampoline to make room. We may safely relocate all instructions in the basic block terminated by the conditional jump.

Though rare, the size of the entire basic block may be smaller than the size of the unconditional jump instruction. To handle this case, we can use a short trap instruction. The trap is caught by a handler that sets the application's PC to be the start address of the edge trampoline. The disadvantage of this approach is that performance suffers due to the cost of handling the trap.

A better strategy uses *function relocation*. When the Performance Consultant determines that the basic blocks relevant to loop instrumentation in a function are not large enough to be efficiently instrumented, it rewrites the function in a new location in the application's address space. When rewriting the function, nop instructions are inserted along with the original code to ensure that the relevant basic blocks are large enough. The Performance Consultant overwrites the beginning of the original function with a jump to the newly created copy.

5 A Loop-aware Search Strategy

Loops form a natural extension to our callgraph-based search (see Figure 1). As such, conceptually the Performance Consultant can treat them as additional steps in its refinement. We have also defined loop instrumentation points similar to those for functions (Section 4). As a result, loops are not just conceptual steps but actual steps in the Performance Consultant's search.

The Performance Consultant performs a breadth-first search of the application's callgraph. Function entry and exit points are instrumented to collect inclusive time-based metrics. If a function is not a bottleneck then it is pruned from the search space. Otherwise, the search continues by instrumenting the functions that it calls. If a function is a bottleneck, then the search continues by instrumenting the functions that it calls that are *not* nested under any loops, as well as the function's outermost loops. If a loop is a bottleneck then we instrument its children: the functions that are called directly within this loop, and the loop's directly nested loops. If a loop is not a bottleneck then the loop, and its descendants are pruned from the search.

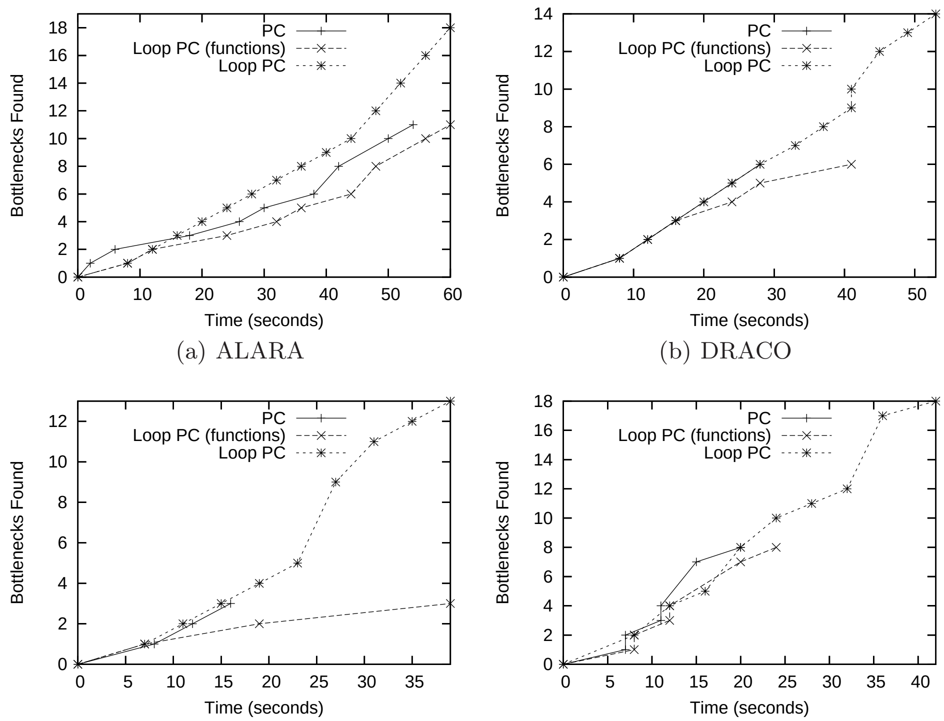
The addition of loops suggests that more experiments will be run. However, pruning a loop in the callgraph means its descendants are also pruned. Instrumenting a non-bottleneck loop which contains multiple function calls reduces the number of experiments run because the called functions do not have to be instrumented. Depending on the structure of the program's code, adding loops to the callgraph can cause more or fewer experiments to be run.

We found our top-down approach successful in practice, though other approaches may work as well. A search strategy can decide whether or not to instrument a function's loops based on static information like the function's depth in the call graph or the structure of the function's flow graph. Dynamic information like the function's current performance data can also be used to influence the decision. Unlike techniques that statically instrument a binary, dynamic instrumentation allows the search to use fine-grained instrumentation only when necessary, and to make the decision of which loops to instrument at run time. This enables our loop search strategy to compliment other strategies, such as Deep Start [19], that are able to quickly locate functions that are performing poorly. Once the problematic functions are found, loop instrumentation can be used to more precisely locate bottlenecks within these functions. A search strategy that dynamically evaluates the tradeoff between the low overhead of function-level instrumentation and the increased precision of loop-level instrumentation can reap the benefits of both techniques.

6 Experimental Results

To evaluate our loop-aware search strategy, we compared it to the Performance Consultant's current search strategy. We performed experiments on two sequential and two parallel (MPI and OpenMP) scientific applications (see Table 1). Table 2 lists more detailed application characteristics, including loop information. We used MPICH version 1.2.5 for our MPI implementation, version 5.2 of the Portland Group Compilers for the Fortran and C applications, and gcc version 3.3.3 for the C++ application.

For all experiments, we used 3GHz Pentium 4 PCs with Hyperthreading enabled, 2 GB RAM, running Tao Linux (a repackaged version of Red Hat



(a) ALARA (b) DRACO

(c) OM3 (d) SPhot

Fig. 3. Search profiles for the sequential (ALARA and DRACO) and parallel (OM3 and SPhot) applications. "PC" indicates the default Performance Consultant search, "Loop PC" indicates our loop-aware search strategy. "Loop PC (functions)" indicates that our loop-aware search strategy was used but only functions were counted as bottlenecks.

Enterprise Linux 3), connected using 100 Mb Ethernet. Both MPI applications were run on identically configured PCs. The Paradyn front-end process was run on a different machine than the applications. We performed multiple runs of each application. During each run, we began the Performance Consultant search once the application reached steady-state behavior.

Name	Version	Type	Nodes	Language	Domain
ALARA	2.7.1	Sequential	1	C++	Induced radioactivity analysis
DRACO	6.0	Sequential	1	Fortran 90	Hydrodynamic simulation
OM3	1.5	MPI	8	C	Global ocean circulation
SPhot	1.0	MPI/OpenMP	8	Fortran 90	Monte Carlo transport code

Table 1. Application characteristics.

Name	Lines	KB	Funcs	Loops	Loops/Func	% Loops at nesting					
						1	2	3	4	5	6
ALARA	21,099	6,382	718	598	0.8	76	20	3	0.8	0.5	0.1
DRACO	72,305	2,516	898	5,477	6.1	47	32	16	4	1	0.1
OM3	2,673	88	28	202	7.2	40	32	21	7	0	0
SPhot	2,932	895	31	106	3.4	53	22	5	18	2	0

Table 2. Application characteristics, including the number of loops, as a percent of the total number of loops, for 6 nesting levels.

	Total Bottlenecks		Leaf Bottlenecks		Experiments/second	
Name	Function	Loop	Function	Loop	PC	Loop PC
ALARA	11	8	4	3	0.9	1.0
DRACO	6	8	2	3	0.4	1.5
OM3	3	10	1	4	1.6	1.7
SPhot	8	10	3	5	2.9	1.9

Table 3. Types of bottlenecks and rate of experimentation. “PC” is the default Performance Consultant search, “Loop PC” is our loop-aware search strategy.

Our experiments indicated that loops were frequently bottlenecks (Table 3), which was expected since we examined scientific applications. In total, the applications contained 10 function bottlenecks that were leaf nodes in the callgraph. Of these 10, 7 contained loop bottlenecks. Loops significantly increase bottleneck precision. For example, OM3 contains a single function bottleneck, `time_step`, that is a leaf node in the callgraph and consumes 85% of the application's CPU time. While this information is useful, `time_step` is a large function that contains 90 loops. The Performance Consultant reports that 8 of these loops are bottlenecks, and that 4 of these bottleneck loops are leaf nodes in the augmented callgraph. Loop-level data indicates that a third of the time spent in `time_step` can be attributed to loop 12.

Figure 3 compares the rate at which bottlenecks are found using the Performance Consultant's default search strategy and our loop-aware strategy. Results are shown both considering and ignoring loop bottlenecks. Both search strategies find bottlenecks at similar rates. The loop-aware strategy finds more total bottlenecks (due to loops) but takes longer to identify function bottlenecks (due to the increased height of the augmented callgraph).

We did not observe significant differences in the rate of experimentation required to search loops (Table 3). For two of the applications the rate of experimentation was almost the same for both search strategies. The loop-aware Performance Consultant required a faster rate of experimentation for one application and a slower rate for the other. In general, we observed more precise results without a major change in search time or the rate of experimentation. Since we use an automated online search strategy that only instruments the loops of the functions currently in question, we only pay the cost of fine-grain instrumentation when necessary.

7 Conclusion

Searching loops proved to be a natural extension of the Performance Consultant's callgraph-based online automated search strategy. Since loops are frequently bottlenecks our strategy often provides a more precise performance diagnosis. Loops partition functions for searching without dramatically increasing the number of necessary experiments. We have defined points in a flow graph that correspond to loop execution, and presented a technique for the efficient instrumentation of these points. This mechanism enables our loop-aware search strategy to more precisely locate bottlenecks without large changes in search time or cost.

References

1. Aho, A., Sethi, R., Ullman, J., **Compilers: Principles, Techniques and Tools**, Addison-Wesley, 1985.
2. Cain, H. W., Miller, B. P., Wylie, B. J.N.: A Callgraph-Based Search Strategy for Automated Performance Diagnosis. *Concurrency and Computation: Practice & Experience* **14**, 3, 203-217 March 2002. Also appears as Euro-Par 2000, Munich, Germany, August 2000.
3. DeRose, L., Mohr, B., Seelam, S.: Profiling and Tracing OpenMP Applications With POMP Based Monitoring Libraries. *Euro-Par*, Pisa, Italy, August 2004, pp. 39-46
4. Gerndt, H.M., Krumme, A.: A Rule-Based Approach for Automatic Bottleneck Detection in Programs on Shared Virtual Memory Systems. *2nd Intl. Workshop on High-Level Programming Models and Supportive Environments*, Geneva, Switzerland, April 1997.
5. Goldberg, A.J., Hennessy, J.: MTOOL: A Method for Isolating Memory Bottlenecks in Shared Memory Multiprocessor Programs. *International Conference on Parallel Processing*, Augsust 1991, pp. 251-257.
6. Graham, S., Kessler, P., McKusick, M.: An Execution Profiler for Modular Programs. *Software Practice & Experience* **13** No. 8, August 1983, pp. 671-686.

7. Helm, B.R., Malony, A.D., Fickas, S.F.: Capturing and Automating Performance Diagnosis: the Poirot Approach. *Intl. Parallel Processing Symposium*, Santa Barbara, California, April 1995.
8. Hirzel, M., Chilimbi, T.: Bursty tracing: A framework for low-overhead temporal profiling. *4th ACM Workshop on Feedback-Directed and Dynamic Optimization*, Austin, Texas, December 2001.
9. Hollingsworth, J.K., Miller, B.P.: Dynamic Control of Performance Monitoring on Large Scale Parallel Systems. *International Conference on Supercomputing*, Tokyo, July 1993.
10. Hollingsworth, J.K., Miller, B.P., Cargille, J.: Dynamic Program Instrumentation for Scalable Performance Tools. *Scalable High Performance Computing*, Knoxville, Tennessee, May 1994.
11. Karavanic, K.L., Miller, B.P.: Improving Online Performance Diagnosis by the Use of Historical Performance Data. *SC99*, Portland, Oregon, November 1999.
12. Larus, J.R., Schnarr, E.: EEL: Machine-Independent Executable Editing. *Programming Language Design and Implementation* (1995), pp. 291-300.
13. Lengauer, T., Tarjan, R.E.: A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems (TOPLAS)* **1**, 1, July 1979, pp. 121-141.
14. Mellor-Crummey, J., Fowler, R., Marin, G.: HPCView: A tool for top-down analysis of node performance. *Los Alamos Computer Science Institute Second Annual Symposium*, Santa Fe, NM, October 2001.
15. Miller, B.P., Callaghan, M.D., Cargille, J.M., Hollingsworth, J.K., Irvin, R.B., Karavanic, K.L., Kunchithapadam, K., Newhall, T.: The Paradyn Parallel Performance Measurement Tools, *IEEE Computer* **28**, 11, November 1995.
16. Muchnick, S., **Advanced Compiler Design and Implementation**, Morgan Kaufmann, 1997
17. Mukerjee, N., Riley, G.D., Gurd, J.R.: FINESSE: A Prototype Feedback-Guided Performance Enhancement System. *8th Euromicro Workshop on Parallel and Distributed Processing*, Rhodos, Greece, January 2000.
18. Reed, D. A., Aydt, R. A., Noe, R. J., Roth, P.C., Shields, K.A., Schwartz, B., Tavera, L.F.: Scalable Performance Analysis: The Pablo Performance Analysis Environment. Anthony Skjellum(ed). *Scalable Parallel Libraries Conference*, October 1993, pp. 104-113.
19. Roth, P.C., Miller, B.P.: Deep Start: A Hybrid Strategy for Automated Performance Problem Searches. *Concurrency and Computation: Practice and Experience* **15** 11-12, September 2003, John Wiley & Sons, pp. 1027-1046. Also appeared in shorter form in Euro-Par 2002, Paderborn, Germany, August 2002, LNCS 2400, Springer Verlag.
20. Srivastava, A., Eustace, A.: ATOM: A system for building customized program analysis tools. *Programming Language Design and Implementation* **11**, June 1994, pp. 196-205.
21. Intel®VTune™Performance Analyzer.
`http://www.intel.com/software/products/vtune/`
22. Official OpenMP Fortran Version 2.0 Specification.
`http://www.openmp.org/drupal/mp-documents/fspec20.pdf`