Incremental Call-Path Profiling[1]

Andrew R. Bernat Barton P. Miller
Computer Sciences Department
University of Wisconsin
1210 W. Dayton Street
Madison, WI 53706-1685
{bernat,bart}@cs.wisc.edu

Abstract

Call-path profiling attributes execution cost to the path taken to reach a function. Previous call-path profilers tracked the call-path at all times, which requires instrumentation of the entire program. Since this instrumentation was frequently executed, they were restricted to calculating simple metrics, such as function call counts.

We present a new method for call-path profiling called incremental call-path profiling. We profile only a subset of the functions in the program, allowing the use of more complex metrics while lowering the overhead. This combination of call path information and more complex metrics is particularly useful for localizing bottlenecks in frequently called functions.

We also describe the implementation and application of iPath, an incremental call-path profiler. iPath was used to profile two real-world applications: the Paradyn instrumentation daemon and the MILC su3_rmd QCD distributed simulation. In both applications we found and removed call-path specific bottlenecks. Our modifications to the Paradyn instrumentation daemon greatly increased its efficiency. The time required to instrument our benchmark program was reduced from 296 seconds to 6.4 seconds, a 98% decrease. Our modifications to su3_rmd reduced the running time of the program from 3001 seconds to 1652 seconds, a 45% decrease.

1. Introduction

Call-path profiling is a mechanism by which the information gathered by the profiler is attributed to paths through the call graph rather than to individual functions. Call-path profiling is able to gather more precise information about the execution of the program than a conventional profiler, but with greater cost. Previous whole-program call-path profilers added instrumentation at all function call sites, entry points and exit points [11,12]. This instrumentation can slow the execution of the profiled program by 300-700% even when performance metrics are not being gathered [10].

Whole-program profilers require complete instrumentation to track the call-paths taken by a program. Due to this requirement the performance metrics supported by many profilers are limited to metrics that are inexpensive to gather. The user of the profiler is often only interested in the performance characteristics of a few functions in the program. In this case the information offered by a whole program profiler is both too broad and too limited.

Incremental call-path profiling is a technique that allows a programmer to examine the behavior of particular functions, rather than the whole program. This method operates by adding instrumentation only to the profiled functions. Overhead is only incurred when the profiled functions are executed. We can also make use of a greater variety of performance metrics than are available from whole-program profilers. The data gathered by an incremental call-path profiler require no postprocessing, and are immediately available to the programmer. Finally, incremental path profiling is amenable to dynamic (run-time) instrumentation.

Call-path profiling information is useful when optimizing commonly used functions, such as a library function, that are called from many different locations. An example of such a function is an MPI communication method or a C

1. This work is supported in part by Department of Energy Grants DE-FG02-93ER25176 and DE-FG02-01ER25510, and Lawrence Livermore National Lab grant B504964. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

library function. In this case, knowing only the aggregate behavior of the function is misleading if the behavior of the function varies widely depending on the caller.

This paper presents the design, implementation, and application of iPath, an incremental call-path profiler. This tool is capable of gathering a wide variety of performance metrics for the call-path while instrumenting only the functions of interest to the user. iPath uses a inexpensive stack walk mechanism to record the call-path to a function and uses this walk to attribute the performance data. iPath supports both counter and timer-based performance metrics and is capable of calculating metrics based on both hardware counters and software resources. The data gathered are available in a shared memory segment for immediate examination and use.

We begin by comparing iPath to existing work in the area of call-path profiling. We then discuss the design of iPath and show that while the instrumentation used is more complex than in static call-path profilers, the overall cost is smaller when small groups of functions are selected. As a test of our technique, we apply iPath to two real systems, the Paradyn dynamic profiler [13] and su3_rmd, a distributed quantum chromodynamics simulation built on the MILC framework [3]. In both applications, we found and removed call-path specific bottlenecks. Our modifications resulted in an almost 98% decrease in Paradyn instrumentation time and a 45% decrease in running time of su3_rmd.

2. Related Work

Call-path profiling is a well-known approach for gathering detailed information about the behavior of a program. Several projects have investigated methods of offering call-path profiling with minimal overhead. These approaches can be divided into three categories based on their mechanism for generating the call-path and their method of data collection. The first category consists of profilers that approximate call-paths and provide only partial path information. The second group maintains a snapshot of the current call-path through the entire execution of a program, updating it at function call boundaries via instrumentation. The final category consists of profilers that use sampling to identify call-paths and gather performance data.

The tools gprof [6] and sprof [15] are both examples of the first category. These profilers use instrumentation to accurately count function entries and exits, and use sampling to approximate CPU usage. From this information partial call-paths of length two are generated. Neither profiler supports the use of hardware performance metrics, limiting the variety of information available to the user.

The second category consists of call-graph profilers that track the current call-path throughout the execution of the program. The first of these, PP, is an intraprocedural path profiler developed by Ball and Larus [2]. PP instruments transitions between basic blocks to track the execution path within individual functions. PP was extended [1] to use the calling context of a function to approximate the call-path. This approximation uses a construct called a *Calling Context Tree (CCT)* that represents the call-tree of a program in a more compact form. However, the CCT cannot track recursive calls, collapsing all recursive calls to a particular function to a single node in the CCT. PP is also capable of accessing the hardware counters on the SPARC platform, but only uses non-virtualized timers. This can lead to inaccuracies due to context switches.

Melski and Reps extended PP with a technique that could directly track interprocedural paths without approximation [11,10]. This approach assigned a unique identifier to every possible path through a program and used the identifier to label collected performance data. Their technique adds instrumentation to all function entries, exits, and call sites. The instrumentation manipulates a counter value that corresponds directly to the current call-path. To handle recursion and function pointers, they require post-execution processing of the data.

Larus also developed a method, Whole-Program Paths (WPP) [8], to record a block-by-block trace of a program's execution and represent it with a compact grammar. He used PP to determine and record the paths taken within a function, and from this information pieced together a representation of the whole program's execution. While this representation included all call-paths taken by the program, they were not associated with any performance information other than function call counts.

The TAU performance tools [9] also provide call-path specific profiling data. The TAU system traces function entries and exits via instrumentation code inserted in the program and uses this information to maintain a stack of currently executing functions. The measurement system walks this stack to associate gathered performance data with the correct call-path. Since TAU tracks function entries and exits directly, pointer-based calls and recursion do not present a problem.

DeRose and Wolf developed CATCH [5], a tool that associates hardware metrics with call-path information for MPI and OpenMP applications. Like iPath, CATCH is built on the Dyninst instrumentation library [4]. CATCH analyzes the call-graph of the program and uses call-site instrumentation to maintain a representation of the current call-path. The user can select subtrees of the call-graph to profile rather than tracking the entire execution of the program. This allows CATCH to reduce the amount of instrumentation inserted in the program. Unfortunately, CATCH does not support profiling of programs that use dynamic calls through function pointers.

Hall developed a profiler, CPPROF, that uses sampling to collect performance data [7]. CPPROF uses the Solaris /proc file system to gather information about a profiled process. Each sample consists of a stack walk and the current value of the performance metrics used. This information is used to generate a complete call-path profile.

Our work introduces a fourth category of call-path profilers. We differ from the first category of profilers by generating precise call-path information instead of an approximation. We also generate the entire call-path instead of only a segment. Finally, we take advantage of a variety of performance data, including hardware performance metrics.

PP, its extension by Melski and Reps, WPP, and TAU all depend on global instrumentation to maintain a current call-path. These profilers all gather information that must be post-processed after the program completes. CATCH uses a similar approach, instrumenting all call-sites in profiled subtrees of the call-graph. Our approach differs by instrumenting only the individual functions that are being profiled and building the call-path on the fly.

CPPROF also makes its profile data available at run-time, and both CPPROF and iPath use a stack walk to determine the current call path. However, the sampling approach used by CPPROF differs greatly from our instrumentation approach. Sampling gathers information about the entire process, with low overhead, but cannot provide precise information about particular functions.

We also present the concept of *incremental* path profiling, in which it is possible to target specific functions to be profiled instead of tracing the entire program. Previous profilers do not target particular functions.

3. Design

Incremental path profiling has four key characteristics. First, only the entries and exits of the functions actively being profiled are instrumented. Second, we determine the current call-path via a stack walk instead of tracking function call sites. Third, profile data is accessible at run-time with no further post-processing required. Finally, instrumentation can be added to or removed from particular functions without instrumenting other functions. In this section, we elaborate on these four points, and then describe iPath, our implementation of an incremental path profiler.

Incremental path profiling operates by adding instrumentation only to the functions that are profiled. This instrumentation determines the current call-path and collects desired performance information. Since instrumentation is executed at only the entry and exits of profiled functions, the overhead of an incremental path profiler should be less than a whole-program path profiler even when more expensive metrics are used.

Our instrumentation determines the current call-path by taking a inexpensive stack walk. Since we are walking the actual stack, we detect recursion and calls through function pointers. In addition, calls from different call-sites within the same function will have different return addresses on the stack and can be distinguished from one another. Stack walking is one of our fundamental mechanisms, so it is important that it is as efficient as possible. Conceptually, all that is required is to follow the frame pointer down the stack until the base is reached. As we discuss in Section 4.3, several optimizations make stack walking more complicated.

Once the stack has been walked to determine the current call-path, the desired performance metrics are sampled. We divide metrics into two categories, counters and timers. Counters, such as the number of times a function is executed on a particular call-path, require only a single point of instrumentation. Once the call-path is determined, the counter associated with that path is incremented at function entry; no exit instrumentation is required. The other category of metrics, timers, measure the change in a metric over the execution of a function and require two points of instrumentation. The value of the metric is recorded at the entry of the function and is used to calculate the change in the metric at the end of the function.

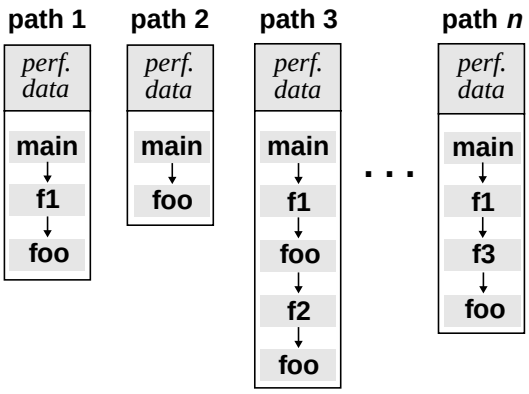

Figure 1: Representation of call-path profiles

The profile data is stored as a collection of call-path data structures. Each call-path structure consists of a list of addresses that identify the call-path and the profiling information associated with that path. This data can be viewed by the user at any time. This makes it possible to profile *incrementally*, continually refining the instrumentation inserted based on the profile data returned. If a dynamic instrumentation library is used, this refinement can be done at run-time.

4. Implementation

We implemented the technique described above in iPath using the DyninstAPI instrumentation library to instrument the profiled program. Dyninst provides an interface that allows us to attach to and instrument a running program. iPath is implemented on POWER/AIX and IA-32/Linux and is capable of using a variety of hardware and software metrics.

iPath consists of two parts: (1) a Dyninst mutator that instruments the application and (2) a run-time library containing the stack walk and profiling logic, which is injected into the application. The profile results are stored in a shared memory segment. iPath provides a command-line interface to identify the functions to be profiled. We currently do not take advantage of Dyninst's ability to modify instrumentation while the program is being executed.

4.1. Mutator

The iPath mutator is responsible for starting the application or attaching to a previously running application. Once we have created or attached to the application we inject our run-time library into the program and initialize it. The initialization process creates a shared memory segment that is used to store profile data. This allows the data to be read at runtime without pausing the application.

We insert calls to our library at the entry and exit of all functions requested by the user. These calls perform the stack walk and collect performance data. If a single requested function name matches multiple functions within the program, each of these functions is instrumented and the performance information is kept separate. Multiple versions of a function can occur in the demangled names in C++ or in local functions whose names are not visible outside of a module.

Once the instrumentation has been inserted, the program is run with no further manipulation by the mutator. Since a shared memory segment is used to store the profile data, it is possible to sample this from the mutator without pausing the program. The mutator periodically prints out a summary of the profile data while the program runs and displays a final version when the program completes. This summary includes function counts, total CPU usage, and average CPU usage for each call-path. iPath also calculates the percentage of the entire execution each call-path takes, which is useful to determine call-path specific bottlenecks.

4.2. Run-Time Library

The iPath run-time library is responsible for performing the call-path-based sampling and recording the data gathered. The run-time library consists of three major segments: the data structures that store profiling data, the stack walk logic, and the entry and exit instrumentation.

iPath stores detected call-paths and their associated profiling information as shown in Figure 1. Each profiled function has its own copy of these structures. Each call-path is associated with the performance data collected for that call-path. If a profiled function is entered recursively, there will be separate call-path for each unique call-path to the function.

Our entry instrumentation walks the stack to determine the current call-path. While a stack walk is conceptually simple, in practice it can be difficult due to compiler optimizations. We discuss these optimizations and our methods for handling them in the next section. The stack walk acquired is used to look up the appropriate call-path profile by searching the table of previously seen call-paths to the profiled function and looking for a matching stack walk. If no match is found, then a new call-path profiling structure is initialized and added to the table.

Once the appropriate profile structure has been determined the desired performance metrics are gathered. Timers, such as CPU time, are started by storing the current timer value as a starting value. Counters, such as function call counts, are simply incremented. Finally, the current call-path is added to a stack of active timers for that particular function. This *active stack* is used by the exit instrumentation to short-cut the call-path lookup process at function exit and is described below.

The exit instrumentation is responsible for stopping any active timers. First, as with entry instrumentation, we determine the active call-path with a stack walk and a table lookup. Unlike entry instrumentation, if no match is found no actions are performed. We do this for safety reasons. Once the active call-path has been determined, timers are stopped by again sampling the current timer value, subtracting the starting value, and accumulating the difference. Finally, the top of the active stack (corresponding to this exit) is removed.

To make our instrumentation as efficient as possible we have included a few optimizations. Most importantly we match function exits to entries through a mechanism we call the *active stack*, which is unique for each profiled function. Normally, the call-path to a function will not change between the entry to this function and its exit. In these cases, the call-path determined by the entry instrumentation can be reused by the exit instrumentation. Whenever a function is entered we push a the current call-path onto the active stack. When we perform our exit instrumentation we compare the call-path to the top of the active stack before searching in the table of paths. We use a stack instead of a single element to handle recursive entry of the instrumented function, which will result in multiple active paths. If the profiled function is never entered recursively, this stack will have a maximum height of 1. Certain programming constructs, such as signals and setjmp/longjmp, can cause a program to enter a function but not leave. For this reason we always ensure the cached path is correct. Finally, the path at the top of the stack is removed.

4.3. Stack Walks

While a stack walk is conceptually simple, several optimizations can make the process more difficult. We identify three categories of optimizations that effect stack walks: functions that do not create stack frames, functions that modify their stack frame during execution, and functions that do not record a pointer to the previous stack frame when creating a new frame. We avoid the complexities of walking partially constructed stack frames by only walking the stack at the entry and exit of a function.

The first category of optimizations consists of functions that do not create stack frames. This type of optimization will cause a function to not appear in a stack walk and therefore the call-path derived from the stack walk. There are two common examples of frameless functions, leaf functions without stack frames and inlined functions. Fortunately, a leaf function will never occur in the middle of a stack walk as it makes no calls. Since we only have to handle optimizations in the caller of the profiled function, any leaf optimizations in the profiled function will not impede our stack walking. Inlined functions are another matter. We do not distinguish inlined functions in our stack walks, relying on other information (such as the symbol table) to reconstruct the original form of the function. Currently iPath presents the call-path without the inlined function.

The second group of optimizations consists of functions that modify their stack frame during execution. We have seen this optimization in several functions in the AIX math library. These functions normally execute without a stack

frame. If an error is detected they create a stack frame before handling the error. We can take accurate stack walks even in the presence of this type of optimization as long as the modifications to the stack frame are completed by the time a call is made. This is true for every case of this optimization of which we are aware.

Our final category consists of optimizations that create stack frames that do not contain a pointer to the previous frame. This makes it impossible to identify the previous stack frame without knowing, through some other mechanism, the size of the current stack frame. We have seen this optimization on IA-32. It uses two registers, the stack pointer and the frame pointer, to track the stack. The stack pointer moves throughout the execution of each function,

	Unoptimized (min:sec)	**Optimized (min:sec)**
Single-threaded	4:30	3:50
5 threads	4:30	3:55
10 threads	5:45	4:10
20 threads	8:00	4:25

Table 1: Performance Consultant Analysis

Time required to complete a full automated analysis of the tested programs. The same program was used for all three multi-threaded tests.

while the frame pointer is static and contains the pointer to the previous frame. It is possible to omit the frame pointer so the register can be reused, making it impossible to find the previous frame. We detect this case and abort the stack walk. We are investigating how often it is feasible to determine the size of the stack through code analysis rather than relying on the frame pointer.

5. Results

We applied our profiler to two different applications, the Paradyn instrumentation daemon and the MILC su3_rmd QCD simulation code. In both cases we were able to use call-path profile data to make large improvements in the running time of the programs. We were able to identify a utility function in the Paradyn daemon that was not well-optimized for its most frequent caller. Optimizing this function resulted in an almost 98% decrease in Paradyn instrumentation time. We also examined the use of the MPI library by su3_rmd and identified several synchronization bottlenecks. By fixing these bottlenecks we were able to reduce the running time overall program by 45%. Our instrumentation added less than 1% overhead to the execution time of the Paradyn daemon and 8% overhead to the execution time of su3_rmd.

5.1. Paradyn Daemon

iPath was used to identify and remove a major bottleneck in the Paradyn instrumentation daemon. Through the use of path profiling, we were able to locate a utility function that was being called frequently. This function was incorrectly optimized, by reimplementing the function, we significantly reduced the overall instrumentation time of the daemon. This reduction in instrumentation time also resulted in a visible user-level performance improvement.

We noted that Paradyn's automated analysis of a multi-threaded program was significantly slower than the analysis of a similar single-threaded program. We investigated this behavior and determined that the slowdown was due to the instrumentation section of the daemon. Requests for instrumentation were taking significantly longer on a multi-threaded program than they were for a single-threaded program.

To evaluate Paradyn's performance, we ran Paradyn's automated search tool, the Performance Consultant (PC), on a multi-threaded program while profiling the daemon with iPath. The daemon received 107 instrumentation requests from the PC, and spent 48.5 seconds inserting instrumentation. Of this time, 45 seconds were spent determining whether to manually trigger instrumentation in a function called `doCatchupInstrumentation`.

We examined this function and its callees, which perform three steps to determine which instrumentation to trigger. First, a stack walk is taken of each thread. Second, each frame on each stack is compared to the instrumentation request to determine whether to trigger the instrumentation. Third, instrumentation is started if necessary. Further

profiling led us to narrow down the problem to a function, `triggeredInStackFrame`, that performed the comparison of frames with instrumentation requests.

We iteratively instrumented this function and its callees with iPath, finally narrowing down the bottleneck to a utility function called `findFuncByAddr`. iPath's profile of `findFuncByAddr` showed that 81% of the calls to this function were from `triggeredInStackFrame`, consuming 41.5 seconds of CPU time. Out of 48.5 seconds of instrumentation time, 85% was being spent in this utility function. Worse yet, this function was called for each frame in each stack walk. As the number of threads or the stack sizes increased, so did the number of calls.

We examined `findFuncByAddr` and pinpointed the location of the bottleneck. This function was used to map from an address to the function that contained the address. The data structure that stored this mapping was a hash table keyed by a single value, the entry address of the function. This caused look-ups of an address within the body of

	Unoptimized (seconds)	Optimized (seconds)
Single-threaded	1.5	0.1
5 threads	48.5	1.2
10 threads	126.9	3.0
20 threads	296.1	6.4

Table 2: Instrumentation Time

Time spent by the daemon performing instrumentation during the Performance Consultant analysis.

a function, such as an address in a stack walk, to be much slower than looking up using the entry address. We needed a structure that could map from a range instead of a single value.

We reimplemented `findFuncByAddr`, using a balanced tree instead of a hash table. This tree structure performed range lookups much faster than the hashtable. In addition, we cached recent results instead of repeating lookups. The results were impressive. When we re-ran our benchmark, instrumentation time was reduced to 1.2 seconds, a speedup of over 40. We timed the old version against the new, with the results in Tables 1 and 2.

In summary, we were able to use iPath to determine two things. First, we were able to determine that the majority of calls to `findFuncByAddr` came from a single source. This led us to reduce the number of calls by reusing results where possible. Second, we determined that `findFuncByAddr` was not optimized for the common call-path. Often it is difficult to determine the most common call-path to a utility function with a standard profiler or through examination of source code, but a call-path profiler is able to determine this information. These two modifications resulted in a substantial performance improvement.

5.2. MILC

We used iPath to investigate su3_rmd, a distributed quantum chromodynamics simulation built on the MILC framework. Our aim was to find synchronization bottlenecks within the program. We gathered a call-path profile of each of the blocking MPI calls used by su3_rmd. Two of these functions, MPI_Allreduce and MPI_Wait, were bottlenecks. We were able to remove the MPI_Allreduce bottleneck by replacing calls to that function by an equivalent asynchronous operation.

The MILC project provides a framework for performing QCD simulations. The framework defines a lattice of data and mechanisms for accessing individual points on the lattice. Applications written with the framework use these mechanisms, which allow the applications to run on single machines or clusters without code modifications. The framework also provides several different mechanisms for determining how the lattice is distributed if the application is run on a cluster.

One of the simulations distributed with the MILC framework is su3_rmd, an implementation of the R algorithm for QCD simulation. The majority of the execution time of su3_rmd is contained within a single function, `ks_congrad`. This function consists of a loop that executes until a result value is less than a given threshold parameter. Each iteration through the loop consists of an interleaved set of three types of operations: gathering information about lattice points from neighboring nodes, performing vector operations on the lattice, and summing the results across all computing nodes.

We ran the su3_rmd simulation on four nodes of an IBM SP. We used iPath to profile all blocking MPI functions called by the simulation. We examined the resulting call-path profile and focused on the paths that passed through `ks_congrad`. This allowed us to unwind the communication abstraction used by the MILC framework. We discovered four synchronization bottlenecks in `ks_congrad`, two gather operations that made several calls to `MPI_Wait` and two calls to `MPI_Allreduce`.

We began by investigating the bottlenecks in `MPI_Wait`. These bottlenecks were caused by the lattice update operations. Each iteration through the loop in `ks_congrad` executes four gather operations. The MILC framework implements a gather as a series of messages finalized with calls to `MPI_Wait`. The two gathers executed in `ks_congrad` resulted in sixteen distinct call-paths to `MPI_Wait`. Our profiling showed that 50% of the execution time of `ks_congrad` was spent in calls to `MPI_Wait`, making it a prime candidate for optimization. In total, the simulation executed for 3001 seconds, as shown in line 1 of Table 3.

Version	Time (seconds)	Change
1. Original	3001	
2. Gather operation optimization	1843	-38.6%
3. MPI_Allreduce optimization	2810	-6.4%
4. Both optimizations	1652	-45.0%

Table 3: su3_rmd Running Time

Time spent in ks_congrad and synchronization bottlenecks before and after optimizations were made.

Our call-path analysis showed that the first call to `MPI_Wait` in each gather operation took up to twenty times longer to complete than the second call. Path profiling significantly simplified finding the particular call. Using this information, we traced the cause of this difference back to the initial send and receive operations. The bottleneck was due to the use of a synchronous send operation (`MPI_Issend`). We modified the send operation to be asynchronous (to use `MPI_Isend`) and reordered the calls to `MPI_Wait` to hide the transfer latency. These changes resulted in a 75% reduction in time consumed by `MPI_Wait` in gather operations and a decrease of 38.6% in the overall running time, as shown in line 2 of Table 3.

We then investigated the `MPI_Allreduce` bottlenecks. This function is used by `ks_congrad` to sum a single floating point value across all nodes executing the simulation. Our profile showed that this was not an efficient operation, with 22% of the total execution time of `ks_congrad` spent in calls to `MPI_Allreduce`. We replaced the calls to `MPI_Allreduce` with non-blocking equivalents that we interleaved into the other loop operations. Unfortunately, data dependencies in the loop prevented us from hiding all of the communication latency. This replacement resulted in a 30% decrease in time spent blocked in `MPI_Allreduce`, and a 6.4% decrease in total running time, as shown in line 3 of Table 3.

In summary, we used call-path analysis to discover four synchronization bottlenecks in the su3_rmd simulation. In all cases we were able to replace blocking or synchronous calls with asynchronous equivalents and reorder operations to hide the message passing latency. These optimizations combined reduced the running time of su3_rmd from 3001 seconds to 1652 seconds, a 45% decrease.

6. Summary

Call-path profiling is a valuable tool for performance analysis. We have presented a method of gathering call-path profile data for particular functions. This approach avoids the overhead incurred by whole-program call-path profilers by instrumenting only the functions of interest instead of all function entries, exits, and call sites within a program. We allow the use of more expensive metrics while reducing the total overhead.

We implemented this method in a tool, iPath, built on the Dyninst instrumentation library. iPath was used to isolate and correct a bottleneck in Paradyn's instrumentation path. This fix resulted in both a 98% decrease in instrumentation time and a corresponding speedup of Paradyn's automated performance analysis tool. We also used iPath to locate synchronization bottlenecks in the MILC su3_rmd simulation. We replaced several blocking and synchronous MPI calls with asynchronous equivalents, resulting in a 45% decrease in running time of su3_rmd. Both of these opti-

mizations were directed by call-path specific profiling data. This information allowed us to focus on only specific call-paths that caused bottlenecks.

7. References

[1] G. Ammons, T. Ball, and J. R. Larus, "Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling," *SIGPLAN '97 Conference on Programming Language Design and Implementation (PLDI)*. Las Vegas, June 1997, pp. 85-96.

[2] T. Ball and J. R. Larus, "Efficient Path Profiling," *29th Annual IEEE/ACM International Symposium on Microarchitecture*, Paris, December 1996, pp. 46-57.

[3] C. Bernard, M.C. Ogilvie, T.A. DeGrand, C. DeTar, S. Gottlieb, A. Kransitz, R.L. Siugar, and D. Toussaint, "Studying Quarks and Gluons on MIMD Parallel Computers", *International Journal of Supercomputer Applications* **5**, 61, 1991.

[4] B. Buck and J. K. Hollingsworth, "An API for Runtime Code Patching", *The International Journal of High Performance Computing Applications* **14**, 4, Winter 2000, pp. 317-329.

[5] L. DeRose and F. Wolf, "CATCH: A Call-Graph Based Automatic Tool for Capture of Hardware Performance Metrics for MPI and OpenMP Applications", *8th International Euro-Par Conference*, Paderborn, Germany, 2002.

[6] S. Graham, P. Kessler, and M. McKusick, "gprof: a Call Graph Execution Profiler", *SIGPLAN Symposium on Compiler Construction*, Boston, June 1982, pp. 120-126.

[7] R. J. Hall, "Call Path Refinement Profiles", *IEEE Transactions on Software Engineering* **21**, 6, June 1995.

[8] J. R. Larus, "Whole Program Paths", *SIGPLAN '99 Conference on Programming Languages Design and Implementation*. Atlanta, May 1999.

[9] A. D. Malony, S. Shende, R. Bell, K. Li, L. Li, and N. Trebon, "Advances in the TAU Performance System", *Performance Analysis and Distributed Computing*, Kluwer, Norwell, MA, 2003.

[10] D. Melski, "Interprocedural path profiling and the interprocedural express-lane transformation" *Ph.D. dissertation*, University of Wisconsin, Madison, 2002.

[11] D. Melski and T. Reps, "Interprocedural Path Profiling," Technical Report TR-1382, Computer Sciences Department, University of Wisconsin, Madison, September 1998.

[12] D. Melski and T. Reps, "Interprocedural Path Profiling", *CC '99: 8th International Conference on Compiler Construction*. Amsterdam, March 1999.

[13] 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 Tools", *IEEE Computer* **28**, 11, November 1995, pp. 37-46.

[14] *PMAPI home page*, http://www.alphaworks.ibm.com/tech/pmapi, February 2004.

[15] SPROF, Linux utility.