Incremental Call-Path Profiling[1]

Abstract

Profiling is a key technique for achieving high performance. Call-path profiling is a refinement of this technique that classifies a function's behavior based on the path taken to reach the function. This information is particularly useful when optimizing programs that use libraries, such as those for communication (MPI or PVM), linear algebra (ScaLAPACK), or threading. The behavior of functions in these libraries often varies widely depending on the caller, or the path taken to reach the caller. Unlike standard profiling techniques, call-path profiling is able to accurately characterize this call-path dependent behavior.

We present a new method for call-path profiling, called incremental call-path profiling. Previous call-path profilers have relied on global pre-instrumentation of the application; the overhead of this instrumentation limits these techniques to providing simple metrics, such as function call counts. In contrast, we instrument the application on the fly, allowing targeted analysis of particular functions. Our method can be applied at run-time, without previous program modification, and allows for the use of complex metrics such as synchronization waiting time, I/O blocking time, memory stall time, and CPU usage. As a result, call-path profiling can be applied to a wider variety of performance problems.

We describe iPath, a prototype incremental call-path profiler, and demonstrate the application of our technique to two real-world applications. iPath was used to analyze both the distributed MILC su3_rmd QCD simulation and the Paradyn instrumentation daemon. In the su3_rmd simulation, iPath detected a communications bottleneck in particular calls to the MPI library. We were able to interleave communication with other operations, achieving a 45% decrease in the running time of the simulation. In the Paradyn daemon, we discovered an incorrectly optimized utility function. By optimizing this function for the most frequent caller, we were able to achieve a 98% decrease in running time, from 296 seconds to 6.4 seconds.

1. Introduction

Profiling is a key component of the optimization process, and is vital to achieving high levels of performance. Function-level profiling, however, is limited in that it only gathers information about the aggregate behavior of functions in a program. This information is not sufficient to completely characterize a function whose behavior varies between calls; for example, a function that performs well with one type of input and poorly with a second. Call-path profiling, an extension to function-level profiling, is a mechanism by which the information gathered by the profiler is attributed to paths through the call graph instead of being aggregated together. This more complete profiling information is particularly useful when optimizing programs that use libraries, such as those for communication (MPI or PVM), linear algebra (ScaLAPACK), or threading.

Call-path profiling allows a user to identify and characterize performance problems that depend on from *where* a function is called, rather than on the function itself. A function may complete quickly for most invocations, but

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.

require excessive time when executed along a particular call-chain. For example, a communication function may block waiting for input in particular cases, or a utility function may perform poorly for a particular input. Call-path profiling also clarifies performance problems caused by inefficient calling behavior, such as when a function is called an excessive number of times. For example, a math function may be called multiple times with identical input. Function-level profiling would mis-identify the math function as a bottleneck, but call-path profiling would identify the calling behavior as the problem.

Previous instrumentating call-path profilers have traced all functions in the program [1,2,5,10,11,12,13]. These tools have tracked the call-path at all times by instrumenting all function entries and exits. While this complete information can be useful in some cases, it generates substantial overhead and can slow the program by up to 700% [12]. Other profilers [7,8,16] have used sampling instead of instrumentation to lessen the overhead involved in profiling, but cannot generate precise results. Our work is motivated by the insight that users often do not desire information for every function in the program, focusing instead on a particular set of functions for which they need more exact information about their behavior. In this case, the information offered by a whole-program profiler is both unnecessarily broad and too expensive. By focusing instead on particular functions, a profiler may reduce the overhead involved in call-path profiling. However, previous call-path profiling algorithms are not amenable to this partial profiling approach, as they require whole-program instrumentation to track the call-path. We have developed a partial profiling method that does not require whole-program instrumentation to generate accurate call-path profiling results. Our technique complements other profiling tools and approaches, and provides an efficient method for profiling the call path of particular functions.

In this paper, we describe a technique that we call incremental call-path profiling. Incremental call-path profiling allows a user to profile particular functions rather than the entire program. The profile data may be examined while the program is running, and the user may refine both the selected functions and the metrics gathered while the program is still running. By profiling particular functions, we provide the benefits of call-path profiling while significantly reducing the overhead incurred.

This method is distinguished by five key capabilities:

- **Targeted**: Our technique is able to profile particular functions, and only these functions are instrumented;
- **Dynamic**: Profiling instrumentation can be inserted and removed at run-time, without prior program preparation;

- **Run-Time Results:** Up-to-date profiling results are available as the program runs, without requiring a post-processing phase;
- **Cost-effective**: Overhead is incurred only when the profiled functions are executed, allowing the use of more complex metrics;
- **Complete**: The entire call-path is captured, including dynamic calls through function pointers.

This paper presents our method for incremental call-path profiling. We begin by discussing existing work in the area of call-path profiling. We then discuss our key technique, the use of a lightweight stack walk to determine the complete call-path. This method allows us to determine the complete call-chain without requiring whole-program instrumentation. We also describe the implementation and application of iPath, a prototype 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 supports both counter and timer-based performance metrics and is capable of calculating metrics based on both hardware counters.

As a test of our technique, we apply iPath to two real systems, the su3_rmd distributed quantum chromodynamics simulation built on the MILC framework [3], and the instrumentation daemon of the Paradyn dynamic profiler [14]. In both applications, we found and removed significant call-path specific bottlenecks. Our modifications resulted in a 45% decrease in running time of su3_rmd, and an almost 98% decrease in Paradyn instrumentation time.

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 of profilers maintain a snapshot of the current call-path through the entire execution of the program, and update this snapshot at function call boundaries via instrumentation. The second category consists of profilers that use sampling, through a third-party debugging interface, to identify call-paths and gather performance data. The third category consists of profilers that approximate call-paths and provide only partial path information.

The first 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. This approach presents call-path profiling information about the entire program. However, the CCT cannot track recursive calls, collapsing all recursive calls to a particular function to a single node in the CCT. PP is 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 [12]. Their 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 generates a unique identifier for call-path *segments*. These segments do not include recursive calls or calls through function pointers; instead, these segments are used to reconstruct the complete call-path in a post-processing phase.

Larus also developed a method, Whole-Program Paths (WPP) [10], 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 [11] 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 generate a stack of currently executing functions. Since TAU tracks function entries and exits directly, dynamic calls and recursive calls are both represented in this stack. The TAU measurement code uses this stack of active functions as a representation of the current call-path. This technique shares many of the advantages of our approach, including partial data availability and a complete representation of the call path. However, their approach uses whole-program instrumentation, whereas we derive the current call-path from walking the program stack.

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. CATCH statically predicts call-paths within the selected subtrees and cannot handle call-paths through dynamic calls. Like CATCH, we only insert instrumentation in selected functions. However, CATCH instrumentation is inserted throughout a subtree of the call graph, whereas we insert instrumentation only in profiled functions.

The second category consists of call-path profilers that use a third-party debugging interface to track the program as it executes. Hall developed a profiler, CPPROF [8], that periodically pauses and samples the profiled application through the Solaris debug interface, `/proc`. Each sample taken by CPPROF includes the current application stack and elapsed CPU time since the previous sample. The elapsed CPU time is accumulated to the call-path represented by the stack. CPPROF allows the user to tune the frequency of sampling. More frequent samples generate larger overhead, as the process is paused during sampling, but contain smaller approximations in the resulting profile. The profile generated by CPPROF is available at run-time. Both CPPROF and iPath use a stack walk to determine the current call path. However, the third-party sampling approach used by CPPROF differs greatly from our first-party instrumentation approach. Our technique operates both within the address space of the program and operates only at function boundaries; as a result can handle cases that CPPROF's technique cannot. Finally, CPPROF cannot provide precise information about particular functions.

The tools gprof [7] and sprof [16] are both examples of the third 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 (the profiled function and its immediate caller) are approximated. This technique provides partial call-path profiling with low overhead, but may introduce errors based on the approximated data. Neither profiler supports the use of hardware performance metrics, limiting the variety of information available to the user.

3. Design

Incremental call-path profiling has five key characteristics. First, only the functions of interest to the user are instrumented. Second, profiling may be started and stopped at run-time, without relying on prior program modification. Third, the profiling data can be examined and analyzed while the program is running. Fourth, overhead is incurred only when profiled functions are executed. Fifth, the entire call-path is captured, including dynamic calls through function pointers. In this section, we describe our approach and how it provides these characteristics.

Our technique uses a technique called *dynamic instrumentation* [4,9] to gather both the call-path and performance metrics. Dynamic instrumentation operates by inserting new code into a program as it is running, without prior modification. This new instrumentation code may access program state, such as program registers and hardware performance counters. By using dynamic instrumentation, we are able to profile running programs without requiring prior binary modification. In addition, we can insert and remove instrumentation while the program is running.

We instrument the entry and exit points of each profiled function. This instrumentation performs three actions. First, it performs an inexpensive stack walk to determine the current call-path. Second, the instrumentation calculates the user-defined performance metric by sampling the desired performance counters. Third, we use the stack walk and performance data to update the current call-path profile of the function. In this section, we describe these three actions.

The core of our design is based around walking the stack to determine the current call-path. Since we are walking the actual process stack, we detect recursion (as multiple frames on the stack) and calls through function pointers, without requiring modification of any other location in the program. In addition, we can distinguish individual call-sites in a function, since their return addresses will be different on the stack. Stack walking is one of our fundamental mechanisms, so it is important that it be as efficient as possible. Conceptually, all that is required is to follow the frame pointer down the stack until the entry function's frame is reached. In practice, several program optimizations complicate walking the stack; we discuss these optimizations and our techniques for handling them in Section 4.3.

Once the current call-path has been obtained, our instrumentation calculates the desired performance metric. We distinguish two types of performance metrics, counters and timers. Counters, such as the number of times a function is executed on a particular call-path or the size of a function's input, 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 point 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. For timers, we record the value of the metric at the entry of the function and use it to calculate the difference when the function exits. Timers may calculate a variety of metrics, including memory stall time, synchronization blocking time, I/O wait time, and elapsed CPU usage.

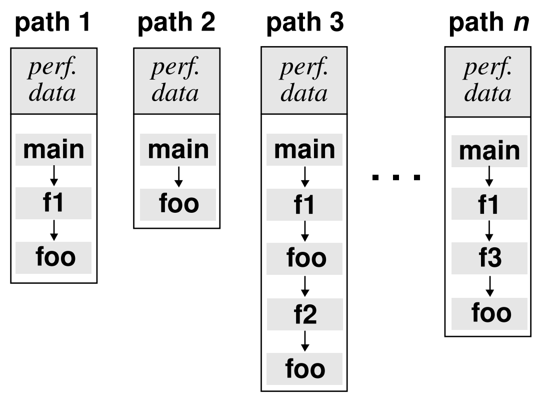

Figure 1: Representation of call-path profiles

The profile data is stored as a collection of call-path data structures. These structures associate performance data with a particular call-path, and have three requirements. First, it must be possible to add a new call-path on-the-fly to represent a newly discovered path (e.g., a new dynamic call). Second, finding the correct structure from a given call-path must be efficient to lower profiling overhead. Third, the structures must allow presentation of partial results while the program is running, without requiring post-processing. We represent call-paths simply as a vector of addresses where each entry is the corresponding call site address. Each vector is associated with the performance data collected for the particular call path, as shown in Figure 1. This performance data may be stored in the form of counters, timers, or a user-defined data structure.

Previous profilers have developed sophisticated techniques to represent call-paths, such as labelling each path with a unique integer identifier [13] or using a tree rooted at the entry function and annotated with profiling data [8]. The unique identifier technique allows constant time look-up speed, since the identifier can be used as an index into an array of performance information. However, this technique also requires static determination of all possible call paths. If dynamic calls are included, the number of possible paths can be unbounded. The tree technique represents call-paths as paths from the root of the tree to an edge node, and can compactly represent multiple call-paths with overlapping sections. The tree used by CPPROF is rooted at the entry of the program, and can describe arbitrary call-paths. We investigated the use of an inverted tree that was rooted at the profiled function. However, the improvement in look-up speed was minor for our application. This was due to two reasons. First, the time required for a stack walk and call-path look-up is small compared to the time required to calculate performance metrics such as synchronization time, and so optimizing the look-up did not result in a significant decrease in profiling overhead. Second, iPath uses heuristics to check the most likely call-paths first, and these heuristics significantly reduce our look-up time. We discuss these heuristics and our look-up techniques in Section 4.2.

One key characteristic of our method is that profiling data is available while the program runs. Since the profiling data is updated when a profiled function exits, it is possible for the user to examine the data at any time. This continuous update technique, combined with dynamic instrumentation, allows a user to continually refine which functions are profiled, and what metrics are gathered, based on preliminary results during a single run of the program. This capability is especially useful for long-running programs, as it allows a user to examine results of profiling without waiting for the program to complete.

4. Implementation

We implemented our call-path profiling method in a tool called iPath, which uses the DyninstAPI library to provide a dynamic instrumentation capability. iPath is implemented on both the IA-32/Linux and POWER/AIX platforms and is capable of using a variety of hardware and software performance metrics.

iPath consists of two parts: (1) a control process that instruments and monitors the profiled application and (2) a library containing our stack walk and profiling logic. The control process, called a *mutator* in Dyninst terminology, monitors the application and generates instrumentation requests. The library is injected into the application by the control process and used to instrument profiled functions. The gathered profile data is stored in a shared memory segment that is attached to both the control process and profiled application.

4.1. Control Process

iPath supports both starting the profiled application or attaching to a currently running application. Once we have created or attached to the application, we inject our run-time library into the program's address space. Since we inject this library at run-time, we do not require any previous program modification. Once the library has been injected, the control process triggers an initialization phase. This phase creates a shared memory segment between the application and control process where profile data is later stored. The control process can access this data at run-time without pausing the process.

Once initialization is complete, the control process inserts calls to our library at the entry and exit of all profiled functions. 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 may be caused by C++ function overloading or by 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 control process. Since a shared memory segment is used to store the profile data, it is possible to access preliminary results from the control process without pausing the program. The control process periodically prints out a summary of the profile data while the program runs and displays a final version when the program completes. iPath also calculates the percentage of the entire execution each call-path takes, which is useful to determine call-path specific bottlenecks. We provide an example of this output in the MILC results section (Figure 2).

4.2. Run-Time Library

The iPath run-time library is responsible for three things: determining the current call path, gathering appropriate performance information (e.g., hardware counter values), and calculating the call-path profile for each function. The run-time library consists of three major segments: the data structures that store profiling data, the stack walk logic, and the instrumentation executed at function entry and exit.

iPath stores detected call-paths and their associated profiling information as a collection of *call path entries* in the shared memory segment. Each call path entry consists of a vector of addresses of call sites in the call path and associated data for the desired performance metric. The format used for the call path is equivalent to the format returned by the stack walking code, so comparing a stack walk to a call path consists of first comparing the size of each vector, and then iteratively comparing the contents. The efficiency of this approach depends heavily on comparing the most likely call-path entry to the stack walk result first. We use two heuristics for guessing the most likely path, one for our entry instrumentation and one for exit instrumentation. The entry heuristic compares the most recently used call-path to the stack walk first. If a function is repeatedly called with the same call-path (e.g., from a loop in the caller), then the call path will not have changed. The exit heuristic matches the stack walk for function entries and exits through a mechanism we call the *active stack*.

The active stack takes advantage of the fact that the call-path rarely changes between the entry and exit of a function. If the call-path has not been changed (e.g., by a tail call optimization or a longjmp), the call-path determined by the entry instrumentation can be re-used by the exit instrumentation. We take advantage of this behavior by maintaining a stack of call-paths for use by our exit instrumentation. When a function is entered, the current call-path is pushed onto the stack. The top call-path on the stack is then checked first when the function is exited. 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 depth of one.

Our entry instrumentation performs three actions. First, we walk the stack and determine the current call-path as described above. It is possible that no call-path structure matches the stack walk; in this case, we create a new call-path structure. This technique allows us to avoid static analysis of program call paths. Second, we gather the desired performance metrics. Timers, such as memory stall time, are started by storing the current timer value as a starting value. Counters, such as function call counts, are simply incremented. Third, the current call-path is added to the active stack. Once the instrumentation completes, control is returned to the profiled application.

The exit instrumentation is responsible for stopping any active timers. If the user has not requested any timer-based metrics, exit instrumentation is unnecessary and none is inserted. First, as with entry instrumentation, we determine the active call-path with a stack walk and a table look-up. Unlike entry instrumentation, if no matching entry is found the instrumentation immediately exits. 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.

4.3. Stack Walk Optimizations

While a stack walk is conceptually simple, several optimizations can make the process more difficult. We identify three categories of optimizations that affect 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. A leaf function will never occur in the middle of a stack walk, as it cannot make calls. Since we walk the stack at function boundaries, we only have to handle optimizations in the callers of the profiled function. As a result, 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. This architecture uses two registers, the stack pointer and the frame pointer, to track the stack. The stack pointer moves throughout the execution of each function, 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 in the HPC domain, the MILC su3_rmd QCD simulation code and the Paradyn parallel performance instrumentation daemon. In both cases we were able to use call-path profile data to make substantial improvements in the running time of the programs. The gathered call-path profiles allowed us to distinguish particular call-paths correlated with poor profiled function performance. By focusing on these particular call-paths, we were able to make significant performance improvements that would have not been possible with a traditional profiler.

5.1. MILC

We used iPath to investigate su3_rmd, a distributed quantum chromodynamics simulation built on the MILC framework [3]. Our aim was to investigate synchronization bottlenecks within the program. We had previously investigated this program with gprof, a function-level profiler, and identified bottlenecks in two MPI functions, `MPI_Allreduce` and `MPI_Wait`. However, the function-level profiler provided insufficient information to determine the cause of these bottlenecks. Both MPI functions were called through MILC wrappers, which caused gprof to identify a single call-path to each MPI function. In addition, there were several calls to each MPI function from a variety of points in the program, and we were unable to determine which calls caused the poor performance. When we applied iPath to the application, we were able to identify the particular call-paths that were responsible for the bottlenecks. With this information, we were able to restructure the program to eliminate the `MPI_Wait` and reduce the `MPI_Allreduce` bottlenecks.

The MILC project provides a framework for performing QCD simulations. The framework defines a lattice of data and mechanisms for accessing individual points in 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 mechanism we are interested in is MPI-based and uses message passing to update lattice nodes. 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 [6]. 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: gather information about lattice points from neighboring nodes, perform vector operations on the lattice, and sum the results across all computing nodes.

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

Table 1: su3_rmd Running Time

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

We ran the su3_rmd simulation on four nodes of an IBM SP, using iPath to profile the `MPI_Wait` and `MPI_Allreduce` functions previously identified as bottlenecks. We examined the resulting call-path profile, focusing on the paths that passed through `ks_congrad`. The use of paths allowed us to unwind the communication abstraction used by the MILC framework. We discovered four specific synchronization bottlenecks in `ks_congrad`, two gather operations that made several calls to `MPI_Wait` and two indirect calls to `MPI_Allreduce` through the MILC framework. In total, the simulation executed for 3001 seconds, as shown in line 1 of Table 1.

We began by investigating the bottlenecks in `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. The blocking time was caused by 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`. Figure 2 visualizes data from iPath's output, showing a section of the complete `MPI_Wait` call-path profile. `MPI_Wait` is called twice in the `wait_gather` function, with the first call (the left branch) taking substantially longer to complete. Furthermore, the first call to the `wait_gather` function from `dslash_special` takes substantially longer to complete than the other three. For readability, we have collapsed the calls to `dslash_special` from `ks_congrad`, as they behaved in a similar fashion.

By using this call-path profile, we were able to trace the cause of this difference in blocking time 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 2a of Table 1.

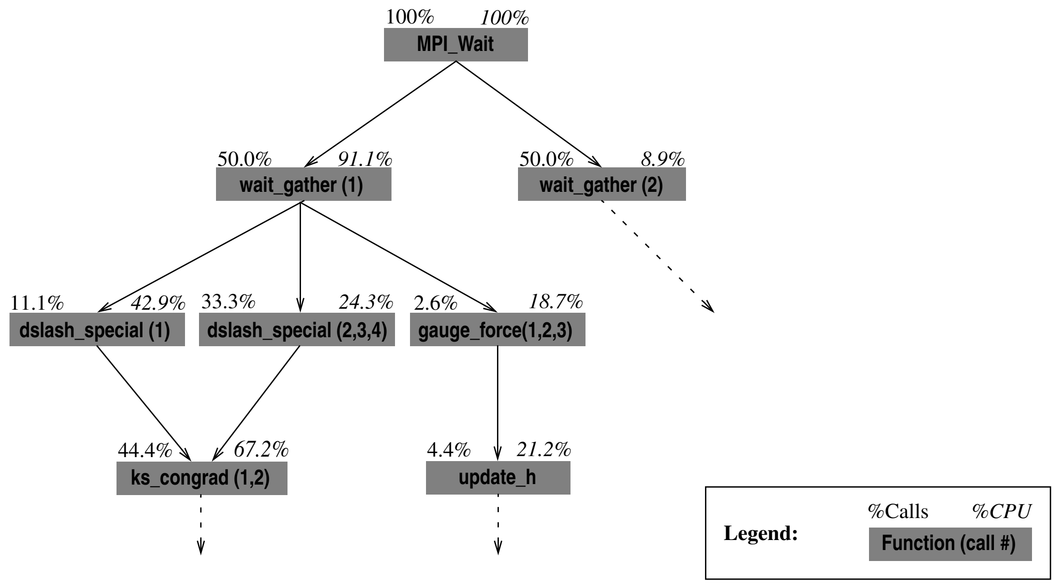

Figure 2: Call-path Profile of MPI_Wait

Partial display of 142 detected call-paths to MPI_Wait. Omitted paths contribute the remaining percentage of calls and synchronization blocking time, measured in CPU cycles.

We then investigated the `MPI_Allreduce` bottlenecks. This function is used by su3_rmd to sum a single floating point value across all nodes executing the simulation. Our call-path profile showed that this operation was not efficient, 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 overlapped with 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 2b of Table 1.

Finally, we ran the original simulation without profiling inserted to calculate the overhead due to our call-path profiling. The worst-case overhead, caused by profiling `MPI_Wait`, was 12% of the total running time of the program.

In summary, we used call-path analysis to discover four synchronization bottlenecks in the su3_rmd simulation. This call-path profiling information allowed us to focus on particular calls that were bottlenecks instead of the seeing only the aggregate behavior of the profiled function. Incremental use of paths allowed us to precisely target our profiling, which kept the overhead to a minimum. 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

	Unoptimized (seconds)	Optimized (seconds)	Change
Single-threaded	1.5	0.1	-93.3%
5 threads	48.5	1.2	-97.5%
10 threads	126.9	3.0	-97.6%
20 threads	296.1	6.4	-97.8%

Table 2: Instrumentation Time

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

reduced the running time of su3_rmd from 3001 seconds to 1652 seconds, a 45% decrease, as shown in line 3 of Table 1.

5.2. Paradyn Daemon

We also used iPath to identify and remove a major bottleneck in the Paradyn instrumentation daemon [14]. Paradyn is a profiling tool that is capable of performing an automated analysis of distributed programs. A frequently called utility function had previously been incorrectly optimized, leading to poor performance. Using iPath we were able to isolate the particular call-path to this utility function responsible for the bottleneck. We then used this call-path information to re-optimize the utility function, resulting in a significant reduction in instrumentation overhead. This reduction in instrumentation time also resulted in a visible user-level performance improvement.

We began by noting that Paradyn's automated analysis of a multi-threaded program was significantly slower than the analysis of a similar single-threaded program. This slowdown increased as the number of threads in the target program increased. We investigated this behavior and determined that the slowdown was due to the instrumentation section of the Paradyn daemon. Requests for instrumentation were taking significantly longer to satisfy for a multi-threaded program than they were for a single-threaded program, as shown in Table 2.

As with su3_rmd, we began by investigating this problem with a traditional profiler. We identified a utility function, `findFuncByAddr`, that consumed the majority of the extra time spent in instrumentation. This function performed a look-up mapping an input address to the function in the target application that contained the address. Comments in the source code indicated that this function was a known bottleneck and had previously been optimized. We investigated further and determined that the majority of callers of `findFuncByAddr` would take advantage of the optimization. However, this function was still performing poorly. Since `findFuncByAddr` had a large number of callers from many different places within the Paradyn daemon, we believed it would be a good target for call-path profiling.

We applied iPath to the problem function to determine if the call-path taken had a large effect on the performance of the function. Although we expected this was the case, we were surprised at the results. The performance of `findFuncByAddr` varied widely depending on the current call-path, from a few thousand CPU cycles per call to over a million cycles per call. Unsurprisingly, the calls that took advantage of the existing optimizations required only a short time to complete. The majority of calls to this utility function came from a single call-path that did not take advantage of any of the existing optimizations. This insight became obvious with the use of the call-path profiling data generated by iPath.

We then investigated why this particular function had such poor performance along a single call-path. We were able to characterize the inputs to the function and discover the root cause of the performance problem: the data structures used to maintain the address to function mapping. The mapping was stored in a hash table keyed on the entry address of the function. This method of storage made looking up a function based on the entry address extremely fast, but any other operation (such as a look-up based on an address within the function) caused a linear search through the hash table. The poorly performing calls, without exception, required this linear search.

We reimplemented `findFuncByAddr`, using a balanced tree instead of a hash table. This tree structure improved the performance of look-ups within the body of a function at the expense of slower entry point look-ups. In addition, we cached recent results instead of repeating look-ups. The results were impressive. When we re-ran our benchmark, instrumentation time was reduced to 1.2 seconds, a 98% performance improvement. We timed the old version against the new, with the results in Tables 2 and 3.

We were also interested in determining the overhead imposed by the use of iPath to gather call-path profiles. We were unable to detect any changes in the running time of the Paradyn daemon due to iPath. This is due to the behavior of the function we were profiling. The execution time per function invocation was large enough that the overhead imposed by iPath was completely hidden.

In summary, we were able to use iPath to localize a severe performance problem. We had previously used a funciton-level profiler and discovered that a utility function, `findFuncByAddr`, was performing poorly. However, both the profile data and our investigation of the code failed to localize the reason for the performance problem. Using iPath,

	Unoptimized (min:sec)	Optimized (min:sec)	Change
Single-threaded	4:30	3:50	-14.8%
5 threads	4:30	3:55	-13.0%
10 threads	5:45	4:10	-27.5%
20 threads	8:00	4:25	-46.9%

Table 3: 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.

we were able to determine a particular call-path that was the cause of the bottleneck. Finally, iPath allowed us to profile only the function that was the bottleneck and not all functions in the Paradyn daemon. Selectivity allowed us to efficiently profile `findFuncByAddr` without excessive overhead.

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. This tool allows a user to target call-path profiling to particular functions, and can take advantage of both hardware and software performance metrics. Our current implementation runs on both IA-32/Linux and POWER/AIX, and can analyze distributed programs as well as sequential programs.

We used iPath to isolate and correct bottlenecks in two programs. iPath located and characterized synchronization bottlenecks in the MILC su3_rmd simulation. Through the use of call-path profile data, we were able to identify particular calls to MPI functions that caused poor performance. Since iPath collects a complete call-path, we were able to unwind through the MILC abstraction layer. We replaced several blocking and synchronous MPI calls with asynchronous equivalents, and reordered the calls to hide the message passing latency. These changes resulted in a 45% decrease in running time of the su3_rmd simulation. We believe call-path profiling can be used to reduce synchronization blocking time in a wide range of distributed programs make use of library functions for communication by identifying the particular call chains that are bottlenecks.

iPath also identified a bottleneck in a sequential program, the instrumentation daemon component of the distributed Paradyn parallel performance tool. We discovered and repaired a single utility function that had been incorrectly optimized based on the results of function-level profiling and examination of the source code. The majority of callers to the function presented similar behavior, and the programmer had optimized the utility function based on that assumption. Through the use of call-path profiling, we determined that a single call was overwhelmingly responsible for the time spent in the utility function, and that this call did not take advantage of the previous optimizations. By reoptimizing the function, we were able to reduce Paradyn instrumentation time by 98%. By using call-path profiling, we were able to accurately characterize the behavior of the utility function based on run-time information rather than analysis of the source code. This situation is an example of how call-path profiling can be used to improve programs that make use of utility functions, especially when source-level analysis is misleading or impossible.

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. Gottlieb, W. Liu, D. Toussaint, R. L. Renken, and R. L. Sugar, "Hybrid-molecular-dynamics Algorithms for the Numerical Simulation of Quantum Chromodynamics", *Physical Review D*, **35**, 2531-2542 (1987).

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

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

[9] J. K. Hollingsworth, B. P. Miller, and J. Cargille, "Dynamic Program Instrumentation for Scalable Performance Tools", *1994 Scalable High-Performance Computing Conf.*, Knoxville, Tenn., 1994.

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

[11] 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.

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

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

[14] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam and T. Newhall. "The Paradyn Parallel Performance Measurement Tools", *IEEE Computer* **28**, 11, November 1995, pp. 37-46.

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

[16] SPROF, Linux utility.