CATCH – A Call-Graph Based Automatic Tool for Capture of Hardware Performance Metrics for MPI and OpenMP Applications

Luiz DeRose and Felix Wolf*

Advanced Computing Technology Center
IBM Research
Yorktown Heights, NY 10598 USA
laderose@us.ibm.com

Research Centre Juelich
ZAM
Juelich, Germany
f.wolf@fz-juelich.de

Abstract. CATCH is a profiler for parallel applications that collects hardware performance counters information for each function called in the program, based on the path that led to the function invocation. It automatically instruments the binary of the target application independently of the programming language. It supports MPI, OpenMP, and hybrid applications and integrates the performance data collected for different processes and threads. Functions representing the bodies of OpenMP constructs are also monitored and mapped back to the source code. Performance data is generated in XML for visualization with a graphical user interface that displays the data simultaneously with the source code sections they refer to.

1 Introduction

Developing applications that achieve high performance on current parallel and distributed systems requires multiple iterations of performance analysis and program refinements. Traditional performance tools, such as SvPablo [7], TAU [11], Medea [3], and AIMS [14], rely on experimental performance analysis, where the application is instrumented for data capture, and the collected data is analyzed after the program execution. In each cycle developers instrument application and system software, in order to identify the key program components responsible for the bulk of the program's execution time. Then, they analyze the captured performance data and modify the program to improve its performance. This optimization model requires developers and performance analysts to engage in a laborious cycle of instrumentation, program execution, and code modification, which can be very frustrating, particularly when the number of possible optimization points is large. In addition, static instrumentation can inhibit compiler optimizations, and when inserted manually, could require an unreasonable amount of the developer's time.

* This work was performed while Felix Wolf was visiting the Advanced Computing Technology Center at IBM Research.

Moreover, most users do not have the time or desire to learn how to use complex tools. Therefore, a performance analysis tool should be able to provide the data and insights needed to tune and optimize applications with a simple to use interface, which does not create additional burden to the developers. For example, a simple tool like the GNU gprof [9] can provide information on how much time a serial program spent in which function. This "flat profile" is refined with a call-graph profiler, which tells the time separately for each caller and also the fraction of the execution time that was spent in each of the callees. This call-graph information is very valuable, because it not only indicates the functions that consume most of the executions time, but also identifies in which context it happened. However, a high execution time does not necessarily indicate inefficient behavior, since even an efficient computation can take a long time. Moreover, as computer architectures become more complex, with clustered symmetric multiprocessors (SMPs), deep-memory hierarchies managed by distributed cache coherence protocols, and speculative execution, application developers face new and more complicate performance tuning and optimization problems. In order to understand the execution behavior of application code in such complex environments, users need performance tools that are able to support the main parallel programming paradigms, as well as, access hardware performance counters and map the resulting data to the parallel source code constructs. However, the most common instrumentation approach that provides access to hardware performance counters also augments source code with calls to specific instrumentation libraries (e.g., PAPI [1], PCL [13], SvPablo [7] and the HPM Toolkit [5]). This static instrumentation approach lacks flexibility, since it requires re-instrumentation and recompilation, whenever a new set of instrumentation is required.

In this paper we present CATCH (Call-graph-based Automatic Tool for Capture of Hardware-performance-metrics), a profiler for MPI and OpenMP applications that provides hardware performance counters information related to each path used to reach a node in the application's call graph. CATCH automatically instruments the binary of the target application, allowing it to track the current call-graph node at run time with only constant overhead, independently of the actual call-graph size. The advantage of this approach lies in its ability to map a variety of expressive performance metrics provided by hardware counters not only to the source code but also to the execution context represented by the complete call path. In addition, since it relies only on the binary, CATCH is programming language independent.

CATCH is built on top of DPCL [6], an object-based C++ class library and run-time infrastructure, developed by IBM, which is based on the Paradyn [10] dynamic instrumentation technology, from the University of Wisconsin. DPCL flexibly supports the generation of arbitrary instrumentation, without requiring access to the source code. We refer to [6] for a more detailed description of DPCL and its functionality. CATCH profiles the execution of MPI, OpenMP, and hybrid application and integrates the performance data collected for different processes and threads. In addition, based on the information provided by the native AIX

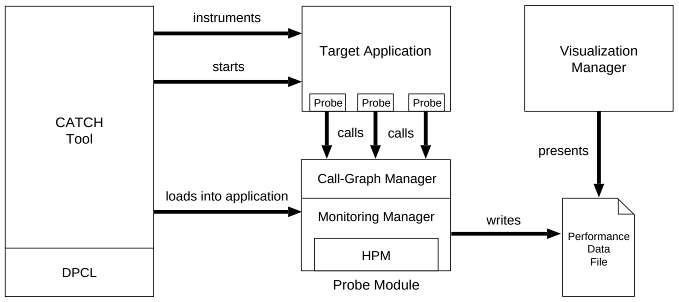

Fig. 1. Overall architecture of CATCH.

compiler, CATCH is able to identify the functions the compiler generates from OpenMP constructs and to link performance data collected from these constructs back to the source code. To demonstrate the portability of our approach, we additionally implemented a Linux version, which is built on top of Dyninst [2].

The remainder of this article is organized as follows: Section 2 contains a description of the different components of CATCH and how they are related to each other. Section 3 presents a detailed explanation of how CATCH tracks a call-graph node at run time. Section 4 discusses related work. Finally, Section 5 presents our conclusions and plans for future work.

2 Overall Architecture

As illustrated in Figure 1, CATCH is composed of the CATCH *tool*, which instruments the target application and controls its execution, and the CATCH *probe module*, which is loaded into the target application by CATCH to perform the actual profiling task. The probe module itself consists of the *call-graph manager* and the *monitoring manager*. The former is responsible for calculating the current call-graph position, while the latter is responsible for monitoring the hardware performance counters. After the target application finishes its execution, the monitoring manager writes the collected data into an XML file, whose contents can be displayed using the *visualization manager*, a component of the HPM Toolkit, presented in [5].

When CATCH is invoked, it first creates one or more processes of the target application in suspended state. Next, it computes the static call graph and performs the necessary instrumentation by inserting calls to probe-module functions into the memory image of the target application. Finally, CATCH writes the call graph into a temporary file and starts the target application.

Before entering the main function, the instrumented target application first initializes the probe-module, which reads in the call-graph file and builds up the probe module's internal data structures. Then, the target application resumes execution and calls the probe module upon every function call and return. The following sections present a more detailed insight into the two components of the probe module.

2.1 The Call-Graph Manager

The probes inserted into the target application call the call-graph manager, which computes the current node of the call graph and notifies the monitoring manager of the occurrence of the following events:

Initialization The application will start. The call graph and the number of threads are provided as parameters. The call graph contains all necessary source-code information on modules, functions, and function-call sites.
Termination The application terminated.
Function Call The application will execute a function call. The current call-graph node and the thread identifier are provided as parameters.
Function Return The application returned from a function call. The current call-graph node and the thread identifier are provided as parameters.
OpenMP Fork The application will fork into multi-threaded execution.
OpenMP Join Multi-threaded execution finished.
MPI Init MPI will be initialized. The number of MPI processes and the process identifier are provided as parameters. When receiving this event, the monitoring manager knows that it can execute MPI statements. This event is useful, for example, to synchronize clocks for event tracing.
MPI Finalize MPI will be finalized. It denotes the last point in time, where the monitoring manager is able to execute an MPI statement, for example, to collect the data gathered by different MPI processes.

Note that the parameterized events listed above define a very general profiling interface, which is not limited to profiling, but is also suitable for a multitude of alternative performance-analysis tasks (e.g., event tracing). The method of tracking the current node in the call graph is described in Section 3.

2.2 The Monitoring Manager

The monitoring manager is an extension of the HPM data collection system, presented in [5]. The manager uses the probes described above to activate the HPM library. Each node in the call graph corresponds to an application section that could be instrumented. During the execution of the program, the HPM library accumulates the performance information for each node, using tables with unique identifiers for fast access to the data structure that stores the information during run time. Thus, the unique identification of each node in the call graph, as described in Section 3, is crucial for the low overhead of the data collection system. The HPM library supports nested instrumentation and multiple calls to any node. When the program execution terminates, the HPM library reads and traverses the call graph to compute exclusive counts and durations for each node. In addition, it computes a rich set of derived metrics, such as cache hit ratios and MFLOP/sec rates, that can be used by performance analysts to correlate the behavior of the application to one or more of the hardware components. Finally, it generates a set of *performance files*, one for each parallel task.

3 Call-graph Based Profiling with Constant Overhead

In this section we describe CATCH's way of instrumenting an application, which provides the ability to calculate the current node in the call graph at run time by introducing only constant overhead independently of the actual call-graph size. Our goal is to be able to collect statistics for each function called in the program, based on the path that led to the function invocation. For simplicity, we first discuss serial non-recursive applications and later explain how we treat recursive and parallel ones.

3.1 Building a Static Call Graph

The basic idea behind our approach is to compute a static call graph of the target application in advance before executing it. This is accomplished by traversing the code structure using DPCL. We start from the notion that an application can be represented by a multigraph with functions represented as nodes and call sites represented as edges. If, for example, a function f calls function g from k different call sites, the correspondent transitions are represented with k arcs from node f to node g in the multigraph. A sequence of edges in the multigraph corresponds to a path.

The multigraph of non-recursive programs is acyclic. From the application's acyclic multigraph, we build a static call tree, which is a variation of the call graph, where each node is a simple path that start at the root of the multigraph. For a path $\pi = \sigma e$, where σ is a path and e is an edge in the multigraph, σ is the parent of π in the tree. We consider the root of the multigraph to be the function that calls the application's main function. This *start* function is assumed to have an empty path to itself, which is the root of the call tree.

3.2 Instrumenting the Application

The probe module holds a reference to the call tree, where each node contains an array of all of its children. Since the call sites within a function can be enumerated and the children of a node correspond to the call sites within the function that can be reached by following the path represented by that node, we arrange the children in a way that child i corresponds to call site i. Thus, child i of node n in the tree can be accessed directly by looking up the ith element of the array in node n.

In addition, the probe module maintains a pointer to the current node n_c, which is moved to the next node n_n upon every function call and return. For a function call made from a call site i, we assign:

$$n_n := child_i(n_c)$$

That is, the ith call site of the function currently being executed causes the application to enter the ith child node of the current node. For this reason, the probe module provides a function `call(int i)`, which causes the pointer to the current node to be moved to child i. In case of a function return, we assign:

$$n_n := parent(n_c)$$

That is, every return just causes the application to re-enter the parent node of the current node, which can be reached via a reference maintained by CATCH. For this reason, the probe module provides a function `return()`, which causes the pointer to the current node to be moved to its parent.

Since DPCL provides the ability to insert calls to functions of the probe module before and after a function-call site and to provide arguments to these calls, we only need for each function f to insert `call(i)` before a function call at call site i and to insert `return()` after it.

Because `call(int i)` needs only to look up the ith element of the children array, and `return()` needs only to follow the reference to the parent, calling these two functions introduces only constant execution-time overhead independently of the application's call-tree size.

3.3 Recursive Applications

Trying to build a call tree for recursive applications would result in a tree of infinite size. Hence, to be able to support recursive applications, CATCH builds a call graph that may contain loops instead. Every node in this call graph can be described by a path π that contains not more than one edge representing the same call site.

Suppose we have a path $\pi = \sigma d \rho d$ that contains two edges representing the same call site, which is typical for recursive applications. CATCH builds up its graph structure in a way, such that $\sigma d = \sigma d \rho d$, that is, both paths are considered to be the same node. That means, we now have a node that can be reached using different paths. Note that each path has still a unique parent, which can be obtained by collapsing potential loops in the path.

However, in case of loops in the call graph we can no longer assume that a node was entered from its parent. Instead, CATCH pushes every new node it enters upon a function call onto a stack and retrieves it from there upon a function return:

$$\begin{array}{ll} push(n_n) & \text{(call)} \\ n_n := pop() & \text{(return)} \end{array}$$

Since the stack operations again introduce not more than constant overhead in execution time, the costs are still independent of the call-graph size.

3.4 Parallel Applications

OpenMP*:* OpenMP applications follow a fork-join model. They start as a single thread, fork into a team of multiple threads at some point, and join together after the parallel execution has been finished. CATCH maintains for each thread a separate stack and a separate pointer to the current node, since each thread may call different functions at different points in time. When forking, each slave thread inherits the current node of the master.

The application developer marks code regions that should be executed in parallel by enclosing them with compiler directives or pragmas. The native AIX compiler creates functions for each of these regions. These functions are indirectly called by another function of the OpenMP run-time library (i.e., by passing a pointer to this function as an argument to the library function). Unfortunately, DPCL is not able to identify indirect call sites, so we cannot build the entire call graph only relying on the information provided by DPCL. However, the scheme applied by the native AIX compiler to name the functions representing OpenMP constructs enables CATCH to locate these indirect call sites and to build the complete call graph in spite of their indirect nature.

MPI*:* CATCH maintains for each MPI process a separate call graph, which is stored in a separate instance of the probe module. Since there is no interference between these call graphs, there is nothing extra that we need to pay specific attention to.

3.5 Profiling Subsets of the Call-Graph

If the user is only interested in analyzing a subset of the application, it would be reasonable to restrict instrumentation to the corresponding part of the program in order to minimize intrusion and the number of instrumentation points. Hence, CATCH offers two complementary mechanisms to identify an interesting subset of the call graph. The first one allows users to identify subtrees of interest, while the second is used to filter out subtrees that are not of interest.

- *Selecting* allows the user to select subtrees associated with the execution of certain functions and profile these functions only. The user supplies a list of functions as an argument, which results in profiling being switched off as soon as a subtree of the call graph is entered that neither contains call sites to one of the functions in the list nor has been called from one of the functions in the list.
- *Filtering* allows the user to exclude subtrees associated with the execution of certain functions from profiling. The user specifies these subtrees by supplying a list of functions as an argument, which results in profiling being switched off as soon as one of the function in the list is called.

Both mechanisms have in common that they require switching off profiling when entering and switching it on again when leaving certain subtrees of the call graph. Since the number of call sites that can be instrumented by DPCL may be limited, CATCH recognizes when a call no longer needs to be instrumented due to a subtree being switched off and does not insert any probes there. By default, CATCH instruments only function-call sites to user functions and OpenMP and MPI library functions.

3.6 Limitations

The main limitations of CATCH result from the limitations of the underlying instrumentation libraries. Since DPCL identifies a function called from a function-call site only by name, CATCH is not able to cope with applications defining a function name twice for different functions. In addition, the Linux version, which is based on Dyninst, does not support MPI or OpenMP applications. Support for parallel applications on Linux will be available when the DPCL port to Linux is completed.

CATCH is not able to statically identify indirect calls made via a function pointer passed at run-time. Hence, CATCH cannot profile applications making use of those calls, which limits its usability in particular for C++ applications. However, CATCH still provides full support for the indirect calls made by the OpenMP run-time system of the native AIX compiler as described in Section 3.4.

4 Related Work

The most common instrumentation approach augments source code with calls to specific instrumentation libraries. Examples of these static instrumentation systems include the Pablo performance environment toolkit [12] and the Automated Instrumentation Monitoring System (AIMS) [14]. The main drawbacks of static instrumentation systems are the possible inhibition of compiler optimization and the lack of flexibility, since it requires application re-instrumentation, recompilation, and a new execution, whenever new instrumentation is needed. CATCH, on the other hand, is based on binary instrumentation, which does not require recompilation of programs and does not affect optimization. Binary instrumentation can be considered as a subset of the dynamic instrumentation technology, which uses binary instrumentation to install and remove probes during execution, allowing users to interactively change instrumentation points during run time, focusing measurements on code regions where performance problems have been detected. Paradyn [10] is the exemplar of such dynamic instrumentation systems. Since Paradyn uses probes for code instrumentation, any probe built for CATCH could be easily ported to Paradyn. However, the main contributions of CATCH, which are not yet provided in Paradyn, are the OpenMP support, the precise distinction between different call paths leading to the same program location when assessing performance behavior, the flexibility of allowing users to select different sets of performance counters, and the presentation of a rich set of derived metrics for program analysis.

OMPTrace [4] is a DPCL based tool that combines traditional tracing with binary instrumentation and access to hardware performance counters for the performance analysis and optimization of OpenMP applications. Performance data collected with OMPTrace is used as input to the Paraver visualization tool [8] for detailed analysis of the parallel behavior of the application. Both OMPTrace and CATCH use a similar approach to exploit the information provided by the native AIX compiler to identify and instrument functions the compiler generates from OpenMP constructs. However, OMPTrace and CATCH differ completely in their data collection techniques, since the former collects traces, while CATCH is a profiler.

GNU gprof [9] creates execution-time profiles for serial applications. In contrast to our approach, gprof uses sampling to determine the time fraction spent in different functions of the program. Besides plain execution times gprof estimates the execution time of a function when called from a distinct caller only. However, since the estimation is based on the number of calls from this caller it can introduce significant inaccuracies in cases where the execution time highly depends on the caller. In contrast, CATCH creates a profile for the full call graph based on measurement instead of estimation.

Finally, PAPI [1] and PCL [13] are application programming interfaces that provide a common set of interfaces to access hardware performance counters across different platforms. Their main contribution is in providing a portable interface. However, as opposed to CATCH, they still require static instrumentation and do not provide a visualization tool for presentation.

5 Conclusion

CATCH is a profiler for parallel applications that collects hardware performance counters information for each function called in the program, based on the path that led to the function invocation. It supports MPI, OpenMP, and hybrid applications and integrates the performance data collected for different processes and threads. Functions representing the bodies of OpenMP constructs, which have been generated by the compiler, are also monitored and mapped back to the source code. The user can view the data using a GUI that displays the performance data simultaneously with the source code sections they refer to.

The information provided by hardware performance counters provide more expressive performance metrics than mere execution times and thus enable more precise statements about the performance behavior of the applications being investigated. In conjunction with CATCH's ability not only to map these data back to the source code but also to the full call path, CATCH provides valuable assistance in locating hidden performance problems in both the source code and the control flow. Since CATCH works on the unmodified binary, its usage is very easy and independent of the programming language.

In the future, we plan to use the very general design of CATCH's profiling interface to develop a performance-controlled event tracing system that tries to identify interesting subtrees at run time using profiling techniques and to record the performance behavior at those places using event tracing, because tracing allows a more detailed insight into the performance behavior. Since now individual event records can carry the corresponding call-graph node in one of their data fields, they are aware of the execution state of the program even when event tracing starts in the middle of the program. Thus, we are still able to map the observed performance behavior to the full call path. The benefit of selective tracing would be a reduced trace-file size and less program perturbation by trace-record generation and storage in the main memory.

References

1. S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A Scalable Cross-Platform Infrastructure for Application Performance Tuning Using Hardware Counters. In *Proceedings of Supercomputing'00*, November 2000.
2. B. R. Buck and J. K. Hollingsworth. An API for Runtime Code Patching. *Journal of High Performance Computing Applications*, 14(4):317–329, Winter 2000.
3. Maria Calzarossa, Luisa Massari, Alessandro Merlo, Mario Pantano, and Daniele Tessera. Medea: A Tool for Workload Characterization of Parallel Systems. *IEEE Parallel and Distributed Technology*, 3(4):72–80, November 1995.
4. Jordi Caubet, Judit Gimenez Jesus Labarta, Luiz DeRose, and Jeffrey Vetter. A Dynamic Tracing Mechanism for Performance Analysis of OpenMP Applications. In *Proceedings of the Workshop on OpenMP Applications and Tools - WOMPAT 2001*, pages 53 – 67, July 2001.
5. Luiz DeRose. The Hardware Performance Monitor Toolkit. In *Proceedings of Euro-Par*, pages 122–131, August 2001.
6. Luiz DeRose, Ted Hoover Jr., and Jeffrey K. Hollingsworth. The Dynamic Probe Class Library - An Infrastructure for Developing Instrumentation for Performance Tools. In *Proceedings of the International Parallel and Distributed Processing Symposium*, April 2001.
7. Luiz DeRose and Daniel Reed. SvPablo: A Multi-Language Architecture-Independent Performance Analysis System. In *Proceedings of the International Conference on Parallel Processing*, pages 311–318, August 1999.
8. European Center for Parallelism of Barcelona (CEPBA). *Paraver - Parallel Program Visualization and Analysis Tool - Reference Manual*, November 2000. http://www.cepba.upc.es/paraver.
9. J. Fenlason and R. Stallman. *GNU prof - The GNU Profiler*. Free Software Foundation, Inc., 1997. http://www.gnu.org/manual/gprof-2.9.1/gprof.html.
10. Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The Paradyn Parallel Performance Measurement Tools. *IEEE Computer*, 28(11):37–46, November 1995.
11. Bernd Mohr, Allen Malony, and Janice Cuny. TAU Tuning and Analysis Utilities for Portable Parallel Programming. In G. Wilson, editor, *Parallel Programming using C++*. M.I.T. Press, 1996.
12. Daniel A. Reed, Ruth A. Aydt, Roger J. Noe, Phillip C. Roth, Keith A. Shields, Bradley Schwartz, and Luis F. Tavera. Scalable Performance Analysis: The Pablo Performance Analysis Environment. In Anthony Skjellum, editor, *Proceedings of the Scalable Parallel Libraries Conference*. IEEE Computer Society, 1993.
13. Research Centre Juelich GmbH. *PCL - The Performance Counter Library: A Common Interface to Access Hardware Performance Counters on Microprocessors.*
14. J. C. Yan, S. R. Sarukkai, and P. Mehra. Performance Measurement, Visualization and Modeling of Parallel and Distributed Programs Using the AIMS Toolkit. *Software Practice & Experience*, 25(4):429–461, April 1995.