Argument Controlled Profiling

Tilman Küstner, Josef Weidendorfer, and Tobias Weinzierl

Technische Universität München, Germany
{kuestner,weidendo,weinzier}@in.tum.de

Abstract. Profiling tools relate measurements to code context such as function names in order to guide code optimization. For a more detailed analysis, call path or phase-based profiling enhances the context by call chains or user defined phase names, respectively. In this paper, we propose *argument controlled profiling* as a new type of context extension using the value of function arguments as part of the context. For a showcase simulation code, we demonstrate that this simplifies and enriches the understanding and analysis of code—in particular recursive functions. Due to the new profiling technique, we found optimizations resulting in more than 16% runtime improvement. Argument controlled profiling is implemented as extension of Callgrind, a simulation-based profiling tool using runtime instrumentation.

1 Introduction

As program codes today typically consist of millions of lines of code, as computer architectures are diverse and sophisticated, and as there is a high pressure on the developer to deliver the whole software on time, writing fast code out of the box is a complex, demanding, almost impossible challenge. On the one hand, it is hard to predict in which code parts most of the runtime is spent. On the other hand, these code parts have to be tailored to the actual hardware. Profiling is the tool of choice to identify the code parts and to get hints on how to do an optimization.

However, use of direct or indirect recursion can make the results of a profiler difficult to understand. To provide useful data such as inclusive cost, functions which are part of recursive cycles need to be suppressed by introducing artificial cycle functions comprising all functions of this recursive cycle. They are identified by strongly connected components in the dynamic call graph [6]. Unfortunately, this discards a lot of information and, sometimes, renders results useless for analysis. The solution is to relate event counts measured by the profiling tool not only to the function name, but to a more elaborated context which better describes the code position. A similar argument holds for non-recursive functions whose runtime behavior depends significantly on their arguments.

One typical extension is the *calling context* [1, 15] which takes the call chain to the current function into account. Unfortunately, this does not help when the behavior of a recursive function does not depend on the call chain leading to it, but on values in data structures instead. This is often the case, and conceals

H.X. Lin et al. (Eds): Euro-Par 2009 Workshops, LNCS 6043, pp. 177–184, 2010.
© Springer-Verlag Berlin Heidelberg 2010

important information. For example, in a code where a recursive function is visiting nodes of a tree or graph structure in some order, it is valuable to to see how often one node type follows after another node type, as this helps optimizing often executed code paths.

Thus, it is useful to enhance the function context by incorporating a program state. To distinguish a function with respect to a state, it has to be specified by the user before entering the function. A natural choice is to use the value of function arguments. In the output of the profiling tool, the same function with different states then is treated as individual symbols. Thus, cost is also assigned to different symbols. This is the main idea presented in this paper: the *argument controlled profiling*.

We apply this technique to our cache simulation tool Callgrind [13], which uses the runtime instrumentation framework Valgrind [9]. The simulation approach using runtime instrumentation has two specific advantages: first, any overhead generated by collecting separate metrics for an increased number of symbols does not matter, as runtime overhead does not influence results of the simulator. Second, it is important to selectively use this feature as the number of symbols can explode. The runtime instrumentation approach allows to do multiple profiling runs with different selective configurations using the same binary without recompilation. The latter is important if long recompilation times required for different instrumentation would be prohibitive for an in-depth analysis.

The remainder is organized as follows: First, we discuss related work. Second, the usage of argument controlled profiling is described with a small example. Third, we provide details on our prototype implementation. A practical showcase is given afterwards, together with the applied optimization approach and results on runtime improvements. A short conclusion with an outlook closes the discussion.

2 Related Work

The simplest useful result of a profiling tool is the *flat profile*. All performance cost such as time (clock ticks) or cache events is related to code which triggered the cost. However, entries in a flat profile may point to code parts which reside in a library unavailable for modifications, or they carry no potential for optimization in themselves. Here, tracking back the call stack for optimizable code parts, i.e. lines which invoke the expensive functions, is valuable. For this, the context is enriched by the call path: it becomes a calling context. A lot of papers in the area of profiling tools discuss the efficient collection of full call paths (starting at the entry point of the program), using precise yet low-overhead measurement strategies [1, 11, 5, 15, 3]. Many profiling tools provide the calling context on request [7, 10, 13, 12, 4].

For tools performing real-time measurements, there is always a tradeoff between accuracy and completeness because of the influence of measurement

overhead. A good compromise is to allow selective instrumentation. The developer helps minimizing overhead by providing hints on what type of information she is looking for before the measurement starts. The TAU Performance System [10] implements so-called *phase-based profiling* [8]. The developer can indicate logical phases of her program by calling a profiling interface (API). These phase names, arbitrary strings, can be constructed to contain function arguments. In contrast to our approach, this technique requires the user to manually modify the source code and recompile it. Another selective instrumentation method is *incremental call path profiling* [2], where specified functions are dynamically instrumented to do full stack walks for determining the calling context.

Sampling is a strategy not based on instrumentation. It allows the overhead to be tunable, but provides statistical results. Only every n-th occurrence of an event of interest is collected. To get the calling context at sample points, stack walks at arbitrary points of execution need to be supported. Some tools such as Intel's PTU [4] rely on debug information generated by the compiler. However, not all compilers produce reliable debug information. By contrast, hpcrun [12] uses binary analysis to generate meta information needed for stack walks.

A lot of the tools mentioned above preserve recursive invocations in the calling context. However, if it is known that the behavior of a recursive function does not depend on recursion depth, an overwhelming amount of unneeded information is generated.

3 Usage and Example

Argument controlled contexts allow to distinguish profiling results of the same function according to function arguments. Yet, this can not be done for every argument-value combination of every function, as this would lead to an explosion of profiling results. Instead, the programmer needs to specify that she wants a distinction for a given function according to selected values of arguments.

In our prototype extension of Callgrind, we support the command line option `--separate-par=`*function*`:`*num*[`:`b_1[`,`b_2...]]. Here, $function$ is the name of the function for which we request the creation of an enhanced context: the *num*-th 4-byte-value on the stack, interpreted as 4-byte integer value (which also works for booleans), is to be incorporated into the context name. Optionally, b_i values can be given, representing bucket borders. E.g. for value x and borders $(5, 10)$, the ranges $x <= 5$, $5 < x <= 10$, and $10 < x$ are distinguished.

We consider the following 2-dimensional Fibonacci-like function

```
int fib2(int i, int j) {
  if ((i<2) || (j<2)) return 1;
  return fib2(i, j-1) + fib2(i-1, j) + fib2(i-1, j-1);
}
```

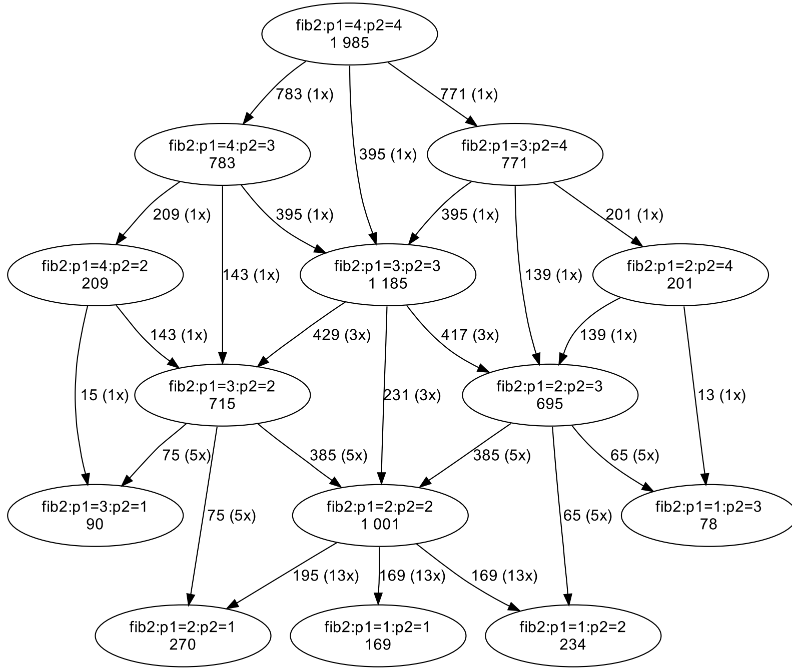

Fig. 1. The call graph of `fib2(4,4)`

together with the `main` function calling `fib2(4,4)`. Executing

```
> valgrind --tool=callgrind \
         --separate-par=fib2:1 --separate-par=fib2:2 ./fib2
```

produces among others profiling data visualized in Fig. 1, if we select the symbol "`fib2:p1=4:p2=4`" to be displayed. Each node represents a profile for function `fib2`, but is distinguished by the requested argument values. Edges between nodes represent calls with the call count shown in parenthesis. The number given in the nodes and next to edges is the inclusive cost for the event "Instructions Executed". This is the event collected by Callgrind when cache simulation is switched off.

Fig. 1 is generated by the "Export Graph" functionality of our profile visualization tool KCachegrind [13]. The screenshot in Fig. 2 shows on the left a list of functions (with `fib2:p1=4:p2=4` selected). The functions are grouped according to the "ELF object" they reside in. These groups are shown in the list above the functions. The selected group "fib2" is the main executable, i.e. the function list only shows functions residing in the binary "fib2". On the right, the interactive call graph visualization for the selected function is displayed. This is just one of many visualization types supported by KCachegrind, such as caller/callee lists, *tree map* visualizations of the callee tree, or source/assembly annotations.

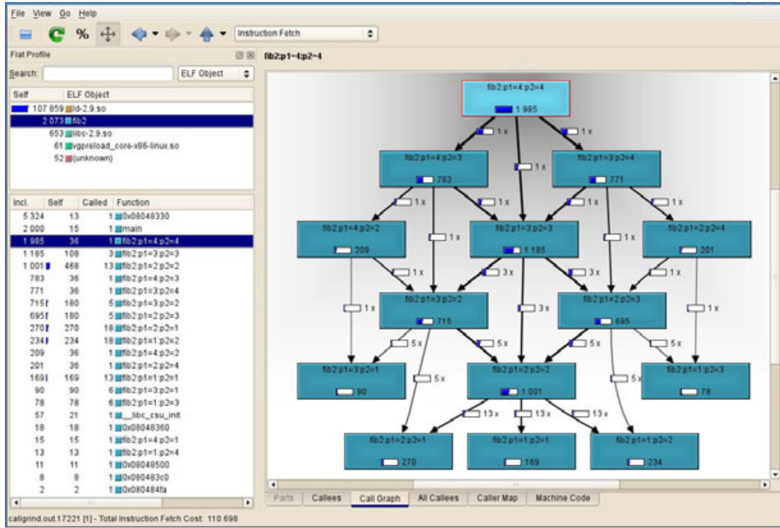

Fig. 2. Screenshot of KCachegrind with call graph view

4 Implementation

For Callgrind, it is important to be able to relate the effect of every memory access to a context (e.g. function name, calling context, thread number, and so forth). Thus, it is helpful to have the context readily available whenever an event is triggered. The calling context is updated every time a function is entered or left. This is done by code inserted by runtime instrumentation. The first time a function (more precisely, a *basic block*) is executed, Valgrind forwards the code to Callgrind which inserts instrumentation as needed. The instrumented code then is put into a *translation cache* and executed instead of the original. At instrumentation time (i.e. only once per *basic block*), the function name is related to the code by a lookup in the debug information. At execution time, inserted code can easily detect when the currently active function changes. Whether this change was triggered by a call or return event is heuristically determined with the help of the instruction opcode, the value of the stack pointer, and a shadow call stack data structure. Whenever the tool detects that a new function is entered or left, the above mentioned calling context is updated.

We modified the existing implementation in the following way to allow for argument controlled profiling. On entering a function selected by `--separate-par`, the inserted code does not directly push the symbol name of the entered function to the calling context, but generates a new symbol by appending argument value information to the function name. At this point in time, the value of the stack pointer is readily available to allow access to the function arguments. Finally, the new symbol name is used to update the calling context.

5 Case Study

In the following, we optimize a PDE solver working on adaptive Cartesian grids [14], and we place special emphasis on the benefit of the argument controlled profiling throughout the manual optimization efforts. Although this is an example from scientific computing only, similar rationale, reasonings, and techniques apply to many fields of application.

The solver traverses a tree-like data structure (Fig. 3) with a recursive algorithm which is split up among several routines, and we pick out and concentrate on one particular subroutine. This routine is computationally demanding, as it realizes both the handling of the adaptivity as well as structure of the grid—the grid structure is analyzed, data is preprocessed, and the recursive calls are invoked—and the grid changes due to refinements as well as coarsenings. The tree data structure may change throughout the computation and the routine has to implement the corresponding data transitions and modifications.

The analyzed code part accepts two parameters indicating, on the one hand, whether the global tree will change at all—an information available due to the data structure's public signature—and, on the other hand, whether the current recursion step corresponds to a leaf, i.e. whether it invokes further recursive function calls or not. We run Callgrind, make it distinguish the values of both boolean arguments, and end up with four different parameter constellations for the observed function. The profiling reveals that the first boolean argument holds for most of the recursive calls. Furthermore, the control flow of a recursion tail identified by the latter argument differs significantly from the control flow of all other recursion steps. Entry conditions for some basic blocks never hold. Consequently, these paths are never executed, and the operations to evaluate the entry conditions—they are realized as calls to small helper routines—are obsolete.

A simple refactoring due to this insight delivered by the new tool facility provides two variants of the original code: A standard variant and a variant coming into play whenever the tree data structure does not change (according to the first function argument) or whenever the end of the recursion is reached. We hereafter continue to analyze the effect of the recursion state/flag combination on the individual traversal code parts and apply similar optimizations on

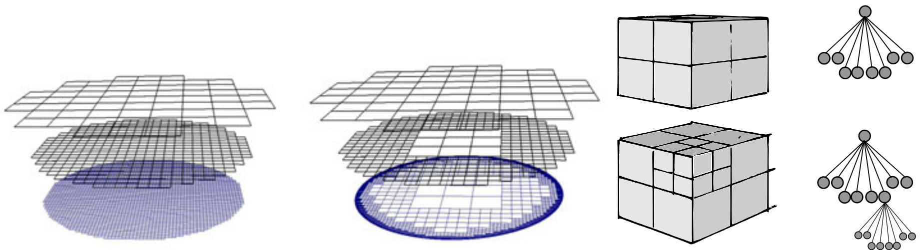

Fig. 3. Cascade of regular 2D Cartesian grids (left), adaptive Cartesian grids (middle), and 3D grids with corresponding tree data structure (right)

several code fragments, i.e. we remove, for one code variant, unnecessary calls to helper functions wherever possible. While all these optimized variants can be derived manually, Callgrind's argument controlled profiling allows us to quantify the performance improvement a priori and often identifies code parts we had not thought about before. The quantification is particularly important if the optimization leads to code duplication, i.e. if it worsens the code's quality. Here, it is important to balance two contradicting coding principles: software quality and performance. Finally, spending only three development days on that kind of optimizations reduced the code's runtime by a factor of more than 16%.

6 Conclusion and Future Work

Argument controlled profiling is a useful feature for analyzing code which makes heavy use of recursive functions. Such code structures are e.g. found within sophisticated PDE solvers where the use of recursive data structures is mandatory for the implementation of adaptive grids. In an industry cooperation, we currently analyze a code which is another candidate for argument controlled profiling: Here, expressions represented by abstract syntax trees (AST) are evaluated. Our approach is valuable in finding often used node type sequences which are candidates for new *super node* types, potentially shrinking the AST—and thus, evaluation time—significantly. To sum it up, we believe that our new profiling technique is of great value for any application traversing tree or graph structures, but also generally for functions whose runtime behavior depends on its arguments.

The current Callgrind prototype is available at `http://mmi.cs.tum.edu`. However, to include our extension into future Callgrind releases, we are going to make it user friendly, flexible, and portable. The user wants to specify function arguments by name. Further, the tool should support more data types (it currently solely works with integer types). Finally, it should run on all architectures supported by Valgrind (currently, it is Linux/x86 only).

Profiling using architecture simulation can be cumbersome to apply to parallel codes. However, for sequential optimization of parallel code, it is usually possible to profile the one-process special case, where our prototype works fine. For future work, it would be interesting to implement our technique also for profilers using real-time measurement, e.g. sampling tools on Linux (such as OProfile [7]). This would make argument controlled profiling applicable also for usage with parallel application runs. The key here is to efficiently and reliably check whether a given instruction address (in the IP register or a return address from the stack) is part of a function whose symbol should be augmented by the value of a function argument, and how to get at this value.

Acknowledgment

This work was supported by the International Graduate School of Science and Engineering (IGSSE), a Ph.D. program at Technische Universität München. This

program emphasizes the cooperation of interdisciplinary research teams. In such a cooperation, the difficulty of analyzing a scientific code—the one used as showcase in this paper—came up and motivated the development of the presented profiling technique.

References

1. Ammons, G., Ball, T., Larus, J.R.: Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling. In: Proceedings of PLDI '97 (June 1997)
2. Bernat, A.R., Miller, B.P.: Incremental call-path profiling. Concurrency and Computation: Practice and Experience 19, 1533–1547 (2007)
3. Bond, M.D., McKinley, K.S.: Probabilistic calling context. In: Proceedings of OOPSLA'07, Montreal, Quebec, Canada (October 2007)
4. Intel Corporation. Intel Performance Tuning Utility, `http://software.intel.com/en-us/articles/intel-performance-tuning-utility`
5. Froyd, N., Mellor-Crummey, J., Fowler, R.: Low-overhead call path profiling of unmodified, optimized code. In: Proceedings of ICS'05, Cambridge, MA (June 2005)
6. Graham, S., Kessler, P., McKusick, M.: GProf: A call graph execution profiler. In: SIGPLAN: Symposium on Compiler Construction, pp. 120–126 (1982)
7. Levon, J.: OProfile, a system-wide profiler for Linux systems, `http://oprofile.sourceforge.net`
8. Malony, A.D., Shende, S.S., Morris, A.: Phase-based parallel performance profiling. In: Proceedings of ParCo'05, Jülich, Germany. Parallel Computing: Current & Future Issues of High-End Computing, vol. 33, pp. 203–210 (2006)
9. Nethercote, N., Seward, J.: Valgrind: A framework for heavyweight dynamic binary instrumentation. In: ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007), San Diego, California, USA (June 2007)
10. Shende, S., Malony, A.D.: TAU: The TAU parallel performance system. International Journal of High Performance Computing Applications 20(2), 287–311 (2006)
11. Spivey, J.M.: Fast, accurate call graph profiling. Software – Practice Experience 34, 249–264 (2004)
12. Tallent, N.R., Mellor-Crummey, J.M., Fagan, M.W.: Binary analysis for measurement and attribution of program performance. In: Proceedings of PLDI'09, Dublin, Ireland (June 2009)
13. Weidendorfer, J., Kowarschik, M., Trinitis, C.: A tool suite for simulation based analysis of memory access behavior. In: Bubak, M., van Albada, G.D., Sloot, P.M.A., Dongarra, J. (eds.) ICCS 2004. LNCS, vol. 3038, pp. 440–447. Springer, Heidelberg (2004)
14. Weinzierl, T.: A Framework for Parallel PDE Solvers on Multiscale Adaptive Cartesian Grids. Dissertation, Institut für Informatik, Technische Universität München, München (2009)
15. Zhuang, X., Serrano, M.J., Cain, H.W., Choi, J.-D.: Accurate, efficient, and adaptive calling context profiling. In: Proceedings of PLDI'06, Ottawa, Ontario, Canada, June 2006. ACM, New York (2006)