Overview of Application Instrumentation for Performance Analysis and Tuning

Ondrej Vysocky[1(✉)], Lubomir Riha[1], and Andrea Bartolini[2]

[1] IT4Innovations National Supercomputing Center,
VŠB – Technical University of Ostrava, Ostrava, Czech Republic
{ondrej.vysocky,lubomir.riha}@vsb.cz
[2] University of Bologna, DEI, via Risorgimento 2, 40136 Bologna, Italy
a.bartolini@unibo.it

Abstract. Profiling and tuning of parallel applications is an essential part of HPC. Analysis and improvement of the hot spots of an application can be done using one of many available tools, that provides measurement of resources consumption for each instrumented part of the code. Since complex applications show different behavior in each part of the code, it is desired to insert instrumentation to separate these parts.

Besides manual instrumentation, some profiling libraries provide different ways of instrumentation. Out of these, the binary patching is the most universal mechanism, that highly improves user-friendliness and robustness of the tool. We provide an overview of the most often used binary patching tools and show a workflow of how to use them to implement a binary instrumentation tool for any profiler or autotuner. We have also evaluated the minimum overhead of the manual and binary instrumentation.

Keywords: Binary instrumentation · Performance analysis · Code optimization · High performance computing

1 Introduction

Developers of HPC applications are forced to optimize their applications to reach maximum possible performance and scalability. This request makes the performance analysis tools very important elements of the HPC systems, that have a goal in the identification of the hot spots of the code that provides space for improvement. Except basic, single-purpose applications every region of an application may have different requirements on the underlying hardware. In general, we may speak about several application kernels, that are bounded due to different (micro-)architectural components (e.g. compute, memory bandwidth, communication or I/O kernels) presented in [1] and evaluated in [13,29].

An application performance analysis tool provides profiling of the application - stores current time, hardware performance counters et cetera, to provide information about the program status at the given time. In general there are

R. Wyrzykowski et al. (Eds.): PPAM 2019, LNCS 12044, pp. 159–168, 2020.
https://doi.org/10.1007/978-3-030-43222-5_14

several ways how to connect the profiling library with the target application to select when the application's state should be captured, (1) insert the profiling library API functions into the code of the profiled application, or (2) the profiling library implements a middleware for a specific functions (e.g. memory access, I/O or MPI etc). Another option could be monitoring or simulating the application process, however these approaches may have a problem to profile exactly the application performance. The advantage of the monitoring is that it does not require to instrument the application, which means that identification of exact location in such code is ambiguous. Instrumentation can be inserted manually to the source code, by a compiler at the compilation time or the application's binary can be patched using dynamic or static instrumentation tools.

1.1 Motivation

The list of the HPC applications profiling tools is quite long, and despite many features are shared among them, every tool brings something extra to provide slightly different insight into the application's behavior. New HPC machines come with new challenges that require different ways how to optimize the code. With the upcoming HPC exascale era, there is pressure to reduce energy consumption of the system and the applications too. Several projects develop autotuning tools for energy savings based on CPU frequencies scaling or using Intel RAPL power capping [9], e.g. GEOPM [6], COUNTDOW [5], Adagio [22] or READEX [20,23].

One of the READEX tools is MERIC [14,29] library, that has been developed, to provide application behavior analysis and information about its energy consumption when different application or system parameters are tuned. MERIC dynamically changes the tuned parameters and searches for the configuration in which each part of the application fully utilizes the system, not to waste the resources and bring energy or time savings. This way user can detect that some parts of the target application when uses just one of two sockets is as fast as when using them both due to strong NUMA (Non-Uniform Memory Access) effect, or that the frequency of the CPU cores can be significantly reduced, due to inefficient memory access pattern. MERIC supports manual instrumentation only, which we have identified as a weak spot on a way to reach maximum possible savings. First of all process of localization where to insert the manual instrumentation to the source code is time consuming, which may lead to the situation that some parts of the code will not be sufficiently covered, and due to that the code analysis may miss identification some of the code's dynamicity and result configuration will be sub-optimal.

It is barely possible to specify a single rule for all autotuning frameworks that decides which parts of a code should be instrumented, but the most universal way is specification of a minimum region size. Under the READEX project has been specified that the minimum size of an instrumented function is 100 ms to the tuning framework be able to change the system settings of the contemporary Intel x86 processors and provide reliable energy measurement for all the

instrumented regions (Intel RAPL counters [12] and HDEEM [11], 1 kHz power-sampling energy measurement systems have been used in the project).

To reach maximum possible savings, the application should contain the maximum possible amount of regions, that may show different behavior. It results in search for all regions that last more than the selected threshold and instrument them, nevertheless the threshold can be extended if the instrumentation is too heavy. In general, too detailed instrumentation can be handled also at the tuning framework side, that may ignore some of the regions, but anyway even this solution will lead to some minimal overhead depending on the framework's implementation (e.g. minimal time between regions' starts, maximum level of nesting). For purpose of identification regions with runtime longer than a specified threshold a Timeprof library [14] has been developed. The library does time measurement of the application's functions and provides a list of functions that fulfill the condition.

2 Performance Analysis Tools

List of HPC tools for application performance analysis is very long so we decided to focus on open-source tools that are selected by OpenHPC [19] project whose mission is to provide a reference collection of open-source HPC software components and best practices, lowering barriers to deployment, advancement, and use of modern HPC methods and tools. The project mentions the following tools:[1]

LIKWID [27] is one of the performance monitoring and benchmarking suite of command-line applications. Extrae [24] is a multi-platform trace-file generator to monitor the application's performance. Score-P is a library for profiling and tracing, that provides core measurement services for other libraries - Scalasca [7], TAU [25], Vampir [17] and Periscope Tuning Framework (PTF) [8]. Scalasca and TAU are very similar profiling and tracing tools that can also cooperate - e.g. Scalasca's trace-files can be visualized using TAU's profile visualizer. Vampir framework provides event tracing and focuses mainly on the visualization part of the analysis process. On the other hand, PTF is an autotuning framework, providing many plugins to tune the application from various perspectives. GEOPM is an autotuning tool focused on x86 systems, that dynamically coordinating hardware settings across all compute nodes used by an application according to the application's behavior and requests from the resource manager. The last tool from our list is the mpiP [21], which is a lightweight profiling library for MPI applications, based on middleware of the MPI functions, despite that it also has a limited list of C API functions to manually instrument the application, as well as all the mentioned tools.

Besides splitting the application into different parts of the code, some tools also provide an opportunity to instrument the most time consuming loops of the target application (e.g. in case of Score-P we speak about a Phase region, GEOPM terminology uses word Epoch, etc.). This kind of annotation is useful

[1] OpenHPC project list of performance analysis tools besides mentioned libraries contain tools without API (e.g. visualization libraries) and also PAPI [26].

especially in case of tools that do not only analyze the application but also provide the opportunity to tune the application performance using some kind of optimization.

3 Manual and Compiler Inserted Instrumentation

Manual instrumentation usually wraps a function, block of functions (with the similar behavior) or is inserted inside a loop body, to detect different behavior within the iterations, or in case of autotuning tools to identify optimal configuration by switching the configuration in each iteration.

Manual source code instrumentation requires access to the source code to insert the API functions and at least a basic knowledge of the application behavior, to instrument the most significant regions. The application must be recompiled for each change in the instrumentation. Due to these requirements, manual instrumentation is time-consuming and inconvenient.

Despite some of the performance analysis tools provides options how to analyze the application without doing changes in the source code, using the middleware (mpiP), compiler instrumentation (Score-P) or binary instrumentation (extrae, TAU), anyway all of the mentioned tools have their own API to let the application user/developer extend the instrumentation about specific parts of the application.

Compiler instrumentation is provided by the Score-P or by the GNU profiler gprof [10], it provides a possibility to wrap applications' functions with the instrumentation at the compilation time. In comparing to the manual instrumentation it removes the requirement to browse the source code to locate the requested functions, however, the handicap of accessing the source code persists. In default settings compiler instruments all the application's functions, without any limit on the function size, which in many cases may cause high overhead of the profiling, when measuring performance of the shortest regions too. Due to that, the compiler provides an option on how to select/filter a subset of the functions to instrument. Unfortunately, it leads to repeated compilation of the target application, which is usually slower than plain compilation (e.g. Score-P does not support parallel compilation).

4 Binary Patching

Binary patching means a modification of an application execution without recompilation of the source code. The modifications can be done dynamically during the application run or statically rewrite the binary with all the necessary changes and store the edited binary into a new file.

Dynamic Binary Instrumentation (DBI) tools [4,16,18] interrupt the analyzed application process and switch context to the tool at a certain point that should be instrumented, and execute a required action. This approach causes an overhead that is usually not acceptable for autotuning or performance analysis. On the other hand, a binary generated by a Static Binary Instrumentation

(SBI) tool should not cause any extra overhead in comparison to manually instrumented code, which confirms our measurements presented later in this section.

SBI tools not only insert functions calls at certain positions in the instrumented binary, but also add all the necessary dependencies to the shared libraries, so it is not required to recompile the application for its analysis. Also, SBI tools can access both mangled and demangled names of the functions even though the application has been compiled without debug information. SBI tools are provided by TAU (using Dyninst [3] or Pebil [15] or MAQAO [2]) and extrae (using Dyninst) and Score-P uses Dyninst to instrument the code by its compiler.

PEBIL is a binary rewriting tool allowing to patch ELF files for the x86-64 architecture. Unfortunately, PEBIL project is closed since 2017, so support for new platforms is not guaranteed. Due to that, we will focus on Dyninst and MAQAO only, from which MAQAO-2.7.0 supports the IA-64 and Xeon Phi architectures only, on the other hand, Dyninst-10.0.0 InstructionAPI implementation supports the IA-32, IA-64, AMD-64, SPARC, POWER, and PowerPC instruction sets and ARMv8 is in experimental status.

We have evaluated overhead of instrumentation when inserted manually with statically inserted instrumentation by MAQAO and Dyninst. We have used MERIC library for this measurement, that reads requested system information and store the value in memory. A single thread application (to remove the influence of an MPI/OpenMP barriers on the measurement) contained one region, that had been performed thousand times. We have not seen any difference in the overhead of manual instrumentation and SBI. Overhead of one instrumentation call on an Intel Xeon E5-2697v4 is:

- 175 µs – when reading timestamp
- 375 µs – when reading energy consumption using Intel RAPL (read four hardware counters and timestamp)

In the case of binary patching of a complex application, the time that is required to insert the instrumentation should not exceed the time needed for the application compilation. According Valensi MAQAO is able to insert 18 000 function calls in less than a minute [28].

4.1 A Binary Parsing

Dyninst as well as MAQAO holds the executable in a structure of components as the application was decomposed by a compiler. The components and their relations are illustrated in the Fig. 1. A binary base element is one or several images, which is a handle to the executable file associated with this binary. Each image contains a list of functions and global variables. A function can be also inspected for local variables and basic blocks (BBs), which is a sequence of the instructions with a single entry point and single exit point. The BBs are organized in a control-flow graph (CFG), that represents the branches of the code. From a basic block, it is also possible to access its instructions.

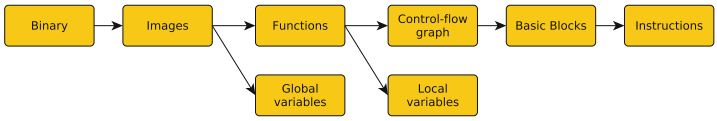

Fig. 1. Components of an application binary produced by a compiler.

When using Dyninst to browse through an application binary for its analysis or patching all the components on higher levels must be accessed first, on the other hand, MAQAO interface allows a user to access them directly. Anyway, we are primarily interested in the insertion of a function call before and after selected functions, we may stay at the level of functions.

4.2 Workflow

In this section, we will present a process of an SBI using MAQAO or Dyninst, with a goal insert a profiler function call before and after a select application function. The patching libraries provide much more functionality than presented (e.g. static binary analysis or insertion of a function call at more general locations), however for most of the profilers and autotuners wrapping a function with its instrumentation should be sufficient.

```
BPatch bpatch;
BPatch_binaryEdit *appBin = bpatch.openBinary("a.out", false);
BPatch_image *appImage = appBin->getImage();
// prepare function printf with its paramters to be inserted
std::vector<BPatch_function*> insertFunc;
appImage->findFunction("printf", insertFunc, true, true, true);
std::vector<BPatch_snippet*> args;
BPatch_snippet* param1 = new BPatch_constExpr("FUNC %s\n");
BPatch_snippet* param2 = new BPatch_constExpr("main");
args.push_back(param1);
args.push_back(param2);
// identify target location for insertion
std::vector<BPatch_function*> functions;
std::vector<BPatch_point *> *points;
appImage->findFunction("main", functions);
points = functions[0]->findPoint(BPatch_entry);
// function call insertion and store the new binary to a file
BPatch_funcCallExpr insertCall(*(insertFunc[0]), args);
appBin->insertSnippet(insertCall, *points);
appBin->writeFile ("b.out");
```

Listing 1: Dyninst code to instrument main function in a.out binary.

Both Dyninst and MAQAO open the binary and starts with its decomposition into the components as it was previously presented. We can select a function to insert from the dependent shared libraries of the application. If the application has been compiled without the profiling library, the first step should be adding all the necessary dependencies, which is a single function call.

With all the necessary dependencies, it is possible to find the function we want to insert under the application image, as well as the functions we want

to wrap into the profiler instrumentation. To find the function that should be instrumented, the binary (modules in case Dyninst) must be browsed for this function. The function may have several code locations that could be instrumented, from which we are interested in its entry and exit points (addresses). With this point, it is possible to associate a function call with the requested list of arguments (be aware that there is no argument type control). This change must be committed to the binary. The edited binary is then stored and is ready to be executed to analyze its performance.

Listings present a code snapshots that insert *printf* call, that will print "FUNC main" at the beginning of execution main function of a C application a.out and stores the binary as b.out using Dyninst (Listing 1) or MAQAO (Listing 2) libraries. The examples assume that printf function is available to be added, otherwise relevant shared library dependency must be added too, also return codes are ignored to reduce size of the Listings.

```
project_t* proj = project_new("instrument_proj");
asmfile_t* asmf = project_load_file(proj, "a.out", NULL);
elfdis_t* elf = madras_load_parsed (asmf);
madras_modifs_init (elf, STACK_KEEP, 0);
fct_t* func = hashtable_lookup(asmf->ht_functions, "main");
if (func != NULL) //if main function has been found in the binary
{   //search for entry instructions of the main function
  queue_t * instructions = fct_get_entry_insns(func);
  list_t* iter = queue_iterator(instructions);
  while (iter != NULL)
  {   //insert printf function call and its parameters
    insn_t * inst = iter->data;
    modif_t* ifct = madras_fctcall_new(elf, "printf", NULL, inst->address, 0, NULL, 0);
    madras_fctcall_addparam_fromglobvar(elf,ifct,NULL,"FUNC %s\n",'a');
    madras_fctcall_addparam_fromglobvar(elf,ifct,NULL,"main",'a');
    iter = iter->next;
  }
  madras_modifs_commit(elf, "b.out"); //store the edited binary to a file
}
project_free(proj);
madras_terminate(elf);
```

Listing 2: MAQAO code to instrument main function in a.out binary.

5 Conclusion

Compiler and binary instrumentation are solution for a fully automatized application analysis and following optimized run of the application, but only in the case that such instrumentation does not lead to significantly higher overhead than in case of the manual instrumentation. Our measurements have not seen any measurable difference in manual and static binary instrumentation provided by MAQAO or Dyninst. We consider SBI as simple and the most powerful solution and based on this conclusion when writing a tool for an application behavior analysis we recommend to provide also an SBI support and present samples of code using both Dyninst and MAQAO to show how simple a basic SBI tool is.

The problem of the *ideal instrumentation* (amount and location of the probes) has a massive impact on the effectiveness of every auto-tuning framework. Autoinstrumentation tool can be written to instrument the analyzed application according to the requirements of the autotuner and its way of tuning the application. Timeprof library helps to identify the significant regions of the application to analyze their behavior. We can easily measure the runtime of all the functions of the application with Timeprof, which will provide us a selection of the regions. Afterward, the identified regions are instrumented with the selected library.

Acknowledgment. This work was supported by The Ministry of Education, Youth and Sports from the Large Infrastructures for Research, Experimental Development and Innovations project IT4Innovations National Supercomputing Center LM2015070. This work was supported by the Moravian-Silesian Region from the programme "Support of science and research in the Moravian-Silesian Region 2017" (RRC/10/2017). This work was also partially supported by the SGC grant No. SP2019/59 "Infrastructure research and development of HPC libraries and tools", VŠB - Technical University of Ostrava, Czech Republic.

References

1. Asanovic, K., et al.: The landscape of parallel computing research: a view from Berkeley. Technical report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, December 2006. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html
2. Barthou, D., Charif Rubial, A., Jalby, W., Koliai, S., Valensi, C.: Performance tuning of x86 OpenMP codes with MAQAO. In: Müller, M.S., Resch, M.M., Schulz, A., Nagel, W.E. (eds.) Tools for High Performance Computing 2009, pp. 95–113. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11261-4_7
3. Bernat, A.R., Miller, B.P.: Anywhere, any-time binary instrumentation. In: Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools. PASTE 2011, pp. 9–16. ACM, New York (2011). https://doi.org/10.1145/2024569.2024572
4. Bruening, D., Zhao, Q., Amarasinghe, S.: Transparent dynamic instrumentation. In: Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments. VEE 2012, pp. 133–144. ACM, New York (2012). https://doi.org/10.1145/2151024.2151043
5. Cesarini, D., Bartolini, A., Bonfà, P., Cavazzoni, C., Benini, L.: COUNTDOWN - three, two, one, low power! A run-time library for energy saving in MPI communication primitives. CoRR abs/1806.07258 (2018). http://arxiv.org/abs/1806.07258
6. Eastep, J., et al.: Global extensible open power manager: a vehicle for HPC community collaboration on co-designed energy management solutions. In: Kunkel, J.M., Yokota, R., Balaji, P., Keyes, D. (eds.) ISC 2017. LNCS, vol. 10266, pp. 394–412. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-58667-0_21
7. Geimer, M., Wolf, F., Wylie, B.J.N., Ábrahám, E., Becker, D., Mohr, B.: The scalasca performance toolset architecture. Concurr. Comput.: Pract. Exp. **22**(6), 702–719 (2010). https://doi.org/10.1002/cpe.v22:6

8. Gerndt, M., Cesar, E., Benkner, S.: Automatic tuning of HPC applications - the periscope tuning framework (PTF). In: Automatic Tuning of HPC Applications - The Periscope Tuning Framework (PTF). Shaker Verlag (2015). http://eprints.cs.univie.ac.at/4556/
9. Gholkar, N., Mueller, F., Rountree, B.: Power tuning HPC jobs on power-constrained systems. In: Proceedings of the 2016 International Conference on Parallel Architectures and Compilation. PACT 2016, pp. 179–191. ACM, New York (2016). https://doi.org/10.1145/2967938.2967961
10. Graham, S.L., Kessler, P.B., McKusick, M.K.: Gprof: A call graph execution profiler. SIGPLAN Not. **39**(4), 49–57 (2004). https://doi.org/10.1145/989393.989401
11. Hackenberg, D., et al.: HDEEM: high definition energy efficiency monitoring. In: 2014 Energy Efficient Supercomputing Workshop, pp. 1–10, November 2014. https://doi.org/10.1109/E2SC.2014.13
12. Hähnel, M., Döbel, B., Völp, M., Härtig, H.: Measuring energy consumption for short code paths using RAPL. SIGMETRICS Perform. Eval. Rev. **40**(3), 13–17 (2012). https://doi.org/10.1145/2425248.2425252
13. Haidar, A., Jagode, H., Vaccaro, P., YarKhan, A., Tomov, S., Dongarra, J.: Investigating power capping toward energy-efficient scientific applications. Concurr. Comput.: Pract. Exp. **31**(6), e4485 (2019). https://doi.org/10.1002/cpe.4485
14. IT4Innovations: MERIC library. https://code.it4i.cz/vys0053/meric. Accessed 21 Apr 2019
15. Laurenzano, M.A., Tikir, M.M., Carrington, L., Snavely, A.: PEBIL: efficient static binary instrumentation for Linux. In: 2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS), pp. 175–183, March 2010. https://doi.org/10.1109/ISPASS.2010.5452024
16. Luk, C.K., et al.: Pin: building customized program analysis tools with dynamic instrumentation. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI 2005, pp. 190–200. ACM, New York (2005). https://doi.org/10.1145/1065010.1065034
17. Müller, M.S., et al.: Developing scalable applications with Vampir, VampirServer and VampirTrace. In: PARCO (2007)
18. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. SIGPLAN Not. **42**(6), 89–100 (2007). https://doi.org/10.1145/1273442.1250746
19. OpenHPC: Community building blocks for HPC systems. https://openhpc.community/. Accessed 21 Apr 2019
20. READEX: Horizon 2020 READEX project (2018). https://www.readex.eu
21. Roth, P.C., Meredith, J.S., Vetter, J.S.: Automated characterization of parallel application communication patterns. In: Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing. HPDC 2015, pp. 73–84. ACM, New York (2015). https://doi.org/10.1145/2749246.2749278
22. Rountree, B., Lowenthal, D.K., de Supinski, B.R., Schulz, M., Freeh, V.W., Bletsch, T.K.: Adagio: making DVS practical for complex HPC applications. In: Proceedings of the 23rd International Conference on Supercomputing. ICS 2009, pp. 460–469. ACM, New York (2009). https://doi.org/10.1145/1542275.1542340
23. Schuchart, J., et al.: The READEX formalism for automatic tuning for energy efficiency. Computing **99**(8), 727–745 (2017). https://doi.org/10.1007/s00607-016-0532-7

24. Servat, H., Llort, G., Huck, K., Giménez, J., Labarta, J.: Framework for a productive performance optimization. Parallel Comput. **39**(8), 336–353 (2013). https://doi.org/10.1016/j.parco.2013.05.004, http://www.sciencedirect.com/science/article/pii/S0167819113000707
25. Shende, S.S., Malony, A.D.: The TAU parallel performance system. Int. J. High Perform. Comput. Appl. **20**(2), 287–311 (2006). https://doi.org/10.1177/1094342006064482
26. Terpstra, D., Jagode, H., You, H., Dongarra, J.: Collecting performance data with PAPI-C. In: Müller, M.S., Resch, M.M., Schulz, A., Nagel, W.E. (eds.) Tools for High Performance Computing 2009, pp. 157–173. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-11261-4_11
27. Treibig, J., Hager, G., Wellein, G.: LIKWID: lightweight performance tools. In: Bischof, C., Hegering, H.G., Nagel, W.E., Wittum, G. (eds.) Competence in High Performance Computing 2010, pp. 165–175. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-24025-6_14
28. Valensi, C.: A generic approach to the definition of low-level components for multi-architecture binary analysis. Ph.D. thesis, Université de Versailles Saint-Quentin-en-Yvelines, July 2014
29. Vysocky, O., et al.: Evaluation of the HPC applications dynamic behavior in terms of energy consumption. In: Proceedings of the Fifth International Conference on Parallel, Distributed, Grid and Cloud Computing for Engineering, pp. 1–19 (2017). https://doi.org/10.4203/ccp.111.3. Paper 3, 2017