Performance Analysis on Energy Efficient High-Performance Architectures

Roman Iakymchuk, François Trahay

Performance Analysis on Energy Efficient High-Performance Architectures

Roman Iakymchuk[1] and François Trahay[1]

[1]*Institut Mines-Télécom – Télécom SudParis*
9 Rue Charles Fourier
91000 Évry France

{roman.iakymchuk, francois.trahay}@telecom-sudparis.eu

Abstract.

With the shift in high-performance computing (HPC) towards energy efficient hardware architectures such as accelerators (NVIDIA GPUs) and embedded systems (ARM processors), arose the need to adapt existing performance analysis tools to these new systems. We present EZTrace – a performance analysis framework for parallel applications. EZTrace relies on several core components, in particular on a mechanism for instrumenting functions, a lightweight tool for recording events, and a generic interface for writing traces. To support EZTrace on energy efficient HPC systems, we developed a CUDA module and ported EZTrace to ARM processors. The evaluation on a suite of the standard computation kernels show that EZTrace allows to analyze HPC applications executing on such systems with the low performance overhead.

Keywords

Performance analysis, energy efficiency, ARM processors, GPUs accelerators.

1 Introduction

For decades the dominant hardware resources in high-performance computing (HPC) were general purpose processors such as the X86 processor line. However, in the past few years this trend started changing towards accelerators and low-power embedded technology. The main reason behind is the energy consumption of the classic HPC clusters and supercomputers. For those, the hardware may cost significantly less than the power supply. Among the first non-traditional HPC resources were accelerators like GPUs from NVIDIA. At first, GPUs appeared as auxiliary resources on the HPC clusters. By the time passed, their niche grown and now they are the fundamental components of the standard HPC supercomputers; in particular, those ranked by both the Green500 [12] and Top500 [20] lists. For instance, the fastest supercomputer in the world by the Top500 – Titan - Cray XK7 – has equal amount of AMD Opteron processors as well as NVIDIA K20 accelerators. Comparing the current fastest to the K supercomputer, which was the number one on June, 2011 and was completely build in the standard manner, we notice that the peak performance of the Titan is twice as the K performance, while the power consumption is 17 % less. Thus, accelerators lower the energy consumption of the supercomputers while delivering better performance. However, the accelerators introduce a new programming model that requires to heavily modify existing applications. Moreover, the efficiency of accelerators highly depends on the type of application: some HPC applications run better on standard CPUs than on GPUs.

Another approach for the energy efficient high-performance computing is to use low-power embedded technologies such as ARM processors. The idea here is to utilize numerous low-power devices to achieve the peak performance comparable to the standard clusters, however with the reduced power consumption. This is an objective of the Mont-Blanc project that aims at designing a new type of computer architecture, which will possibly set a new future HPC standards, that potentially will be able to deliver the EFlops performance while consuming 15-30 times less power [18]. The advantage of using embedded systems like ARM processors is rather the straightforward mechanism for adjusting current parallel applications. Unlike for GPUs, there is no need to change the programing model and applications can use the traditional MPI approach.

With the grown concern regarding energy efficiency that caused the development of new low-power consuming HPC systems, the developers of high-performance and parallel applications face two main difficulties:

- Programmability. Due to the shift towards new HPC architectures, either parts of the software or the whole software have to be modified or rewritten;
- Portability. Moving the current applications to the new HPC systems may result in either multiple bugs or poor performance.

The latter can be resolved with the help of performance analysis tools. The purpose of such tools is not only to fix bugs, but it also covers performance analysis and profiling applications behavior that aims at improving their efficiency [13].

We develop the EZTrace framework [5, 8] that aims at automatic generation and analysis of execution traces from high-performance and parallel applications. The generation of such traces requires to intercept the calls to a set of key functions – such as MPI communication primitives and synchronization functions – and to record events in a trace file. EZTrace relies on the instrumentation of functions that does not modify the program's behavior and, therefore, causes the low overhead.

The goal of this paper is to provide users and developers with an ability to analyze the performance of parallel applications on energy efficient HPC systems such as accelerators (NVIDIA GPUs) and embedded systems (ARM processors). For that, we develop the EZTrace CUDA module and port EZTrace to the ARM architecture. The latter is the main concern of this article.

The remaining of the paper is organized as follows. Section 2 provides literature overview of performance tracing on both the standard and embedded systems. Section 3 presents the EZTrace framework. We evaluate EZTrace on ARM processors in Section 4, and discuss conclusions and the future work in Section 5.

2 Literature Overview

While developing high-performance or parallel applications, a big concern is put on the optimization and the communication between threads/processes/nodes. In order to create scalable software with the optimal execution time, performance analysis and, especially, performance tracing are being involved; numerous research has been conducted on those two topics. As a result, many tools were designed for tracing the execution of parallel applications for better understanding their behavior as well as finding the performance bottlenecks. Those tools can be divided into two categories: those that are specific to one programming model or library; and those for general purpose that cover multiple models and, therefore, can track calls to multiple libraries on various architectures. A particular examples of the first group are: MPI Parallel Environment (MPE) [7] that targets MPI applications; POSIX Thread Trace Toolkit (PTT) [10] aims at applications that use POSIX threads; OMPtrace [6] instruments OpenMP applications. The second group can be characterized by the following tools: VampirTrace [14], TAU [16], and Scalasca [11]; these tools can be used to instrument custom libraries or applications through the manual or automatic instrumentation of their code.

Orensanz provides an overview of tools for performance analysis covering the performance tracing topic too [15]. Even though performance analysis is considered in the scope of smartphones, tablets, and mobile Internet devices, the presented tools can be also applied for the other purposes like HPC applications on ARM processors.

Our interest is on the instrumentation-based performance tracing software on embedded systems. The common approach used to achieve this software relies on the manual or automatic modification of applications or their binaries. An example of such software is the Android TraceView tool [2, 15]. This tool generates time-based logs and profiling reports for Android Java applications. The disadvantage of using TraceView is that it slows down the application dramatically due to the instrumentation and the programming language used.

ARM CoreSight technology[1] [4, 15, 13] provides a set of modules for debugging and performance tracing with the focus on an entire system-on-chip (SoC). A particular examples of CoreSight trace modules are Embedded Trace Macrocells (ETM), Program Trace Macrocells (PTM), and Instrumentation Trace Macrocell (ITM). The ETM and PTM generate compressed traces of all instructions executed by the processor. Those instruction traces are 100 % non-intrusive and show the comprehensive visibility across the SoC at the finest possible granularity. The known limitations of the ETM and PTM: the grown number of processors (and their frequency) on a single SoC may slow the bandwidth; as Linux systems are multi-layered, this can make difficult to find

[1] ARM CoreSight is an architecture by itself including hardware and software layers.

events of interest in the trace. The ITM aims at supporting `printf` style debugging as well as recording OS and application events. Applications running on ARM can rapidly send data to the ITM. Then, this information can be transmitted to external monitoring tools for further analysis.

3 EZTrace

EZTrace is a general framework for analyzing high-performance and parallel applications. EZTrace relies on the tracing mechanism that executes an application only once in order to record the execution trace; the latter depicts the behavior of the program. After the execution of the application, EZTrace analyzes the resulting trace and extracts various statistics. It is also possible to generate trace files that can be visualized with tools as Vampir [14] and ViTE [19].

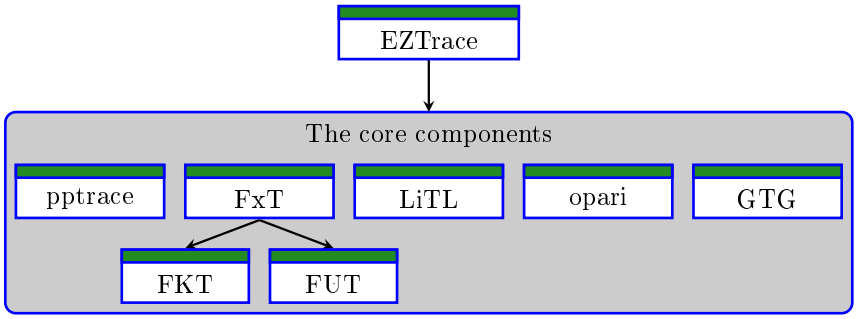

Figure 1: Structure of the EZTrace framework.

The structure of EZTrace core components is illustrated on Fig. 1. The `pptrace` tool is responsible for the instrumentation of functions. The instrumentation is crucial for the application performance. In EZTrace, it is performed by dynamically modifying the application using the low overhead mechanism [5]. `opari` is a source-to-source instrumentation tool for OpenMP and hybrid applications. For recording events, EZTrace relies on one of two different libraries: the Fast Kernel/User Trace (FxT) library [9]; the Lightweight Trace Library (LiTL). The first is a multi-level trace library that permits to record events from both user-space and kernel-space in a binary format. FxT can be used on embedded systems with some modifications, but the constraints of the target systems as well as the complexity of the library made us reconsider the usage of FxT for recording events there. Because of that, we developed our own tracing library named LiTL, which provides similar functionalities to FxT, for recording events from user-space with the low overhead. The last component of EZTrace is the Generic Trace Library (GTG) [17]. This tool provides a unique abstract interface for recording traces that allows to create traces in multiple formats like Pajé and OTF without any modifications.

EZTrace is shipped with modules for the common libraries used in parallel programming (MPI, OpenMP, etc.), and it allows third-party developers to create modules that instrument their own functions [21]. For instance, the PLASMA [1] linear algebra solver integrated an EZTrace module that permits to visualize the task scheduling performed by this library.

For recording trace files, EZTrace uses `pptrace` to modify on-the-fly the application's binary and inserts probes before and after the functions of interest [5]. Then, the program is executed in a usual way. When the program enters one of the instrumented functions (e.g. `MPI_Send`), EZTrace records an event, (`MPI_SEND_ENTRY`) executes the actual function (`MPI_Send`) and records another event that marks the end of the function (`MPI_SEND_EXIT`). Each event consists of a timestamp, an event code, the identifier of the thread that records this event as well as additional data depending on the event type. In case of the `MPI_SEND_ENTRY` event, as additional data EZTrace records the destination process, the message length, and the message tag. At the end of the application execution, events are written into trace files.

Afterwards, EZTrace reads the trace files that were generated and analyzes the events. Depending on the user choice, during the analysis phase various statistics, e.g. the time spend in the MPI communication, can be extracted and/or Pajé or OTF files can be generated; the latter can be visualized by both Vampir and ViTE.

3.1 The EZTrace CUDA module

In order to analyze applications that exploit GPUs, we develop an EZTrace module for the CUDA programming model. In the current EZTrace modules the instrumentation of functions is performed by recording an event

before and/or after the routine. However, this mechanism cannot be applied on the accelerators, because CUDA primitives are used on the CPU for offloading kernels to the GPU. So, recording an event at the beginning and the end of these functions only gather the information regarding the CPU execution.

For that reason, we use the `cuEvent` interface provided by CUDA to get additional information on the execution of applications on the accelerators. Using `cuEvents`, it is possible to know when a kernel or a memory transfer starts and ends on the GPU. EZTrace records this information to depict the behavior of the application on both CPUs and GPUs. These preliminary results on the CUDA module already allow us to analyze with EZTrace any CUDA application, including those that rely on the mixed programming models like MPI with CUDA.

3.2 EZTrace on embedded systems

When porting EZTrace to embedded systems, the first problem arose due to the mechanism that is applied to instrument functions. During the instrumentation, EZTrace modifies the binary code of the application in order to insert probes. These modifications require to inject binary instructions in the application memory space. The goal of this is to change the process flow of a set of functions, so that probes are called at the entry and exit of these functions.

Although the algorithm for inserting instructions is generic, the *opcodes* that are inserted are CPU-dependent. Porting EZTrace to a new type of CPUs is, thus, straightforward since it converges to identifying the opcodes that correspond to this CPU type. However, in case of the ARMv7 processors another problem appears due to the two different execution modes, namely ARM (32-bit) and Thumb (16-bit); the latter is set by default. As the main concern is on using the 32-bit ARM mode, we require applications to be compiled and executed under the ARM mode. The performance results of applying the EZTrace instrumentation can be found in Section 4.2.

4 Performance results

When analyzing the execution of high-performance or parallel applications that generate millions of events, it is important to keep the overhead of the performance tracing tool as low as possible. In this Section, we evaluate the performance and the overhead of EZTrace with LiTL on the standard application kernels.

4.1 Experimental Setup

The odroid cluster is equipped with six ARM Cortex-A9 (Cortex-A9) processors and 2 GB of RAM per each node. The Cortex-A9 processor has four cores operating at 1.6 GHz. Each core can execute one Flops per cycle for a peak performance of 1.6 GFlops/sec per core or 6.4 GFlops/sec per socket.

The odroid cluster's nodes execute the Ubuntu operating system release 12.11 (Linaro) with the 3.0.63 Linux kernel. On the odroid cluster, we installed EZTrace of version 0.9 and also the NAS Parallel Benchmarks (NPB) of version 3.3. Both were compiled with version 4.6.3 of the *gcc* compiler with the optimization level three enabled. To run the NPB benchmarks we used the OpenMPI library of version 1.6.3.

We measured the performance of the odroid cluster using the High-Performance LINPACK (HPL) library [3] of version 2.1 and the multi-threaded implementation of Basic Linear Algebra Subroutines (BLAS) from the Automatically Tuned Linear Algebra Software (ATLAS) library [22] of version 3.10.1. These libraries were compiled with the gcc compiler; the first under the optimization level three enabled; for the seconds the default optimization were applied, because versions of gcc up to 4.7.0 do not optimize the ATLAS code very well on ARM processors leading to the reduced performance. The result of the HPL execution on one node for a problem size 14080 is 3.79 GFlops; the energy consumption is less than 7 W. On the whole odroid cluster we were able to achieve 16.13 GFlops for a problem size 35840 consuming roughly 36 W.

4.2 NAS Parallel Benchmarks (NPB)

In order to evaluate the overhead of EZTrace with LiTL, we measure its performance for the NAS Parallel Benchmarks.NPB is a set of computation kernels derived from computational fluid dynamics applications to evaluate performance of parallel and high-performance applications. Later, the benchmark suite was extended including new kernels, e.g. for parallel I/O and computational grids. The problem sizes of those kernels are predefined and divided into eight classes; this value depends upon the kernel.

We conducted our experiments with 16 computing processes running the class B kernels of the NPB benchmarks on four nodes[2]. The MPI functions of the NPB kernels were instrumented with EZTrace. Each kernel was executed five times and, then, the *median* time was selected.

Table 1: The time measurements on NAS Parallel Benchmarks running for CLASS B problems with 16 MPI processes on the 4-node odroid cluster. In the table, secs stands for seconds.

Kernel	NPB(secs)	EZTrace(secs)	Overhead(%)
BT	295.97	297.74	0.60
MG	20.83	21.12	1.39
LU	324.27	310.41	-4.27
FT	283.01	284.00	0.35
IS	28.80	28.87	0.24
SP	483.02	484.10	0.22
CG	529.75	529.26	-0.09
EP	19.98	19.85	-0.65

Tab. 1 presents the results of the raw NPB kernels execution as well as their execution with the EZTrace instrumentation applied. For few kernels the overhead is negative, especially for LU it is -4.27 %. We investigated this issue: the minimum timings for the raw LU execution and with the EZTrace instrumentation are 305.65 and 304.63 secs, respectively; while the maximum timings – 326.34 and 327.77 secs, accordingly. Thus, the reason of such difference lays in the variation of the time measurements. In general, the performance results show that the EZTrace instrumentation of functions on ARM processors causes the small performance overhead – in most cases, the deviation is within 1 %.

5 Conclusions and Future Work

The new era in HPC is driven by the power consumption issue resulting in design and usage of the new hardware resources like the GPUs accelerators and the low-energy embedded systems. These changes are also reflected on many high-performance and parallel applications from various disciplines like finance and biology. Such applications either support the new HPC hardware or have completely moved there. Tuning those applications is a tedious task due to the complexity of hardware and the usage of hybrid programming models that combine MPI, OpenMP, and/or CUDA. This task cannot be solved without applying performance analysis tools.

We presented EZTrace – an open-source framework for performance analysis and tracing of high-performance and parallel applications. The focus of this article is on extending the framework to the recent energy efficient HPC systems like the GPU accelerators and the low-power ARM processors. To support the CUDA programming model, we developed a new EZTrace module. We also ported EZTrace with its components to ARM processors. In order to verify EZTrace on the latter hardware, we used a suite of NAS Parallel Benchmarks and compared the raw execution with the execution when the instrumentation is applied. As a result, the EZTrace instrumentation shows the low performance overhead. Therefore, it is not harmful for the applications performance.

Our future focus will be on optimizing the performance of the EZTrace CUDA module. Also, we plan to develop a module for the OpenCL programming model. As the new energy efficient architectures appeared, namely Intel Phi and Kalray Multi-Purpose Processor Array (MPPA), we would like to port EZTrace, including all core components, to these architectures as well.

References

[1] Emmanuel Agullo, Jim Demmel, Jack Dongarra, Bilel Hadri, Jakub Kurzak, Julien Langou, Hatem Ltaief, Piotr Luszczek, and Stanimire Tomov. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. *Journal of Physics: Conference Series*, 180, 2009.

[2] Android Developer Team. Profiling with traceview and dmtracedump. Available via the WWW. Cited 07 April 2013. `http://developer.android.com/tools/debugging/debugging-tracing.html`.

[2]We did not utilize the whole six-node cluster, because the major part of kernels requires the number of MPI processes to be a power of two.

[3] Antoine Petitet, R. Clint Whaley, Jack Dongarra and Andrew Cleary. HPL – A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers, September 2008. Available via the WWW. Cited 19 January 2012. http://www.netlib.org/benchmark/hpl/.

[4] ARM Ltd. ARM CoreSight. Available via the WWW. Cited 07 April 2013. http://www.arm.com/products/system-ip/coresight/index.php.

[5] Charles Aulagnon, Damien Martin-Guillerez, François Rue, and François Trahay. Runtime function instrumentation with EZTrace. In *Proceedings of the PROPER – 5th Workshop on Productivity and Performance*, Rhodes, Greece, August 2012.

[6] Jordi Caubet, Judit Gimenez, Jesús Labarta, Luiz De Rose, and Jeffrey S. Vetter. A Dynamic Tracing Mechanism for Performance Analysis of OpenMP Applications. In *OpenMP Shared Memory Parallel Programming, International Workshop on OpenMP Applications and Tools, WOMPAT 2001, West Lafayette, IN, USA, July 30-31*, pages 53–67, 2001.

[7] Anthony Chan, William Gropp, and Ewing L. Lusk. An efficient format for nearly constant-time access to arbitrary time intervals in large trace files. *Scientific Programming*, 16(2-3):155–165, 2008.

[8] Kevin Coulomb, Augustin Degomme, Mathieu Faverge, and François Trahay. An open-source tool-chain for performance analysis. *Tools for High Performance Computing 2011*, pages 37–48, 2012.

[9] Vincent Danjean, Raymond Namyst, and Pierre-André Wacrenier. An efficient multi-level trace toolkit for multi-threaded applications. In *Proceedings of the 11th international Euro-Par conference on Parallel Processing*, Euro-Par'05, pages 166–175, Berlin, Heidelberg, 2005. Springer-Verlag.

[10] Sebastien Decugis and Tony Reix. NPTL Stabilization Project. In *Linux Symposium*, volume 2, page 111, 2005.

[11] Markus Geimer, Felix Wolf, Brian J. N. Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. The Scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience*, 22(6):702–719, 2010.

[12] GREEN500.Org. The Green500's Energy-Efficient Supercomputers Sites. Available via the WWW. Cited 07 April 2013. http://www.green500.org/.

[13] Roberto Mijat. Better trace for better software. In *ARM Ltd*, 2010.

[14] Matthias S. Müller, Andreas Knüpfer, Matthias Jurenz, Matthias Lieber, Holger Brunst, Hartmut Mix, and Wolfgang E. Nagel. Developing scalable applications with vampir, vampirserver and vampirtrace. In *Parallel Computing: Architectures, Algorithms and Applications, ParCo 2007, Forschungszentrum Jülich and RWTH Aachen University, Germany, 4-7 September*, pages 637–644, 2007.

[15] Javier Orensanz. Performance analysis on ARM embedded Linux and Android systems. *Electronics World*, 117(1901):16–19, May 2011.

[16] Sameer S. Shende and Allen D. Malony. The Tau Parallel Performance System. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311, May 2006.

[17] GTG Developer Team. GTG – Generic Trace Generator. Available via the WWW. Cited 07 April 2013. http://gtg.gforge.inria.fr.

[18] Mont-Blanc Team. Mont-Blanc: European Approach Towards Energy Efficient High Performance. Available via the WWW. Cited 07 April 2013. http://www.montblanc-project.eu/.

[19] ViTE Developer Team. ViTE – Visual Trace Explorer. Available via the WWW. Cited 07 April 2013. http://vite.gforge.inria.fr.

[20] TOP500.Org. Top 500 Supercomputer Sites. Available via the WWW. Cited 07 April 2013. http://www.top500.org/.

[21] François Trahay, François Rue, Mathieu Faverge, Yutaka Ishikawa, Raymond Namyst, and Jack Dongarra. EZTrace: a generic framework for performance analysis. In *Proceedings of the IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, Newport Beach, CA, USA, May 2011.

[22] R. Clint Whaley and Jack J. Dongarra. Automatically Tuned Linear Algebra Software. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '98, pages 1–27. IEEE Computer Society, 1998.