2015 4th Eastern European Regional Conference on the Engineering of Computer Based Systems

A Profiling Tool for Heterogeneous Multi-core Systems

Ivan Povazan, Momcilo Krunic
RT-RK Institute for Computer Based Systems
Novi Sad, Serbia
ivan.povazan@rt-rk.com, momcilo-krunic@rt-rk.com

Miroslav Popovic
Faculty of Technical Sciences, University of Novi Sad
Novi Sad, Serbia
miroslav.popovic@rt-rk.com

Abstract — **Profiling is a process of collecting relevant data about the execution of an application. This is done in order to reveal code bottlenecks, measure the performance, and give detailed information about source code function contents and call graphs of the examined application. In this paper we present one such profiling tool designed for multi-core systems, which collects information about the multi-core activity and machine code statistics. The tool is implemented as part of the framework for development, deployment, debugging and control of DSP applications, and includes a profiling view which improves user experience and understanding of the profiling reports. Profiling reports can be used for improving the quality of the written code as well as to ease the process of debugging.**

Keywords - Profiling tool; Profiling View; Heterogeneous muti-core systems

I. INTRODUCTION

It has always been a challenge to meet the needs and constrains for DSP applications. Developers tend to achieve different goals considering different types of architectures. In terms of quality, applications and their code need to be as small as possible, as quick as possible, as portable as possible, and finally they need to be power efficient. To achieve these goals developers need a feedback about the execution and behavior of their applications. With this information various improvements to the code can be made.

Profiling is a process of collecting relevant data for analyzing program execution and behavior. There are different techniques for collecting the data including code instrumentation, hardware interrupts, instruction set simulation, etc. One rely on hardware support and require dedicated hardware for measuring and gathering information about executed applications [1], while others are based on software implementations inserting necessary code into the original program [2] [3].

In the embedded world there are many approaches describing various profiling techniques. A well-known profiler like GProf [3] uses code instrumentation technique, which requires the change in program flow for invoking the monitoring routine. However, this overhead can sometimes be significant and can affect the results. In paper [4] it is proposed in which way the cost of instrumented code can be lowered. On the other hand, Intel VTune Amplifier [5] performs time and event based sampling of examined applications in order to collect relevant data.

While the most of the existing profilers tend to support single-core architectures, others can perform profiling on multi-core systems [6] and analyze parallel executions. Apart from that the type of the profiler is determined by the kind of information it should acquire. Usually the timing of the execution is required and source code level analyses have to be performed. In this paper we present a different approach where timings are not essential, but the details about machine code execution and relation between the cores activity in the multi-core system. Proposed technique relies on recording program counter values during step-by-step execution. This is very similar to sampling method which records program counter values at certain time intervals. Recorded values are examined using debug information provided during the build process of the program in order to retrieve execution paths. Disadvantage of this approach is that it is time consuming. However, it collects data with great precision, and does not include any instrumentation code in the examined application nor the need for additional hardware profiling support.

Profiling tool presented in this paper is based on Eclipse plug-in and resides as part of integrated development environment. It is used for profiling applications on heterogeneous multi-core system on chip (SoC). The tool is used for gathering information about the execution and behavior of the DSP applications, processing the gathered information, generating profiling reports, and displaying collected data to the user in simplified way.

II. PROFILER

Proposed profiling tool is implemented as part of a framework for development, deployment, debugging and control of DSP applications [7] [8]. The framework consists of an Eclipse-based integrated development environment (IDE) and a TCP debug proxy server used for

978-1-4673-7967-0/15 $31.00 © 2015 IEEE
DOI 10.1109/ECBS-EERC.2015.31

IEEE computer society

communication between the client side (IDE) and target side (virtual platform-simulator or actual hardware). The debug proxy server enables users to run applications on the desired target and control their execution by interacting with the IDE. User actions like: deploying application, stepping through source code, starting the debug session, reading register values, displaying memory contents, etc., are translated into TCP protocol messages and sent to the debug proxy server. After receiving request messages the server then performs requested actions on the target device. When the action is performed the response message or the result is returned to the IDE and to the user. The supported target devices can be:

- single core virtual platform (simulator of a single core system)
- multi-core virtual platform (simulator of a multi-core system)
- single core DSP
- multi-core DSP

Our target device was a heterogeneous multi-core DSP, consisting of five cores: two numerical acceleration DSPs and three general purpose DSP cores. One of the general purpose DSP cores is a micro-controller which is used for controlling the whole system. Our approach is based on a simple technique of tracing the execution of each core in the multi-core system by logging program counter values, as well as monitoring the cores' states. Combining collected data and debug information of the analyzed code a profiling report is generated. Generated report can then be viewed and analyzed in the profiling view. In order to collect mentioned data debug session needs to be active. During the debug session user is able to initiate the collecting of profiling data and to stop it at any time. However, this can lead to inaccurate results thus it is advisable to place breakpoints around the code which needs to be analyzed. When the "Start profiling" action is performed, debug proxy server starts collecting relevant data until a breakpoint is reached, or the user interrupts this process. During this profiling process, debugging of the examined application is disabled. If a breakpoint is not placed anywhere in the code, user can always choose to stop the profiling by performing the "End profiling" action. When the profiling is finished it is possible to generate the profiling report and to view its contents in the profiling view.

III. PROFILING

The profiling process can be divided into three stages:

A. Generating debug information of the examined application
B. Collecting relevant data
C. Generating report

A. Generating debug information

Creating debug information is essential for our approach. It contains details about the source code, as well as the machine code, generated during the build process. Some of the information of our interest is listed below:

- list of defined symbols in the source code
- information about symbol type (function or a variable)
- origin of each generated machine instruction
- address of each machine instruction

A tool-chain provided by the manufacturer for the examined architecture provides the ability to generate debug information in XML format. We have used this feature and unmarshalling technique of the JAXB tool [9] to read and store necessary information in a simple way. This is done upon start of the debug session and is used for different plug-ins and views in the IDE and for generating the profiling results.

B. Collecting the data

In order to collect relevant data about code execution user needs to start a debug session and choose the "Start profiling" action mentioned before. This action turns the IDE and the debug proxy server into profiling mode during which profiling data is collected and debugging disabled. When profiling is initiated the debug proxy server starts to single step the target device, performing multi-core debug command "step all" which synchronously steps all cores. After each step, debug server reads the program counter register of each core and writes the read values into the log file. Along with PC values debug server also stores information about the core state, whether a core is in reset or not. Apart from program counter values and cores' states system cycle count is also stored. All these information is later on used for generating profiling report and populating the profiling view. The profiling mode is active until a breakpoint is reached, or the user interrupts it with "End profiling" action.

C. Generating the report

When the debug proxy server finishes collecting data it is required to transform it into readable format. By performing "Generate report" action, log file is read and its contents are combined with debug information to generate various reports. Reports can be displayed in the profiling view which is implemented by using BIRT plug-in [10]. One useful feature of the BIRT plug-in is the option of exporting all report types to various file formats like doc, xls, pdf, and csv, which makes user able to view the reports in other types of viewers.

The profiling view consists of:

- Call graph view
- Function statistics view
- Instruction histogram view
- Core activity view

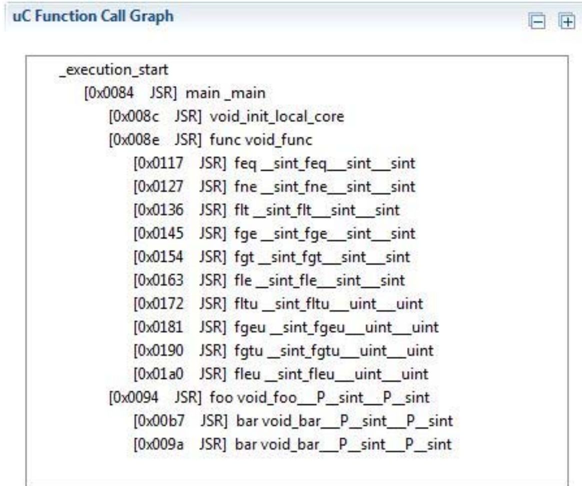

Figure 1. Call graph view

Call graph is created based on logged core's program counter values. Values represent addresses of machine instructions which have been executed during the profiling process on each core. By using debug information we have implemented a simple algorithm that analyzes the execution flow and creates a call graph based on executed jump to subroutine instruction. Figure 1 shows an example of a call graph generated with our profiler. The calls to the subroutines are prefixed with the address of the call instruction. During the debug session this address can be double clicked which results with opening the code editor and selecting the line with that call instruction.

Function statistics view displays details about each source level function: name of the function, where the function resides in memory, how many times function has been called, which machine instructions belong to which

_main			
	Low PC	134	
	High PC	162	
	Size in PMEM	28	
	Call count	1	
	Cycle count	13	
	Instruction count	13	
PC	Assembly	Exe. count	Source references
0086	LOAD 0,A0	1	main.c@293
0087	BITSET 52,A0	1	sdk_uc_clock_reset.h@20268
0088	STORE A0,E[61446]	1	sdk_uc_clock_reset.h@20269
008a	NOP ## NOP	1	sdk_uc_clock_reset.h@20270
008c	JSR 162	1	main.c@294
008e	JSR 273	1	main.c@296
0090	LOAD 256,PX0	1	main.c@297
0092	LOAD 257,PX1	1	main.c@297
0094	JSR 175	1	main.c@297
0096	LOAD 256,PX0	1	main.c@298
0098	LOAD 257,PX1	1	main.c@298
009a	JSR 164	1	main.c@298
009c	IDLE 0	1	main.c@302

Figure 3. Function statistics view

function, and how many times each machine instruction has been executed. All this information is generated by analyzing the debug information and logged data. An example of this type of the report is displayed in Figure 2.

Instruction histogram view is populated with machine instructions of the examined application and number of their occurrences during the execution. Information about the amount of memory access instructions can be used for power consumption and performance analysis. Instruction histogram is shown in Figure 3.

Finally, the core activity view uses information about system cycle count and changes in core states in order to graphically display the comparison between cores' activity. Low-power devices use various techniques in order to reduce power consumption. One of those techniques is clock gating which saves power by disabling parts of circuits [11]. In multi-core systems, core clocks are usually set to automatic gating keeping their clocks disabled while the core is in idle state. It is not uncommon to have one or more cores inactive at certain point of the multi-core processing which makes the power consumption analysis

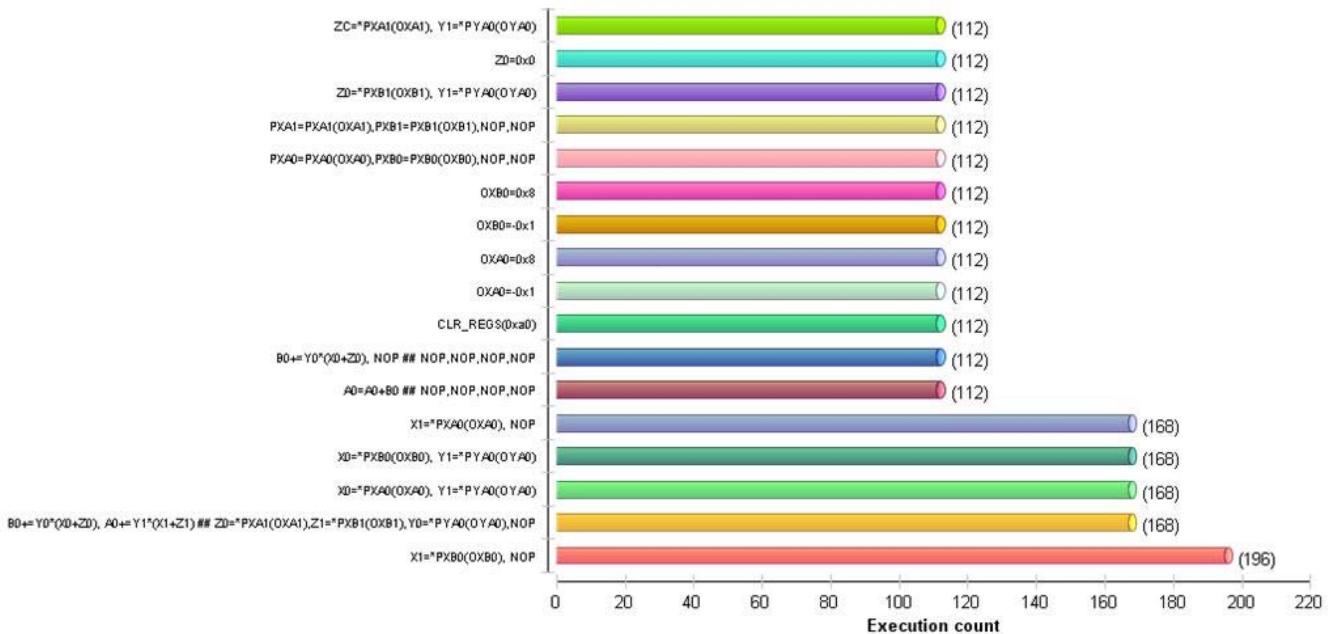

Figure 2. Instruction histogram view

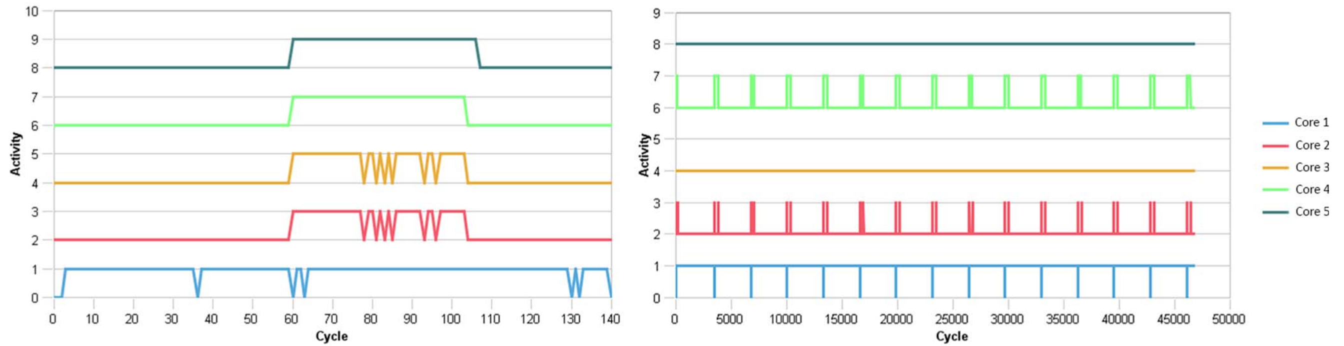

Figure 4. Multi-core activity view

of the whole system difficult. The purpose of core activity view is to simplify this analysis and to give more details about the cores' activity. Figure 4 displays two different activity reports. Report on the left includes profiling of several dozen of cycles, while the report on the right includes around 45000 cycles. The vertical axis values describe whether the core was in active or inactive state. Values 0, 2, 4, 6, and 8 represent inactive states, while 1, 3, 5, 7, and 9 represent active states for cores 1 to 5 (core 1 is the micro-controller core).

IV. Conclusion

We have presented a profiling tool for heterogeneous multi-core systems which maturity is proven by exploitation in industry. The tool generates information about source code function details, call graph, machine instruction occurrences, and multi-core activity which can be used to further improve the quality of the code, to simplify power consumption analyses, and to ease the process of debugging. The profiler is an ongoing project and we tend to advance its functionality and to add new features. One of the main features to consider is adding a support for performance measurements, which would require implementation of timing mechanism. This would further improve analysis of the code execution.

Acknowledgment

The paper is a part of the research done within the Grant TR 32030. The authors would like to thank to the Ministry of Education and Science of the Republic of Serbia.

References

[1] J. Dean, J. E. Hicks, C. A. Waldspurger, W. W. Weihl, and G. Chrysos, "ProfileMe: hardware support for instruction-level profiling on out-of-order processors," in *, Thirtieth Annual IEEE/ACM International Symposium on Microarchitecture, 1997. Proceedings*, 1997, pp. 292–302.

[2] "Valgrind" http://www.valgrind.org

[3] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "Gprof: A Call Graph Execution Profiler," in *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, New York, NY, USA, 1982, pp. 120–126.

[4] M. Arnold and B. G. Ryder, "A Framework for Reducing the Cost of Instrumented Code," in *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, New York, NY, USA, 2001, pp. 168–179.

[5] "VTune amplifier" https://software.intel.com/en-us/intel-vtune-amplifier-xe

[6] S.-H. Hung, C.-H. Tu, and T.-S. Soon, "Trace-based performance analysis framework for heterogeneous multicore systems," in *Design Automation Conference (ASP-DAC), 2010 15th Asia and South Pacific*, 2010, pp. 19–24.

[7] M. Krunic, N. Cetic, M. Djukic, I. Povazan, and M. Popovic, "Integrated Development Environment for Multi-core Systems," *Teh. Elektrotehnika*, vol. 69, no. 5, pp. 818–826, 2014.

[8] M. Krnjetin, B. Rankov, M. Djukic, and V. Kovacevic, "Implementation of a universal framework for deployment, debugging and control of applications on DSP targets," presented at the 18th Telecommunications forum TELFOR 2010, Belgrade, Serbia, 2010, pp. 1289–1292.

[9] "JAXB Project" https://jaxb.java.net/

[10] D. Peh, N. Hague, and J. Tatchell, *BIRT: A Field Guide*. Addison-Wesley Professional, 2011.

[11] C. Leech and T. J. Kazmierski, "Energy Efficient Multi-Core Processing," *Electron. ETF*, vol. 18, no. 1, p. 3, Jun. 2014.