From Hardware Trace to System Knowledge – Data-intensive Hardware Trace Analysis

Andreas Gajda, Rainer G. Spallek
Department of Computer Science
Technische Universität Dresden (TUD)
Dresden, Germany
{Andreas.Gajda|Rainer.Spallek}@tu-dresden.de

***Abstract*—The capture of large amounts of hardware trace data by recent hardware trace units in System-on-a-Chip (SoC) defines the new requirements for hardware trace analysis. Several hundreds of MiB need to be processed in near real-time. The requirements on analyses are further increased with long-term trace capture and increasing frequencies of SoC. Hence, hardware trace analysis has become a data-intensive computing task. The article proposes an approach based on data, functional and pipeline parallelism in combination with data stream processing. As this is a 'work in progress', we will only outline the approach and the preconditions and decisions leading to the approach.**

***Keywords*-data-intensive computing; large data streams; data-processing pipelines; hardware trace analysis; analysis framework**

I. Introduction

Embedded Systems, also called System-on-a-Chip (SoC), facilitate multiple system units on a single chip. Current SoC consist of multiple processors, I/O units, graphic accelerators and highly specialized data processing units. Because most embedded systems cannot be easily debugged in-situ, vendors use special hardware units to gather information, so-called traces, of the system's behavior.

These hardware trace units are configurable devices, that record parts of the hardware context data and publish the records as a packet stream via dedicated hardware interfaces. Capturing the trace data influences the runtime behavior of the processor to varying degrees, depending on the kind of hardware unit. If the runtime behavior is not influenced by the trace capture, the trace capture is called *non-intrusive*. The captured context data contain the control and data flow, event counter data, and ownership identifier.

Trace data provide valuable insight into the dynamic behavior of the system, because non-intrusive data collecting in hardware concurrently with the execution represents the original system's behavior, which might have strict constraints on runtime. And it yields more detailed information about system events: the location, time, and order of their occurrence.

However, hardware trace units produce a great amount of data. These data must be transferred outside the chip by low bandwidth interfaces. This problem is addressed by the techniques of on-chip filtering, compression, and widening the bandwidth of trace interfaces. Nevertheless, observing the system's behavior over long periods of time results in a large amount of trace data. With future prospects of approximately 10^9 bytes of trace data to analyze, the recent trace data analysis, designed for small time periods, needs new approaches to yield results shortly after the data is captured.

The basic idea of our approach, is to preserve the natural format of a *trace stream* and apply all analyses as a *stream transformation*. The analyses aim to be highly parallel, because they are decomposed and arranged by data, function, *and* pipeline parallelism.

Of course, this requires the trace to express software level concepts that cannot be solved in current hardware trace formats. Software trace formats can carry this information, but their ability to carry hardware information is limited.

We will describe some distinctions and similarities of both trace formats (Section II), to motivate their combination into one trace format. The requirements of trace analysis (Section III) and the properties of a trace analysis framework (Section IV) will be described in order to provide a rough view of the idea.

II. Characterization of Trace

The properties of hardware (HW) and software (SW) trace are summarized in the format, content, and amount of the trace. We refer to the IEEE-ISTO 5001™ [1] standard (NEXUS) and Open Trace Format [2] (OTF) for examples of hard- and software trace. The same properties can be found in other formats as well (e.g., [3], [4]).

A. Trace Format

Hardware traces [3] are transmitted out of chip via data and signal lines. The data are submitted sequentially, such that they can be packed into a stream format, consisting of packets of variable length. The NEXUS standard already defines packets as ther basis of its message system. Software trace is written to file by instrumented software and can have several distinct formats, e.g., unstructured files, structured log files, or packet based software trace formats[1]. For comparison, we will only focus on the latter.

In packet based trace formats, the packet consists of a fixed size header and a variable packet payload. For some packets, the payload size is zero, because the header already

[1]In the literature, hardware trace packets are often referred to as 'messages', whereas software trace packets are referred to as 'records'.

Copyright (c) IARIA, 2012. ISBN: 978-1-61208-188-5

TABLE I: Comparison of not-expanded raw (NEXUS) trace data and the same data expanded without (UTF) and with inherent compression ($\mathrm{UTF_c}$) for the AutoBench benchmarks of the EEMBC [5] benchmark suite.

Benchmark	**Covered Ticks (Million)**	**Size (MiB)**			**Emit Volume (Byte per Tick)**			**Expansion Factor**	
		NEXUS	UTF	$\mathrm{UTF_c}$	NEXUS	UTF	$\mathrm{UTF_c}$	UTF	$\mathrm{UTF_c}$
a2time	9.18	27.44	111.95	63.37	3.14	12.79	7.24	4.08	2.31
aifftr	28.86	88.47	440.99	178.50	3.21	16.02	6.49	4.98	2.02
aifirf	9.32	27.84	114.09	64.27	3.13	12.84	7.23	4.10	2.31
aiifft	27.93	85.53	424.58	172.10	3.21	15.94	6.46	4.96	2.01
basefp	9.19	27.56	114.46	63.34	3.14	13.05	7.22	4.15	2.30
bitmnp	9.20	27.20	110.94	63.38	3.10	12.64	7.22	4.08	2.33
cacheb	9.18	27.59	112.46	63.11	3.15	12.84	7.21	4.08	2.29
canrdr	9.22	26.45	112.90	64.14	3.01	12.83	7.29	4.27	2.43
empty	9.21	27.52	112.19	63.59	3.13	12.77	7.24	4.08	2.31
idctrn	9.22	28.04	119.62	62.62	3.19	13.60	7.12	4.27	2.23
iirflt	9.27	27.73	113.37	63.97	3.14	12.82	7.23	4.09	2.31
matrix	156.91	596.35	2447.58	1013.95	3.99	16.36	6.78	4.10	1.70
pntrch	9.20	26.96	110.10	64.00	3.07	12.55	7.30	4.08	2.37
puwmod	9.19	25.91	113.97	64.70	2.96	13.00	7.38	4.40	2.50
rspeed	9.22	27.21	112.89	63.96	3.09	12.84	7.28	4.15	2.35
tblook	9.17	27.36	112.41	63.49	3.13	12.86	7.26	4.11	2.32
ttsprk	9.23	26.93	113.43	64.25	3.06	12.88	7.30	4.21	2.39
average	20.16	67.77	288.11	132.75	3.36	14.29	6.59	4.25	1.96
median	9.22	27.52	113.37	63.97	2.98	12.30	6.94	4.12	2.32

carries all of the information. Trace formats typically utilize several kinds of compression, summarization, and conditional information. Conditional information is indicated by length fields or signaled by bits in the packet header and/or packet payload.

B. Trace Content

Most packets can be categorized into ‘control flow’, ‘data flow’, ‘event counter’, and ‘run/trace control’. Hardware trace may contain additional packets for the manipulation of register and memory content. The NEXUS distinction of ‘ownership trace’ and ‘device ID’ is included in the categories of ‘control flow’ and ‘data flow’. Software trace may provide additional packets for concepts above the software level, like peer to peer and collective communication.

Hardware trace content consists of memory and instruction addresses, device and thread identifiers, timestamp and tick counter information, memory content, break- and watchpoint events, and synchronization events.

Software trace content consists of numerical or text identifiers of processes, functions and variables, accessed variable value and performance counter value. Usually the content is ordered with respect to the execution, but timestamps or tick counter value provide an inherent (weak) ordering of concurrent processes and additional timing information for functions. Some trace formats allow global or interval based summaries.

C. Trace Amount

All formats support some kind of in-stream compression to reduce the bandwidth needs for transferred data.

In hardware trace streams, the used methods comprise transmitting differences to reference value, leading zero suppression, sampling and tagging of repeated data. Some hardware trace units use an on-chip filter, to limit the location and time of observation to points of interest. Software trace uses data reduction analysis, e.g., smart compression algorithms and sampling.

The size of a single packet in both formats is small and limited, but recording billions of events sums up to a few hundred MiB. See Table I for a few examples of NEXUS trace file sizes. Typically, hardware trace records a fine grained execution trace of the processors in a few seconds, and software trace records a coarse-grained program execution in minutes or hours.

The amount of the trace stream grows with the observation time and depth of analysis. One important use of hardware trace units will be the constant monitoring of the SoC, so-called long-term or endless trace, especially for near real-time analysis. Once available, the amount of data to process requires more computing power to extract the knowledge hidden in the data.

Experiences from working with the EEMBC [5] benchmark (Table I) show the median trace file size reaching 28MiB raw trace data for roughly 100ms of a single benchmark run. However, trace data volumes up to 600MiB were also retrieved. These trace data are compressed by any measures available in the trace format, including a branch based representation (i.e., *not-expanded trace*) of the control flow. When expanding the trace data to a single instruction based control flow trace (i.e., *expanded trace*), the expansion factor is about 4, which means dealing with about 115MiB of trace data.

Based on these data, transferring and analyzing the raw trace stream in near real-time, requires approximately 315MB/s bandwidth and data processing power for a single 100 MHz embedded target processor.

III. Hardware Trace Analysis

Trace stream analyses consist of filtering and analyzing the content. Filtering is the reduction of the trace events to a small

Copyright (c) IARIA, 2012. ISBN: 978-1-61208-188-5

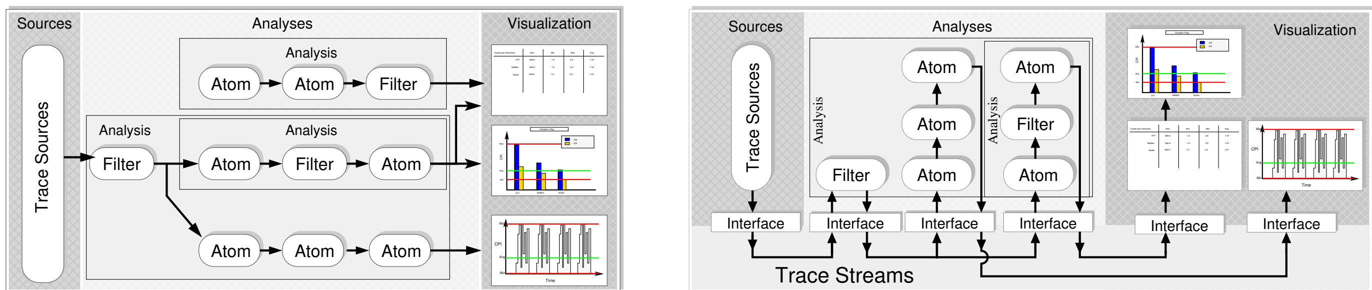

Fig. 1: Vertical and horizontal view of the analysis framework. The views are simplified, by not showing the interfaces between the atoms within an analysis. The arrows point in the direction of the data flow.

window of interest. This process is done at a minimum of two times: On-chip filter restrict the trace capture to code ranges, data ranges, or other user specified trigger events. These filter are set-up *pre-trace* versus *post-trace* filter, which reduce the amount of captured trace by suppressing trace packets. Post-Trace filter analyze the trace with similar criterion like pre-trace filter, but mostly more specific to the follow up analysis. Some filters are analysis inherent, like restricting the analysis to a certain kind of packets, e.g., those for data flow trace.

The analysis of hardware trace content maps the raw trace data back to different levels of the executing environment abstractions. For instance, analyses reconstruct the control or data flow at the instruction level, analyze the call graph at the function level and observe the scheduling behavior at the system level. Analyses may be built up on other analyses too, like measuring function execution times and calculating the statistical values afterwards. The structure of intermediate results might change, like reconstructing the instruction control flow from raw trace and mapping it onto control flow graphs. Intermediate results might be used multiple times, like creating the static and dynamic call graph from the software level function trace.

All (intermediate) results need appropriate structures and storage management, which provide fast access and large storage. Moreover, the intermediate and final results of the analysis need to be stored externally, e.g., for creating checkpoints or comparison with the results of other traces.

IV. Trace Analysis Framework

This section introduces the trace analysis framework. We will describe the framework and the decisions that lead to the structure of it, on an abstract level. We will introduce the aspect of analysis parallelization, the internal structure of the framework, and the internal trace format and provide the current state of implementation.

We call it ‘trace analysis framework’ to emphasize the aim of hardware and software trace analysis, using the same generic structure.

A. Parallelization

As shown before, the challenges of hardware trace analysis result mostly from the amount of data to process. Furthermore, since analysis is an interactive process, the period between trace capture and visualization of the analysis results must be short, since the developer might need to refine or restart the trace capturing process. Displaying intermediate results gives additional feedback of the analysis progress. In addition, the results from long-term trace are expected to be available *before* the trace has finished. Therefore, the trace analysis should be near real-time at least.

To achieve a speedup in data processing, the analysis task needs to be parallelized. The fundamentals of parallelization are the decomposition of data and function and the concurrent use of processing units to execute (different) functions on (different) data. Since all possible tasks for hardware trace analysis need to be decomposed for that purpose, this task might provide enough work for years.

Another approach is to create a parallelized analysis framework, which provides the means for complex analysis and the inherent decomposition of the analysis itself.

B. Framework Structure

The analysis framework (see Figure 1) shall exploit functional and data parallelism as much as possible. The analysis process is broken down into the functional units of the trace data source, analyses, and visualization. All the data exchanged between these components are transferred as trace data streams. All the data are encapsulated in one trace format, which is designed to host the hardware trace data and the analyses results at once.

The analyses are decomposed into atomic functions, which are also connected via trace data streams. Each atomic analysis has a minimal functionality. The atoms are combined into a complex analysis by connecting them via trace streams, where the output of an atom serves as the input to several other atoms. The analysis atoms might filter, transform, or enrich the trace stream, such that the output of the analyses can be visualized.

Since the atoms depend only on each other via their interconnections, the whole framework is highly parallel. The ability to duplicate the trace stream and use different atoms at once, using the same input data, provides a functional decomposition of complex analysis. The ability to duplicate and enrich the trace stream, provides additional data decomposition. Both kinds of decompositions are arranged via the

Copyright (c) IARIA, 2012. ISBN: 978-1-61208-188-5

network, which is generated by the trace stream connections between the atoms.

The acceleration of the analysis is expected to come from the concurrent execution of different analyses at once. The input trace stream and any intermediate trace data can by duplicated and serve as input to several subsequent atomic analyses, which extract the different properties of the data concurrently. The independence of the atoms allows concurrent execution, only with synchronization at the input and output channels.

The visualization of the intermediate results benefits from the concurrent handling of raw, intermediate, and final trace data. If the visualization receives the first result, the first atom still processes trace data from the trace source.

The framework is able to balance the workload by examining the amount of data transferred between the atoms. As long as there is no input for an atomic analysis, the task does not need to be processed. Since this enables the dynamic switching between active and dormant state of atoms, this might lead to further optimizations.

We understand that a complex analysis might not be decomposable, or at least not without introducing significant overhead. In this case, the analysis can be treated as an atom itself, without the loss of generality of the framework.

C. An Analysis-oriented Trace Format

Several reasons lead to the design of a trace format for the framework's internal use: At first, there are several hardware trace formats defined and used by different vendors. Supporting all of these formats for all analyses is an expensive task. Second, hardware trace formats are *bandwidth-oriented*. They invoke several compression techniques, reducing redundant or otherwise reconstructible data from the transferred data, to decrease the bandwidth requirements for on-chip trace hardware. While this is very desirable for communication purposes, it is a large overhead creating task for trace analysis, to expand the trace into an analysis-usable format. Third, hardware trace represents events, control, and data flow at the hardware level only, which is fine for the purpose for which they were defined, but makes it impossible to represent software level trace. Software trace formats were designed to represent, e.g., the control flow at the function level and, therefore, lack (efficient) support of control flow trace at the instruction level.

These are enough reasons to create a new *analysis-oriented* trace format, which

1) is structurally compatible with the existing hardware traces,
2) provides a framework to host analysis results,
3) provides a framework to host a software level representation of hardware level events, and
4) has a low overhead impact on analysis tasks.

We call it *'Universal Trace Format' (UTF)*. It is packet based, provides a means for the representation of control and data flow at the hard- and software level, multidimensional event counter and control packets for trace formatting, messaging and other events. The definition space for new packets is still extendible, such that new concepts can be introduced in later versions.

Since we expect trace stream transfers to the hard disk and over networks, the trace format includes an inherent lossless compression, which reduces the transferred number of bytes. To avoid a large computational overhead, the compression algorithm is kept simple. Table I shows the reduction of the trace file sizes, when using the compressed trace format.

D. Work in Progress

The framework exists in an experimental stage, with a few atoms and analyses implemented. In this stage of implementation, we focus on the proof of concept. While we are already testing complex trace analysis, no results are published yet. The first implementations focus on instruction level analysis, including instruction covering, control flow analysis, and timing analysis. The next steps include block level analysis such as call stack and call graph visualization.

Looking beyond this framework, the visualization of the results needs to be included. To date, there are only textual representations of the trace data, but we know the importance of the graphical representation. We imagine a generic framework, which will be fed from the trace data and provides generic visualizations of flow graphs, counter time lines, diagrams, and animations.

V. Conclusion and Future Work

This paper has provided a view of our parallel trace analysis framework. We presented the preconditions of the analysis and the resulting conclusions. Hardware and software trace properties were explained and the implications for the combination into a single trace format given. The internals of the framework were outlined and an outlook of future development directions were given.

The framework still has to prove its usefulness, but we expect performance results soon. Further research in parallel trace analysis and performance improvement is possible and encouraged. We also imagine extending the framework for analyses at abstract levels, e.g., the 'runtime verification' in the field of model checking.

VI. Acknowledgments

The authors gratefully acknowledge the financial support of the European Union and the Free State of Saxony.

References

[1] (2011, Oct) Nexus 5001 Forum™ Standard. Website. [Online]. Available: http://www.nexus5001.org/standard
[2] (2011, Oct) Open Trace Format – Homepage. Website. [Online]. Available: http://www.tu-dresden.de/zih/otf
[3] C. MacNamee and D. Heffernan, "Emerging on-ship debugging techniques for real-time embedded systems," *Computing Control Engineering Journal*, vol. 11, no. 6, pp. 295 –303, dec 2000.
[4] (2011, Oct) Introduction into ParaVer. Website. [Online]. Available: http://www.bsc.es/ssl/apps/performanceTools/files/docs/intro2paraver_MPI.tar.gz
[5] (2011, Oct) The embedded microprocessor benchmark consortium. Website. [Online]. Available: http://www.eembc.org