Systematic Modeling of Workflows in Trace-Based Software Debugging and Optimization

Salman Rafiq and Adriaan Schmidt
Fraunhofer Institute for Embedded Systems and Communication Technologies ESK
Munich, Germany
Email: {salman.rafiq, adriaan.schmidt}@esk.fraunhofer.de

***Abstract*—Tracing is a tool frequently used in the debugging and optimization of software. While there exist different tracing solutions, each of them comes as a tightly coupled trace collection, analysis and visualization bundle, and thus, it can only be used to answer a narrow range of questions. Due to this limitation and the complex nature of software workflow in the embedded domain, we believe that tracing and the analysis of traces have to be flexible and extensible. In this paper, we propose a methodology of trace processing. We introduce a generic model of describing traces and operations that are performed on them, irrespective of the tracing solutions being used. Also, with the help of our model, one can describe new processes and workflows that involve trace data from a combination of sources. To present the use of our methodology, we systematically model four use cases that solve complex debugging and analysis tasks. At the end, we show how one of these use cases fits into a modular framework using a prototype implementation.**

Keywords*—*Tracing; trace-processing; workflow modeling; debugging; multicore.

I. Introduction

With the ever-increasing complexity of hardware and software systems, the task of programming and maintaining software has become more and more challenging. This is especially true when considering parallel systems, i.e., multicores and Systems-on-Chip (SoCs). To debug and optimize these systems, classical debugging tools and methods are often insufficient.

Tracing, that is the recording of data on the dynamic behavior of a software system, has been introduced with great success in some problem domains, e.g., performance optimization in High-Performance Computing (HPC) or debugging of embedded and real-time systems. So far, however, existing trace solutions cover only few specific use cases at a time. The technologies of trace collection and visualization are customized to these cases. In HPC, tools are specialized to deal with highly parallel programs, typically using the message passing model of programming. Trace analysis and graphical visualization is tailored to the task of performance optimization of such applications. In the area of embedded computing, we will find tools that can record a system's execution with hardware assistance, at a cycle-accurate level, without changing the timing behavior of the target. This data is then used for debugging and analysis purposes. Due to its non-intrusive behavior, this method is suitable for debugging timing related issues in real-time systems.

Existing trace solutions, such as [1, 2, 3], collect a vast amount of data, which is then processed and presented to the developer. However, these products mostly come as integrated solutions, tightly coupling trace collection, analysis and visualization.

With the afore-mentioned increased complexity in systems, we believe that tracing and the analysis of traces need to be flexible and easy to handle and extend. For instance, tools have to be extensible to fit complex debugging and optimization tasks. Keeping these properties in mind, we introduce a methodology to describe traces and operations on them. This methodology can be used to model different elements of a trace-based analysis and debugging workflow. Moreover, it provide ways to model complex processes that use the trace data from different tools or sources. We also present how different debugging and analysis use cases can be efficiently modeled using our methodology. By modeling a use case, dependencies between different trace data involved, and the interfaces between different tools become obvious, which helps during concrete implementation. At the end, we show how one of the modeled workflows can be mapped on to a modular framework, as part of our prototype implementation.

Previously, there has been some work on languages describing event traces. Auguston [4] suggested FORMAN language, which is used to describe computations over event traces. It uses an event grammar to define intended program behavior during debugging or testing of programs. Boroday et al. [5] presents a generalized formal framework to model event traces in a distributed system. Another language called Tiddle is proposed by Sadowski and Yi [6] to test dynamic analyses by generating concurrent benchmarks. However, these languages do not provide a way to define complex analysis workflows, that involve trace data from different sources. Also, these modeling techniques cover specific use cases, e.g., communication, or concurrency bugs, rather than being generic and scalable to other use cases as well.

Visualization tool by McGavin et al. [7] describes a methodology to explore large sets of execution traces. It gives control to the user to filter events and get on-demand information related to a particular object, after loading all the trace data into the tool. Whereas with our modular framework consisting of transformation modules, pre-filtering of trace data can be done before loading it into an output module, i.e., visualization or analysis tool. Hamou-Lhadj and Lethbridge [8] discuss different analyses and visualization tools for Object-Oriented systems, and the possibility to combine features from each tool into a common framework as their future work. The discussion centers on trace exploration and compression techniques to reduce the volume of generated traces.

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-304-9

The remainder of this paper is structured as follows. In Section II, we present a brief introduction to tracing and an overview of the current tracing, analysis and visualization technologies. In Section III, we describe our model and methodologies for trace data manipulation and workflow description. Section IV presents use cases which utilize our model to describe complex trace-analysis workflows. Section V shows the mapping of one of the modeled use case to a modular framework, before Section VI concludes this paper.

II. Survey of Tracing Technologies

Tracing can be seen as directly derived from one of the oldest methods in debugging: the use of print statements to output program state at runtime. The method has, however, evolved, and the use of tracing techniques yields a much more systematic approach to debugging than the simple insertion of print statements. Especially in complex parallel systems, the messages produced by manually printing program state will be hard to read and interpret. This is where tracing solutions offer graphical visualization that helps the developer to comprehend the recorded trace data.

In this section, we present a brief overview of trace-collection methods. We show the basic sources of trace events that are used as inputs to our trace-processing workflows.

A. Instrumentation-Based Tracing

One important method of tracing is instrumentation: additional code is added to the software at points of interest, which causes the target itself to generate event traces. These are then stored or transmitted for later analysis. There are different ways of instrumenting the target software. The most basic is manual insertion of instrumentation code by the developer. Alternatively, the insertion can be automated, often with the help of the compiler. Another class of tools performs dynamic instrumentation of the target application, that is, they change the code at run time.

The recording of event traces with the help of instrumentation can be performed in different components of a system. Aside from the application itself, instrumentation placed within the operating system can produce valuable information on system execution, capturing the interaction of several applications, together with global resources like device drivers.

Instrumentation-based trace techniques are used in different domains, and several specialized solutions are available: in the area of HPC, tools like [1, 2, 3] together with their respective visualization front-ends, are popular to optimize the performance of highly parallel applications. Instrumentation of the operating system kernel is used in debugging, but also on-line monitoring of systems. Solutions are available for numerous platforms, including Linux [9], Windows [10], BSD or QNX [11]. Also for embedded systems, specialized solutions are available, e.g., for low-overhead collection of events for timing analysis [12].

The advantage of using instrumentation to collect trace events is that it allows good control on the type and number of events to be collected. In this way, the volume of the generated trace can be limited to the events actually needed. Manual instrumentation of the application is easy to use, and can access application-specific data. Common examples would be the indication of program states, or the value of internal program variables.

However, instrumentation does influence the run-time behavior of the target application. So, in debugging timing-related issues, the target may exhibit a changed behavior due to the instrumentation, and the results may be useless. Also in systems that are already operating at the limit of CPU utilization may not be suitable for instrumentation, as the additional overhead may render the system dysfunctional.

B. Hardware-Based Tracing

Modern processors implement hardware interfaces that generate execution traces. Here, event traces are generated by the hardware and are typically on a low level, i.e., the execution of single machine instructions. The generated data is transferred off-chip via a high-speed serial interface, e.g., Serial Wire Debug (SWD) [13]. To receive the trace data, usually another hardware device (hardware debugger) is needed that decodes the event stream and transfers the data to a debugger application on the host computer. Examples of such solutions are [14, 15, 16].

A hardware trace contains detailed data on the execution of all software. In contrast, an instrumentation-based trace captures events only in the parts of the system that are instrumented, so it captures events only the developer expected. To also see unexpected events like hardware interrupts or unexpected memory accesses due to corrupted pointers, a hardware trace is much more useful. Another great advantage of hardware-based tracing is that it does not influence the timing behavior of the target system. This means that it is also suitable to debug timing issues in real-time systems.

The transfer of a target-system trace to a host computer can require extremely large bandwidths. If we take the ARM platform as an example, this can lead to a required bandwidth of 1 Gbit/s per core for a simple instruction trace and up to 16 Gbit/s per core for a complete data trace [17]. When considering multicores and SoCs with many on-chip trace sources, one quickly sees limitations in the amount of trace data that can be transported from the target to the development host. Obtaining a complete execution trace of a multicore processor is at the least challenging, and often impossible due to bandwidth limitations.

Realizing this, hardware vendors have started the integration of more-flexible tracing logic into their chips. Examples are ARM CoreSight^{TM} [18], or Infineon's MCDS [19]. With its help, it is possible to program flexible triggers and filters, and thus reduce the volume of the trace data and the bandwidth needed to transfer it.

While typically hardware-based trace solutions capture data from the processor cores, and thus data specific to the software being executed, other devices can be traced as well. Examples include traces of on-chip buses and interconnects [20] or peripheral devices [21].

C. Other Data on Dynamic System Behavior

Apart from traces collected using dedicated software or hardware solutions, there exist many other useful sources of

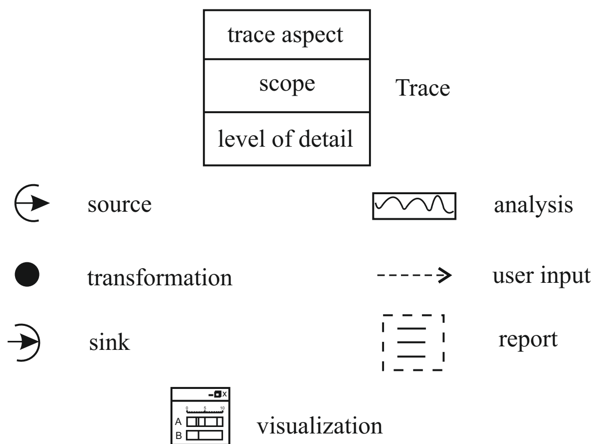


Figure 1: Elements used in the graphical representation of trace processing workflows.

data on the runtime behavior of a system. Some of them can be useful in an analysis to provide a context in which the software was executed. This is especially valuable when dealing with embedded systems that have close interaction with their surroundings.

Examples are traces of communication links, which can be captured externally using a network monitor or a bus analyzer, or information from external sensors and actuators. Also other existing sources of data present in software can be exploited in trace analysis, e.g., existing log files, providing a high-level view of the system's activities.

III. Methodology of Trace Processing

In this section, we introduce a model to describe traces and operations on them. Our objective is to provide a basis on which to argue about transformations, analyses, and the graphical visualization of trace data. Our approach can be used to describe workflows of trace processing in an abstract way, independent of a particular trace-collection technology. With the help of our methodology, complex applications in the area of trace analysis can be systematically described.

Throughout this paper, we use a simple graphical representation to visualize applications of our model. An overview of elements is shown in Fig. 1. They are described in detail in this section.

A. Trace Features

Traces are the central artifact of our model representation. Our model annotates a trace with three features: the Aspect captured by the trace, the Scope of the trace data, and its Level of Detail. We do not model the details of the trace data itself, or impose any restrictions on trace formats and representations. For the purpose of our model description, a basic definition of traces is sufficient. Thus, we define a "trace" to be an ordered sequence of events. The order may be established by associating a timestamp with each event. Additionally, trace events may contain arbitrary data; again, there is no restriction imposed by our methodology.

We do not provide a formal language for the description of trace features in our model, but rather focus on the high-level representation of traces using plain English. It is up to the developer, how detailed this description should be. In our graphical representation, a trace is depicted as a rectangular box with three fields, one for Aspect, Scope and Level of Detail.

1) Trace Aspect: The trace Aspect describes which properties of the target system are captured in the trace data. It is the most important of our three properties, as it describes the nature of the respective trace. The Aspect of a trace determines which questions can be answered by interpretation of the trace data, and it is often closely linked with the method of trace collection.

Examples of trace Aspects can be "program execution flow, i.e., which instructions or functions were executed at which time", "program state, i.e., the values of variables over time", "packets seen on a network link", "transactions on an internal bus", "performance metrics", and "inter-process communication".

2) Trace Scope: The Scope of a trace describes which parts or which components of a system are covered by the trace. Components of a system may be hardware devices, such as CPU cores or communication interfaces, or software entities, such as applications, threads, or objects.

In trace analysis, it is important that the trace data captures the right scope. For efficient operation, the trace should not contain more or less information than is needed to answer the developer's questions. If the scope is too broad, it may be difficult to grasp the essential information, and if it is too narrow, interactions between several components may be lost from the trace.

As the selection of a Scope for a trace defines which subset of the set of all available events is contained within the trace, it directly influences the volume of the resulting trace. Thus, the trace Scope determines the bandwidth required for transmission of the trace, or the capacity needed for its storage. The possible values of the Scope feature naturally depend on the Aspect captured by the trace. When considering a trace of the program execution flow, the Scope may be, for example, "instructions executed by application A", "instructions executed in interrupt service routines", or "instructions executed on Core n". In contrast, a trace of network packets or protocols may have scopes like "TCP, HTTP, or telnet session", "connection in a client-server scenario" or "communication link".

3) Level of Detail: The Level of Detail (LoD) gives information on the resolution or precision of the trace. It can be described as a set of information captured to fulfill the requirements in achieving a certain trace Aspect.

Similar to the Scope feature, the LoD in a trace depends upon the trace Aspect. Also it influences the trace volume as in the case of a Scope. Carefully choosing the LoD can simplify the trace collection, transmission, and storage in cases where the lower LoD is tolerated by the analysis in question.

For example, a trace Aspect "execution flow" may have LoD like "complete set of instructions", "only branches", or "only function entries and exits". On the other hand trace Aspect "data values" can have LoD like "modifying a certain memory

location", "reader or writer process/thread", or "reads/writes within a specific region of the code".

B. Basic Trace Transformations

A trace transformation is a step of processing, which takes one or more traces as input, and based on them generates one or more traces as output. Within our model, a transformation always affects at least one of the three trace properties.

The effects of some operations are easy to capture and understand. For example, filtering operations will usually either narrow the Scope of a trace, or lower its Level of Detail. Examples would be the processing of a hardware trace containing all executed instructions through a filter, which reduces the data to include only the executed calls and returns, thus yielding a lower LoD (function entry/exit instead of instruction accurate). Also, a reduction in Scope is common, and can be seen as a filter operation performed on the trace. An example is the filtering of a trace of scheduling events of the operating system, to extract only those events related to a certain application. These two operations are easily implemented as filters, extracting some events from a trace and discarding the others.

Other transformations are more complex. Among them are: broadening of a Scope; which combines traces with distinctive features. This combination of different traces with distinctive features has the potential of substantially increasing the value of tracing. In systems with complex interactions of different components, it is often necessary to combine data from many different sources to track down the origin of a software failure. If this integration of data sources can be performed systematically, as opposed to the developer switching back and forth between tools, debugging and optimization can be greatly simplified.

Changing the Aspect of a trace is another complex transformation, which can require some more-elaborate analysis and processing. A complete trace of program execution could be processed, such that instead of the execution flow of the application it reflects the value of certain variables over time.

In the processing of hardware traces, there is a commonly used transformation that increases the level of detail. The trace unit of modern CPUs compresses the trace data to reduce bandwidth. Usually, only the branch instructions are captured, knowing that from this information the instructions executed between branches can be reconstructed. This step is implicitly performed by the host-side tracing tools.

C. Trace Sources and Sinks

To model trace-processing workflows, it is not only necessary to describe trace transformations, but also to specify where traces come from (their sources) and what eventually happens to them (their sinks). Trace sources are means of trace collection, of which we already described several in Section II. We distinguish two kinds of sinks: visualization and analysis.

The trace sources are the point where traces enter our modeled workflows. The trace-collection method determines the features of the trace at this point. Multiple trace sources can be employed in one workflow, e.g., combining traces that capture different Aspects or Scopes of the same system.

It is important to mention that workflows described by our model can have many processing steps, not all of which will be actually implemented in software. It is possible for some of the steps to be implicitly performed in hardware, e.g., on-chip filtering of trace events. In these cases it is up to the developer, whether this implicit processing step is modeled, or whether the model uses the already filtered data as a trace source.

A visualization presents the trace data to the user, showing the trace events on a time line. On the other hand, an analysis takes a trace as input and generates results in arbitrary form. Their semantics differ from the transformations described earlier, in that their output is not a new trace. Thus, the analysis result, which might take the form of a textual or graphical report, lies outside the scope of our model. Examples of analyses are: timing properties like average execution times of functions or distributions of response times, analysis of lock contention, or analysis of data accesses to detect race conditions.

Both visualization and analysis can present data to the developer in an interactive way. It may be possible to "browse" the trace graphically, scroll, zoom, etc. User interaction can influence the processing inside the modeled workflow. By selecting elements from the analysis or visualization, the user could change the parameters of certain transformations, causing a re-calculation of the results. Such an influence on the processing of traces is represented by a dashed arrow in our graphical representation.

IV. Modeling of Trace-Analysis Workflows

In this section we present four examples, how our methodology can be applied to model certain analysis workflows. In addition to describing complex workflows, our methodology can be utilized to constitute a flexible modular framework. The sources providing trace data as an input, translate into input modules. Also the user interaction becomes part of an input module. The sinks utilizing trace data for visualization and analysis are mapped as output modules. Finally, trace transformations that involve processing of traces are implemented as transformation modules.

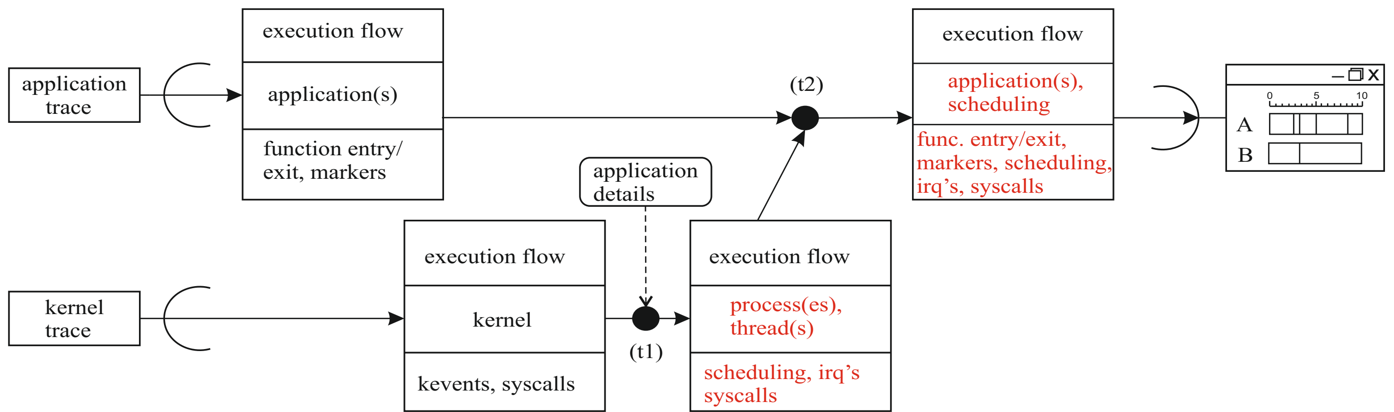
In the following section, we present the prototype framework implementation using one of the described examples. For the rest of the examples, our model documents trace-processing workflows in an abstract way, and does not concern itself with concrete implementations.

A. Combination of Application and Kernel Trace

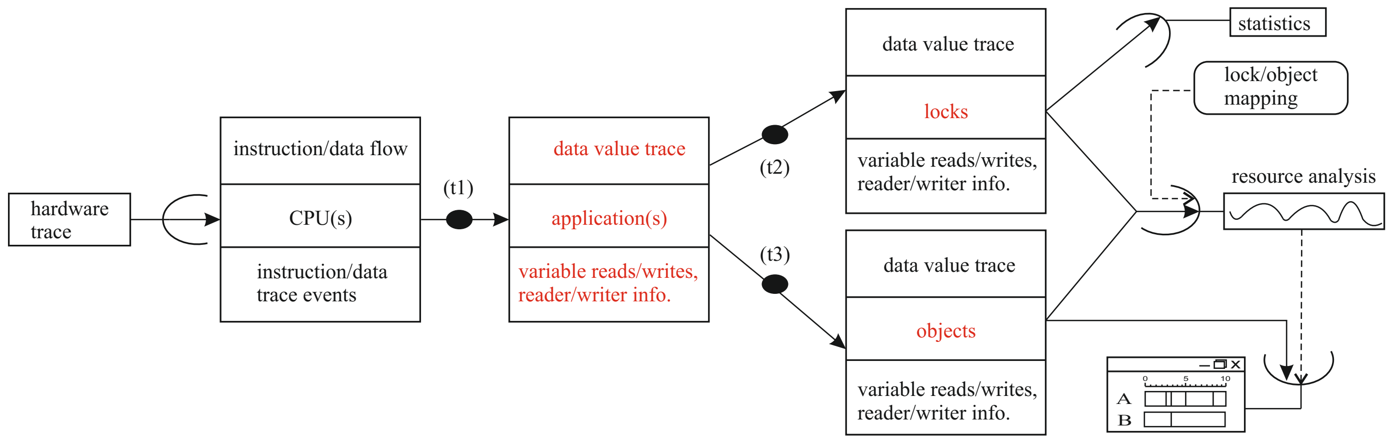
Manually instrumenting a target application is an efficient way of collecting trace data. The developer can select instrumentation points, and can thus easily create a trace containing application state, phases in program execution, and values of important variables. However, one weakness of this approach is that it can capture only the single application that is instrumented. Especially in embedded systems, this is often insufficient, and instead the developer needs a view of the complete system, showing interactions between different applications, and between applications and the operating system.

The first use case of our methodology addresses this issue, by processing separately collected traces from the user application and the operating system kernel, and integrating

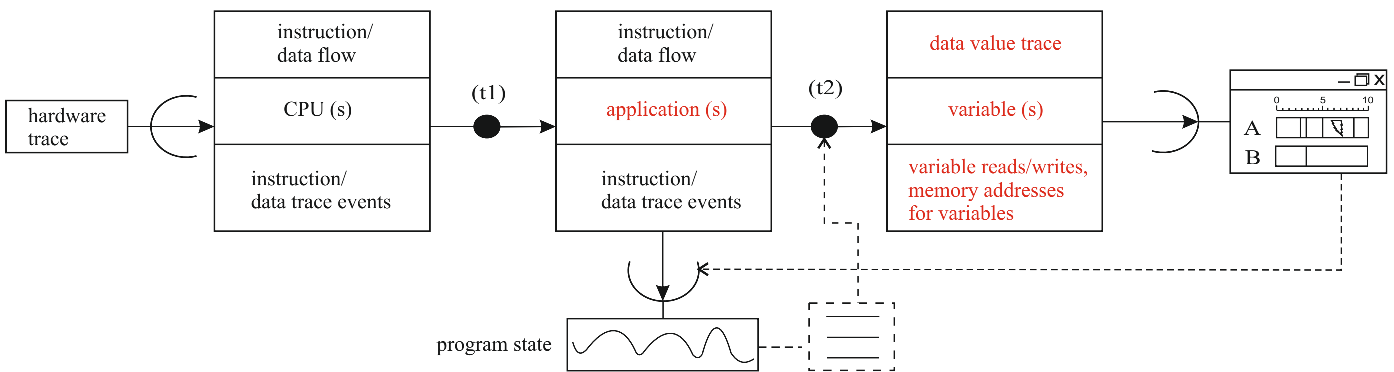
Copyright (c) IARIA, 2013. ISBN: 978-1-61208-304-9



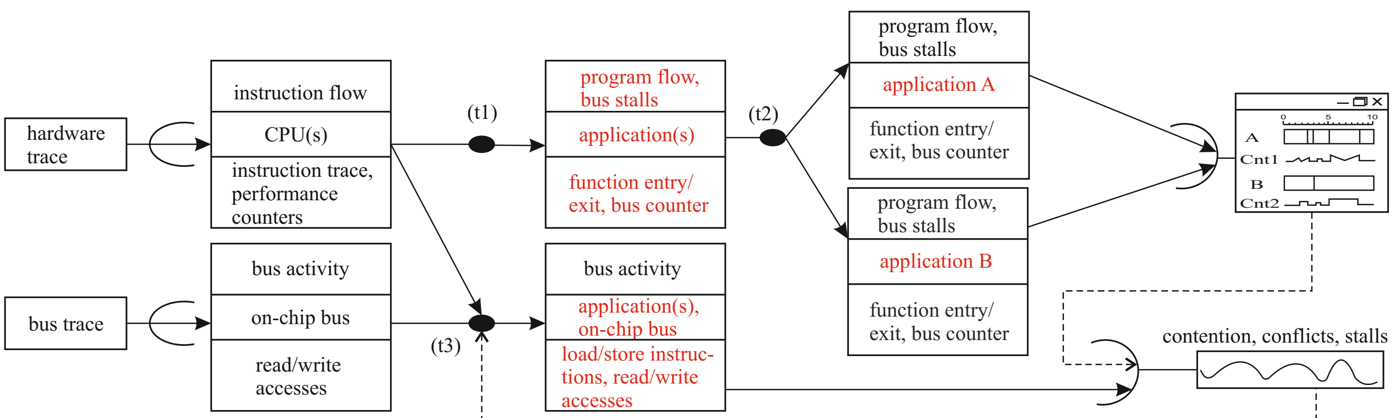
(a) Use case described in Section IV-A. Combination of application and kernel traces.



(b) Use case described in Section IV-B. Modeling of trace features for data-centric analysis.



(c) Use case described in Section IV-C. Modeling analysis workflow of memory related issues.



(d) Use case described in Section IV-D. Integration of on-chip bus trace for enhanced analysis.

Figure 2: Example use cases

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-304-9

them into one combined view. The workflow is shown in Fig. 2a.

The essential transformation (t2) in this workflow performs a “broadening” of trace Scope by combining the execution flow of the application with the relevant scheduling events from the kernel trace. This way, the application’s timing can be displayed in the context of the complete system, and in case of timing anomalies a picture of the system state with the current constellation of running applications and occurrences of interrupts is available.

While the application trace is directly used by transformation (t2), the kernel trace first undergoes some filtering: from a kernel trace containing a multitude of different events, we extract only events related to scheduling, system calls, and interrupts that affect our target application as shown by transformation (t1). This transformation also uses auxiliary application information that helps in mapping of target application to kernel trace.

Note that the trace Aspect remains unchanged by all transformations. We always consider the program execution flow. Also, the Level of Detail is largely kept unchanged. All processing and analysis is done on a relatively high level of abstraction, considering program phases and single scheduling events, but not further details on instructions executed.

B. Extraction of Data-Centric Trace Information

In object oriented programs, it can become useful to keep track of an object’s lifetime. It starts from the object creation, different processes/threads accessing it, interactions with other objects, and finally deletion of the object. Moreover, the inclusion of locks to protected shared objects makes interactions more complex and difficult to analyze. With the help of such an analysis, one can find anomalies in software like data races (simultaneous access to an object with at least one of them being a write operation), locking violations, contention for guarded shared objects, and memory leaks (objects created but never deleted).

Hardware tracing allows the developer to generate cycle-accurate event traces. It can include a complete set of instructions, and reads and writes to memory addresses. This use case of our methodology uses hardware tracing as a source, and reduces the LoD to specific objects and locks with the help of the developer. The workflow is shown in Fig. 2b.

The transformation (t1) filters the Scope of the hardware trace from the CPU to the target application. Furthermore, (t1) also performs an important transformation of Aspect of the trace to “data value trace”. This is achieved by extracting the selected LoD from instruction and data-trace events, e.g., memory reads and writes, and reader/writer process or thread information.

Later, the transformations (t2) and (t3) are performed, generating new traces with similar Aspects as before, i.e., data value trace. These transformations reduce the LoD to specific shared objects and locks inside the application, rather than a complete set of variables. From this point on, the transformed trace can be used as an input to the trace-analysis tool. The user may also provide auxiliary information regarding the relation between a lock and an object to analyze them over time. This mapping helps to find any problems with regard to simultaneous accesses from different writers/readers and the locks guarding them.

The trace data with object as a Scope can then be used as an input to the visualization tool for a graphical representation of the object’s lifetime. The dotted arrow (user input) from the analysis tool to the visualization sink shows that the user can also influence the graphical time line of an object of interest.

C. Backtracing of Memory Issues

Root causes of memory-related faults become more difficult to find, especially when software is running on a multicore system with shared memory resources. Consider a case where an incorrect value is being assigned to a memory location, causing the program to become dysfunctional. It can be of great help if a backward chain of calculations can be analyzed to know the source code location causing that faulty write. To achieve this, it requires the program state to be maintained by the tool for that point in time. This use case addresses workflow-modeling of a similar scenario.

For example, the event traces of specific variables are visualized in a tool showing data values being assigned to them over a time period. The tool allows the user to inquire about the incorrect value on demand. This interaction from the user triggers the backward analysis by inquiring the program state being maintained by the analysis tool. Finally, an interactive report provides the instruction writing the wrong value to the variable. Fig. 2c explains the steps involved to model such an analysis with the help of a hardware trace source.

It is important to mention that transformation (t1) may be implicitly performed by configuring a hardware trace-recording tool to record events related to a single application only. In this case the model will directly contain trace data with “application” as the Scope. Information like memory addresses, data reads/writes, and program-counter values acquired from instruction and data trace are used by the analysis tool to maintain the program state.

The next transformation (t2) changes the Aspect of the trace to “data-value trace” and filters the Scope to a particular variable in an application using variable reads/writes, address of variable in memory etc. as LoD. Finally, the visualization tool uses this trace to show variable data values over time.

The dotted line from the visualization tool represents the user interaction. The analysis report from the program state further represents the subpart of the analysis tool that is interactive and allows the user to influence the transformation (t2) for a refined visualization.

D. Incorporation of Data from on-chip Bus Trace

Since embedded systems are composed of a set of different components, traces from buses, peripherals, and controllers can add value for a comprehensive analysis. Specially by combining an event trace containing all bus transactions with an instruction trace can provide information related to program read and write accesses, thus also broadening the Scope.

The only problem with such a combination is that it requires the target to export all the required information to clearly map

load/store instructions to read and write accesses. This can become difficult with limited bandwidth of trace ports.

This use case presents a similar scenario where combining hardware and bus traces can help the user to investigate unusual bus activities. The user can interactively examine a region in the visualization tool that shows a high number of average bus stalls. This inquiry leads to the analysis tool which, with the help of the combined trace, maintains reports related to program read and write accesses, contention among different bus transactions or any conflicts. For instance, the user may find the application or source-code location that is causing the extra stalls by holding the bus, after looking into this analysis report. Moreover, it can be used to find if the CPU is waiting for a response from a peripheral device that also shares access from another application. Fig. 2d provides the model of this workflow.

Hardware trace in this case contains an extra LoD: Performance counters, which are hardware registers commonly available in most CPUs for measuring performance metrics like cache misses, instruction count, cycle count, bus stalls and so on. The transformation (t1) changes the Aspect of the trace from "instruction flow" to "program flow" by extracting the data events related to executed calls and returns, thus lowering the level of detail to function entry/exit.

Transformation (t2) further filters down the trace to selective target applications and their respective LoD, i.e., function entry/exit and counter values for bus stalls. The visualization tool then can be used to graphically present function entries/exits, along with the average bus-stalls histogram for the duration of the function execution. A similar workflow can also be modeled using instrumentation-based tracing as a source only.

In case of higher bus-stall cycles causing longer executions for a function, the user can select that region from the histogram for a detailed analysis. This is shown by the dotted line from the visualization tool which influences the information used as input to the analysis tool.

The transformation (t3) takes data like load and store instructions from the instruction trace and maps this information with bus transactions to get the trace with application-specific reads and writes. The analysis tool uses this trace data to provide a more-detailed analysis related to any conflicts, contention, and stalls over the system bus. Moreover, the user can influence the transformation (t3) for a refined analysis represented by the dotted arrow from the analysis tool.

V. Case Study Implementation: Integrated Application and Kernel Tracing

The objective of this section is to present how a modeled workflow can be translated into a modular framework application. In order to support this mapping, we implemented the first example described in the previous section as a prototype. This example integrates different trace sources, not only to broaden the scope but also to provide a comprehensive visualization for the developer. It shows how this improved visualization can help to find sporadic errors in an application.

A. Prototype implementation

As a proof of concept, we began with the implementation of combined application and kernel tracing. We chose VampirTrace [1] for the application and LTTng [9] for kernel tracing respectively. Event traces from VampirTrace were stored in the Open Trace Fromat (OTF) [22], and in the Common Trace Format (CTF) [23] from LTTng.

We manually instrumented two applications using VampirTrace for entry and exit of different phases (functions). One of these applications was periodic with timing constraints (soft-RT) task, while the other was non-periodic and without timing requirements (GPtask). For the soft-RT task, a marker API was also added to get the events for the cases in which the application may miss any deadline. Both of these applications were scheduled on separate cores (core affinity). At the same time with the help of LTTng, kernel events (system calls, interrupts and scheduling) were recorded.

The OTF streams generated by the applications were then fed as an input to the VAMPIR [24] visualization tool to visualize the program execution flow. Fig. 3 shows a time slice of soft-RT and GP application events.

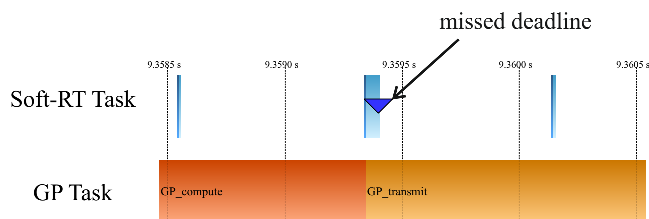


Figure 3: VAMPIR screen-shot: Visualization of application traces for real-time and general-purpose tasks.

The two different colors in soft-RT task's timeline represent different functions being executed, whereas the periodic nature of the task can be seen from invocations of these two functions at distant time intervals. Also, there is a triangle on top of one of the invocations, indicating a missed deadline. The lower time line indicates two different phases of the GP task. With this view in the visualization tool, it is difficult to speculate about any reason for the missed deadline.

For our prototype framework implementation, we used a modular approach in mapping sources, transformations, and sinks. Since the trace data in this use case comes from different sources with different trace formats, we chose OTF as an internal format for the framework. For this reason, the kernel trace which is stored in the CTF format is converted into the OTF format.

The input module provides CTF traces depicting the execution flow of both applications to the transformation module. It then performs the necessary processing to reduce the Scope of the trace to application specific kernel events. This module also correlates the kernel and application traces with the help of timing information, and converts the kernel trace from CTF to OTF. After the conversion, another independent transformation module merges the application trace (input module) with the transformed kernel trace, in order to prepare it for the output module (VAMPIR tool).

Fig. 4 shows a screen shot from the tool with integrated kernel tracing using the transformed trace information, as shown previously in Fig. 2a. The additional two time lines indicated

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-304-9

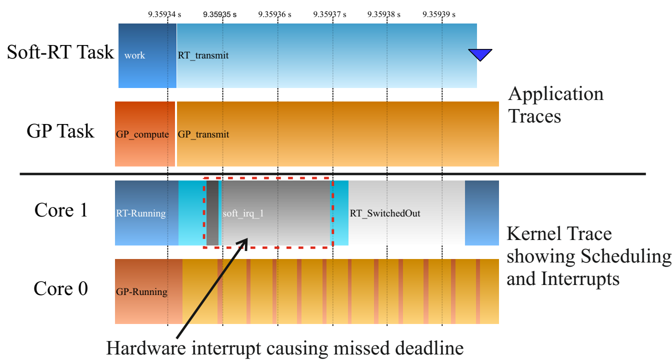


Figure 4: VAMPIR screen-shot: Combined application and kernel scope with zoom-in view for missed deadline.

by Core 1 and Core 0 represent the actual mapping of tasks onto the hardware. In this specific case, the soft-RT task was scheduled on core 1 and the GP task on core 0. Moreover, it can be seen that now the timeline shows extra events like hardware/software interrupts and scheduling events, e.g., thread switch in/out during the same execution period. In other words, the correct notion of function execution time is being depicted using kernel trace data.

Finally, by looking into the integrated view, the user can now identify the actual reason behind the soft-RT task missing its deadline, which in this case is caused by a hardware interrupt being serviced by Core 1.

VI. Conclusion and Future Work

The methodology introduced in this paper can be used to systematically describe complex analysis workflows. The possibility of constructing methods for complex workflows, which can utilize trace data from multiple sources, can be used to develop flexible tools for trace-based debugging and optimization. Our goal is to encourage the use of existing sources of trace data.

With the help of provided modeling notations, one can document the new processes and workflows in an abstract way. By modeling a workflow, the dependencies between different trace data involved and the interfaces between tools become transparent for the implementation. Our methodology can be used not only for modeling purposes, but also for providing a basis for mapping the modeled workflow to a flexible and extensible framework. We have shown this by translating one of the use cases to a modular framework in our prototype implementation.

As part of our future work, we intend to extend the framework with an internal format other than the OTF. Also, the trace features and transformations that are modeled using simple graphical notations will be described formally with the help of a machine-readable language. Furthermore, the details about the trace data (events and their semantics) will be represented using our model. These additions will help in automation of the trace processing. Finally, the modeled workflows will support automatic generation of "glue code" for a framework implementation.

References

[1] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel, "The vampir performance analysis tool-set," in *Tools for High Performance Computing*. Springer, 2008, pp. 139–155.

[2] V. Pillet, J. Labarta, T. Cortes, and S. Girona, "Paraver: A tool to visualize and analyze parallel code," *WoTUG-18*, pp. 17–31, 1995.

[3] F. Wolf, B. J. Wylie, E. Abrahám, D. Becker, W. Frings, K. Fürlinger, M. Geimer, M.-A. Hermanns, B. Mohr, S. Moore *et al.*, "Usage of the scalasca toolset for scalable performance analysis of large-scale parallel applications," in *Tools for High Performance Computing*. Springer, 2008, pp. 157–167.

[4] M. Auguston, "Building program behavior models," *Engineering Automation for Reliable Software*, p. 35, 2000.

[5] S. Boroday, H. Hallal, A. Petrenko, and A. Ulrich, "Formal modeling of communication traces." in *ISTA*. Citeseer, 2003, pp. 97–108.

[6] C. Sadowski and J. Yi, "Tiddle: a trace description language for generating concurrent benchmarks to test dynamic analyses," in *Proceedings of the Seventh International Workshop on Dynamic Analysis*. ACM, 2009, pp. 15–21.

[7] M. McGavin, T. Wright, and S. Marshall, "Visualisations of execution traces (vet): an interactive plugin-based visualisation tool," in *Proceedings of the 7th Australasian User interface conference-Volume 50*. Australian Computer Society, Inc., 2006, pp. 153–160.

[8] A. Hamou-Lhadj and T. C. Lethbridge, "A survey of trace exploration tools and techniques," in *Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, 2004, pp. 42–55.

[9] M. Desnoyers and M. R. Dagenais, "The lttng tracer: A low impact performance and behavior monitor for gnu/linux," in *OLS (Ottawa Linux Symposium)*. Citeseer, 2006, pp. 209–224.

[10] I. Park and R. Buch, "Event tracing- improve debugging and performance tuning with etw," *MSDN magazine*, p. 81, 2007.

[11] T. Beauchamp and D. Weston, "Dtrace: The reverse engineer's unexpected swiss army knife," *Blackhat Europe*, 2008.

[12] N. Merriam, P. Gliwa, and I. Broster, "Measurement and tracing methods for timing analysis," *International Journal on Software Tools for Technology Transfer*, vol. 15, no. 1, pp. 9–28, 2013.

[13] M. Williams, "Low pin-count debug interfaces for multi-device systems," 2009.

[14] (2013, August) LAUTERBACH Development Tools. [Online]. Available: http://www.lauterbach.com/

[15] (2013, August) PLS Development Tools. [Online]. Available: http://www.pls-mc.com/

[16] (2013, August) iSYSTEM. [Online]. Available: http://www.isystem.com/

[17] W. Orme, "Debug and trace for multicore socs," ARM Limited, White Paper, 2008.

[18] "Coresight^{TM} components technical reference manual," ARM Limited, Tech. Rep., 2009.

[19] N. Stollon, "Infineon multicore debug solution," in *On-Chip Instrumentation*. Springer, 2011, pp. 219–230.

[20] "Amba ahb trace macrocell (htm) technical reference manual," ARM Ltd., San Jose, CA, Tech. Rep. ARM DDI 0328E.

[21] "Usb event tracign for windows," Microsoft, White Paper, March 2010.

[22] A. Knüpfer, R. Brendel, H. Brunst, H. Mix, and W. E. Nagel, "Introducing the open trace format (otf)," in *Computational Science–ICCS 2006*. Springer, 2006, pp. 526–533.

[23] (2013, August) Common trace format (ctf). EfficiOS. [Online]. Available: http://www.efficios.com/ctf

[24] W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach, *VAMPIR: Visualization and analysis of MPI resources*. Citeseer, 1996.

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-304-9