DiaSys: On-Chip Trace Analysis for Multi-Processor System-on-Chip

Philipp Wagner, Thomas Wild, and Andreas Herkersdorf

Lehrstuhl für Integrierte Systeme, Technische Universität München,
Arcisstraße 21, 80333 München, Germany
{philipp.wagner,thomas.wild,herkersdorf}@tum.de
https://www.lis.ei.tum.de

Abstract. To find the cause of a functional or non-functional defect (bug) in software running on multi-processor System-on-Chip (MPSoC), developers need insight into the chip. For that, most of today's SoCs have hardware tracing support. Unfortunately, insight is restricted by the insufficient off-chip bandwidth, a problem which is expected to become more severe in the future as more functionality is integrated on-chip. In this paper, we present a novel tracing system architecture, the diagnosis system "DiaSys." It moves the analysis of the trace data from the debugging tool on a host PC into the chip, avoiding the off-chip bandwidth bottleneck. To enable on-chip processing, we propose to move away from trace data streams towards self-contained diagnosis events. These events can then be transformed on-chip by processing nodes to increase the information density, and then be transferred off-chip with less bandwidth. We evaluate the concept with a prototype hardware implementation, which we use to find a functional software bug. We show that on-chip trace processing can significantly lower the off-chip bandwidth requirements, while providing insight into the software execution equal to existing tracing solutions.

Keywords: Debugging, Tracing, MPSoC, SoC Architectures

1 Introduction

To write high-quality program code for a Multi-Processor System-on-Chip (MPSoC), software developers must fully understand how their code will be executed on-chip. Debugging or diagnosis tools can help developers to gain this understanding. They are a keyhole through which developers can peek and observe the software execution. Today, and even more in the future, this keyhole narrows as MPSoCs integrate more functionalities, at the same time as the amount of software increases. Furthermore, the interaction between software and hardware components increases beyond the instruction set architecture (ISA) boundary. Therefore, more, not less, insight into the system is required to keep up or even increase developer productivity.

Many of today's MPSoCs are executing concurrent code on multiple cores, interact with the physical environment (cyber-physical systems), or need to finish execution in a bounded amount of time (hard real-time). In these scenarios, non-intrusive observation of the software execution is required, which tracing provides.

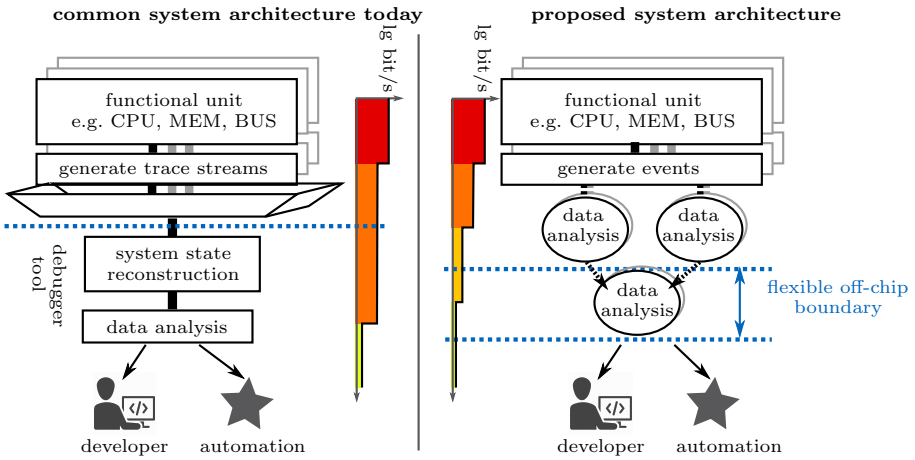

Fig. 1. Comparing a traditional SoC tracing system architecture (left) with our proposed architecture (right). Only the trace data path is shown.

Instead of stopping the system for observation (as it is done in run-control debugging), the observed data is transferred off-chip for analysis. Unfortunately, observing a full system execution would generate data streams in the range of petabits per second [14, p. 16]. As the bandwidth of available I/O interfaces is limited, only a part of the system can be observed at any given time. In summary, the lack of sufficient off-chip bandwidth is the most significant drawback of tracing.

Today's tracing system architectures, like ARM CoreSight [1] or NEXUS 5001 [2] follow a design pattern as shown on the left in Figure 1. The foremost design goal is the bandwidth-efficient transmission of data streams from CPUs, memories and interconnects through an off-chip interface to a host PC. This is achieved by configurable filtering, (cross-)triggering and compression methods. On the host PC a debugger tool reconstructs the SoC system state using the program binary and other static information. It then extracts useful information out of the data and presents it to a developer or to another tool, e.g. for runtime verification.

The **main idea** in this work is to move the trace data analysis (at least partially) from the host PC into the chip, as shown on the right side of Figure 1. Bringing the computation closer to the data reduces the off-chip bandwidth requirements, and ultimately increases insight into the software execution.

To realize this idea, in this paper we present a novel tracing system architecture, the "diagnosis system" which enables on-chip trace data analysis. As part of this system we introduce a general-purpose programmable on-chip data analysis element, the "diagnosis processor." It executes a trace analysis program on-chip and increases the information density of the off-chip traffic.

The further paper is structured as follows. Based on the related work in the field of SoC tracing and scriptable and event-based debugging in Section 2, we present our concept of the diagnosis system in Section 3. The feasibility of the concept is shown with a hardware implementation presented in Section 4.1, which is used in Section 4.2 to find a functional bug in a software program.

2 Related Work

Our approach relates to works from two fields of research. First, trace-based debugging for SoCs, and second, scriptable debugging and trace analysis.

Today's **tracing solutions for SoCs** are structured as shown in Figure 1. First, a trace data stream is obtained from various functional units in the system, like CPUs, buses and memories. Then, this data is spatially and temporally reduced through the use of filters and triggers. Finally, the redundancy in the data is removed by the use of compression algorithms. The resulting trace data stream is then transferred off-chip (live or delayed through an on-chip memory buffer). On a host PC, the original trace stream is reconstructed and analyzed by debuggers or profilers.

All major commercial SoC vendors offer tracing solutions based on this template. ARM provides its licencees the CoreSight intellectual property (IP) blocks [1], which are used in SoCs from Texas Instruments, Samsung and STMicroelectronics, among others. Vendors such as Freescale/NXP include tracing solutions based on the IEEE-ISTO 5001 (Nexus) standard [2], while Infineon integrates the Multi-Core Debug Solution (MCDS) into its automotive microcontrollers [9]. The main differentiator between the solutions is the configurability of the filter and trigger blocks.

Driven by the off-chip bottleneck, a major research focus are lossless trace compression schemes. Program trace compression available in commercial solutions typically requires 1 to 4 bit per executed instruction [8, 12], while solutions proposed in academia claim compression ratios down to 0.036 bit per instruction [13]. Even though data traces contain in general no redundancy, in practice compression rates of about 4:1 have been achieved [8].

Scriptable or programmable debugging applies the concept of event-driven programming to debugging. Whenever a defined *probe point* is hit, an event is triggered and an *event handler* executes. Common probe points are the execution of a specific part of the program (like entering a certain program function), or the access to a given memory location. The best-known current implementations of this concept are DTrace, SystemTap and ktap, which run on, or are part of, BSDs, Linux, and MacOS X (where it is integrated into the "Apple Instruments" product) [3, 5]. The concept, however, is much older. Dalek [11] is built on top of the GNU Debugger (GDB) and uses a dataflow approach to combine events and generate higher-level events out of primitive events. Marceau et al. extend the dataflow approach and apply it to the debugging of Java applications [10]. Coca [4], on the other hand, uses a language based on Prolog to define conditional breakpoints as a sequence of events described through predicates for debugging C programs.

However, all mentioned scriptable debugging solutions are implemented in software running as part of the debugged system and are therefore intrusive. The design decisions reflect the environment of desktop to high performance computers and need to be reconsidered when applying the concept to SoCs.

3 DiaSys: A System for On-Chip Trace Analysis

3.1 Requirements for the Diagnosis System

Before presenting the concept of the diagnosis system, we first formulate a set of requirements, which guide both the development of the general concept, as well as the specific hardware implementation.

First and foremost, the diagnosis system must be able to reduce the amount of trace data as close to the source, i.e. the functional units in the SoC, as possible. Since the data sources are distributed across the chip, the diagnosis system must also be distributed in the same way.

Second, the diagnosis system must be non-intrusive (passive). Non-intrusive observation preserves the event ordering and temporal relationships in non-deterministic concurrent executions, a requirement for debugging multi-core, real-time, or cyber-physical systems [6]. Non-intrusiveness also makes the diagnosis process repeatable, giving its user the confidence that he or she is searching for the bug in the functional code, not chasing a problem caused by the observation (a phenomen often called "Heisenbug" [7]).

Third, the design and implementation of a tracing system involves a trade-off between the provided level of observability and the system cost. The two main contributions to the system cost are the off-chip interface and the used chip area. The diagnosis system concept must be flexible enough to give the chip designer the freedom to configure the amount of chip resources, the off-chip bandwidth and the pin count in a way that fits the chips target market. At the same time, the system must be able to adapt to a wide range of bugs by being tunable at run-time to observe different locations in the SoC and execute different types of trace analysis.

3.2 The Concept of the Diagnosis System

The starting point for all observations is a functional unit in the SoC. A functional unit can be a CPU, a memory, an interconnect resource such as a bus or a NoC router, or a specialized hardware block, like a DMA controller or a cryptographic accelerator. Each of these components has a state, which changes during the execution of software. Capturing and analyzing this state is the goal of any debugging or diagnosis approach. In the following, we refer to the full state of a functional unit F at time t as $S(F,t)$, or, in short, S. A subset of this state is what we call a "partial state of F", $S_P(F,t) \subset S(F,t)$. P describes which elements of the full state S are included in the subset S_P.

Our diagnosis system architecture consists of three main components: event generators, processing nodes, and event sinks. Between these three components, events are exchanged.

A **diagnosis event** $E = (t, C, S_P)$ is described by the 3-tuple of a trigger condition C, a partial state snapshot $S_P(F, t)$ and the time t when the event was created. Together, the event answers three questions: when was the event generated, what caused its generation, and in which state was the functional unit at this moment in time.

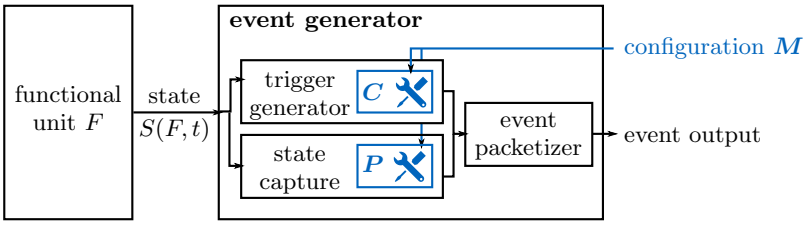

Fig. 2. A block diagram of a generic event generator.

Event generators produce events based on the observation of the state $S(F, t)$ of the functional unit F they are attached to. A schematic block diagram of an event generator is shown in Figure 2. The event generators can be configured with configuration sets $\boldsymbol{M} = \{M_1, M_2, ..., M_n\}$ with $M = (C, P)$. The trigger condition C achieves temporal selection by defining a condition which causes an event to be generated. P describes which parts of the full state S are included in the event.

Event generators can be attached to all types of functional units, for example NoC routers, memories or CPUs. For example a CPU event generator generally supports trigger conditions based on the program counter (PC) value, which essentially describes the execution of a line of code. In this case, a specific example for a trigger condition C is `PC == 0x2020` which causes an event to be generated if the program counter reaches the value `0x2020`.

Processing nodes transform events by consuming a set of incoming events $\boldsymbol{E}_i = \{E_{i,1}, E_{i,2}, ..., E_{i,n}\}$, and possibly producing new events $\boldsymbol{E}_o = \{E_{o,1}, E_{o,2}, ..., E_{o,m}\}$ as result. The goal of this transformation is to increase the information density of the data contained in the events. The applied transformation function $f : \boldsymbol{E}_i \rightarrow \boldsymbol{E}_o$ depends on the type of the processing node, and possibly its run-time configuration.

For example, a simple processing node could just compare the $S_{P,i}$ of an incoming event with an expected value, and only produce a new event E_o if this value is not found. A more complex processing node could calculate statistical metrics out of the incoming data streams, such as averages or histograms. All processing nodes have in common that they provide a platform to apply knowledge about the system and the software execution to the events.

In this paper, we present a processing node called "diagnosis processor" which can be programmed by the user to perform a wide range of analysis tasks.

Event sinks consume events. They are the end of the event chain. Their purpose is to present the data either to a human user in a suitable form (e.g. as

a simple log of events, or as visualization), or to format the events in a way that makes them suitable for consumption by an automated tool, or possibly even for usage by an on-chip component. An example usage scenario for an automated off-chip user is runtime validation, in which data collected during the runtime of the program is used to verify properties of the software.

Together, event generators, processing nodes and event sinks build a processing chain which provides powerful trace analysis according to the requirements outlined in the previous section. In the next section we present a specific type of a processing node, the diagnosis processor.

3.3 The Diagnosis Processor: A Multi-Purpose Processing Node

The diagnosis processor is a freely programmable general-purpose processing node. Like any processor design, it sacrifices computational density for flexibility. Its design is inspired by existing scriptable debugging solutions, like SystemTap or DTrace, which have shown to provide a very useful tool for software developers in a growingly complex execution environment. The usage scenario for this processing node are custom or one-off trace data analysis tasks. This scenario is very common when searching for a bug in software. First, a hypothesis is formed by the developer why a problem might have occurred. Then, this hypothesis must be validated in the running system. For this validation, a custom data analysis script must be written, which is highly specific to the problem (or the system state is manually inspected). This process is repeated multiple times, until the root cause of the problem is found. As this process is approached differently by every developer (and often also influenced by experience and luck), a very flexible analysis runtime is required.

We present the hardware design of our diagnosis processor implementation in Section 4.1.

We envision the programming of the diagnosis processor being done through scripts similar to the ones used by SystemTap or DTrace. They allow to write trace analysis tasks on a similar level of abstraction as the analyzed software itself, leading to good developer productivity.

4 Evaluation

In the following we show how to realize the diagnosis system concept in a hardware implementation. We then apply this hardware implementation in a use case showing how to find a functional bug in a software.

4.1 Prototype Implementation

Based on the concept of the diagnosis system as discussed in the previous section, we designed a diagnosis extension for a 2×2 tiled multi-core system. The functional system consists of four OpenRISC CPU cores with attached distributed memory components and a mesh NoC interconnect, as shown in Figure 3 (components

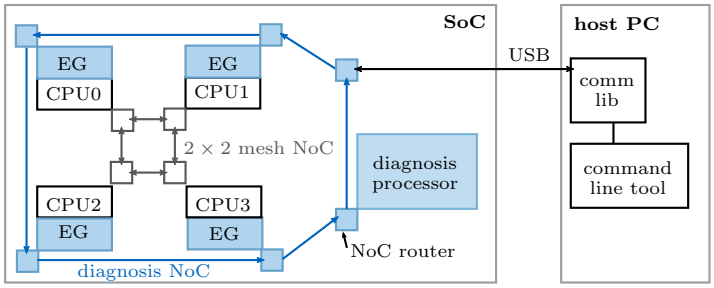

Fig. 3. Block diagram of the prototype implementation.

with white background). This system is representative of the multi- and many-core architecture template currently in research and available early products, such as the Intel SCC or the EZchip (formerly Tilera) Tile processors.

The diagnosis system, depicted in blue, consists of the following components.

- Four event generators attached to the CPUs (marked "EG" in Figure 3).
- A single diagnosis processor.
- A 16 bit wide, unidirectional ring NoC, the "diagnosis NoC," to connect the components of the diagnosis system. It carries both the event packets as well as the configuration and control information for the event generators and processing nodes.
- A USB 2.0 off-chip interface.
- Software support on the host PC to control the diagnosis system, and display the results.

All components connected to the diagnosis NoC follow a common template to increase reusability. Common parts are the NoC interface and a configuration module, which exposes readable and writable configuration registers over the NoC. In the following, we explain the implementation of the main components in detail.

CPU Event Generator The CPU event generator is attached to a single CPU core. Its main functionality is implemented in two modules, the trigger module and the system state snapshot unit. The trigger unit of the CPU event generator fires on two types of conditions: either the value of the program counter (PC), or the return from a function call (the jump back to the caller). At each point in time, 12 independent trigger conditions can be monitored. The number of monitored trigger conditions is proportional to the used hardware resources. Our dimensioning was determined by statistical analysis of large collections of SystemTap and DTrace scripts: ≤ 9 concurrent probes are used in 95 % of SystemTap scripts, and ≤ 12 concurrent probes cover 92 % of the DTrace scripts. The partial system state snapshot $S_P(\text{CPU})$ can contain the CPU register contents and the function arguments passed to the function. A block diagram of the CPU event generator is shown in Figure 4.

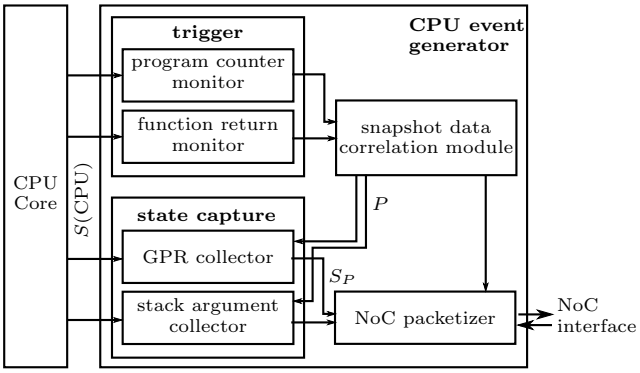

Fig. 4. Block diagram of the CPU event generator.

It is possible to associate an event with parts of the system state: the contents of the CPU general purpose registers (GPR), and the arguments passed to the currently executed function.

The passing of function arguments to functions depends on the calling convention. On OpenRISC, the first six data words passed to a function are available in CPU registers, all other arguments are pushed to the stack before calling the function. This is common for RISC architectures; other architectures and calling conventions might pass almost all arguments on the stack (such as x86). To record the function arguments as part of the system state we therefore need to create a copy of the stack memory that can be accessed non-intrusively. We do this by observing CPU writes to the stack memory location.

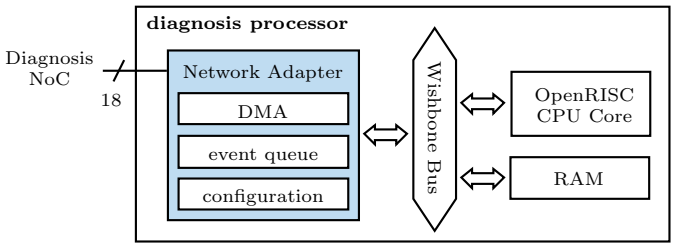

Fig. 5. Block diagram of the diagnosis processor.

Diagnosis Processor The diagnosis processor design is based on a standard processor template, which is extended towards the use case of event processing. The main components are a 32 bit in-order RISC processor core with the or1k instruction set (mor1kx) and a SRAM block as program and data memory. This

system is extended with components to reduce the runtime overhead of processing event packets.

First, the network adapter, which connects the CPU to the diagnosis NoC, directly stores the incoming event packets in the memory through a DMA engine. All event packets are processed in a run-to-completion fashion. We can therefore avoid interrupting the CPU and instead store the address of the event to be processed next in a hardware "run queue". A "discard queue" signals the hardware scheduler which events have been processed and can be purged from memory.

Module	**LUTS**	**REGS**	**RAMS**
functional system	40625	29638	80
1 compute tile (system contains 4)	$\sim$ 7232	$\sim$ 4763	20
2×2 mesh NoC	10791	9964	0
support infrastructure (DRAM if, clock/reset mgr)	904	623	0
diagnosis system	19556	19140	147
1 CPU Event Generator	3603	6521	2
1 CPU Event Generator (CoreSight-like functionality)	1365	1594	0
1 Diagnosis Processor	8614	4549	145
diagnosis NoC	2520	2926	0

Table 1. The resource usage of a CPU diagnosis unit.

Resource Usage The prototype of the tiled MPSoC with the diagnosis extensions was synthesized for a ZTEX 1.15d board with a Xilinx Spartan-6 XC6SLX150 FPGA. The relevant hardware utilization numbers as obtained from a Synplify Premier FPGA synthesis are given in Table 1.

The functional system, even though it consists of four CPU cores, is relatively small, as the used mor1kx CPU cores are lightweight (comparable to small ARM Cortex M cores). The functional system contains no memory, but uses an external DDR memory.

In this scenario, the full diagnosis system rather large. We have implemented two types of CPU event generators. A "lite" variant of the event generator can trigger only on a program counter value, and not on the return from a function call. This reduced functionality makes the event generator comparable to the feature set of the ARM CoreSight ETM trace unit, which is said to use $\sim$ 7,000 NAND gate equivalents [12], making it similarly sized as our event generator. The possibility to trigger also on the return from a function call significantly increases the size of the event generator, mostly due to additional memory. The diagnosis processor is about 20 percent larger than a regular compute tile, as it contains an additional DMA engine and the packet queues. It also contains 30 kByte of SRAM as program and data memory, which is not present in a regular compute tile.

In summary, the resource usage of the diagnosis system is acceptable, especially if used in larger functional systems with more powerful CPU cores. At the same

time, the implementation still contains many opportunities for optimization, which we plan to explore in the future. Also, a full system optimization to determine a suitable number of diagnosis processors for a given number of CPU cores is future work.

4.2 A Usage Example

In the previous section we described an implementation of our diagnosis system architecture containing a diagnosis processor. As discussed in Section 3.3, this processing node is especially suited for hypothesis testing in functional bugs. In the following, we show how to find a functional bug in the C program presented in Listing 1.1.

```
1  void write_to_buf(char* string, uint32_t size) {
2    struct {
3      char buf[99];
4      char var;
5    } test;
6    /* ... */
7    strncpy(test.buf, string, size);
8    /* ... */
9  }
10
11 int main(int argc, char** argv) {
12   char teststr[100] = "string_100_chars_long...";
13   for (int i = 0; i < 10000000; i++) {
14     uint32_t len = (i % 100) + 1; /* len [1;100] */
15     write_to_buf(teststr, len);
16   }
17   return 0;
18 }
```

Listing 1.1. A buggy C program.

The program repeatedly calls the function `write_to_buf` with a string and the size of the string. The value of the `size` argument sweeps between 1 and 100. Inside `write_to_buf`, the string is copied into the buffer `test.buf` using the `strncpy` C library function.

The code contains a bug. The `test.buf` variable holds only 99 characters, while with `size == 100` hundred characters are copied into it – a buffer overflow occurs. This causes the data in the variable `test.var` to be overwritten. In the best case this data corruption is annoying, whereas in the worst case this results in a critical security issue.

The debugging process might start with a bug report describing a data corruption on the `text.var` variable. To find the cause of such a defect, a developer might first rely on automated analysis tools. But since out-of-bounds errors on the stack (as in this case) are hard to find, neither the GCC compiler nor Valgrind (with the "exp-sgcheck" tool) issue any warning or error.

Since automated tools did not report anything suspicious, the developer needs to form a hypothesis what might have caused the defect, and test this hypothesis

by collecting live data during the application run. The hypothesis in this case is "the value passed as `size` is greater than 99."

Using a traditional tracing system like ARM CoreSight ETM or Nexus 5001 Class 3, we would obtain a full program trace, together with a data trace of writes to the `size` variable.[1] The program trace is compressed to 2 bit/instruction, and the data trace is not compressed. Scaling to the same execution speed as in our prototype, which runs at 25 MHz and executes an average of one instruction every five cycles, this would result in an off-chip bandwidth requirement of 10 Mbit/s. It must be noted that in a faster system this number scales linearly, quickly reaching typically available off-chip interface speeds.

Now we turn to our implementation. First, we measured the execution time of the program by inserting program code to count the number of executed cycles between line 12 and line 16 in Listing 1.1. The measurement showed an equal number of executed cycles if the diagnosis system was enabled or disabled, i.e. that our solution is non-intrusive to the program execution.

Second, the on-chip traffic was analyzed. The event generator creates an event packet every time the function `write_to_buf` is executed. Every event packet consists of six NoC flits (each 16 bit wide): one header flit (with destination and packet type), a 32 bit wide time stamp, a 16 bit wide event identifier, and two flits containing the state snapshot, i.e. the value of the `size` variable. This leads to a NoC traffic of 4.3 Mbit/s, or 12 % of the theoretical NoC bandwidth. This shows two things: first, the NoC link was dimensioned wide enough to connect all four CPU event generators, and second, the event generators are sufficiently selective not to overwhelm the NoC with too many events, which are later discarded.

After processing in the diagnosis processor, only every 100^{th} event generates an off-chip data packet. The off-chip packet is similar to the on-chip packets, but consists only of four flits: one header, a 32 bit time stamp, and a 16 bit event identifier. This results in 0.029 Mbit/s of off-chip traffic, which can easily be handled by cheap interfaces like JTAG. Compared to the compressed full trace, which required a bandwidth of 10 Mbit/s, the off-chip traffic was reduced by a factor of 345.

5 Conclusions

In this paper we presented a novel tracing system architecture for MPSoCs. We base our design on the observation that bringing the computation required for trace analysis closer to the data source can solve the off-chip bottleneck problem, which limits system observability in today's tracing solutions. The system architecture itself uses diagnosis events as method to transport data in a processing pipeline. Diagnosis events are transformed in a processing pipeline in order to increase their information density. The processing pipeline consists of event sources, which are attached to the SoC's functional units, processing nodes which transform the data to increase their information density, and event sinks, which are usually located on a developer's PC, but can also be on-chip. We also

[1] We assume that a single memory location stores the `size` variable. In our compilation this is the case.

present a powerful processing node, the diagnosis processor. Based on a prototype implementation we show how our system architecture increases the insight into the software execution and makes it possible to find a bug in a program in an intuitive manner.

In the future, we plan to extend this system with more specialized processing nodes, which are suited for common analysis tasks. We also investigate how machine-learning approaches can be used to dynamically adjust the analysis tasks during runtime.

Acknowledgments. This work was funded by the Bayerisches Staatsministerium für Wirtschaft und Medien, Energie und Technologie (StMWi) as part of the project "SoC Doctor," and by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre "Invasive Computing" (SFB/TR 89). The responsibility for the content remains with the authors.

References

1. CoreSight - ARM. `http://www.arm.com/products/system-ip/coresight/`
2. The Nexus 5001 Forum Standard for a Global Embedded Processor Debug Interface, Version 2.0. Tech. rep. (Dec 2003)
3. Cantrill, B.M., Shapiro, M.W., Leventhal, A.H.: Dynamic Instrumentation of Production Systems. In: Proceedings of the General Track: 2004 USENIX Annual Technical Conference. ATEC '04, USENIX Association, Berkeley, CA, USA (2004)
4. Ducassé, M.: Coca: An Automated Debugger for C. In: Proceedings of the 21st International Conference on Software Engineering. pp. 504–513. ICSE '99, ACM, New York, NY, USA (1999)
5. Eigler, F.C., Prasad, V., Cohen, W., Nguyen, H., Hunt, M., Keniston, J., Chen, B.: Architecture of systemtap: a Linux trace/probe tool (2005)
6. Fidge, C.: Fundamentals of distributed system observation. IEEE Software 13(6), 77–83 (Nov 1996)
7. Gray, J.: Why do computers stop and what can be done about it? In: Symposium on reliability in distributed software and database systems. pp. 3–12. Los Angeles, CA, USA (1986)
8. Hopkins, A.B.T., McDonald-Maier, K.D.: Debug support strategy for systems-on-chips with multiple processor cores. IEEE Transactions on Computers 55(2), 174 – 184 (Feb 2006)
9. IPextreme: Infineon Multi-Core Debug Solution: Product Brochure (2008)
10. Marceau, G., Cooper, G., Krishnamurthi, S., Reiss, S.: A dataflow language for scriptable debugging. In: 19th International Conference on Automated Software Engineering, 2004. Proceedings. pp. 218–227 (Sep 2004)
11. Olsson, R.A., Crawford, R.H., Ho, W.W.: A Dataflow Approach to Event-based Debugging. Softw. Pract. Exper. 21(2), 209–229 (Feb 1991)
12. Orme, W.: Debug and Trace for Multicore SoCs (ARM white paper) (Sep 2008)
13. Uzelac, V., Milenković, A., Burtscher, M., Milenković, M.: Real-time unobtrusive program execution trace compression using branch predictor events. In: Proceedings of the 2010 International Conference on Compilers, Architectures and Synthesis for Embedded Systems. pp. 97–106. CASES '10, ACM, New York, NY, USA (2010)
14. Vermeulen, B., Goossens, K.: Debugging Systems-on-Chip: Communication-centric and Abstraction-based Techniques. Springer, New York (Aug 2014)