# Knowledge-Based On-Chip Diagnosis for Multi-Core Systems-on-Chip

**Philipp Wagner*, Lin Li‡, Thomas Wild*, Albrecht Mayer‡, Andreas Herkersdorf***

*Lehrstuhl für Integrierte Systeme, TUM, München
{philipp.wagner, thomas.wild, herkersdorf}@tum.de

‡Infineon Technologies AG, München
{lin.li, albrecht.mayer}@infineon.com

## Abstract

Finding the cause of a failure in software running on a modern heterogeneous System-on-Chip (SoC) is challenging and time consuming. This challenge originates from two main issues. First, insight into the chip is very limited due to a huge gap between the size of the system state and the chip's ability to transfer this state off-chip. Second, the growing diversity in SoC architectures requires the developers to know the hardware architecture beyond the instruction set (ISA) interface to understand the program execution. In this paper, we show how traditional approaches to SoC debugging do not scale to future MPSoC, and present a novel approach which integrates expert knowledge into the diagnosis solution to raise the abstraction level of its output from raw data to useful information. The feasibility of the approach is shown by an industrial case study.

## 1 Introduction

*What happens in a SoC, stays in a SoC.* It is this reality that troubles many developers working on software running embedded on a System-on-Chip: finding defects in such software is difficult. It is difficult because today's heterogeneous multi-core SoC architectures make it hard to form an expectation of how the program will execute; and the limited observability of the execution (caused by the inability to transfer the full system state in the range of petabits per second off-chip) makes it hard to validate those assumptions in the running system.

Today a significant percentage of the development time of an embedded system is already spent on software development. In software development, studies have shown that 50 to 75 percent of the total development cost is spent in verification, testing and debugging [5]. Finding problems in software applications faster also often implies an earlier time to market and can decide about the economic success or failure of a product.

Software diagnosis is the task of finding the root cause of a failure, i.e. a violation of a functional or non-functional requirement. Diagnosis, also commonly called debugging (mostly for functional problems), tracing, profiling (for non-functional problems), or (more positively) performance engineering, is done in two steps: in the first step, the developer creates a (mental) model of what will happen during the program execution on the given target architecture. In the second step, he or she compares this expectation to the actual program execution, i.e. to what actually happens when the program runs.

This work was partly supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre "Invasive Computing" (SFB/TR 89).

Both steps in diagnosis are more challenging in SoC environments. Modern heterogeneous SoCs tightly integrate different processor cores, hardware accelerators, input/output (I/O) interfaces, and much more. They use optimized memory layouts next to traditional shared coherent memories, on-chip networks (NoC), and other techniques to make the chip ideally fit the intended purpose. While this helped greatly increase the system's performance, it also increased its complexity. Fortunately, for most functional concerns this complexity is hidden from the application developer by the processor's instruction set (ISA). But while hiding complexity, the ISA also hides SoC details that can influence the program execution both in functional as well as in non-functional ways. It does not describe how long an instruction is expected to take to execute, or how memory-mapped peripherals share resources. This, in turn, makes it hard or even impossible for the developer to form an expectation how the program will execute, and which behavior should be considered "expected" – the essential first step of software diagnosis.

The second step requires getting insight into the system to see how the actual execution proceeds. On standard PC platforms, the tool used to gain this insight, usually a debugger or profiler, runs alongside the analyzed program. On a SoC, this tool is outside the chip and all necessary data needs to be transferred to it through the chip boundaries. Representing the full system state requires petabits per second ($10^{15}$ bit/s) [14, p. 16], the off-chip bandwidth available for diagnosis information is in the range of gigabits per second in today's chips. This huge gap of several magnitudes severely limits the

insight into the system. Still, increasing the off-chip bandwidth would not solve the problem: processing petabits of data per second is a non-trivial task, especially when considering that most of it is not relevant to the diagnosis problem at hand.
Software diagnosis for SoCs is challenging. Complex hardware architectures lead to a gap of understanding how the software will execute, and increasing logic density leads to a huge observability gap during the program execution. To tackle those challenges, we propose a novel diagnosis solution which integrates expert knowledge (e.g. from a chip manufacturer or an experienced developer) into the diagnosis solution. The whole solution can then be integrated onto the chip, reducing the required off-chip bandwidth significantly.
In this paper, after presenting today's SoC diagnosis methods and approaches to formulate expectations about a program execution in more detail in Section 2, we present in Section 3 our novel approach to SoC diagnosis, which integrates expert knowledge to greatly improve diagnosis productivity. To validate this approach we present a case study in Section 4. There we show how we added knowledge about the SoC architecture and latencies of memory transfers to a diagnosis solution to guide the developer towards possibly problematic areas of the chip, while reducing the bandwidth requirements.

# 2 Related Work

## 2.1 SoC Diagnosis Today

To find the cause of a defect, developers typically tend to exhaust all other options before debugging the software directly on the SoC. This can include running the software on a x86 developer machine or in a virtual prototype of the SoC. In some cases, however, the only option is debugging on the target system: no virtual prototype might be available, I/O interfaces are missing, models of connected peripherals are missing, or the timing behavior is different.
In those cases, today, two diagnosis approaches for SoCs are used: run-control debugging and tracing. The first one, also known as stepping, run/stop debugging or just debugging, is an interactive approach that works by temporarily interrupting the program execution at a defined point and inspecting the current system state. This method is supported by most available SoCs today and also well known to software developers from tools like the GNU Debugger (gdb). It is intuitive to use and provides good insight into the processor and memory state. Since the program execution is halted, a developer can iteratively query many properties of interest and transfer them over a low-bandwidth interface out of the chip, keeping the cost of the off-chip interface low (in many cases, the JTAG ISO 1149.1 interface is used for this task).
While the halting of the program execution is the reason for most of the benefits of run-control debugging, it is also its greatest drawback. In many environments, especially in the embedded domain, the program execution may not be interrupted. In cyber-physical systems with hard real-time requirements, a deadline violation caused by a stalled system during debugging can cause catastrophic failures. Halting the program execution may also change the (temporal) behavior of a program, hiding problems like race conditions. Thus, for debugging many of today's SoC applications, an "invisible" diagnosis solution, i.e. a solution without probe effect, is required [10].
This different approach is called tracing. Instead of interrupting the program flow, the system state is continuously observed without obstructing it, and transferred off-chip, where it can be analyzed with virtually no time or processing power restrictions. While this method is in theory able to resolve all the drawbacks of run-control debugging, it is heavily limited by the size of system state required to reconstruct the program behavior. To allow for continuous, non-intrusive operation, the off-chip bandwidth needs to be dimensioned with the peak data rate, which is commonly in the range of multiple MBit/s up to some GBit/s. Comparatively, transferring the full system state would require a bandwidth in the range of multiple PBit/s. If continuous operation is not required, on-chip memories may serve as buffer.
To reduce the required off-chip bandwidth for tracing the developer needs to limit the observation focus both temporally and spatially. This limits the data collection to certain points in time (usually, when a point in the execution flow is hit), or to some parts of the chip (e.g. a single CPU). In tracing solutions, those limitations are called events or triggers and filters. In addition, the generated trace stream is compressed with close to no loss. Compressing instruction trace streams has been extensively studied, resulting in compression down to 0.036 bit per single-core instruction [13]. Compressing data (memory) traces is less efficient due to the generally random nature of the data, but in practice a compression to 76.3 percent of the original size has been achieved [6].
Industry solutions for tracing are common today. ARM provides the CoreSight IP cores [1] for its licensees, which include the "ETM" module for instruction traces. Many vendors, such as Texas Instruments or Samsung, integrate these IP cores into their ARM-based products. Other vendors, such as NXP, STMicroelectronics or Freescale, integrate debugging and tracing support based on the NEXUS 5001 standard. Infineon has introduced its own tracing solution, the Multi-Core Debug Solution (MCDS) [7], which is part of its automotive microcontroller family. The main differentiators between the industry tracing solutions is the varying degree of configurability at run-time, such as trigger and filter options, and the types of data sources available for inspection.

## 2.2 Formalizing Diagnosis Knowledge

The ability to formulate expectations of how a program should behave, or inversely, which behavior describes a bug, is essential when integrating diagnosis on-chip. Two main categories of bugs exist: functional and non-functional bugs.

Functional or correctness problems are the majority of bugs developers are trying to resolve. In most cases, the intended behavior of the program is codified in various types of tests, be it unit tests, regression tests, system tests, etc.

In concurrent software on multi-core processors, additional problems arise: race conditions, deadlocks and atomicity violations [9]. A lot of research has been carried out to detect data races, with a focus on two main algorithms describing correct patterns to access shared data: happens-before relations [8] and lock-sets [12]. Building on those two algorithms, methods and tools for detecting data races have developed, including hardware based solutions [15, 16].

For non-functional requirements inefficiency patterns or performance bottlenecks have been studied, mostly in the domain of High-Performance Computing (HPC) for OpenMP or MPI-based parallel applications. The Paradyn Performance Consultant [11] contains a set of experiments to test a hypothesis, such as "the application is CPU bound." The APART Specification Language (ASL) is a formal language for describing inefficiency patterns such as imbalances in thread execution times [4]. Like much of the work done in the HPC community, the presented ideas in these papers is of increasing relevance to the SoC community as system complexity increases. Due to the different programming paradigms, such as OpenMP and MPI, the approaches need to adapted to be applicable to the embedded domain.

# 3 Knowledge-Based SoC Diagnosis

An in-depth study of the existing approaches to SoC diagnosis as outlined in the previous section reveals two insights, which guide the design of our new approach to SoC diagnosis.

**Further improvements in lossless trace compression algorithms will not close the off-chip bandwidth gap.** Figure 1 shows the predicted evolution of three key figures in a SoC. The logic size follows Moore's law. This results in an exponential growth of the internal state of a SoC, increasing $24.4\times$ by the year 2024 as compared to 2011. While the number of signal I/O pins per package is predicted to increase $1.67\times$ by 2024, the test I/O speed saturated at a factor of $1.48$ compared to 2011. At the same time, trace compression rates have not seen significant improvements over the last years. Especially the largely uncorrelated nature of data value traces does not present much opportunity for data compression.

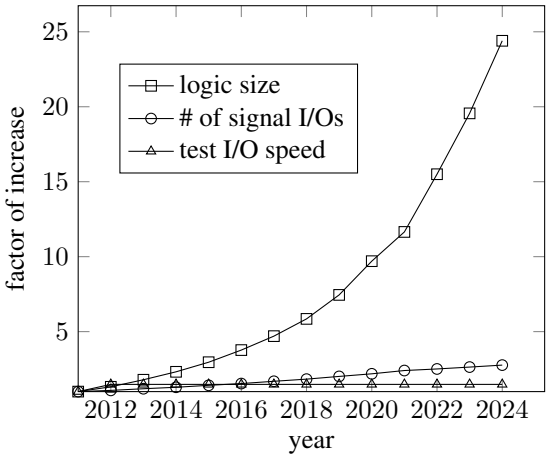


**Figure 1** Predicted evolution of key figures for SoCs. Data normalized to 2011 baseline from the ITRS Roadmap 2013 Edition [2]

**Increasing diagnosis productivity requires a solution that guides the developer towards problems.** The rapidly increasing system complexity requires equivalently rapid increases in diagnosis productivity to meet time to market demands. Today, SoC tracing tools focus on efficiently presenting data, but take little part in increasing its abstraction level; the questions "what to look for," "where to look for it," and "when to look for it" are left for the developer to answer. A productivity increase can be achieved if the developer can spend more time on fixing problems, and less time on finding the cause of it; a diagnosis solution which is able to pinpoint problems without being explicitly asked for it can help with that goal.

Based on these two insights, we propose a novel approach to SoC diagnosis. Our top goal is to raise the abstraction level of the output of the diagnosis solution from *data* to *information*. Data describes the system state as it can be observed in the system. Information, however, is data with added meaning and purpose. As such, information can be the answer to a question solving a developer's problem. The increase in abstraction results in more dense representations, it reduces the number of bits required to describe it. If the SoC diagnosis solution does not transfer observation data off-chip as it does today, but meaningful information instead, the off-chip bandwidth is not a limitation any more. At the same time, the goal of a diagnosis solution should not be the collection of raw data, but the answering of questions with relevant information. If this goal can be achieved, increased developer productivity is achieved as well.

Raising the abstraction level from data to information can be achieved through the integration of expert knowledge into the diagnosis solution. We consider three types of knowledge, as visualized in Figure 2.

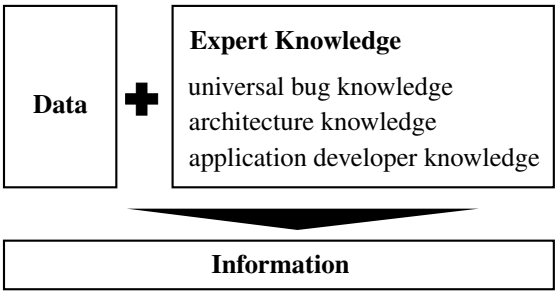


**Figure 2** Adding expert knowledge to data results in information.

- Universal bug or inefficiency patterns: knowledge about how a bug-free and efficient application behaves. This includes, for example, knowledge how race conditions present themselves, or which application behavior is considered a deadlock.
- Architecture knowledge: the execution behavior of an application depends on the SoC architecture. "An access to flash memory in general takes this long" or "if the I/O module sets an error flags, it could be caused by one of those problems" are examples.
- Finally, application developer knowledge. The line between benign and malign behavior is in general not fixed, but dependent on the use case and thus the application developer. For example, a data race can be intended for performance reasons, but it also can be a sign of a serious problem with the program code.

These three types of knowledge form the basis for the transformation from data to information. In order to use this knowledge to find the cause of failures in a SoC, we encapsulate it inside diagnostic *tests*. A test consists of three components:

- A *hypothesis*, an assumption about the reason behind a problem,
- an *experiment* which confirms or negates the hypothesis, and
- a set of *child tests* to further refine the hypothesis.

The tests are organized in a tree, with each subtree refining the hypothesis of its parent until a leaf is reached. This leaf then describes the most likely reason cause of the failure as determined by the SoC diagnosis solution. This approach formalizes the way software developers are debugging software manually.

Integrating different types of expert knowledge creates a diagnosis solution which goes beyond collecting data to presenting the developer with relevant information about the SoC status, while at the same time avoiding the off-chip interface bottleneck. A case study showing this is presented in the next section using an Infineon automotive SoC.

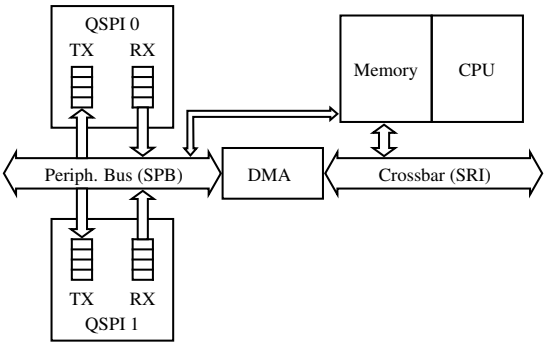


**Figure 3** Subset of the simplified Infineon AURIX TC29x SoC architecture diagram, as relevant to the presented case study.

# 4 Case Study

In the previous section, we have motivated the need for an knowledge-based SoC diagnosis solution. To show the power of this approach, we present in the following a case study with an Infineon AURIX TC29x SoC, an automotive chip mainly used in powertrain applications. The relevant part of the SoC architecture is shown in Figure 3. The components relevant to this case study are the CPU, two QSPI (queued serial peripheral interface) modules, and a DMA module. The QSPI modules are connected to a peripheral bus, the SPB. The DMA engine allows the connected peripherals to exchange data with the memory.

The QSPI module implements the SPI protocol. It can operate independent of the CPU by filling the transmit and receive buffers directly from the memory using DMA transfers. In the following case study, the module is configured for transmissions consisting of eight data words. The communication is full-duplex, i.e. sending and receiving always happen in combination.

Before we go into the details of the implementation of the diagnosis solution, we describe the diagnosis flow from a developer's point of view.

## 4.1 User Perspective

```
1  while (1) {
2    // wait until at least one QSPI module is ready
3    s0 = IfxQspi_SpiMaster_getStatus(&spiChannel);
4    s1 = IfxQspi_SpiMaster_getStatus(&spiChannel2);
5    while (s0 == SpiIf_Status_busy ||
6           s1 == SpiIf_Status_busy);
7
8    initiate_next_transmission()
9  }
```

**Listing 1** Simplified C code of the user program showing the issue we are applying the knowledge-based diagnosis to.

A software developer has written an application that sends data to peripherals like D/A converters attached over SPI to the QSPI modules of the SoC. An excerpt from the used application code is presented in Listing 1. A processing loop waits until both QSPI modules are

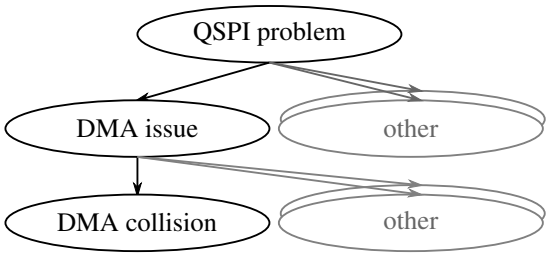


**Figure 4** Tree of conducted tests during diagnosis. Each bubble represents a hypothesis.

free and initiates the next SPI transmission.
The application runs as expected in general, but sometimes it "hangs," it does not send out SPI data any more. This problem becomes apparent since the attached SPI peripherals do not receive any more updates. The software developer notices this, and realizes that the program does not proceed beyond the `while`-loop in line 5/6 (e.g. from an instruction trace, or other debug techniques). Since the issue happens in a line of code related to the QSPI the developer formulates the first working hypothesis "The problem is QSPI related." Based on this hypothesis, an automated diagnosis solution can be applied, which is described in the following.

## 4.2 Diagnosis System Setup

To find the cause of the described failure, we integrate chip architecture knowledge from the Infineon, as well as universal knowledge how DMA congestions manifest themselves into the diagnosis solution. This creates a tree of diagnostic tests. In Figure 4 the hypotheses of these tests are shown. In this case study, we focus on two tests. The first hypothesis is "There is a DMA issue." If this hypothesis holds, we test the hypothesis "A DMA collision occurred."

## 4.3 Test 1: DMA Issue

### 4.3.1 Hypothesis

The first diagnostic test accepts or rejects the hypothesis "A DMA issue exists."

### 4.3.2 Motivation and Integrated Knowledge

This test is motivated by the industry experience that DMA-related problems can be often detected by observing the latencies on a DMA channel. A latency is defined as the time between the submission of the request and the beginning of the corresponding data transfer. In case of a normal operation, the DMA latencies show a regular, repeating pattern. The regularity results from the fact that control-oriented real-time applications are in general periodic; they continuously read sensor values, process them, and write actuator values. Irregularities in the DMA latency patterns can be an indicator for a (at this stage undefined) DMA issue, and thus validate the hypothesis. Further tests are required to refine it.

### 4.3.3 Implementation of the Experiment

First, the latencies are measured for each used DMA channel on the peripheral bus using the the Multi-Core Debug Solution (MCDS) IP core embedded on the TC29x SoC. MCDS is configured generate events if a request is sent off and if a DMA channel becomes active, i.e. if the transfer of the requested data begins. The time difference between the two events is the DMA latency. All measurements for each DMA channel $c$ are stored in a vector $v_c \in \mathbb{N}^n$. Each vector consists of $n$ latencies in MCDS clock cycles.
Based on this data, the search for irregular DMA latency patterns is implemented in two steps. First, the individual latencies are grouped into periods, and then these periods are analyzed for irregularities.
To determine the number of elements in each group (i.e. period), we minimize the Euclidean distance between two consecutive groups. Given a maximum expected group size of $g_{\max}$, the group size $g$ is described as

$$\underset{g \in 1..g_{\max}}{\arg\min} \sqrt{\sum_{i=1}^{g} (v_i - v_{g+i})^2}$$

Note that the presented algorithm expects that the first two analyzed periods are representative of the data set, and that the latencies inside a period form a sufficiently distinct pattern. In our tests, this was shown to be the case.
Based on the group size $g$ we now can group the measured latencies in $v$ into a two-dimensional matrix $M^{\lfloor n/g \rfloor \times g}$, in which each row describes the measurements of one period.
In the second processing step, our goal is to find anomalies or outliers between the periods, i.e. between the rows in $M$. We use an established algorithm to detect outliers in data sets, the Local Outlier Factor [3]. The LOF algorithm is a "soft" outlier detection algorithm, it assigns a factor to each data object representing a likeliness of being an outlier in a local search environment of the $k$ closest neighbor objects.
For the data set of DMA latencies, we found that good results are achieved by setting $k = 2$ and treating a period, i.e. a row $p$ in $M$, with $\mathrm{LOF}(M_p) > 10$ as outlier. An outlier in the data set of DMA latencies describes a possible DMA issue.

### 4.3.4 Experimental Results

In our experimental setup five DMA channels are used. Channels 1 through 4 are used to transfer the receive (RX) and send (TX) buffer contents of the two QSPI modules, respectively. Channel 10 is used to copy data in a distributed memory, an operation representative of sudden bursts of DMA traffic. In the following, we focus on channel 1 which fills the sending (TX) buffer of the QSPI 0 module. The other channels used by the QSPI modules (2, 3 and 4) show similar patterns.

Applying the two analysis steps to the measured latencies, we first determine the group size $g$ to be 9. This equals the expected period size, as a single SPI transmission consists of an initial word with configuration information, followed by eight data words. Grouping the measured latencies into a matrix, we get

$$M = \begin{pmatrix} 7 & 1 & 1 & 3 & 1 & 1 & 1 & - & - \\ 7 & 1 & 1 & 4 & 1 & 1 & 1 & - & - \\ & & & & \vdots & & & & \\ 7 & 1 & 1 & 51 & 7 & 21 & - & - & 0 \end{pmatrix}$$

In this matrix "$-$" represents a DMA request without a reply. This happens either if the request is ignored because the DMA channel was inactive, or if the request was lost due to a collision. (A definition of a DMA collision follows in the next section.)
The matrix $M$ can now serve as input to the LOF-based outlier detection in the second step, in which the last row is classified as outlier or anomaly. We therefore consider the hypothesis "A DMA issue exists" to be proven for channel 1. For a more detailed analysis, we continue with a new set of hypotheses. For this case study, we focus on one child hypothesis, "A DMA collision occurred."

## 4.4 Test 2: DMA Collision

### 4.4.1 Hypothesis

The second test assumes a collision of DMA requests happened, and tries to accept or reject this hypothesis.

### 4.4.2 Motivation and Integrated Knowledge

The TC29x SoC contains two DMA engines, allowing two concurrent DMA transfers. For each channel, one request can be queued and wait for a DMA engine to become available. If another request is received on a channel where a request is already waiting to be served, the first request is dropped. This behavior is called DMA collision. Indicative for a DMA collision is the fact that more than two DMA channels are active at the same time if an anomaly at a transmission was found.
Infineon has seen DMA collisions occur at customer setups if DMA channels were assigned wrongly. Detecting this problem based on a trace requires knowledge about the number of DMA engines and other implementation details of the SoC.

### 4.4.3 Implementation of the Experiment

To validate or deny the hypothesis in an experiment we use MCDS again. It is configured to record the state (active/inactive) of all relevant DMA channels, i.e. if the channel is currently used to transfer data or if the channel is idle.
The test "DMA Issue" in the parent node of the diagnosis tree has determined a channel $c$ containing a likely DMA issue. To confirm the refining hypothesis that a DMA collision occurred, we search for the set of DMA channels active at the same time as $c$ and store it. If more than two other active channels are found while a DMA issue was detected, the hypothesis of a DMA collision is accepted. Using the stored set of channels which are usually active at the same time we can pin-point the channel(s) which are active in addition to the normal case and which are most likely the cause for the DMA collision. This information gives the developer a good starting point towards solving the problem.

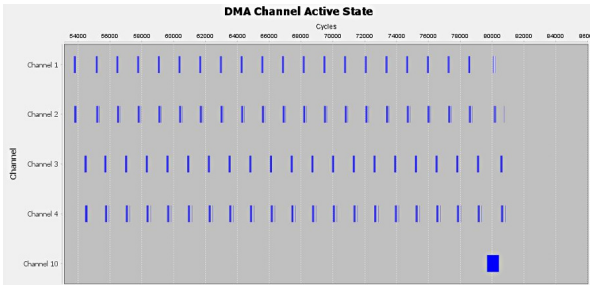


**Figure 5** Activity on the DMA channels (blue) over time, as visualized by our diagnosis tool (screen shot).

### 4.4.4 Experimental Results

In the same experimental setup as before we record the active state of all DMA channels. This results in the data set which is visualized in Figure 5. The previous test resulted in a possible DMA issue on channel 1. Searching through the data set for channels which are active at the same time as channel 1 shows that around clock cycle 80,000 channels 2 and 10 are also active. The stored set of usually active channels lists channel 1 and 2. We can use this data to conclude that the DMA collision was most likely caused by the activity on channel 10.

## 4.5 Case Study Summary

The case study showed how knowledge can be integrated into a diagnosis solution to provide the application developer with useful information instead of raw trace data. From the SoC manufacturer (Infineon) we included knowledge about the architecture, like the number of DMA engines present in the chip, and the way excessive DMA requests are handled, as well as experience about the most likely causes for a failure. From the application developer we integrated the knowledge what software was run on the system, in this case a periodic control-oriented application. We showed how to formalize this knowledge for two specific examples by encapsulating it inside diagnostic tests, which are organized in a tree. By refining a hypothesis from general to specific using this test tree we were able to determine the cause of a hanging application to be a DMA collision.

# 5 Conclusions and Outlook

Finding the root cause of a software bug in a SoC is difficult. The increasing system complexity due to shrinking feature sizes and new architecture templates require equivalently increased knowledge about the hardware architecture from software developers, which were previously able to limit their understanding of the hardware to the ISA boundary. At the same time, an observability gap prevents insight into the full system state, which makes the confirmation of assumptions on how the hardware operates difficult. In this paper, we have shown that those two problems will increase in significance over time, requiring a new approach to software diagnosis on a SoC.

Such a new approach has been presented in this paper. We integrated expert knowledge from developers and SoC manufacturers, together with the best-practices of software development and debugging into a diagnosis solution. This raises the abstraction level of the output the diagnosis solution, it presents not only raw data as it was captured in the system, but adds meaning and purpose to this data, making it useful information. As a way to integrate knowledge inside a diagnosis solution, we used a tree structure of automated tests, following a scientific approach of confirming or denying a hypothesis using a targeted experiment. A case study based on an Infineon automotive SoC showed the power of the approach.

The case study has also uncovered new difficulties. Formulating hypotheses and writing highly specific experiments to test them is tedious, and most likely does not scale with the growing system complexity. At the same time, the SoC design paradigm is based on the modularization and abstraction of individual components. Yet, the interaction between the different system components can lead to subtle and hard-to-find failures, which require the combined expert knowledge from different design teams.

In follow-up work, we currently investigate the applicability of supervised and reinforcement-based machine learning techniques in order to tackle those challenges. Supervised learning techniques may help putting expert and system knowledge in relationship to the effectiveness and possible simplification of individual experiments performed within a diagnosis framework. Unsupervised learning or data mining techniques bear the potential to discover entirely new patterns and anomalies in the trace data deluge.

# 6 Literature


[1] CoreSight - ARM.

[2] International Technology Roadmap for Semiconductors, 2013.

[3] M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander. LOF: Identifying Density-based Local Outliers. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, SIGMOD '00, pages 93–104, New York, NY, USA, 2000. ACM.

[4] M. Gerndt and K. Fürlinger. Specification and detection of performance problems with ASL. *Concurrency Computat.: Pract. Exper.*, 19(11):1451–1464, Aug. 2007.

[5] B. Hailpern and P. Santhanam. Software Debugging, Testing, and Verification. *IBM Syst. J.*, 41(1):4–12, Jan. 2002.

[6] A. B. T. Hopkins and K. D. McDonald-Maier. Debug support strategy for systems-on-chips with multiple processor cores. *IEEE Trans. Comput.*, 55(2):174 – 184, Feb. 2006.

[7] IPextreme. Infineon Multi-Core Debug Solution: Product Brochure, 2008.

[8] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, July 1978.

[9] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou. Bugbench: Benchmarks for evaluating bug detection tools. In *Workshop on the Evaluation of Software Defect Detection Tools*, pages 1–5, June 2005.

[10] C. E. McDowell and D. P. Helmbold. Debugging Concurrent Programs. *ACM Comput. Surv.*, 21(4):593–622, Dec. 1989.

[11] B. Miller, M. Callaghan, J. Cargille, J. Hollingsworth, R. Irvin, K. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn parallel performance measurement tool. *Computer*, 28(11):37–46, Nov. 1995.

[12] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, Nov. 1997.

[13] V. Uzelac, A. Milenković, M. Burtscher, and M. Milenković. 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*, CASES '10, pages 97–106, New York, NY, USA, 2010. ACM.

[14] B. Vermeulen and K. Goossens. *Debugging Systems-on-Chip: Communication-centric and Abstraction-based Techniques*. Springer, New York, Aug. 2014.

[15] C.-N. Wen, S.-H. Chou, C.-C. Chen, and T.-F. Chen. NUDA: A Non-Uniform Debugging Architecture and Nonintrusive Race Detection for Many-Core Systems. *IEEE Trans. Comput.*, 61(2):199 – 212, Feb. 2012.

[16] P. Zhou, R. Teodorescu, and Y. Zhou. HARD: Hardware-Assisted Lockset-based Race Detection. In *IEEE 13th International Symposium on High Performance Computer Architecture, 2007. HPCA 2007*, pages 121–132, Feb. 2007.