Fakultät für Elektrotechnik und Informationstechnik
Technische Universität München

A Traced-based Automated System Diagnosis and Software Debugging Methodology for Embedded Multi-core Systems

Lin Li

Vollständiger Abdruck der von der Fakultät für Elektrotechnik und Informationstechnik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs (Dr.-Ing.)

genehmigten Dissertation.

Die Dissertation wurde am 19.09.2018 bei der Technischen Universität München eingereicht und durch die Fakultät für Elektrotechnik und Informationstechnik am 22.06.2019 angenommen.

Abstract

Nowadays devices in our daily life e.g. self-driving cars, phones and smart factories are becoming smarter because powerful embedded systems are widely applied. This trend demands the support of more advanced and more sophisticated embedded systems. As the complexity of both hardware and software in embedded systems increases, the gap between the developers' understanding and the actual system status grows broader, resulting in increasing time and effort spent on software validation and debugging. Worse still, new issues also occur accompanying the current trend of multi-core/manycore embedded systems.

Conventional debugging and diagnosis tools are not efficient enough to handle these new challenges. First, many conventional debugging methods may encounter problems like "heisenbugs" that seem to disappear or alter the behavior when one tries to debug them. This is because many methods based on e.g. setting breakpoints or software instrumentation change the timing behavior of systems, which can jeopardize the reproduction of bugs. Second, the observability and the traceability of complex embedded systems are limited and not all information can be transferred out of chip. The modern single-core system already produces more than 10 gigabits per second of raw trace data while the multi-core systems are even higher. The off-chip tracing bandwidth cannot catch up the increase of raw data generated on chip so not all data can be captured continuously without halting the system. Picking up the "right" raw data to be traced with limited bandwidth in such a huge volume of raw data is challenging. Third, conventional debugging solutions e.g. breakpoints are inefficient to deal with some issues introduced by multi-core architectures. For instance, the performance impact due to shared resource contention is usually underestimated. Such contention e.g. program flash contention is transparent to software developers and improperly emphasized. For multi-core timing related issues that occur only sporadically, manual debugging is very time-consuming or sometimes even impossible.

Therefore, a novel automated system diagnosis and debugging methodology is proposed to tackle the above challenges. The proposed methodology is named "embedded health" because the initial idea is from one of best practices to cure diseases – medicine. It can be performed as a generally applicable automated diagnosis for various embedded system issues. This is very similar to a blood test via which several diseases can be rapidly and precisely detected. This methodology makes use of hardware tracing that is supposed to be supported intrinsically by the embedded system, and detects special symptoms replying on the predefined indicators. Hardware tracing is non-intrusive and thus the system timing is not influenced by the debugging process. The predefined indicators are the most important part in this methodology. The design of the indicators is independent of applications and operating systems. The low-level hardware-related behavioral patterns are used to design the indicators, similar to the metric system in the medical “blood test”. If the patterns are hit, the related indicator becomes positive and the corresponding issue is detected. The debugging process is automated and the manual involvement is minimized.

Several aspects that are not usually well covered by conventional tools are taken care of by this methodology, for instance, program flash contention, data locality and data memory contention, hardware configuration validation, DMA channel visualization, lock usage profiling etc. Data memory contention is studied by much research but the program flash memory contention is rarely covered. The program flash contention can also greatly influence the performance in multicore embedded systems by jeopardizing the instruction fetch. However, its impact is not well studied and underestimated. Worse still, the performance impact is hard to measure precisely as only contention that occurs at critical time points degrades performance. In this dissertation, an indicator is designed to handle this problem and it is the first one to detect program flash contention spots and estimate the performance impact accurately.

For multicore systems, locks are widely used to regulate the accesses from different tasks to the shared resources. The efficiency of locks is an important factor of system performance so many profiling methods are introduced for various operating systems. An indicator that profiles the lock performance independent of operating systems and reminds developers of improper lock behavior is introduced.

Embedded systems distinguish themselves from general purpose computers by many peripherals for various applications. These peripherals are supposed to be configured properly in order to have the expected functionalities. The hardware configurations are error-prone and issues caused by misconfigurations are sometimes hard-to-detect. In the proposed methodology, a method is designed to efficiently detect configuration issues even before the symptom appears.

The proposed methodology differentiates from related research in several aspects. First, it makes use of non-intrusive hardware tracing to monitor the system execution. Hardware tracing is non-intrusive, which is mandatory for analyzing hard real-time applications. Second, different indicators are investigated and designed for various issues independent of applications, meaning that the same indicator is also feasible for another application. Third, the target issues are rarely covered or not well solved by existing solutions. Issues e.g. program flash contention are even not detectable with existing tools. Finally, the detection of issues relies on the root causes instead of the superficial symptoms. It is able to detect issues even before the symptom appears and estimate the severity of the issues.

The proposed methodology was proven with experiments and then implemented as an automated system diagnosis and debugging tool named ChipCoach working for Infineon’s AURIX microcontroller family which is a popular microcontroller in the automotive industry. ChipCoach is currently used quite intensively by a group of customer support engineers and it receives very positive feedback. It continues to be improved based on field experience and will be released in a commercial way.

Acknowledgment

I would like to express my sincere appreciation and thanks to my PhD adviser Prof. Dr. sc. techn. Andreas Herkersdorf for the continuous support of my PhD study and master study. His guidance enlightened my path of exploration in the academic world and helped me all the time during my research and thesis writing. He encouraged me to cultivate innovative and critical thinking, which will benefit my whole life. Without his guidance and help this thesis would not have been possible. I also would like to thank my PhD adviser at Infineon Dr. Albrecht Mayer who provided direct support and guidance to me. He established the firm foundation of my research and showed me the promising direction that led to my current work. His experience enabled me to avoid a lot of repeated work and his optimism always heartened me when I was frustrated in the darkness.

I would also like to thank Prof. Dr.-Ing. Frank Slomka who provided me invaluable feedback in my dissertation. I thank very much my colleague Jens Harnisch, with whom I discussed a lot and improved many ideas. His experience in both the technical field and project management greatly optimized the development efficiency and the quality of the ChipCoach project. A special thanks to my project partner Philipp Wagner, who spend much time on our papers and the cooperated funding project. And also a special thanks to Max Brand, who had fruitful discussions with me and helped me with my German translation. Moreover, I am thankful to working students who contributed to the ChipCoach project and spent days on debugging.

Last but not the least, I would like to thank my family: my parents for supporting me spiritually throughout my life and my wife for her understanding. I would also like to thank all of my friends who supported me in writing, and incented me to strive towards my goal.

Contents

1 Introduction

1.1 Background

Software bugs are causing great losses to modern society. In 2002, a study commissioned by the US concluded that "Software bugs or errors, are so prevalent and so detrimental that they cost the US economy an estimated \$59 billion annually" [1]. Testing and debugging of software also consume a significant amount of time. Debugging can take up to 50% of the development time [2]. There are many famous accidents caused by software bugs resulting in great losses of not only money but also time and lives. For instance, the American robotic spacecraft "Mars Pathfinder" which landed on Mars in 1997, jeopardized the mission owing to the constantly computer resetting caused by a priority inversion bug in software. The mission was interrupted and heavily delayed just by this bug. Fortunately, this software was fixed with a short C program after hours' debugging. An unfortunate example is the first test flight of the Ariane 5 rocket. It exploded just 37 seconds after launch because of the arithmetic overflow in the hardware caused by a piece of software which is in fact not required at all, resulting in a loss of millions of dollars. The most dangerous example is the cold war missile crisis, which could have led to a nuclear war and destroyed the world. The Soviet early warning satellite system sent out a false warning that five missiles were launched by the US, which could be a trigger of a nuclear war. Fortunately, it was avoided by a smart duty officer. Afterwards, they found that the root of the false warning was also a software bug.

The examples given above are from years ago, when both hardware and software architecture were much simpler than the current ones. Modern devices are smarter e.g. self-driving cars, phones and smart factories, and embedded systems play important roles in these devices. This requires more advanced and higher performance systems. The rapid advances in System on Chip (SoC) have led to powerful multicore/manycore systems. For instance, it is said that today's cell phones have more computing power than the computer NASA used to go to the moon in the 1960s [3]. As a side effect, many new-coming issues are also introduced together with the application of multicore/manycore systems, e.g. limited observability & traceability, sporadic timing issues, shared resource contention, multicore synchronization issues and complex hardware configurations in modern embedded systems. Several new issues are described as examples in the following.

As the number of cores and the core clock frequency increase, the amount of trace data generated per second also explodes. Modern single-core system already produces more than 10 gigabits per second of raw trace data [4]. The tracing bandwidth for multi-core systems should be much higher than for the single-core as the number of cores multiplies. However, the number of pins and the bandwidth of pins are not following the same trend as the required tracing bandwidth so not all data can be traced continuously without halting the system. The observability and the traceability become more limited due to this bandwidth bottleneck.

Many issues are sporadic, relying on the system timing that is sensitive to software instrumentation. Setting a breakpoint or modifying software changes the timing, leading to a different system behavior, which impedes the debugging process. For such issues, the debugging solution is supposed to be non-intrusive.

Contention for shared resources emerges after the introduction of multicore/manycore architecture. When two cores access the same shared resource simultaneously, usually one core has to wait until the first one finishes, if this shared resource does not support accesses in parallel. Contention delays the access and has an influence on the system performance. Many contention issues are underestimated as they are usually transparent to software developers and also are not detected easily by conventional tools.

The conventional debug methods, for example, setting a breaking point and stepping through a program fulfill the basic debugging requirements, but they are not efficient in handling new coming challenges. First, many conventional debug methods change the timing behavior of systems, which may hide bugs and cause heisenbugs because the changes in timing can obstruct the reproduction of bugs. To deal with multicore concurrent debugging, concurrent execution control is needed. Usually the solution is that one more selected core can be halted by external pins. External pins can be used to signal other cores to halt. However, this solution suffers from significant latency problems [5]. Setting a break point cannot ideally halt all hardware modules at the same time in a multi-core system. Therefore, it is not suitable for handling timing-sensitive issues. Second, conventional debug solutions are not efficient to deal with certain new issues [6] such as performance issues introduced by multi-core architecture. Some new-coming issues are sporadic, e.g. deadlocks and priority inversions. Manually debugging sporadic issues is time-consuming or even not possible. Some have no obvious symptoms e.g. improper type of locks and inefficient lock granularity compared to the conventional functional bugs. Finally, knowing where to explore is extremely challenging in a complex system with limited observability. A normal debug process works in this way: observe symptom, look into related parts and find issue. This process is not easy to apply to a highly complex system with many inter-dependencies. Meanwhile, conventional debug tools need precise human control to configure the analysis scope, which is not feasible for complex multi-core debugging.

Two complex real-life examples that are hard for conventional debugger to deal with are introduced below. As the complexity of SoC increases, the integration of more and more hardware components like peripherals into a single chip resulted in complex rules governing the configuration and the use of these components. Not completely following the rules of proper hardware usage can result in functional bugs and performance bugs. A real life example for Infineon's AURIX microcontroller: Clock Control Unit (CCU) in AURIX is responsible for the clock distribution. The clock configuration is changed by writing several control registers. The UP bit in the control register works as an update switch for activating the whole configuration consistently when the last configuration register is written. If this field is not set, the CCU still works in the previous setting, though the registers show the newly written values, which is misleading for developers. The issue with this configuration is that the register does not show the actual state when the UP field is neglected. A conventional debugger may also contain a feature to show the current clock configuration but it is usually based on the current misleading register values. The wrong displayed clock states provide the software developer with wrong information, which could lead to a wrong direction. Another example about complex hardware configuration is about flash wait state configuration, which relies on the system clock frequency. An improper flash wait state configuration will either result in sporadic bus errors or lower performance. A conventional debugger is not capable of handling such issues.

1.2 The proposed solution

As mentioned in the above section 1.1, conventional tools are not able to address the new challenges brought by multi-core architectures efficiently. A novel debug methodology is needed to help software developers to conquer the debugging of complex software in multi-core embedded systems. In the following, a methodology is proposed to deal with the mentioned new challenges.

1.2.1 Methodology – embedded health

The goal of this research is to create a novel system diagnosis methodology that can automatically diagnose certain new-coming system issues which are rarely covered or not well addressed by conventional solutions, to increase the debugging efficiency. This research covers the debug challenges in the area of shared resource contention, atomic violations, memory locality, lock profiling and hardware configuration validation for embedded systems etc. Based on this research, a novel debug methodology is designed and implemented to automatically detect these issues in embedded systems.

Modern embedded systems are becoming more complex as the number of cores and peripherals increases. The software complexity also explodes in the same trend. Embedded systems are complex from both software and hardware perspectives but even worse, the observability is limited. A simple question whether there is anything wrong inside of an embedded system is not so easy to answer anymore, let alone the question of what causes the wrong behavior. How can we diagnose issues in a complex system?

This question of how to figure out what is wrong in a complex system is not new for doctors in the medicine field who have been facing the same challenge for thousands of years. They found that the symptoms of diseases are depending on individuals. For instance, catching a cold for some patients means sniffling while for the others may be sneezing. Only relying on the superficial symptoms to diagnose a disease is not very reliable as there are also many diseases sharing the same symptoms and symptoms are individualized on different persons. Accordingly, doctors invented many diagnosis methods such as blood tests, X-ray tests, Magnetic Resonance Imaging (MRI) and biopsies. One common characteristic of these diagnosis methods is that they are not only relying on the superficial symptoms but also the radical causes. One typical successful well-known example is a blood test, via which many diseases such as leukemia, hepatitis, malnutrition etc. can be diagnosed. Blood tests are standardized and can be conducted for different patients and different diseases. Many reference ranges for blood tests that are usually given as what are the usual values of substance levels found in the population are created to help to diagnose physical problems[7]. They may also be called standard ranges and work as the indicators of issues. Many diseases caused by virus are also detectable via a blood test.

With many similarities in complexity, a complex embedded system is also limited in observability as a human body that cannot be opened to see what is going on as wishes. Bugs in the system are similar to diseases in body. The lessons "Do not only rely on the superficial symptoms but also the radical causes." learned by doctors can also help to debug complex embedded systems. Therefore, the root cause of issues should be focused via the limited observability.

For embedded systems, hardware tracing in the embedded field provides certain observability like the blood test in the medicine field. It also brings the internal information in a system to the outside world for analysis. Hardware tracing is widely supported by many Commercial Off-The-Shelf (COTS) devices e.g. Infineon's Multi-Core Debug Solution (MCDS), ARM's Coresight and Freescale Nexus 5001. The issue diagnosis in embedded system should not depend on specific superficial symptoms but on the internal root causes. Therefore, similar to blood test indicators, indicators that are extracted from hardware-related and low-level information are designed for embedded systems. Different indicators are supposed to show different system issues corresponding to different diseases in medicine. In a human body, usually the concentration level of a specific substance in blood tests is an important indicator. The indicators in the embedded system are usually patterns e.g. a memory access pattern and collections of various kinds of information. The design of indicators is the key in this methodology.

Based on the blood test idea, in this thesis a trace-based automated system diagnosis and debug methodology as shown Figure 1.1 is proposed to cover issues that cannot be efficiently solved by conventional tools. It makes use of hardware tracing and it is aimed to deal with various system issues instead of limiting to a specific issue. The hardware tracing provides hardware-level trace data including program flow, bus transfers, memory accesses etc. As the trace data provides the execution history and the execution modules, the time and location information is intrinsically contained. Based on such low-level and hardware-related information, various indicators are designed to detect different target system issues e.g. data memory contention, program flash contention, hardware configuration issues. The indicators are combinations of different types of information depending on time and location.

The diagnosis process is generally applicable and automated, meaning that it should be applicable for a wide range of issues similar to a blood test without pre-knowledge of where to investigate. It should be effective for most applications and the involvement of developers should be minimized, so only the basic debug information e.g. an .elf file is needed as the input for the methodology. Depending on the type of analysis, the tracing hardware is configured accordingly and performed by the methodology. The

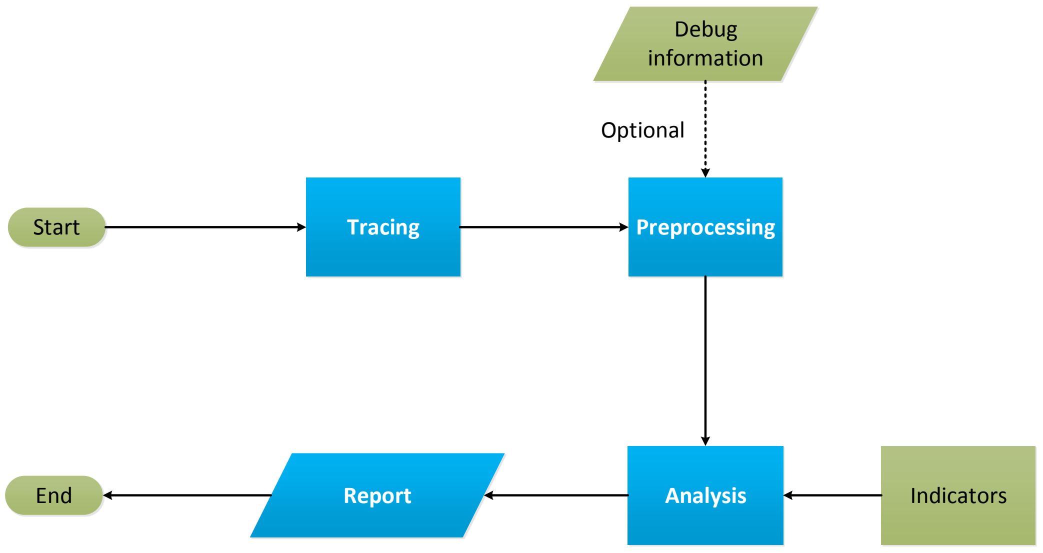


Figure 1.1: The generic flow chart of the proposed automated system diagnosis and debug methodology

collected trace data is processed and analyzed with the help of pre-defined indicators and a report is created by the methodology for software developers to give hints of issues and suggestions. In this way, automated diagnosis can be conducted and many system issues can be detected automatically. The proposed methodology applies post-processing analysis, which means that trace data is first collected by the tracing hardware during execution and then analyzed by the proposed methodology offline.

For a certain issue, one indicator or several indicators are usually designed to detect it. The generic design process of indicators for a certain issue is illustrated in Figure 1.2. First, the target issue is analyzed at the hardware level. A coarse indicator is extracted and then tested to make sure the target issue can be effectively detected. If not, either the refinement of the existing indicator or the extraction of a new indicator can be applied to increase the detection rate. The iteration ends until the designed indicators are effective enough for the issue detection. The details of indicator design are described in the following chapters.

1.2.2 Implementation – ChipCoach

Based on the proposed automated system diagnosis and debug methodology, a tool named ChipCoach was designed and implemented for Infineon's AURIX devices. The purpose of ChipCoach is to provide an efficient and convenient solution that detects system issues in seconds. Several experiments were also conducted on ChipCoach to prove the feasibility and effectiveness of the designed indicators. ChipCoach runs on Windows and currently supports Infineon microcontroller AURIX and AURIX 2G devices. The

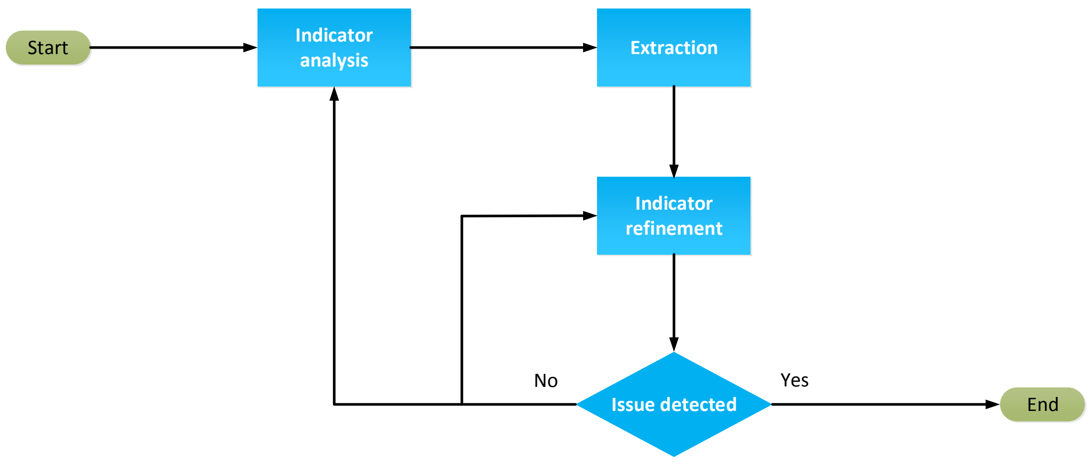


Figure 1.2: The generic design flow of indicators

Table 1.1: The overview of features in ChipCoach

Features	Target issues
Clock configuration visualization	Clock configuration issues
DMA activity visualization	DMA related issues
Data memory analysis	Data locality and memory contention
Hardware configuration validation	Hardware configuration issues
Lock profiling analysis	Lock issues
MPU configuration visualization	Memory protection issues
Program flash contention analysis	Program flash contention
Performance counter statistics	Performance issues

main part of ChipCoach is programmed in Java and the lower layers are in C/C++. It is based on Device Access Server (DAS) and MCDS Trace Viewer (MTV) which will be introduced later. There are separate views in ChipCoach and they are almost independent. Each view is focused on a special type of issues. The features cover functional issues and performance issues as show in TABLE 1.1.

Basic features including Direct Memory Access (DMA) channel activity visualization, memory protection configuration visualization, clock tree configuration visualization, performance counter statistics benefit the exploration and the understanding of the system. Advanced functions including data memory contention analysis, program flash contention analysis, hardware configuration issues detection, spinlock profiling are specialized in a specific issue.

ChipCoach can be applied as a generally applicable validation and debug method like a blood test in software development process. It can be run even when software developers do not know if there is an issue and which issue it is. Different analyses are conducted one by one automatically and finally a report is generated to show the health status of the target system.

ChipCoach based on hardware tracing is designed to detect different system issues. So far ChipCoach supports Infineon's AURIX family, which provides hardware tracing facilities named MCDS. MCDS is able to trace a system non-intrusively, meaning no software instrumentation and no impact on normal execution. During the tracing phase, MCDS is configured by ChipCoach according to the type of analysis. The trace data is transferred to the computer. Then the trace messages containing hardware-related operations are linked to binary and symbolic information available in the .elf file. In the indicator extraction phase, the trace data is compared against the pre-designed indicators. Finally, a report is generated to show the detected issues based on the comparison. The user involvement and information from the user side are minimized, facilitating the analysis automation. The report contains the information of the issues and root causes. In some functions, suggestions to solve the issues are also proposed by ChipCoach. ChipCoach has been used widely by internal application engineers and it is being commercialized by a tool partner of Infineon.

1.2.3 Contributions

In this thesis, an innovative system diagnosis and debug methodology based on hardware tracing is proposed to bridge the gap between increased debug difficulty and inefficient debug solutions. The main contributions of this thesis are as follow:

- The proposed methodology makes use of hardware tracing, which is non-intrusive meaning that the normal execution is not impacted. This feature is preferred by real-time systems because changes in timing also impact the behaviors of the system.

- The indicators of the methodology are designed for the hardware platform instead of applications. No training is needed. The only information needed can be the .elf file which contains binary and symbolic information.

- It focuses on many specific issues that are usually underestimated and rarely covered by other solutions, e.g. program flash contention.
- It detects the system issues and reports the root causes of the issues. The detection relies on the root causes instead of the superficial symptoms. It is able to detect issues even before the symptom appears and estimate the severity of the issues.

Hardware tracing has the advantage that no software instrumentation, binary instrumentation or source code instrumentation is needed. The observed timing behavior of the system is not impacted by the observation itself, which provides more advantages over software instrumentation. As timing is critical for embedded real-time systems, different timing may result in different behaviors and therefore changes to timing should be avoided. For example, a heisenbug [8] disappears when one attempts to observe it via debugger or software instrumentation. The proposed methodology uses hardware tracing and does not modify software, facilitating the analysis for real-time applications.

In order to boost the efficiency of the system diagnosis and debug tools, automation is usually applied. Therefore, human involvement is minimized. The proposed methodology requires only the .elf file that provides binary and symbolic information, and no input e.g. judgment criteria from user side is needed. It is a general solution of different applications and is not limited for a specific application.

As discussed above, many new challenging issues were introduced by multi-core/manycore embedded systems. These issues are not well solved efficiently or even not covered by existing solutions. The proposed methodology is focused on such issues or specific aspects, showing an innovative way to handle such issues.

Many existing debug tools detect problems based on the symptoms of the problems. This is a practical solution, but what will happen when there is no symptom at all? Merely relying on the symptoms is not sufficient and sometimes does not work well. The proposed solution also makes use of hardware tracing, observes the low-level hardware operations instructed by software, and derives issues from the root cause. In this way, many issues even without symptoms can be detected by this methodology. The detailed discussions are introduced in the following chapters.

2 Related Work

The emerge of multicore/manycore systems introduces many new issues that are difficult to be handled with the conventional debugging/testing tools. The gap between the developers' understanding and the actual status of complex multicore/manycore systems increases. In order to bridge this gap, many new methods have been invented and automated system diagnosis solutions provide an efficient way. In this part, different automated system diagnosis solutions for various issues are discussed in detail.

2.1 System diagnosis classification

There are many types of automated system diagnosis solutions. In this part, these solutions are introduced in different sections depending on analysis type and data collection method.

2.1.1 Analysis types

According to the relation between analysis and system execution, automated system diagnosis solutions can be classified into three categories, i.e. **pre-execution analysis**, **runtime analysis** and **post-execution analysis**, as shown in Figure 2.1. Pre-execution analysis is also named as **static analysis** meaning that it analyzes the program without actual execution of programs. Compared to static analysis, runtime analysis usually conducts the analysis while the system is running. Accordingly, post-execution analysis is a kind of **post-processing analysis** method or postmortem analysis, which is usually based on the data collected during runtime. In the following, the related research belonging to the three types is described in detail.

2.1.1.1 Static analysis

Static analysis has several advantages over dynamic analysis. It adds no overhead to the normal execution and its analysis scope is not limited by the execution path. Many quite specific issues are covered by static automated diagnosis tools.

A performance analysis tool named CARAMEL was designed to detect and fix performance bugs that are related to a scenario with a condition and a loop [9]. When the condition turns true during the execution, the remaining computation performed in the loop will be wasted, resulting in a performance loss. This type of issue can be fixed by simply adding a break line.

Static analysis is not only applied to diagnose performance issues but also functional issues. Polyspace [10] is a static analysis tool to detect and prove the absence of certain

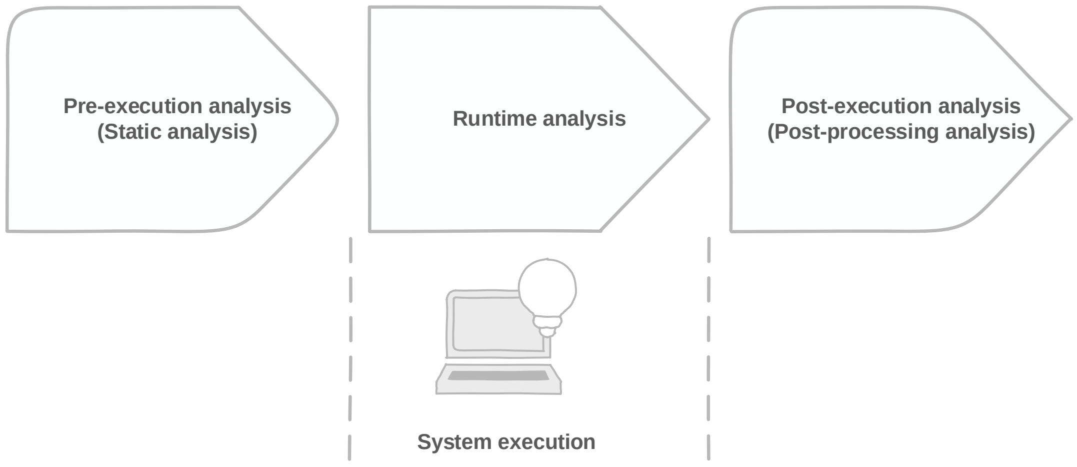


Figure 2.1: Three different types of analyses

run-time errors for C, C++ and Ada. It examines the code to detect issues such as arithmetic overflow, division by zero and others. Wang et al. proposed a static analysis method to show that code blocks are atomic, meaning that every execution is equivalent to the one in which the code block is executed serially, i.e., without interruption by other threads [11]. Atomic issues are critical in multicore systems because unexpected program interrupt may change its behavior, leading to malfunctions. Atomic issues are typical depending on timing and might result in a sporadic behavior. There is much research conducted in this field using static analysis. For example, Flanagan et al. created a technique to verify the atomicity of some problematic cases by applying reduction to an abstraction of the program [12]. They presented the abstraction notations based on purity and insatiability. The correctness of an atomic procedure can be verified using sequential reasoning. Another common functional issue is caused copy-paste operations by developers. In the software development, copy-paste is widely used by developers. However, copy-paste is also error-prone so a tool named CP-miner using data mining techniques to efficiently identify copy-pasted code in large software suites and detect copy-paste bugs was implemented [13]. This tool is token-based and the program is divided into a stream of tokens. Duplicated sequences are then identified.

Static analysis is also applied to deal with issues in the embedded field. For instance, a user-supplied compiler extension [14] was designed to pinpoint the cache coherence errors in embedded flash, which show up sporadically and are difficult to find. It checks the source code and compares the code against the invariant defined by users.

Static analysis has many advantages but it is also limited in several aspects. It checks all possible program paths even including paths that will never be executed, leading to many false alarms. Many issues depending on timing are not easy to be detected via static analysis for instance shared resource contention.

2.1.1.2 Runtime analysis

Compared to the static analysis, automated system diagnosis methods based on runtime analysis typically have execution overhead. They are able to collect various execution information during runtime and detect different problems. There exist a lot of research and methods focused on runtime analysis. In the following, some research and methods are introduced depending on the target issues.

There are two types of issues including functional issues and non-functional issues. Non-functional issues in this dissertation are mainly about performance issues. Various causes can lead to performance issues, e.g. low performance algorithm, inefficient lock usage, shared resource contention etc.

2.1.1.2.1 Performance issues

Shared resource contention introduced by multicore architecture degrades system performance. The contention is located in many shared resources e.g. memory, bus and I/O. It attracts much attention of researchers owing to its high importance. General solutions dealing with general shared resource contention (not specific to a special resource) detect shared resource contention and also estimate the performance impact caused by contention. For example, Dey et al. [15] proposed a general methodology for the characterization of multicore applications based on experimental comparison. In their methodology, two different configurations namely one baseline configuration and one contention configuration are executed and the runtime difference is compared to derive the impact of shared resource contention. However, it is impractical to obtain such two different configurations just to estimate the performance difference. Kumar et al. [16] designed a probabilistic approach to estimate the performance impact due to contention based on real execution, which is more accurate and much faster than simulation. The duration of shared resource usage by one application is analyzed. Then the blocking ratio that is defined as the chance of a core blocked by another core occupying the busy resource is calculated based on uniform distribution. The expected waiting time caused by shared resource contention is derived.

Such general solution estimates the performance impact at the application granularity. More detailed contention information requires more precise analysis methods. A very sophisticated hardware approach [17] to build accurate Cycle Per Instruction (CPI) stacks was created to use hardware counters to measure the waiting time for each shared resource. It makes use of a top down approach that overcomes the previous bottleneck: counting contention events without considering how these contention events affect overall performance.

The impact of a specific shared resource contention is also interesting. For example, data cache, that has a great influence on the system performance, is studied by many researchers. Chandra et al. [18] proposed three performance models that predict the cache impact on the performance on co-scheduled threads. The input of these models is the stack distance information and circular sequence profiles of each thread. This information can be collected during runtime. Scheduling as an attractive tool to deal with shared resource contention, draws a lot of attention [19][20].

Contention does not only happen at the interface of hardware modules but also at software level. Lock is a common method to sequence the accesses from different threads to a shared resource. Contention may also happen when several threads are competing for the same lock, leading to performance degradation. Therefore, lock contention should be avoided. Scheduling has been utilized to relieve lock contention. For example, Pusukuri et al. [21] proposed a scheduling framework named Shuffling that likely migrates the threads accessing a common lock to the same socket. This is because the threads competing for a common lock on the same socket may result in less cache misses in Last Level Cache (LLC), meaning better performance compared to the scenario with a remote socket. Heavy lock contention will lead to scalability collapse [22]. Based on this observation, Cui et al [22] designed a scheduler which monitors each thread's percentage of lock waiting time continuously. If the waiting time is above a predefined threshold, the monitored thread is migrated to Special Set of Cores (SSC), in order to avoid scalability collapse. Some lock contention is introduced by improper software design. For example, unnecessary lock contention is analyzed in the research [23]. The goal of this research is to detect the false inter-thread dependencies, which causes unnecessary lock contention. It records the program execution and finds the unnecessary lock contention patterns based on the tracing.

Data cache false sharing is also one of the common performance issues. Different from lock contention, false sharing is typically invisible in the source code. In order to detect false sharing, tools are designed to detect it during runtime. For example, SHERIFF-DETECT [24] finds false sharing issues by comparing cache updates by different threads in the same cache line, and ranks them according to the performance impact. The performance impact can also be reduced by migrating false sharing from different threads to separated physical addresses. To detect latent false sharing, Liu et al. [25] proposed a tool called PREDATOR. It is able to precisely predict false sharing including latent false sharing problems that can happen later but are not observed during the current execution. This latent problem detection ability overcomes the limit of other detection approaches [25]. PREDATOR performs instrumentation at an intermediate presentation level.

The above solutions are specific to one performance issue. There are also general solutions for performance issues. Eyerman et al [26] proposed the speedup stack, which quantifies the impact of the various speedup limiting factors of multithreaded application in a single stack. These limiting factors include synchronization, contention, cache coherency, imbalance etc. This solution provides a reference to analyze parallel performance and identify scaling bottlenecks. It modifies the hardware modules to count the cycles that are wasted due to various reasons [27].

2.1.1.2.2 Functional issues

Regard functional issues, automated tools that locate and correct erroneous programs will reduce the cost of software development significantly. Fault localization is a process to find the location of a fault given failed executions and passed executions. Many automated tools are focused on this process. For instance, a statistical approach to localize software faults without prior knowledge of program was designed by Liu et al. [28]. This approach improves the previous methods in modeling the divergence of predicated evaluations between correct and incorrect runs by a hypothesis testing-based approach. Moreover, another fault localization method, which is able to identify the state differences that cause the failure by searching in both space and time, was proposed by Cleve [29]. These two approaches are focused on fault localization. A step further than fault localization is automated fault correction. For example, the approach by He [30] et al. automatically corrects an erroneous statement in a faulty function, based on the assumption that the correct specification of the erroneous function is available in the form of preconditions and postconditions of the function. This approach combines ideas from software testing and weakest preconditions used in correctness proof methods to locate a likely erroneous statement.

2.1.1.3 Post-processing analysis

Runtime analysis typically has overhead on the normal execution. The overhead can be relieved by migrating the processing part afterward, which means post-processing also named as postmortem processing. Post-processing techniques can be applied to different aspects. A tool called Memphis based on Instruction Based Sampling (IBS) [31] supported by AMD is to pinpoint the low performance memory accesses including remote accesses and contended accesses. A post-processing part of this tool interprets the raw data collected by kernel. OProfileBM [32] is a Linux system profiler linked to standalone applications that run on bare metal cores without an operating system and collects samples in the background during runtime. For critical lock analysis [33], a new method for diagnosing critical section bottlenecks in multithreaded applications was introduced to improve the overall performance. Only the critical sections which appear on the critical path that have a direct impact on the completion time of multithreaded applications are related to the overall performance improvement. Post-processing tools are also utilized for detecting functional errors. For example, AVIO-S [34] was proposed to detect concurrent bugs based on a novel observation called access interleaving invariant. Compared to AVIO-H, AVIO-S is more suitable for bug detection and postmortem bugs.

2.1.2 Data collection methods

Data collection is an important part in analysis. Several classes of data collection methods are sorted according to the way of collecting data. Various data collection methods including software-based, hardware-based, hybrid-based and simulation-based have different characteristics. Depending on the method applied, certain aspects are sacrificed, for example, usually execution overhead is added to the analysis based on a software-based method. For each class, several examples related to system diagnosis are briefly introduced.

2.1.2.1 Software-based

Software-based method is also named as software instrumentation. It means that additional software functions are inserted to the normal software product to diagnose errors and trace information [35]. Typically, software instrumentation causes additional execution time and changes the system timing. Two types of software instrumentation approaches are available either source code instrumentation or binary instrumentation.

A tool called HAVE [36] was implemented to tackle atomic violations in Java program based on source code instrumentation. In this tool, static analysis is first performed to obtain summaries of synchronizations and accesses to shared variables. Then dynamic analysis is applied to refine the analysis results e.g. branches that are not taken, to reduce the number of false positives. Another tool named Pin [37] that performs runtime binary instrumentation of Linux applications was designed to provide an instrumentation platform for building a wide variety of program analysis tools.

The overhead incurred by software instrumentation is harmful to the system analysis because any change to the execution time may cause changes in scheduling, resulting in very different timings. Therefore, minimizing the overhead is extremely important especially for hard real-time applications, the timing of which is critical.

2.1.2.2 Hardware-based

To reduce the overhead caused by software instrumentation, hardware-based data collection methods are applied. The hardware of hardware-based solutions can either already exist in products or be specially designed upon request.

The specially designed hardware is flexible and has the advantage to collect different types of information at the cost of die area. A novel implementation of hardware performance counters for building accurate CPI stacks was proposed by Eyerman et al. [17]. Special hardware counters are added to the hardware module interfaces. Detecting data races by software solutions generally incurs large overheads so that hardware support for race detection is applied in the research proposed by Zhou et al. [38].

The already existing hardware is able to collect a predefined range of information. An example [39] to make use of existing hardware support in COTS devices is about memory placement in Non-Uniform Memory Access (NUMA) systems. It considers NUMA together with contention in kernel level and optimizes performance by several measures. The data collection is supported by IBS in AMD and Intel, leading to much lower overhead than software-based solutions.

Hardware-based data collection method is usually limited by the hardware implementation. As platform users, they may not have the freedom to change the hardware as they wish, and thus making use of existing supported hardware is a preferred approach.

2.1.2.3 Hybrid-based

A hybrid-based data collection method usually combines both advantages of hardware-based and software-based methods such as low overhead and good flexibility. For example, Huang et al. [40] noticed that lock profiling tools usually store profiling data directly into local memory which results in significant memory interference on normal programs' execution. Therefore, they proposed a hardware-assisted lock profiling approach called HaLock which combines software-based lock detector and hardware-based trace collector, achieving negligible overhead. Another example is AVIO-H [34] which extends several additional bits to the cache coherence hardware to detect atomic violations with negligible overhead compared to AVIO-S that is a pure software-based solution.

2.1.2.4 Simulation-based

Simulation is a powerful approach to diagnose systems because of the high observability and the high flexibility. It can be used to detect different issues. For instance, Lagraa et al. [41] proposed an approach making use of a virtual prototyping simulation and data mining to find data memory contention. Latent performance bugs severely degrading end-to-end system performance in distributed systems can be detected by the research [42] based on simulation. Memory system behaviors for both parallel and sequential programs and even memory performance bottlenecks are explored by a simulator-based tool called MemSpy [43].

The disadvantages of simulation are that the detailed models of commercial embedded systems are typically not available and the plant models that interact with the simulated embedded systems are also missing. Compared to on-chip analysis, simulation costs much more time especially when more details are included in the output of the simulation.

2.2 The automated "blood test" in system diagnosis

In this dissertation, a system diagnosis methodology is proposed based on the hardware tracing, meaning that no software instrumentation is needed. The utilized hardware tracing method is supposed to be intrinsically supported by COTS system e.g. Infineon's MCDS, Freescale's Nexus and ARM's Coresight. Such kind of hardware tracing e.g. Infineon's MCDS has no impact on the normal execution. Therefore, the system timing is not modified by the observation and this feature is preferred by hard real-time systems where the timing is critical. The proposed method is a post-processing method. The execution data is collected during runtime and analyzed afterwards by the host computer.

The proposed methodology proposed in this dissertation is inspired by the blood test as described in the previous chapter. The blood test is a standard method that automatically detects diseases in a complex system and covers various diseases. This methodology is designed to have similar characteristics.

Compared to other research, the proposed methodology has the following advantages. First, it is able to automatically detect with a wide range of system issues from performance bottlenecks to functional errors. The other research usually focuses on a single problem or a small group of problems. Second, the methodology is capable of solving many issues that are not emphasized by other research. For instance, the design of indicators in this methodology contains low-level hardware information. This facilitates the analysis many issues that are closely related to the hardware. Third, the proposed methodology is independent of the applications of embedded systems and is based on hardware tracing. The design of indicators doesn't rely on the applications and the pre-knowledge of the applications. With the help of hardware tracing, software instrumentation can be avoided. These features make the automation of system diagnosis easier. The details of these three points are described below in detail.

Most automated system diagnosis research focuses on a specific issue or a small group of issues. For example, a famous tool named "Eraser" [44] that dynamically detects the data races in lock-based multi-threaded programs. It uses the binary rewriting to monitor the shared memory reference and verify the locking behavior. It is designed to only deal with data races. Similar examples are SHERIFF [24] or AVIO [45]. Some research is targeted at a group of issues. For instance, several papers from Eyerman et al. based on a similar methodology are mainly focused on performance problems. It is able to provide the insight into the scaling behavior on multi-core hardware [26], detect the influence of inter-thread on the performance [27] and compute the CPI loss caused by multi-core effects based on hardware counters [17]. The target of this series of research is to detect the performance impact by multithread and multi-core systems. Another example is the BugFix [46] that automatically analyzes the debugging situation and reports a list of relevant bug-fix suggestions. It requires as input a faulty program and a corresponding failing test case. The target scope of this tool covers functional errors. Compared to the above research, the proposed methodology has the capability to solve many various problems. For example, it is able to detect performance issues such as program flash contention [47] and data memory contention [48]. It detects functional errors such as hardware configuration errors [49] and lock usage errors [50]. Moreover, it has the potential to automatically diagnose many other issues especially issues in the low-level. By designing new indicators for new groups of issues, this methodology will gain the ability to analyze new problems, which is easy to extend.

Owing to the applied data collection method, the proposed methodology is able to collect much low-level information and solve many issues closely related to hardware. Many of such issues are rarely emphasized by the existing research. For instance, shared resource contention in multi-core systems is well studied but most research is focused on the data memory or bus contention instead of program flash contention. Program flash contention also influences the system performance in multi-core systems [47]. The impact of program flash contention and the performance impact estimation are not studied by the other research. A similar story happens to the hardware configuration problems. The hardware configuration problems are time-consuming to debug but they are also not thoroughly studied [49].

Most tools for automated software debugging needs the pre-knowledge of the running software. For example, many lock profiling tools like ANOLE [51], HaLock [40] or LiMit [52] are designed for a specific Operating System (OS) or a virtual machine. Some tools like BugFix [46] or Atomizer [53], need the manual input or manual annotation by the software developers. The indicators in the proposed methodology are independent of the software and the OS. This independence provides advantages in embedded systems as there are no dominant OSes like Windows or Linux on PC. The methodology is based on the trace data collected from hardware tracing and the data collection can be non-intrusive. It is also preferred in the analysis of hard real-time systems because the timing of the systems is critical.

2.3 Target issues of the proposed system diagnosis methodology

In the previous part, the comparison of the methodology with the related work in a high level is given. In this methodology, the indicators play a very critical role and each indicator is targeted at a specific problem. Therefore, the related work of different target issues is described in this part. The proposed methodology is designed to conquer many multi-core issues that are not yet well solved. These issues cover many different aspects such as hardware configuration, multi-core contention, synchronization etc.

2.3.1 Hardware configuration issues

Embedded systems are usually equipped with many different types of peripherals. The peripherals are supposed to be configured properly in order to have the expected functionalities. However, there are rules that define the right procedures to configure in the long tedious user manual and misconfigurations may happen when these rules are not correctly followed. Misconfigurations can lead to issues such as malfunction, low performance and even sporadic errors. In this dissertation a methodology focused on hardware configurations is proposed.

It is based on Linear Temporal Logic (LTL), which is applied as a standardized rule format in the methodology. The related research about LTL is first introduced. Then research related to runtime monitoring/verification is described.

2.3.1.1 Linear Temporal Logic

LTL is a specification language for programs [54]. Various types of temporal logic were created, e.g. trace propositional temporal logic (TrPTL) [55]. For software debugging, a temporal debugger was applied in the work [56]. LTL is used in this work to control stepping through different states of a concurrent program. It detects finite sequences of states that satisfy a predefined given specification and then stops the system at that state. In our approach, LTL is applied to describe the legal register access sequence rule.

2.3.1.2 Runtime Monitoring

Rule-based runtime monitoring & verification have been studied for decades. The characteristics of embedded systems such as limited resources and limited observation points, introduce particular challenges in runtime monitoring/verification. To conquer such challenges, a plenty of methods have been applied including hardware, software or hybrid probes [57]. In the following, research related to rule-based runtime monitoring/verification is introduced.

The monitored system executions are checked against the properties, which are often derived from the software requirement specification [57]. One concern of runtime monitoring/verification method is the overhead to normal execution. The additional execution time caused by runtime monitoring/verification does not only make the execution slower but also changes the timing and scheduling, which also changes the system behavior. Therefore, some research is conducted to reduce the execution overhead. A framework was presented in [58] to reduce the intrusiveness of the runtime monitoring. The proposed framework applies hardware and hybrid probes, which provide system-wide observability and minimize the impact on normal executions. The hardware area requirement is also reduced incorporating the presented framework. Barringer et al. [59] proposed a finite tracing monitoring logic called EAGLE. Its purpose is to check whether the program execution during runtime conforms to the requirement specification, involving rule definition, manipulation and execution. In the rule definition part, LTL is applied to describe the requirement specification. Based on this, another monitoring method RULER [60] was then proposed to improve the simplicity and efficiency as well as to compile a wide range of temporal logic and other formalisms.

Hardware-related issues that the proposed method focuses on are also studied by other researchers. For example, a runtime monitoring system for real-time embedded system was presented in [61] and it makes use of a special FPGA device plugged to a peripheral bus to observe the bus transfers including writes and reads. The purpose of this approach is to guarantee the safety when a violation is detected, by restoring the system to a safe state. Compared to this work, this method is targeted at detecting hardware configuration issues.

Debugging deeply embedded peripherals is challenging due to the limited observability. The research [62] has shown that hardware/software interface debugging is the most time-consuming step and shortening the debugging time is a key to shorten the total design cycle. Peripheral debugging and hardware configuration validation have been studied in several papers. An assertion-based online debug environment was raised by Peterson et al. [63]. It is designed to handle the challenges of debugging high-speed complex SoCs. It requires additional hardware and also hardware changes to the existing SoCs. The focus of this method is on the hardware validation and testing instead of software debugging. Hardware probes are inserted to SoCs and validated by an external FPGA.

Compared to the above work, the proposed method distinguishes itself in several aspects. First, it covers both the SoC development phase and the software development phase. The rules are directly input by SoC designers and are valid for a specific device regardless of applications. Usually runtime monitoring & verification also make use of rules that are extracted from the software specifications only valid for a particular application. Second, this method applies hardware tracing, meaning no extra hardware is needed. One advantage of hardware tracing is that software instrumentation is avoided, no change of the system timing. Third, the methodology is focused on hardware configuration issues, which usually takes a lot of time to be detected but are not thoroughly studied. In this approach, an automated technique is performed to increase the productivity.

2.3.2 Lock issues

For multi-core systems, locks are widely used to regulate the accesses from different tasks to the shared resources. The efficiency of locks is an important factor of system performance so many profiling methods are introduced for various operating systems. In this dissertation, a profiling method that profiles the lock performance independent of operating systems and reminds developers of improper lock behavior is introduced. To detect spinlocks, an understanding of different types of spinlock implementations is required. The findings on this are introduced in the first part of this section. After that the state of the art in the development of lock profilers is discussed [50].

Spinlocks have been studied for years as a classic synchronization method to regulate shared resource accesses for multithread systems. A primitive and simple spinlock implementation is the test-and-set spinlock. It always uses atomic operations to compare and modify the lock value, which is harmful to the performance of both the current spinning thread and the threads [64] from other cores. This is because atomic operations block the memory interface longer than normal memory accesses and other threads using this memory will be influenced. Due to this drawback, an improved implementation test-test-and-set lock was proposed [65]. It spins with normal a read operation first and only tries to apply atomic operations when the lock is potentially free. The performance of this implementation is better than the previous one but it still has many issues. In a cache-coherent machine, if several threads are waiting for a lock, only one will acquire it after releasing. The others' cache will still be invalidated, causing additional memory accesses and lower performance. Therefore, backoff spinlock was created to avoid such collisions [64]. However, all locks mentioned above do not guarantee fairness, so queue-based spinlocks e.g. MCS [66] were designed to fulfill this requirement. The MCS spinlock has a queue to sequence the lock acquisition, which is suitable for many threads competing for one lock [67]. There are many more lock types with different advantages and disadvantages. Recent research from Guiroux et al. [68] has found that no single lock is best for all applications. The best lock varies with the number of threads, the number of cores and the hardware architecture, so the insight into the system is needed to find a proper lock type.

Lock profilers are tools that give insight into the system and enable developers to improve lock usage. Many lock profiling tools for different applications are already available as shown in Table 2.1. Locks and wait analysis in Intel VTune [69] uses interrupts to sample basic information about threads and synchronization objects. VTune works for x86 architecture instead of embedded systems. HPCToolKit [70] utilizes sampling instead of software instrumentation. It can be applied from desktop to supercomputers [71]. To make use of hardware performance counters efficiently, Demme et al. created LiMiT [52] that instruments applications and provides fast access to performance counters without kernel calls, achieving a lower overhead. For the same reason, HaLock [40] is applied and a specific uncacheable memory address is reserved to record lock information so that the memory inference to normal memory operations is minimized. HaLock leverages a hybrid mechanism which combines software-based lock detector and hardware-based trace collector. Lockmeter [72] was developed and released as a Linux kernel patch to record

Table 2.1: Comparison of Lock Profiling Tools

Tool	*Data collection method*	*Target*
ANOLE [51]	Virtual machine instrumentation	Linux KVM
Free Lunch [74]	Virtual machine instrumentation	Java
HaLock [40]	HW-assisted kernel instrumentation	Linux
HPCToolKit [70]	Sampling	Desktop [71]
Jucprofiler [75]	Library instrumentation	Java
LiMiT [52]	Software instrumentation	Linux
Lockmeter [72]	kernel instrumentation	Linux
PEPs [73]	Application instrumentation	OS specific
Vtune [69]	Interrupt-based sampling	x86
Our method	Hardware tracing	Embedded systems

the spinlock usage by applications. A framework named PEPs [73] allows developers to manually annotate the interesting source code. Then timestamped event tracing per thread is collected during execution and presented to the developer. The target platform of PEPs is not specified but it relies on a specific OS. Lock profiling tools are also useful to virtual machines. In a kernel based virtual machine, a virtual CPU (vCPU) could be preempted while holding a lock. Then other vCPUs have to spin extra-long time to acquire this lock, resulting in a waste of performance. Zhang et al. proposed ANOLE [51], which instruments the kernel based virtual machine to minimize this issue. Lock profiling tools for Java applications also have been studied, where instrumentation can be either done in the Java virtual machine [74] or in Java libraries [75].

Compared to the above existing tools, the proposed approach is focused on binary semaphore spinlocks operating on a single variable without OS dependency. This feature is favored by the embedded field due to the fact that there exist many embedded OSs. There are no other studies focused on the lock profiling for embedded systems without OS dependency and the proposed approach is the first pure hardware-based solution. It makes uses of non-intrusive hardware tracing. In this way, software instrumentation and execution overhead are avoided.

Based on the lock statistics measured by the lock profilers, software developers can make their judgment on the lock usage. One step further, showing improper lock selection and inefficient lock behaviors directly by the profilers is more helpful. A recent study [68] has shown that choosing the right lock for a particular scenario is challenging as there is no clear guideline for developers to follow. For many applications, software developers have freedom to choose their lock types and they may choose an improper lock based on their hypothesis during programming. There are several studies existing. Lozi et al. [76] proposed a new lock type named remote call locking that is aimed to accelerate the execution of legacy software on multi-core systems. A special lock profiler is also developed to identify the existing locks that can benefit from remote call locking. A method by Pan et al. [77] forecasts lock contention before adopting another lock, providing useful information to make a decision. Our approach indicates basic improper lock selection and inefficient lock usage based on the hardware-level behaviors. It is flexible and widely applicable.

2.3.3 Interrupt performance issues

Interrupts are critical for timely handling of events in real-time systems. Unfortunately, interrupts are difficult to predict: they alter the program control flow and complicate the invariants in low-level code [78]. Therefore, an interrupt profiling methodology is introduced to provide the insight of the interrupt usage of embedded systems.

There are several papers focused on interrupts. Brylow et al. [79] proposed and implemented a static checker for interrupt-driven Z86-based software with hard real-time requirements. The proposed checker is based on static analysis and assembly instructions are detected. The control flow graph is then generated and interrupt latency analysis is conducted. The implemented checker was tested with a 1000-line benchmark. However, there are several limitations of this work. First, many false positives will be generated as this is a static analysis method. Some of them are caused by paths that will actually not be executed. Second, the proposed method does not consider many dynamic problems related to multi-core, for instance shared resource contention. Such issues will degrade the performance and this information is missing in this work. Third, the analysis becomes very complex when the size of software increases because many paths and branches will be generated exponentially. The Worst case execution time analysis of interrupt is of interest for embedded operating systems. For example, Carlsson et al. [80] developed a tool, the goal of which is to produce flow information graphs from a number of source code files and an .elf containing object code binaries. This tool applies also a static analysis method and has the intrinsic disadvantages of static analysis, as also the author claims in the conclusion that the tool is not sure whether the WCET is found and not comparable to physical tests.

Dynamic solutions to measure interrupts are also investigated by researchers. For instance, the interrupt handler performance is profiled by a method proposed by Moore et al. [81]. This method instruments the Linux kernel and uses the system performance counters provided by IA-32 processors. Interesting results show that the performance of interrupts does not scale proportionally to overall system performance. Software instrumentation is flexible but it changes the system timing of the instrumented system. Simulation is also a good solution to obtain the interrupt information as described in the research from Yu [82]. The purpose of this research is to calculate the Worst Case Interrupt Latency (WCIL). However, simulation has the problem that the simulation models and plant models are usually missing for COTS platforms. Also, the simulation speed will be slow for very accurate detailed hardware models.

2.3.4 Shared resource contention

Significant research has been done for shared resource contention [19, 83, 84, 85, 86, 87, 20, 88, 89, 27]. A general system-level solution considers contention of different shared resources. For example, Du Bois et al. [27] proposed a method that modifies the hardware modules to count the cycles that are wasted due to various reasons including memory contention, cache contention and bus contention. The research [85] distinguishes and measures the impacts due to sharing of different individual resources e.g. cache, bus and memory, by comparing the runs of different process patterns. The method in the paper [90] copes with the contention for shared cache and bus with modeling. Another solution is simulation that is time-consuming but is able to provide accurate detailed information. Lu et al. [83] proposed a SystemC modeling focused on shared resource conflicts while accurate timing information is provided and the simulation speed is also accelerated.

2.3.4.1 Program flash contention

Many solutions are only focused on the contention of a specific shared resource. Cache as an performance-critical module can be shared among several cores/threads. The cache contention heavily degrades performance [91, 92]. A fast and accurate shared cache aware performance model named CAMP was proposed [89] that considers distance histograms, cache access frequencies, and the relationship between the throughput and cache miss rate of each process It models both cache miss rate and performance degradation and provides information about cache behavior to improve system throughput and reduce power consumption. The research [93] is focused on the performance optimization by considering both data allocation and data cache contention for NUMA systems.

Memory contention is also of interest. Liu et al. [87] proposed the notion of memory access intensity to facilitate quantitative analysis of program's memory behavior in multi-cores. In other studies [19, 20, 88], contention-aware schedulers are used to minimize the impact on the performance from shared resource contention.

Shared resource contention is also considered in the worst case analysis field. For instance, Lampka et al. [84] proposed an approach for bounding worst case response time considering share resource contention especially memory contention. Pellizzoni et al. [86] introduced an analysis methodology to compute the upper bounds of the task delay due to data memory contention.

The above research performs the analysis of the contention problem via simulation [83, 86, 84], static source code analysis, statistical analysis [85], hardware modification [27] or software instrumentation [19, 87, 20]. However, none of them has used hardware tracing. Hardware tracing has advantages such as non-intrusive and no software instrumentation. Most research focuses on the data memory contention while this methodology is addressing program flash contention, which is applied to store instructions and constant data. Compared to other shared resource contention, program flash contention is difficult to measure but can cause significant performance degradation.

2.3.4.2 Data memory contention and data locality

The performance penalties incurred by both data locality and memory contention must be analyzed before performance optimization as data locality and contention are mutually dependent. In the proposed methodology, a new memory contention indicator is designed to show the memory contention penalties quantitatively, which is comparable to data locality penalties, providing a reference for performance optimization. It detects both data locality and memory contention. Related research about indicators for these two effects is introduced below.

The location of shared variables makes a difference in a NUMA system. Early research [94] shows that the location of data and code impact the performance of NUMA system. In order to evaluate the performance influence, the embedded generic allocator (EGA) [95] was proposed, in which different allocations are loaded and the performance difference is measured directly on the target hardware. Evaluation directly on the target hardware is practical but consumes much time especially when there are many allocation possibilities. Static analysis is applied to deal with this challenge. For example, in the study [96], remote accesses are recognized by creating a data access relationship graph between memory access instructions and data at compile time. Similarly, sophisticated interprocedural analysis techniques are used to determine such relation graphs for both static global variables and dynamic global variables in [97]. Another solution is dynamic analysis. The data locality analysis can be performed at runtime via e.g. software instrumentation, hardware monitoring and simulation. In the research [98] conducted by Diener et al. the kernel is instrumented to use the virtual memory implementation of operating systems to characterize the memory access behavior. The observed information can be then used to optimize the affinity of data and thread. Another study by Molina et al. [99] uses Simics simulator to collect memory accesses and to optimize the allocation.

Memory contention also degrades system performance. It depends on the system timing and the contention symptoms cannot be observed directly on commercial embedded systems. Therefore, contention metrics should be designed to indicate the contention level. The most common contention indicator is the LLC miss rate. For example, the miss-rate heuristic – a measure of LLC misses per thousand instructions in distributed intensity [19] is used as an approximation of contention of shared resources including cache, memory and bus. The core idea is that a higher miss rate means higher access frequency, leading to more load to shared resources and higher contention level. Similarly, Blagodurov et al. [100] also used this to detect memory contention. More sophisticated metrics are also created for example Pain [100], Animal [101] based on the stack distance profile. The stack distance profile is a compact summary of cache reuse pattern. For some COTS architectures like x86, IBS is supported. An indicator that includes more information such as memory access latency is designed based on IBS and this indicator is transferred to the scheduler by interrupts. The memory access latency shows whether there is a contention and how many clock cycles are added by the contention. In order to calculate the worst case execution time tightly, usually shared resource has to be considered. A measurement-based approach considering both memory and bus contention is proposed in [102] for worst case analysis. A further solution is to analyze the contended memory accesses by simulation. Virtual prototyping is applied to find memory contention patterns [41], which utilizes a data mining approach to facilitate the contention pattern extraction. To avoid the software instrumentation, hardware solutions are studied. For instance, hardware counters are implemented to count the number of wait cycles due to memory contention was implemented by Du Bois et al. [27]. They increment when a core is waiting for another core's completion. The accumulated values are accounted as memory contention metrics to calculate the performance penalties.

The performance penalties by these two data locality and memory contention should be analyzed in a quantitative, comparable way to provide a reference to optimization. Blagodurov et al. [100] discovered that state-of-the-art contention management algorithms fail to be effective on NUMA systems and may even hurt performance relative to a default OS scheduler. This is because, in order to avoid contention, data allocation is modified, which could result in increased remote accesses and additional cycles. A similar conclusion was drawn in [103], system software must take both data locality and memory contention into account to optimize performance.

The method proposed here is distinguished from other methods by three aspects. First, most previous research [19, 39, 31, 41, 99, 98] detecting data locality and contention makes use of software instrumentation or simulation to trace memory accesses, and none of them is utilizing non-intrusive hardware tracing. Software instrumentation is commonly used but it changes the timing behavior of the system, which is critical for hard-real-time and safety-critical applications. The cycle-accurate simulation is powerful, whereas the simulation model is not available for most commercial embedded systems. Moreover, the plant model is missing in such simulation. Pure hardware solution avoids software instrumentation but needs hardware modification that limits its usage. Second, a new contention indicator applicable for embedded systems is proposed to evaluate the contention impact quantitatively and precisely. The occurrence of remote accesses and memory contention and the overall penalties, including locality penalties and memory contention penalties, are also summed and compared, as a reference to performance optimization. Third, the proposed method is based on hardware tracing which is suitable for commercial embedded systems. No hardware modification or special hardware e.g. IBS is needed. The only hardware support necessary is tracing hardware already supported by many COTS embedded systems as mentioned above.

3 Preliminaries

3.1 TriCore

TriCore is a 32-bit microcontroller architecture, optimized for real-time embedded systems. It unites the elements of a Reduced Instruction Set Computer (RISC) processor core, a microcontroller and a DSP in one chip package [104]. The Instruction Set Architecture (ISA) supports both 16-bit and 32-bit instruction formats. TriCore avoids long multi-cycle instructions and provides hardware-supported interrupt to reduce the interrupt latency and real-time responsiveness [105].

The TriCore architecture has 32 general purpose registers, Program Counter (PC) and two information registers as shown in Figure 3.1. A$[0_H - F_H]$ store the address information and D$[0_H - F_H]$ store the data. The Previous Context Information Register (PCXI), Program Status Word Register (PSW) and PC registers are used for storing and restoring a task's context. There are two groups of registers namely upper registers and lower registers. Lower registers include A$[0_H - 7_H]$ and D$[0_H - 7_H]$ while upper registers include A$[8_H - F_H]$ and D$[8_H - F_H]$.

However, different terms namely lower context and upper context are used for context switching. Upper context consists of A$[10_H - F_H]$, D$[8_H - F_H]$, PCXI and PSW. Lower context has A$[2_H - 7_H]$, D$[0_H - 7_H]$, A$[B_H]$ and PCXI. These two groups are used for different scenarios. For example, a context switching usually occurs when an event or instruction causes a program execution break. The current execution state shall be stored in order to continue to run the previous program. Upper context will be saved automatically due to interrupts, function calls or traps. Lower context can be saved explicitly by instructions e.g. Save Lower Context (SVLCX).

The 32 general purpose registers usually work as the operand in the instruction set. Some registers are explicitly stated in the instructions and some are implicit. For example, a CALL instruction that is used to call a function also pushes the upper context into stack. On the contrary, a RET instrumentation that performs the return jump also restores the upper context from the stack.

3.2 Infineon AURIX

The AURIX family is especially designed for automotive applications, e.g. Advanced Driver Assistance Systems (ADAS), powertrain control units and chassis control units for braking, steering and suspension. There are many device classes namely AURIX TC29x, TC27x, TC26x and TC23x. The Infineon AURIX TC29x device is as an example shown in Figure 3.2. It combines three powerful technologies within one silicon die including

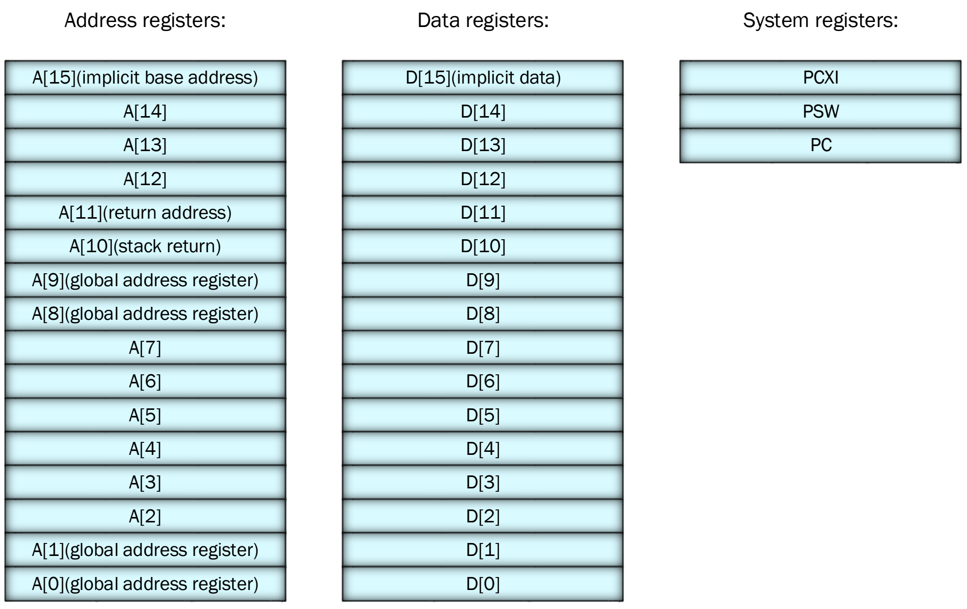


Figure 3.1: TriCore registers[105]

RISC processor architecture (TriCore), DSP and on-chip memory/peripherals [106]. The TC29x device consists of three TriCore processors running at frequencies up to 300 MHz. A TriCore processor has L1 data cache and program cache implemented. Both program cache and data cache are two-way set associative and least recently used (LRU) based.

AURIX TC29x is a NUMA system. Each core has both local data memory and local program memory attached. Local data memory in the core actually consists of several memory modules namely Data Scratch Pad RAM (DSPR), Data Cache (DCACHE) and cache tag memory. Similar to local data memory, local program memory has several blocks including Program Scratch Pad RAM (PSPR), PSPR and cache tag memory. Cache tag memory is meant to be used as general memory blocks but for testing purpose, which is not covered in this research. All these memory blocks are Error Correcting Code (ECC) protected. Global memories such as Local Memory Unit (LMU) and Program Memory Unit (PMU) are used to store data and program. LMU's primary purpose is to provide local memory for general purpose usage and it also provides access to separate block of Emulation and debug Memory (EMEM). Data stored in LMU is also protected by ECC. A special feature named Online Data Acquisition (OLDA) is a range of addresses which can be written without causing errors but no memory is really addressed, and it is also supported by LMU. The PMU controls the flash memory and the boot ROM. Flash memory is composed of both Data Flash (Dflash) and Program Flash (Pflash), which are also ECC protected.

There are two types of interconnects including Shared Resource Interconnect (SRI)

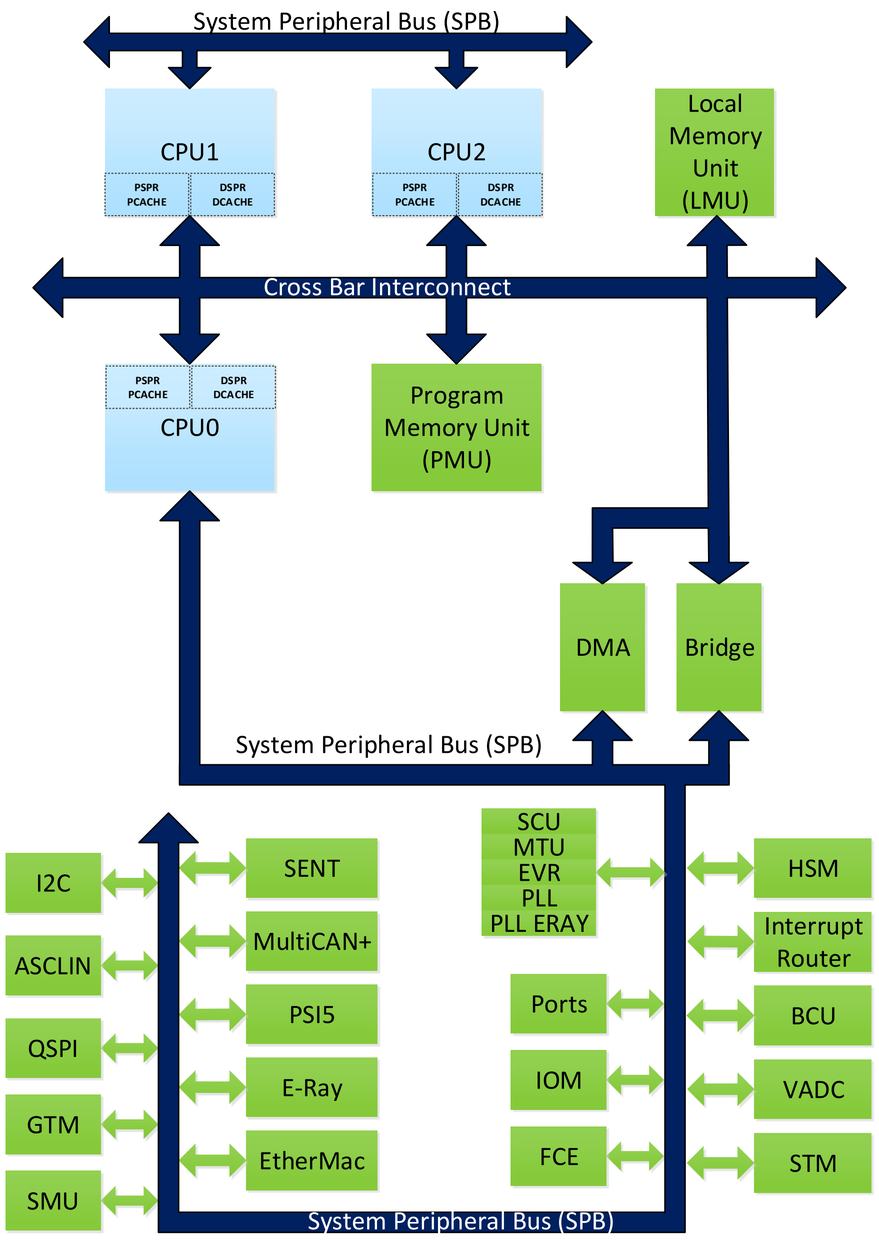


Figure 3.2: Infineon AURIX TC29x system diagram (some models are omitted)

and System Peripheral Bus (SPB). The SRI allows parallel transactions among different hardware modules. The SPB mainly connects CPUs and peripherals.

Another safety feature that differentiates AURIX from general-purpose consumer microcontrollers is lockstep. The lockstep logic provides one of the cores with an identical checker core which runs the same operations as the main core. This is meant to guarantee the detection of transient failures and permanent faults. The comparison between the checker core and the main core is delayed with two clock cycles to create a temporal barrier. The space separation is also guaranteed by placing the checker core not close to the main core.

In order to fulfill the increasing demand of various applications in the automotive field [107], many peripherals have been implemented on AURIX. These peripherals include Analog-to-Digital Converter (ADC), Queued SPI Controller (QSPI), Ethernet Media Access Controller (MAC) etc. They are not the focus of this research and will not be described in detail.

3.2.1 Atomic instructions

Atomic operations are beneficial to the performance of multithreaded systems as they are often applied in the synchronization methods. The TriCore architecture supports atomic instructions which read and/or write memory in atomic fashion:

- LDMST (Load, Modify, Store)
- SWAP.W (Swap register with memory)
- ST.T (Store bit)
- CMPSWAP.W
- SWAPMSK.W

3.2.2 Interrupt system

Multiple modules such as peripherals or external interrupts or software, acting as interrupt sources, can generate interrupt requests to interrupt service providers such as CPUs or DMA as shown in Figure 3.3. Each source is connected to a Service Request Node (SRN) and assigned a unique interrupt priority number that is used to prioritize between different interrupt requests coming at the same time. The arbitration is conducted by the Interrupt Control Unit (ICU), which is implemented for each service provider. Each SRN contains a Service Request Control (SRC) register to configure the service request regarding e.g. priority, mapping to the corresponding service providers. When an interrupt request is triggered and the information stored in the SRC is forwarded to all ICUs. The configured ICU runs the interrupt arbitration and the winning request is reported to the service provider by the ICU. The processing of service request starts after the acknowledgment by the service provider. If the requested service provider is a CPU, then interrupt service routine is entered and the interrupt system is globally disabled.

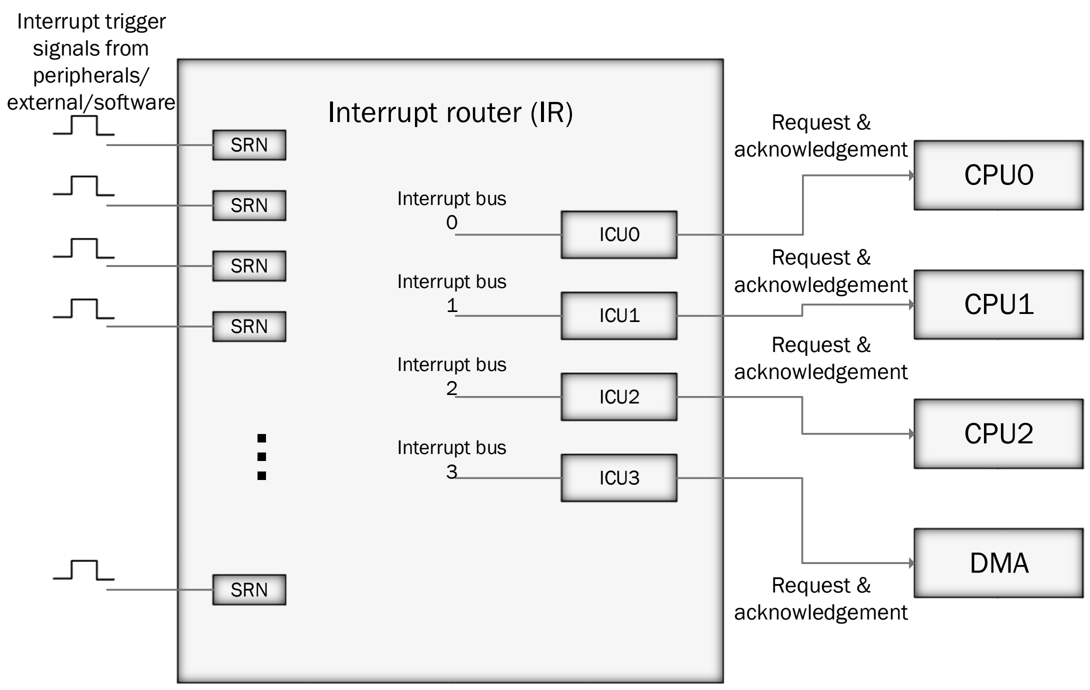

Interrupt trigger signals from peripherals/ external/software
Interrupt router (IR)
SRN
SRN
SRN
SRN
SRN
Interrupt bus 0
Interrupt bus 1
Interrupt bus 2
Interrupt bus 3
ICU0
ICU1
ICU2
ICU3
Request & acknowledgement
Request & acknowledgement
Request & acknowledgement
Request & acknowledgement
CPU0
CPU1
CPU2
DMA

Figure 3.3: The simplified diagram of the TC29x Interrupt System [106]

If the requested service provider is the DMA, then the configured DMA channel will be activated.

3.2.3 OCDS

AURIX devices have many resources for debugging and performance optimization which provide high visibility and controllability of software, hardware and system [108]. The On-Chip Debug Support (OCDS) infrastructure is not a single block but a network closely coupled in other system modules, as depicted in Figure 3.4. It involves features such as debugging and calibration. In Figure 3.4, the closely coupled parts in other system modules are also named OCDS. Many peripherals can be monitored by OCDS Trigger Switch (OTGS) such as DMA, Generic Timer Module (GTM), CAN, FlexRay and Ethernet. The collected information by OTGS can then forwarded to MCDS and stored in the trace memory, which only exists in the emulation device. The emulation device provides additional trace/calibration memory and trace logic than a normal production device.

The connection between an AURIX device and a debugger can be either via Device Access Port (DAP) or via Joint Test Action Group (JTAG), which are connected to the IOClient. Infineon's DAP that allows robust high speed connections over a long cable is a two-wire tool access port for microcontrollers [109]. JTAG supports up to 40 MHz serial clock while 2-pin and 3-pin DAP works up to 160 MHz serial clock. Even though

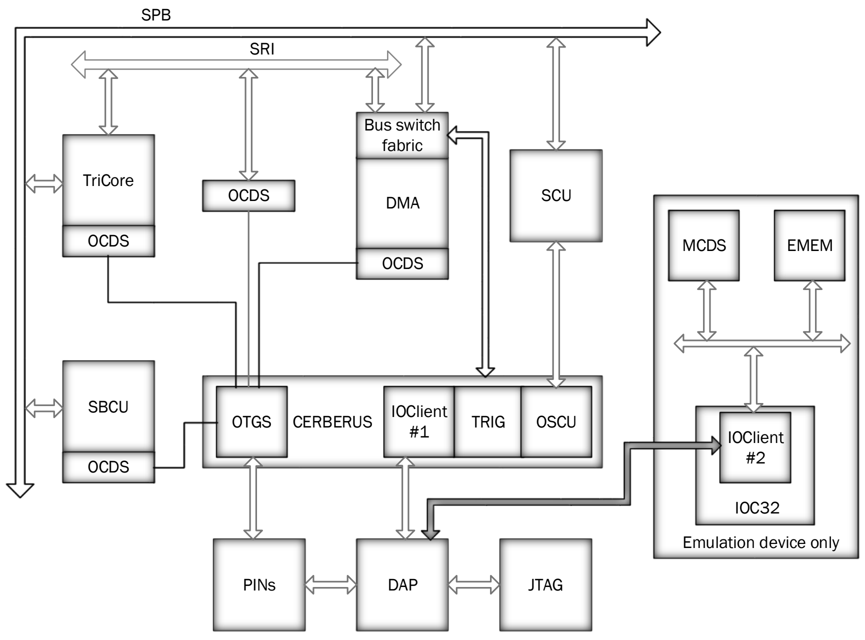


Figure 3.4: OCDS Components of AURIX TC29x [108]

the theoretical bandwidth can be 320 Mb/s, the on-chip trace generation bandwidth can still easily exceed the limit. The solution is controlling the data generation bandwidth in a continuous tracing. Another alternative is storing the trace data in the trace memory and then transferring the data out of chip.

3.3 Infineon's MCDS

MCDS is used for tracing and is the most important tool used in this dissertation. EMEM acts as tracing memory or calibration memory or normal memory. In ADAS applications EMEM is however frequently used as normal memory and is not available for on-chip trace storage.

For each device class, there are two types of products including Production Device (PD) and Emulation Device (ED) [110]. The TC2xxED is the ED of the corresponding TC2xxPD. It consists of the unchanged product chip part (SoC) and the Emulation Extension Chip (EEC) part, as shown in Figure 3.5. Both parts are on the same chip. PD and ED have similar packages to reduce the overhead of Printed Circuit Board (PCB) design.

The MCDS is designed for debugging and tracing. It is part of EEC. The hardware-level operations are observed by MCDS via observation blocks. There are two types of observation blocks: Processor Observation Block (POB) for core monitoring and Bus Observation Block (BOB) for bus monitoring. In AURIX, each core to be traced can be paired with a dedicated POB. Two cores can be traced in parallel because there are

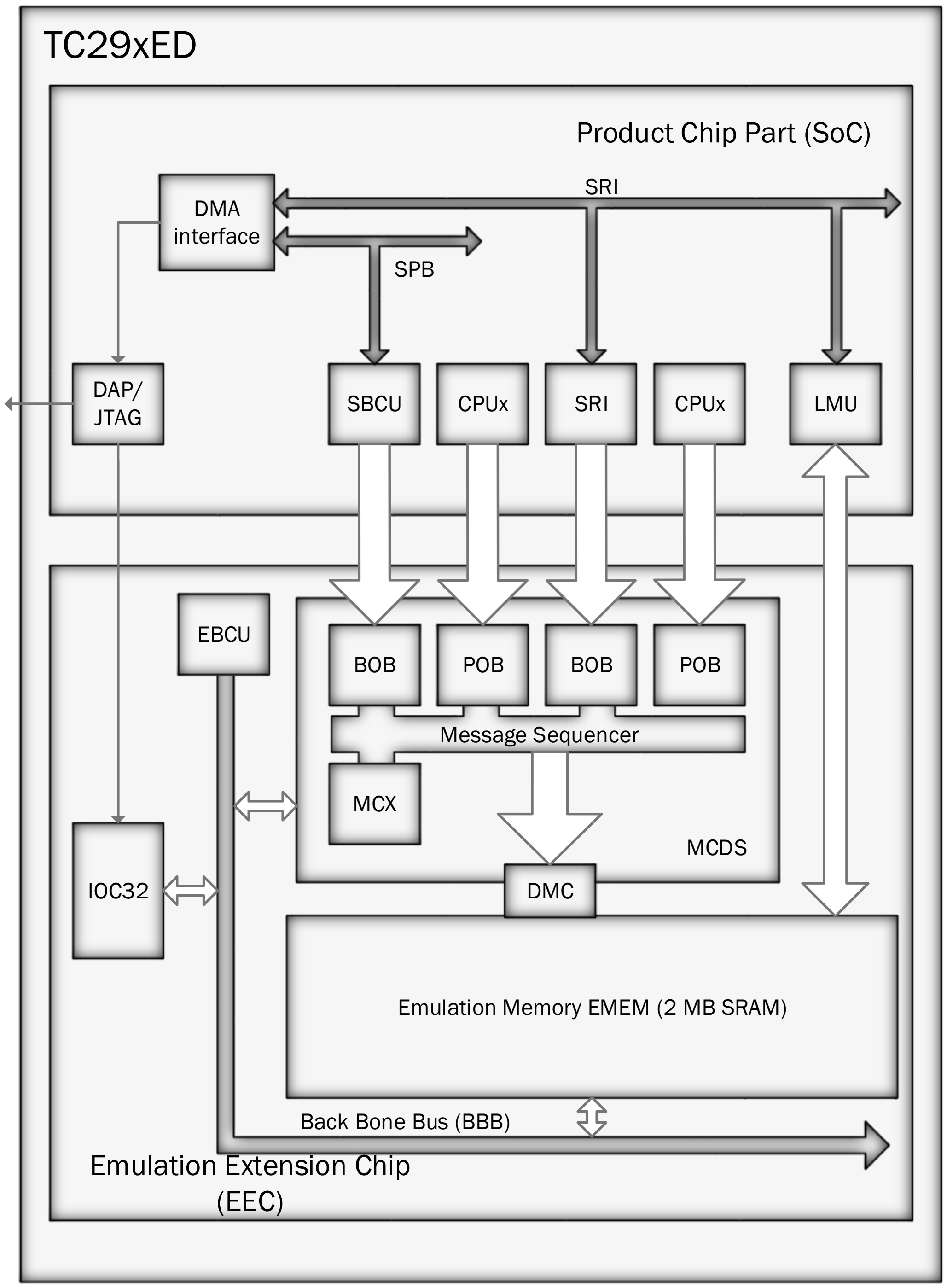


Figure 3.5: Infineon AURIX TC29xED diagram

only two POBs. As discussed above, two different buses SRI and SPB are implemented in AURIX. In order to trace the both buses, two BOBs are implemented in MCDS.

Multi Core Cross-connect (MCX) is meant to guarantee a consistent view of the system components running independently and concurrently. It creates time stamps on the buffered trace messages. On the other hand, it also handles the trace qualification across boundaries. For example, if a core executes a specific routine, bus tracing is then enabled. Debug Memory Controller (DMC) is a hardware module that is responsible for packing tracing messages from different sources into the trace buffer (EMEM). It keeps the temporal order of messages.

3.4 MCDS tracing

various kinds of information can be collected by MCDS. A POB is able to trace core related information including program flow, data accesses and core states. For program flow tracing, three alternatives with different granularity are provided, which are instruction tracing, flow tracing and compact function tracing. The alternative is selected as a compromise of tracing duration and granularity. The instruction tracing records the ending time of each instruction execution as shown in TABLE 3.1. It provides very detailed information of the instruction execution but consumes the trace buffer fast. Flow tracing offers a balance between the tracing duration and the tracing granularity as shown in TABLE 3.2. It does not trace detailed instruction execution but the discontinuity of the program flow, which is resulted from function calls, branches and interrupts. Only the discontinuity recording is necessary because by default the PC is continuously incremented. In this way, the ending time of each instruction is not available but it lowers the data generation bandwidth. The least bandwidth-demanding alternative is compact function tracing, which only generates messages when function calls and function returns are observed TABLE 3.3. This alternative consumes little bandwidth but it also omits detailed execution information. For instance, the detailed branch execution is not contained in compact function tracing [111].

The three tables 3.1 3.2 3.3 show the three alternative examples. They trace the same program execution scope, in which `Ifx_OSTask_Application` is called in `main` and then `EE_oo_StartOS` is executed. The examples above are decoded by MTV, which is an MCDS configuration and trace decoding tool. The .elf file is also given to provide the detailed binary and symbolic information. **TimeR** shows the time stamp information. The time stamp is added in MCDS after a message is generated. For instruction tracing, a message is generated when an instruction is executed. For flow tracing, the generation of a message is triggered when a discontinuity happens. The exact time information of each instruction is missing in flow tracing. Accordingly, a function call or return creates a compact function tracing message. **Opoint** is short for observation point, corresponding to the observation blocks, where the message is captured. **Origin** indicates the origin of the operation. **Operation** is meant to distinguish different types of operations. It is noted that `EE_oo_StartOS` is called by a branch instruction from `Ifx_OSTask_Application` so the operation of the last message in `Ifx_OSTask_Application` in TABLE 3.1 is IP

Table 3.1: An instruction tracing example

Index	TimeR	Opoint	Origin	Operation	Address	Symbol
1	1	CPU0	CPU0	IP	A001C2CC	core0_main
2	1	CPU0	CPU0	IP	A001C2D0	core0_main
3	3	CPU0	CPU0	IP	A001C2D4	core0_main
4	4	CPU0	CPU0	IP	A001C2D8	core0_main
5	6	CPU0	CPU0	IP	A001C2DC	core0_main
6	8	CPU0	CPU0	IP	A001C2DE	core0_main
7	9	CPU0	CPU0	IP	A001C2E2	core0_main
8	11	CPU0	CPU0	IP	A001C2E6	core0_main
9	12	CPU0	CPU0	IP	A001C2EA	core0_main
10	14	CPU0	CPU0	IP	A001C2EE	core0_main
11	16	CPU0	CPU0	IP CALL	A001C2F2	core0_main
12	25	CPU0	CPU0	IP	A00022C4	Ifx_OSTask_ApplicationInit
13	26	CPU0	CPU0	IP	A00022C6	Ifx_OSTask_ApplicationInit
14	40	CPU0	CPU0	IP	A001F2E0	EE_oo_StartOS
15	41	CPU0	CPU0	IP	A001F2E4	EE_oo_StartOS
16	41	CPU0	CPU0	IP	A001F2E8	EE_oo_StartOS
17	41	CPU0	CPU0	IP	A001F2EC	EE_oo_StartOS

Table 3.2: A flow tracing example

Index	TimeR	Opoint	Origin	Operation	Address	Symbol
1	17	CPU0	CPU0	IP	A001C2CC	core0_main
2	17	CPU0	CPU0	IP	A001C2D0	core0_main
3	17	CPU0	CPU0	IP	A001C2D4	core0_main
4	17	CPU0	CPU0	IP	A001C2D8	core0_main
5	17	CPU0	CPU0	IP	A001C2DC	core0_main
6	17	CPU0	CPU0	IP	A001C2DE	core0_main
7	17	CPU0	CPU0	IP	A001C2E2	core0_main
8	17	CPU0	CPU0	IP	A001C2E6	core0_main
9	17	CPU0	CPU0	IP	A001C2EA	core0_main
10	17	CPU0	CPU0	IP	A001C2EE	core0_main
11	17	CPU0	CPU0	IP CALL	A001C2F2	core0_main
12	26	CPU0	CPU0	IP	A00022C4	Ifx_OSTask_ApplicationInit
13	26	CPU0	CPU0	IP	A00022C6	Ifx_OSTask_ApplicationInit
14	53	CPU0	CPU0	IP	A001F2E0	EE_oo_StartOS
15	53	CPU0	CPU0	IP	A001F2E4	EE_oo_StartOS
16	53	CPU0	CPU0	IP	A001F2E8	EE_oo_StartOS
17	53	CPU0	CPU0	IP	A001F2EC	EE_oo_StartOS
18	53	CPU0	CPU0	IP	A001F2F0	EE_oo_StartOS
19	53	CPU0	CPU0	IP	A001F2F4	EE_oo_StartOS
20	53	CPU0	CPU0	IP	A001F2F6	EE_oo_StartOS

Table 3.3: A compact function tracing example

Index	TimeR	Opoint	Origin	Operation	Address	Symbol
1	17	CPU0	CPU0	IP CALL	A001C2F2	core0_main
2	17	CPU0	CPU0	IP CALL	A00022C4	Ifx_OSTask_Application
3	77	CPU0	CPU0	IP CALL	A001F36E	EE_oo_StartOS
4	77	CPU0	CPU0	IP CALL	A0001C50	StartupHook

instead of IP CALL. Such kind of compiler optimizations makes the compact function trace interpretation sometimes challenging for the user. The **Address** column shows the IP address of executed instructions, based on which the detailed instructions and function symbols can be parsed with the binary and symbolic information contained in an .elf file.

BOBs are designed to capture bus transfer operations and bus states. Information related to a bus transfer including address, data, time stamps is collected. A bus tracing example of SPB is shown in TABLE 3.4. Similar to the instruction tracing, the time stamp is added to each message, which is corresponding to a bus transfer. The operation contains the bus transfer type information about the accessing type and accessing width. The **Symbol** column indicates the target hardware module, which is post-processed based on the device information. The first message in TABLE 3.4 describes that CPU0 reads 0xF0036100, corresponding to a register in the System Control Unit (SCU) module.

Table 3.4: A bus transfer tracing example

Index	TimeR	Opoint	Origin	Data	Operation	Address	Symbol
1	19889	SPB	CPU0.DMI	FFFC000E	R32 SV	F0036100	.SCU
2	19911	SPB	CPU0.DMI	FFFC000E	R32 SV	F0036100	.SCU
3	19916	SPB	CPU0.DMI	FFFC000E	R32 SV	F0036100	.SCU
4	19934	SPB	CPU0.DMI	FFFC00F1	W32 SV	F0036100	.SCU
5	19938	SPB	CPU0.DMI	FFFC00F2	W32 SV	F0036100	.SCU

3.5 AURIX tools

The implementation of the proposed methodology is developed on the basis of several existing tools. DAS and MTV are the most important ones. DAS deals with the connection to devices while MTV is focused on MCDS tracing and decoding.

3.5.1 DAS

The DAS architecture was designed for multi-device multi-core systems with very demanding emulation requirements [112]. It provides one single interface for different types of tools. The same tooling interface supports various device representations from ESL model to end product to reduce cost and risk.

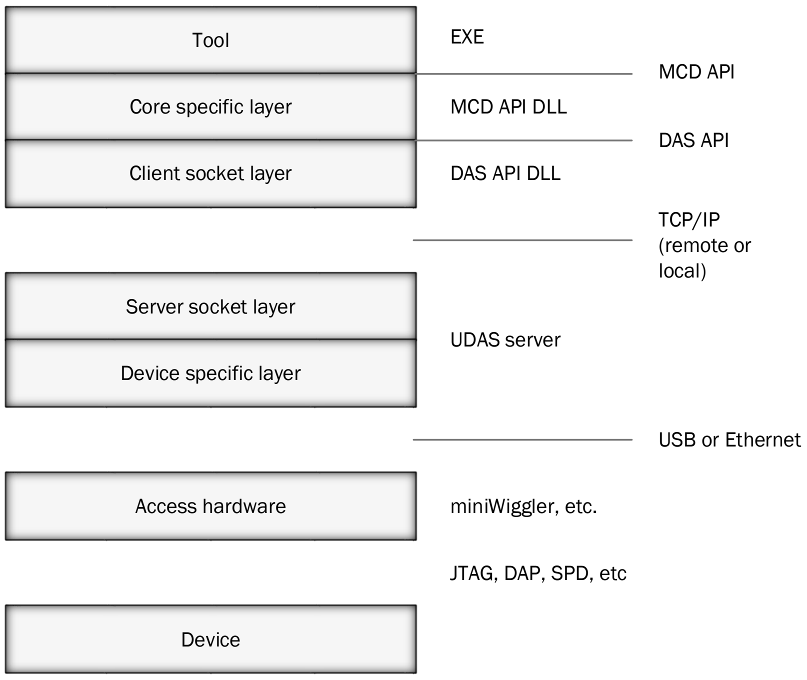


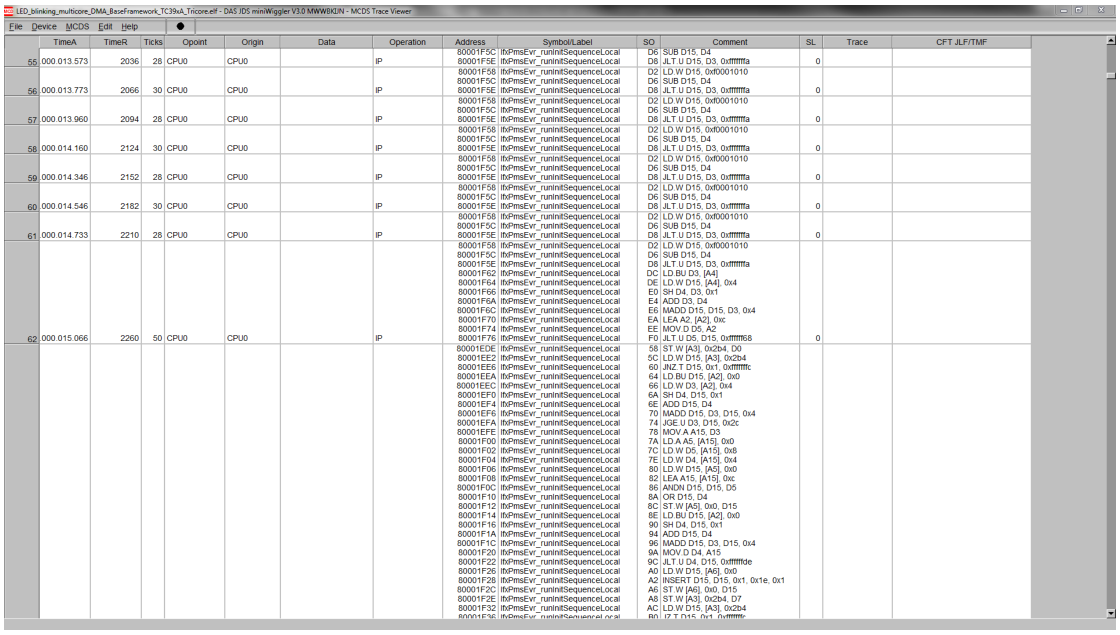
Figure 3.6: The block diagram of Infineon's DAS [112]

The detailed structure of DAS is shown in the block diagram Figure 3.6. The tool interface is on software level (DAS Application Programming Interface (API)), which is implemented in a .dll file. This layer provides the abstraction of the device connection. In this way, the connection will be transparent for the tool. On top of this layer, MCD API is created to better fulfill the requirement mentioned in [112]. The connection between computer and access hardware is via either USB or Ethernet. The access hardware here means miniWiggler, that is a converter between USB and DAP/SPD/SWD/JTAG. For development boards provided by Infineon, an on-board miniWiggler is already integrated on the PCB so that the board can be connected directly from the computer via USB.

3.5.2 MTV

MTV is developed on top of DAS. The goal of MTV is to utilize the power of MCDS in an easy and user-friendly way. It has main functions including MCDS configuration, MCDS tracing and trace data decoding. MTV is programmed in C++ and FLTK that is a cross-platform Graphical User Interface (GUI) library.

As mentioned in the previous sections, MCDS is very flexible. Various triggers and qualifications are possible. It is capable of tracing many hardware modules flexibly with different levels of details. To perform various triggers and qualifications, configurations directly on MCDS registers are needed, which are complex for software debugging. MTV provides a user-friendly GUI and facilitates the configuration of MCDS. Instead



LED_blinking_multicore_DMA_BaseFramework_TC39xA_Tricore.elf - DAS JDS miniWiggler V3.0 MWWBKIJN - MCDS Trace Viewer

	TimeA	TimeR	Ticks	Opoint	Origin	Data	Operation	Address	Symbol/Label	SO	Comment	SL	Trace	CFT JLF/TMF
								80001F5C	IfxPmsEvr_runInitSequenceLocal	D6	SUB D15, D4			
55	000.013.573	2036	28	CPU0	CPU0		IP	80001F5E	IfxPmsEvr_runInitSequenceLocal	D8	JLT.U D15, D3, 0xfffffffa	0		
								80001F58	IfxPmsEvr_runInitSequenceLocal	D2	LD.W D15, 0xf0001010			
								80001F5C	IfxPmsEvr_runInitSequenceLocal	D6	SUB D15, D4			
56	000.013.773	2066	30	CPU0	CPU0		IP	80001F5E	IfxPmsEvr_runInitSequenceLocal	D8	JLT.U D15, D3, 0xfffffffa	0		
								80001F58	IfxPmsEvr_runInitSequenceLocal	D2	LD.W D15, 0xf0001010			
								80001F5C	IfxPmsEvr_runInitSequenceLocal	D6	SUB D15, D4			
57	000.013.960	2094	28	CPU0	CPU0		IP	80001F5E	IfxPmsEvr_runInitSequenceLocal	D8	JLT.U D15, D3, 0xfffffffa	0		
								80001F58	IfxPmsEvr_runInitSequenceLocal	D2	LD.W D15, 0xf0001010			
								80001F5C	IfxPmsEvr_runInitSequenceLocal	D6	SUB D15, D4			
58	000.014.160	2124	30	CPU0	CPU0		IP	80001F5E	IfxPmsEvr_runInitSequenceLocal	D8	JLT.U D15, D3, 0xfffffffa	0		
								80001F58	IfxPmsEvr_runInitSequenceLocal	D2	LD.W D15, 0xf0001010			
								80001F5C	IfxPmsEvr_runInitSequenceLocal	D6	SUB D15, D4			
59	000.014.346	2152	28	CPU0	CPU0		IP	80001F5E	IfxPmsEvr_runInitSequenceLocal	D8	JLT.U D15, D3, 0xfffffffa	0		
								80001F58	IfxPmsEvr_runInitSequenceLocal	D2	LD.W D15, 0xf0001010			
								80001F5C	IfxPmsEvr_runInitSequenceLocal	D6	SUB D15, D4			
60	000.014.546	2182	30	CPU0	CPU0		IP	80001F5E	IfxPmsEvr_runInitSequenceLocal	D8	JLT.U D15, D3, 0xfffffffa	0		
								80001F58	IfxPmsEvr_runInitSequenceLocal	D2	LD.W D15, 0xf0001010			
								80001F5C	IfxPmsEvr_runInitSequenceLocal	D6	SUB D15, D4			
61	000.014.733	2210	28	CPU0	CPU0		IP	80001F5E	IfxPmsEvr_runInitSequenceLocal	D8	JLT.U D15, D3, 0xfffffffa	0		
								80001F58	IfxPmsEvr_runInitSequenceLocal	D2	LD.W D15, 0xf0001010			
								80001F5C	IfxPmsEvr_runInitSequenceLocal	D6	SUB D15, D4			
								80001F5E	IfxPmsEvr_runInitSequenceLocal	D8	JLT.U D15, D3, 0xfffffffa			
								80001F62	IfxPmsEvr_runInitSequenceLocal	DC	LD.BU D3, [A4]			
								80001F64	IfxPmsEvr_runInitSequenceLocal	DE	LD.W D15, [A4], 0x4			
								80001F66	IfxPmsEvr_runInitSequenceLocal	E0	SH D4, D3, 0x1			
								80001F6A	IfxPmsEvr_runInitSequenceLocal	E4	ADD D3, D4			
								80001F6C	IfxPmsEvr_runInitSequenceLocal	E6	MADD D15, D15, D3, 0x4			
								80001F70	IfxPmsEvr_runInitSequenceLocal	EA	LEA A2, [A2], 0xc			
								80001F74	IfxPmsEvr_runInitSequenceLocal	EE	MOV.D D5, A2			
62	000.015.066	2260	50	CPU0	CPU0		IP	80001F76	IfxPmsEvr_runInitSequenceLocal	F0	JLT.U D5, D15, 0xffffff68	0		
								80001EDE	IfxPmsEvr_runInitSequenceLocal	58	ST.W [A3], 0x2b4, D0			
								80001EE2	IfxPmsEvr_runInitSequenceLocal	5C	LD.W D15, [A3], 0x2b4			
								80001EE6	IfxPmsEvr_runInitSequenceLocal	60	JNZ.T D15, 0x1, 0xfffffffc			
								80001EEA	IfxPmsEvr_runInitSequenceLocal	64	LD.BU D15, [A2], 0x0			
								80001EEC	IfxPmsEvr_runInitSequenceLocal	66	LD.W D3, [A2], 0x4			
								80001EF0	IfxPmsEvr_runInitSequenceLocal	6A	SH D4, D15, 0x1			
								80001EF4	IfxPmsEvr_runInitSequenceLocal	6E	ADD D15, D4			
								80001EF6	IfxPmsEvr_runInitSequenceLocal	70	MADD D15, D3, D15, 0x4			
								80001EFA	IfxPmsEvr_runInitSequenceLocal	74	JGE.U D3, D15, 0x2c			
								80001EFE	IfxPmsEvr_runInitSequenceLocal	78	MOV.A A15, D3			
								80001F00	IfxPmsEvr_runInitSequenceLocal	7A	LD.A A5, [A15], 0x0			
								80001F02	IfxPmsEvr_runInitSequenceLocal	7C	LD.W D5, [A15], 0x8			
								80001F04	IfxPmsEvr_runInitSequenceLocal	7E	LD.W D4, [A15], 0x4			
								80001F06	IfxPmsEvr_runInitSequenceLocal	80	LD.W D15, [A5], 0x0			
								80001F08	IfxPmsEvr_runInitSequenceLocal	82	LEA A15, [A15], 0xc			
								80001F0C	IfxPmsEvr_runInitSequenceLocal	86	ANDN D15, D15, D5			
								80001F10	IfxPmsEvr_runInitSequenceLocal	8A	OR D15, D4			
								80001F12	IfxPmsEvr_runInitSequenceLocal	8C	ST.W [A5], 0x0, D15			
								80001F14	IfxPmsEvr_runInitSequenceLocal	8E	LD.BU D15, [A2], 0x0			
								80001F16	IfxPmsEvr_runInitSequenceLocal	90	SH D4, D15, 0x1			
								80001F1A	IfxPmsEvr_runInitSequenceLocal	94	ADD D15, D4			
								80001F1C	IfxPmsEvr_runInitSequenceLocal	96	MADD D15, D3, D15, 0x4			
								80001F20	IfxPmsEvr_runInitSequenceLocal	9A	MOV.D D4, A15			
								80001F22	IfxPmsEvr_runInitSequenceLocal	9C	JLT.U D4, D15, 0xffffffde			
								80001F26	IfxPmsEvr_runInitSequenceLocal	A0	LD.W D15, [A6], 0x0			
								80001F28	IfxPmsEvr_runInitSequenceLocal	A2	INSERT D15, D15, 0x1, 0x1e, 0x1			
								80001F2C	IfxPmsEvr_runInitSequenceLocal	A6	ST.W [A6], 0x0, D15			
								80001F2E	IfxPmsEvr_runInitSequenceLocal	A8	ST.W [A3], 0x2b4, D7			
								80001F32	IfxPmsEvr_runInitSequenceLocal	AC	LD.W D15, [A3], 0x2b4			
								80001F36	IfxPmsEvr_runInitSequenceLocal	B0	JZ.T D15, 0x1, 0xfffffffc			

Figure 3.7: The screenshot of MTV

of handling register values, users can select a tracing mode from a software developers' perspective. Then, the corresponding MCDS configurations are generated and automatically configured.

The collected trace data is stored in the EMEM. It must be decoded in order to be interpreted. The decoded messages are displayed in a table as shown in the screenshot in Figure 3.7.

4 Embedded Health – How to diagnose a complex system?

Testing and debugging of software consume a significant amount of time and money. Debugging, testing and verification activities can take up from 50% to 75% of the development cost [113]. The conventional debugging tools fulfill the basic debugging requirements. However, the rapid advances in SoCs have led to powerful multi-core/manycore systems, in which new issues are introduced. These issues include shared resource contention, atomic violations, deadlocks, priority inversion and inefficient type of locks. Some of these issues are sporadic, e.g. deadlocks and priority inversions. Some of them have no obvious symptoms e.g. inefficient lock usage and memory contention compared to the conventional functional bugs. The conventional debugging solutions like breakpoint/step are not efficient enough for these cases. The gap between conventional debugging tools and the debugging requirements for SoC increases. As a result, the conventional debugging tools can hardly deal with the new-emerging issues [6]. Quality of the software has not kept the pace [114] of SoC advances. Software bugs are still frequent and even harder to detect. New debugging challenges with complex SoCs and multi-core architectures have to be managed by a new powerful tool.

4.1 Learn from medicine – embedded health

Modern embedded systems are complex from both software and hardware perspectives. The challenge how to figure out what is going wrong in a complex system is not new. Doctors in the medicine field have been facing the same challenge for thousands of years, in terms of the human body and the diseases it faces. Doctors know that the diagnosis based only on the superficial symptoms is not a reliable best-practice. A disease like leukemia for example, can exhibit symptoms that are similar to the flu and other common diseases. Only more detailed tests, such as blood tests, will help reveal if it is actually leukemia. Thus in medicine standardized diagnosis procedures are used to systematically check many points rather than relying on assumptions based on the initial superficial symptoms. As in our example, a blood test is a typical case of such a procedure that is very useful for diagnosing different diseases and can be used for a great number of patients. Blood tests are standardized and can be conducted for different individuals. In blood tests, various indicators are designed to show the health state of a body.

Embedded health, based on the concept of a blood test, incorporates an innovative system diagnosis and debug methodology to automatically detect issues in embedded systems and act as a generally applicable diagnosis tool for a variety of system issues. By making use of hardware tracing supported by the target system, the proposed methodology detects "diseases" by using predefined indicators. The predefined indicators are low-level hardware-related behavioral patterns acting as the metric system in our "blood test". For complex embedded systems, software debugging and system diagnosis can be solved in a similar way [115]. Hardware tracing is supported by many COTS devices e.g. Infineon's MCDS, ARM's Coresight and Freescale's Nexus 5001. It can help developers figure out what is going on in systems. One advantage of hardware tracing is that it is non-intrusive, thus there is no impact on system timing and no software instrumentation is required. This is especially important for real-time systems as a bug may not be reproducible with a different timing after software instrumentation. Various indicators are designed based on the trace data to show different system issues corresponding to "diseases" in medicine. Usually the concentration of a specific substance in blood tests is an important indicator and however it is not sufficient for embedded systems.

4.1.1 An automated system diagnosis and debugging methodology

In order to solve the new-coming challenges, in this chapter a trace-based automated system diagnosis and debugging methodology is proposed to cover issues in embedded systems that cannot be efficiently solved by conventional tools. It is based on the embedded health and makes use of hardware tracing like blood tests. The hardware tracing provides hardware-level trace data including program flow, bus transfers, memory accesses etc. However, hardware tracing is limited by the bandwidth and the trace buffer size [116]. It is not feasible to trace all program flows and memory addresses continuously. Tracing qualification is usually used to reduce the amount of trace data, meaning tracing only the necessary information. The decision what should be traced depends on the analysis, which is conducted by the tool. Various indicators are designed to detect different system issues e.g. data memory contention, program flash contention, hardware configuration issues. Applications with potential issues are examined by the diagnosis tool and a diagnosis report is generated automatically without the involvement of users. The detected issues are described in detail on the report and appropriate solutions can be then applied.

Indicators are independent of applications and software. Accordingly, the proposed automated system diagnosis and debugging methodology is a general solution regardless of applications.

The design of indicators is critical for the detection of issues. An indicator can be simply a temporal rule or a complex model. For instance, a write operation to a protected register is a violation of a rule (a protected area cannot be written). One write operation to a disabled register before enabling it is also a violation of a rule. This rule also contains temporal information (before enabling). The violation of the rule can be treated as an indicator. A more complex indicator example is the memory access contention indicator, which is a model depending on the previous memory accesses, access origin, access destination and even interval between two memory accesses. In summary, an indicator is a criterion to judge whether an issue occurs and how severe the issue is, given an input. The input here in the proposed methodology is trace data as shown in Figure 4.1. The mapping between issues and indicators is not necessarily one-to-one mapped. An issue

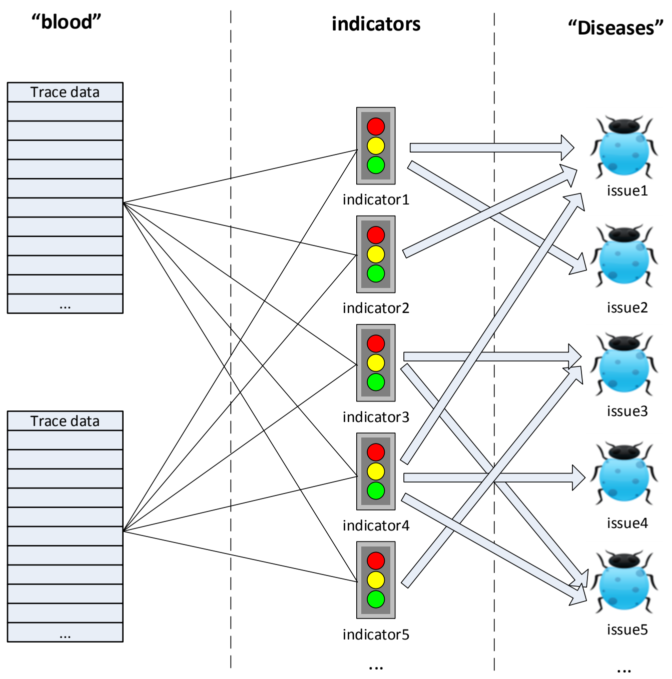


Figure 4.1: The basic work flow of the indicators in embedded health

may only be determined if several indicators are positive. For example, the stalling of a CPU pipeline can be an indicator of program flash contention. It can also be an indicator of data memory contention. If another indicator that shows the existence of a conflicting data memory access, the data memory contention issue is confirmed. Therefore, more than one indicator might be needed to make sure which issue it is.

The indicator plays an important role in the proposed methodology. The designed indicators influence directly the detection of issues. Figure 4.2 illustrates the design flow of indicators. Before the design of indicator, the first step is to decide the target issue. When the target issue is selected, analysis can be performed to check the related effects in the hardware level. For example, CPU pipeline stall can be related to memory contention. Then, a coarse indicator can be extracted and corresponding testing is applied to make sure that the issue can really be detected with the designed indicator. If the indicator is not effective, another indicator may be needed or the indicator should be refined. For instance, some hardware configuration issues can be detected by checking the configuration process of the registers, so one indicator is usually sufficient for the detection. While, for flash contention, relying on one indicator may result in many false alarms and therefore several indicators are needed to confirm the contention. The details of these two examples will be introduced later. If the designed indicators are good enough for the detection, then mapping of indicators and the issue will be stored in the indicator pool.

The proposed methodology is aimed to solve the new-coming issues introduced by com-

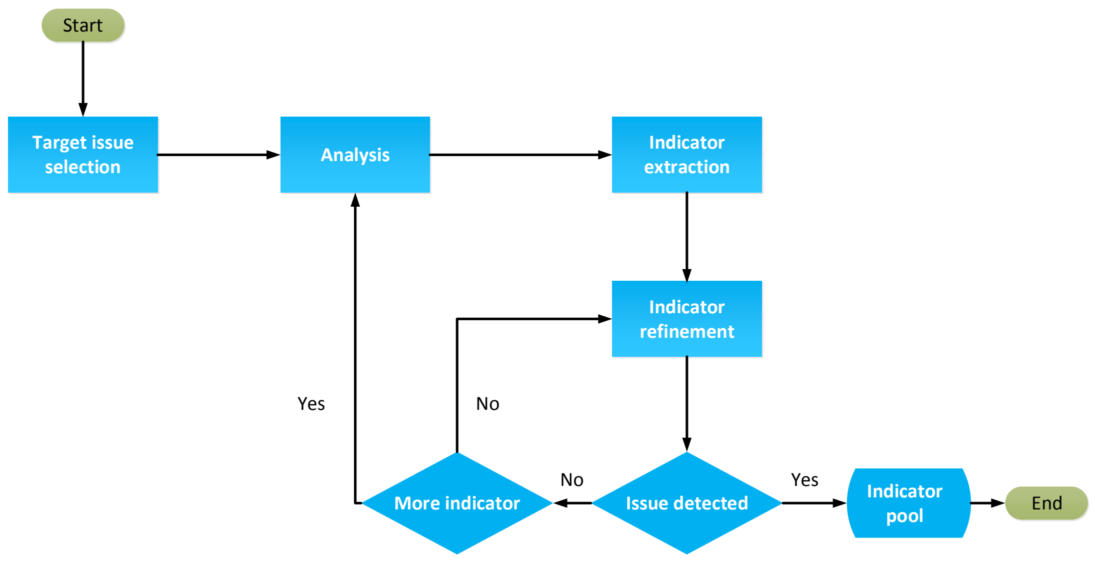


Figure 4.2: The basic design flow of indicators

plex hardware and software in an automated and efficient way. It runs on a computer and connects to target devices via debug interfaces, via which trace data is transferred from chip to the computer. It automatically configures the tracing hardware that is intrinsically integrated to the embedded platform, and decides the scope of tracing. The tracing hardware is responsible for trace message generation, compression and temporal storage. The trace data provides low-level information of executed hardware operations including program flow, bus transfers, memory accesses, based on which various indicators are created to detect different target system issues e.g. data memory contention, program flash contention, hardware configuration issues. The indicators can be patterns of operations or a sequence of patterns. A specific issue can be detected relying on a pre-defined indicator. For different diagnoses, the tracing hardware is configured accordingly to obtain the required data. The only input file from users is the .elf file that provides symbolic and section information. If a user has no idea about where to look at, the proposed methodology will start the analysis from the beginning, e.g. after reset and run a general physical examination.

During the tracing phase, the tracing hardware is configured automatically by the proposed methodology as shown in Figure 4.3. Then the trace messages containing hardware-related operations are linked to binary and symbolic information available in the .elf file. In the indicator extraction phase, the trace data is compared to the standard indicators. The indicators can be stored in the tool or in a special format that is easy to be read. In this way, this methodology can be applied for different hardware platforms given the necessary input indicators. Finally, a report is generated to pinpoint the detected issues based on the comparison. In the report, issues are highlighted and even suggestions may be raised automatically by the implemented tool.

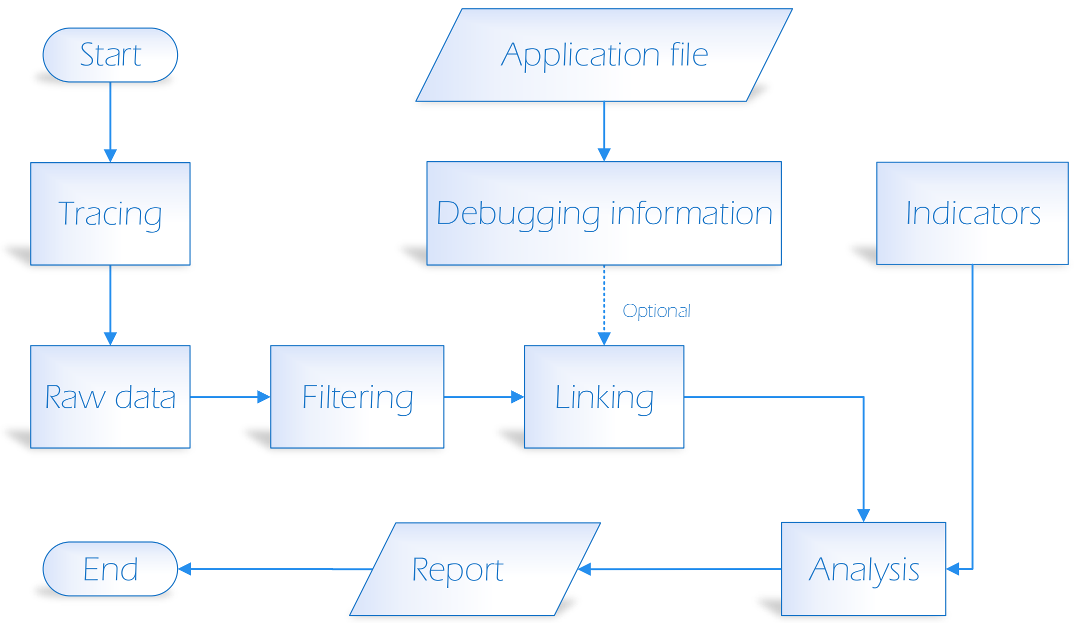


Figure 4.3: The work flow of the automated system diagnosis methodology

As shown, the proposed methodology has an automated work flow from conventional debugging process, in which software developers are highly involved. The comparison between conventional debugging process and the proposed debugging process is illustrated in Figure 4.4.

In the scenario if the clock configuration is wrong, using the conventional debugging approach, the software developer has to collect the observed data and read through the clock configuration part in the user manual. Then he has to do a considerable amount of analysis to figure out the root cause. Several hypotheses might be assumed by the developer and verification measures are applied. If the hypothesis is right, he can fix the issue. Otherwise, he has to repeat this process until the real root cause is found, though new observed data may be added after each iteration. It is obvious that the conventional debugging approach is a manual approach, which involves the software developer a lot. The duration of this debugging process depends on the experience level of the software developer.

The right part in the Figure 4.4 shows the proposed work flow using the proposed automated debugging tool. Other than the conventional method, the software developer is not required to know what exactly is wrong, even without knowing the clock configuration is wrong. For instance, he can directly run the tool to do a full check, which covers several groups of issues. Afterward, a report is available right away. The clock configuration error is highlighted in the report as the configuration violates the predefined rules. Accordingly, the configuration can be fixed immediately by the developer. As shown, this process involves much less the software developer, increasing the efficiency of the debugging process. The debugging duration is not relying on the experience level as the

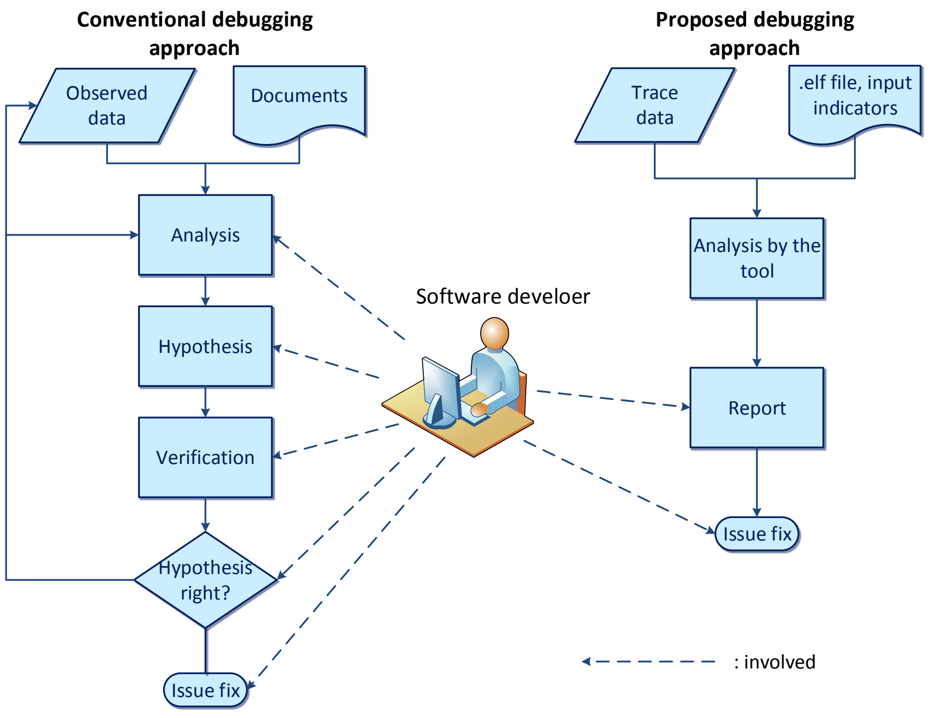


Figure 4.4: The comparison of the conventional debugging flow and the proposed work flow

knowledge can be shared even for different applications.

There are many automated debugging tools emphasized on a specific software issue. Compared to such tools, the proposed methodology covers different types of issues, which are rarely emphasized by existing solutions. In summary, the proposed methodology is designed to be an innovative general and efficient solution for software debugging.

4.1.2 Issue classification

Software bugs can be classified into several groups [117] namely arithmetic, logic, syntax, resource, multithreading, interfacing, performance, team working. Arithmetic bugs are mainly calculation related such as division by zero and arithmetic overflow as shown in TABLE 4.1. Logic bugs are logic related, e.g. off-by-one error. Syntax bugs are bugs for example wrong usage of operators and are usually caught by compilers. Resource errors are closely related to memory and data structure usage, for instance stack overflow. Logic bugs, syntax bugs and resource bugs are classic and have been studied for quite a long time.

The above classification is comprehensive and general. It covers a very broad range of software bugs. In this dissertation, only new-emerging ones that are rarely covered by the existing studies are focused. Therefore, a different classification is used to emphasize this small group.

Merely replying symptoms is not sufficient to diagnose system issues. A criterion is

Table 4.1: Software bug classification according to Wikipedia [117]

Software bug types	Examples
Arithmetic	Division by zero, arithmetic overflow
Logic	Infinite loops, counting one too many while looping
Syntax	Wrong programming language usage
Resource	Memory leakage, access violations, using uninitialized variable
Multithreading	Dead lock, race condition
Interfacing	Incorrect hardware handling, incorrect APU usage
Performance	random memory accesses
Teamworking	Documentation errors

necessary to define what is right. This criterion could be binary, either good or bad. It can also be more complex and fuzzy. For example, some issues may slightly impact the performance but they are not fatal. These issues might be tolerated by the system. Some other issues may heavily influence the performance resulting in system functional changes. They are supposed to be notified to developers. Based on this classification, two different categories are created, namely **functional issues** and **non-functional issues**. The overview of both categories is explained in the Figure 4.5. Some of the example issues are covered by the proposed methodology in this dissertation.

The methodologies to solve these two different types of issues are described in the following two sections. The implementation and the experiential results for each analysis are introduced in the next chapter 5.

4.2 Diagnosis methodologies of functional issues

In this section, a methodology dealing with hardware configuration issues is introduced. Hardware configuration issues are mostly binary issues as the valid hardware configuration is clearly explained by the rules documented in user manual. A small part of hardware configurations may be related to performance degradation but still this group of issues is assigned to functional group.

4.2.1 Hardware configuration validation

4.2.1.1 Introduction

More and more processing elements and functional units are integrated into multi-core SoC, fueled by exponentially increasing transistor numbers. The integration of more components into a single chip leads to complex configurations. Most configurations are configured by software during runtime. Many rules are defined to regulate the procedure to set them, for example, a rule defining which write value is allowed for a specific register. Usually these rules are described in user manual and the growth

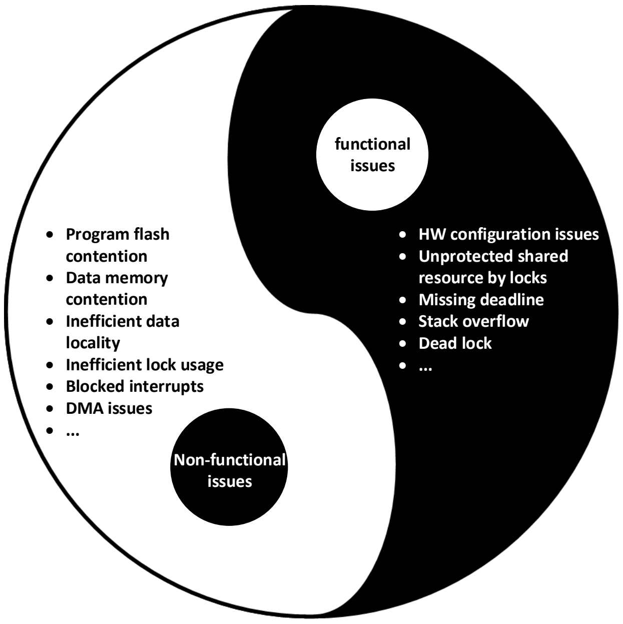


Figure 4.5: The overview of two categories of example issues: functional issues and non-functional issues

in configuration complexity can be illustrated by the length of the user manual. For example, the user manual of Infineon AURIX TC29 exceeds 6000 pages.

The violations of the rules defined in the user manual may lead to many issues. These issues include performance issues and functional issues. If the issues have obvious symptoms, they can be easily noticed. Otherwise, they may be ignored or hard to detect. In the following, two real-life examples based on Infineon AURIX are described.

In the Infineon AURIX devices, on-chip flash stores target code and constant data. Flash is slower than SRAM and needs at least certain time to access. This time is to be configured by a special register, which notifies the bus when to fetch the read data after the read data is available at the interface between the flash and the bus. The detailed requirements are shown in TABLE 4.2. The flash wait state configurations depend on the clock frequencies.

Table 4.2: Wait Cycle Calculation on AURIX TC29 [49]

Delay	Register Field	Minimum Value	Constant
program flash read access delay	`FCON.WSPFLASH`	$\lceil t_{\mathrm{PF}} \cdot f_{\mathrm{FSI2}} \rceil - 1$	t_{PF}
program flash ECC decode delay	`FCON.WSECPF`	$\lceil t_{\mathrm{PFECC}} \cdot f_{\mathrm{FSI2}} \rceil - 1$	t_{PFECC}
data flash read access delay	`FCON.WSDFLASH`	$\lceil t_{\mathrm{DF}} \cdot f_{\mathrm{FSI}} \rceil - 1$	t_{DF}
data flash ECC decode delay	`FCON.WSECDF`	$\lceil t_{\mathrm{DFECC}} \cdot f_{\mathrm{FSI}} \rceil - 1$	t_{DFECC}

The **Delay** column indicates different delays required by the flash configuration. There are two different flashes namely Pflash and Dflash. For each type of flash, read access delay and ECC decode delay have to be configured to properly guarantee that the data is already available at the interface to be fetched by the bus. **Register Field** as the name explains shows which field in a register should be set. The minimum value in this field is calculated according to the formula defined in the **Minimum Value** column. Constants such as t_{PF}, t_{PFECC}, t_{DF} and t_{DFECC} are defined in a data sheet.

If the waiting time is below the minimum requirement, the read operation may fail, leading to bus errors. Depending on the working condition e.g. temperature, the bus errors happen sporadically from a software developer's view. The bus errors then trigger exceptions, which are usually solved in the following process: The exception name indicates which exception type it is and the instruction causing the exception can also be derived. However, this instruction may differ due to the fact that this error is sporadic, which is usually confusing for software developers. Furthermore, linking the bus errors to flash configurations is also not straightforward, which may take a long time to figure out without experienced expert's help.

It looks safer to set the value larger than the necessary minimum value. In fact, the configuration with a value larger than needed means that the bus waits longer than necessary, causing larger access delay and lower performance. Such non-optimal configurations are also subtle because they don't have obvious symptoms e.g. exceptions.

Another example is about clock configurations. In an embedded system, many hardware modules are running at different frequencies. A CCU is responsible for the clock distribution in AURIX. The clock configuration is set by writing the configuration register and then by setting the update request (UP) field that works as an update switch for updating the module frequencies. An issue with this is that the register doesn't show the actual state when the UP field is neglected. The newly-written value is read but the clock runs in the previous state in such situation. A conventional debugger may have a clock configuration view based on the current register values, which provides misleading information to software developers. A wrong clock configuration could lead to many problems in hardware modules due to the close relation between the clock and the module. Debugging becomes even more complex and challenging based on this wrong information.

The above two real-life examples show the consequences of wrong hardware configurations and the challenges in debugging hardware configuration. These challenges are not easy to be conquered by conventional debugging methods as shown in left part of Figure 4.6. In the conventional process, a user manual is the key and is created by a hardware designer. The user manual defines all the rules which are supposed to be followed to use hardware properly. Then, a software developer receives a user manual copy and programs hardware relying on this user manual. If anything goes wrong with the hardware configuration, the software developer has to go through the user manual again and try to figure out the root cause. Even worse, sometimes there are no obvious symptoms of a misconfiguration or the symptoms are not well understood by the software developer. The error may go into production without any notice, which might be life-threatening for products like cars.

An improved hardware configuration validation methodology is proposed to increase

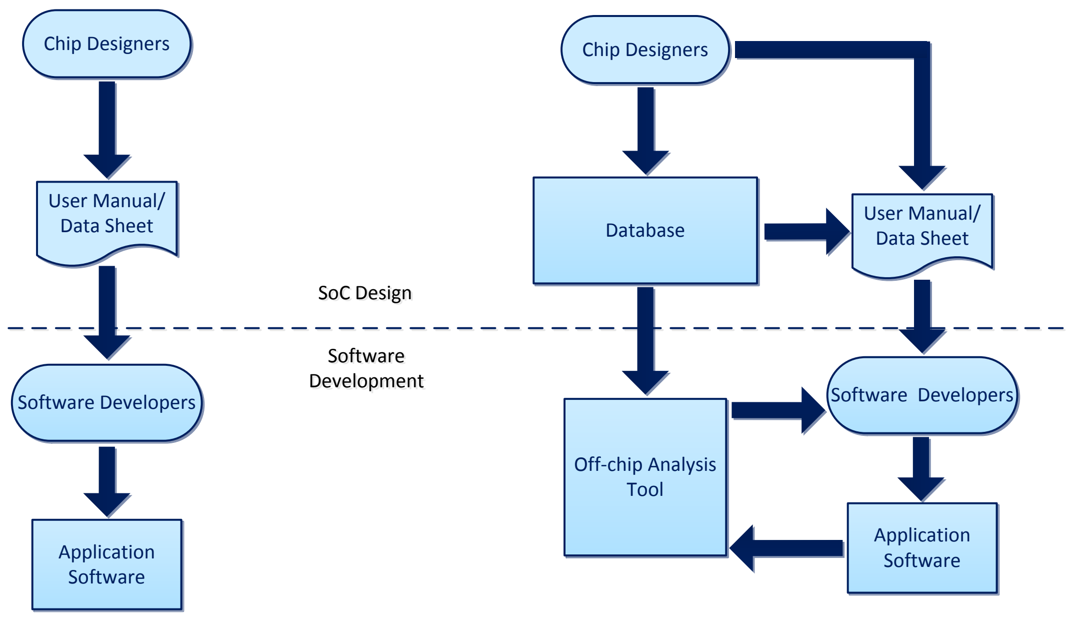


Figure 4.6: Comparison of the conventional design flow (left) and the methodology proposed in this paper (right) [49].

the developers' productivity [49]. It is shown in the right part of the Figure 4.6. Instead of only replying on user manual as the channel between hardware developers and software developers, the proposed methodology add a new channel that is a database and provide a connection with an off-chip analysis tool. The hardware configurations can be checked by the off-chip analysis tool directly on the running system using hardware tracing. This checking is not relying on the symptoms, thus many hard-to-find configuration issues can be detected easily by this tool. The software developers will be notified if any issues appear.

The methodology consists of three major components [49]:

- A formal **configuration rule language** based on LTL describes valid hardware configurations.
- A **configuration rule database** stores those rules.
- An off-chip **analysis tool** checks whether violations of the configuration rules occurred at runtime. The tool uses the tracing facilities available in many of today's COTS chips to gather data from the running system without modifying its target software. It then compares the traced system behavior to the rules and verifies whether the hardware-related configurations are properly applied.

Compared to the other related research, this methodology covers both the SoC development phase and the software development phase. The rules are directly input by SoC designers and are valid for a specific device regardless of applications It uses no-intrusive hardware tracing and needs no additional hardware. Moreover, it can efficiently solve hardware configuration issues that are time-consuming and hard-to-detect.

4.2.1.2 Methodology

The proposed methodology has two phases namely the SoC design phase and the software development phase.

SoC Design Phase: at the SoC design phase, basic software rules are defined by the hardware developers. These rules should be complied during runtime in order to work as expected. Instead of the conventional way, in which these rules are written into the user manual, the rules are formalized to a standardized format. The standardized format acts as a bridge between the SoC design phase and the software development phase. It is easier for automation tools to read. The rules are also transformed into the user manual in an unambiguous mathematical format. The rules should be presented precisely.

Software Development Phase: the software development phase is corresponding to the software development on the given hardware by software developers. The rules defined by the hardware vendors are supposed to be followed by the software developers. An off-chip analysis tool helps to guarantee that the pre-defined rules are not violated by the software during runtime. Those rules are stored in a database that provides connections to the off-chip analysis tool. The analysis tool configures the tracing hardware to focus only on the interesting register accesses. Then a diagnosis report is generated to inform software developers of improper configurations and the rules by comparing the traced access history to the rules.

The rules are managed in a hierarchical structure as shown in Figure 4.7. The SoC level consists of several SoC hardware modules, which contains detailed rules. Core modules means that they are always utilized by the application and should be always checked, including CPU, flash, buses and clock configurations etc. Other modules e.g. DMA, Ethernet MAC are optional to be validated. With this structure, the rules are organized in a logical and convenient way.

4.2.1.2.1 Linear Temporal Logic

The rules are supposed to be easy to understand and contain temporal information. LTL is applied to define the rules because of several advantages. It was first proposed for formal verification in [54] and is a core part of several runtime verification systems [59] due to its flexibility and simplicity. As the name explains, it is an important method to express the time related properties. It is also expressive and unambiguous.

An introduction of LTL is described here [49]. A LTL (φ) is built up from a set of propositional variables ($\boldsymbol{P}$), Boolean operators ($\neg, \wedge, \vee ...$) and temporal operators (**X, G, F, U ...**). For example,

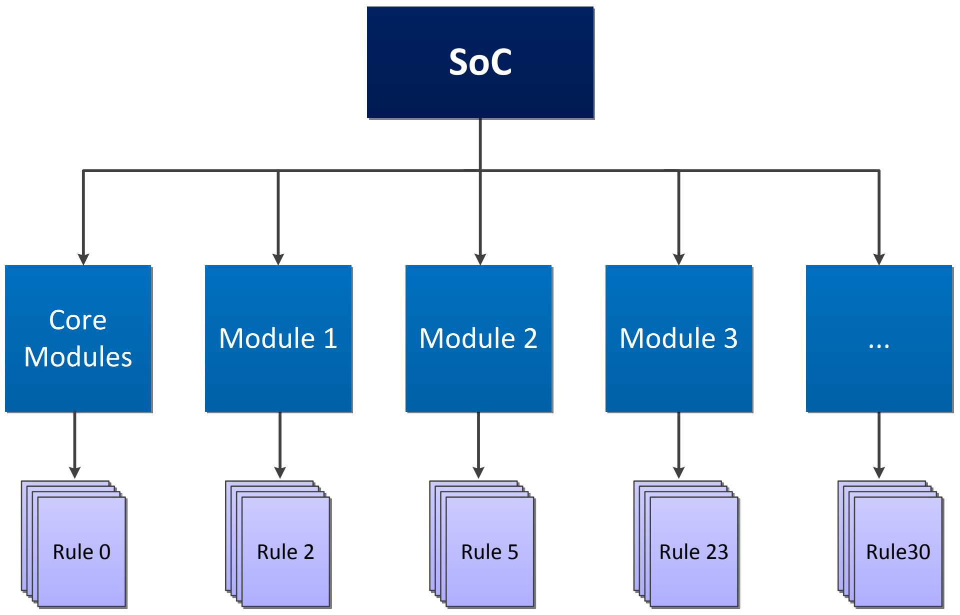


Figure 4.7: The rule management hierarchy in the off-chip analysis tool [49]

$$\varphi = (p_1 \vee p_2) \ \mathbf{U} \ \neg p_3 \tag{4.1}$$

with $p_1, p_2, p_3 \in \boldsymbol{P}$.

The LTL example shown above means that either p_1 or p_2 is valid until p_3 is not valid. A subset of temporal operators, that indicate the temporal relations, is shown below.

- **X** ($\bigcirc$) for next.
- **G** ($\Box$) for always.
- **F** ($\Diamond$) for eventually.
- **U** for until.

If w is a sequence of $\boldsymbol{P}$, the satisfaction relation $\models$ means that w satisfies the LTL formula φ. Several satisfaction examples Equation 4.2 4.3 4.4 4.5 with different temporal operators are given below, corresponding to the diagrams **(A)**, **(B)**, **(C)** and **(D)** in Figure 4.8 respectively.

$$w_1 \ \models \ \bigcirc \, p_1 \tag{4.2}$$

$$w_2 \ \models \ \Box \, p_1 \tag{4.3}$$

$$w_3 \;\models\; \Diamond\, p_1 \tag{4.4}$$

$$w_4 \;\models\; p_1 \mathbf{U}\, p_2 \tag{4.5}$$

The current state is at index 0. The sequence **(A)** satisfies the LTL formula Equation 4.2 because at the next state ($w_1(1)$) p_1 holds. The Equation 4.3 is also satisfied since p_1 always holds. In the sequence **(A)**, p_1 will be finally true, which also satisfies the Equation 4.4. The last Equation 4.5 means that p_1 must be true until p_2 becomes true, which is exactly the case of the sequence **(D)**.

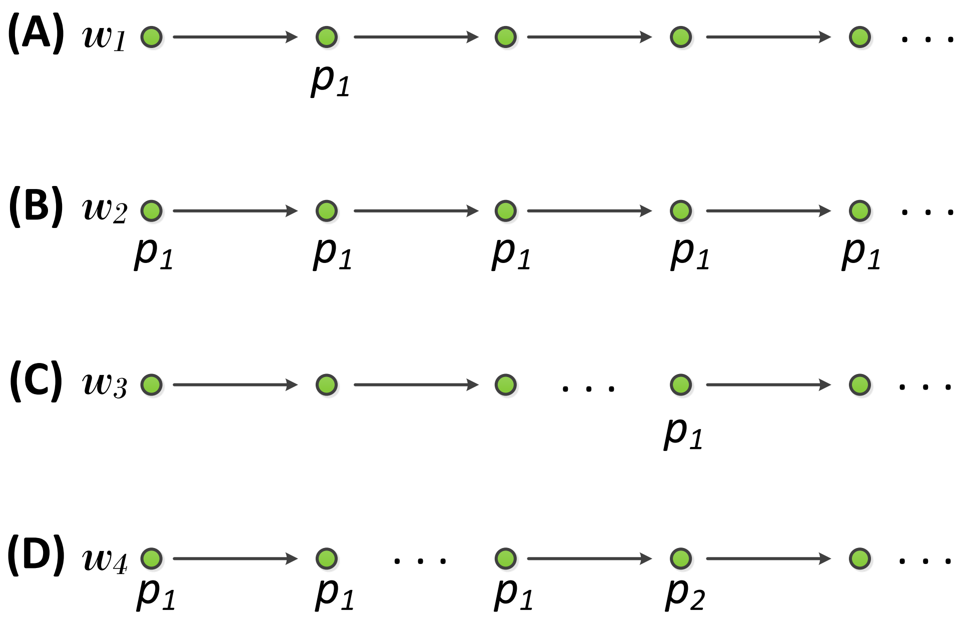


Figure 4.8: Diagrams of temporal operators

The description of LTL is not the focus of this methodology so only simple examples are explained. The detailed information about LTL is available in [54, 59, 118].

4.2.1.2.2 Rule Definition

There are two types of rules namely combinational rules and sequential rules. LTL is applied to both of them.

A combinational rule defines the combinational issues that depend only on the present state. The flash configuration issue belongs to this group. The combinational rule is defined in Equation 4.6

$$R := Rule(Combinational, \varphi); \tag{4.6}$$

A sequential rule covers the sequential issues that depend not only on the present state but also on the previous history, for instance, the clock configuration update issue. It is defined in Equation 4.7. The *direction* shows in which direction should be searched starting from the condition, either forward or backward. It is used for two scenarios e.g. an enabling access should be before the register access or an updating access should be after the register access.

$$R := Rule(Direction, \varphi); \tag{4.7}$$

φ as defined above, is an LTL formula describing the register access sequence that is supposed to be obeyed. The propositional variables that are the basic elements in an LTL are defined as $R_{\text{register}} \in \boldsymbol{P}$. R_{register} is a qualification function that validates whether a register access belongs the predefined access as shown in Equation 4.8. The *origin* as the name explains indicates the origin of the access, e.g. CPU0, DMA. This deals the cases that only specific modules are allowed or not allowed to access the some registers. The *type* argument defines the access type and access width e.g. a read 32-bit access. The other arguments including register, field and value specifies the detailed register access information. It is noted that not all arguments must be filled to qualify an access and the granularity of the access qualification is flexible. If an argument is not defined, this argument doesn't matter in the qualification.

$$R_{\text{register}} = f(origin, type, register, field, value) \tag{4.8}$$

An example based on a real-life case is given here. The configuration of a module can be modified by writing the register `CONFIG`. In order to validate the newly written configuration, the register `UPDATE` has to be set, which is commonly applied in many cases. A rule in Equation 4.9 is defined for this example. The equation indicates that a sequential rule which means that a write access to register `CONFIG` happens, then eventually a set access to register `UPDATE` must be observed, as defined in Equation 4.10.

$$R = Rule(forward, \varphi); \tag{4.9}$$

$$\varphi = \Box(R_{CONFIG} \Rightarrow \Diamond R_{UPDATE}) \tag{4.10}$$

4.2.1.2.3 Rule Checker

A rule list can be generated by selecting the rules by developers as in Figure 4.9. Once a module is selected, all rules under this module will be added to the list and checked during runtime. If necessary, a SoC restart can also be triggered by the tool. To shorten the tracing duration, the proposed method only configures the MCDS to trace limited to the interesting scope, which further decreases the data generation bandwidth. The interesting tracing scope differs for each rule as different rules may involve different registers and address ranges. Then, the trace data is processed by the tool and compared

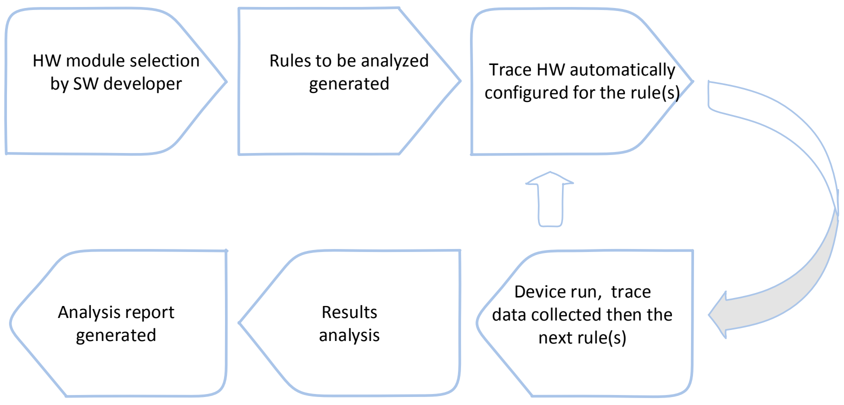

HW module selection by SW developer
Rules to be analyzed generated
Trace HW automatically configured for the rule(s)
Device run, trace data collected then the next rule(s)
Results analysis
Analysis report generated

Figure 4.9: The analysis steps of the proposed method

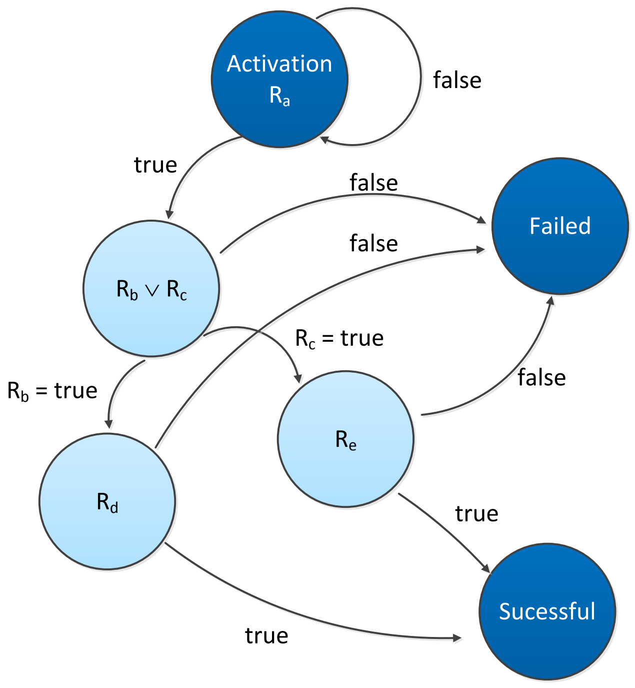


Figure 4.10: A rule checker example

to the existing rules in database automatically. Finally, a report is created by the method and detected issues are highlighted to developers.

A finite state machine (FSM) is utilized to check whether the satisfaction relation between w and φ is true as shown in Equation (4.11).

$$w \models \varphi \ ? \tag{4.11}$$

The collected trace data is compared against the pre-defined selected rules by a rule checker that is constructed as an FSM diagram based on the rule. A rule checker example is given in Figure 4.10 and the corresponding equation is in Equation (4.12). It starts with an activation condition, whose fulfillment means rule activation. An intermediate state always falls into failed state when the current requirement is not fulfilled. Then, if the successful state is reached, the rule is qualified and fulfilled. Otherwise, a rule violation is reported.

$$\varphi = \Box(R_a \Rightarrow (\Diamond R_b \wedge (R_b \Rightarrow \Diamond R_d))) \vee \Box(R_a \Rightarrow (\Diamond R_c \wedge (R_c \Rightarrow \Diamond R_e))) \tag{4.12}$$

A node in an FSM stands for a propositional variable, as a collection of R_{register}. It defines the register access requirements in the current state e.g. a write access of a specific value to a field. A false condition means that the current node is not fulfilled. Depending on the result of propositional statement, the next state will be reached. The arrow between two states indicates the next state condition.

4.3 Diagnosis methodologies of non-functional issues

In this section, non-functional issues are described. These issues may cause problems e.g. performance degradation which is not fatal. However, these issues may also be harmful e.g. missed real time deadlines due to the issues, which may not be unconsidered in the system timing analysis.

First a non-intrusive approach to observe and analyze DMA channel activities is designed. For DMA channel activities, there are no well-defined rules to determine whether the activities are right.

Then, two methodologies focused on the profiling of interrupts and locks are introduced. These methodologies cover both functional issues and non-functional issues. The interrupt profiling methodology is mainly focused on the interrupt performance. However, if an important interrupt cannot be handled in time, it may lead to a functional error. The lock profiling methodology also emphasizes on lock performance. Functional synchronization errors can also be detected by this methodology. Finally, contention analysis methodologies of flash memory and data memory are described. They are aiming at performance issues.

Finally, contention for shared resources is analyzed. It is usually invisible for software developers but can heavily impact the performance in multi-core systems. In this chapter, two methodologies with two different emphases are introduced. One is focused on program and the other one is related to data. The program flash contention analysis methodology is able to detect program flash contention which is rarely detectable by other existing tools. The data memory contention methodology is capable of analyzing both data contention and data locality in a quantitative way, facilitating the balanced performance optimization.

4.3.1 DMA channel activity analysis

4.3.1.1 Introduction

In the automotive industry more cars are equipped with ADAS and there is a trend that autonomous driving cars will be released. In order to cope with new challenges introduced by these sophisticated features, more requirements of communication and sensing are posed to embedded systems, and therefore more peripherals are added to embedded systems. Using DMA is more preferred as this avoids the involvement of CPUs and requests can usually be serviced faster and it is quite common to have peripherals cooperated with DMA channels. Thus, DMA has been widely applied in modern embedded systems.

In the recent multi-core/manycore design, a combination of Scratchpad Memories (SPM) [119] and DMAs has been proposed as an alternative to traditional caches, where data transfers through the memory hierarchy are explicitly managed by the software [120]. The advantages of this alternative include reduced power consumption, and better performance;. However, this alternative also increases the programming complexity that is challenging for software developers to manipulate memory accesses through multiple layers, leading to more error-prone software.

Until now, most debugging solutions are CPU-centric as CPU is the module executing software, for example, setting break point, performance analysis of specific tasks, race condition etc. DMA as a type of widely utilized resource is underestimated and there are not many debugging solutions focusing on DMA. As a classic debugging solution, setting a break point may be the first thing a software developer trying to do when malfunction happens, which freezes the execution and gives the software developer a "screenshot" of what is happening at that moment. However, a break point cannot stop a DMA channel as DMA usually is not halted by the debugging module. The interaction between DMA channels and peripherals cannot be frozen by a break point. Software developers can only imagine what is going on according to the DMA configurations, and this might be different from the actual situation.

To close this gap between software developers' understanding and systems' real states, a solution which is able to illustrate what is happening to DMA channels and peripherals is required. In this section, a new method making use of MCDS is proposed to trace and visualize the DMA channel activities and function information. Function tracing and also flow tracing are added to make the results more comprehensive. This information is visualized via an easy-to-understand Gantt chart.

DMA is widely used in embedded systems to service peripherals and to boost the system performance by using DMA to handle memory accesses explicitly. However, only a few studies are focused on the analysis of DMA [120]. For instance, in order to deal with the DMA channel contention that means two DMA channels operate on the same memory and at least one modifies the memory, an automatic formal verification tool called SCRATCH was designed by Donaldson et al. [121, 122]. SCRATCH runs on a C program for Cell BE processors, which consist of two different types of computational units: a power processing element (PPE) and several synergistic processing elements (SPE) [123]. The Cell architecture was initially designed for gaming and used for PlayStation 3. Then it is embraced by the scientific community for high performance computing. In the Cell architecture, DMA as a central means of intra-chip data transfer is equipped for each SPE and PPE and is controlled directly by software developers. The SCRATCH tool uses a combination of SAT-based bounded model checking and k-induction to detect or prove absence of DMA races [121, 122]. In order to detect DMA races, a dynamic approach based on software instrumentation was proposed by Saidi et al. [120]. It is a monitoring algorithm which observes during runtime dynamically the arrival of a sequence of DMA events and emits a verdict each time a DMA race is detected. Moreover, it can enforce a correct and DMA race-free execution of the program.

4.3.1.2 Solution

The DMA analysis proposed here is based on features supported by Infineon's MCDS. The DMA channel activity information is collected by MCDS and then presented to users. Before that, the background knowledge of interrupt handling and DMA operations is introduced.

4.3.1.2.1 The DMA module in AURIX

DMA is a hardware module that is able to access system memories independent of the CPU, which reduces the service latency and saves computation power. The DMA module in AURIX supports 128 DMA channels and the channel index indicates the priority. DMA channel 127 has the highest priority while channel 0 has the lowest. There are two DMA move engines for parallel execution of requests. DMA channel requests coming from the interrupt router (IR) can initiate a DMA transfer. Figure 4.11 shows the block diagram of the DMA module and the closely related modules such as buses and interrupt router are also illustrated. The transaction control sets are used to store DMA channel configurations. Requests from the IR are handled and arbitrated in the DMA channel request and arbitration sub-block. The DMA module is also able to generate requests to the IR that can then be served either by CPU or DMA. Programming of the DMA transaction control sets is via a bus slave interface while DMA data operations are via a master interface.

A DMA move is the basic operation which consists of a read operation that loads data from a data source and a write operation that stores data to a data destination as shown Figure 4.12. Each DMA move can read and write data with a width of 8 bit, 16 bit or 32 bit for SPB and 8 bit, 16 bit, 32 bit, 64 bit, 128 bit or 256 bit for SRI. A DMA transfer can be composed of several DMA moves while a DMA transaction is composed of several DMA transfers. For instance, the example in the figure has a DMA transfer

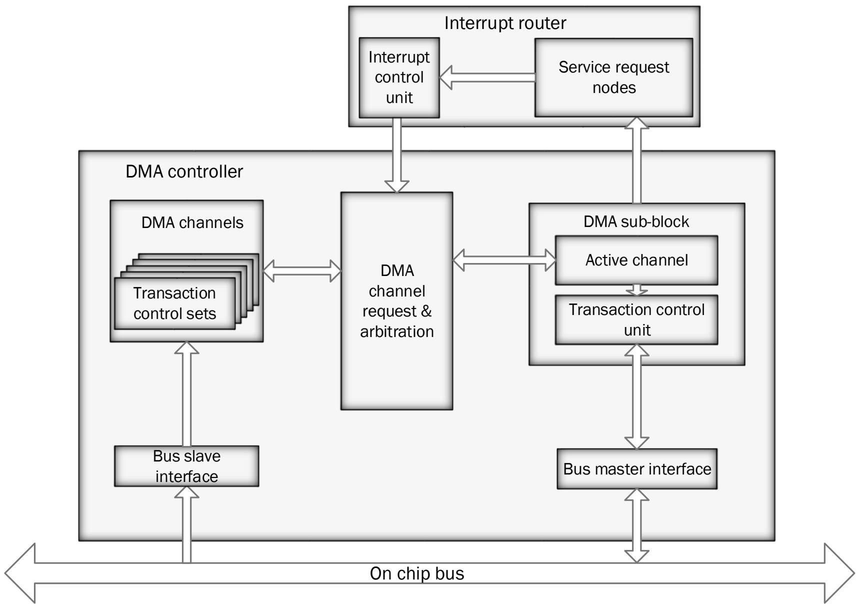


Figure 4.11: The DMA block diagram in AURIX [106]

with three DMA moves.

4.3.1.2.2 DMA tracing

MCDS has many designated features for different hardware modules. DMA as a widely applied module is also supported by the MCDS tracing. The DMA module provides a 8-bit vector that is traced and collected by MCDS. The meaning of each bit is described in TABLE 4.3. The lower 7 bits show which DMA channel is active in the move engine 0 and the 7th bit indicates the availability of the move engine 0. Similarly, the upper 8 bits contain the same information of the move engine 1. The traced data in DMA will be transferred to OCDS Trigger Bus (OTGB) and trace messages are generated there. During the message generation, time stamps are also created and attached to messages.

Table 4.3: TS16_PF Trigger Set Channels [106]

Bit	Name	Description
[6 : 0]	CH0	Channel number active in move engine 0
7	ME0	Move engine 0 idle/active
[14 : 8]	CH1	Channel number active in move engine 1
15	ME1	Move engine 1 idle/active

Based on DMA tracing, the activity status of DMA channels become available. Sometimes, a DMA request comes and it is not handled in time because DMA move engines are fully occupied by other channels. In order to observe this delayed DMA channel han-

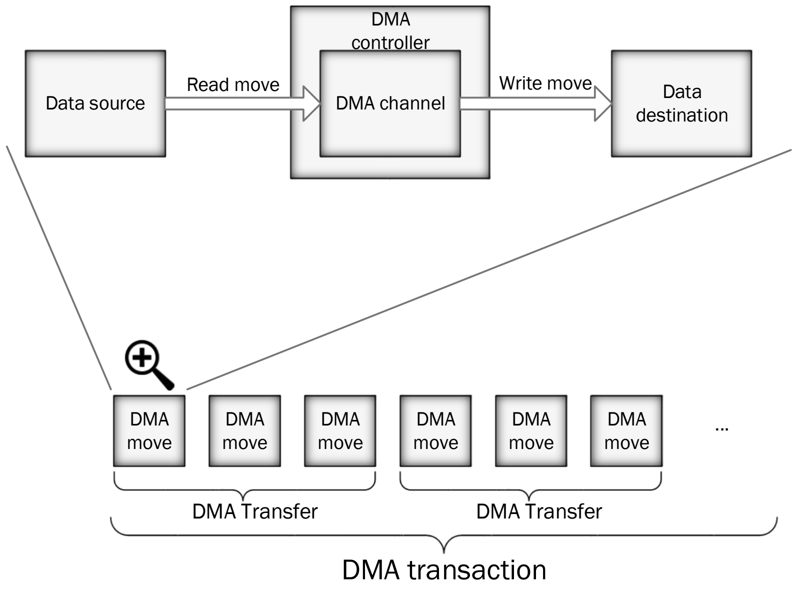


Figure 4.12: DMA terms: DMA moves, DMA transfer and DMA transaction in AURIX [106]

dling, service requests from hardware modules are also traced by MCDS, which involves interrupt router. Four different service requests can be traced in parallel by configuring MCDS as described in TABLE 4.4. The selected service requests should be configured before tracing in the interrupt router. It can be either configured by the application software or by the debugging interface. In this proposed method, it is configured during the configuration of MCDS via the debugging interface. When a service request comes, the corresponding bit will become one, meaning the arrival of the service request. The collected data is transferred to MCDS also via OTGB. With the help of this information, the service request delay which is defined as the delay between request's arrival and its DMA service is measured.

Table 4.4: Trigger Set Interrupt Selection [106]

Bit	**Name**	**Description**
0	AIL	Any interrupt trigger lost
1	SIL	Selected (INT_OIXMS) interrupt trigger lost
2	SI0	Selected (INT_OIXS0) interrupt 0
3	SI1	Selected (INT_OIXS1) interrupt 1
4	SI2	Selected (INT_OIXS2) interrupt 2
5	SI3	Selected (INT_OIXS3) interrupt 3
[7 : 6]		Reserved

Only relying on DMA channel information to diagnose issues and anomalies is not easy. This is because software developers are usually thinking from a software-centric perspective. It will be easier to link DMA activities to the software execution for example functions that are executed by CPUs when some DMA channels are active. Fortunately, it can be fulfilled with MCDS function tracing or flow tracing. The details of function tracing and flow tracing are explained before.

4.3.2 Interrupt profiling analysis

4.3.2.1 Introduction

An interrupt is a signal to indicate an event that needs immediate attention [124] by service providers. Interrupts are widely used in both embedded systems and general purpose computers. They are designed for multi-tasking especially for real-time computing. Depending on the priority, the interrupted CPU may suspend its execution and store the current states e.g. current register values. Then the interrupt handler will be executed. Usually this process takes only a few clock cycles and acts as an efficient solution. The service providers are not limited to CPUs and the DMA can also be a service provider. Many peripherals work in cooperation with DMA as discussed in the previous section. For example, an SPI module triggers a service request to the interrupt router to empty its received data. Then the interrupt router will forward this request to the configured service provider which is the DMA module. The corresponding DMA channel will then be activated to handle the task.

In real-time systems, interrupts are applied to trigger tasks which are supposed to be finished in time. Timely handling of interrupts is critical for such systems, especially when the task is safety critical. Unfortunately, interrupts are difficult to predict: they alter the program control flow and complicate the invariants in low-level code [78]. Interrupt handling can be delayed or blocked by many reasons. For instance, the interrupt system is disabled by software or higher priority interrupt occupies the service provider or critical interrupt is frequently preempted by other tasks. Therefore, interrupt profiling that helps software developers to understand their system and find the suspicious interrupt behavior benefits reliable and efficient system development. As the system timing behavior is critical for real-time systems which are aimed to be not changed, the interrupt profiling method is designed to be non-intrusive and based on hardware tracing.

With the help of the proposed interrupt profiling method, interrupt handling information such as interrupt arriving time, response latency, interrupt duration, blocking time is measured. Software developers will get direct impression of how interrupts work in their systems. Based on the measured information, suspicious interrupt behaviors will be highlighted, warnings and suggestions will be proposed by the interrupt profiler. Performance information of the interrupt system is also measured and available for users. Software developers have the freedom to decide whether they should implement the suggestions.

This method has several advantages over the other research. First, it uses the dynamic analysis instead of the static analysis to achieve more accurate results and it applies non-intrusive hardware tracing. It is the first interrupt profiling method based on hardware tracing as far as known. With the help of non-intrusive tracing, the timing of interrupts is not influenced by the observation. Second, the method is independent of OSes and applications. This is preferred for the analysis of embedded systems because there is no dominant OS on embedded systems like Windows on PC.

4.3.2.2 Solution

In this chapter, an interrupt profiling methodology is designed for Infineon TriCore based on the special features provided by MCDS. In order to facilitate the linking between the trace data and interrupts, several terms of time point related to interrupts are defined.

4.3.2.2.1 Interrupt time points

The term definitions here are different from other research as these terms are defined from the tracing perspective as shown in Figure 4.13.

Service Request Arriving Time (SRAT) the time when the service request arrives at the interrupt router system, corresponding to the corresponding bit of this service request in OTGB becomes active.

Service Request Arbitration Time (SRART) the time interval between the SRAT and the time when the arbitration winner that is acknowledged by the service provider is generated. Note: cannot be observed if the arbitration winner of current round is the same as that of the last arbitration round in current MCDS implementation.

Service Routine Starting Time (SRST) the time when the first instruction of the Interrupt Service Routine (ISR) of this service request is executed, corresponding to the time when the first instruction of the ISR is finished.

Service Routine Ending Time (SRET) the time when the last instruction of the ISR of this service request is executed, corresponding to the time when the RFE instruction of the ISR is finished.

Service Request Response Latency (SRRL)) the time interval between the SRAT and the SRST.

Service Routine Duration (SRD) the time interval between the SRST and the SRET.

Service Duration (SD) the time interval between the SRAT and the SRET.

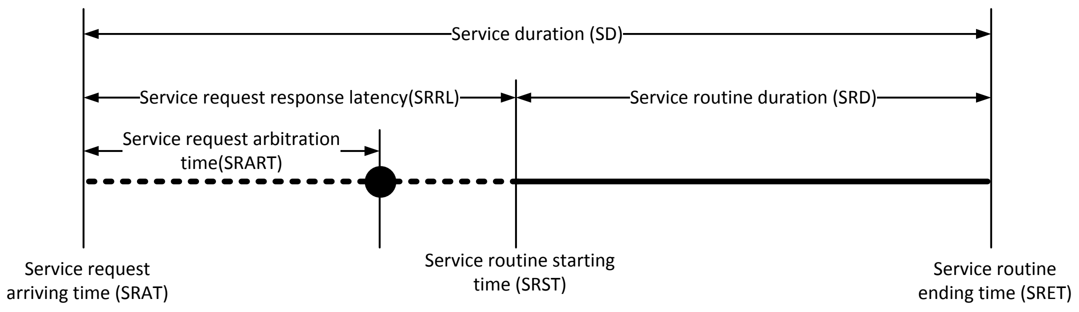

Service duration (SD)
Service request response latency(SRRL)
Service routine duration (SRD)
Service request arbitration time(SRART)
Service request arriving time (SRAT)
Service routine starting time (SRST)
Service routine ending time (SRET)

Figure 4.13: Definition of interrupt from the tracing perspective

If several service requests to the same service provider come one by one, requests will be handled by the service provider relying on the priority, which is stored in the interrupt router module. An example of this case is illustrated in the Figure 4.14. When an interrupt routine is entered, usually the interrupt will be disabled to guarantee that it is executed without interruption. Some interrupts may also have interrupt enabled to allow higher-priority interrupts to preempt. In such cases, interrupts will be embedded and stacked, meaning several interrupts are running on the same service provider and low priority interrupts are interrupted by higher ones.

In AURIX, four different interrupts can be traced in parallel as shown TABLE 4.4 that is introduced in section 4.3.1. Additionally, a trigger set which records for a specific service provider is also provided as described in TABLE 4.5. With the help of this trigger set, the SRN information of the previous arbitration winner is recorded. In this way, the service routine execution can be linked to the SRN and the blocking duration can also be derived. In AURIX, the arbitration process always continues, meaning the arbitration winner may change when a request with higher priority comes before the acknowledgment from service provider. The change of arbitration winner is able to be traced in MCDS.

Table 4.5: TS16_SP Trigger Set Service Provider

Bit	Name	Description
[9 : 0]	ASR	SRN index of last arbitration winner for this service provider
[14 : 10]		Selected (INT_OIXMS) interrupt trigger lost
15	SIP	One or more requests waiting for arbitration by this service provider.

The traced trigger set information is combined with the flow tracing, which contains the whole execution history of one CPU. Interrupts are supposed to be started from entries in the interrupt vector table, the address of which is stored in the base interrupt vector. Interrupt priorities are calculated based on the offset from the base address, given the size of individual entry. For example, if the interrupt priority is 0, then the interrupt routine's entry of this interrupt is supposed to be the first one.

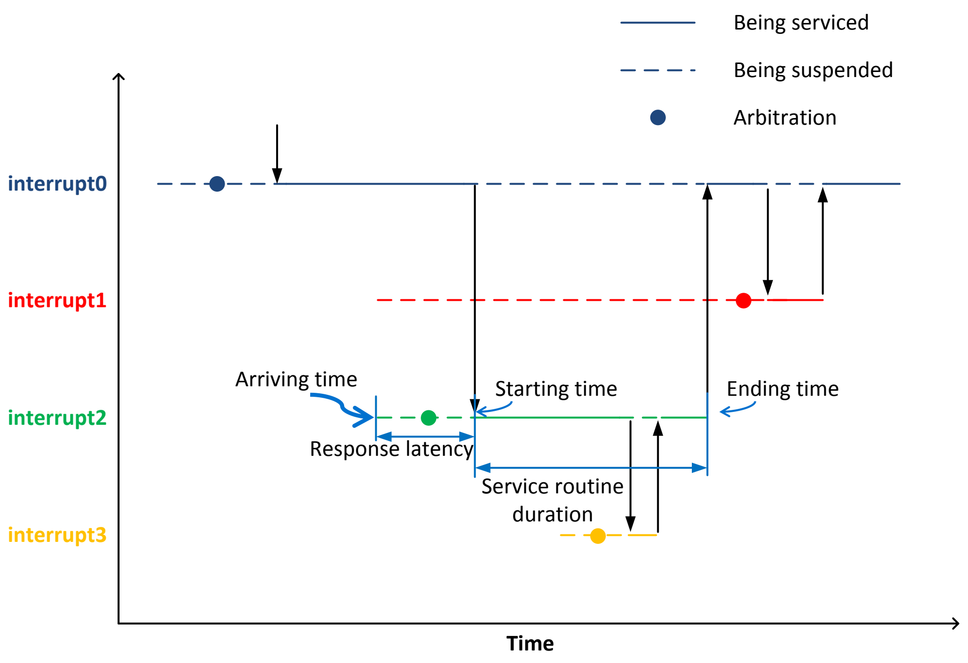


Figure 4.14: An embedded interrupt example: four interrupts with different priorities come one by one

SRAT: for each tracing, MCDS is able to trace four interrupts in parallel and $SRAT$ is measured by observing the corresponding bit in TABLE 4.4.

SRART: similarly, $SRART$ is measured by observing the arbitration winners in TABLE 4.5.

SRST: a service routine always starts from the interrupt vector table so the $SRST$ can be easily obtained by recording the execution finishing time of the first instruction of an interrupt.

SRET: owing to the fact that an interrupt also ends with an RFE instruction, the $SRET$ is derived by recording the execution finishing time of the RFE instruction.

SRRL: $= SRST - SRAT$.

SRD: $SRET - SRST$.

SD: $= SRET - SRAT$.

The whole process of handling an interrupt looks like this: an interrupt request is raised by a node. Then, the request is forwarded to the corresponding ICU as described in Figure 3.3. If the configured service provider is a CPU, CPU is informed by the ICU. After the acknowledgment of request from the CPU, the CPU enters the interrupt vector table after storing all current running contexts. Depending on the interrupt priority, an entry is selected and entered. Afterward, the CPU can jump to a programmed interrupt function.

Priority information is important for the analysis but it is not contained in the trace data. Instead it should be derived afterward during the post-processing. This is achieved by making use of the interrupt vector table that is the entry for entering interrupt routine. The flow trace provides the executed program flow. As the base address of the Base Interrupt Vector (BIV) is static information and can be obtained by reading the corresponding register, the starting point of this table is available. The offset address then can be calculated and the priority is derived according to the Equation 4.13.

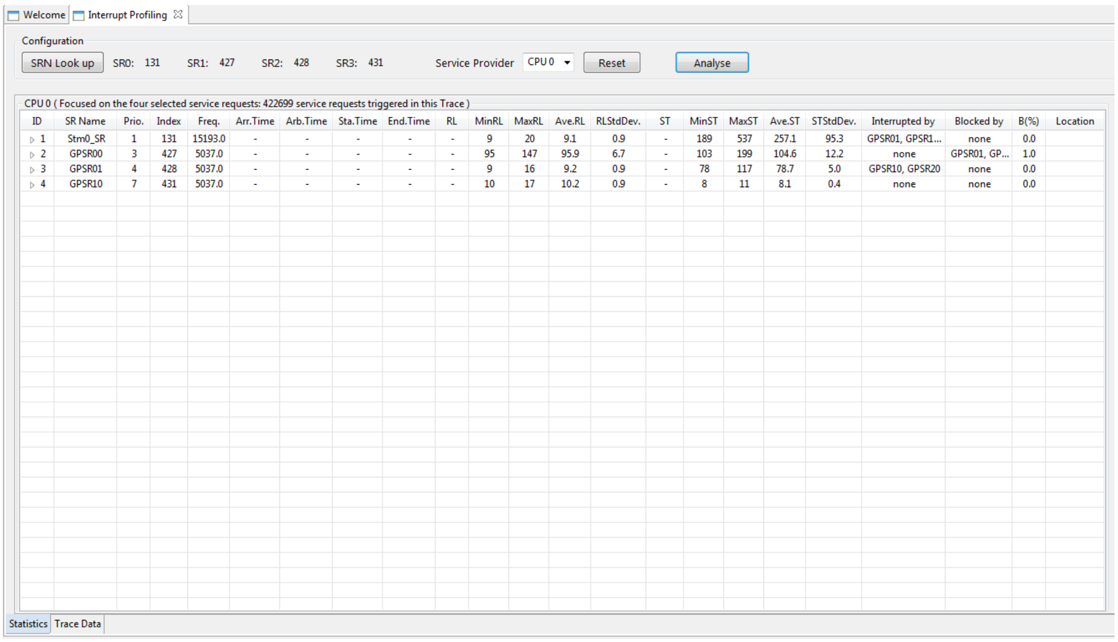
$$Entry\_address = BIV + priority * single\_entry\_size \tag{4.13}$$

Blocking are caused by either an interrupt with higher priority or disabled interrupt system. An interrupt with lower priority must wait until an interrupt with higher priority is finished. In this case, *SRRL* will be extended. The interrupt system can be disabled by Hardware (HW) or Software (SW). For AURIX, the interrupt system is automatically disabled when an interrupt is being serviced. It can then be enabled manually by software, allowing embedded interrupts. It is also quite often to disable the interrupt system manually by software when it is not supposed to be preempted, for instance, during the watchdog servicing. If the interrupt system is disabled and no interrupt can be serviced, leading to longer *SRRL*. Both cases can be derived as the priority information is already available and manually enabling or disabling instructions are observable in the flow tracing.

Based on the above basic measurement, more sophisticated metrics are designed. The contended interrupt is defined as the interrupt blocking the measured interrupt. It can be detected by checking if measured *SRRL* is longer than normal and another interrupt is also running before the measured interrupt serviced. A list of contended interrupts is summarized and shown. Blocking rate is the chance of being blocked by another interrupt or by the disabled interrupt system. The maximum and minimum *SDs* are also calculated to show the stability of the measured interrupts. If the blocking rate of an interrupt is very high, it can be optimized by rescheduling or changing the priority numbers.

The locations of interrupt vector table and interrupt routine do matter as they heavily influence the interrupt performance. For example, if an interrupt vector table and interrupt routines are stored in the PSPR0 which is inside CPU0, the interrupt handling for CPU0 will be faster compared to the case that they are stored in the program flash. Suggestions can be provided to allocate the interrupt vector table and the interrupt routine to a better location if the interrupt performance can be optimized.

Both basis measurement and more sophisticated metrics are measured and displayed



SR0: 131 SR1: 427 SR2: 428 SR3: 431 Service Provider: CPU 0

CPU 0 (Focused on the four selected service requests: 422699 service requests triggered in this Trace)

ID	SR Name	Prio.	Index	Freq.	Arr.Time	Arb.Time	Sta.Time	End.Time	RL	MinRL	MaxRL	Ave.RL	RLStdDev.	ST	MinST	MaxST	Ave.ST	STStdDev.	Interrupted by	Blocked by	B(%)	Location
▷ 1	Stm0_SR	1	131	15193.0	-	-	-	-	-	9	20	9.1	0.9	-	189	537	257.1	95.3	GPSR01, GPSR1...	none	0.0	
▷ 2	GPSR00	3	427	5037.0	-	-	-	-	-	95	147	95.9	6.7	-	103	199	104.6	12.2	none	GPSR01, GP...	1.0	
▷ 3	GPSR01	4	428	5037.0	-	-	-	-	-	9	16	9.2	0.9	-	78	117	78.7	5.0	GPSR10, GPSR20	none	0.0	
▷ 4	GPSR10	7	431	5037.0	-	-	-	-	-	10	17	10.2	0.9	-	8	11	8.1	0.4	none	none	0.0	

Figure 4.15: The screenshot of the interrupt profiling feature

in table. An example table is shown in the Figure 4.15. The table contains the overall information in the first level and the tree can be expanded by clicking. More details about specific interrupt services are in the second level. The highlighted suggestion information is displayed in the console part.

4.3.3 Lock profiling

4.3.3.1 Introduction

Multi-core embedded systems are now commonly applied in daily life. Synchronization methods are needed to regulate the accesses to shared resources in multithreaded systems. As a classical synchronization method, locks are widely used due to the advantages such as easy applicability and simple implementation. They have been studied for years to improve performance, fairness, predictability etc. There is a variety of lock implementations, which are usually optimized for specific scenarios. Applying an inappropriate lock type for a particular scenario may result in issues like low performance or bad fairness [68]. Hugo Guiroux et al. have found that no single lock is systematically the best and some locks are harmful for specific applications. The performance of lock depends on many factors e.g. contention level and the number of threads. The lock type selection is based on the developers' hypothesis, which might be improper. A gap exists between developers' hypothesis and the actual situation. Therefore, a lock profiling tool is needed to increase the transparency of the system and bridge this gap.

For general-purpose computers, there are many lock profiling tools designed for particular operating systems or virtual machines. These tools rely on specific lock acquisition and releasing routines to profile the lock usage. However, there is a different situation in the embedded field, which has no standardized operating systems like Windows or Linux. Creating a lock profiling tool for each operating system is challenging considering the fact that numerous embedded operating systems exist. Thus, a lock profiling tool independent of operating systems is preferred in the embedded world.

In this section, an operating-system-independent spinlock profiling method is proposed. At the moment, the focus is single-binary-semaphore-based spinlocks because they are simple and popular in many applications. For instance, in AUTOSAR Real-time Operating System (RTOS) standard a spinlock interface is defined [125]. There are two common symptoms within different spinlock implementations. First, atomic operations are usually applied during lock acquisition. Modern embedded cores intrinsically support atomic instructions and those instructions are usually used during lock operations. Second, the value of a spinlock can only be modified by its owner. When a thread accesses an occupied lock, it is allowed to read or write the lock but the value of the lock is not supposed to be changed. The proposed spinlock profiling method utilizes the above symptoms to detect the lock location. Given the lock location, lock acquisition and lock releasing information is derived. Then, more detailed information including protected shared resource, waiting time, holding time and failed attempts is collected and analyzed. Basic improper lock behavioral will also be reported to developers.

An example of improper lock type is test-and-set lock, which has been thoroughly studied. Compared to test-test-and-set lock, it has better performance if there is almost no contention. Otherwise, the thread keeps on spinning using the test-and-set atomic operation, which locks the memory until the operation has finished. This spinning not only degrades the current thread but also blocks memory accesses from other threads. The whole system performance is then impacted. With the help of the proposed spinlock profiling tool, this low-performance pattern that is spinning with the atomic operation will be detected and reported to developers. A better suitable lock type can then be selected.

The proposed method makes use of hardware tracing, which is supported by many COTS embedded systems. Such hardware tracing provides information including memory accesses and program flow without instrumenting software. This information can be used to extract the lock candidates and then verify whether a memory location is a lock instance according to the two common symptoms.

This method has two main advantages for embedded systems:

- The lock profiling is not relying on a specific OS. Software instrumentation can also be avoided.
- It not only reports lock statistics but can also notify software developers about improper lock behavior.

4.3.3.2 Lock Profiling Approach

Our approach is designed to detect and analyze spinlocks in an OS-independent, non-intrusive way. In the following, we present it in three parts. First, the spinlock characteristics from a hardware perspective are discussed. In the second part, we show how to link low-level hardware-related trace data to lock operations based on these characteristics. Low-level hardware-related trace data can be used to detect spinlocks and associated lock operations. Finally, in the third part profiling statistics are defined and symptoms to recognize the inefficient spinlock behaviors are described.

4.3.3.2.1 Spinlock Characteristics

In order to detect spinlocks without instrumentation at the source code level and without knowing the lock and unlock function calls, our approach relies merely on the data available on chip. This data includes all program instructions as they are executed on CPU, and all accesses to data memory. Using this information, we formulate a pattern to detect binary semaphore spinlocks.

A binary semaphore spinlock has only a single semaphore for the availability. In this section, only the ones based on a single variable are considered. This group includes many classic spinlock types e.g. test-and-set [64], test-test-and-set [65] and backoff locks [64]. Many queue based locks are not included because they usually have a copy for each thread. Several assumptions are made to limit our analysis scope. (i) the architecture supports atomic instructions and the target lock types also use atomic instructions. Modern multi-core architectures support atomic instructions and most lock types employ these instructions. (ii) lock variable is allocated statically, which means lock variables are allocated during compile time. Many embedded applications especially safety-critical applications utilize static allocation. (iii) thread ID information is provided. Thread tracing is already supported by many commercial tools and is not the focus of this approach, so this information is considered as given.

During the initialization, the initialized value is considered as free while the opposite value is defined as busy. A binary semaphore spinlock, described by the memory address of its lock variable, is observed, if all three conditions below hold.

1. An atomic instruction writing to a memory location is observed.
2. Only the binary values 0 or 1 are written to this memory location.
3. The value of this memory location is only changed by its owner thread, which is defined as the thread that temporarily owns the memory location. A thread becomes the owner by changing a lock's value from free to busy. Then it changes the value back to free, meaning not the owner anymore. During this time, the lock value is not supposed to be changed by any other threads.

4.3.3.2.2 Runtime Spinlock Detection

By applying the described characteristic pattern matching to the observations from an embedded system, spinlocks during the program execution can be detected. A prerequisite is the ability to collect the required data during runtime. For this task, we use the hardware tracing support embedded in most of today's COTS chips.

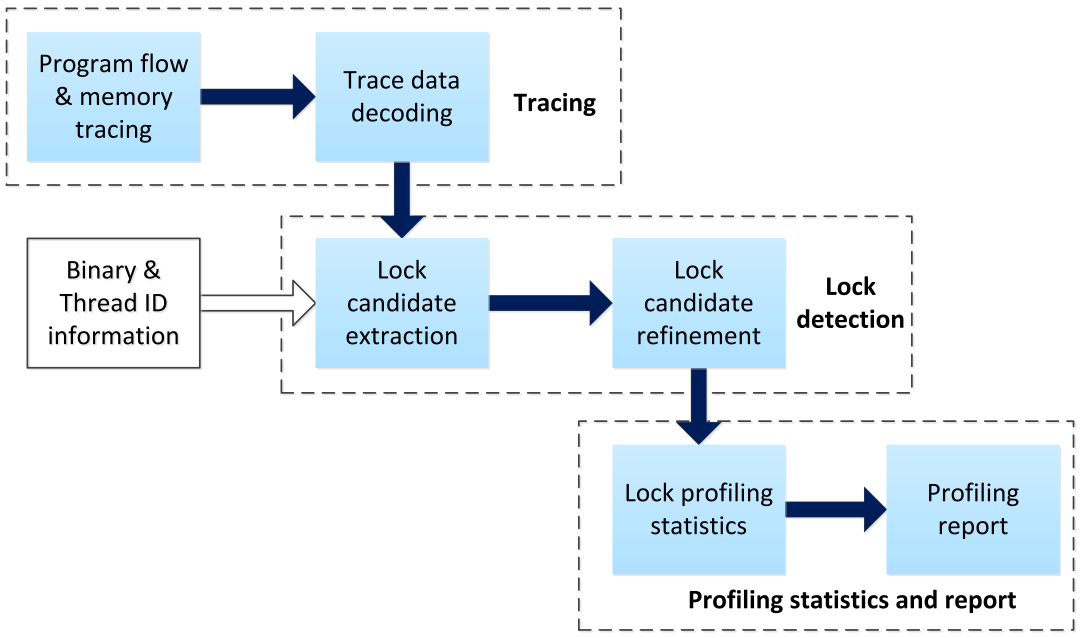

Program flow & memory tracing
Trace data decoding
Tracing
Binary & Thread ID information
Lock candidate extraction
Lock candidate refinement
Lock detection
Lock profiling statistics
Profiling report
Profiling statistics and report

Figure 4.16: The workflow of the proposed multi-phase approach

Tracing is able to observe low-level hardware-related operations e.g. program flow, bus transfers and memory accesses. The traced data is decoded by an off-line post-processing tool. Table 4.6 shows one tracing example from Infineon's MCDS. This example is decoded and slightly simplified.

Table 4.6: A hardware tracing example

Index	*Time*	*Opoint*	*Origin*	*Address*	*Operation*	*Value*
1	19	CPU0	CPU0	0x80000A48	IP	-
2	20	CPU0	CPU0	0x80000A4C	IP	-
3	21	CPU1	CPU0	0x70000030	R32	0x0
4	30	CPU1	CPU0	0x70000030	R32	0x0
5	37	CPU1	CPU0	0x70000030	W32	0x1

A time stamp records the ending time when an operation finishes. For example, the message in the first row records an instruction (0x80000A48) executed by CPU0. The fifth message shows that CPU0 issues a 32-bit write to CPU1's local memory.

Lock detection: hardware tracing enables us to gain insight into the embedded system. On top of this monitoring technique, the spinlock profiling approach is built, as it is presented in Figure 4.16.

The lock detection process is split into two parts. First, we obtain a list of candidate locks by observing the full program stream and picking out all program counters which execute atomic instructions. In COTS tracing systems, this requires the program binary, as the information which instruction was executed is omitted as part of tracing compression process. The memory locations that are accessed by the atomic instructions are marked as lock candidates. The corresponding variable information can also be extracted including symbol, type and size.

Second, the obtained list of lock candidates is refined by following the above requirements 2 and 3. The memory locations with non-binary values are removed from the list. The rule that the busy lock's value is not allowed to be modified except for its owner should be conformed to. This rule is checked against the remaining lock candidates and the not complying candidates are abandoned.

Using this approach, we are able to obtain a list with all memory locations of spinlocks used by the application.

4.3.3.2.3 Profiling statistics and report

Given the lock locations after the refinement, all the memory accesses to these locations are collected and ordered according to time stamps. After initialization, lock acquisition is identified by detecting a value change and a write operation to the lock with the busy value. Only a write to a lock with the busy value does not indicate a lock acquisition. This is because a thread may write the busy value without a value change depending on the lock implementation. For instance, a thread uses SWAP instruction to acquire a lock. It always writes the busy value to the lock no matter which value the lock currently has. It only successfully acquires the lock when the swapped value is free. Accordingly, lock releasing is marked by a similar mechanism with the free value. For example, the last three messages in Table 4.6 show a lock acquisition example. A lock exists at 0x70000030 in CPU1's local memory. CPU0 successfully acquires this lock using the test-test-and-set mechanism.

The *Origin* shows which CPU issues a memory access. Using the information when a given thread is active on a core, a mapping between thread IDs and lock operations is established. The proposed approach itself is independent of thread ID tracing. In order to validate the proposed approach, a tool was implemented based on this approach. In this tool, memory accesses to a special range of memory named OLDA are traced. From the operating system side, thread ID is sent by the operating system. The operating system writes the ID to OLDA when a thread is started or resumed. The OLDA range is located inside the LMU and is a virtual memory. The write operation does not store anything but this bus transfer can be recorded by MCDS. In this way, thread ID information is also included in the memory access tracing.

For hardware platforms that do not support continuous tracing, the trace buffer is limited. This situation is improved by a two-step strategy. First, all memory accesses are recorded by tracing and locks inside are analyzed. This tracing provides a good overview, even though the tracing duration is short. Second, an interesting lock can be selected. At this time, only the selected lock is traced again and the tracing duration is much longer, which allows developers to conduct a deeper analysis about one lock. The two-step strategy offers both an overview and details. Another solution could be continuous hardware tracing using high performance tool hardware, which provides high-speed connection to the chip and a large trace buffer.

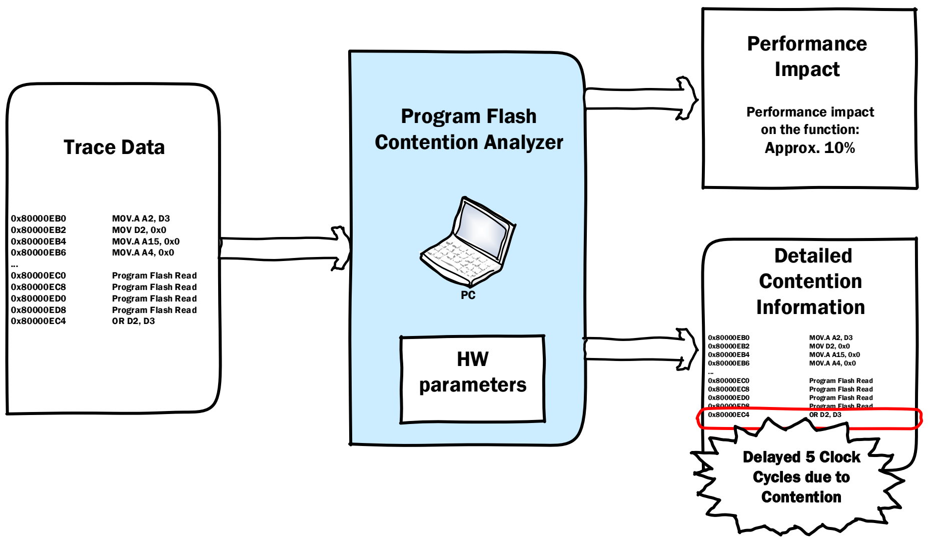
A lock protects a shared resource, i.e. a shared variable or an I/O. It is possible to associate a shared resource with a lock variable using the Lockset algorithm [44, 53]. The algorithm first creates a candidate list for each shared resource. Each time a resource is accessed, locks except for the ones that are currently held are deleted from the list. After several iterations, only the lock that is always held when the resource is accessed remains. This lock is then interpreted as protecting the resource. Resources which are shared among threads but not protected by locks are highlighted for developers.

Many statistics can be calculated to show the lock usage. Waiting time is defined as the time spent from the first attempt to acquire a lock until the lock acquisition succeeds. Holding time is the time between the lock acquisition and the lock releasing. The number of failed attempts is the number of attempts from one thread to an occupied lock by another thread. The first attempt acquisition ratio, as the name indicates, is the ratio of the number of attempts that successfully acquire the lock at the first try to the total number of first attempts. It is calculated for each thread and each lock, showing the contention level. All these statistics are based on memory access tracing in which time stamp information is already provided. Inefficient lock behaviors observed during the tracing such as spinning with atomic operations and thread preemption while holding a lock are reported as warnings. Improvements could be adopted by developers based on this information.

4.3.4 Program flash contention

4.3.4.1 Introduction

Contention for shared resources degrades the overall system performance and the real-time performance of embedded multi-core systems. Program flash that stores program instructions and constant data is frequently accessed by cores. Contention can happen at the program flash interface when more than one core attempts to read it, leading to a larger delay. For example, if a core accesses the program flash that is currently occupied by another core, it has to wait until the previous one finishes. The waiting time can be longer when there are more than two cores. Even worse, the read latency of flash is much larger than the read latency of Random Access Memory (RAM). The number of core stalls caused by program flash contention is much bigger than the number of core stalls by RAM contention. Although in most cases the average number of core stalls is largely reduced by a program cache, the program flash contention impact on a specific non-cached function could be high. This is particularly a problem for systems when such a function is in a critical path, as it cannot be guaranteed to always have the needed instructions in program cache. In order to avoid such problems, program flash contention should be detected and analyzed, which is currently not solved with existing tools. Only when the contention details and the performance loss are analyzed in detail, can it be efficiently reduced by solutions such as program code relocation or a change in the scheduling of program execution on the different cores.


Trace Data
Program Flash Contention Analyzer
PC
HW parameters
Performance Impact
Performance impact on the function: Approx. 10%
Detailed Contention Information
Delayed 5 Clock Cycles due to Contention

```
0x80000EB0        MOV.A A2, D3
0x80000EB2        MOV D2, 0x0
0x80000EB4        MOV.A A15, 0x0
0x80000EB6        MOV.A A4, 0x0
...
0x80000EC0        Program Flash Read
0x80000EC8        Program Flash Read
0x80000ED0        Program Flash Read
0x80000ED8        Program Flash Read
0x80000EC4        OR D2, D3
```

```
0x80000EB0        MOV.A A2, D3
0x80000EB2        MOV D2, 0x0
0x80000EB4        MOV.A A15, 0x0
0x80000EB6        MOV.A A4, 0x0
...
0x80000EC0        Program Flash Read
0x80000EC8        Program Flash Read
0x80000ED0        Program Flash Read
0x80000ED8        Program Flash Read
0x80000EC4        OR D2, D3
```

Figure 4.17: The overview of the proposed program flash contention analysis method

A novel analysis methodology is proposed to detect program flash contention and estimate the performance loss caused by the contention [47]. The methodology is based on the non-intrusive tracing in embedded multi-core systems. Many COTS chips provide such solutions. As a requirement of the methodology, the tracing hardware needs to be able to trace non-intrusively, which means without any impact on execution and timing. The trace data has to include fine grained time stamps for instructions and bus transfers related to the program flash. Based on the trace data and several hardware parameters, program flash contention can be spotted and the performance loss introduced by the contention can also be estimated as shown in Figure 4.17.

After execution, the collected low-level trace data is read into the program flash contention analyzer to find all the spots where the core is idle and halted, resulting in performance loss. These suspicious spots are then further analyzed according to the proposed methodology. Referring to the trace data and several HW parameters available in data sheets, the analyzer decides for each spot whether the delayed instruction fetch is caused by a contention or other factors. Finally, the program flash contention spots are derived and the stalled cycles due to each contention spot are estimated. The output information contains instruction-level program flash contention information, showing every instruction that is fetched late because of contention.

The performance loss of functions or threads can be easily accumulated from this detailed instruction-level contention information. This is important in the real-time field especially for specific real-time critical functions. When known, the program flash

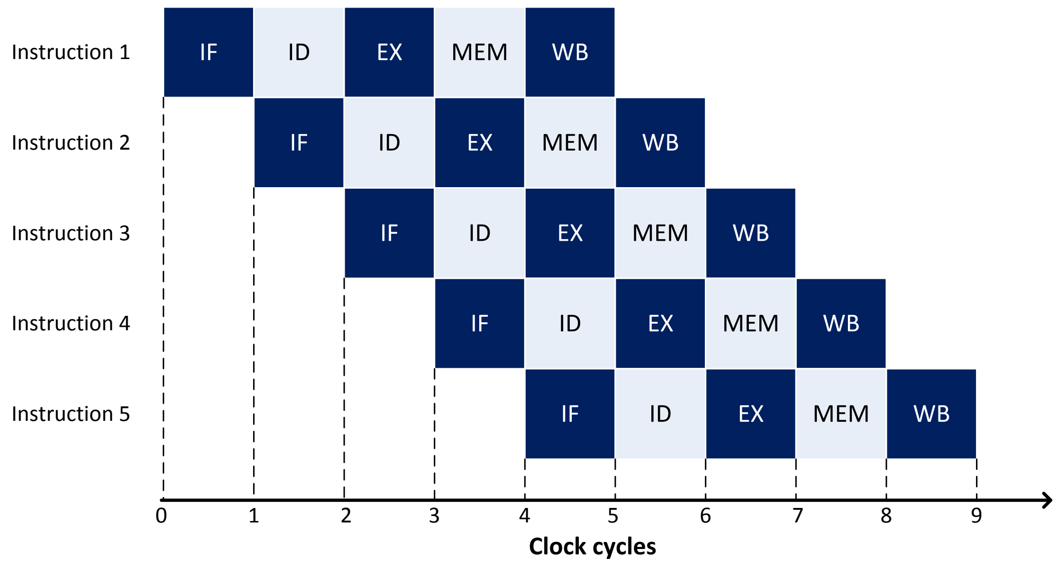


Figure 4.18: Basic pipeline stages:(IF = Instruction Fetch, ID = Instruction Decode, EX = Execute, MEM = Memory access, WB = Register write back) [126]

contention can be reduced by adopting appropriate measures. For instance, the program flash contention can be avoided by instruction relocation to a different flash bank if applicable. Another solution can be copying the instructions from the flash to program RAM, which can be accessed faster. Furthermore, this information can also act as the input to contention-aware schedulers [19] which ensure that congested functions are executed at different points in time.

This methodology has three contributions:

- As far as known, the detection and analysis of program flash contention are currently missing in the existing tools. This is the first approach to detect program flash contention based on the non-intrusive trace and also estimate the performance impact of program flash contention. The analysis results facilitate the performance optimization for multi-core embedded systems.
- The proposed approach is demonstrated to have high contention detection rate and accurate performance impact estimation, which gives users a direct impression of how bad the contention is.
- It provides an example of how to make use of non-intrusive tracing to analyze system issues automatically for commercial off-the-shelf embedded systems.

4.3.4.2 Methodology

Modern core architectures make use of pipelines to gain fast CPU throughput. An instruction is split up into a sequence of steps and these steps are executed in the pipeline. In this way, several instructions can be executed concurrently. The Figure 4.18 illustrates a basic pipeline diagram. Usually the first step is to fetch the instruction and then execute it in the following stages. If one stage halts, the whole pipeline may stall and require longer time, resulting in lower performance. There are many causes e.g. branches, hazards which can lead to pipeline halt. Instructions are usually fed to the pipeline at the beginning pipeline stage. This is guaranteed by a hardware module called Pre-fetch Unit (PFU) which pre-fetches instructions from either flash or cache. A pipeline cannot run smoothly without continuous instruction feeding.

Ideally, a core should always have instructions running in its pipeline and never wait for the instruction fetching. However, in reality it has to halt its pipeline due to many factors and one of them is flash contention. This phenomenon is called *Delayed Instruction Fetch (DIF)* [47]. In brief, the basic idea of this methodology is deriving subtle program flash contention based on the detectable *DIFs*.

The proposed methodology is introduced in detail in these following three parts. These three parts namely input, analysis and output are organized according to methodology phases. The first input part describes the input of the methodology, which is hardware trace data. The second part introduces the different factors leading to *DIFs* and the way to derive program flash contention. The last part shows the analysis results including contention spots and performance impact due to contention.

4.3.4.2.1 Input: trace data acquired by hardware tracing

This methodology makes use of hardware tracing, which consists of program flow and bus transfer information. A simplified block diagram of an Infineon AURIX SoC is shown in Figure 4.19 to explain the contention spots and tracing points. Program flash contention happens at the interface of the program flash when more than one core accesses the program flash. From the hardware tracing, program flow information TABLE 4.7 and bus transfer information TABLE 4.8 are obtained by the tracing hardware. The time stamps (***TimeR***) indicate the time information when an operation finishes. The beginning time of an operation is usually not included in the tracing so the duration of an operation is not available in tracing. The executed specific instructions are decoded when the binary information is given, e.g. an .elf file. ***Address*** column contains instruction pointer address, which is also the program flash address that stores instructions. The bus transfer tracing scope is limited to the program flash.

Table 4.7: An instruction trace

Index	***TimeR***	***Opoint***	***Origin***	***Address***	***Instruction***
0	19	CPU0	CPU0	80000A48	MOVH.A A15,xF0030000
1	20	CPU0	CPU0	80000A4C	LEA A15, [A15], 0x6100
2	20	CPU0	CPU0	80000A50	LD.W D15, [A15], 0x0
3	25	CPU0	CPU0	80000A52	MOVH D2, 0x70020000

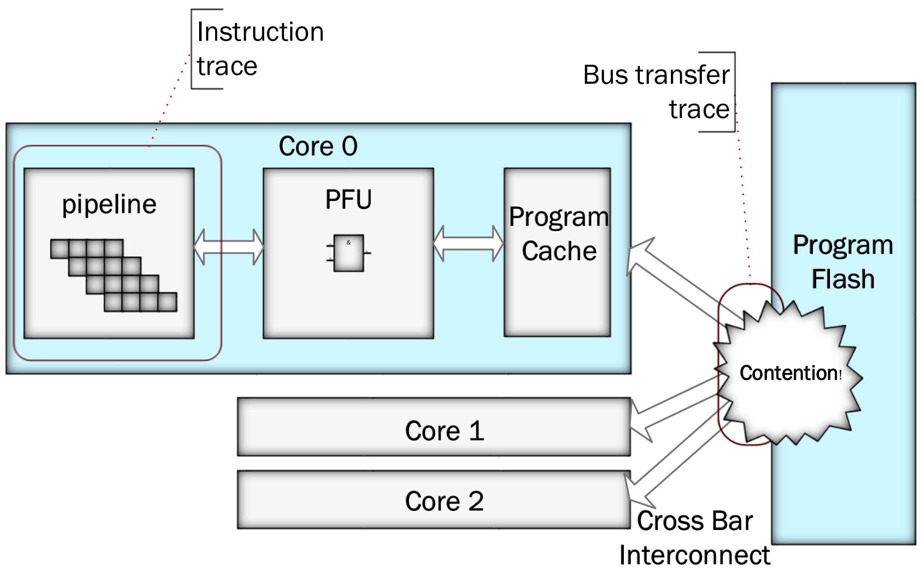


Figure 4.19: Example: An instruction trace and a bus transfer trace in AURIX TC29x

Table 4.8: A bus transfer trace

Index	*TimeR*	*Opoint*	*Origin*	*Address*
0	59516	PF0	CPU0.PMI	800006A0
1	59517	PF0	CPU0.PMI	800006A8
2	59518	PF0	CPU0.PMI	800006B0
3	59519	PF0	CPU0.PMI	800006B8
4	59526	PF0	CPU1.PMI	80002AE8

4.3.4.2.2 Analysis: contention symptoms

The obtained trace data only has the ending time of many operations and no contention information is given directly by the hardware tracing. It is also not straightforward to derive the contention which impacts the system performance. In this part, the question how to find program flash contention based on the trace data is answered.

Causal Delayed Instruction Fetch: Not all the program flash contention has an influence on the core's pipeline and the core performance. A PFU is normally implemented in a core to ensure a core pipeline's continuous execution. PFU inside a core prefetches the instruction from the memory. This prefetch operation could be much earlier in advance than necessary. In this case, the core pipeline has a large tolerance of program flash contention. This is because an instruction can still arrive at the pipeline in time before causing the pipeline to halt in spite of slight program flash contention. Such cases are not considered since the methodology is only targeted at the program flash contention degrading the performance.

Table 4.9: A CDIF example

Index	*TimeR*	*Opoint*	*Origin*	*Address*	*Instruction*
1	59526	PF0	CPU0.PMI	80000A40	-
2	59527	PF0	CPU0.PMI	80000A48	-
3	59528	PF0	CPU0.PMI	80000A50	-
4	59529	PF0	CPU0.PMI	80000A58	-
5	59532	CPU0	CPU0	80000A40	MOVH.A A15,xF0030000
6	59533	CPU0	CPU0	80000A44	LEA A15, [A15], 0x6100
7	59533	CPU0	CPU0	80000A58	LD.W D15, [A15], 0x0
8	59538	CPU0	CPU0	80000A5A	MOVH D2, 0x70020000

For the performance-degrading *DIFs*, a new term *Causal DIF (CDIF)* is introduced, which means that a core finishes the execution of the current instruction and waits for the next instruction fetch. As explained before, this should not exist in an ideal case but in reality it can be detected in hardware tracing. *CDIFs* can be detected by recognizing a special pattern that is an instruction fetch closely followed by the execution of the fetched instruction. For example, an instruction fetch operation (yellow rows 1 to 4) happens just before an instruction execution (green row 5) without another instruction executed in between in TABLE 4.9. The time offset between the executed instruction and the fetch is relying on the specific design of PFU and CPU pipeline. Here the threshold ($t_{threshold}$) to recognize a *CDIF* is defined as the time offset in a scenario when the CPU runs out of instructions and fetches instructions from memory. If the time offset in the trace is equal or smaller than $t_{threshold}$, the fetch operation is treated as a *CDIF*, as in Equation 4.14. The t_{instr} and t_{fetch} are the time points recorded by the tracing hardware. According to the above definition, *CDIFs* are picked out.

$$CDIF : t_{instr} - t_{fetch} \leq t_{threshold} \tag{4.14}$$

The target group of this methodology is *Program Flash Contention (PFC)* that influences the CPU performance. There are many factors leading to *CDIFs* including *PFC* and *Other Delay Factor (ODF)s*. In order to derive *PFC*, *ODFs* are introduced.

Other delay factors: There are various factors that can cause *CDIF*s including wrong branch predictions by PFU, reaching the maximum throughput limit of the program flash and instructions that cannot be predicted by PFU. To guarantee the continuous pipeline execution, the instructions are fetched in advance, so a branch prediction is needed in the PFU. Accordingly, the prediction result can be either a correct branch prediction or a wrong branch prediction. For a correct prediction, a core is supposed to run without waiting for an instruction fetch. On the contrary, a wrong branch prediction will result in a wrong instruction fetch and a core idle state waiting for the instruction fetch, which could be a *CDIF*.

The *Wrong Branch Prediction (WBP)* has symptoms and can be detected in trace data. There are two different ways to recognize *WBP* spots. First, for PFUs with static branch prediction algorithm that is the simplest prediction algorithm, the prediction merely on the branch instruction itself. A prediction whether it is right or wrong can be calculated by comparing the prediction result to the real execution result. The prediction algorithm is needed as the input of the methodology. Second, for PFUs with other branch prediction algorithms, the *WBP* spots are detected by comparing the program execution flow and the instruction fetch sequence. The comparison depends on the specific design of CPU and PFU. An easy method can be checking whether the fetched instructions are executed in the next certain number of instructions. For instance, an instruction fetch happens in a bus transfer tracing and the fetched instruction is not executed in the next for example 20 instructions. In this case, this branch prediction causing the instruction fetch is considered to be wrong. By either of two ways, *WBP* spots can be collected. In the implemented methodology, the second way is adopted.

Another factor leading to *CDIF*s is program flash *Overload (OL)*. Usually the designed program flash throughput satisfies the requirements of continuous accessing. However, the flash is accessed block by block. The size of a block is normally large and they are aligned, e.g. 32 bytes in AURIX TC29. If many jump instructions happen in a short time, the maximum designed throughput can be reached even for a single core application. In this case, *CDIF*s happens without program flash contention. The *OL* spots have a straightforward symptom, which is the time interval between two program flash accesses equal to the minimum interval limited by the hardware design. The minimum interval also depends on the application configuration related to the flash module.

The last factor is named *Unpredictable (UP)*, meaning the branch instructions that cannot be predicted by the PFU. This group strongly relies on the CPU architecture. For AURIX, some instructions e.g. indirect branch instruction, in which the jump destination address depends on the register value, are listed as the *UP*. Interrupts usually cannot be predicted either. *UP*s cause *CDIF*s in a similar way as *WBP*s. The consequence of the unpredictable instructions is that the next instructions may have to be fetched after the execution of the unpredictable instructions

The set *ODF* includes all the other delay factors including *WBP*, *OL* and *UP* as shown in Equation 4.15.

$$ODF = WBP \cup OL \cup UP \tag{4.15}$$

Contention: the factors described above can overlap. For this situation, it is analyzed by comparing to the sole situation in the system. For example, an instruction fetch after an unpredictable branch takes much longer than a normal contention-free fetch, which is a contention indicator. This group is denoted by *Contention and Others (CO)*.

Finally, after analyzing all these factors, the *PFC* is the intersection between set *CDIF* and complement set *ODF*, and then union the set *CO*, as shown in Equation 4.16.

$$PFC = (CDIF \cap \overline{ODF}) \cup CO \tag{4.16}$$

In the implementation, pre-selection can be conducted to improve the methodology performance. The pre-selection here means only the interleaved program flash accesses from different cores have the potential to have contention so that only these spots are considered in the methodology.

4.3.4.2.3 Output: Contended instructions and performance impact

The proposed methodology outputs both contention spots information and performance impact information.

Contended instructions: after the analysis, *PFC* spots are calculated and picked out. The affected instructions are defined as contended instructions, which are delayed by *PFC*. Based on this information, functions with contention spots are also obtained when the symbolic information and range information are provided by e.g. an elf. file.

Performance impact: the performance impact is defined as the extra clock cycles added by program flash contention. It is estimated by comparing the contention case to an ideal contention-free situation. The extra clock cycles are estimated for each contended instruction. Then, performance impact at the function level or thread level is calculated. With the help of this information, the program flash contention can be reduced for a specific function or a specific thread by developers.

Others: the contended functions are of interest for many users as they can make use this information to avoid the conflicts by allocating a function to a different location. For instance, a function can be allocated to a different flash bank or scratch pad memory. If task/thread information is also provided, then a task/thread-level analysis can be performed, which provides the contended task/thread information. Then the usage of the tasks or threads can be optimized.

4.3.5 Data memory analysis

4.3.5.1 Introduction

The data memory performance has been the bottleneck of modern multi-core embedded systems for quite a while. In order to break this barrier, NUMA architectures with scratchpad memory become popular. In a typical NUMA architecture each core has its own local memory, allowing fast accesses from the core to the local memory. However, new issues as side effects are also introduced. The local memory accesses consume less time but the remote memory accesses take longer. Data allocation determining the local and remote accesses should be considered and optimized carefully.

Another bottleneck of multi-core systems is shared resource contention. When a core accesses a shared resource occupied by another core, it has to wait until the previous one finishes. Data memory, as a frequently accessed shared resource, is vulnerable to contention. Data memory contention is supposed to be detected and measured. Compared to remote accesses, memory contention effect has no obvious symptoms and cannot be detected directly by observing the memory accesses in commercial embedded systems. An indicator e.g. data cache miss rate is needed to show the effect indirectly [19].

The performance penalties incurred by both data locality and memory contention must be analyzed before performance optimization [48]. This is because data locality and contention are mutually dependent. The study conducted by Blagodurov et al. [100] shows that the-state-of-the-art contention management algorithms fail to be effective in NUMA systems and may even degrade performance compared to a default OS scheduler. In the experiment, the data allocation was optimized to avoid data memory contention, which, however, led to increased number of remote accesses. On the contrary, Majo et al. [103] concluded that maximizing data locality does not always minimize execution time because of memory contention. It may be beneficial for performance to allocate data to remote memory to avoid memory contention. Merely detecting the occurrence of remote accesses or memory contention is not sufficient. The performance penalties by both effects should be quantified and be compared to provide a reference for performance optimization, which has to be kept in balance for both data locality and memory contention effects to achieve the optimal performance. Therefore, a novel memory access analysis approach is proposed to evaluate data locality penalties and memory contention penalties in a quantitative, comparable way.

The proposed approach has two contributions:

- To the best of my knowledge, the proposed method is the first non-intrusive post-processing analysis method estimating both NUMA data locality and memory contention penalties.

- A new memory contention indicator is proposed to show the memory contention penalties quantitatively, which is comparable to data locality penalties, providing a reference for performance optimization. The experiment section also indicates accurate penalty estimation.

4.3.5.2 Methodology

The proposed methodology provides a generic flow to estimate extra memory access latencies due to both data locality and memory contention based on hardware trace data. In the following, some basic terms and assumptions for the methodology are defined first. Then the way to measure data locality penalty is described. After that, a model to detect memory contention spots and estimate memory contention penalty is introduced. Finally, the both memory contention and data locality are considered together.

4.3.5.2.1 Prerequisite terms

- **Consumer (C):** a memory consumer, which accesses a memory [48]. E.g. CPUs
- **Provider (P):** a provider is identical with a memory. A provider has a name, start address and a size.
- **Variable (V):** a variable stored in a provider and accessed by a consumer. It has a name, address and size.

As the input for this methodology, a memory access MA is defined as follows:

$$MA = (time, source, address, mode) \tag{4.17}$$

A memory access contains the hardware-level information including time, source, address and mode. Time stamps, which indicate the ending time of an access, are stored in the time information. Source shows the master of a memory access and address tells the address of the memory access. The mode information is used to distinguish the write operations from read operations. The access width is also contained in the mode information. Every access can be mapped to a provider and a variable in the application using the address of the access. It should be noted that the memory access does not contain the starting time, which is usually the case with COTS embedded systems.

For this methodology, we need to make several assumptions [48]. (i) the arbitration does not need additional time. (ii) the memory controller does not have a request buffer. (iii) the memory providers have a static response time (from request until data available). (iv) the memory is globally addressable. (v) the memory module is only capable of handling one access request at a time. The assumptions above are proposed based on ordinary embedded systems e.g. Infineon's AURIX.

4.3.5.2.2 Data locality

To distinguish local memory accesses from remote memory accesses, the access source information has to be compared to the access target information. Given the hardware model, the target of a memory access is calculated using the address.

Knowing which accesses are remote accesses, the *Locality Penalty (LP)*, which is defined as the extra time added by remote accesses, is calculated according to Equation 4.18. In Equation 4.18, the function $L(a \in MA)$ returns the expected additional latency for a given memory access a compared to local access. The latency mainly from interconnect so it is usually static depending on the locations of both consumer and provider. For each hardware platform, this information can be organized as a look-up table.

$$LP := \sum_{a \in MA} L(a) \tag{4.18}$$

4.3.5.2.3 Memory contention

As discussed above, the time stamp indicates the ending time of an operation. Whether an operation is delayed by contention cannot be observed directly. Therefore, a contention indicator based on a memory contention model is created. The memory model is described in the following.

Merely relying on the time stamps cannot determine if contention happens because a contention-free memory access may have the same access pattern as a contention memory access. Figure 4.20 demonstrates this case and introduces the basic of the designed contention indicator. When memory accesses are investigated, the INT between two traced memory accesses to the same memory provider can be observed. If two memory accesses a and a' are issued to the provider interface at the same time (Figure 4.20.a) and the provider can only handle one request at a time, a' has to wait, then a' is delayed by exactly INT clock cycles. INT is assumed to be 3 in this case, so a' finishes at the $a_{endTime} + 3$ clock cycle. In the second example (Figure 4.20.b) a' is issued INT cycles after a. The ending time of a' is INT cycles after the ending time of a, which is exactly the same as the first scenario (Figure 4.20.a). Therefore, two cases cannot be distinguished. However, the second case is contention-free while the first case is with contention. In order to link this hardware-level information to memory contention, a statistical method is introduced to design a contention indicator.

Several different memory access possibilities may result in the identical access pattern from the trace data perspective. A new assumption (vi) is made here: the probability of all memory access possibilities for the identical access pattern is the same, meaning that the probability of the arriving time of a memory access is uniformly distributed in the small time window $(INT + 1)$. The probability that a potential contention is a real contention can be calculated in (4.19).

$$PoC_{wp} = \frac{INT}{INT + 1} \tag{4.19}$$

Besides the probability of a contention, the expected additional cycles for the delayed access a' compared to a contention-free case is also of importance [48]. In the previous example, 3 delay cycles can be observed when both accesses are issued at the same time (Figure 4.20.a) and 0 delay cycles when a' is issued 3 cycles after a (Figure 4.20.b). The rest can be done in the same manner. 2 delay cycles is observed if a' is issued 1 clock

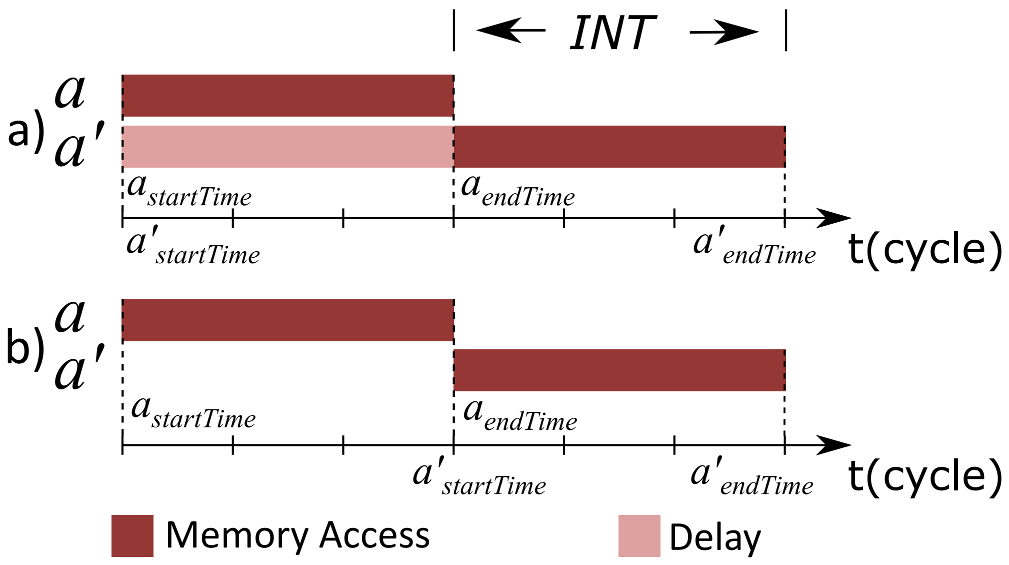


Figure 4.20: a) shows a contention sequence; b) shows a contention-free sequence

cycle after a and 1 delay cycle if a' is issued 2 cycles after the access a. Considering all the possibilities, the contention penalties (CP) can be defined as follows:

$$CP_{wp} = \sum_{i=0}^{INT} \frac{1}{INT+1} \times i \tag{4.20}$$

The example shown in (Figure 4.20) has the identical memory access pattern. All of the possibilities for this pattern has expected penalties ($\frac{1}{4} \times 0 + \frac{1}{4} \times 1 + \frac{1}{4} \times 2 + \frac{1}{4} \times 3 = 1.5$) according to Equation 4.20.

The above example shows the case that a has priority over a' as a is first handled when two come at the same time. In a different scenario, if a has no priority, then a' will be serviced first when they are simultaneously, which is not same as the depicted pattern. For the depicted pattern, the possibility that they are arriving simultaneously can be eliminated for the case that a has no priority. Accordingly, the probability of contention and contention penalties equations with subscript $_p$ are defined for contention with priority as follows:

$$PoC_p = \frac{INT - 1}{INT} \tag{4.21}$$

$$CP_p = \sum_{i=0}^{INT-1} \frac{1}{INT} \times i \tag{4.22}$$

For dynamic priority assignment techniques (e.g. round-robin) noted by $_d$, a weighted equation is applied, shown in (Equation 4.23) and (Equation 4.24). w is the probability of access without priority.

$$PoC_d = w \times PoC_{wp} + (1 - w) \times PoC_p \tag{4.23}$$

$$CP_d = w \times CP_{wp} + (1 - w) \times CP_p \tag{4.24}$$

4.3.5.2.4 Sum penalties

Both locality penalties and contention penalties are supposed to be analyzed in a quantitative and comparable way. They are both calculated in cycle so the sum penalties that are defined as the penalties caused by both effects are simply derived by adding them up according to Equation 4.25

$$P := LP + CP \tag{4.25}$$

In this way, sum penalties are considered for each memory access in the trace data. The total penalties to functions or threads or cores can be then analyzed, given the memory trace data. The optimization direction can be then well-adjusted according to the analysis results.

5 Implementation and experimental Evaluations

In the previous chapter, a methodology that automatically detects system issues is proposed. In this chapter, a software tool named ChipCoach implemented based on the proposed methodology is described. First, ChipCoach is introduced in detail. Then, experimental evaluations and case studies using ChipCoach are described to show the feasibility and the effectiveness of the proposed methodology.

5.1 The implementation of the proposed methodology — ChipCoach

The implemented software tool named ChipCoach is a post-processing tool for embedded systems, which makes use of MCDS to gather trace data and diagnoses issues automatically from different perspectives.

5.1.1 ChipCoach

ChipCoach is a post-processing system diagnosis tool. It exploits the tracing and debugging functionalities of the MCDS. The main purposes of using ChipCoach include:

- Automated hardware configuration check: ChipCoach is designed to monitor the configuration process of hardware modules and report the detected violations of rules, which are predefined by the SoC designer and stored in ChipCoach.
- Performance profiling & bottleneck analysis: ChipCoach targets at the automated detection of several types of performance issues including shared resource contention, data locality issues, lock contention, blocked interrupts etc. These issues are usually hard-to-detect or even invisible to software developers.
- Specific module support: ChipCoach facilitates the usage of several important hardware modules for instance, memory protection unit, DMA, clock control unit etc.
- System exploration: ChipCoach helps users to understand and explore their systems from different perspectives.

ChipCoach is programmed in Java and is based on Rich Client Platform (RCP). The hardware-related operations are handled by lower layers such as the MTV layer and the

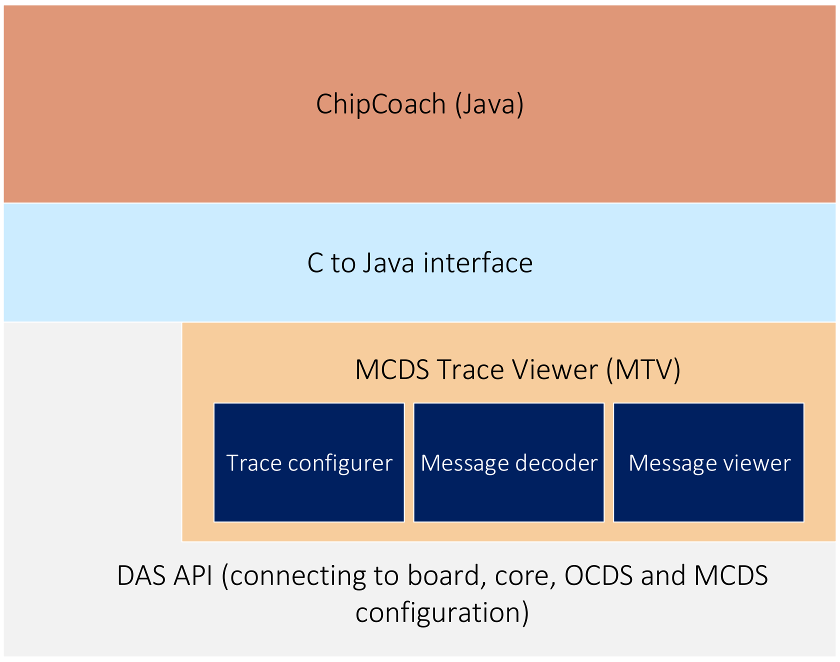


Figure 5.1: The architecture of ChipCoach

MCD API layer Figure 5.1. The MTV layer is responsible for MCDS configuration, tracing and decoding. This layer has the most features provided by MTV and the details of this layer are described in the previous chapter 3. The DAS API layer deals with the device connection and device-related operations. It is programmed in C/C++ and provided by Infineon.

In order to call APIs in the lower layer from ChipCoach, Java Native Interface (JNI) is applied as an interface between Java and C/C++. The lower layers including the MTV layer and the MCD API layer are programmed in Microsoft Visual C++. MTV provides tracing support for ChipCoach. The MCDS configurations are stored in the .mcdsc file that contains a list of MCDS register information including addresses and values. These files are the input of MTV. More specific MCDS configuration files could be created based on these template MCDS configuration files. The APIs between the Java project and the C++ project are first defined in the Java project and then a C header file that will be included in the C++ project is generated with JNI. Finally, a .dll file that will be used directly by ChipCoach is generated within the C++ project.

ChipCoach is designed to be as easy as possible to apply. It works on a normal computer with Windows installed. The connection to the device is also supposed to be simple and easy as shown in Figure 5.2. It is via DAS server [112]. DAS server supports both DAP and JTAG connections.

As can be seen from Figure 5.2, ChipCoach uses only the on-chip tracing feature provided by the AURIX device instead of using simulation. No expensive hardware

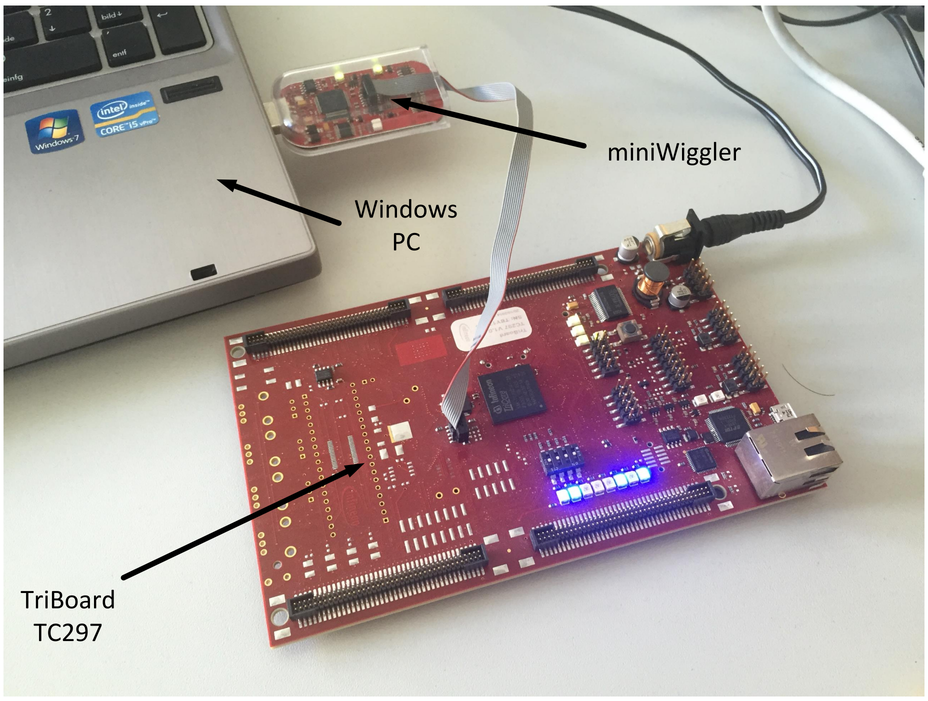


Figure 5.2: An AURIX device is connected to ChipCoach via miniWiggler

boxes are needed to temporarily store the trace data. The miniWiggler acts only as a converter between DAP and USB as described in the previous Chapter 3. The cost of the debug solution using ChipCoach is low, compared to commercial debuggers.

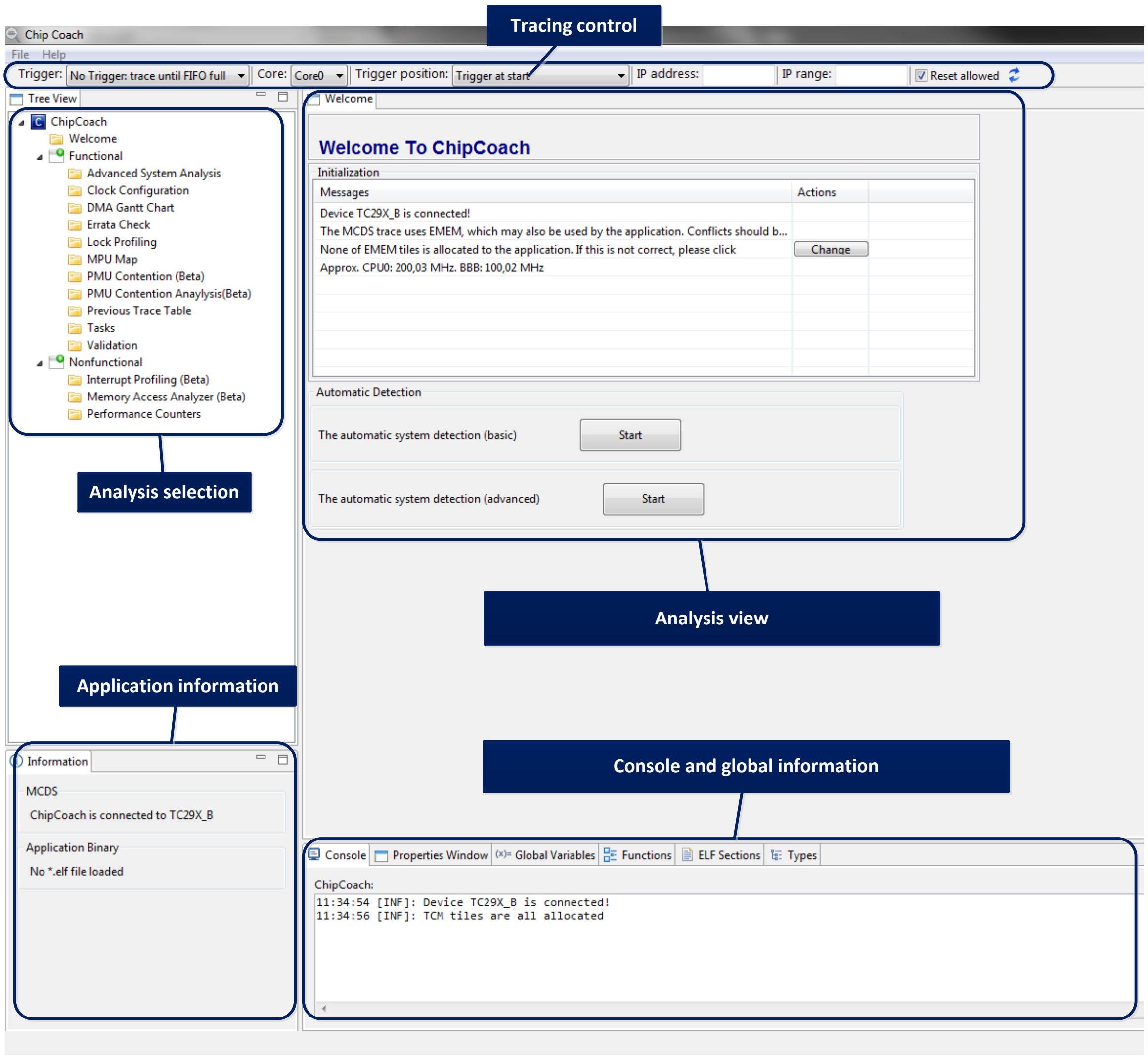
Figure 5.3 shows a screenshot of ChipCoach. Different analyses can be triggered by clicking the corresponding items in the tree view. The triggered analysis will then be displayed in the middle named analysis view as a tab. For analysis with tracing, the tracing control can be applied to configure the tracing scope and trace triggers, allowing flexible analysis. Information related to devices, .elf file and messages are shown in the bottom views.

In the following, the functions that are globally used by other features are described first. Then, all feasible features are individually introduced. Many of the features are implemented based on the methodology in Chapter 4. Moreover, several practical features that are not described in Chapter 4 are also provided and elaborated.

5.1.1.1 Global functions

5.1.1.1.1 Device-related information

The device-related information is needed by ChipCoach, as for example SRC indexes, register information changes for different connected devices. In ChipCoach, this information is handled with the same strategy: integrating this information directly into ChipCoach source code. This is because the strategy is faster and more efficient com-



Trigger: No Trigger: trace until FIFO full | Core: Core0 | Trigger position: Trigger at start | IP address: | IP range: | ☑ Reset allowed

ChipCoach
- Welcome
- Functional
  - Advanced System Analysis
  - Clock Configuration
  - DMA Gantt Chart
  - Errata Check
  - Lock Profiling
  - MPU Map
  - PMU Contention (Beta)
  - PMU Contention Anaylysis(Beta)
  - Previous Trace Table
  - Tasks
  - Validation
- Nonfunctional
  - Interrupt Profiling (Beta)
  - Memory Access Analyzer (Beta)
  - Performance Counters

Welcome To ChipCoach

Initialization

Messages	Actions
Device TC29X_B is connected!	
The MCDS trace uses EMEM, which may also be used by the application. Conflicts should b...	
None of EMEM tiles is allocated to the application. If this is not correct, please click	Change
Approx. CPU0: 200,03 MHz. BBB: 100,02 MHz	

Automatic Detection

The automatic system detection (basic) Start

The automatic system detection (advanced) Start

MCDS

ChipCoach is connected to TC29X_B

Application Binary

No *.elf file loaded

ChipCoach:
```
11:34:54 [INF]: Device TC29X_B is connected!
11:34:56 [INF]: TCM tiles are all allocated
```

Figure 5.3: The screenshot of ChipCoach

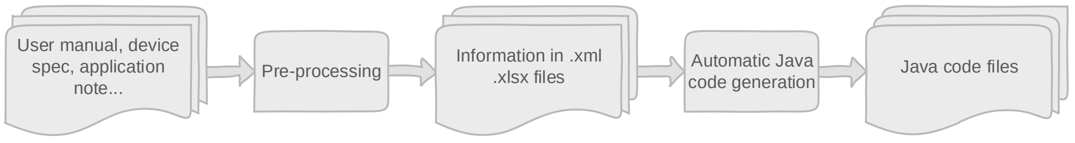

User manual, device spec, application note...
Pre-processing
Information in .xml .xlsx files
Automatic Java code generation
Java code files

Figure 5.4: The generation flow of the device-related information

pared to processing this information during runtime. Moreover, it is feasible for a small number of supported devices.

The device-related information contained in the user manual, device specification and other files as shown in Figure 5.4. These files are reprocessed first to transform them into files that are easier for machine processing. In the ChipCoach project, they are converted into .xml and .xlsx files. Then, an automatic code generator that is programmed also in Java is applied to generate Java source code. For each device, a corresponding class is created and can be dynamically loaded during runtime depending on the connected device.

5.1.1.1.2 Debugging information

Debugging information is necessary to make the debugging process feasible and efficient. It is usually generated together with the binary by compilers. It indicates the relation between the executable program and the original source code. The debugging information is encoded into a pre-defined format which is debugging with attributed record format (DWARF).

DWARF is a widely used, standardized debugging data format [127]. It was designed along with Executable and Linkable Format (ELF). A lightweight DWARF parsing algorithm was already implemented which is public and is used to read function information. For the memory access analysis more DWARF information is needed, so the parsing algorithm is extended for ChipCoach. The details of the parsing algorithm and DWARF are not described in this dissertation.

With the help of the DWARF parser in ChipCoach, various kinds of information contained in the .elf file is extracted, for instance, function information including the function symbol, the address range, the function arguments and even the function return type, variable information including the structure, the symbol, the location and the type. The DWARF parser is globally accessible and used by all features in ChipCoach. It significantly facilitates the mapping between addresses and symbols.

5.1.1.1.3 ChipCoach tracing

Most analyses in ChipCoach apply tracing to detect issues. As the most commonly used function, the tracing function is also globally accessible similar to the DWARF parser. Its main tasks are tracing configuration, trace data processing and lower layer handling.

The tracing function provides a type of tracing for each analysis. The tracing type implicitly defines the interesting tracing scope e.g. bus transfers, program flow or data accesses. Several function arguments further specify the detailed tracing configuration. For example, the hardware configuration validation needs the trace of register accesses. Therefore, the corresponding trace type covers register accesses. During the analysis, which hardware module is supposed to be validated can be further specified via arguments.

5.1.1.1.4 Tracing control

MCDS allows various triggers to be configured according to different scenarios. A trigger is a sequence of specific operations or even a specific operation that fulfill the pre-defined requirements. It could be an instruction executed at a specific address, a bus transfer, a memory access to a pre-defined address or a counter reaching its limit. The detailed trigger configuration is available in the manual [110]. The configuration of a trigger is flexible. With the help of a trigger, the starting point or the ending point of tracing can be controlled. A trace can start when a trigger is met until the trace buffer is full. It can also start immediately and always overwrite the trace buffer which is a ring buffer until the trigger is fulfilled. The bandwidth of the trace data generation can be further reduced by applying tracing qualification that defines which kind of message is qualified to be stored.

In ChipCoach, the tracing control is available at the top of the view as shown in Figure 5.3. Users can define their own triggers to only focus on specific parts of programs, increasing the efficiency of the analysis. The tracing control is globally effective to all analyses in ChipCoach.

5.1.1.2 automated system diagnosis

The welcome view is the first view available after launch. It tells the basic facts about the connected systems. When a device is connected and ChipCoach is being launched, ChipCoach detects the device's ID and also measures the basic information about the device for example clock frequencies as shown in Figure 5.3. EMEM usage is also verified in order to avoid collisions between MCDS tracing and the application usage. Some applications use EMEM for calibration and large data storage, for example ADAS applications. The EMEM can be split into two parts, one for tracing and the other one for the application. The allocated tracing EMEM capacity determines the size of tracing, which is important especially for features such as profiling and performance analysis.

One advantage of using ChipCoach is that it also helps developers to figure out what is going wrong even when they have no idea where to look at. It is realized with the system diagnosis function as shown in the lower part of the welcome view. There are two types of system diagnoses namely basic system detection and advanced system detection, corresponding to the automated diagnosis with functional issues and non-functional issues described in Chapter 4. The purpose of placing these two diagnoses here is to explore the system in just one click. After the connection, a “blood test” can be performed to check the health status of the system, by clicking the start button. A sequence of analyses covering various types of potential issues will be run automatically one by one. A report is generated to show the system status from different perspectives, which provides a convenient way for users to understand their systems better.

The system diagnoses are collections of different analyses. The generated report is a summary of results from each analysis. A potential issue can then be found in this report and the individual analysis can be applied.

5.1.1.3 Hardware validation analysis

In ChipCoach, the hardware validation flow is not fully implemented. The rules are currently hard-coded instead of storing in the database. However, the essential idea is identical.

To facilitate the rule definition, a regular expression is applied to register names. This helps the rule definition with groups of registers. A group of registers with similar names may have an identical rule. In the rule definition, the regular expression in the rule indicates a group of registers instead of defining the similar rules repeatedly.

For the value argument defined in the register access (Equation (4.8)), it is not limited to a single value. More flexible inputs are allowed. In the current implementation, several types are created.

- $Default$: A single value.
- Not: A single value that is not allowed.
- $List$: A white list that contains the allowed values.
- Not_list: A black list that contains the disallowed values.

The motivational example from Section 4.2.1 about the `UP` issue could be checked by the rule in Equation (5.1). The details of register accesses are specified in Equations (5.2)(5.3). These equations describe the rule that once one of the registers `CCUCON0`, `CCUCON1` and `CCUCON5` is written, a set operation will eventually happen to the `UP` field in one of these registers.

$$\varphi = \Box((R_0 \vee R_1 \vee R_5) \Rightarrow (\Diamond\ (R'_0 \vee R'_1 \vee R'_5))) \tag{5.1}$$

with $i = 0,\ 1,\ 5$.

$$R_i = f(write, CCUCONi) \tag{5.2}$$

$$R'_i = f(write, CCUCONi, UP, [1]) \tag{5.3}$$

Another typical scenario is a sequence of accesses to the same register field. For example, an OCDS exists in the AURIX device. The OCDS is responsible for debugging and calibration. It is usually disabled due to safety reasons. The enabling of OCDS is protected by a password pattern that is a sequence of write accesses with predefined different values to the field `PAT` in the register `OEC`. The sequence of the write accesses must be exactly the same as specified. A rule could be defined to check the sanity.

A sequence of numbers "0", "1", "2" is assumed to be the password pattern.

$$\varphi = \Box(R \Rightarrow (\Diamond R' \land (\neg R'' U R') \land \Box(R' \Rightarrow (\Diamond R''' \land (\neg R'''' U R'''))))) \tag{5.4}$$

with

$$R = f(write, OEC, PAT, [0]) \tag{5.5}$$

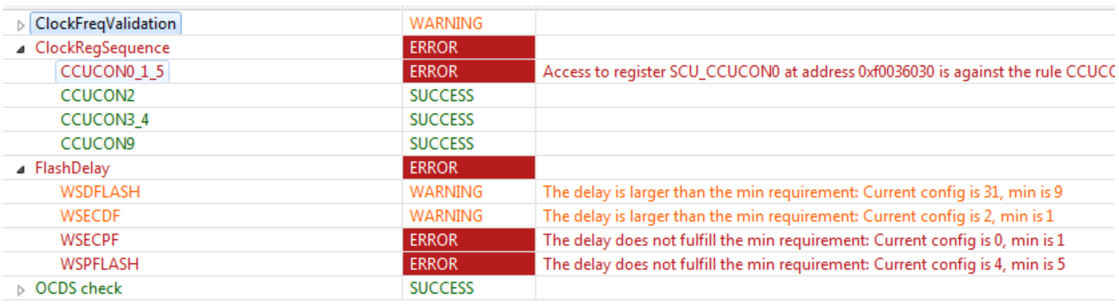
$$R' = f(write, OEC, PAT, [1]) \tag{5.6}$$

$$R'' = f(write, OEC, PAT, [Not\ 1]) \tag{5.7}$$

$$R''' = f(write, OEC, PAT, [2]) \tag{5.8}$$

$$R'''' = f(write, OEC, PAT, [Not\ 2]) \tag{5.9}$$

As can be seen, the rule definition is quite flexible and it covers a variety of cases.



▷ ClockFreqValidation	WARNING	
◢ ClockRegSequence	ERROR	
CCUCON0_1_5	ERROR	Access to register SCU_CCUCON0 at address 0xf0036030 is against the rule CCUCC
CCUCON2	SUCCESS	
CCUCON3_4	SUCCESS	
CCUCON9	SUCCESS	
◢ FlashDelay	ERROR	
WSDFLASH	WARNING	The delay is larger than the min requirement: Current config is 31, min is 9
WSECDF	WARNING	The delay is larger than the min requirement: Current config is 2, min is 1
WSECPF	ERROR	The delay does not fulfill the min requirement: Current config is 0, min is 1
WSPFLASH	ERROR	The delay does not fulfill the min requirement: Current config is 4, min is 5
▷ OCDS check	SUCCESS	

Figure 5.5: The screenshot of the hardware configuration validation view

For combinational issues, no tracing is needed. The register value is obtained by reading registers directly via debug interface e.g. JTAG after the configuration phase. For sequential issues, a trace is necessary to obtain the register access history. For each product generation and each device, the rules could differ. Therefore, only the valid rules for a device should be selected. For each device, a list of available rules is provided. Different rules are selected and validated depending on the connected device. For each check, the device could be restarted by the tool to trace complete access history starting from the beginning. ChipCoach searches through the trace from the beginning, e.g. a reset. A rule becomes valid when an activation condition is fulfilled. Then the searching direction becomes either forward or backward, depending on the rule definition, as shown in section 4.2.1.

5.1.1.3.1 Demonstration

Several rules are implemented in the analysis tool. A diagnosis report is generated as shown in Figure 5.5. The report is formed as a table, in which the check result that is either a successful message or an error message is displayed. Errors and warnings are highlighted and a short description of the problem is also given. The second category named "ClockRegSequence" contains the rules of the `UP` issues. The first error in this category shows that `CCUCON0` is modified but it is not updated. The next category is named "FlashDelay", which is also described in the motivational examples. The errors mean that the minimum requirements are not fulfilled by the current configuration, while the warnings remind the developer that the performance is not optimal. The last category checks whether OCDS is accessed by the application software and whether it is enabled before accesses.

5.1.1.4 DMA activity analysis

The DMA activity analysis feature is meant to help users to explore what is really happening to the DMA channels in systems. DMA is very widely applied in embedded systems to transfer data between peripherals and memory. There are many DMA channels (128 channels in TC29x) and quite flexible configurations for different use cases. Compared to the program execution, which can be analyzed by setting break points and single stepping in conventional debuggers, DMA channel activities are difficult to be "frozen" by setting break points. Usually, users assume that DMA channels work as they are configured, which could not be the case. The designed feature enlightens the darkness between the reality and the assumptions.

As described in the previous section 4.3.1, DMA activities are traced via OTGB. The trace data is then visualized as a Gantt chart by ChipCoach as shown in Figure 5.6. The Gantt chart displays the starting time and the ending time of a channel's active state. All DMA channel activities during the tracing are shown without any need of configurations by the user.

This feature is not limited to DMA channel activities. The service latency, which is defined as the time from a service request arrives until the corresponding DMA channel is active, is traced also via OTGB and displayed in the Gantt chart. It benefits the understanding of the request load situation in the system. Due to the limitation of MCDS, four service requests can be traced in parallel. Thus, users have to decide which four requests are interesting and critical. Once decided, the target service requests are configured via writing the corresponding SRN indexes. In this view, a look up table is provided to facilitate the searching of the corresponding SRN indexes. The users just need to know the hardware module information and the corresponding SRN indexes are calculated by ChipCoach.

Moreover, the Gantt chart provides a possibility to show both program execution and DMA channels in the same view as shown in Figure 5.7. It has two options, including function tracing and flow tracing. In order to use this option, the .elf file that contains the symbolic and the binary information is supposed to be available. The details of difference between the flow tracing and the function tracing are discussed in Chapter 3. The function tracing supports longer tracing with fewer details than the flow tracing. A strategy can be applied: first, having a function trace to get the overview then investigating an interesting area with a flow trace.

Stack levels and also function duration are also calculated and visualized based on the

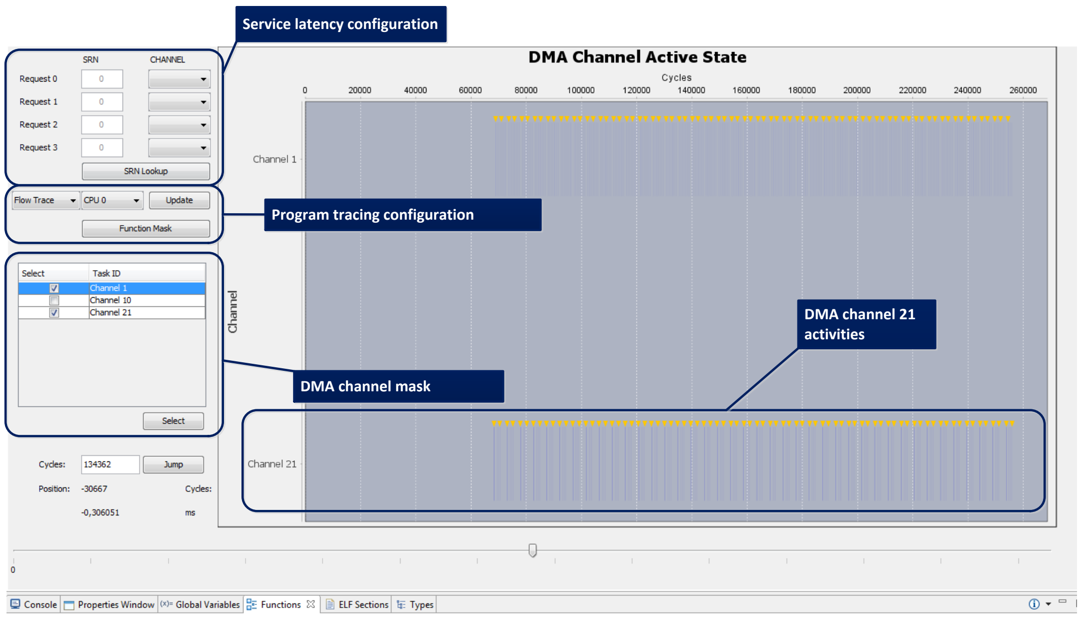


Figure 5.6: The screenshot of ChipCoach's DMA activity view

program trace. Stack levels are increased or decreased by detecting calls and returns. Different colors are applied to distinguish the stack depth. Combined with DMA channel activities, it becomes straightforward to notice the internal system execution.

5.1.1.5 Interrupt profiling

The interrupt profiling analysis is also implemented as a tree table 5.8. There are two tabs that are the summary statistics and the detailed trace data in this view. The summary statistics contain statistics defined in the Section 4.3.2 e.g. blocking time, contended interrupts, *SRRL* and *SRD*. The detailed trace data helps users to figure out what really happens.

Several options can be applied to trace the interrupt execution. A two-step strategy is advised. First, one CPU can be selected as the target service provider and all interrupts going to this CPU will be recorded as explained in TABLE 4.5. The arriving time of interrupts with this option is missing. Second, four additional interrupts that should be investigated deeply can be drawn into the view by using the SRN look up table. Then, more detailed information of these four interrupts is displayed.

In the tree table, the interrupt function name is defined as the first function executed after jumping out of the interrupt vector table. The service request name is obtained from the look up table that contains the mapping between the SRN indexes and SRN names. The frequency indicates how often interrupts occur. Additionally, minimum service time, maximum service time, average service time and deviation are calculated. A list of interrupts that preempt the current running interrupt is also given.

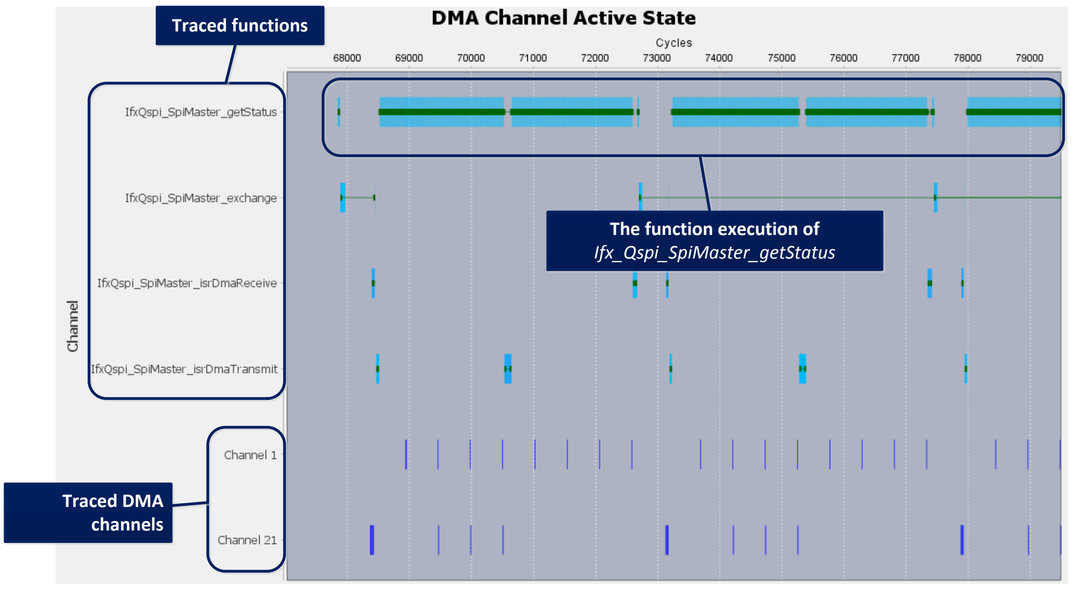
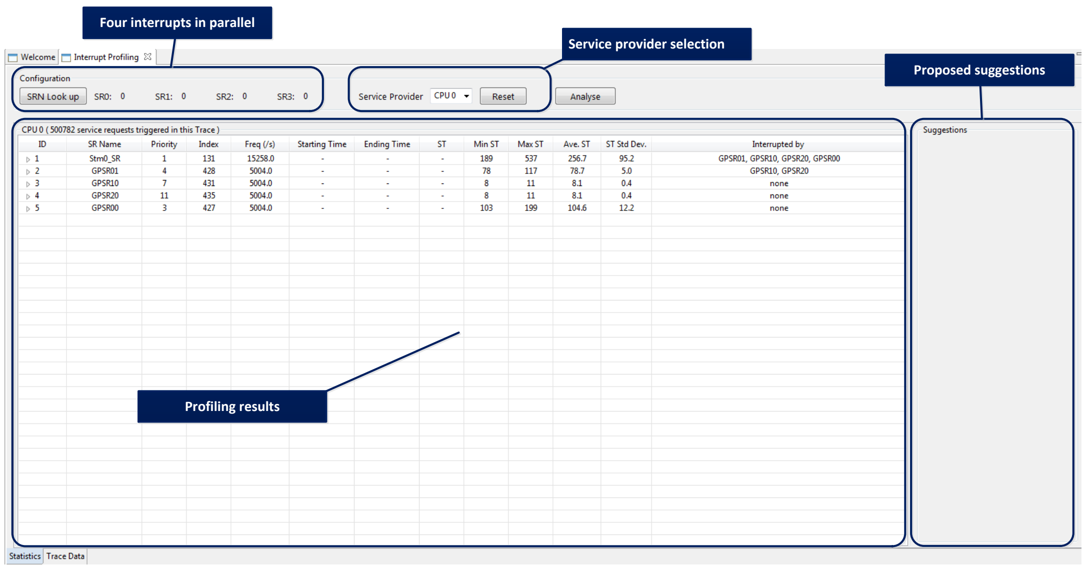

Traced functions
DMA Channel Active State
Cycles
68000
69000
70000
71000
72000
73000
74000
75000
76000
77000
78000
79000
Channel
IfxQspi_SpiMaster_getStatus
IfxQspi_SpiMaster_exchange
IfxQspi_SpiMaster_isrDmaReceive
IfxQspi_SpiMaster_isrDmaTransmit
The function execution of *Ifx_Qspi_SpiMaster_getStatus*
Channel 1
Channel 21
Traced DMA channels

Figure 5.7: The screenshot of flow tracing and DMA channel activities in ChipCoach's DMA activity view



Configuration

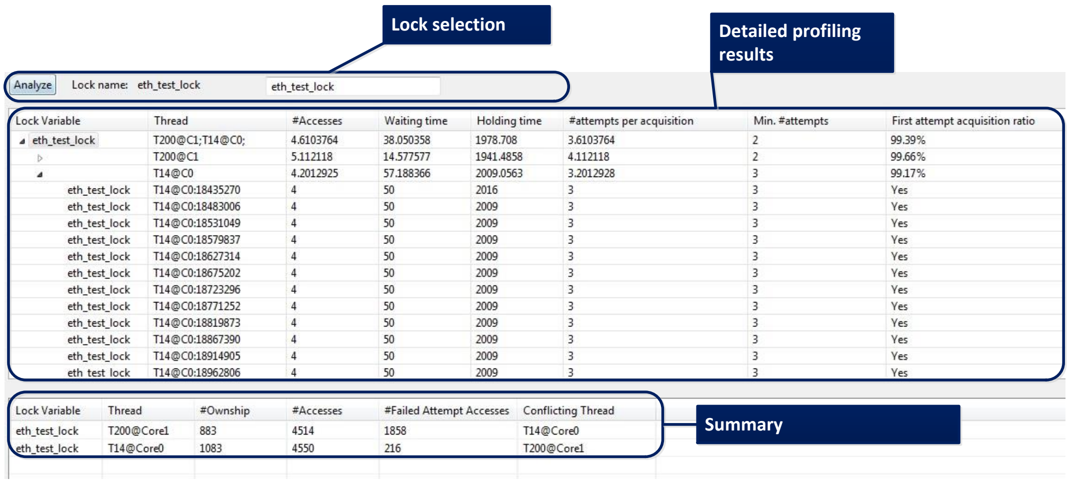
SR0: 0 SR1: 0 SR2: 0 SR3: 0

Service Provider CPU 0

CPU 0 (500782 service requests triggered in this Trace)

ID	SR Name	Priority	Index	Freq (/s)	Starting Time	Ending Time	ST	Min ST	Max ST	Ave. ST	ST Std Dev.	Interrupted by
1	Stm0_SR	1	131	15258.0	-	-	-	189	537	256.7	95.2	GPSR01, GPSR10, GPSR20, GPSR00
2	GPSR01	4	428	5004.0	-	-	-	78	117	78.7	5.0	GPSR10, GPSR20
3	GPSR10	7	431	5004.0	-	-	-	8	11	8.1	0.4	none
4	GPSR20	11	435	5004.0	-	-	-	8	11	8.1	0.4	none
5	GPSR00	3	427	5004.0	-	-	-	103	199	104.6	12.2	none

Figure 5.8: The screenshot of the interrupt profiling view in ChipCoach



Lock name: eth_test_lock   eth_test_lock

Lock Variable	Thread	#Accesses	Waiting time	Holding time	#attempts per acquisition	Min. #attempts	First attempt acquisition ratio
eth_test_lock	T200@C1;T14@C0;	4.6103764	38.050358	1978.708	3.6103764	2	99.39%
	T200@C1	5.112118	14.577577	1941.4858	4.112118	2	99.66%
	T14@C0	4.2012925	57.188366	2009.0563	3.2012928	3	99.17%
eth_test_lock	T14@C0:18435270	4	50	2016	3	3	Yes
eth_test_lock	T14@C0:18483006	4	50	2009	3	3	Yes
eth_test_lock	T14@C0:18531049	4	50	2009	3	3	Yes
eth_test_lock	T14@C0:18579837	4	50	2009	3	3	Yes
eth_test_lock	T14@C0:18627314	4	50	2009	3	3	Yes
eth_test_lock	T14@C0:18675202	4	50	2009	3	3	Yes
eth_test_lock	T14@C0:18723296	4	50	2009	3	3	Yes
eth_test_lock	T14@C0:18771252	4	50	2009	3	3	Yes
eth_test_lock	T14@C0:18819873	4	50	2009	3	3	Yes
eth_test_lock	T14@C0:18867390	4	50	2009	3	3	Yes
eth_test_lock	T14@C0:18914905	4	50	2009	3	3	Yes
eth_test_lock	T14@C0:18962806	4	50	2009	3	3	Yes

Lock Variable	Thread	#Ownship	#Accesses	#Failed Attempt Accesses	Conflicting Thread
eth_test_lock	T200@Core1	883	4514	1858	T14@Core0
eth_test_lock	T14@Core0	1083	4550	216	T200@Core1

Figure 5.9: The screenshot of ChipCoach's lock profiling view

Given an .elf file, the implemented interrupt profiling view automatically measures the current interrupt execution and provides beneficial information. Suggestions are also proposed to increase the performance of specific interrupts or even all interrupts. Based on the measured statistics, users acquire the accurate impression of how interrupts are being serviced in the system.

5.1.1.6 Lock profiling

The lock profiling analysis displays the results in several tables. As mentioned in Section 4, one of the given prerequisites is that the task ID information is provided. In ChipCoach, this is realized by tracing the OLDA range. The OS is modified to write the task ID when a task is started and resumed. Due to the limited trace buffer in AURIX, a two-step strategy is applied in the ChipCoach. First, all memory accesses are recorded by tracing and locks inside are detected. This tracing provides a good overview. Second, an interesting lock can be selected. At this time, only the selected lock is traced again and the tracing duration is much longer, which allows developers to conduct a deeper analysis about one lock. The two-step strategy offers both a balanced compromise of an overview and details. The implemented lock profiling view is shown in Figure 5.9. Different statistics are calculated including waiting time, holding time, failed attempts etc. By clicking a lock name, all threads accessing this lock are expanded. Further details are also available by expanding the specific thread ID. A console under the tables shows more flexible information such as warnings and suggestions.

5.1.1.7 Program flash contention

According to the proposed methodology of program flash contention analysis, the performance impact on all cores can be measured and estimated in just one run. However,

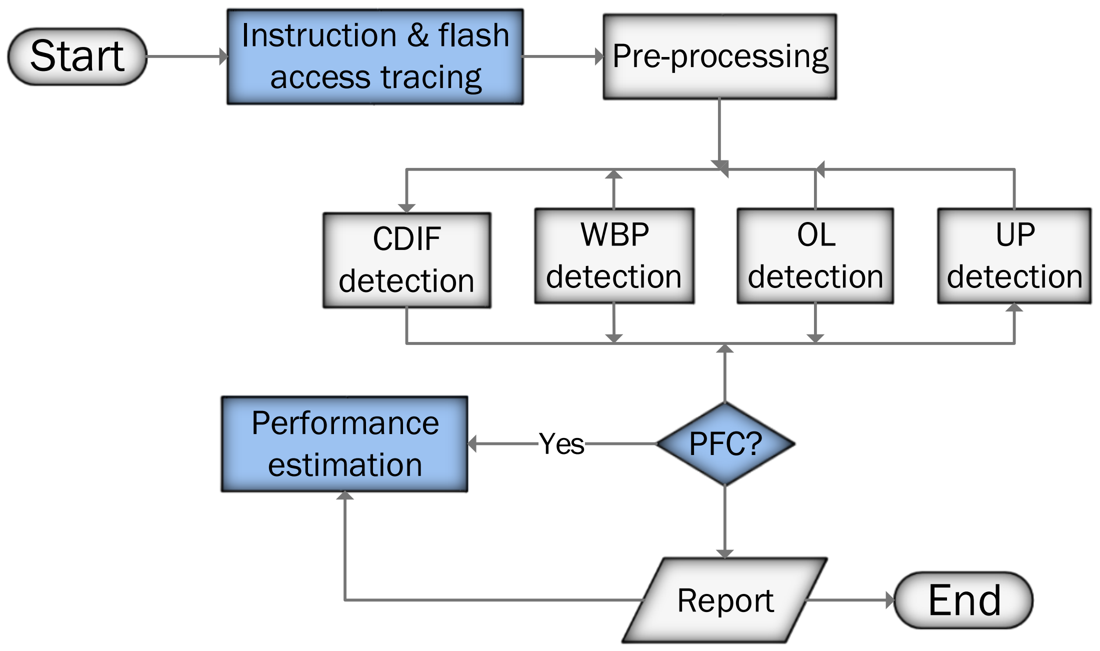
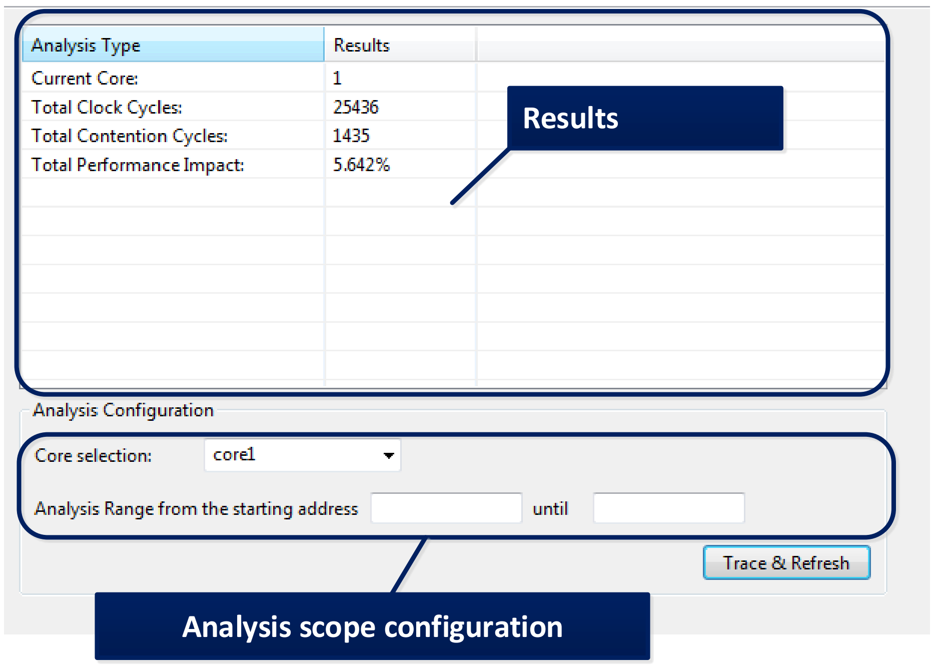


Figure 5.10: The analysis flow of the program flash contention analysis in ChipCoach

the implementation in ChipCoach is compromised due to the limitation of the MCDS that not all cores can be observed in parallel. Therefore, the implemented analysis is designed to detect program flash contention and estimate the performance impact on one core. It is sufficient for most use cases. A different core can be analyzed by rerunning the application.

The analysis flow is implemented as depicted in Figure 5.10. Both the instruction flow and the bus transfers related to program flash are traced. Based on the trace data, different possibilities including *CDIF*, *WBP*, *OL* and *UP* are detected respectively. Then, the program flash contention spots are derived and the corresponding performance impact is also estimated. Finally, a report is generated.

The program flash contention feature has a view to show performance penalties caused by program flash contention as shown in Figure 5.11. In the view, a core can be selected to be analyzed and also a range of program addresses can be typed into the view to limit the analysis to several specific functions. The view shows the overview of the results including the performance impact and the exact number of clock cycles added by program flash contention. A more detailed report is also generated locally as a text file, in which the delayed numbers of cycles and the number of contention spots for each instruction are available. By utilizing this report, the program flash contention can be reduced for a specific function, a thread or even a core.



Analysis Type	Results
Current Core:	1
Total Clock Cycles:	25436
Total Contention Cycles:	1435
Total Performance Impact:	5.642%

Analysis Configuration

Core selection: core1

Analysis Range from the starting address until

Figure 5.11: The screenshot of the program flash contention analysis view

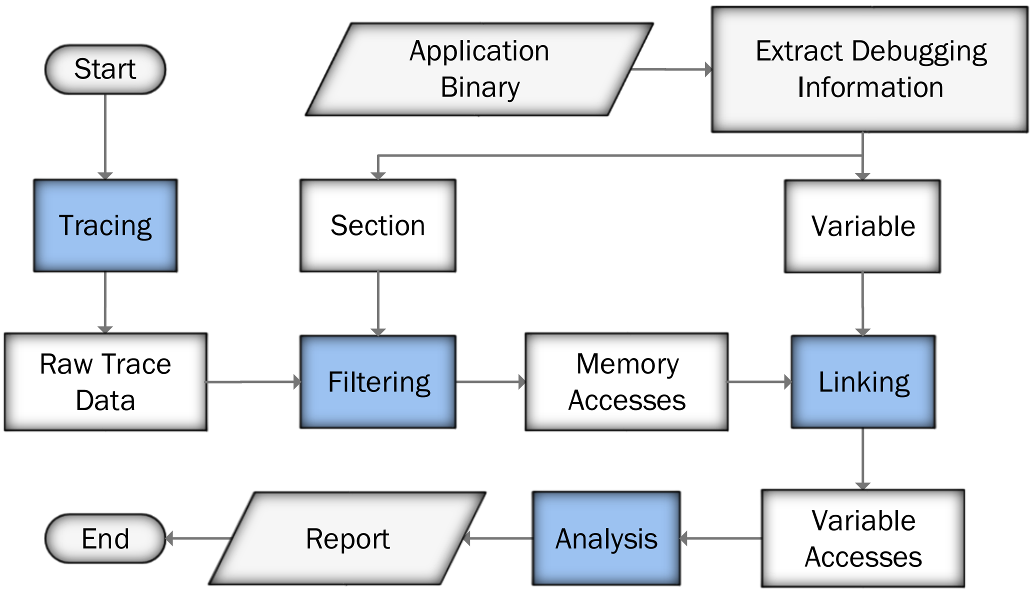


Figure 5.12: The workflow of the memory contention analysis feature in ChipCoach

5.1.1.8 Data memory contention

In ChipCoach, the data memory contention analysis is implemented according to Figure 5.12. At the beginning, ChipCoach configures MCDS to trace the memory accesses to a specific memory in the **Tracing** phase. Similar to the program flash contention analysis, not all memory modules can be traced at one time. For data memory contention analysis, the limitation is solved by automatic application rerunning. The tracing trigger is configured so that the tracing always starts at the point, which is guaranteed by the MCDS trigger support. Then the raw data is then reprocessed including filtering, which filters out the unneeded trace data and merges data from different tracing. The contention analysis for a specific memory module here is reliable because the proposed data memory contention methodology relies on the timing of memory accesses to the specific memory module, which are traced in one tracing. The merging operation here is for comparing the contention situations in different memory modules. Given the .elf file, symbolic and binary information is provided to the ChipCoach, which allows the ChipCoach to link memory accesses to variables and sections in the **Linking** phase. Finally, some metrics are calculated for each variable and each core in the **Analysis**, providing reference to the user to optimize the memory performance.

To start with the analysis, firstly the hardware dependent properties including the system consumers, providers, access latencies and the minimum memory access cycle interval (INT) have to be defined [48]. The AURIX TC29x device has the following consumers:

- **Core0, core1, core2**: every core is considered as a unique consumer.

- **DMA**: AURIX TC29x has a DMA which can also be a consumer.

Other potential consumers such as High Speed Serial Link (HSSL) and Ethernet are treated similar to DMA.

In addition, TC29x has the following providers:

- **DSPR0, DSPR1, DSPR2**: are the local scratchpad SRAMs directly coupled to their cores. They are accessible by their local cores and every other consumer connected to the SRI.
- **LMU** is a global data memory that is not directly associated with any core.

Latencies First, the memory access latencies between consumers and providers have to be identified [48]. Table 5.1 can be obtained from the AURIX user manual [106], presenting the read latencies in the AURIX TC29x system. It also shows the maximum CPU stall cycles due to access latency. The write latencies can be omitted, because a write operation is posted and does not halt the issuing core. Write operations could however introduce a bus (SRI) contention, which then delays the read operations. In AURIX, this penalty is mostly relieved by core-coupled store buffers that hold the write requests (store operations) from the core temporally. A read request has priority over the write requests in the store buffer so that it is not delayed by the previous write operations. In this implementation, the remote write access penalties are ignored. Table 5.1 states that accesses to the local memory consume zero clock cycles. The memory access latency for LMU is not provided in the user manual but measured in experiment.

Table 5.1: Read latencies (clock cycle) in AURIX TC29x

	DSPR0	**DSPR1**	**DSPR2**	**LMU**
core0	0	8	8	11
core1	8	0	8	11
core2	8	8	0	11

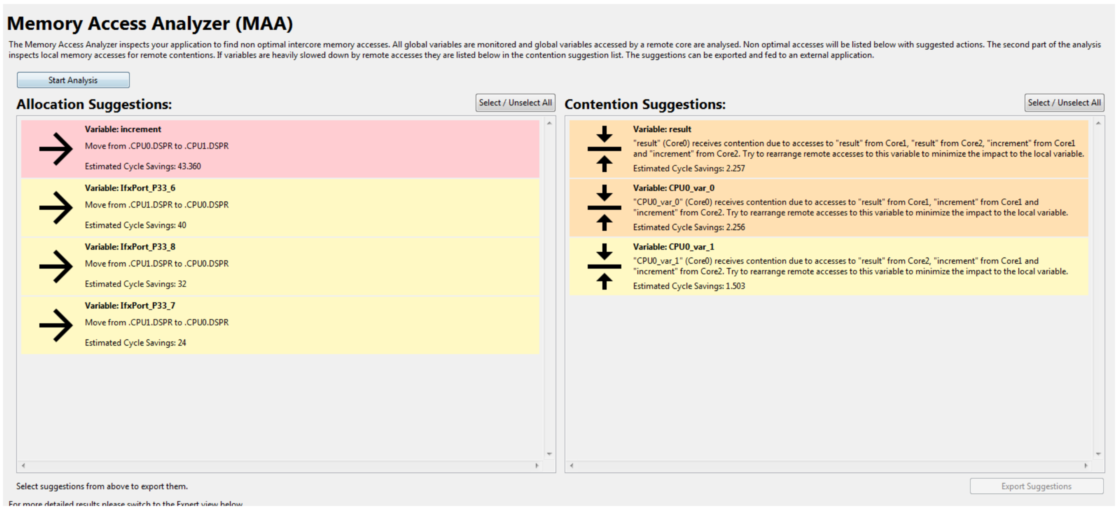
Minimum memory access cycle intervals: for AURIX TC29x, the following INTs are observed with several experiments.

Local to remote access: an access to the local memory has the *INT* of **one clock cycle**, to the nearest remote access from a different core to the same memory.

Remote to remote access: when two remote cores are concurrently accessing the same DSPR, the *INT* is **three clock cycles**.

Remote to remote access: if the LMU is accessed by two cores, the *INT* is **four clock cycles**.

The analysis results are displayed in two modes namely the basic view and the expert view, as shown in Figure 5.13 and Figure 5.14 respectively. The basic view only displays the basic information and suggestions without any details, which is suitable for users who would like to have a quick view. In the basic view, simple suggestions are also provided to give quick hints to improve the performance. The expert view shows all



Memory Access Analyzer (MAA)

The Memory Access Analyzer inspects your application to find non optimal intercore memory accesses. All global variables are monitored and global variables accessed by a remote core are analysed. Non optimal accesses will be listed below with suggested actions. The second part of the analysis inspects local memory accesses for remote contentions. If variables are heavily slowed down by remote accesses they are listed below in the contention suggestion list. The suggestions can be exported and fed to an external application.

Allocation Suggestions:

Variable: increment
Move from .CPU0.DSPR to .CPU1.DSPR
Estimated Cycle Savings: 43.360

Variable: IfxPort_P33_6
Move from .CPU1.DSPR to .CPU0.DSPR
Estimated Cycle Savings: 40

Variable: IfxPort_P33_8
Move from .CPU1.DSPR to .CPU0.DSPR
Estimated Cycle Savings: 32

Variable: IfxPort_P33_7
Move from .CPU1.DSPR to .CPU0.DSPR
Estimated Cycle Savings: 24

Contention Suggestions:

Variable: result
"result" (Core0) receives contention due to accesses to "result" from Core1, "result" from Core2, "increment" from Core1 and "increment" from Core2. Try to rearrange remote accesses to this variable to minimize the impact to the local variable.
Estimated Cycle Savings: 2.257

Variable: CPU0_var_0
"CPU0_var_0" (Core0) receives contention due to accesses to "result" from Core1, "increment" from Core1 and "increment" from Core2. Try to rearrange remote accesses to this variable to minimize the impact to the local variable.
Estimated Cycle Savings: 2.256

Variable: CPU0_var_1
"CPU0_var_1" (Core0) receives contention due to accesses to "result" from Core2, "increment" from Core1 and "increment" from Core2. Try to rearrange remote accesses to this variable to minimize the impact to the local variable.
Estimated Cycle Savings: 1.503

Select suggestions from above to export them.

For more detailed results please switch to the Expert view below

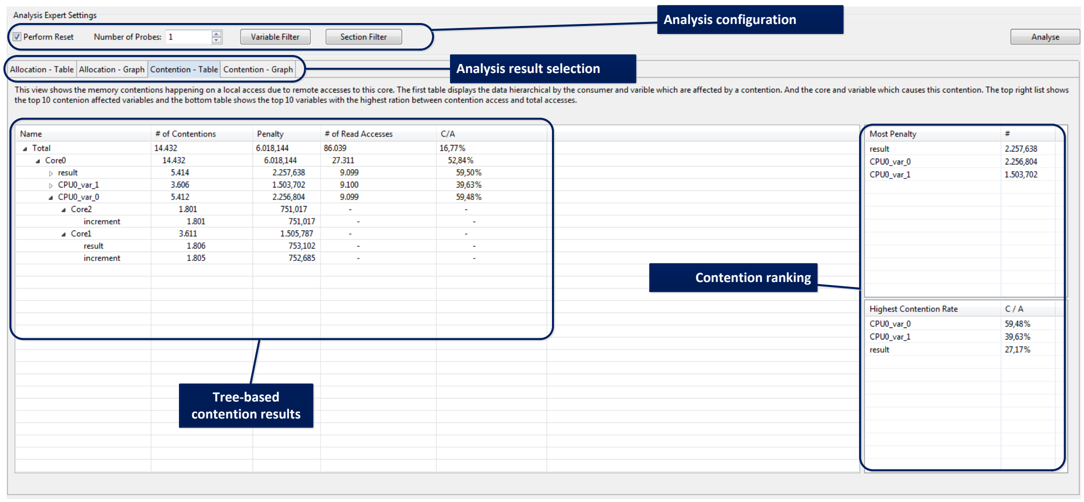
Figure 5.13: The screenshot of the basic view in the data memory analysis

detailed statistics of every detected global variable. With the expert view, users know the penalties of both data locality and contention of each variable. The switch of these two views is via the tab under the view.

Both the data locality analysis and the contention analysis are visualized as tables. In Figure 5.14, the memory contention analysis table is shown. The results are organized in a hierarchical way. Global variables are arranged under different core groups. Under a specific global variable, the contended cores and contended variables are shown by clicking the variable name. Penalties for both the data locality and the memory contention are calculated and displayed. The number of contention spots and the contention ratio are also provided to the users.

5.1.1.9 Clock configuration view

The clock distribution in state-of-the-art microcontroller is complex. The clock configuration feature shows the frequency of each module in AURIX and their hierarchical dependencies depicted in Figure 5.15. The hierarchical dependencies are indicated in the CCU registers that control the ratio between two modules. The root of the clock tree is starting from either an external oscillator or an internal clock. The frequency of the external oscillator is not known from the register configurations and it must be measured. In ChipCoach, either the clock frequency of CPU or the frequency of MCDS can be measured. The performance counters in CPU are used to measure the CPU frequency, while clock counter register in MCDS is applied to measure the frequency of MCDS. Then the frequency of the clock tree root can then be derived first and other modules can also be calculated. The CCU configuration process is also validated by rules that involve clock configurations as introduced before.



Analysis Expert Settings

☑ Perform Reset Number of Probes: 1 Variable Filter Section Filter

Allocation - Table | Allocation - Graph | Contention - Table | Contention - Graph

This view shows the memory contentions happening on a local access due to remote accesses to this core. The first table displays the data hierarchical by the consumer and varible which are affected by a contention. And the core and variable which causes this contention. The top right list shows the top 10 contenion affected variables and the bottom table shows the top 10 variables with the highest ration between contention access and total accesses.

Name	# of Contentions	Penalty	# of Read Accesses	C/A
◢ Total	14.432	6.018,144	86.039	16,77%
◢ Core0	14.432	6.018,144	27.311	52,84%
▷ result	5.414	2.257,638	9.099	59,50%
▷ CPU0_var_1	3.606	1.503,702	9.100	39,63%
◢ CPU0_var_0	5.412	2.256,804	9.099	59,48%
◢ Core2	1.801	751,017	-	-
increment	1.801	751,017	-	-
◢ Core1	3.611	1.505,787	-	-
result	1.806	753,102	-	-
increment	1.805	752,685	-	-

Most Penalty	#
result	2.257,638
CPU0_var_0	2.256,804
CPU0_var_1	1.503,702

Highest Contention Rate	C / A
CPU0_var_0	59,48%
CPU0_var_1	39,63%
result	27,17%

Figure 5.14: The screenshot of the expert view in the data memory analysis

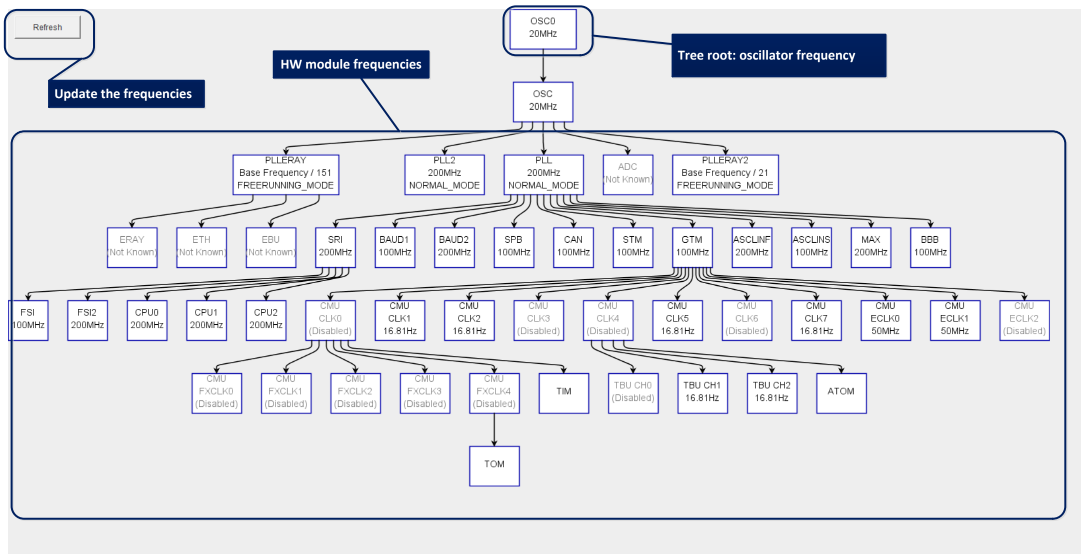
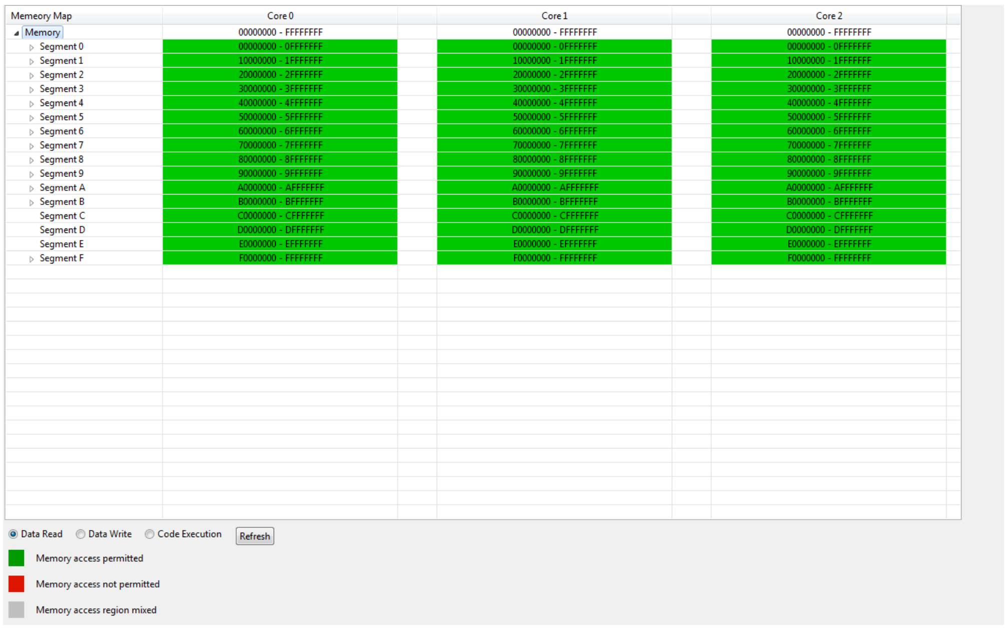


Figure 5.15: The screenshot of the clock configuration view



Memeory Map	Core 0	Core 1	Core 2
Memory	00000000 - FFFFFFFF	00000000 - FFFFFFFF	00000000 - FFFFFFFF
Segment 0	00000000 - 0FFFFFFF	00000000 - 0FFFFFFF	00000000 - 0FFFFFFF
Segment 1	10000000 - 1FFFFFFF	10000000 - 1FFFFFFF	10000000 - 1FFFFFFF
Segment 2	20000000 - 2FFFFFFF	20000000 - 2FFFFFFF	20000000 - 2FFFFFFF
Segment 3	30000000 - 3FFFFFFF	30000000 - 3FFFFFFF	30000000 - 3FFFFFFF
Segment 4	40000000 - 4FFFFFFF	40000000 - 4FFFFFFF	40000000 - 4FFFFFFF
Segment 5	50000000 - 5FFFFFFF	50000000 - 5FFFFFFF	50000000 - 5FFFFFFF
Segment 6	60000000 - 6FFFFFFF	60000000 - 6FFFFFFF	60000000 - 6FFFFFFF
Segment 7	70000000 - 7FFFFFFF	70000000 - 7FFFFFFF	70000000 - 7FFFFFFF
Segment 8	80000000 - 8FFFFFFF	80000000 - 8FFFFFFF	80000000 - 8FFFFFFF
Segment 9	90000000 - 9FFFFFFF	90000000 - 9FFFFFFF	90000000 - 9FFFFFFF
Segment A	A0000000 - AFFFFFFF	A0000000 - AFFFFFFF	A0000000 - AFFFFFFF
Segment B	B0000000 - BFFFFFFF	B0000000 - BFFFFFFF	B0000000 - BFFFFFFF
Segment C	C0000000 - CFFFFFFF	C0000000 - CFFFFFFF	C0000000 - CFFFFFFF
Segment D	D0000000 - DFFFFFFF	D0000000 - DFFFFFFF	D0000000 - DFFFFFFF
Segment E	E0000000 - EFFFFFFF	E0000000 - EFFFFFFF	E0000000 - EFFFFFFF
Segment F	F0000000 - FFFFFFFF	F0000000 - FFFFFFFF	F0000000 - FFFFFFFF

[✓ Data Read] Data Write Code Execution Refresh

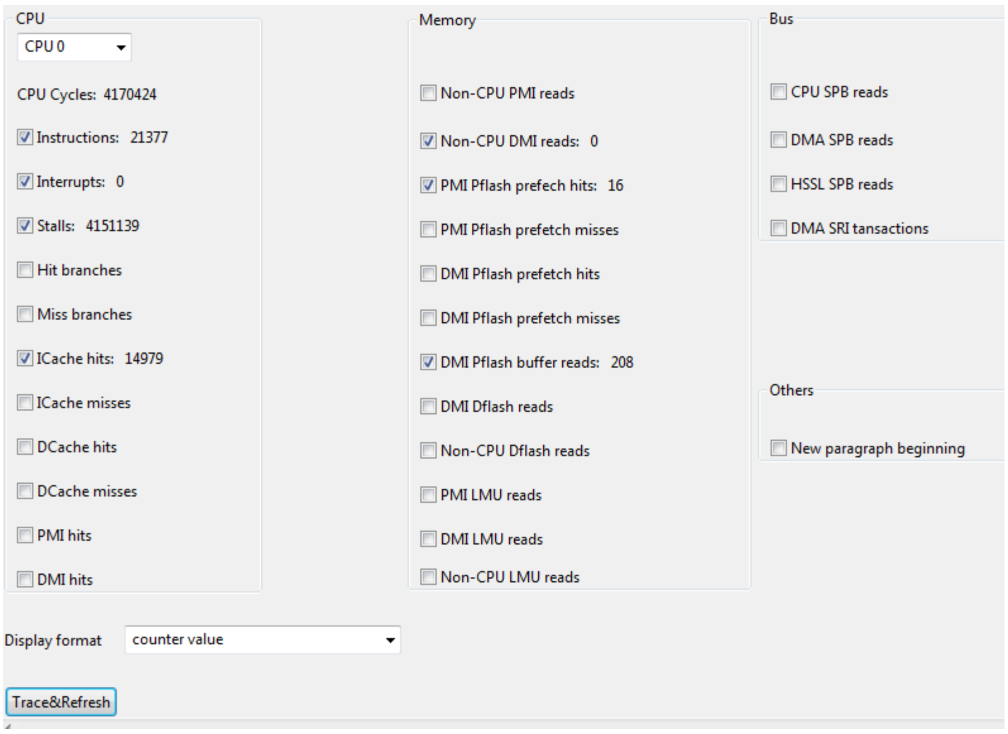
🟩	Memory access permitted
🟥	Memory access not permitted
⬜	Memory access region mixed

Figure 5.16: The screenshot of ChipCoach's memory protection unit view

5.1.1.10 Other features

The Memory Protection Unit (MPU) is used to prevent unintended memory accesses, which is critical for safety-related applications. The CPU MPU provides multiple protection sets with multiple protection ranges. It is meant to guarantee that no unintended memory access is initiated by a particular core. An alarm will be triggered if a violation happens. The CPU MPU is supposed to be configured at the beginning of a task phase. ChipCoach provides visualization support for the CPU MPU configurations, which allows users to see the current valid MPU configuration, as shown in Figure 5.16. The view displays the current setting for different cores and for different operations (read, write, execution).

Another performance profiling feature in ChipCoach is the performance profiling counter visualization. MCDS supports performance counter tracing. This includes many different counters like instruction, interrupt, memory accesses, bus transfers, DMA transactions, cache hit, cache miss etc. as shown in Figure 5.17. These counters can be used to analyze the system performance. ChipCoach also displays the counter values in different formats. It provides the original counter values, the value over cycle number (e.g. Instruction Per Cycle (IPC)) and the value over instruction number (e.g. cache misses per instruction). These formats can be chosen by selecting the dropdown list.


CPU

CPU 0

CPU Cycles: 4170424

☑ Instructions: 21377

☑ Interrupts: 0

☑ Stalls: 4151139

☐ Hit branches

☐ Miss branches

☑ ICache hits: 14979

☐ ICache misses

☐ DCache hits

☐ DCache misses

☐ PMI hits

☐ DMI hits

Memory

☐ Non-CPU PMI reads

☑ Non-CPU DMI reads: 0

☑ PMI Pflash prefech hits: 16

☐ PMI Pflash prefetch misses

☐ DMI Pflash prefetch hits

☐ DMI Pflash prefetch misses

☑ DMI Pflash buffer reads: 208

☐ DMI Dflash reads

☐ Non-CPU Dflash reads

☐ PMI LMU reads

☐ DMI LMU reads

☐ Non-CPU LMU reads

Bus

☐ CPU SPB reads

☐ DMA SPB reads

☐ HSSL SPB reads

☐ DMA SRI tansactions

Others

☐ New paragraph beginning

Display format: counter value

Figure 5.17: The screenshot of the performance counters view in ChipCoach

5.2 Experimental evaluation

The proposed methodologies are first implemented as features in ChipCoach. In order to prove the effectiveness and the accuracy of the proposed methodology and the implementation, several experiments were designed and experimental results are evaluated. The experimental evaluations cover only the quantitative analyses such as the lock profiling analysis, the program flash contention analysis and the data memory analysis.

5.2.1 Lock profiling analysis

In this part, a case study is done with ChipCoach to analyze the lock appliance of two applications on Infineon AURIX TC29x. These two applications run on the OSs Erika and FreeRTOS respectively. Active locks on both OSs are detected and reported to the developers. One of the applications, i.e. an Ethernet demonstration application on Erika, is described in detail and then the profiling results obtained with ChipCoach are discussed in this section.

5.2.1.1 Ethernet demonstration application

The Ethernet demonstration application is designed to show the basic functionality of Ethernet that is currently being used to transfer large amounts of data in cars. It consists

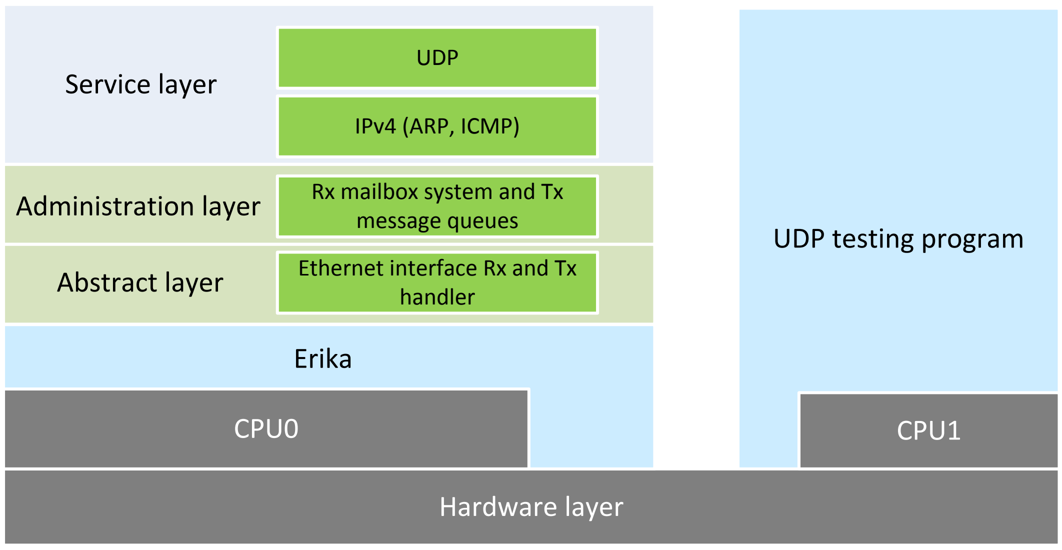


Figure 5.18: The architecture of the Ethernet demonstration application

of two parts as shown in Figure 5.18. The first part is the main functionality running on CPU0, which provides UDP service. It is capable of receiving, processing and sending UDP frames. It has several layers including abstract layer, administration layer and service layer. All these layers are scheduled using Erika, a hard real-time embedded OS. Threads can be preempted depending on the priority. The second part is the UDP testing program running on CPU1 without OS. The main purpose of this part is to stimulate and verify the first part. It periodically generates data and then sends it out using the UDP service provided in the service layer. It also checks the received UDP frames and validates the correctness. This part is periodically triggered by the System Timer (STM). The interaction between the UDP testing program (Thread 200) and the UDP service (Thread 14) is protected by a spinlock.

5.2.1.2 Lock profiling results

ChipCoach is applied to check the spinlock usage in the Ethernet demonstration application. After the first run, a lock named `eth_test_lock` is detected as a spinlock protecting shared variables, which are commonly used data buffers for the UDP testing. Then a second run with tracing limited to only `eth_test_lock` is performed. The tool displays the statistics in the table and also indicates two bad lock behaviors to the user. Those are spinning with atomic operations and thread preemption while holding a lock. It is noticed that the current lock is a test-and-set lock and interrupts are not disabled inside the critical section. Two optional improvements are available i.e. applying the test-test-and-set lock and disabling interrupts inside the critical section.

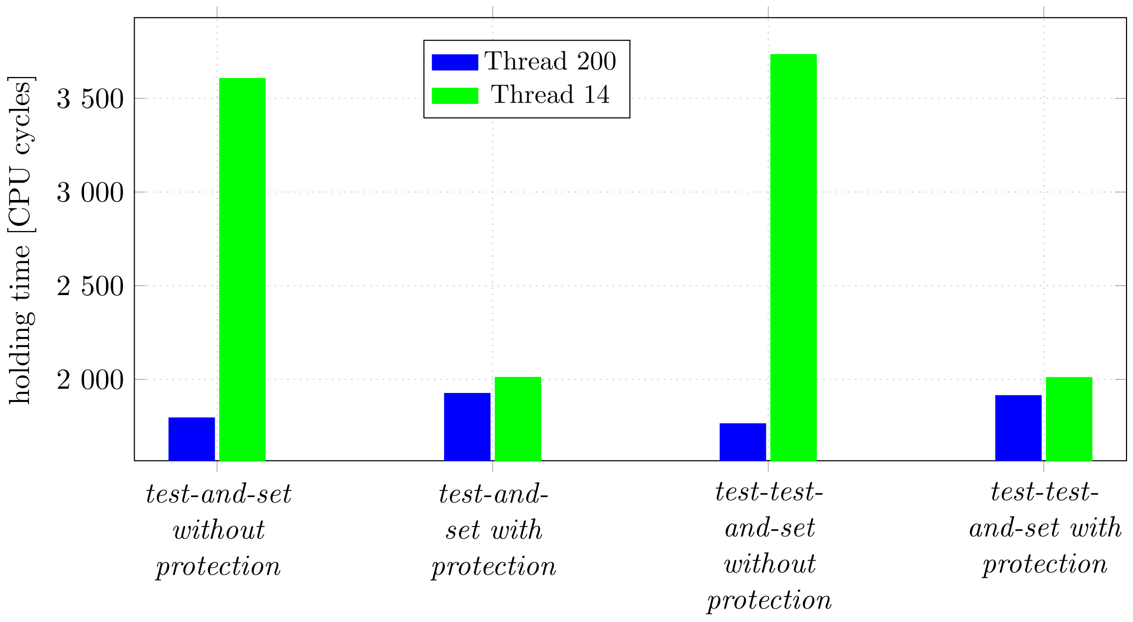


Figure 5.19: The holding time of thread 14 and thread 200 owning `eth_test_lock`

With these two options, four alternatives namely *test-and-set without protection* (without any improvement), *test-and-set with protection*, *test-test-and-set without protection* and *test-test-and-set with protection* are investigated. The holding time and the waiting time for the four alternatives are depicted in Figure 5.19 and Figure 5.20 respectively. The thread 200 is the UDP testing program while the thread 14 is the UDP service. The values shown in the figures are the average numbers of ten measurements.

Due to the fact that interrupts are not disabled in alternatives *test-and-set without protection* and *test-test-and-set without protection*, thread 14 could be preempted while holding the lock. This results in extra-long holding time as shown in Figure 5.19, while the holding time of thread 200 does not change much among the different alternatives.

As the holding time of thread 14 without protection is much larger than the one with protection, the blocking chance of thread 200 is also higher, resulting in longer waiting time of thread 200 in Figure 5.20. The waiting time of thread 14 with test-test-and-set lock is always higher than the first two alternatives. This is because the spinlock is located in the CPU1's local memory and thread 200 accesses this lock much faster than the remote thread 14. It is easier for a local thread to acquire with the test-test-and-set implementation. *test-test-and-set without protection* has extremely long waiting time for both threads. They are having many collisions in this timing. According to the results, protection is necessary to avoid preemption while holding a lock. The test-test-and-set lock gives the local thread a better chance. For this Ethernet demonstration application, the test-and-set lock with protection could be selected to decrease the waiting time of the thread 14.

The experience in profiling applications on Erika and FreeRTOS indicates the feasibil-

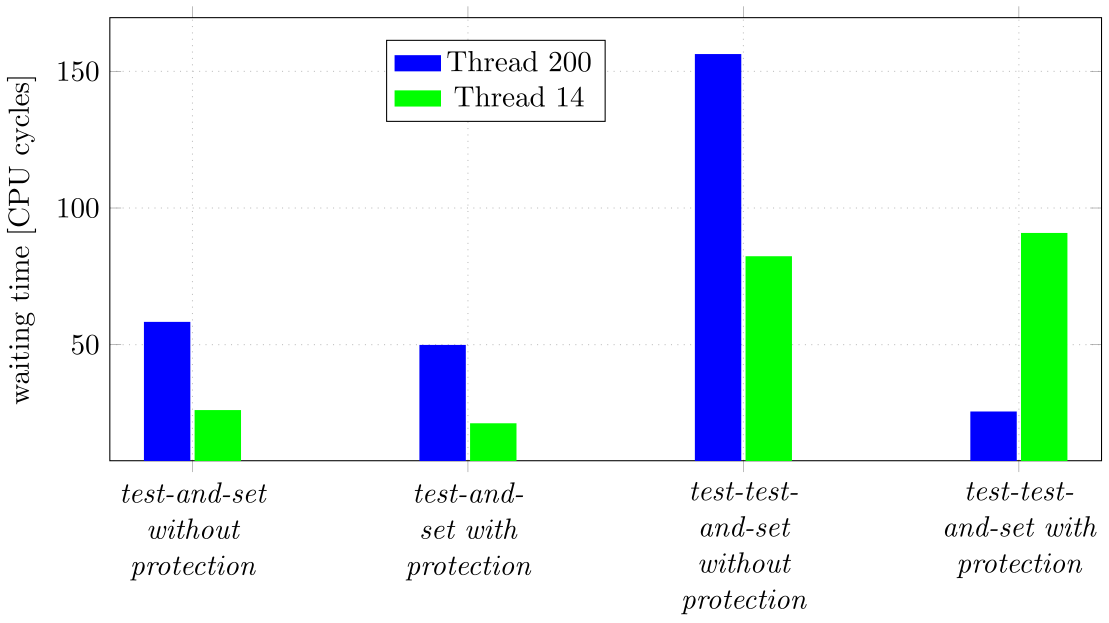


Figure 5.20: The waiting time of thread 14 and thread 200 acquiring `eth_test_lock`

ity of the proposed OS-independent lock detection approach. The above profiling analysis explains how to apply the proposed approach to increase the system transparency and improve the lock usage. The lock usage situation is complex and sometimes out of developer's expectation. The suitable lock type depends on different working conditions. A profiling tool clarifying this complex situation is therefore necessary. Especially for embedded applications, such an OS-independent tool is convenient to use.

5.2.2 Program flash contention analysis

To prove the feasibility and the effectiveness of the proposed program flash contention analysis algorithm, an experiment was designed using third-party test program on AURIX TC29x. Owing to the limitation that program flash cannot be detected with the existing tools, a controlled experiment, consisting of a Control Group (CG) and several Experimental Group (EG)s, is created to derive the contention spots and performance impact caused by the contention. In this way, the results analyzed by ChipCoach can be compared against the results measured by the controlled experiment, showing the level of accuracy.

5.2.2.1 Experiment setup

The experiment is conducted on AURIX TC29x with disabled program cache which facilitates the observation of program flash contention. Instructions have to be fetched from the program flash and more program flash contention should be observed. The proposed contention analysis method is independent of cache configuration and it also

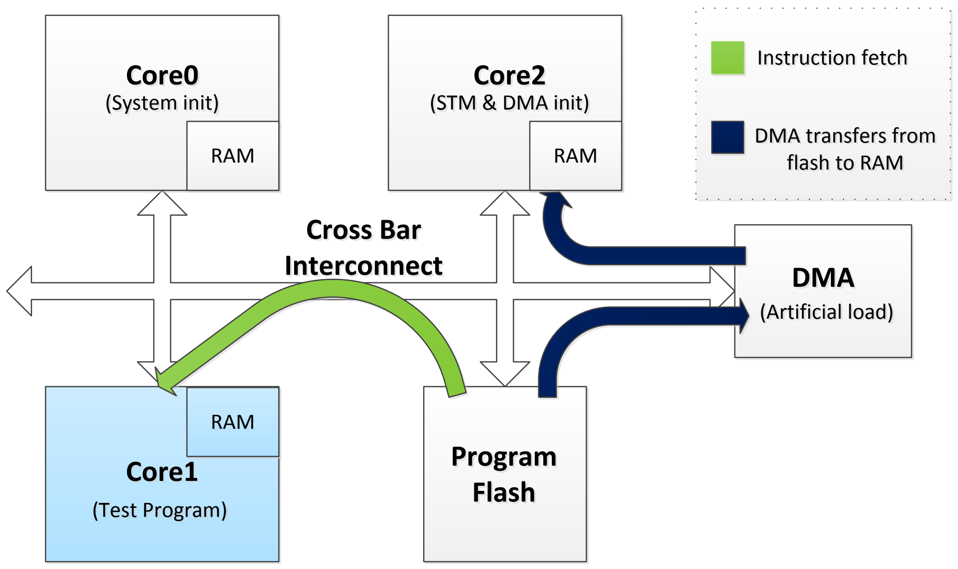


Figure 5.21: Experiment setup: The control group has only instruction fetches while the EG has both instruction fetches and DMA transfers that cause program flash contention.

works well with enabled program cache. In this experiment, the frequency of the core is set at 80MHz that is the same as the frequency of MCDS.

The program flash contention can be generated in two ways. (a) a real multicore application running on AURIX and program flash contention might be introduced. However, whether there is contention and the degree of the flash contention depends on the characteristics of the application itself, which is difficult to control. (b) a single-core application running on AURIX and program flash contention can be introduced by adding artificial load generated by DMA as shown in Figure 5.21. The program flash contention occurs when the DMA accesses are conflicting with the normal instruction fetches by the tested core. The artificial load is totally under control so the degree of the program flash contention can be also tuned. The above two ways have advantages and disadvantages. The way (a) is the closer to reality but the flash contention is determined by the application itself, making it difficult to measure different contention degrees. The way (b) outpaces the way (a) in controllability. The accuracy of the proposed approach can be evaluated in different contention scenarios in the way (b). Therefore, the way (b) is chosen in experiment.

A time-triggered DMA channel is applied to generate the artificial load by transferring data from the program flash to DSPR in the CPU2. The trigger is controlled by the STM, which sends out a DMA request when the time reaches a pre-defined value. This pre-defined value is set in the STM by adjusting the lower compare bits. Each time, a 32-bit-word is read from the program flash by DMA. The read addressing increments by one word after each transfer to avoid the "cache" effect since a small buffer exists in the program flash. Accordingly, the artificial load is tuned by setting the different trigger periods in STM. This artificial load generation is considered to have no impact on normal operations of the CG except for the program flash contention as the test program does not use DMA and STM, meaning no sharing between the artificial load generation and the normal operations.

The single-core test programs mentioned in the way (b) include two benchmarks namely EEMBC and Dhrystone [128] for AURIX. The EEMBC is AutoBench version 1.14 from Automotive Subcommittee and the Dhrystone is C 2.1 version. In both the controlled group and the EG, non-intrusive instruction tracing is applied to record the time information of each instruction, artificial flash accesses and instruction fetches. As the program flash contention is the only difference between two groups, the extra time consumed in the EG compared to the CG is then only caused by the program flash contention. Then the instructions in the EG consuming more clock cycles than those in the CG are treated as contended instructions, meaning that they are impacted by the contention.

To evaluate the methodology accurately and comprehensively, several terms are defined as follow [47].

Measured means the value is measured by comparing the EGs to the contention-free CG. It shows the actual performance impact and the contended instructions.

Estimated means the value is estimated by ChipCoach based only on the EG.

Load is defined as the DMA read frequency. The average access period cycles (access intervals) are measured and then the load is calculated, which is presented in read/cycle.

Correctly Estimated Contention (CEC) is defined as the intersection set between measured contended instructions and estimated contended instructions. To show the correctness of the method, detection rate defined the ratio of correct contention estimations to measured contention, is introduced.

False Positives in this experiment, means the overestimated contention that actually is not contention.

False Negatives in this thesis, is the underestimated instructions that indicate no contention in the estimation but actually have contention.

Performance Impact is defined as the ratio of additional clock cycles over the measured total clock cycles. For each contention, the extra clock cycles are estimated by the methodology and then added up to have an overall performance impact in the experiment.

5.2.2.2 Results

The tracing scope is adjusted to the same sequence of the instructions in all tests to make the comparison reproducible. The experiments were repeated 5 times and the variances are only several clock cycles, which can be ignored. This is because a hardware reset is performed for each measurement and the identical trace scope is also guaranteed by the MCDS tracing trigger. In the experiment, the measured contention spot numbers and the estimated contention spot numbers are recorded. The detection rate as defined above is also calculated.

Two experiments with different test programs were conducted. Experiment 1 is with the EEMBC benchmark and Experiment 2 is with Dhrystone. The results of both experiments are shown in TABLE 5.2 and TABLE 5.3 respectively.

The measured tracing scope starts from the reset and ends until the trace buffer is full. As shown in Table 5.2, always the same number of instructions in the identical sequence is considered as the measurement scope. For each row in the table, the only setup difference is the degree of the artificial load or access frequency. The measured contention is calculated by comparing the EGs to the contention-free CG while the estimated contention is estimated by ChipCoach based only on the EG. However, some contention spots may be overlooked (false negative) by the analysis method and some spots may be overestimated (false positive). Only the common spots between the measured and the estimated are treated as correctly estimated contention.

Table 5.2: The experiment results of the EEMBC benchmark

Load (read/cycle)	Instructions	Measured contention	Estimated contention	CEC	False negatives	False positives	Detection rate
0,008	14486	90	89	81	9	8	90,00%
0,016	14486	189	182	170	19	12	89,95%
0,031	14486	366	365	343	23	22	93,72%
0,063	14486	761	751	724	37	27	95,14%
0,124	14486	1708	1675	1645	63	30	96,31%

Table 5.3: The experiment results of the Dhrystone benchmark

Load (read/cycle)	Instructions	Measured contention	Estimated contention	CEC	False negatives	False positives	Detection rate
0,008	16367	95	86	74	21	12	77,89%
0,016	16367	158	158	133	25	25	84,18%
0,031	16367	342	330	304	38	26	88,89%
0,062	16367	615	646	592	23	54	96,26%
0,124	16367	1429	1427	1373	56	54	96,08%

In order to show the impact of the program flash contention better, the performance impact is also analyzed compared to the ideal scenario that is without program flash contention i.e. the controlled group. Both the measured performance impact and the estimated performance impact for the Experiment 1 and the Experiment 2 are shown in Figure 5.22 and Figure 5.23.

5.2.2.3 Assessments

The artificial load shown in Table 5.2 is doubled in every row by using fewer comparison bits for STM. The number of contention spots also increases according to the artificial load as shown in TABLE 5.2. The overlooked contention spots which are false negatives are mainly slight contention. As discussed in the methodology Section 4.3.4.2, only

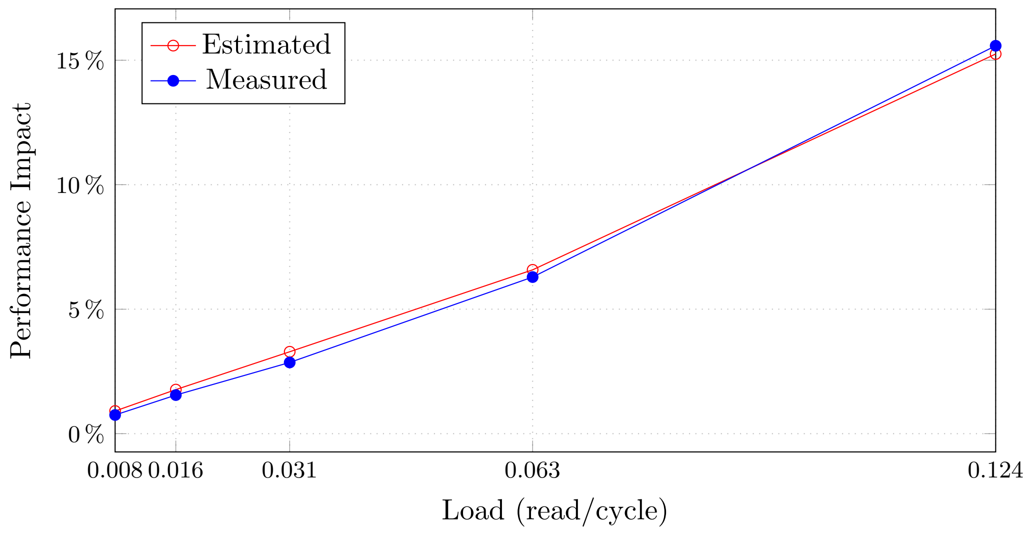


Figure 5.22: The performance impact in the Experiment 1 with the EEMBC benchmark

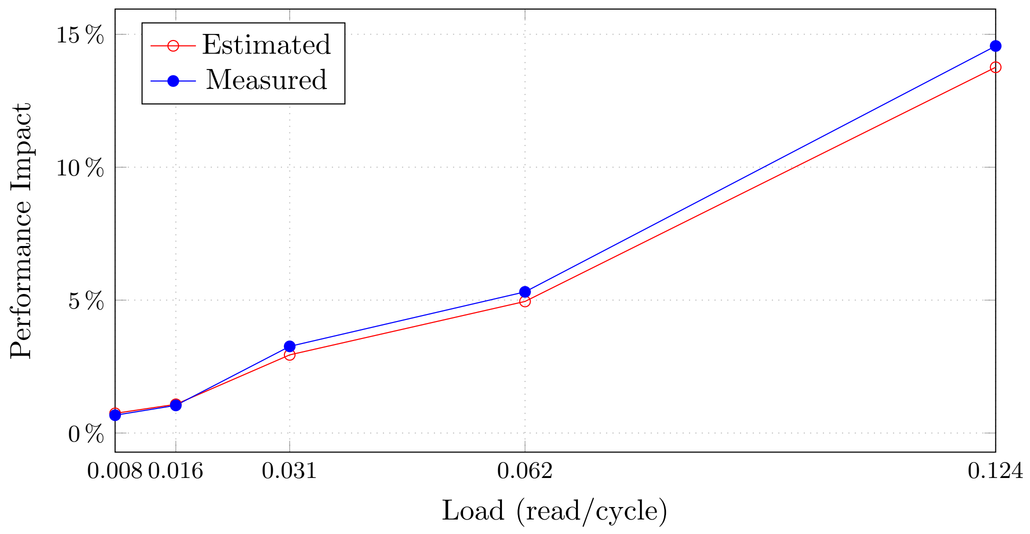


Figure 5.23: The performance impact in the Experiment 2 with the Dhrystone benchmark

CDIFs which delay a lot and even stop the core running are analyzed. The slight contention spots also slightly disturb the normal execution of the core's pipeline but the influence is slight. Therefore, the performance impact caused by slight contention spots can be ignored, which is also verified by the performance impact in Figures 5.22 5.23 that have high estimation accuracy. The detection rates in both experiments indicate that the detection rate increases when load is higher. This is because the faction of slight contention spots is less in higher load scenarios.

The maximum measured performance impact is up to 16%. Even though this is only the worst case with disabled program cache, this shows the significant impact of program flash contention and the importance of the program flash contention analysis. In fact, program cache may not help if the routine has not been executed before. For example, an interrupt happens and it is not cached. The execution of this interrupt can be severely delayed by program flash contention, which may result in issues.

In this experiment, the performance impact estimation matches the measured performance impact quite well but it still has deviations. The performance impact deviations come from the estimation of the ideal scenario. The ideal scenario can only be coarsely estimated without simulations. The real scenario is usually much more complex than the ideal case. The flash-contention-free scenario could be influenced by other factors such as other resource contention, extremely long instruction execution time and pipeline hazards. All these other non-ideal cases could cause errors in the ideal case estimation. For example, in the ideal case an instruction takes 2 cycles. In the analysis, we observe 5 cycles caused by program flash contention. The additional 3 cycles cannot be imputed to only the program flash contention. This instruction may also have data memory contention, which leads to e.g. 1 additional cycle, so the program flash contention is only responsible for additional 2 cycles.

The flash contention is not only limited to the instruction fetch contention but also happen to constant data read. Usually const data is stored together with instructions. This allocation has two possibilities: contention between instruction and const data or between constant data and constant data/instruction. For the first possibility, they could also be analyzed and estimated by this method. The second possibility that has no impact on the instruction fetch but have influence on the constant data fetch, are not detectable with the proposed method. For example, a const data fetch is delayed due to a simultaneous instruction fetch by another core. This situation can also be solved similarly by the proposed method 4.3.5.

5.2.3 Data memory analysis

Similar to the previous situation in the Section 5.2.2, a controlled experiment is designed to measure the accuracy and the effectiveness of the proposed method in Section 4.3.5. Both the performance impact and the numbers of memory contention spots are estimated by ChipCoach and compared to the experimental results. As mentioned before Section 5.2.2, there are two ways to generate memory contention. One is using a real multi-core application and the other is based on a single-core application with artificial load. In this experiment, the second way is still selected as it is more controlled and more

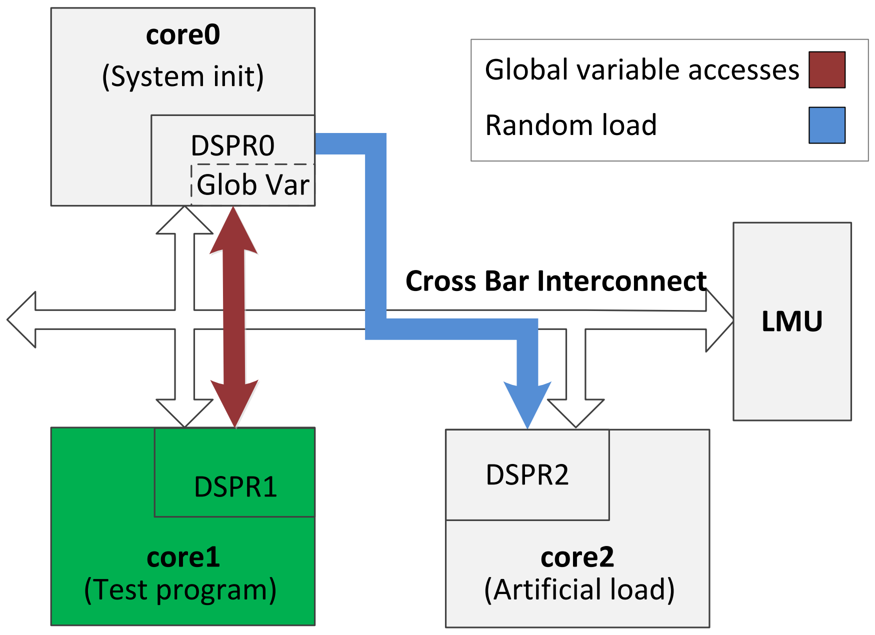


Figure 5.24: AURIX TC29x diagram. Experiment setup: core0 initializes the system; core1 runs the test program; core2 adds load to DSPR0

flexible. The single-core test programs in this experiment include the EEMBC benchmark and the Dhrystone benchmark. The EEMBC is AutoBench on matrix calculation from Automotive Subcommittee and the Dhrystone is C version 2.1. The runtimes of the two test programs under various conditions that are with/without data locality/ memory contention are manipulated and compared to show the performance impact.

5.2.3.1 Setup

This experiment is based on AURIX TC29x with both program cache and data cache enabled. The experiment setup is as shown in Figure 5.24. The Tasking (TriCore v4.3r3) compiler was used to compile the application. As shown in the figure, each core has its local data memory and the global variables can be allocated to any of these data memories at compile time. The data allocation is done by pragma commands, which are supported by the compiler. Different experimental scenarios can be created by manipulating the location of these global variables. The *CG* as a reference group is designed to be the basic group. The EGs always deviate from the CG by only one factor. This guarantees that the result difference is caused by the deviated factor.

In this experiment setup, core0 is responsible for the system initialization. When the initialization is finished, an interrupt is triggered to wake up both core1 and core2. core1 runs the single-core test program, which is either the Dhrystone or the EEMBC benchmark. core2 is designated to generate artificial load to core0 by accessing a variable *ld_sourceTarget* in core0.DSPR0 randomly. The random access here means that the time

Table 5.4: Dhrystone: analyzed penalties

Group	Runtime	Locality Penalties	Contention Penalties	Sum Penalties
CG	215 406	35 200	0.00	35 200.00
EG0	216 528	35 200	1 080.70	36 280.70
EG1	216 806	35 200	1 335.10	36 535.10
EG2	180 590	0	0.00	0.00

Table 5.5: Dhrystone: measured performance impact

Group	Contention Impact (EG-CG)	Locality Impact (CG-EG2)	Total Impact (EG/CG-EG2)	Error
CG	-	34 816	34 816	1.10%
EG0	1 122	-	35 938	0.95%
EG1	1 400	-	36 216	0.88%
EG2	-	-	-	-

interval between two read operations is a random number of cycles.

The *CG* is the group with remote accesses but without potential contention. This is achieved by locating the global variable on core0's DSPR0 and switching off the random access generation on core2. Two EGs (*EG0*, *EG1*) are designed to have both potential contention and data locality issues. The difference between these two groups is that the accessing intervals of the artificial load are generated by two different random seeds. In the last EG (*EG2*), which is meant to be the optimal group, the global variable is moved from core0 to core1. Therefore, no potential memory contention and data locality issues are possible.

5.2.3.2 Results

Tables 5.4, 5.5, 5.6 and 5.7 show the results of the experiment with the two test programs. The **Runtime** shows the total runtime in CPU cycle of the test programs in different groups, which are measured by the CPU clock counter. The locality penalties and contention penalties are analyzed and estimated by ChipCoach while the locality impact and the contention impact are measured calculating the runtime difference between the *EG0*, *EG1* groups and the *CG* group. The group name (*CG*, *EG0...*) in the following calculations is short for the runtime of this group. The locality penalties are the same for *CG*, *EG0* and *EG1* groups, since the numbers of remote accesses to the global variable are identical. **Error** is calculated according to Equation 5.10.

$$Error = \left| \frac{\textbf{Sum Penalties} - \textbf{Total Impact}}{\textbf{Total impact}} \right| \tag{5.10}$$

Table 5.6: EEMBC: analyzed penalties

Group	Runtime	Locality Penalties	Contention Penalties	Sum Penalties
CG	626 547	27 352.00	0.00	27 352.00
EG0	627 486	27 352.00	796.70	28 148.70
EG1	627 407	27 352.00	902.07	28 254.07
EG2	599 601	0.00	0.00	0.00

Table 5.7: EEMBC: measured performance impact

Group	Contention Impact (EG-CG)	Locality Impact (CG-EG2)	Total Impact (EG/CG-EG2)	Error
CG	-	26 946	26 946	1.51%
EG0	939	-	27 885	0.95%
EG1	860	-	27 806	1.61%
EG2	-	-	-	-

Figures 5.25 and 5.26 display the measured runtime (blue) of all the experiment groups. The middle bar shows the optimal runtime estimated by ChipCoach, which is defined as the runtime excluding the estimated sum penalties. The measured optimal runtime is *EG2*.

The next Figures 5.27 and 5.28, show the contributions to the sum penalties from locality issues and memory contention issues. The group *CG* only has the locality penalties because the random access generation is inactive in core2. *EG* has neither the locality penalties nor the contention penalties.

5.2.3.3 Assessments

In the *EG2*, all three bars are the same height as expected, because there are no memory contention and data locality penalties in this group. The runtime of the contention groups *EG0EG1* is slightly higher the reference group *CG*, which is caused by the introduction of data memory contention. The reason why the impact is slight is that the contention only exists in the accesses to the global variable, which impacts only a small part of the test program. Another reason is that only partial accesses have contention with the artificial load. The estimated optimal runtime by ChipCoach is quite close the real measured optimal runtime *EG2*, meaning that the estimation accuracy of the proposed method is good. The comparison of the **Locality Impact** column and the **Locality Penalties** column in tables 5.4 5.5 5.6 5.7 also indicates a close estimation and it is similar to the contention comparison. The estimated locality penalties are always higher than the measured locality impact. The reason is that the locality penalties are described as maximum core stall cycles according to the user manual so they are the

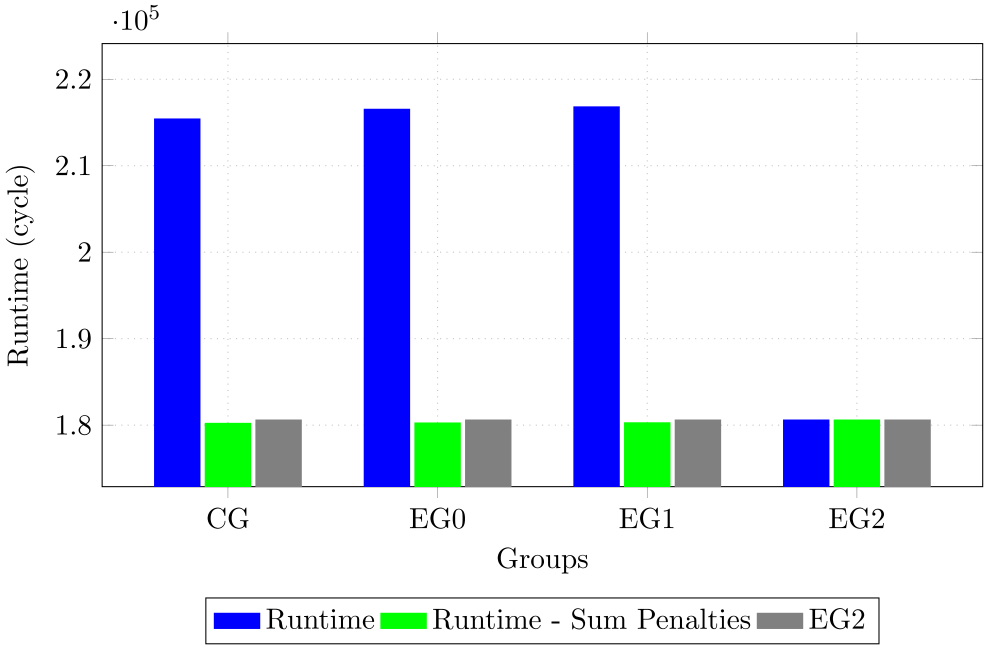


Figure 5.25: Dhrystone runtime comparison: the first bar shows the measured runtime (blue); the second bar shows the estimated optimal runtime (green) and the third bar shows the measured optimal runtime (gray).

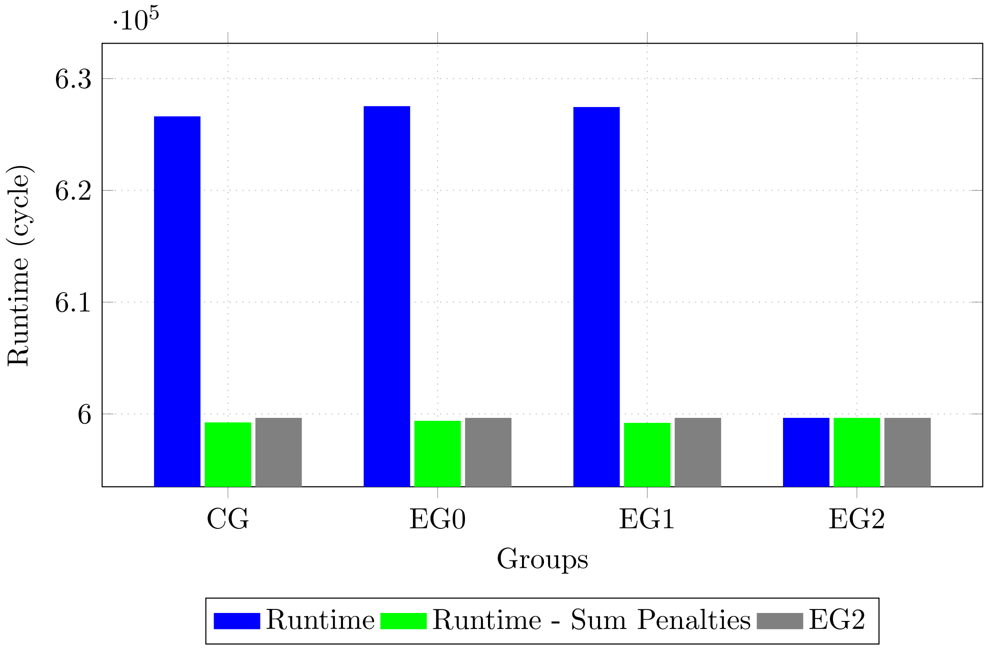


Figure 5.26: EEMBC runtime comparison: The first bar shows the measured runtime (blue); the second bar shows the estimated optimal runtime (green); and the third bar shows the measured optimal runtime (gray).

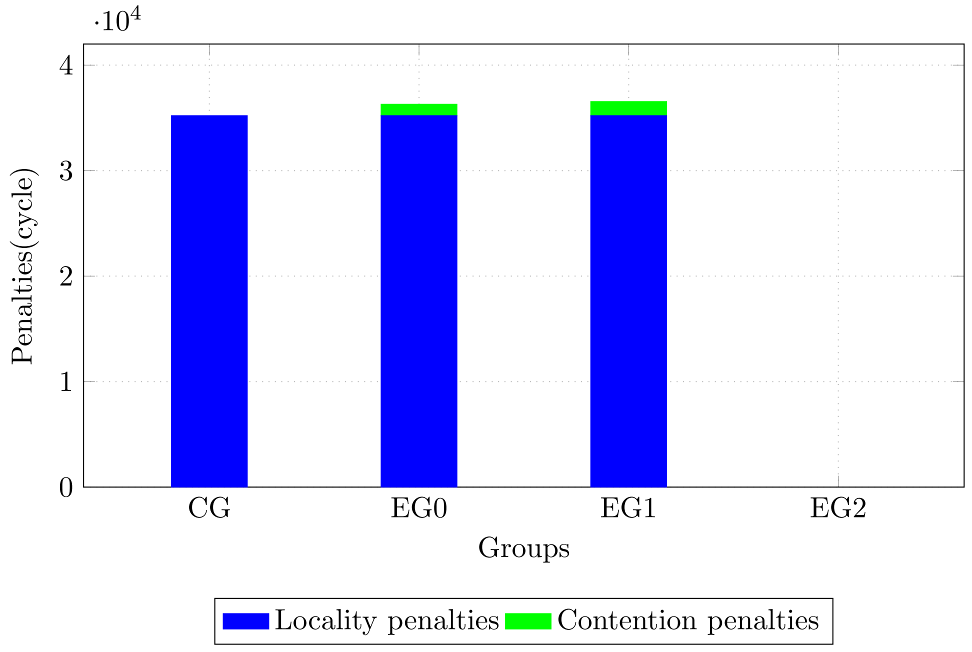


Figure 5.27: Dhrystone allocation and contention penalties: Estimated locality penalties (blue) and contention penalties (green) of the different groups.

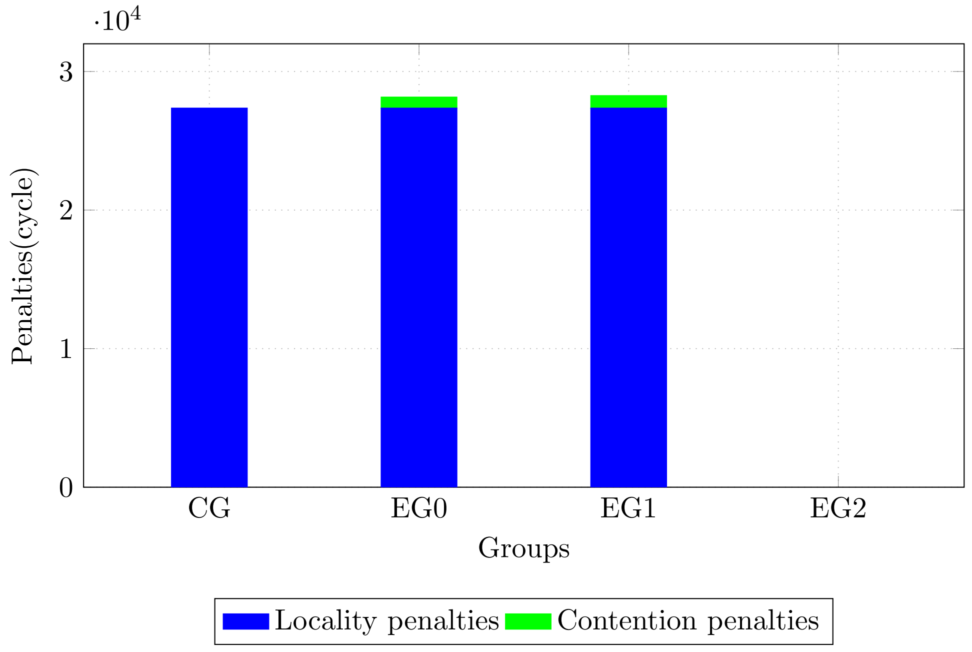


Figure 5.28: EEMBC allocation and contention penalties: Estimated locality penalties (blue) and contention penalties (green) of the different groups.

upper bound of the performance impact. The calculated errors in the tables are very small and can be negligible. By comparing the EGs *EG0*, *EG1* to the optimal group *EG2*, nearly 20% performance impact can be saved by just relocating the global variable from core0 to core1. Thus, locality and contention issues should be considered carefully in multi-core embedded systems.

Figure 5.27 and Figure 5.28 show that the locality penalties strongly dominate the performance impact in this experiment, meaning that the data locality should be optimized preferentially. However, this does not mean that we do not have to care the memory contention, and it only shows the characteristics of this test program and characteristics of Infineon's AURIX TC29x. For this test program, only one global variable, which is related to only a small part of the test program, has potential contention. If more cores and peripherals (e.g. DMA) are involved, the contention can be much higher. From a architecture point of view, for the AURIX TC29x devices, the latency difference between a local memory access and a remote memory access is large, while the scratchpad memory contention penalties are low. The characteristics could be totally different for various applications on distinguished platforms, which on the other side shows the importance of our proposed method. Before performance optimization, those characteristics should be analyzed by this method and be understood by software developers. Then the optimization can be in the right direction.

The proposed contention indicator is based on a statistical method with the assumption (vi) that memory accesses are uniformly distributed in a small time window. Hence, the accuracy is better for a large number of samples. If this assumption is not the case, the probability and the expected penalties can be adjusted accordingly.

6 Conclusions

As the complexity of embedded hardware and software increases, the effort spent on software debugging and performance analysis becomes high. The gap between the developers' understanding and the actual system status becomes wider. One reason is that the bandwidth of information generated on-chip is much higher than the bandwidth available for transferring it out of chip. Worse still, new issues such as shared resource contention, lock contention, and atomicity violations are introduced by multi-core and many-core architectures.

Conventional debug and diagnosis tools are not efficient in handling these challenges for several reasons: First, many conventional tools change the timing behavior of systems, which may jeopardize the debug process. Second, some issues do not have obvious symptoms in the lab environment, and hence usually no debugging or analysis takes place. Third, knowing where to look is extremely challenging in a complex system due to limited observability. A normal debug process works in this way: symptom reported, look into related parts, find the issue. Conventional debugging tools e.g. breakpoints need precise human control to configure the analysis scope, which makes debugging very time-consuming.

The challenge of how to figure out what is going wrong in a complex system is something that doctors in the field of medicine have been dealing for a long time, in terms of the human body and the diseases it faces. Doctors know that attempting to diagnose based only on the superficial symptoms is not a reliable best-practice. A disease like leukemia for example, can exhibit symptoms that are similar to the flu and other common diseases. Only more detailed tests, such as blood tests, will help reveal if it is actually leukemia. So in medicine standardized diagnosis procedures are used to systematically check many points rather than relying on assumptions based on the initial superficial symptoms. As in our example, a blood test is a typical procedure that is very useful for diagnosing many diseases and that can be used for a great number of patients.

6.1 Embedded Health – How to diagnose a complex system?

In this dissertation, a methodology based on the concept of a blood test, incorporates an innovative system diagnosis and debug methodology to automatically detect issues in embedded systems and act as the generally applicable diagnosis tool for a variety of system issues. By making use of hardware tracing of the target system, the proposed methodology detects diseases by using predefined indicators. The predefined indicators are low-level hardware-related behavioral patterns acting as the metric system in a blood test. To deal with different system issues, special indicators are designed for different diagnoses.

For complex hardware configuration issues that are hard to detect and sometimes even without symptoms, LTL is applied to define rules in the SoC development phase. The purpose of using LTL is to define rules in an unambiguous and expressive way. These LTL rules are stored in database and checked against the traced register operations. Moreover, they can be applied to generate unambiguous rules in user manual. Whether they are violated by the running software is considered as indicators of hardware configuration issues. With the help of these indicators, many subtle hardware configuration issues that impact the system functionality and system performance can be detected immediately.

The introduction of multi-core and many-core architectures greatly improves system performance. However, new performance problems such as shared resource contention also come along. In this dissertation, two types of shared resources namely program flash and data memory are investigated as they are critical for system performance. When two cores access the same flash interface, one core has to wait until the other finishes. The access latency of program flash is much larger than normal RAM, causing a great contention penalty. A core cannot run continuously if the instruction fetch is strongly delayed by flash contention. However, this severe issue is hard to notice because it is usually invisible to software developers. In order to cope with this issue, an indicator based on instruction tracing and bus tracing is designed. It detects program flash contention spots by finding the delayed instruction fetch excluding the scenarios with wrong prefetch prediction, unpredictable instruction and fully loaded flash interface. The performance impact is estimated by the proposed indicator. In this way, program flash contention can be detected and avoided by re-scheduling of tasks or re-allocation of program instructions. Compared to program flash contention, the contention penalty of data RAM is not so high but it also greatly impacts system performance considering the frequency of data accesses. The detection of memory contention is not new. In this dissertation, memory contention is not considered alone. Instead, data memory contention together with data locality is calculated and compared because merely detecting the occurrence of remote accesses or memory contention is not sufficient [100][103]. The performance penalties by both effects need to be quantified and be compared to provide a reference for the performance optimization, which has to be kept in balance for both data locality and memory contention effects to achieve the optimum. A statistic indicator is designed to estimate the contention penalty and the performance impact due to both data locality and memory contention is considered. This indicator helps developers to improve system performance by data reallocation.

Performance issues are not the only effect caused by multi-core architecture. Locks as a classic way to regulate accesses to shared data, is commonly applied to multi-core architectures. There are many types of spinlock implementations. However, no lock implementation is the best for all scenarios so lock profilers are useful and help developer to optimize their systems. Usually lock profilers are designed for specific OSs or APIs. For embedded systems, there is no dominating OS like Windows or Linux on computers. Therefore, a method to profile spinlock usage without OS/API knowledge is proposed. This method recognizes spinlocks relying on the characteristics instead of a specific routine or API. Performance issues and mapping issues can be diagnosed by this method.

Many other indicators are also designed for issues related to e.g. DMA, interrupts, clock configurations. Similar to a physical examination, a system is supposed to be checked against all indicators even without any symptoms. Then a report is available for developers to help them to understand their systems.

6.2 Implementation – ChipCoach

ChipCoach was implemented relying on the proposed methodology. It is designed for Infineon's AURIX and AURIX2G. It is non-intrusive, so there is no impact on system timing and no software instrumentation is required. This is especially important for real-time systems as bugs may not be reproducible with a different timing. Many issues which are usually not covered by conventional tools are efficiently detected by ChipCoach. Because the detection methods do not rely on the superficial symptoms but on the essential causes, issues and potential issues without obvious symptoms can be detected in advance. The designed indicators are independent of the application implementation so the tool works effectively for a wide variety of applications, while user-involvement is kept to an absolute minimum. ChipCoach runs on computers and connects to target devices via debug interfaces. It automatically configures the tracing hardware (MCDS) that is integrated into the embedded platform, and decides the scope of tracing. The output is a report that reflects the health status of the target system and guides developers to the root causes of issues. As an internal tool, ChipCoach has been used by many internal application engineers. It is also being commercialized by a tool partner from Infineon.

6.3 Summary and future work

In general, the proposed methodology deals with many issues. It has several advantages over the other solutions. First, it leverages on-chip hardware tracing module to collect trace data and then diagnoses system's health status with the help of designed indicators. Hardware tracing is non-intrusive, which is mandatory for hard real-time applications. Software instrumentation is also unnecessary, which facilitates automated system diagnosis. The whole system can be run while different checks are on-going without disturbing the normal execution. Second, various indicators are proposed and designed for different issues covering both functional and performance issues. These indicators are based on the knowledge of the hardware architectures and basic parameters. The indicator design is independent of applications, meaning that the same indicator is also feasible for another application. No additional input is needed. Third, the target issues are rarely covered or not well solved by existing solutions. Issues e.g. program flash contention are even not detectable with existing tools. The proposed methodology solves these target issues from a totally innovative perspective: root causes at the hardware-operation-level. Finally, this methodology tells developers where they should look at, even for a "healthy" system. It detects the system issues and reports the issues, relying on the root causes instead of the superficial symptoms. It is able to detect issues even before the symptom appears and estimate the severity of the issues. A health status report tells developers the potential functional and performance issues. One step further, suggestions are also raised to solve these issue.

There are still many limitations of the proposed methodology and the implementation. The proposed methodology reports statistics and issues directly to developers. Potentially intelligent judgment could be conducted already by the tool. For example, a performance issue can be ignored if it is really slight. The definition of "slight" is not easy because it is usually different for applications and the tool should be intelligent enough to make this judgment whether it should be reported to developers or just ignored. The proposed methodology is powerful for specific issues but it is not a general solution to replace traditional debuggers. More indicators should be designed to enrich the group of target issues. Machine learning may also be applied to automatically extract indicators for each individual application.

The current implementation does not support continuous tracing, which is able to provide higher tracing capacity. All trace data is temporarily stored in the on-chip trace buffer and the size of this buffer is limited. Therefore, the duration of tracing might be a limiting factor especially for profiling tasks. For example, the interesting issues may not be detected if they are out of the scope. Another limitation is the number of observation points available on chip. It will become more powerful if more types of data can be collected directly from the tracing hardware module.

In the future, the indicator design continues and more special indicators will be available for interesting issues. If possible, machine learning algorithms can be introduced. ChipCoach will continue to be an internal tool. More practical features are supposed to be added to fulfill the actual industrial debugging requirements. Continuous tracing is feasible and might be implemented for ChipCoach.

List of Figures

List of Tables

Acronyms

ADAS	Advanced Driver Assistance Systems.
ADC	Analog-to-Digital Converter.
API	Application Programming Interface.
BIV	Base Interrupt Vector.
BOB	Bus Observation Block.
CCU	Clock Control Unit.
CDIF	Causal DIF.
CEC	Correctly Estimated Contention.
CG	Control Group.
CO	Contention and Others.
COTS	Commercial Off-The-Shelf.
CPI	Cycle Per Instruction.
CPU	Central Processing Unit.
DAP	Device Access Port.
DAS	Device Access Server.
DCACHE	Data Cache.
Dflash	Data Flash.
DIF	Delayed Instruction Fetch.
DMA	Direct Memory Access.
DMC	Debug Memory Controller.
DSPR	Data Scratch Pad RAM.
ECC	Error Correcting Code.
ED	Emulation Device.
EEC	Emulation Extension Chip.
EG	Experimental Group.
ELF	Executable and Linkable Format.
EMEM	Emulation and debug Memory.
GTM	Generic Timer Module.
GUI	Graphical User Interface.
HSSL	High Speed Serial Link.

HW	Hardware.
IBS	Instruction Based Sampling.
ICU	Interrupt Control Unit.
INT	Minimum memory access cycle Interval.
IPC	Instruction Per Cycle.
ISA	Instruction Set Architecture.
ISR	Interrupt Service Routine.
JNI	Java Native Interface.
JTAG	Joint Test Action Group.
LLC	Last Level Cache.
LMU	Local Memory Unit.
LP	Locality Penalty.
LTL	Linear Temporal Logic.
MAC	Media Access Controller.
MCDS	Multi-Core Debug Solution.
MCX	Multi Core Cross-connect.
MPU	Memory Protection Unit.
MRI	Magnetic Resonance Imaging.
MTV	MCDS Trace Viewer.
NUMA	Non-Uniform Memory Access.
OCDS	On-Chip Debug Support.
ODF	Other Delay Factor.
OL	Overload.
OLDA	Online Data Acquisition.
OS	Operating System.
OTGB	OCDS Trigger Bus.
OTGS	OCDS Trigger Switch.
PC	Program Counter.
PCB	Printed Circuit Board.
PCXI	Previous Context Information Register.
PD	Production Device.
PFC	Program Flash Contention.
Pflash	Program Flash.
PFU	Pre-fetch Unit.
PMU	Program Memory Unit.
POB	Processor Observation Block.
PSPR	Program Scratch Pad RAM.

PSW	Program Status Word Register.
QSPI	Queued SPI Controller.
RAM	Random Access Memory.
RCP	Rich Client Platform.
RISC	Reduced Instruction Set Computer.
RTOS	Real-time Operating System.
SCU	System Control Unit.
SD	Service Duration.
SoC	System on Chip.
SPB	System Peripheral Bus.
SPM	Scratchpad Memories.
SRART	Service Request Arbitration Time.
SRAT	Service Request Arriving Time.
SRC	Service Request Control.
SRD	Service Routine Duration.
SRET	Service Routine Ending Time.
SRI	Shared Resource Interconnect.
SRN	Service Request Node.
SRRL	Service Request Response Latency.
SRST	Service Routine Starting Time.
SSC	Special Set of Cores.
STM	System Timer.
SVLCX	Save Lower Context.
SW	Software.
UP	Unpredictable.
WBP	Wrong Branch Prediction.
WCIL	Worst Case Interrupt Latency.

Bibliography

[1] M. Newman. Software errors cost us economy $59.5 billion annually. *NIST Assesses Technical Needs of Industry to Improve Software-Testing*, 2002.

[2] H. Agrawal, R. A. DeMillo, and E. H. Spafford. Efficient debugging with slicing and backtracking. *Software Practice & Experience*, 23(6):589–616, 1993.

[3] Your smartphone is millions of times more powerful than all of nasa's combined computing in 1969. `http://www.zmescience.com/research/technology/smartphone-power-compared-to-apollo-432/`. [Online; accessed 09-June.2017].

[4] A. Mayer, H. Siebert, and K. D. McDonald-Maier. Boosting debugging support for complex systems on chip. *Computer*, 40(4):76–81, 2007.

[5] E. Mitchell. Multi-core and multi-threaded socs present new debugging challenges. *Datasheet MIPS Technologies*, pages 1–6, 2003.

[6] H. Park, J.-Z. Xu, K. H. Kim, and J. S. Park. On-chip debug architecture for multicore processor. *ETRI Journal*, 34(1):44–54, 2012.

[7] Reference ranges for blood tests. `https://en.wikipedia.org/wiki/Reference_ranges_for_blood_tests`. [Online; accessed 17-May.2017].

[8] Heisenbug. `https://en.wikipedia.org/wiki/Heisenbug`. [Online; accessed 11-Jan.2017].

[9] A. Nistor, P.-C. Chang, C. Radoi, and S. Lu. Caramel: Detecting and fixing performance problems that have non-intrusive fixes. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 902–912. IEEE, 2015.

[10] Polyspace. `https://en.wikipedia.org/wiki/Polyspace`. [Online; accessed 21-June.2017].

[11] L. Wang and S. D. Stoller. Static analysis of atomicity for programs with non-blocking synchronization. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 61–71. ACM, 2005.

[12] C. Flanagan, S. N. Freund, and S. Qadeer. Exploiting purity for atomicity. *IEEE Transactions on Software Engineering*, 31(4):275–291, 2005.

[13] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on software Engineering*, 32(3):176–192, 2006.

[14] A. Chou, B. Chelf, D. Engler, and M. Heinrich. Using meta-level compilation to check flash protocol code. *ACM SIGOPS Operating Systems Review*, 34(5):59–70, 2000.

[15] T. Dey, W. Wang, J. W. Davidson, and M. L. Soffa. Characterizing multi-threaded applications based on shared-resource contention. In *(IEEE ISPASS) IEEE International Symposium on Performance Analysis of Systems and Software*, pages 76–86. IEEE, 2011.

[16] A. Kumar, B. Mesman, H. Corporaal, B. Theelen, and Y. Ha. A probabilistic approach to model resource contention for performance estimation of multi-featured media devices. In *2007 44th ACM/IEEE Design Automation Conference*, pages 726–731. IEEE, 2007.

[17] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith. A performance counter architecture for computing accurate cpi components. *ACM SIGOPS Operating Systems Review*, 40(5):175–184, 2006.

[18] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *11th International Symposium on High-Performance Computer Architecture*, pages 340–351. IEEE, 2005.

[19] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. volume 45, pages 129–142, 03 2010. `doi:10.1145/1736020.1736036`.

[20] S. Blagodurov, S. Zhuravlev, and A. Fedorova. Contention-aware scheduling on multicore systems. *ACM Transactions on Computer Systems (TOCS)*, 28(4):8, 2010.

[21] K. K. Pusukuri, R. Gupta, and L. N. Bhuyan. Shuffling: a framework for lock contention aware thread scheduling for multicore multiprocessor systems. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 289–300. ACM, 2014.

[22] Y. Cui, Y. Wang, Y. Chen, and Y. Shi. Lock-contention-aware scheduler: A scalable and energy-efficient method for addressing scalability collapse on multicore systems. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):44, 2013.

[23] L. Zheng, X. Liao, B. He, S. Wu, and H. Jin. On performance debugging of unnecessary lock contentions on multicore processors: A replay-based approach. In *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 56–67. IEEE, 2015.

[24] T. Liu and E. D. Berger. Sheriff: precise detection and automatic mitigation of false sharing. *ACM Sigplan Notices*, 46(10):3–18, 2011.

[25] T. Liu, C. Tian, Z. Hu, and E. D. Berger. Predator: predictive false sharing detection. In *ACM SIGPLAN Notices*, volume 49, pages 3–14. ACM, 2014.

[26] S. Eyerman, K. Du Bois, and L. Eeckhout. Speedup stacks: Identifying scaling bottlenecks in multi-threaded applications. In *2012 IEEE International Symposium on Performance Analysis of Systems & Software*, pages 145–155. IEEE, 2012.

[27] K. Du Bois, S. Eyerman, and L. Eeckhout. Per-thread cycle accounting in multicore processors. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):29, 2013.

[28] C. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff. Statistical debugging: A hypothesis testing-based approach. *IEEE Transactions on software engineering*, 32(10):831–848, 2006.

[29] H. Cleve and A. Zeller. Locating causes of program failures. In *Proceedings of the 27th international conference on Software engineering*, pages 342–351. ACM, 2005.

[30] H. He and N. Gupta. Automated debugging using path-based weakest preconditions. In *International Conference on Fundamental Approaches to Software Engineering*, pages 267–280. Springer, 2004.

[31] C. McCurdy and J. Vetter. Memphis: Finding and fixing numa-related performance problems on multi-core platforms. In *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*, pages 87–96. IEEE, 2010.

[32] A. Schmidt. Profiling bare-metal cores in amp systems. In *System, Software, SoC and Silicon Debug Conference (S4D), 2012*, pages 1–4. IEEE, 2012.

[33] G. Chen and P. Stenstrom. Critical lock analysis: Diagnosing critical section bottlenecks in multithreaded applications. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 71. IEEE Computer Society Press, 2012.

[34] S. Lu, J. Tucek, F. Qin, and Y. Zhou. Avio: detecting atomicity violations via access interleaving invariants. In *ACM SIGOPS Operating Systems Review*, volume 40, pages 37–48. ACM, 2006.

[35] Instrumentation (computer programming). `https://en.wikipedia.org/wiki/Instrumentation_(computer_programming)`. [Online; accessed 28-Nov.-2016].

[36] Q. Chen, L. Wang, Z. Yang, and S. D. Stoller. Have: Detecting atomicity violations via integrated dynamic and static analysis. In *International Conference on Fundamental Approaches to Software Engineering*, pages 425–439. Springer, 2009.

[37] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *ACM Sigplan Notices*, volume 40, pages 190–200. ACM, 2005.

[38] P. Zhou, R. Teodorescu, and Y. Zhou. Hard: Hardware-assisted lockset-based race detection. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 121–132. IEEE, 2007.

[39] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema, and M. Roth. Traffic management: a holistic approach to memory placement on numa systems. In *ACM SIGPLAN Notices*, volume 48, pages 381–394. ACM, 2013.

[40] Y. Huang, Z. Cui, L. Chen, W. Zhang, Y. Bao, and M. Chen. Halock: hardware-assisted lock contention detection in multithreaded applications. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 253–262. ACM, 2012.

[41] S. Lagraa, A. Termier, and F. Pétrot. Data mining mpsoc simulation traces to identify concurrent memory access patterns. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 755–760. EDA Consortium, 2013.

[42] C. Killian, K. Nagaraj, S. Pervez, R. Braud, J. W. Anderson, and R. Jhala. Finding latent performance bugs in systems implementations. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 17–26. ACM, 2010.

[43] M. Martonosi, A. Gupta, and T. Anderson. Memspy: Analyzing memory system bottlenecks in programs. In *ACM SIGMETRICS Performance Evaluation Review*, volume 20, pages 1–12. ACM, 1992.

[44] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)*, 15(4):391–411, 1997.

[45] S. Lu, J. Tucek, F. Qin, and Y. Zhou. Avio: detecting atomicity violations via access interleaving invariants. In *ACM SIGOPS Operating Systems Review*, volume 40, pages 37–48. ACM, 2006.

[46] D. Jeffrey, M. Feng, N. Gupta, and R. Gupta. Bugfix: A learning-based tool to assist developers in fixing bugs. In *Program Comprehension, 2009. ICPC'09. IEEE 17th International Conference on*, pages 70–79. IEEE, 2009.

[47] L. Li and A. Mayer. Trace-based analysis methodology of program flash contention in embedded multicore systems. In *Proceedings of the 2016 Conference on Design, Automation & Test in Europe*, pages 199–204. EDA Consortium, 2016.

[48] L. Li, M. Fussenegger, and G. Cichon. A data locality and memory contention analysis method in embedded numa multi-core systems. In *IEEE 10th International Symposium on Embedded MulticoreMany-core Systems-on-Chip (MCSoC-16)*, pages 85–92. IEEE, 2016.

[49] L. Li, P. Wagner, R. Ramaswamy, A. Mayer, T. Wild, and A. Herkersdorf. A rule-based methodology for hardware configuration validation in embedded systems. In *Proceedings of the 19th International Workshop on Software and Compilers for Embedded Systems*, pages 180–189. ACM, 2016.

[50] L. Li, P. Wagner, A. Mayer, T. Wild, and A. Herkersdorf. A non-intrusive, operating system independent spinlock profiler for embedded multicore systems. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pages 322–325. IEEE, 2017.

[51] J. Zhang, Y. Dong, and J. Duan. Anole: a profiling-driven adaptive lock waiter detection scheme for efficient mp-guest scheduling. In *2012 IEEE International Conference on Cluster Computing*, pages 504–513. IEEE, 2012.

[52] J. Demme and S. Sethumadhavan. Rapid identification of architectural bottlenecks via precise event counting. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 353–364. ACM, 2011.

[53] C. Flanagan, C. Flanagan, and S. N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *ACM SIGPLAN Notices*, volume 39, pages 256–267. ACM, 2004.

[54] A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57. IEEE, 1977.

[55] P. Thiagarajan. A trace based extension of linear time temporal logic. In *Logic in Computer Science, 1994. LICS'94. Proceedings., Symposium on*, pages 438–447. IEEE, 1994.

[56] E. Gunter and D. Peled. Temporal debugging for concurrent systems. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 431–444. Springer, 2002.

[57] C. Watterson and D. Heffernan. Runtime verification and monitoring of embedded systems. *Software, IET*, 1(5):172–179, 2007.

[58] J. C. Lee, A. S. Gardner, and R. Lysecky. Hardware Observability Framework for Minimally Intrusive Online Monitoring of Embedded Systems. In *Engineering of Computer Based Systems (ECBS), 2011 18th IEEE International Conference and Workshops on*, pages 52–60. IEEE, 2011.

[59] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 44–57. Springer, 2004.

[60] H. Barringer, D. Rydeheard, and K. Havelund. Runtime Verification: 7th International Workshop, RV 2007, Vancover, Canada, March 13, 2007, Revised Selected Papers. pages 111–125. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[61] R. Pellizzoni, P. Meredith, M. Caccamo, and G. Rosu. Hardware runtime monitoring for dependable cots-based real-time embedded systems. In *Real-Time Systems Symposium, 2008*, pages 481–491. IEEE, 2008.

[62] M.-W. Youssef, S. Yoo, A. Sasongko, Y. Paviot, and A. A. Jerraya. Debugging HW/SW interface for MPSoC: video encoder system design case study. In *Proceedings of the 41st annual Design Automation Conference*, pages 908–913. ACM, 2004.

[63] K. Peterson and Y. Savaria. Assertion-based on-line verification and debug environment for complex hardware systems. In *Circuits and Systems, 2004. ISCAS'04. Proceedings of the 2004 International Symposium on*, volume 2, pages II–685. IEEE, 2004.

[64] T. E. Anderson. The performance of spin lock alternatives for shared-money multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, 1990.

[65] L. Rudolph and Z. Segall. *Dynamic decentralized cache schemes for MIMD parallel processors*, volume 12. ACM, 1984.

[66] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9(1):21–65, 1991.

[67] P. H. Ha, M. Papatriantafilou, and P. Tsigas. Reactive spin-locks: A self-tuning approach. In *8th International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN'05)*, pages 6–pp. IEEE, 2005.

[68] H. Guiroux, R. Lachaize, and V. Quéma. Multicore locks: The case is not closed yet. In *USENIX Annual Technical Conference*, pages 649–662, 2016.

[69] H. Shojania. Hardware-based performance monitoring with vtune performance analyzer under linux.

[70] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. Hpctoolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, 2010.

[71] J. Mellor-Crummey, L. Adhianto, M. Fagan, M. Krentel, and N. Tallent. Hpctoolkit user's manual.

[72] R. Bryant and J. Hawkes. Lockmeter: Highly-informative instrumentation for spin locks in the linux kernel. In *Proc. Fourth Annual Linux Showcase and Conference, Atlanta*, 2000.

[73] Z. Benavides, R. Gupta, and X. Zhang. Parallel execution profiles. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, pages 215–218. ACM, 2016.

[74] F. David, G. Thomas, J. Lawall, and G. Muller. Continuously measuring critical section pressure with the free-lunch profiler. In *ACM SIGPLAN Notices*, volume 49, pages 291–307. ACM, 2014.

[75] Profiling java.util.concurrent locks. `https://www.infoq.com/articles/jucprofiler`, 2010. [Online; accessed 31-July-2016].

[76] J.-P. Lozi, F. David, G. Thomas, J. Lawall, and G. Muller. Fast and portable locking for multicore architectures. *ACM Transactions on Computer Systems (TOCS)*, 33(4):13, 2016.

[77] X. Pan, D. Klaftenegger, and B. Jonsson. Forecasting lock contention before adopting another lock algorithm, 2015.

[78] X. Feng, Z. Shao, Y. Dong, and Y. Guo. Certifying low-level programs with hardware interrupts and preemptive threads. In *ACM SIGPLAN Notices*, volume 43, pages 170–182. ACM, 2008.

[79] D. Brylow, N. Damgaard, and J. Palsberg. Static checking of interrupt-driven software. In *Proceedings of the 23rd international conference on software engineering*, pages 47–56. IEEE Computer Society, 2001.

[80] M. Carlsson. Worst case execution time analysis, case study on interrupt latency for the ose realtime operating system. *Master's Thesis in Electrical Engineering, Royal Institute of Technology, Stockholm, Sweden*, 3:18, 2002.

[81] B. Moore, T. Slabach, and L. Schaelicke. Profiling interrupt handler performance through kernel instrumentation. In *Computer Design, 2003. Proceedings. 21st International Conference on*, pages 156–163. IEEE, 2003.

[82] T. Yu, W. Srisa-an, M. B. Cohen, and G. Rothermel. Simlatte: A framework to support testing for worst-case interrupt latencies in embedded software. In *Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on*, pages 313–322. IEEE, 2014.

[83] K. Lu, D. Müller-Gritschneder, and U. Schlichtmann. Analytical timing estimation for temporally decoupled tlms considering resource conflicts. In *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1161–1166. IEEE, 2013.

[84] K. Lampka, G. Giannopoulou, R. Pellizzoni, Z. Wu, and N. Stoimenov. A formal approach to the wcrt analysis of multicore systems with memory contention under phase-structured task sets. *Real-Time Systems*, 50(5-6):736–773, 2014.

[85] R. Hood, H. Jin, P. Mehrotra, J. Chang, J. Djomehri, S. Gavali, D. Jespersen, K. Taylor, and R. Biswas. Performance impact of resource contention in multicore systems. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–12. IEEE, 2010.

[86] R. Pellizzoni, A. Schranzhofer, J.-J. Chen, M. Caccamo, and L. Thiele. Worst case delay analysis for memory interference in multicore systems. In *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*, pages 741–746. IEEE, 2010.

[87] L. Liu, Z. Li, and A. H. Sameh. Analyzing memory access intensity in parallel programs on multicore. In *Proceedings of the 22nd annual international conference on Supercomputing*, pages 359–367. ACM, 2008.

[88] F. Pinel, J. E. Pecero, P. Bouvry, and S. U. Khan. Memory-aware green scheduling on multi-core processors. In *2010 39th International Conference on Parallel Processing Workshops*, pages 485–488. IEEE, 2010.

[89] C. Xu, X. Chen, R. P. Dick, and Z. M. Mao. Cache contention and application performance prediction for multi-core systems. In *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*, pages 76–86. IEEE, 2010.

[90] S. Chattopadhyay, A. Roychoudhury, and T. Mitra. Modeling shared cache and bus in multi-cores for timing analysis. In *Proceedings of the 13th international workshop on software & compilers for embedded systems*, page 6. ACM, 2010.

[91] A. Fedorova, M. Seltzer, and M. D. Smith. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 25–38. IEEE Computer Society, 2007.

[92] L. R. Hsu, S. K. Reinhardt, R. Iyer, and S. Makineni. Communist, utilitarian, and capitalist cache policies on cmps: caches as a shared resource. In *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pages 13–22. ACM, 2006.

[93] Z. Majo and T. R. Gross. Memory management in numa multicore systems: trapped between cache contention and interconnect overhead. In *Acm Sigplan Notices*, volume 46, pages 11–20. ACM, 2011.

[94] R. P. LaRowe Jr and C. S. Ellis. Experimental comparison of memory management policies for numa multiprocessors. *ACM Transactions on Computer Systems*, 9(4):319–363, 1991.

[95] D. Cousins, J. Loomis, F. Roeber, P. Schoeppner, and A.-E. Tobin. The embedded genetic allocator-a system to automatically optimize the use of memory resources in high performance, scalable computing systems. In *SMC'98 Conference Proceedings. 1998 IEEE International Conference on Systems, Man, and Cybernetics (Cat. No. 98CH36218)*, volume 3, pages 2166–2171. IEEE, 1998.

[96] A. Marongiu and L. Benini. An openmp compiler for efficient use of distributed scratchpad memory in mpsocs. *IEEE Transactions on Computers*, 61(2):222–236, 2010.

[97] M. L. Chu and S. A. Mahlke. Compiler-directed data partitioning for multicluster processors. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 208–220. IEEE Computer Society, 2006.

[98] M. Diener, E. H. Cruz, P. O. Navaux, A. Busse, and H.-U. Heiß. kmaf: automatic kernel-level management of thread and data affinity. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 277–288. ACM, 2014.

[99] E. H. M. da Cruz, M. A. Z. Alves, A. Carissimi, P. O. A. Navaux, C. P. Ribeiro, and J.-F. Méhaut. Using memory access traces to map threads and data on hierarchical multi-core platforms. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 551–558. IEEE, 2011.

[100] S. Blagodurov, S. Zhuravlev, M. Dashti, and A. Fedorova. A case for numa-aware contention management on multicore systems. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, pages 1–1, 2011.

[101] Y. Xie and G. Loh. Dynamic classification of program memory behaviors in cmps. In *the 2nd Workshop on Chip Multiprocessor Memory Systems and Interconnects*, 2008.

[102] J. Jalle, M. Fernandez, J. Abella, J. Andersson, M. Patte, L. Fossati, M. Zulianello, and F. J. Cazorla. Bounding resource contention interference in the next-generation microprocessor (ngmp). 2016.

[103] Z. Majo and T. R. Gross. Memory system performance in a numa multicore multiprocessor. In *Proceedings of the 4th Annual International Conference on Systems and Storage*. ACM, 2011.

[104] Infineon tricore. `https://en.wikipedia.org/wiki/Infineon_TriCore`. [Online; accessed 12-April.2017].

[105] Infineon. *TriCore TC1.6P & TC1.6E Core Architecture 32-bit Unified Processor Core.*

[106] Infineon Technologies. *AURIX TC29x B-step User's Manual.*

[107] G. Macher, A. Höller, E. Armengaud, and C. Kreiner. Automotive embedded software: Migration challenges to multi-core computing platforms. In *2015 IEEE 13th International Conference on Industrial Informatics (INDIN)*, pages 1386–1393. IEEE, 2015.

[108] Infineon Technologies. *AURIX TC29x A-step User's Manual.*

[109] Infineon Technologies. *Application Note: AURIX, TriCore, XC2000, XE166,XC800 Families DAP Connector.*

[110] Infineon Technologies. *AURIX TC29/7/6/3x ED Target Secification.*

[111] A. Mayer and R. Deml. Compact function trace (cft). In *Proceedings of the 2012 System, Software, SoC and Silicon Debug Conference*, pages 1–2, Sep. 2012.

[112] A. Mayer. A seamless tool access architecture from esl to end product. In *System, Software, SoC and Silicon Debug Conference (S4D)*, 2009.

[113] B. Hailpern and P. Santhanam. Software debugging, testing, and verification. *IBM Systems Journal*, 41(1):4–12, 2002.

[114] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, pages 291–301. IEEE, 2002.

[115] A. Mayer. Embedded health the step beyond debugging. In *Multicore Application Debugging (MAD) Workshop 2013*, 2013.

[116] P. Wagner, L. Li, T. Wild, A. Mayer, and A. Herkersdorf. Knowledge-based on-chip diagnosis for multi-core systems-on-chip. In *edaWorkshop 15*, pages 39–45, 2015.

[117] Software bug. `https://en.wikipedia.org/wiki/Software_bug`. [Online; accessed 07-Feb.2017].

[118] G. De Giacomo and M. Y. Vardi. Linear temporal logic and linear dynamic logic on finite traces. In *Proceedings of the Twenty-Third international joint conference on Artificial Intelligence*, pages 854–860. AAAI Press, 2013.

[119] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *Proceedings of the tenth international symposium on Hardware/software codesign*, pages 73–78. ACM, 2002.

[120] S. Saidi and Y. Falcone. Dynamic detection and mitigation of dma races in mpsocs. In *Digital System Design (DSD), 2015 Euromicro Conference on*, pages 267–270. IEEE, 2015.

[121] A. F. Donaldson, D. Kroening, and P. Rümmer. Scratch: a tool for automatic analysis of dma races. In *ACM SIGPLAN Notices*, volume 46, pages 311–312. ACM, 2011.

[122] A. F. Donaldson, D. Kroening, and P. Rümmer. Automatic analysis of dma races using model checking and k-induction. *Formal Methods in System Design*, 39(1):83–113, 2011.

[123] L. Ionkov, A. Nyrhinen, and A. Mirtchovski. Cellfs: Taking the "dma" out of cell programming. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–8. IEEE, 2009.

[124] Interrupt. `https://en.wikipedia.org/wiki/Interrupt`. [Online; accessed 05-April.2017].

[125] A. Wieder and B. B. Brandenburg. On spin locks in autosar: Blocking analysis of fifo, unordered, and priority-ordered spin locks. In *2013 IEEE 34th Real-Time Systems Symposium*, pages 45–56. IEEE, 2013.

[126] Instruction pipelining. `https://en.wikipedia.org/wiki/Instruction_pipelining`. [Online; accessed 03-April.2017].

[127] Dwarf. `https://en.wikipedia.org/wiki/DWARF`. [Online; accessed 01-June.2017].

[128] Wikipedia. Dhrystone. `https://en.wikipedia.org/wiki/Dhrystone`. [Online; accessed 20-August-2015].