Real-Time Trace Decoding and Monitoring for Safety and Security in Embedded Systems

Zur Erlangung des akademischen Grades eines

DOKTOR-INGENIEURS

von der KIT-Fakultät für
Elektrotechnik und Informationstechnik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

DISSERTATION

von

M.Eng. Augusto Wankler Hoppe
geb. in Porto Alegre

Tag der mündlichen Prüfung: 14.12.2021

Hauptreferent: Prof. Dr.-Ing. Dr.h.c Jürgen Becker
Korreferent: Prof. Dr. Fernanda Kastensmidt

Abstract

Integrated circuits and systems can be found almost everywhere in today's world. As their use increases, they need to be made safer and more performant to meet current demands in processing power. FPGA integrated SoCs can provide the ideal trade-off between performance, adaptability, and energy usage. One of today's vital challenges lies in updating existing fault tolerance techniques for these new systems while utilizing all available processing capabilities, such as multi-core and heterogeneous processing units. Control-flow monitoring is one of the primary mechanisms described for error detection at the software architectural level for the highest grade of hazard level classifications (e.g., ASIL D) described in industry safety standards ISO-26262. Control-flow errors are also known to compose the majority of detected errors for ICs and embedded systems in safety-critical and risk-susceptible environments [5].

Software-based monitoring methods remain the most popular [6–8]. However, recent studies show that the overheads they impose make actual reliability gains negligible [9, 10]. This work proposes and demonstrates a new control-flow checking method implemented in FPGA for multi-core embedded systems called control-flow trace checker (CFTC). CFTC uses existing trace and debug subsystems of modern processors to rebuild their execution states. It can identify any errors in real-time by comparing executed states to a set of permitted state transitions determined statically.

This novel implementation weighs hardware resource trade-offs to target multiple independent tasks in multi-core embedded applications, as well as single-core systems. The proposed system is entirely implemented in hardware and isolated from all monitored software components, requiring 2.4% of the target FPGA platform resources to protect an execution unit in its entirety. Therefore, it avoids undesired overheads and maintains deterministic error detection latencies, which guarantees reliability improvements without impairing the target software system. Finally, CFTC is evaluated under different software

fault-injection scenarios, achieving detection rates of 100% of all control-flow errors to wrong destinations and 98% of all injected faults to program binaries. All detection times are further analyzed and precisely described by a model based on the monitor's resources and speed and the software application's control-flow structure and binary characteristics.

Resumo

Circuitos integrados estão presentes em quase todos sistemas complexos do mundo moderno. Conforme sua frequência de uso aumenta, eles precisam se tornar mais seguros e performantes para conseguir atender as novas demandas em potência de processamento. Sistemas em Chip integrados com FPGAs conseguem prover o balanço perfeito entre desempenho, adaptabilidade, e uso de energia. Um dos maiores desafios agora é a necessidade de atualizar técnicas de tolerância à falhas para estes novos sistemas, aproveitando os novos avanços em capacidade de processamento. Monitoramento de fluxo de controle é um dos principais mecanismos para a detecção de erros em nível de software para sistemas classificados como de alto risco (e.g. ASIL D), descrito em padrões de segurança como o ISO-26262. Estes erros são conhecidos por compor a maioria dos erros detectados em sistemas integrados [5].

Embora métodos de monitoramento baseados em software continuem sendo os mais populares [6–8], estudos recentes mostram que seus custos adicionais, em termos de performance e área, diminuem consideravelmente seus ganhos reais em confiabilidade [9, 10]. Propomos aqui um novo método de monitoramento de fluxo de controle implementado em FPGA para sistemas embarcados multi-core. Este método usa subsistemas de trace e execução de código para reconstruir o estado atual do processador, identificando erros através de comparações entre diferentes estados de execução da CPU.

Propomos uma implementação que considera trade-offs no uso de recuros de sistema para monitorar múltiplas tarefas independetes. Nossa abordagem suporta o monitoramento de sistemas simples e também de sistemas multi-core multitarefa. Por fim, nossa técnica é totalmente implementada em hardware, evitando o uso de unidades de processamento de software que possa adicionar custos indesejáveis à aplicação em perda de confiabilidade. Propomos, assim, um mecanismo de verificação de fluxo de controle, escalável e extensível, para proteção de sistemas embarcados críticos e multi-core.

Zusammenfassung

Integrierte Schaltkreise sind heutzutage fast überall zu finden. Mit zunehmender Nutzung müssen sie sicherer und leistungsfähiger gemacht werden, um den aktuellen Anforderungen an die Rechenleistung gerecht zu werden. FPGA-integrierte SoCs bieten den perfekten Kompromiss zwischen Leistung, Anpassungsfähigkeit und Energieverbrauch. Eine der großen aktuellen Herausforderungen besteht darin, die vorhandenen Fehlertoleranz Techniken für diese neuen Systeme zu aktualisieren. Die Überwachung des Kontrolle Flusses ist einer der Hauptmechanismen für die Softwarefehlererkennung auf der höchsten Gefahrenstufe (z.B. ASIL D), wie in ISO-26262 beschrieben. Es ist auch bekannt, dass Kontrollflussfehler die Mehrheit der erkannten Fehler für eingebettete Systeme in sicherheitskritischen Umgebungen ausmachen [5].

Obwohl softwarebasierte Überwachungsmethoden am beliebtesten sind [6–8], neuere Studien zeigen, dass die tatsächlichen Zuverlässigkeitsgewinne geringer sind als erwartet [9, 10]. Wir schlagen eine neue FPGA-basiert Kontrollfluss-Überwachungsmethode vor, für eingebettete Mehrkernsysteme. Diese Methode verwendet vorhandene Trace- und Debug-Subsysteme moderner Prozessoren, um den aktuellen Status des Prozessors festtellen. Dabei werden etwaige Fehler identifiziert, indem ein geänderter CPU-Status mit einer statisch bestimmten Menge zulässiger Statusübergänge verglichen wird.

Wir schlagen eine Implementierung vor, die Ressource Kompromisse berücksichtigt, um mehrere unabhängige Aufgaben in einer insgesamt größeren eingebetteten Anwendung abzudecken. Unser Ansatz unterstützt die Überwachung nicht nur von Single-Task- und Single-Core-Anwendungen, sondern auch von Multi-Tasking- und Multi-Core-Systemen. Schließlich ist unser vorgeschlagenes System vollständig in Hardware implementiert, wodurch die Verwendung zusätzlicher Verarbeitungseinheiten vermieden wird, die unerwünschte Kosten in Bezug auf die Fläche, Erkennungslatenzen oder eine verringerte Zuverlässigkeit verursachen können. Wir schlagen daher einen vollständig

skalierbaren und erweiterbaren Kontrollflussprüfungsmechanismus für mehrkernige sicherheitskritische eingebettete Systeme vor.

Acknowledgements

I want to thank my family for all the support and love they gave throughout these PhD years. My parents Martha and Marco, for their guidance in my moments of doubt. My sister Mariana and my nieces Sophia and Michelle, for the joy and vacations among family in the short moments I was able to visit. Even far away, our connection only grows stronger.

I would also like to thank my advisors, Professor Fernanda Kastensmidt and Professor Jürgen Becker, for believing in me and my research. I started this doctorate young and with little understanding of all that it encompassed. Prof. Kastensmidt was essential in helping put my plans and ideas in place and in helping me find my research path. I want to give my gratitude to Prof. Becker for welcoming me to Karlsruhe and ITIV. I want to give my thanks for the many research and project opportunities inside - and outside - of KIT and for providing me with a great work environment.

I want to thank my research colleagues and friends in both Brazil and Germany for all the hours and discussions that helped me in this research and in all the projects we shared. To everyone at ITIV that welcomed me and helped me with the ins and outs of this new life in Germany, for the discussions at barbecues and coffees, at Vogel or the cinema. To my D&D group, Nidhi, Simon, Gabriela, Florian, and Tim. To Steffen for geeking out with me about films and games. To my colleagues at the PES lab. Finally, to all my new and old friends, in Porto Alegre and Karlsruhe, who made this period all the better.

To all of you, my sincere thanks.

Karlsruhe, December 2021
Augusto Hoppe

Table of Contents

1 Introduction

Fault tolerance has been an important topic of interest since the beginning of computer history and long before the widespread use of personal computers. It has been a major research point in military and industrial-grade systems. The first ever computer systems had to perform exact calculations for military applications and space missions. Today, we are confronted with electronics and embedded systems in almost all aspects of our lives. These play an increasingly important role and accompany us through our everyday lives, be it in radios, electric toothbrushes, smartphones, or automotive and mobility systems. Processors and other integrated circuits have significantly improved their performance in the last couple of decades due to significant advances in the semiconductor industry. These advances have led to the fabrication of high-density integrated circuits (ICs) that form the base of many of the technologies we use today. Technology and software-based systems drive current innovation, and current systems are being operated at their performance limit and cannot meet all requirements satisfactorily.

Even today, a system's ability to cope with faults and continue its correct operation is a central concern. At the same time, the semiconductor industry provides increasingly high-performance computing units with ever-smaller structure sizes. For a long time, manufacturing advancements followed Moore's law, which states that the number of components in a chip doubles annually, later re-stated as a doubling every two years. However, the integration density of transistors and achievable clock frequencies increasingly approach the physical and thermal limits of a transistor's gate [11, 12], with current technology feature sizes that span the width of tens of atoms. This progress has also reduced the transistor's reliability by reducing threshold voltages, node capacitance, and tightening signal to noise margins [13].

Figure 1.1 shows how, since the beginning of the last decade, clock speeds plateaued while device integration kept increasing. The new way forward is device specialization and multiple execution units. To further increase the

computing power of processors, it is no longer possible to increase the clock frequency. Increasingly, manufacturers are switching to processors that integrate more than just one computing core and are operated with the same or even lower clock frequencies, as can also be seen in Intel processors' development (Fig. 1.1). The last decade saw further miniaturization developments, with Samsung and TSMC starting volume production of 5 nm chips in 2020 [14]. However, it is visible that we will soon reach a limit on transistor miniaturization.

It is not just the failure to replicate components within a multi-core processor a limitation. The application's characteristics and domain also limit the increase in performance. One of the first theoretical formulas to predict the speedup of an application when parallel processing resources are introduced comes from Amdahl [16]. Amdahl evaluated the actual increases in performance given

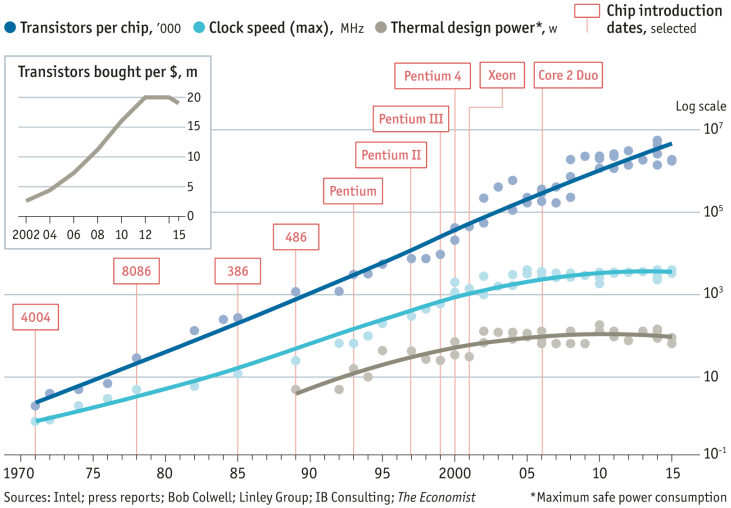


Figure 1.1: Processor development by Intel [15].

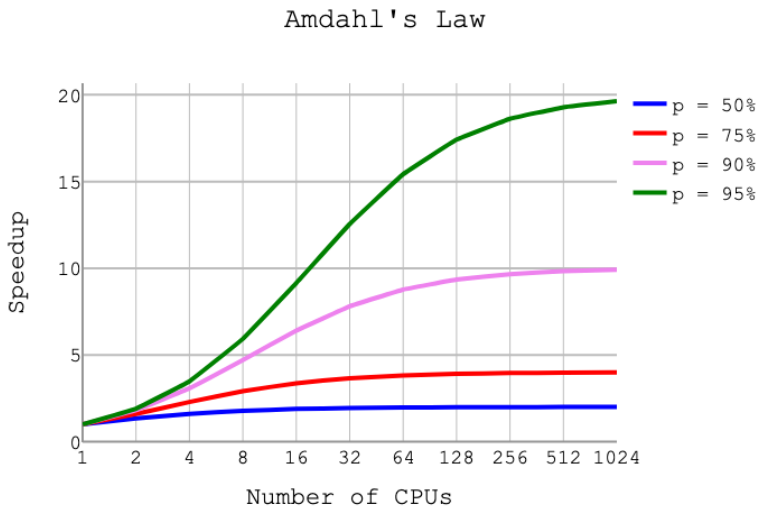


Figure 1.2: Speedup from Amdahl's Law.

available computing cores and the application's inherent parallelism (Fig. 1.2). The formula for this speedup (S), also called Amdahl's Law, can be stated as:

$$S = \frac{T_{seq}}{T_{par}} = \frac{s+p}{(s+p/N)} = \frac{1}{((1-p)+p/N)} \tag{1.1}$$

Where T_{seq} and T_{par} are the execution times before and after parallelization. s and p are the fractions of time an application spends in uniquely sequential and parallel tasks, respectively. These percentages add up to one ($s+p=1$). N is the number of parallel resources used for the speedup application or the number of processors used for parallel software applications. The lack of performance increase is justified by the resulting increase in communication and synchronization requirements between cores. Different processing units must coordinate the use of different component resources and intermediary results. Amdahl's law might appear pessimistic, but it makes the critical assumption that the parallel portion of the application stays constant, which is not necessarily true.

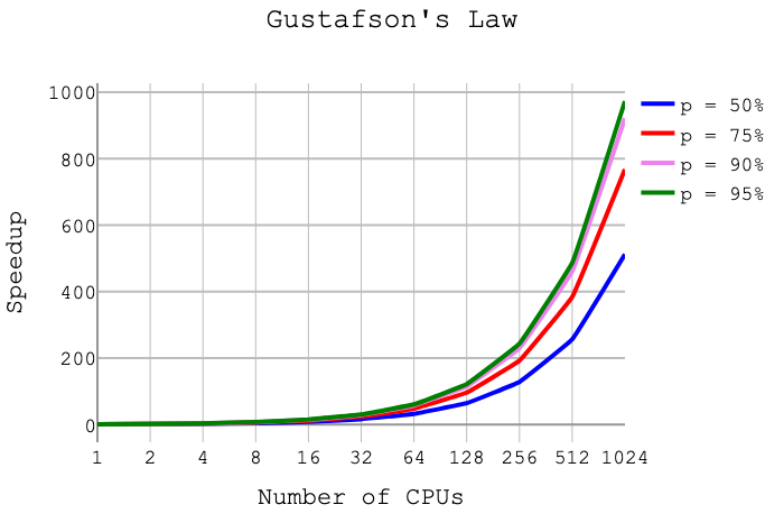


Figure 1.3: Speedup given by Gustafson's Law.

With this in mind, Gustafson proposed a revision of this formula [17]. His new scaled speedup equation, now known as Gustafson's Law, is shown in Fig. 1.3 and is expressed as:

$$S = \frac{T_{seq}}{T_{par}} = \frac{s + p * N}{(s + p)} = s + p * N \tag{1.2}$$

This new formulation assumes an increase in parallel execution proportionate to the increase in additional parallel units, i.e., additional parallel units enable us to process more data so the application's workload profile changes.

We often observe this in real-world scenarios. As parallelism is introduced, the workload of the system also increases. In essence, as new processing units are made available, designers and engineers can afford to process more data simultaneously and find new applications and use cases that were impractical before. This effect is visible in the automotive domain, as more sensors are added to vehicles and functionalities are integrated and combined into a smaller number of electronic control units (ECUs).

Real speedups are constrained by both models, as the introduction of more computing power and resources also increases the serial overhead of computations. Nonetheless, multi-core processor technology is seen as the principal technology available to meet the increasing performance requirements.

1.1 Motivations and Problem Statement

The advances in processor technology observed in recent years show a clear trend towards multi-core processors. These architectures are already widespread in personal, information, and entertainment domains. They show performance improvements and an increased digital computing power requirement due to their highly parallel software execution. Concurrently, power demands are increasing for embedded systems in mobility domains (e.g., automotive, rail, and aviation). Together with all this, diverse and often much stricter requirements must be met in these systems.

These demands result from the fact that such systems typically perform safety-critical functions, such as control devices in vehicles (e.g., ABS, ESP), aircrafts (e.g., flight controls, engine controls), or railways. In this context, a safety-critical system is understood as one where the failure or deviation of one of its functions leads to considerable danger to human lives or economic damage. For this purpose, norms and standards are provided for mobility domains that must be considered during their development. The strict requirements often result from the relevant standards for developing electronic components and systems in their respective domains.

The increased demands on computing power in mobility domains can be attributed to new functionalities and the integration of multiple different functions in one control device. In many domains, these attempts for more functionality are often characterized by increasingly complex software-based systems to boost product quality and competitiveness. Due to the non-functional requirements (Space, Weight, and Power (SWaP)), it is impossible to implement this on multiple control units based on processors with low computing power. In the automotive industry in particular, a further trend towards ever more powerful domain control computers can be observed, in which all functionality within a vehicle domain (e.g., chassis, power train, body) is integrated into a smaller set of ECUs [18]. Aviation applications also present similar

changes. Functions traditionally distributed over several control units, with each possessing a dedicated processor, now share a common hardware platform (including execution and memory units). Since SWaP requirements are typically very high in mobility domain, they present the highest efforts to reduce the number of control units through integration.

These critical applications demand fault-tolerant and detection techniques with minimum implementation and performance overheads. Fault tolerance techniques based on software rely on using additional instructions to the original program code to detect or correct faults [19]. They may be able to detect faults that affect the data and the control flows. Software-based techniques provide high flexibility, short development time, and lower cost since there is no need to modify the hardware. Furthermore, new generations of microprocessors can be used that do not implement Radiation Hardened (RadHard) technologies. As a result, aerospace applications can use commercial off-the-shelf (COTS) microprocessors with RadHard software. However, some results from random fault injection have shown the impossibility of achieving complete fault coverage when using only software-based techniques due to an increase in control-flow errors [20]. This limitation is due to the software's inability to protect against all possible control-flow errors in the microprocessor. One example would be a fault in the program counter (PC) that causes the processing unit to only execute no-operation (NOP) instructions. In such a case, it does not matter how many safety instructions or data are added to the original code because the processor will not execute them, and such a failure will not be detected.

Due to the redundant instructions inserted in the original program code, software-based techniques have drawbacks and overheads in data and program memory. They also present degraded performance due to the increased computation time required to execute the program. The program memory increases due to additional instructions inserted in the original program code, while the data memory may increase due to replication of variables and structures, which can be, for example, the replication of all stored data in the working memory. Performance degradation comes from the execution of redundant instructions. These overheads have the propensity to significantly increase the overall system's vulnerability. These nullifies the actual reliability gains from such techniques, and in some cases, even makes the software where these techniques are applied to less reliable than before [9, 10].

Hardware-based techniques usually change the original processor architecture by adding redundant logic, such as module replication with majority voters, information redundancy, error correction codes (ECC), or time redundancy. Peripheral monitoring devices added to the system's architecture, such as a verification hardware module, are also less invasive ways to implement such techniques. They change the system's architecture, but not the processor's architecture, making them less or non-intrusive. Classic hardware-based techniques such as lockstep and triple-modular redundancy (TMR) may increase the processor's area around two or three times its original size, which leads to more power consumption and production costs. Nevertheless, non-intrusive hardware modules, such as watchdogs, can have a more negligible impact on the area and operating frequencies, but they require special access to system interfaces.

Control-flow monitoring has also experienced a resurgence in interest from an embedded security perspective through attack mitigation techniques known as control-flow integrity (CFI). A Microsoft security report suggested that more than 90% of recent exploits use code-reuse attacks (CRA) [21]. In these exploits, an attacker changes the application's control-flow through existing code with a malicious result, posing a high risk to the system's integrity. Constraining control-flow for security purposes is not new. For example, computer hardware has long been able to prevent code execution from data memory regions, and the latest x86-64 and armv8 processors support this feature. At the software level, several existing mitigation techniques restrain control-flow in some way, e.g., by checking stack integrity and validating function returns [22, 23], or by encrypting function pointers [24].

Control-flow errors (CFEs) are also known to compose most detected errors for ICs and embedded systems in safety-critical and risk-susceptible environments [5]. Even in modern real-time central processing unit (CPU) architectures, only 10% of a processor's elements are responsible for more than 70% of detected errors at its outputs [25]. Even after years of developments in the field, the detection of CFEs in embedded systems is still a challenge that must be overcome. New hardware-based control-flow monitor techniques are exploring the use of non-invasive debug and trace interfaces. Despite this, these investigations do not address how such techniques translate across different processor architectures.

Most trace interfaces and protocols implement high data compression rates. The aim of such interfaces is to transmit data as fast as possible through restricted serial-wire debug (SWD) or JTAG connections, or to store the program trace data in an intermediary buffer memory. Besides the overheads to decode this information, most protocols also assume that trace analyzers have information about or access to the running binary application. Therefore, we must understand the relationship between high-level software components such as core synchronization functions, exception handling, and the binary interfaces used in their implementation.

There is a clear trend in the increased use of multi-core processors in mobility domains to fulfill higher computing power demands. However, the architectures inherent to a multi-core processor result in various challenges that must be mastered for safe and reliable use when executing a safety-critical function. To achieve and ensure these requirements, new methods, concepts, and patterns must be explored that enable error detection and avoidance while minimizing impacts on overheads.

1.2 Objectives and Contributions

This thesis performs an in-depth investigation of program trace interfaces of modern COTS processors and their capacity to provide real-time control-flow monitoring capabilities and serve as the basis of new control-flow checking solutions. It proposes a new control-flow checking (CFC) hardware architecture based on such interfaces that increases the safety and security capabilities of software systems running in modern processors. The main questions addressed and answered by this thesis are:

- What are the added requirements for using debug and trace interfaces designed and employed in offline tests for online and real-time monitoring? How do we relate the information provided by these interfaces to the target binary program?
- How can we leverage hardware accelerators to implement efficient real-time monitors that scale with the number of CPU cores? How can this architecture promote ease of implementation and portability according to a designer's needs?

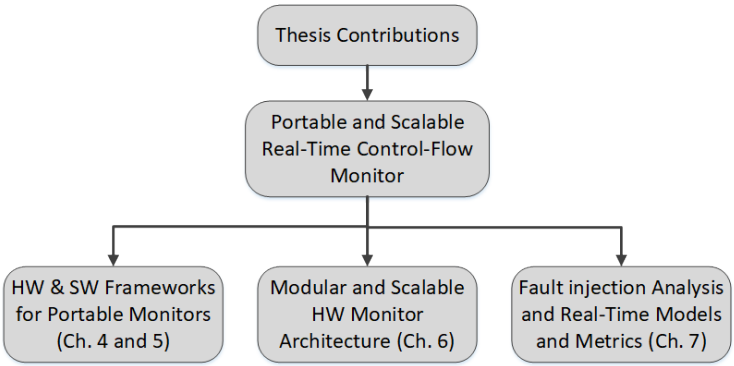


Figure 1.4: Main and secondary contributions to this Thesis.

- What are the metrics associated with such components for their adoption in safety-critical embedded systems? What are the limits of this technology in regards to detection timings and processing capabilities?

This work proposes and evaluates a new dedicated hardware control-flow monitor architecture implemented in a field-programmable gate array (FPGA). In the course of explaining this architecture, this work answers the previously raised questions and addressed topics. It shows that trace-based monitors can detect at least 98% of all faults in program branches and 100% of all explicit control-flow errors, providing clearly defined real-time and latency bounds. The analysis presented here shows how these techniques can be expanded to detect timing violations on software components while maintaining independence and isolation from the running application. This technique can be incorporated into System-on-Chip (SoC) designs or implemented as FPGA accelerators to existing COTS processors.

Figure 1.4 shows the breakdown of the main contribution of this thesis. The main contribution of this thesis is the development of the first demonstrably portable, modular, and scalable hardware-based control-flow error detection system. This thesis presents relevant secondary contributions for the domain and community of safety and error detection in the development of this architecture. It first analyzes existing systems and outlines a hardware and software framework for portable monitors in Chapters 4 and 5. The modular hardware

architecture with trace-decoder acceleration is presented in Chapter 6. Finally, Chapter 7 presents a novel timing analysis and modeling concept for real-time estimation of detection response times.

This thesis's contributions were also published throughout the development of this work in conferences, books, and academic journals. In [1] the first methods for a generic trace-based program analysis were published. Fault injections were performed over a series of benchmarks while control-flow traces were collected. The acquired information showed the reliability of the trace information in identifying errors and anomalies. The first portable architecture for a trace-based hardware monitor was published in [2]. The architecture was described in detail and presented with further fault-injection validations, comparisons to similar relevant methods, and implementation measures. The final component required less than 2% of FPGA resources to secure a single-core of the CPU.

The portability of this monitor was further analyzed in [3], where existing trace architectures were profiled and their features presented within the scope of monitoring purposes. The study proved that all existing architectures provide the means for proper monitoring capabilities, given a proper decoder unit. In [4] the first hardware-based control-flow monitoring architecture for multi-cores was presented together with the first real-time delay model for trace-based monitoring. The models and architecture were validated through interruption-based fault injections to branch instructions. The response time results combined with the presented model showed the necessary metrics and trade-offs fundamental to any control-flow monitoring system. The results first presented in this study show the minimum performance required by any monitor to secure a program in real-time, given the its inherent control-flow and binary characteristics.

1.3 Thesis Organization

This thesis answers the laid out questions and, in the process, proposes and evaluates a novel hardware control-flow monitor implemented in FPGA. Chapter 2 introduces definitions and the background knowledge necessary for the understanding of this work. It also presents an overview of the target processing platform, the dual-core Cortex-A9 System-on-Chip, with an integrated FPGA.

It details some crucial aspects of architectures, trace systems, and instruction set architectures (ISAs).

Chapter 3 presents relevant and related works done in the domain of fault tolerance. It shows current and accepted approaches for software, hardware, and hybrid CFC. It briefly explains the main way in which fault-tolerance techniques work. It then highlights the different existing CFC architectures, together with their main characteristics, advantages, and drawbacks.

Chapter 4 shows the major debug and trace solutions of today. It offers and analysis of their features for the purposes of real-time monitoring. Presenting a detailed overview of the connection between trace data and executed binary branch instructions. Meanwhile, Chapter 5 shows a quick overview of one simple binary analysis method used in this work for embedded applications. Binary analysis enables us to observe the control-flow structures of modern programs and to translate these structures into data that can be used by the real-time monitor units. The information in these chapters is used to build a framework for how the monitoring mechanisms operate on in the hardware architecture.

Chapter 6 describes the proposed architecture, including the trace decoder unit's pipeline, the configuration memory implementation and its stored control-flow graph data structure, and our fine-grained checking unit core and its pipelined implementation. In Chapter 7 the fault injection evaluation methodology is explained together the newly developed real-time model for hardware CFC monitors. The proposed architecture is shown to have small costs and be an effective technique to identify changes in the control-flow. It can be incorporated into SoC designs or implemented as a FPGA accelerator to existing COTS processors. The chapter goes into detail on the processing and real-time requirements for the implementation of trace-based control-flow monitors. These requirements also show the first limits and trade-offs for trace-based systems in real-time scenarios.

Finally, Chapter 8 summarizes the findings in this work. It brings this Thesis' results back into perspective according to their place within the state of the art. It also discusses its place within the future of control-flow monitoring, especially with the significant recent emergence of trace-based security integrity techniques.

2 Definitions and Background

The basics listed in this chapter provide a foundation for understanding the target technologies of this work and their underlying environments. For this purpose, this chapter gives a brief overview of their application domains, focusing on the relevant metrics for safety-critical systems. It discusses the specific characteristics of such metrics to increase their understanding and motivation for further improvements in the field. It additionally introduces the processor technologies and architectures referenced in the further course of this work and used to implement the case study. The context and the resulting specifics on using multi-core processors in safety-critical applications are also briefly explained.

2.1 Concepts in Safety of Semiconductor Devices

We routinely use terms such as "faults" and "failures" to describe unexpected and unwanted incidents surrounding our use of tools and machines. However, as these devices become too complicated for a single individual to comprehend fully, we need to define precisely the meanings behind each term. This section presents the essential collection of definitions that are used throughout this work. The definitions here range from *defects* and *upsets* in semiconductor devices that occur at the individual logic gate level, to *faults*, *errors*, and *failures* of system-level components. Our focus here is to define the terms necessary to understand the failures of embedded execution units, particularly CPUs and Microcontrollers.

2.1.1 Sources of Defects, Faults, and Upsets

Defects or upsets are described as unintended differences between the implemented electronic system and its expected function. Defects are used to label deviations caused by variations during the manufacturing process. The term upset, in turn, describes electromagnetic perturbations caused by environmental phenomena. Ionizing radiation, including energized particles such as neutrons, heavy ions, and electromagnetic radiation, may interact with integrated circuits producing different outcomes. These effects correspond to natural phenomena, typically a product of solar activity, deep cosmic sources, and earth materials with decaying particles.

Faults are the abstraction of a physical defect or upset at the logical level. They describe the changes in a device's logic function caused by a defect or upset. Therefore, *faults* are defined here as any variation from the expected logical behavior of the underlying hardware. They can be further detailed as transient, intermittent, or permanent. Transient faults occur and then soon disappear. They manifest from effects that can occur for a short period during the component's lifetime. Intermittent faults are characterized as a fault occurring, then vanishing, and then reoccurring, and so on. Examples of intermittent faults are signal interference, such as cross-talk between connections or communication lines. Permanent faults manifest and exist within the system until the defective component is repaired or replaced. These faults commonly occur due to manufacturing defects or physical damage to CMOS gates due to high charges. It is also possible that the device's electrical properties may mask some defects or upsets, causing no faults to appear.

With nanometer feature sizes dominating current semiconductor technologies, CMOS transistors have become more susceptible to faults caused by radiation interference due to their reduced threshold voltages, reduced node capacitance, and tightened noise margins [13]. Radiation effects may occur when a single ionizing particle strikes the silicon, creating trapped charges. These charges can cause current and voltage pulses that may be sampled and eventually change the logic state of the system. These issues are also known as single-event effect (SEE) and can be destructive or non-destructive. An example of a destructive effect is a single-event latch-up (SEL) that results in a high operating current, possibly above device specification, causing severe damage and requiring correction via a power cycle or reset. Non-destructive effects

correspond to transient faults produced by the interaction of a single energized particle and the PN junction of an off-state transistor (Fig. 2.1).

When a transient pulse occurs in a memory element, such as a register, it is classified as a single-event upset (SEU) (Fig. 2.3). When an ionized particle hits a combinational logic component, it creates a glitch or pulse that may be propagated and sampled by a sequential component. This type of upset is classified as a single-event transient (SET) (Fig. 2.2). Together, these transient effects are also known as *soft-errors* [13]

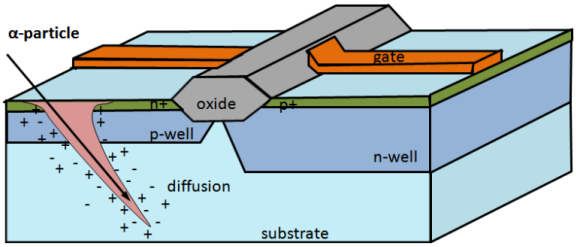


Figure 2.1: Diagram of the effects of an SET on a logic circuit.

Figures 2.2 and 2.3 show examples of SET and SEU soft-errors in a logic circuit. Figure 2.2 displays a particle hitting the XNOR gate of the combinational logic. This collision causes a glitch on the signal path amplified by the inverter and then sampled by the flip-flop. Figure 2.3 shows an SEU effect, where a particle, represented by the bolt, hits the register forming part of the sequential logic of this module, changing the element's logical value and directly affecting the rest of the component as wrong generated outputs are passed on.

Maiz et al. [26] showed that 95% of SEUs and soft-errors lead to single bit-flips in sequential logic components, with the remaining 5% spreading up to five bit-flips at maximum. These observations have been further confirmed and identified in modern integrated hardware designs [13, 27, 28]. Soft-errors can be detected and corrected at runtime by components inside a fault-tolerant system, which means they do not require a hard reset from the system. The rate at which a device or system encounters soft-errors is defined as its soft-error rate (SER). This rate is dependent on the type of transistor, manufacturing process, technology node, and the target running environment for the final system.

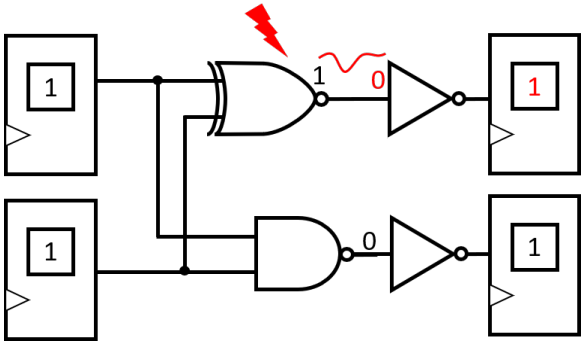


Figure 2.2: Effects of an SET on a logic circuit.

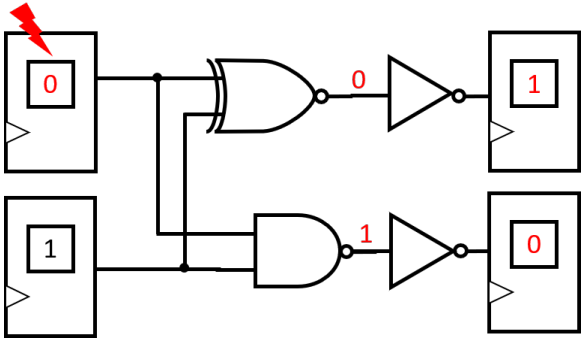


Figure 2.3: SEU effects on a logic circuit.

For CMOS transistors operating at a ground-level environment (as opposed to space or high-altitudes), the rate, or the probability, of faults occurring in a device increases with smaller node technologies [29]. The SER in such a scenario is proportional to the diffusion area (A_{diff}) of the trapped charges, the collected charge on the gate (Q_{col}) from the ionization of atoms through the collision path, and the minimum critical charge (Q_{crit}) required by the gate to represent a change in operation os state :

$$SER \approx A_{diff} * \exp(-\frac{Q_{crit}}{Q_{col}})$$

The diffusion area and collected charges depend on the topology of the collision and the energy of the particle, but the critical charge is mainly related to the technology's driving voltage and load capacitance, as shown in Figure 2.4. As the minimum critical charge required for a soft-error decreases, the probability of a fault increases.

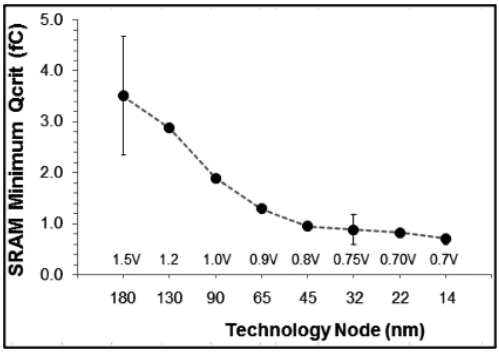


Figure 2.4: Minimum Critical Charge in SRAM Cells by driving Voltage and Technology Node [29].

2.1.2 Error, Failures, and Fault-Tolerance

Computers are possibly the most complex physical systems ever developed. Whenever a complex architecture behaves unexpectedly or fails to provide the function for which it was designed, it is typically said to be experiencing a system failure. To properly avoid such failures, we must understand their origins, causes, and effects and how these are related. Therefore, the meanings behind concepts like faults, errors, failures, and their relations must be precisely defined.

According to Avizienis et al. [30], a system is an entity that interacts with other entities, be they other hardware or software systems or the physical world. A complex system is also itself made of components or subsystems with their respective functions that interact to deliver the service promised by the system. A *system failure*, also called merely *failure*, is characterized by the system's inability to provide its service or function. A service is a sequence of states

performed by the system. Therefore a failure implies that one or more of its states deviated from the set of correct possible values. This deviation is called an *error*. For this reason, errors are also states of the system that may lead to a failure if left uncorrected. The faults in the underlying electronic hardware are ultimately the cause of such errors in the overall system.

More broadly, a *fault* is the underlying logical cause of an error. It is a manifestation of either a hardware defect or a software/programming mistake (bug) [31]. The origin of hardware faults comes from either internal or external effects. Internal effects might include defects due to fabrication processes, packaging materials, or aging and device deterioration. External effects might include interactions between the integrated transistors and randomly occurring radiation, such as SETs and SEUs explained in Section 2.1.1. However, some faults may be masked by the component's electrical characteristics, internal logic, or the running application. In such cases, no errors arise, and the system retains its correct execution state. It is essential to mark the difference between this definition of error and soft-errors defined in Section 2.1.1. Soft-errors are, within this work, a type of randomly occurring fault that affects integrated sequential logic components such as embedded processors, memories, and hardware accelerators. Finally, the relation between faults, errors, and failures can be summarized as follows:

1. **Faults**: are the logical outcome of a physical effect on hardware that modifies the information therein.
2. **Errors**: represent a new state or action taken by the system or subsystem because of a fault, outside of its intended or designed scope.
3. **Failures**: arise from the system's inability to keep providing its service or function after the appearance of one or more errors.

In other words, faults might lead to errors, eventually leading to the system's failure. Faults can also be further classified concerning their persistence in the system as transient, intermittent, or permanent. A transient fault happens randomly and only for a short period. These are most commonly SEUs and SETs. Intermittent faults correspond to a repetitive malfunction of a device or system that occurs at intervals, while a permanent fault is continuous in time. *Fault Tolerance* is thus the system's ability to provide the means to avoid service failures in the presence of faults. Therefore, this ability depends on the

specific type of tolerated fault and the maximum number of faults the system can withstand. Finally, *Error Detection* is the ability to detect the effects of faults after they influence the system or system components but before they cause a failure in this entity. Proper error detection capabilities are fundamental for the development of robust fault-tolerant systems.

2.1.3 Reliability and Failure Rates

When stipulating guidelines and rules for designing and implementing safety-critical systems, we require metrics to compare the relevant differences between multiple designs or implementation proposals. These criteria ultimately stem from the notions of errors and failures developed thus far. As we effectively wish to avoid or reduce failures, the first relevant analysis comes from the average number of failures we expect an application or function to experience over time, i.e., its failure rate. This section considers a single component of a more complex system and shows how reliability and Mean Time to Failure (MTTF) arises from the basic failure rate concept. Increasing the reliability and MTTF is inherent to the search for fault-tolerant architectures.

Let T denote the lifetime of a component (the time until failure), and let $f(t)$ and $F(t)$ denote the probability density function of T and the cumulative distribution of T, respectively. Therefore, $F(t)$ represents the probability that the component will fail at or before a time t ($P(T \leqslant t)$). A *component's reliability* is defined as the probability that it provides its function until a given time t ($P(T > t)$) [31]. The relationship between these definitions is thus:

$$R(t) = 1 - F(t), \quad F(t) = \int_0^t f(\tau)d\tau, \quad \int_0^\infty f(\tau)d\tau = 1$$

The MTTF of a component is equal to its expected lifetime, $E[T]$, i.e., the average or the mean value of the random lifetime variable T. In the case of random, independent, and transient faults in non-deteriorating systems, the probability of failure events is described through an exponential distribution. In such common scenarios the reliability $R(t)$ and MTTF are expressed by Eqs. (2.1) and (2.2), where λ represents the constant failure rate of the component.

$$R(t) = e^{-\lambda t} \tag{2.1}$$

$$MTTF = E[t] = \int_0^\infty t f(t) dt = \frac{1}{\lambda} \tag{2.2}$$

While $f(t)$ represents the probability that a new component will fail at time t in the future, we are also interested in knowing the probability that a working component of current age t will fail in the next moment dt. This can be defined as the conditional probability of $f(t)$ on $R(t)$. This conditional probability is denoted as the *hazard function* ($z(t)$). For exponentially distributed failures, the hazard function is constant and equals the given component failure rate given by Eq. (2.3).

$$z(t) = \frac{f(t)}{R(t)} = -\frac{1}{R(t)} \frac{dR(t)}{dt} = \lambda \tag{2.3}$$

Minimizing hazards is a significant concern in the design of safety-critical systems and an indispensable feature of safety standards described in Section 2.4. While an exponential distribution describes transient faults such as soft-errors in integrated circuits, they are not directly equal to λ. The failure rate of an integrated device will only equal its SER in the worst case when all faults occurring on the component cause functional failures, which is not the case in reality. Some faults do not provoke errors or do not change the component's functionality. They are *safe* faults. Faults that consistently cause failures, on the other hand, are called *dangerous* faults. Safety mechanisms increase the number of safe faults and decrease the number of dangerous faults.

The failure rate is relative to the rate of dangerous and undetected faults $\lambda_D U$ for systems experiencing faults and with error detection capabilities. Assuming that an appropriate repair mechanism is in place once a fault is detected, improving the rate of dangerous and detected faults $\lambda_D D$ will keep the system safely functioning. Therefore, studying and identifying the most common and most critical types of faults can show how to develop the best cost-effective methods to reduce hazards and the failure rate of critical components. Real-time monitor components that look for the errors from such faults can

improve system reliability, giving the system time to repair the error before a catastrophic failure.

2.2 Target Architecture

This section explains the major components of the Zynq-7000 SoC device used in our case study. Zynq-7000 is a commercial-off-the-shelf device designed by Xilinx and built with a 28 nm technology node from Taiwan Semiconductor Manufacturing Company (TSMC). This thesis uses a device version XC7Z020-CLG484, embedded in a commercially available ZedBoard Development Board. This SoC is also known as a type of heterogeneous programmable logic device. It integrates an ARM Cortex-A9 multi-core processor with a FPGA [32]. First, an overview of the Zynq-7000 is given, and then the characteristics of its implemented processor are detailed.

2.2.1 Heterogeneous Programmable Logic Devices

In keeping with the constant need for processing power growth, multiprocessing architectures have become ubiquitous in most domains. Heterogeneous architectures are a new step in this trend, where processing power and energy efficiency are also a concern. These architectures combine traditional general-purpose processors with additional hardware components that can perform better in specific application scenarios. Such components vary from graphics processing units (GPUs), custom application specific integrated circuits (ASICs), and FPGAs.

Custom logic can provide the most energy-efficient form of computation (100-1000x improvement in either efficiency or performance) through ASICs customized to a specific task [33]. However, ASICs are exceptionally costly to develop and produce, and as the name suggests, they are very optimized for a single use-case or application. Therefore, manufacturers and developers cannot easily reuse them in new applications. GPUs can be acquired as COTS components and have also been shown to outperform conventional microprocessors in target applications significantly [34]. They derive most of their optimization from large data segments' vectorization and symmetric multi-threading to

diminish large memory latencies. GPUs are very suitable for homogeneous data-parallel tasks but lose efficiency in other application-specific use-cases.

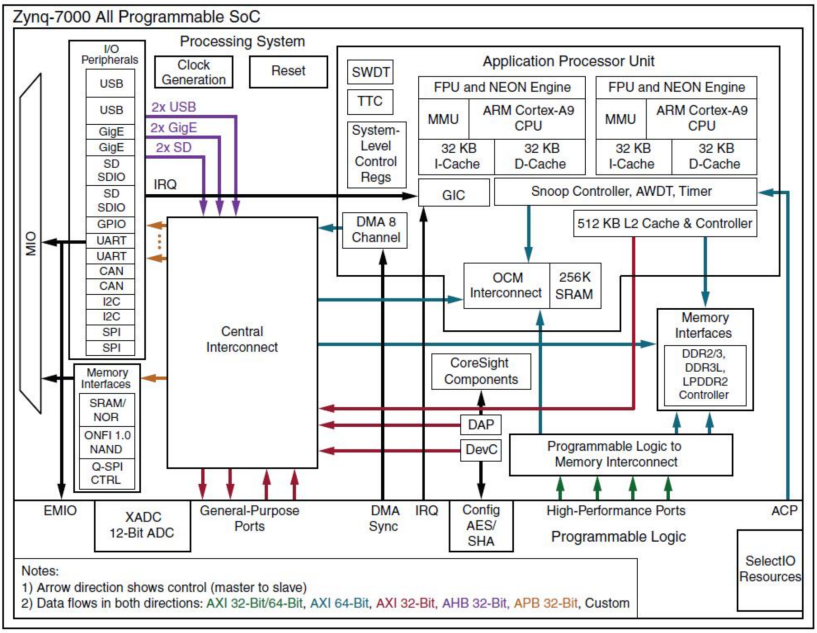


Figure 2.5: Block diagram of the processing system (PS) component of the Zynq-7000 [32].

Unlike ASICs and GPUs, FPGAs enable flexibility through programmable Look-Up Table (LUT) cells used to implement arbitrary logic circuits. Typical implementations of ASICs and FPGAs can present a 10-100x difference in area and power [35]. However, their increased flexibility, lower development costs, and widespread use in COTS components provide a valuable trade-off. They stand as a desired alternative between the performance of ASICs and the ease of use of GPUs. In this thesis, our use-case focuses on heterogeneous FPGA SoCs and a new error detection method for multi-core processors that fully utilize their unique advantage as a connected and flexible hardware resource. Ultimately, the proposed monitor technique can also be implemented as one component of a larger ASIC or SoC if COTS devices are not a possibility or a concern for the system developers.

2.2.2 The Cortex-A9 MPCore

The case study multiprocessor used in this work is a dual-core ARM's Cortex-A9. It implements a standard processor architecture based on the ARMv7-A instruction set architecture (ISA) [36]. Processors based on ARM designs, or its ISA, are used in all classes of computing devices. Systems based on their processor families occupy large market shares in mobile, communication, server, and even safety-critical systems, including automotive and space. In 2019, 22.8 billion ARM-based chips were shipped worldwide, with ARM systems holding 90% of embedded IoT, 75% of automotive advanced driver-assistance systems (ADAS), and 32% of networking infrastructure markets [37].

Figure 2.6 presents the general architecture of the processor. Besides the two processing cores, it also contains a 512 kB level two (L2) cache 8-way set-associative memory and an SRAM On-Chip Memory (OCM). Timers and a generic interrupt controller (GIC) are other functional blocks located in the processing unit. Finally, a Snoop Control Unit (SCU) [1] forms a bridge between the ARM cores, the caches, and the OCM. The SCU undertakes several tasks relating to interfacing between the processors and L1 and L2 cache memories. The SCU can further interface with the FPGA, also called the programming logic (PL) within the Zynq SoC, through an Accelerator Coherency Port (ACP).

The internal structure of each Cortex-A9 Core is presented in Fig. 2.7. They each have a dedicated single-instruction multiple-data (SIMD) media NEON processing engine, memory management unit (MMU), and separate 32 kB level one (L1) instruction and data 4-way set-associative caches. Each core implements a Harvard memory architecture internally, with a superscalar, variable-length pipeline from 9 to 12 stages. Its pipeline also employs advanced fetching, dual-issue, out-of-order instruction execution with dynamic branch prediction that decouples branch resolution from potential instruction stalls [38]. Up to four instruction cache lines are pre-fetched to reduce the impact of memory latency on instruction throughput. The fetch unit continuously forwards two to four instructions per cycle to the instruction decode buffer to ensure efficient pipeline utilization. Its superscalar decoder can decode two full instructions per cycle, and any of the four CPU pipelines can select

[1] Snooping is one of the several mechanisms for ensuring cache coherency, i.e., managing the consistency of shared cacheable data resources.

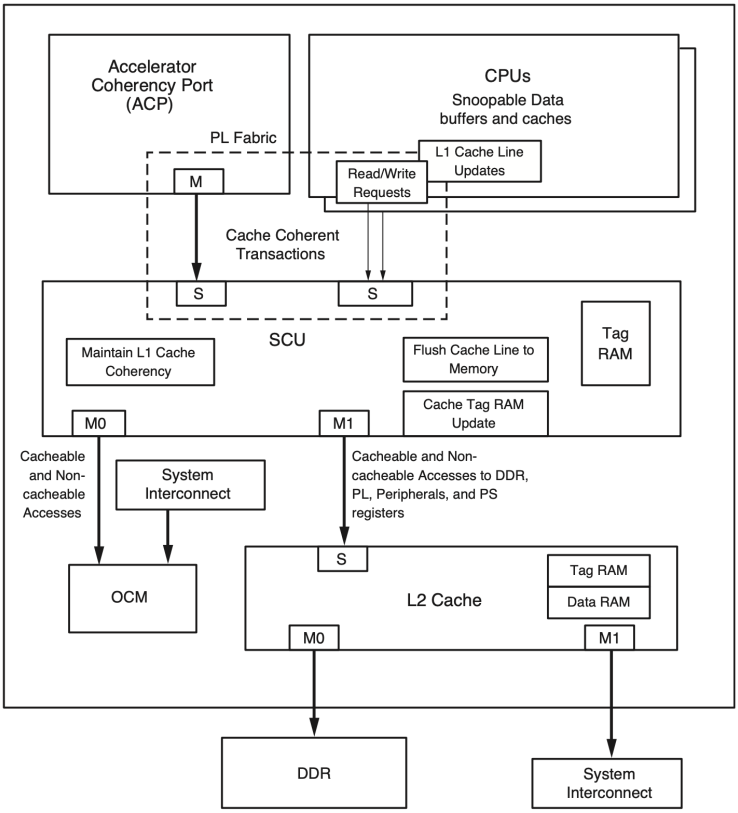


Figure 2.6: Cortex-A9 MPCore Architecture [32].

instructions from the issue queue. The parallel pipelines support concurrent execution across full dual arithmetic units, a parallel write-back unit, plus the resolution of any branch every cycle (Fig. 2.8).

The CPU pipeline employs speculative execution through the dynamic renaming of physical registers into a pool of available virtual registers. It also uses register renaming to eliminate dependencies without jeopardizing the correct program's execution. This feature allows code acceleration through effec-

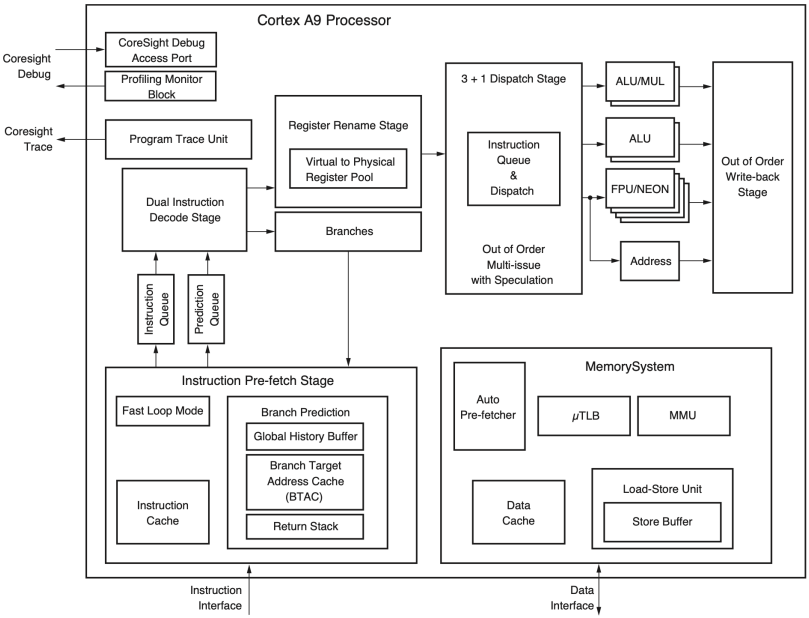


Figure 2.7: Cortex-A9 Internal Core Architecture [39].

tive hardware-based unrolling of loops and increases pipeline utilization by removing adjacent instructions' data dependencies.

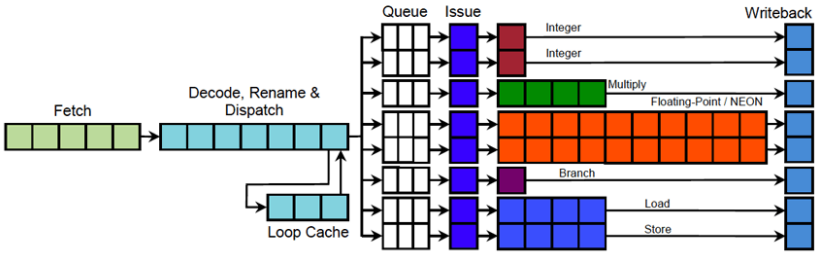


Figure 2.8: Simplified Cortex-A9 Pipeline Diagram.

To minimize the branch penalty in its highly pipelined CPU, the Cortex-A9 implements both static and dynamic branch prediction. Static branch prediction is provided by the instructions and is determined during compilation. Dynamic branch prediction uses the outcome of a specific branch's previous executions to determine whether the branch should be taken or not. The dynamic branch prediction logic employs a global branch history buffer (GHB) which is a 4,096 entry table holding 2-bit prediction information for specific branches and is updated every time a branch gets executed. The branch execution and the overall instruction throughput are also largely dependent on the branch target address cache (BTAC) which holds the target addresses of the recent branches. This 512-entry address cache is organized as 2-way × 256 entries and provides the target address for a specific branch to the pre-fetch unit before the actual target address is generated based on the calculation of the effective address and its translation to the physical address. Cortex-A9 also employs an 8-entry return stack cache containing the active 32-bit subroutine return addresses. This feature remarkably reduces the penalty of executing subroutine calls and can address nested routines up to eight levels deep.

Finally, each processor core implements its program trace unit through a Program Trace Macrocell (PTM). This unit supports the generation of CoreSight messages for all branch and control-flow operations, processor-specific events, and system-coherency events. The CoreSight and other market standard trace architectures are presented in more detail in Chapter 4.

2.3 Radiation Effects on Semiconductor Devices and CPUs

A processor is nothing more than a group of sequential and combinational circuits combined into one intricate component. This blend of different circuits makes processors susceptible to radiation hitting different areas with different effects. A processor's components are roughly divided into five groups according to the physical area hit by an SEU: program memory, data memory, register bank, control path, and data path; and into two groups describing the logical effects caused by a fault: data-flow and control-flow. The following subsections will address how soft-errors affect these different parts and analyze their effects from the program's point of view.

2.3.1 Soft-Errors in Processor Sub-components

Modern processing units are composed of highly complex combinatorial and sequential circuits that can all be affected by SEEs. Each region or area susceptible to SEEs is approximately categorized as being part of or corresponding to either:

- the instruction memory, e.g., Instruction Caches;
- the data memory, e.g., Data Caches;
- the register banks, e.g., General Purpose registers;
- the control path, e.g., Branch Predictors;
- the data path, e.g., Arithmetic and Logic Units (ALUs);

All types of memory are sequential logic circuits and, therefore, susceptible to SEUs. Due to the regular physical structure of memory cells, they are optimized to fit in smaller die areas than standard circuits and generally possess higher operating frequencies. Therefore radiation effects, such as multiple-bit upset due to a single particle, are intensified in memory components. There are two main types of available memories for embedded devices: Flash and SRAM memory. The first one is less sensitive to radiation effects, as it requires a higher voltage to change the state of its memory cells, typically over 5 V. Its main drawbacks are its constrained durability with a finite number of program-erase cycles, meaning that a memory position can only be written around 100,000 times before deteriorating its integrity. While the memory cells are more resilient against radiation effects, Flash components require more complex configuration logic to control this higher voltage and its required program-erase cycles, which is sensitive to radiation effects. On the other hand, static random-access memory (SRAM) memory cells are more sensitive to radiation effects since they operate at standard voltages. However, they also have better performance, less power consumption, and are not constrained by a finite number of program-erase cycles.

Two memory organization frameworks reign the design of current processors. They can access a single memory, where program and data memory are stored together, so-called *Von Neumann* architectures. Another possibility is that data and instructions each occupy independent memory components, with

the possibility of independent parallel access, which is the case of *Harvard* architectures. When stored separately, instructions are usually stored in flash memory and the data memory in an SRAM. Doing so makes it possible to reduce the number of upsets in the program memory, while fault-tolerant techniques can protect the data memory. When sharing the same memory, program and data memories are typically implemented on SRAMs. Since fault tolerance techniques are too expensive to protect the program memory, low-level approaches, such as error-detection and correction (EDAC) codes, are implemented. However, not all internal memory elements are protected. The Cortex-A9 processor, for example, does not support Parity error detection on the branch GHB RAM for large entry sizes.

The register bank is mostly a sequential circuit, just like the program and data memory components. Because of that, it is sensitive to SEUs. The register bank can be implemented through an SRAM or by using flip-flops. In the first case, we apply the same principles for data program memory. In the second case, hardware replication or software-based techniques can replicate and protect the information. While the total number of physical and pipeline registers of the Cortex-A9 processor is not made available, we can estimate its size and complexity by the number of required architectural registers.

The ARM architecture provides sixteen 32-bit general-purpose registers (R0-R15) for software use [38]. Fifteen of them (R0-R14) can be used for general-purpose data storage, while R15 is the program counter (PC) whose value increments every cycle, maintaining a pointer to the currently executed instruction. A direct write to R15 by a software operation will alter the program flow. R13 and R14 are additional special general-purpose registers, respectively the stack pointer (SP) the link register (LR). The SP hold the current value of the top of the program stack for the current function context, while LR holds the value for the current context's return address. The architecture also implements a dedicated current program status register (CPSR), and sometimes the execution mode may implement a copy CPSR register from a previous mode, called the saved program status register (SPSR).

As Cortex-A processors implement a modal architecture, non-banked general-purpose registers must be saved to the stack between context changes [39]. Figure 2.9 shows all nine execution modes of the processor. The lowest privilege mode is User mode, which most applications use. Hypervisor (HYP) mode has a higher privilege level but is not considered for secure execution.

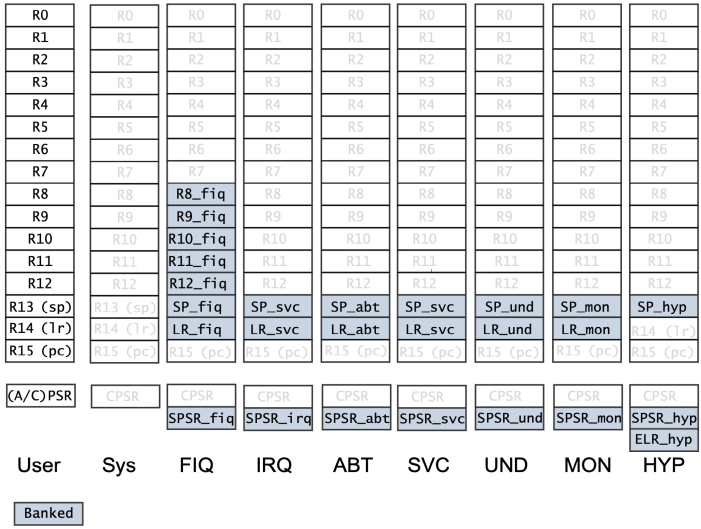


Figure 2.9: Architectural register banks and modes of the Cortex-A architecture [39].

It is used to control and switch between guest Operating Systems (OSs) that execute in a secure mode. Similarly, Moniotor (MON) mode is used to switch between secure and non-secure operation modes. System (SYS) mode is used by the base OS and shares the same register view as User mode but in a secure privileged context. All other modes correspond to internal interruption contexts, such as interruption request (IRQ) and fast interruption request (FIQ), Supervisor Call (SVC), and Hardware Abort (ABT) and Undefined Instruction Abort (UND) contexts.

In all modes, "Low Registers" (R0 to R7) and PC share the same physical storage location. Figure 2.9 shows how some 'High Registers' (R8 to R14) are banked for specific modes. For example, R8 to R12 are banked for FIQ mode, i.e., accessing these registers leads to a different physical storage location. For all modes other than User and System modes, SP, LR value, and the SPSRs are banked. The values of banked registers are highly critical. They maintain the correct execution flow state between different contexts and simultaneously have an extremely long liveliness, i.e., the time during execution that the register holds a valid, or *live*, value. Errors in any banked register can represent a

corruption of the execution flow upon the end of the context and may induce unwanted access to an invalid stack location and corrupt data.

The data path represents the computing circuit of the processor. It is defined as the circuit leading from a stored value (in the memory or the register bank) through the ALU or Floating Point Unit (FPU) and back to a storage element. It is composed of both combinational and sequential logic since the data path processes data and crosses the register barriers from the pipeline stages. Therefore, it is sensitive to SEUs (in the pipeline registers) and SETs (in the computing logic, such as the ALU). The effect of a fault in the data path usually leads to a wrong result by the end of the operation but hardly leads to an infinite loop or a control-flow error.

The control path is responsible for all decision logic of the processor. It calculates the next instruction to be fetched and sets required internal flags, such as control of the ALU to sum or subtract or if a branch is to be taken or not. The control path also encompasses all internal branch prediction logic, including branch history buffers and target caches. The control logic in modern CPU architectures is incredibly large and one of its most complex features.

2.3.2 Data-Flow Error Effects

Data-flow effects are those errors arising from a variable during its computation. They can manifest in a program as an execution that was accurately followed through but ended incorrectly. An example would be the operation `add R3, R1, R2` that adds the values stored in registers `R1` and `R2`, and saves it to `R3`. If a soft-error affected bit N of `R2`, it would change the operation to $R3 < -R1 + R2 \pm 2^N$. In such a case, the processor correctly performed the sum in the ALU but register $R2$ had an incorrect value. Embedded memories play an essential role in the overall reliability and system performance. They represent around 60% of the chip area in current processors. Memory cells, especially in caches, are built to be fast, efficient, and as small as possible with very low capacitance [40]. As a consequence, such characteristics make them particularly vulnerable to SEEs.

Data-flow effects are troublesome to detect or identify without redundancy techniques or sanity checks, where the data is compared to a golden or expected result or range of results. If left for too long, data errors may lead to incorrect

decisions or actions on the application's part, leading to catastrophic events. While such outcomes are of extreme importance in electronics safety and reliability, they are not the focus of this thesis.

2.3.3 Control-Flow Error Effects

Given a defined input sequence, any program executing in a modern processor or a similar Turing machine model of computation has a fixed series of executed operations. control-flow errors (CFEs) are characterized as errors in the program execution where the running application's instruction flow does not follow this defined sequence. In some cases, a fault that directly causes a data-flow effect may also cause a control-flow effect. An example, an SET could hit the ALU as it computes the comparison before a conditional branch, causing a branch to be taken when it should not. Ultimately, such types of errors are considered as a CFE. While L1 and L2 caches can be configured and either enabled or disabled depending on safety and performance requirements, branch caches and branch prediction tables cannot. An SEE into any of the complex branch prediction sub-components could disrupt the program's flow without warning. Figure 2.10 shows an example of a soft-error that changes the result of a branch target address and the corresponding change in the resulting control-flow graph (CFG).

As we can see, CFEs can be caused by faults in several different sub-components. A fault in the branch predictor might cause the system to determine a wrong target address or mispredict a conditional branch. Faults in the PC might cause a fetch of wrong instructions or the branch to an inaccurate PC-relative address. The architectural effects that SEEs may cause on the control path of a processor can be resumed to the following cases, also shown in Fig. 2.11 :

Branch Creation: A branch instruction is executed when there was previously no codified branch, e.g., sequential instruction is converted into a branch, and then this illegal branch incorrectly alters the program flow.

Branch Deletion: A branch instruction is not executed when it should, e.g., a branch is converted into a non-branch instruction. Therefore, a control flow path is not taken when it should be.

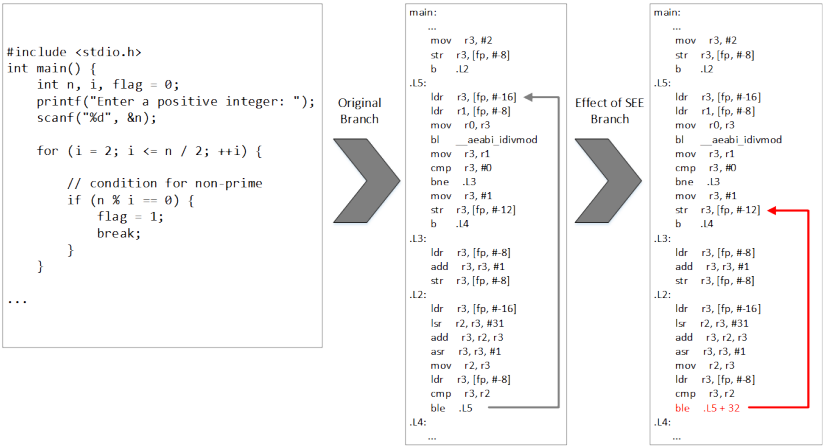


Figure 2.10: Control-flow of a program showing example of an error on a target branch.

Direct PC Error: It directly changes the next instruction fetched by the processor. It has the same effect as branch creation, but it does not appear on instruction trace streams as a branch.

Incorrect Target: An SEU modifies the value of the target address of a branch instruction, e.g., the LR used to return from a subroutine. This change will make the program's execution follow a branch to an incorrect address region.

Incorrect Condition Check: Happens when a branch that should go in one direction, based on a comparison, goes in the other direction, i.e., a branch is not taken when it should be, or it is taken when it should not be.

Another way to categorize CFEs is regarding their logical program outcomes [10]. This top-down categorization looks at outcomes from a branch operation's execution and the operation that succeeded it. These erroneous branch outcomes are classified as:

Not successor CFEs: The execution flow changes to an incorrect target from the current instruction. For example, the corrupted target address of a branch.

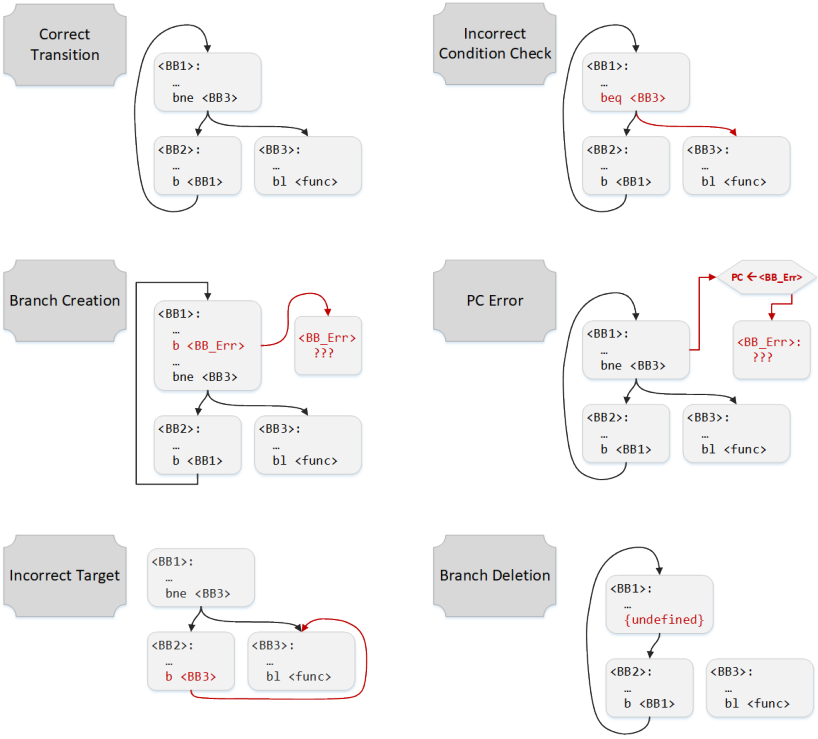


Figure 2.11: Control-flow graph diagram with different types of CFEs.

Wrong successor CFEs: The execution flow changes to a possible destination of the current instruction, which would not have been chosen in a correct execution state. For example, a conditional corrupted branch is taken when it should not.

Recent studies show that approximately 87% of all CFEs in processing units corresponded to not successor CFEs [41]. After injecting around 500k errors in 8 different benchmarks, the study determined that among all detected CFEs, what they called Direction Errors, equivalent to wrong successor CFEs, only represented 13%. Figure 2.12 shows the equivalence between these different classifications of CFEs. While almost all architectural faults can generate a not successor CFE, wrong successors can only happen from a deleted conditional

branch or a wrongly executed condition check. Data-path errors can eventually cause wrong successor errors, given the use and structure of the internal data. Therefore, protection against these types of errors correlates more closely to data protection techniques. Consequently, and due to the higher criticality of the first type of outcomes, this work targets not successor CFEs.

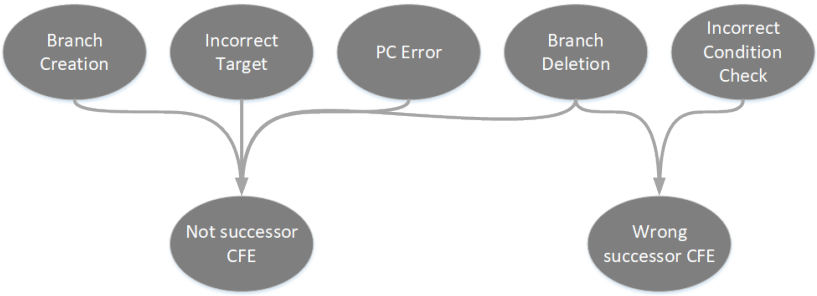


Figure 2.12: Equivalence between architectural CFEs and logical CFEs.

One of the most researched types of control-flow monitoring techniques is called control-flow checking (CFC). In the safety domain, several CFC schemes strive to provide a fine-grained, orthogonal hardening of individual control flows, with industry standards such as the automotive ISO-26262 [42] safety standard recommending their use. They work by identifying allowed branches after analyzing the program's CFG and issuing interruptions to the application if an unallowed branch is detected. They check in runtime the correctness of the control-flow through additional software operations after branches or by monitoring hardware interfaces. However, it has been repeatedly shown that evaluating such techniques is more challenging than we originally thought [9, 10]. The modifications applied to the running software by most control-flow protection techniques increase the overall system vulnerability, going against the design objectives. Chapter 3 explores existing CFC techniques and the problems many of them carry.

2.3.4 Architectural Vulnerability and CPU Faults

Figure 2.13 shows a general processor architecture with 5 pipeline stages. It shows examples of possible effects according to the component that was

affected. On the left, a fault in the instruction memory may corrupt a target branch, causing a wrong control flow path on the program. On the right, soft-errors in the data memory may change data, which may cause the system to output erroneous data or be propagated back to the system and cause other errors. Depending on the sub-component and the logic functions it fulfills, a data-flow or control-flow error may occur. The sub-component functionality is essential as it dictates the actual severity of possible errors in the system given a fault in this component. This subsection explores the use of architectural vulnerability factor (AVF) to estimate and compare which important functional components are more susceptible to errors.

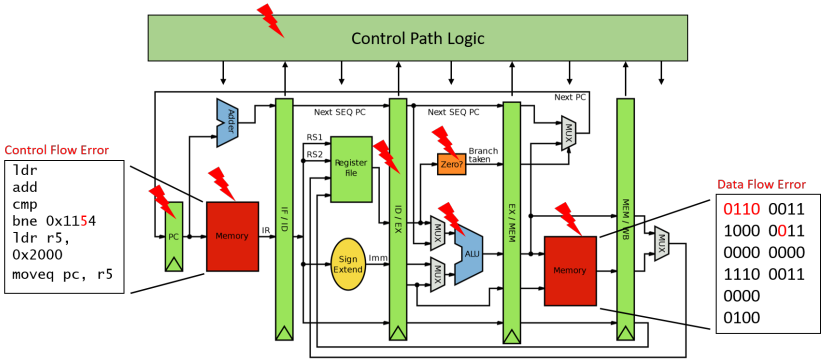


Figure 2.13: Sensitive areas of a general CPU architecture under SEEs.

When analyzing hardware components and systems' susceptibility, one of the most important parameters used is its soft-error rate (SER) [31]. This rate is the expected number of faults per unit time that a currently sound component will suffer in a given future time interval, usually expressed as a constant (λ_{SER}). This rate determines the final distribution of faults during the lifetime of a system. It is important to note that any hardware component's SER is determined by its integrated circuit technology and fabrication processes, as it corresponds to SEEs on the level of transistor gates. However, only comparing the SER is not enough, as depending on the component's architecture and complexity, a fault may be more or less masked, generating different rates of errors. Therefore, the AVF is the probability that a fault in a particular processor's structure will result in a visible error in the final output of the program [43]. Therefore, a structure's effective error rate is the product of its

raw circuit SER and its AVF. The overall processor error rate is calculated by adding the effective error rate of its N total components with Eq. (2.5).

$$\lambda_{CPU} = \sum_{i}^{N} \lambda_{SER}^{i} \times AVF_i \tag{2.4}$$

To compute the AVF, we must analyze which bits in a component's structure are required for an architecturally correct execution (ACE), also called the *ACE bits* [43]. Any fault in a sequential cell that contains one of these bits will cause a visible error in the final output. The other bits that do not contribute to the final output are then called *un-ACE bits*. It is important to note that the status of a bit stored in a memory structure inside the CPU might change during a program's execution. This can be easily seen by taking the example of a general-purpose register that in one cycle is not used and in another cycle holds the target address value for a branch instruction, e.g., the LR.

Thus, to calculate the AVF of a hardware structure, we must find the probability in time that an upset in that cell will cause an error. This probability corresponds to the average time a critical bit is alive in the component divided by the system's total execution time. It can be calculated by counting all pairs of ACE bits and cycles for a given hardware structure and dividing it by the total component area (in bits) times its running time (in cycles). This is expressed by Eq. (2.5), where N_{ACE}^{i} is the total number of ACE bits during the execution cycle i of the component, N_{bits} is the total number of bits in the hardware component, and T_{exec} is the total execution time in which the components was active, in cycles.

$$AVF = \frac{1}{N_{bits} \times T_{exec}} \sum_{i}^{T_{exec}} N_{ACE}^{i} \tag{2.5}$$

Through analysis of AVF, it is visible that not all faults in a microarchitectural structure affect a program's outcome. Some of the significant factors contributing to high failure rates are the complete instruction and data memory areas used by the processing system and its total execution time. Many models have been proposed to calculate the AVF for integrated systems. Simulation or HDL-based analysis can provide individual values for the components inside

a processor [25, 44]. Online performance metrics can also be used to estimate the AVF of large structure sections of a processor [45, 46].

However, this type of analysis requires detailed knowledge of the register transfer level (RTL) description of all components. This knowledge is not always available depending on different license agreements between design houses and IP users. Therefore another way to calculate the AVF is by estimating its average total ACE-bits through its bandwidth and execution time [43]. Using Little's Law, derived in the framework of queue systems, we can compute the mean number of resident critical bits in a resource and, therefore, the AVF of the resource [47]. This relation provides a way to compute the AVF while treating the component as a black-box. This law can be translated into Eq. (2.6), by N_{ACE} as the average number of bits in the black-box component being equal to the product of the average bandwidth of ACE bits into the component (B_{ACE}) times the average waiting time or residence in cycles of an ACE bit in the element (W_{ACE}). Thus, dividing it by the total number of bits in the black-box (N_{Total}) gives us an estimation for the AVF.

$$AVF = \frac{N_{ACE}}{N_{Total}} = \frac{B_{ACE} \times W_{ACE}}{N_{Total}} \tag{2.6}$$

Among ARM processors, the Cortex-R family is closely related to the Cortex-A, such as the Cortex-A9, but developed specifically for real-time and critical systems. Iturbe et al. [25] analyzed the vulnerability of different sub-components of an ARM Cortex-R5 CPU under soft-error simulations. This study identifies less than 10% of CPU sequential elements accounting for more than 70% of all errors. They also highlight the most critical sub-components in this CPU architecture, which are the following:

1. Register files
2. Cache store buffers
3. Branch prediction table
4. Branch return address stack
5. Exception handler logic
6. Instruction queue
7. PC logic

8. Register translation logic
9. Control registers

Almost all of these structures are either directly part of the CPU's control-path or responsible for the correct execution of control-flow operations. Other studies have also shown that between 33% and 77% of transient faults occurring in processors cause control-flow errors [5]. These studies show that ascertaining the correct execution of the system's control flow corresponds to identifying the majority of errors that may occur.

Understanding the correspondence of AVF to the final system reliability and what changes to hardware and software architecture affect it is extremely important as we develop fault tolerance techniques. Reliability estimates based only on raw device fault rates will be pessimistic, leading architects to overdesign their processor's fault-handling features. Overly complicated designs can be dangerous as any increase in overhead, be it in the hardware resource area, execution time, data, or instruction memories, increases the possibility of any fault occurring. These overheads can poorly impact the combined software and hardware system's final effective error rate and overturn any increases in the technique's reliability. While some studies argue that the architectural details required to perform a correct ACE analysis undermine its usefulness [48], others [49] show that carefully choosing the details in the system description can result in very precise AVF calculations.

2.4 Safety Standards for Critical Systems

Standards and norms must generally be observed and taken into account when developing safety-critical applications. The standards form a guide for development and thus also set implementation requirements. They describe processes and suggestions to be followed during development for safeguarding a safety-critical system, usually on a very abstract level. These processes guarantee users and consumers that the product's functionalities meet standard criteria and are within the defined safety metrics. Functional safety is described in detail in the standard IEC-61508 - "Functional Safety of electric / electronic / programmable electronic safety-related systems" [50].

The IEC-61508 standard provides the basis for different domain-specific standards related to different application areas. Among these, we draw attention to the ISO-26262 [42] used in Automotive development and DO-178C [51] and DO-254 [52] used in Avionic software and hardware components, respectively. One crucial term defined and used in such standards is *risk*. The term risk describes the expected frequency with which an event that can lead to damage will occur in combination with the damage's expected extent. This frequency is typically composed of:

- the duration of stay in the danger area,
- the ability to avert the danger,
- the likelihood of an undesirable event occurring.

The international standard IEC-61508 deals with the safety requirements for electrical/electronic/programmable electronic systems (E/E/PES). These requirements serve to define exact limit values and safety goals to which the finished system must adhere. For this purpose, it defines tolerable risk (residual risk). The handling and minimization of calculated or identified risk are one of the main concerns of such standards. Through such procedures, we can avoid or control dangerous failures of a system.

For these reasons, IEC-61508 places requirements on the procedure for developing such systems in order to be able to meet the safety requirements. These requirements include, among other things, implementation of risk analysis and the design of the hardware or software according to certain specified principles [53]. The standards derived from IEC-61508 also follow this essential procedure but contain industry-specific changes. Before development, it must be checked which area-specific norms and standards must be followed. Therefore, we must thoroughly test safety-critical systems after integrating a component that has been manufactured or procured in-house following a specific standard. The standard does not make any requirements for manufacturing or procurement processes. Integration tests should only prove that the modules interact correctly and fulfill their specified functions. Manufacturers need to check only hardware and interface functions, as the integration tests of associated software only take place in the software's safety lifecycle. Following this integration, the validation inspections can take place in order to prove that the system complies with the specified safety requirements [50].

2.4.1 Safety Levels

To make the system's safety classification clear, IEC-61508 uses the so-called Safety Integrity Levels (SILs). These SIL are the central aspect of IEC-61508. Function classification based on these levels is carried out according to the extent of damage due to the function failure, the likelihood of avoiding the danger, the amount of time a person is subject to the dangerous operation, and the probability of functional failure. The corresponding SIL is determined from this classification of the required function.

If the standard itself does not stipulate any special safety requirements after going through this scheme, the manufacturer's quality assurance aspects should apply. It becomes clear that a Safety Integrity Level is determined based on various, not precisely defined, conditions. The determination of the SIL is therefore not an exact process. The relative parameters must be determined by the development team and communicated consistently. It specifies four Safety Integrity Levels, the highest (SIL 4) intended for those functions that have the most devastating effects in the event of failure. After determining the SIL, the permissible failure rates of the respective systems or components can be taken from the tables of IEC-61508, Table 2.1.

Table 2.1: Acceptable failure rates according to IEC-61508 [50].

Safety Integrity Level	*Failure Probability for a function on demand*	*Continuous Failure Rate*
SIL 1	$10^{-2} \leqslant F < 10^{-1}$	$10^{-6} \leqslant \lambda < 10^{-5}$
SIL 2	$10^{-3} \leqslant F < 10^{-2}$	$10^{-7} \leqslant \lambda < 10^{-6}$
SIL 3	$10^{-4} \leqslant F < 10^{-3}$	$10^{-8} \leqslant \lambda < 10^{-7}$
SIL 4	$10^{-5} \leqslant F < 10^{-4}$	$10^{-9} \leqslant \lambda < 10^{-8}$

As shown in Table 2.1, SIL requirements make a distinction between functions executed on demand, and for continuously executing functions. The probability of failure F is used for seldom-used functionalities and computed over the whole function's execution time. For often or continuously executed functions, the failure rate (λ) is instead used, conventionally expressed in terms of Failure in Time (FIT) units. One FIT represents the failure of a system or components after 1 billion (10^9) hours of operation. Therefore, a functionality classified as

SIL 4 should have a failure rate on the order of 1 FIT or 1 failure per billion hours of execution.

ISO-26262 [42] is a specific standard relating to passenger cars. This standard deals with the development of electrical and electronic systems in automobiles This derivation and specialization emerged due to uncertainties in the application of IEC-61508 in relation to motor vehicles. It presents modifications that are specially designed for the series production of vehicles. The general standard also does not address the significant number of networked control devices present in automotive systems.

Given its specific industry requirements regarding the classification of vehicles, ISO-26262 offers a SIL alternative or extension called the Automotive Safety Integrity Level (ASIL). In the automotive sector, it can be assumed that the existing systems will never achieve SIL 4 since the IEC-61508 classification SIL 4 is reserved for hazards that result in a catastrophe with many deaths. Most of the safety-critical systems in this area are located between SIL 2 and SIL 3. Since the assignment to one of the two levels is very difficult here, the introduction of ASIL leads to an assignment in which ASIL-C is between SIL 2 and SIL 3 of IEC-61508. The standard also defines a new level not present previously. Quality management (QM) is assigned to functions that do not require additional safety measures in line with ISO-26262 requirements.

For the avionic domain, two essential guidelines are used: DO-254 [52] is used to develop hardware in avionics, while DO-178C [51] defines requirements for avionic software components. These are *de facto* standards, i.e., guidelines applied by all aviation authorities such as the Federal Aviation Administration (FAA) and European Aviation Safety Agency (EASA) [54, 55]. These guidelines describe the life cycle of hardware and software development and the activities and objects associated with the respective phases. They use different Design Assurance Levels (DALs), from DAL-A to DAL-E, representing the avionic counterpart to SIL of IEC-61508. The criticality and meaning of every level are defined as [56]:

- **DAL-A** - *Catastrophic*: Aircraft destroyed and many fatalities.
- **DAL-B** - *Hazardous*: Damage to aircraft with an overextended crew, occupants hurt with possible fatalities.
- **DAL-C** - *Major*: Large reduction in safety margins and possible injury to occupants.

- **DAL-D** - *Minor*: Little effect to operation of aircraft and on crew workload.
- **DAL-E** - *No effect*: No effect on aircraft or crew.

In this case, failure of DAL-A system results in a catastrophe, while a DAL-E system does not affect an aircraft's safety in the event of failure. The influence of the DO-254 extends from the aircraft manufacturer to the semiconductor manufacturer. Therefore, all components used in avionic systems must provide artifacts and documentation corresponding to the DO-254 and DO-178C [55]. Table 2.2 presents an overview of the general compatibility or equivalence between the different domain safety-levels described here.

Table 2.2: Compatibility between safety-levels of the investigated domain standards.

	Domain		
	General (IEC-61508 [50])	Automotive (ISO-26262 [42])	Avionic (DO-178C,DO-254 [51, 52])
Safety-Levels	-	QM	DAL-E
	SIL 1	ASIL-A	DAL-D
	SIL 2	ASIL-B ASIL-C*	DAL-C
	SIL 3	ASIL-D	DAL-B
	SIL 4	-	DAL-A

2.4.2 Safety Monitoring

Besides outlining the processes for classifying the suitable safety-level for a system's functionalities, safety standards also layout and suggest specific mechanisms to increase the safety and reliability of components. Safety monitoring is a means of protecting against specific failure modes or conditions by directly monitoring a function for errors that would result in a failure. Monitoring functions can be implemented in hardware, software, or a combination of hardware and software.

DO-178C explicitly points out the importance of software monitoring techniques. It incorporates guidelines that control-flow and data-flow should be monitored when safety-related requirements dictate, e.g., for DAL-A or B as-

signed software. The standard specifies requirements that allow a monitor to be assigned to a software component, given the safety level associated with the loss of its related system function. The important attributes of the monitor that should be determined to allow such an assignment are defined as [51]:

1. *Software level*: The safety monitor is assigned the safety level associated with the most severe failure condition category for the monitored function.
2. *System fault coverage*: Assessment of the system fault coverage of a monitor ensures that the monitor's design and implementation are such that the faults it is designed to detect will be detected under all necessary conditions.
3. *Independence of function and monitor*: The monitor and protective mechanism are not rendered inoperative by the same failure that causes the failure condition.

ISO-26262 similarly defines software safety requirements as all objectives needed to avoid a violation of its functions or properties. It further defines safety-related functionality as [42]:

- functions related to the detection, indication, and mitigation of faults of safety-related hardware elements;
- monitoring functions related to the detection, indication, and mitigation of failures in the operating system, basic software or the application software itself;

It also highlights the importance of isolation between the monitor and monitored function. A monitor cannot operate correctly if both the monitor and the monitored function are subjected to the same event or situation due to a common cause failure. These common causes of failures are therefore of notable concern. The standard in its latest iteration included new examples and classifications of failure modes for digital components such as CPUs and their sub-components, see Table 2.3. Due to modern CPU complexity, such hardware failure modes are hard to identify individually. They are also a primary root of dependent failures between software and hardware elements. To further reduce the probability of a safety goal violation due to dependent faults in hardware, such as in a CPU, safety measures must be implemented, such

as program flow monitoring independently and separately from the monitored software components.

Table 2.3: Possible failures modes for digital components.

Part / Subpart	Function	Failure Mode Aspects
CPU	Execute instruction flow according to given ISA.	• Instruction flow not executed (total omission) • unintended instruction flow executed (comission) • incorrect instruction flow timing (too early/late) • incorrect instruction flow result
CPU Interrupt Handler	Execute interruption service routine (ISR) according to interrupt request.	• ISR not executed (omission/too few) • un-intended ISR execution (comission/too many) • delayed ISR (too early/late) • incorrect ISR execution
Interrupt Control Unit	Send IRQs to given CPU according to defined events and to intended quality of service (e.g. priority).	• IRQ to CPU missing • IRQ to CPU without triggering event • IRQ too early/late • IRQ sent with incorrect data

Therefore, the importance of proper program monitoring in safeguarding critical software components is evident. However, the current state of program monitors and CFC is predominantly software-based, with few proposed hardware techniques, as presented in Chapter 3. Most software methods fail to increase reliability due to isolation problems and common causes of failures. Also, most hardware methods are architecture-specific and do not demonstrably scale well to multiple or different CPUs. Safety standards outline software designs and implementation guidelines, which help guide our monitoring objectives.

2.5 Security Vulnerabilities and Control-Flow Attacks

Our work is also related to research on intrusion detection techniques. Together with safety, the security of electronic and embedded systems is a significant concern. While the safety domain's main goal lies in preventing unforeseen random events that may cause system failure, the safety domain is concerned with protecting and ensuring the integrity of the system given deliberate attacks through events intentionally generated by a third party. While this work addresses the error-detection capabilities gained by the implementation of dedicated control-flow monitors, it must also reference the number of investigations using control-flow monitoring in the security domain for intrusion detection, particularly on the family of attack patterns called CRAs.

In these attack techniques, a malicious agent changes the application's control-flow through existing code in its memory, posing a high risk to the system's integrity. The most common type of such attack is called return-oriented programming (ROP). ROP attacks allow this agent to hijack the control-flow by using *gadgets* (code segments ending with a "return from function" instruction,e.g., `ret` in x86 or `pop lr` in armv7 [57]). Figure 2.14 shows an example of such an attack in which the stack values are altered to generate jumps between memory regions, executing different segments of the same application's code. ROP ROP attacks usually leverage buffer overflow or use-after-free vulnerabilities to overwrite the return address inside the stack. The control-flow is then illegally transferred to a gadget. This gadget changes the stack pointer to the beginning of the modified portion of the stack, redirecting it again to the next gadget by executing another return instruction. These gadgets execute one by one, forming a chain until the attacker's desired functionality is executed and the system is fully compromised.

control-flow integrity (CFI) is a general approach used for preventing such code-reuse attacks. It restricts control transfers along the edges of a program's predefined CFG [59]. The CFG is constructed by statically analyzing a program's source code or binary, but this statically constructed CFG is normally an over-approximation of the valid runtime target addresses for indirect branches, including indirect call, jump,and ret instructions. This is a particular problem for returns from frequently called functions that might have many valid target addresses [60]. Limited context information makes this backward-

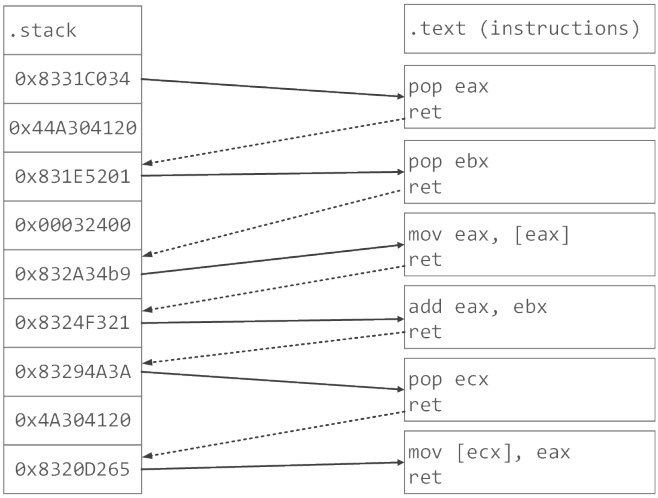


Figure 2.14: ROP attack example in x86 architecture [58].

edge CFI vulnerable to bending in which attackers make a program return to a different address within the set of valid target addresses [61]. Protection techniques should be simple to comprehend and enforce yet provide strong guarantees against powerful adversaries if they mean to be trustworthy. On the other hand, in order to be deployed in practice, CFI techniques should apply to existing code (preferably even to legacy binaries) and incur low overheads [62].

The main differences between monitoring architectures from the safety and security domains originate from distinctions between CFI attack models and CFC failure models. Fault-tolerance techniques are focused on one-time random soft-errors. CFI, on the other hand, is concerned with a persistent, adversarial attacker that can arbitrarily change data memory (in particular, by exploiting program vulnerabilities) but makes certain assumptions on register contents. Most CFC work provides probabilistic guarantees, while CFI entails that even a motivated, powerful adversary can never execute even one instruction outside the legal CFG. However, CFI does not aim to provide fault tolerance.

While CFI and CFC techniques are not always immediately comparable, the recent growing security concerns and the interest in CFI brings new similar techniques that we must observe for a fair analysis of our control-flow monitor method. In the past five years alone (2015 to 2020), there were 82 publications about CFI in relevant security journals according to the web of science portal [63]. In comparison, during the same period, only 12 journals were published about new advances regarding CFC. Similarly, the number of citations surrounding journals about control-flow integrity grows almost exponentially every year (Fig. 2.16), while the same interest surrounding control-flow checking seems to have slowed (Fig. 2.16). Although, the problems in the field of CFC are still not solved.

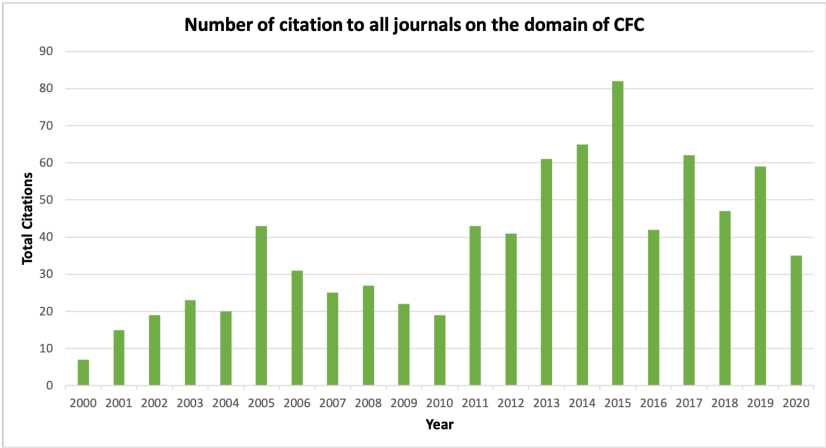


Figure 2.15: Total citations per year for journal in the domain of safety CFC architectures (data source: Web of Science [63])

2.6 Summary

This chapter explained the main differences among the safety-critical concepts of single-events, faults, soft-errors, control-flow errors, and failures. These concepts and terminology are the basis from which we build the understanding of safety and use them to relate to other state-of-the-art techniques in the same

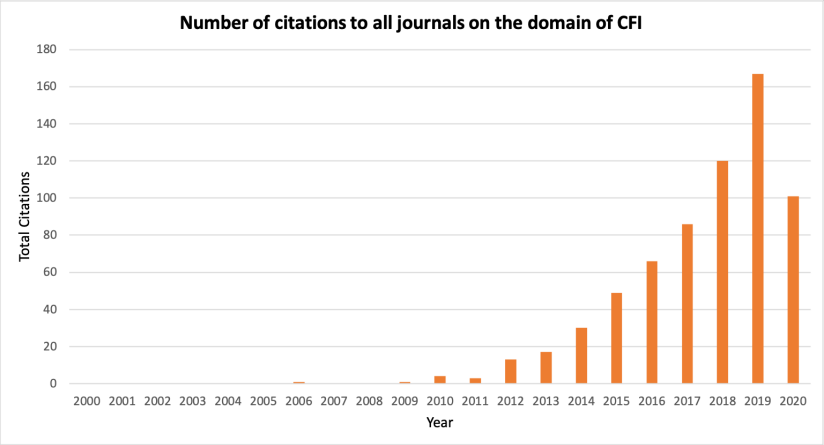


Figure 2.16: Total citations per year for journal in the domain of security CFI architectures (data source: Web of Science [63])

domain. In safety-critical environments, all applied systems must conform to reliability and availability standards. Several procedures and methods exist to help the final system achieve a tolerable failure rate. Such safety measures are specified in domain-specific standards, like the ISO-26262 for automotive systems, and both DO-178C and DO-254 for avionics [42, 51, 52]. However, specific safety techniques for multi-core and heterogeneous components remain still poorly detailed or lacking.

Information and communication systems play an ever-increasing role in satisfying customer requirements in today's high-end automotive and avionic systems. Especially in the last two decades, these industries had to face growing new functionalities mainly realized through electronic systems and software. As greater processing power and the need for higher function integration on single chips drive the market, semiconductor IP core vendors will focus even more on multi-core processing units, making single-core processors possibly not readily available in the future. Thus, new ECU architectures for automotive and integrated modular avionic (IMA) systems are needed. Multi-core architectures especially are more sensitive to failures due to their higher integration density [64]. Such failures can originate from SEUs caused by radiation effects.

One significant challenge in this context is the proper segregation between different applications considering unintentional interferences. Accordingly, an essential requirement for certification is precise and reliable isolation and monitoring of safety-critical applications. CFC schemes can present a useful method for both cases, providing a fine-grained, orthogonal hardening of individual control flows. The techniques are already highly recommended in current standards, with control-flow monitoring being highly recommended as an error detection mechanism for the highest safety integrity levels outlined in the automotive standards (i.e., ASIL C and D). Simultaneously, control-flow analysis is also highly recommended to verify architectural software designs at those same safety levels.

This chapter also detailed the necessary safety concepts regarding CRA that will be used in Chapter 3 to understand better and later compare relevant techniques from this domain. This thesis considers a fault-model that assumes faults and soft-errors as originating from SEEs, such as SETs pulses that may occur in the combinational logic and SEUs that translates into bit-flips that may occur in memory elements. This work targets the error detection of transient or intermittent faults on embedded software components' control-flow path at the hardware level.

3 Related Works

This chapter goes over well-established and current advances in the field of fault tolerance for integrated and embedded systems. It performs an overview of the area and the general approaches proposed through the years. The text then goes into detail about CFC techniques that focus on control-flow error detection, both SW-based and HW-based. The main differences between software- and hardware-based techniques are explained, as well as CFC and CFI techniques. In the end, some of their most significant drawbacks are outlined, including the points that guide further design considerations of this thesis. Relevant hardware CFI techniques are also analyzed due to their implementation similarities to the method proposed in this thesis and described in Chapter 6.

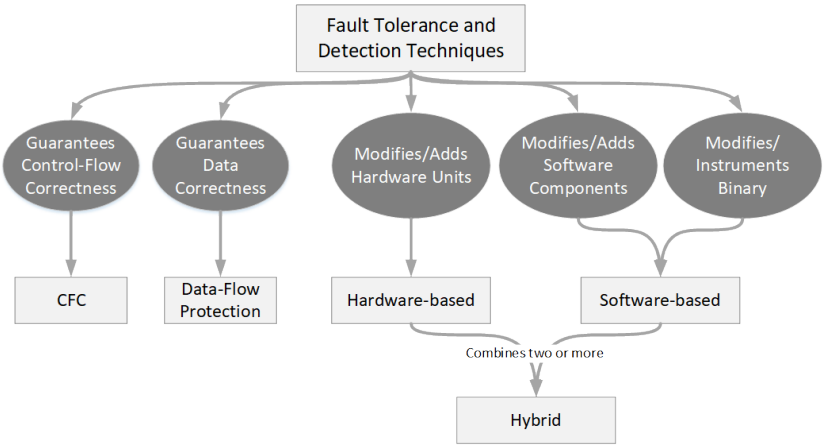


Figure 3.1: Taxonomy of fault tolerance techniques according to protection goals and implementation methods.

Figure 3.1 shows a simplified diagram of the taxonomy of solutions talked about in this work. Techniques are classified in two main ways: according to

their goals and guarantees or according to their implementation approaches. This chapter draws attention and compares the architecture proposed in this thesis with other similar techniques. It then extends its scope to analyze the differences and strategies of leading fault tolerance solutions. Control-flow protection solutions are looked into in further detail, including software-based approaches.

3.1 Comparison of Hardware CFC Techniques

This thesis proposes a fully hardware-based technique that explicitly avoids any execution and code overheads over the application called control-flow trace checker (CFTC). To achieve this, CFTC uses dedicated program trace interfaces provided in modern processor architectures. Table 3.1 presents a comparison between the technique described in this work and different hardware techniques [8, 41, 65–67]. The table presents the data reported by each paper. The cell was left blank when no data was provided ('-'). A star ('*') denotes cells where no data was provided but which could be calculated from other given information. Portability is defined here based on how reliant the method is on its target system architecture, interface, or platform and how much knowledge or access to RTL design specifications it requires.

Table 3.1: Comparison of different HW CFC techniques according to reported results.

Technique	*Portability*	*Performance Overhead*	*CPU Area Overhead*	*Data/Code Overhead*	*Detection Coverage*
CFCBTE [68]	Low	110%-340%	-	33%-44%	89%-94%
CFCET [65]	Low	33%-140%	-	-	79%-96%
Ragel et al. [69]	Low	2.83%	2.70%	9.3%-20%	100%
HETA [8]	Low	8%-34%	11.2%	46%-48%	100%
Nostradamus [70]	Medium	0%	10.9%	-	88%-100%
IDSM [71]	Medium	0%	≈25%	73%-94%	85%-95%
CFEC [41]	Medium	0.76%	2.5%	0%	86%-99%
CFC-ST [66, 72, 73]	Medium	0%	1.4%-2%	*45%	100%
PDTC [67, 74]	Medium	11%-51%	*0.33%	-	86%-96%
Proposed CFTC	High	0%	0.6%-1.3%	3%-17%	100%

Control-flow checking using branch trace exceptions (CFCBTE) [68] was one of the first proposed techniques that used PowerPC trace packet exceptions to detect execution errors on an application. However, this PowerPC-based method performs all checking operations on the same target processor, causing significant performance overheads. The first trace-based hardware technique, control-flow checking by execution tracing (CFCET), proposes an automatic watchdog processor (WDP) generation based on the verification of executed branches of a Intel Pentium processor [65]. It checks the linear addresses at which branch instructions are stored, and their target addresses with corresponding addresses calculated at compile time. This approach's main drawbacks are its dependence on an early and specific tracing system and using the main system bus to send and collect program trace packets, which adds considerable execution overheads. Their coverage results also relied on instruction vulnerability calculations that were correct for old variable size instructions based on CISC architectures but do not necessarily correspond to newer RISC processors and more recent microarchitecture-based x86 systems. CFCET [65] relies on a specific trace packet architecture and direct communication with the software application.

Following CFC techniques, Ragel et al. [69] and hybrid error-detection technique using assertions (HETA) [8], combine previous software protection with processor hardware extensions. They make up some of the first proposed hybrid approaches. Both add separate hardware RTL units controlled by supplementary software micro-instructions. They are assigned low portability due to the need to modify both hardware and software. Nostradamus [70] proposes a dedicated watchdog architecture for superscalar and out-of-order cores able to predict the outcome and effect of fetched instruction before their commit stage. This particular core requires at least direct access to the monitored processor's fetch stage interface and internal state flags. Improved disjoint signature monitoring (IDSM) [71] proposes a dedicated watchdog that checks the instruction and data interfaces between a processor and its first-level caches. A drawback of this approach is the lack of information available to the monitoring unit about the processor's internal operation, particularly with complex processors. Control-flow error checking (CFEC) [41] modifies only the hardware but requires the full RTL description of the processor and is not usable by most COTS devices. It is assigned medium portability as it only requires hardware modifications, and the software is left unchanged.

Du et al. [66] presents one of the newest developments in the domain of hardware-based control-flow checking. They propose a new hardware technique that uses the instruction trace stream data to detect illegal control-flow operations. Their CFC module performs two checking processes: the first checks the target of branch operations to guarantee that they were correctly executed; the second calculates a signature key in real-time based on the opcodes and addresses (PC values) of executed instructions. At the moment of a branch, i.e., the end of a basic block, it compares this signature to a pre-calculated and expected value.

While check one can only detect that branch instructions have performed a correct jump, it will not identify an error of previously modified branches. The received machine code would be wrong, but its target branch would be seen as correct. The second check operation is more powerful as it uses information about the program flow, i.e., CFG, to perform its checks. This information is encoded and stored in their CFC Signature Table (CFC-ST) memory. The second check is able to detect the incorrect execution of any instructions inside a block, but the requirements for such an implementation are not as prevalent in modern processor architectures. For example, both Cortex-A9 and Cortex-A53 FPGA embedded systems provide sequential instruction tracing interfaces, but their interfaces do not provide PC values and opcodes of executed instructions.

They propose two different algorithms to fill the block address and signature table, one static and another dynamic [66]. The different table filling algorithms are used to select which basic blocks to monitor in applications with a much larger number of blocks than the total available size for the table. For the static method, they further propose three ways to rank basic blocks in order to choose which to fill their checking table with: first by total instruction size (S), second by the number of times they are executed (N), and third by the total size times the number of times it is executed (SxN). They try with this method to identify which blocks have a higher likelihood of being executed. Their results show that the final approach outperformed the first two. The last approach can be understood as the best measure of likelihood by considering each basic block's total execution rate from the entire program's execution time. If an SEU has an equal probability of occurring in every cycle, the most relevant basic blocks to be protected are those with a higher percentage of the total system execution time. In this sense, a better approach would be not to use total program size but instead the average block execution time, as a block

may have fewer instructions but have a longer total execution time if it does more memory accesses.

Their dynamic CFC (Dyn-CFC) table filling algorithm, on the other hand, works by filling the table in runtime with calculated signatures. The table starts empty and is filled with a new basic block entry if it is not on the table. If there is a key collision when inserting a basic block, a least recently used (LRU) replacement algorithm is used. The dynamic algorithm was more effective than the static one for large applications combined with minimal table size, on the order of 16 entries. Their approach relies on collecting both *opcode* and *PC* values from the execute stage of the pipeline at every execution cycle. This type of trace data collection is only possible on their chosen target architectures, the miniMIPS [75] and Leon3 [76] cores, or other FPGA programmable soft-cores, as they give the flexibility to collect such information and enable the implementation of dedicated FPGA components that execute at the same clock domain and frequency as the central processing unit. Therefore, these techniques cannot be applied to more powerful hard-core processors. For SoCs with embedded FPGAs, these check operations are also much more onerous. Hard-core processors and FPGAs typically run in different clock domains, with processors running at much higher frequencies than the FPGA. Additionally, such SoCs usually do not provide FPGA interfaces to perform direct internal pipeline readings.

program and data trace checker (PDTC) is another fully trace-based hardware technique [67]. In this approach, a dedicated monitor was placed in FPGA and used to monitor an integrated hard-core processor's data- and control-flow path. Their chosen target system was an ARM Cortex-A9 processor. Besides using provided trace messages to identify control-flow errors, the technique also relies on data access trace messages generated by a particular component of the ARM SoC. This Instrumentation Trace Macrocell (ITM) generates traces driven by special software operations. It produces multiple data packets providing compressed information. Their component analyzes software instrumentation trace packets to monitor the data-flow. These packets export any 32-bit data value generated by the software to the trace interface. Protected applications write to stimulus ports of the ITM, mapped in memory as special registers to trigger this packet generation.

To retrieve PC and control-flow related data, they use the integrated PTM component of the ARM CoreSight trace. Their implementation focuses on trace

packets that give only explicit PC address information, i.e., I-sync, Branch Address, and Waypoint Update packets. This component is also one of the principal implementation targets of this thesis. Section 4.3.1 gives a more detailed description of its functioning and message descriptions. PDTC's control-flow checking mechanism relies on coarse-granular comparisons of retrieved current instruction address information. They compare the PC address with up to eight user-programmable address ranges to determine if execution has reached a forbidden or unexpected region. CFC-ST [66] and PDTC [67] are two of the newest techniques that rely on trace data to monitor CFEs. The monitor raises an error signal if the incoming address values are outside the configured bounds. Even though both methods do not modify the processor, they are intrinsically tied to its trace system architecture. As a result, they are only portable to systems that implement these processors or trace systems and are therefore given medium portability.

The current trend is that newer approaches focus on direct monitoring of trace interfaces instead of data and memory buses but still primarily use specific architectural features. Non-invasive instruction tracing is already present in most newer processors and microcontrollers, but they do not follow a standard protocol nor usually provide all the same information due to different implementation constraints. Their trace information is provided first and foremost with testing and debugging in mind, not for online and real-time monitoring. To properly create a modular and effective control-flow monitor, we must find out what similar features different protocols provide. The overlap in trace data across platforms is used to build a portable monitor over target architectures. Section 4.8 outlines a minimal trace interface that provides the least amount of information required, including the order of sequentially taken or non-taken branches and their target addresses. These basic features are combined with binary static analysis to monitor received trace packets and corresponding transitions in the program CFG.

Most of these techniques do not consider different target architectures and the ways their methods escalate to other more powerful devices. Hardware techniques require modification of the underlying hardware system or additional components, FPGA-embedded COTS devices make it possible to use such techniques while still maintaining low implementation costs. FPGAs are optimized for power, and many techniques already exist to protect and secure their configuration fabrics and memories against SEUs [77]. Processors integrated

into FPGA SoCs are also much more performant than any soft-cores available nowadays. If we wish to have such useful systems implemented in critical environments, we must also have proper techniques that ascertain their safe use.

Primarily, this work improves on PDTC and CFC-ST approaches by not relying on a single trace interface's specific features. It focuses only on features shared between different commonly used trace architectures such as CoreSight, NEXUS-5001, and Intel Processor Tracing. CFTC also does not require code instrumentation such as PDTC or tight processor coupling such as CFC-ST. PDTC was the only method implemented on the same trace system as CFTC. However, they add significant performance overheads, shown to be a significant drawback for reliability, and do not explore the latencies and possible vulnerabilities introduced by the trace bus. The architectural results presented for CFTCs in Table 3.1 are explained in Chapter 7.

The proposed system of this thesis focuses on protection against all not successor CFEs. It provides higher and finer-granular coverage than previous software-based approaches without incurring additional execution overheads in the performance or code area. CFTC provides similar coverage to fine-grained trace-based techniques while requiring fewer resources. It is additionally portable to multiple systems with minimal re-implementation costs. The only methods with more CFE coverage include full or partial data-flow protection, therefore also identifying wrong successor errors. However, this additional protection usually requires the re-execution of binary instructions or the replication of pipeline units. It can cause overheads and problems for the final system reliability and are beyond the scope of this thesis.

3.2 Fault Tolerance Techniques for Embedded Processors

Throughout the past 40 years, many methods to detect and correct errors of integrated circuits have been proposed [66, 68, 78–80]. These techniques rely fundamentally on information redundancy to detect and correct faults in complex systems, whether structural or data-based. Fault tolerance techniques

aiming to protect integrated processing units are divided into three broad categories:

Software-Based Techniques: are the techniques that work by altering or modifying the application software. They can be used in almost any type of system.

Hardware-Based Techniques: are techniques that rely on modifications of the underlying hardware architecture without any need to modify the running application. They may rely on either structural hardware redundancy or on extra monitoring hardware that periodically checks for faults.

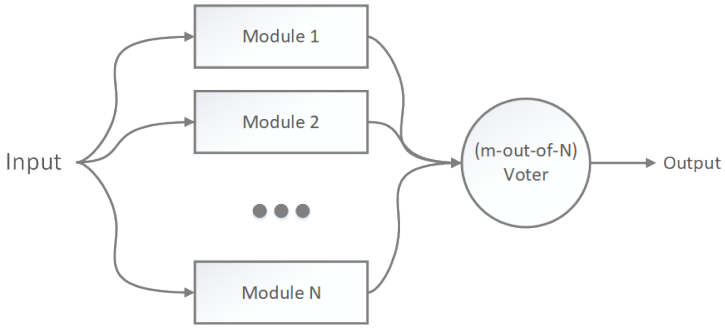
Hybrid Techniques: are methods that combine special software functions or modify the application and rely on additional hardware units. One example of this is lockstep processors with implemented software checkpointing rollbacks.

These various techniques can be further applied at different architectural levels of the system description. Different implementations can focus on the architectural instruction level, the component structure level, or the logical and transistor level. This thesis focuses on system protection through hardware-supported techniques at the architectural instruction level.

The less elaborate hardware fault tolerance techniques rely on multiplying the existing hardware architecture to add redundancy mechanisms. TMR is a well-known hardware-based method that relies on triplicating hardware structure and comparing their results for the same input with a voter. Such a method can then decide the correct response with a 2 out of 3 majority vote [31]. Alternatively, if normal dual redundancy is used, errors can be detected, but it is unknown which component is currently experiencing a failure. More generally, N-modular redundancy (NMR) is defined as a technique based on component replication to achieve fault-tolerance. NMR is characterized by the use of N instances of the same system, module, or component, where the resulting combination is correct if m out of the N copies of the system agree on their outputs, also called an *M-out-of-N system*.

Figure 3.2 shows the diagram of an NMR system, with a majority m-out-of-N voter. The original reliability of the protected module is R_M, and the voter reliability is R_V. For a TMR with an ideal 2-out-of-3 voter ($R_V = 1.0$), the final

reliability resolves to $R_{TMR} = 3R_M^2 - 2R_M^3$. As an example, for a module with a 5% failure rate, i.e., $R_M = 0.95$, the failure rate with TMR will be 0.725% ($R_{TMR} = 0.99275$).



$$R_{NMR} = R_V \sum_{i=m}^{N} \binom{N}{i} R_M^i (1 - R_M)^{N-i}$$

Figure 3.2: Diagram implementation of NMR (M-out-of-N redundancy) and its resulting reliability.

Hardware error detection based on simple redundancy still requires methods to place the system back in a known state after an error occurs. This repair mechanism can be done through software either by resetting the application or performing some checkpoint and rollback mechanism for processing units. In the case of an FPGA, the board can be partially reprogramed, correcting both transient and permanent errors that might have happened in hardware units. Spatial replication techniques may change according to design constraints and can be applied to different levels of granularity through the hardware architecture [80]. Although such techniques can provide high fault tolerance capabilities, they introduce large area overheads, with TMR causing a factor increase of about 3.5 times in the original circuit's final area, which leads as well to much higher power consumption.

Many modified processor architectures have been proposed that implement TMR and other safety techniques [81]. Some processors are specially designed and sold with hardware-based techniques already implemented. They are usually known as RadHard processors. Examples of RadHard processors

are Cobham Gaisler's LEON 3FT/4FT and Space Micro's Proton200k DSP Processor Board [76]. However, they are usually much more expensive than COTS processors and do not achieve the performance of newer high-end architectures [82, 83]. Figure 3.3 presents a comparison between what is the state of the art for COTS and RadHard processors performed by [82] showing a 10-year gap in their throughput.

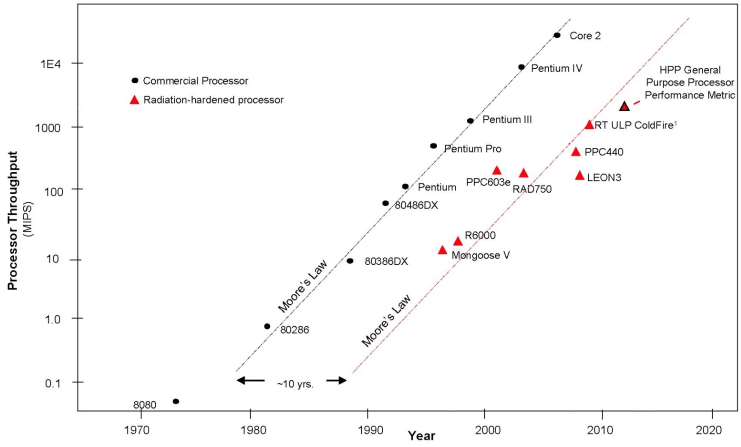


Figure 3.3: Evolution of performance between COTS and RadHard processors [82].

As an alternative to redundancy, other hardware-based techniques focus on implementing monitoring blocks, also called WDP, to ensure that the system correctly performs specific operations [84, 85]. Such devices monitor the control flow of software systems running inside the processor and perform memory accesses' data flow. In order to accomplish this, monitoring components work by performing usually specific types of operations [78, 79, 84]: memory access checks, which look for unexpected memory accesses such as unused memory areas and restricted function memory areas; consistency checks, where the monitoring units verify that data held in memory or register components is adequate by employing knowledge on the currently executing CPU task; alternatively, control-flow checks that use instruction tracing mechanisms or special modifications to a processor's microarchitecture. Section 3.3.3 explores in detail how well-known hardware-based monitoring techniques are implemented.

Software-based techniques, also called Software Implemented Hardware Fault Tolerance (SIHFT), focus on concepts of operation of hardware resources, time, and information redundancy to detect the presence of faults during program execution [86]. These techniques automatically apply instruction patterns and modifications to a program during its development or assembly phases, replicated instructions ensure correct execution, or replicate certain control flow condition checks to ensure that the correct path is taken. These SIHFT techniques usually have significant drawbacks, causing performance degradation due to the additional instructions used to improve reliability, increasing the overall execution time and program memory usage.

Section 2.3 shows that faults occurring in different CPU structures will have different kinds of associated errors depending on what kind of structures they were, either data path or control path. Therefore, fault tolerance techniques that aim to protect the software at the architectural level can be characterized according to their aims or fault detection goals:

- data-flow checking techniques that aim to detect faults affecting the data flow,
- and control-flow checking techniques aim to prevent errors that may happen to a program's execution control flow.

The first group targets data structures such as variables, registers, execution units, and data memory. On the other hand, CFC tries to secure the normal execution of a program, preventing such errors as wrong target branches or incorrect conditional executions, causing the execution of a path that should not have been taken. CFC techniques are fundamental, as the most sensitive components to faults are those related to the control flow logic. Their use is also heavily recommended in most safety-critical application domains.

3.3 Control-Flow Error Detection Techniques

Control-flow error detection techniques, also called control-flow checking or program monitoring techniques, aim to detect incorrect branches during program execution. They codify behavioral information about the program and verify it in runtime, detecting possible CFEs. They check every or most

control-flow decision point in a program during its execution by replicating and checking that instructions were correctly executed or by checking the processor's interfaces.

Most techniques work by first performing a static analysis of the code that will be executed and generating from that the program control-flow graph (CFG) structure [87]. The CFG is built as a directed graph, $DG = \{V, E\}$, with vertices (V), and edges (E), such that the vertices correspond to the basic blocks (BBs) of the program and all edges correspond to possible transitions between basic blocks. A BB corresponds to a sequence of instructions that execute sequentially with no branches coming into or leaving from the sequence, except at the first and last instructions. Every branch's target must point to the start of a basic block, and the instruction following every branch corresponds to the start of a new basic block. These blocks end at a branch or at the instruction that precedes a basic block's start, i.e., a branch target.

Figure 3.4 shows the basic blocks of an example program presented in in Listing 3.1, codified in ARMv7 assembly language [36]. The example code implements the call for a recurrent subroutine, a recurrent Fibonacci function. It compares the input value and returns the same input value on a 0 or 1. Otherwise, it calls itself twice at the end of blocks 7 and 8. It must also be noted that the realization of a program's execution, also seen as a walk of its CFG, can only be known at runtime, as its inputs will determine how many times a function is called as shown in this example.

Listing 3.1: Example ARMv7 Assembler code.

```
1  B0:
2  _main:
3      push {fp, lr}
4      add fp, sp, #4
5      sub sp, sp, #8
6      mov r3, #0
7      mov r0, #8
8      bl _fibonacci
9  B1:
10     ...
11 B2:
12     _fibonacci:
13     push {r4, fp, lr}
14     add fp, sp, #8
15     sub sp, sp, #12
16     mov r0, [fp, #-16]
```

```
17      ldr r3, [fp, #-16]
18      cmp r3, #0
19      bne B4
20  B3:
21      mov r3, #0
22      b B9
23  B4:
24      ldr r3, [fp, #-16]
25      cmp r3, #1
26      bne B7
27  B5:
28      mov r3, #0
29      b B9
30  B6:
31      ldr r3, [fp, #-16]
32      sub r3, r3, #1
33      mov r0, r3
34      bl _fibonacci ; B2
35  B7:
36      mov r4, r0
37      ldr r3, [fp, #-16]
38      sub r3, r3, #1
39      mov r0, r3
40      bl _fibonacci ; B2
41  B8:
42      mov r3, r0
43      add r3, r4, r3
44  B9:
45      mov r0, r3
46      sub sp, fp, #8
47      pop {r4, fp, pc}
```

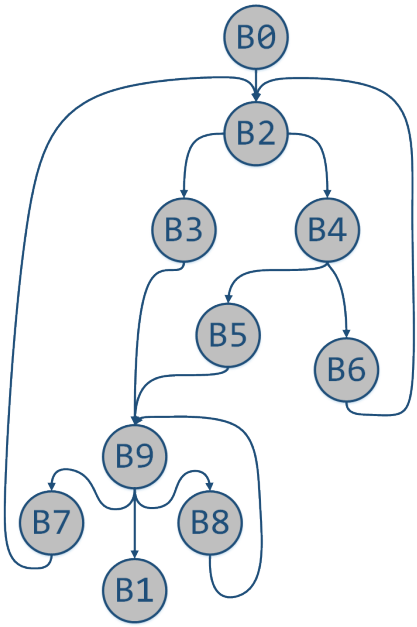


Figure 3.4: Example control-flow graph partition of a program.

3.3.1 Software-Based Techniques

Software-based monitoring techniques for CFC work through additional instructions. These are inserted into the target application, generally at the beginning and end of basic blocks. These instructions serve to check that the last executed branch was executed correctly. Due to the nature of such techniques, they are generally unable to certain types of CFEs, especially those that occur inside the same block. Many methods based on this idea have been proposed in the literature throughout the years, such as:

- Control Flow Checking Using Assertions (CCA) [88]
- Enhanced Control Flow Checking Using Assertions (ECCA) [89]
- Control Flow Checking by Software Signatures (CFCSS) [90]

- Assertions for Control Flow Checking (ACFC) [91]
- Yet Another Control Flow Checking using Assertions (YACCA) [92]
- Control Flow Error Detection using Assertions (CEDA) [7]
- Selective software-only error-detection technique using assertions (SETA) [6]

CCA, and other SIHFT techniques that followed it, assign signatures to sequential blocks of a program and use assertions during runtime to verify them [88]. CCA implements a Block Identifier (BID) and a Control Flow Identifier (CFID) for its assertions. The first identifier is a unique value assigned to each BB, while the second identifies all permissible transitions or edges present in the CFG. Thus, it checks the control flow's correctness by setting and verifying each block's BIDs and CFIDs as the program runs. The BID is saved in a dedicated register at the entry point of a new block. We compare the current block's BID to the saved value upon exit. If a block is entered from the middle, then the current BID will not match, and an error will be detected. It ensures that all BBs are only accessed at their input point and only leave at their exit point.

While BID ensures that the CPU correctly executes individual blocks, the CFID secures the blocks' correct execution order. CCA stores these values in a two-element queue and initializes the queue with the starting CFG vertex's identifier. When a new block starts, the next blocks' CFID are queued. Once it finishes, the top of the queue is read and compared with the current block's identifier. If there is a mismatch, it means a wrong predecessor block was taken. While ingenious for a software-based verification approach, this model cannot detect CFEs inside a sequential block. It requires that all blocks from the same predecessor share the same CFID. Therefore it is also unable to detect wrong conditional branches.

ECCA [89] improves on this concept by using a new signature scheme and generating block groups that overcome some of the shortfalls of CCA. It combines CCA's BID and CFID into one signature, increasing the complexity of comparison functions and instructions added at the beginning and end of BBs. This technique is sensitive to different sizes of BBs due to its signature calculation scheme. A small BB increases error detection capabilities and decreases detection latency. However, its main drawback is its increased memory and performance overheads.

CFCSS [90] creates a random signature for each basic block and a new global signature G stored in a register updated at runtime. The technique performs XOR operations between origin and successor block signatures and G to encode and decode the correct control-flow sequence. The new signature is continuously computed at runtime and compared to an expected signature every time a branch occurs. This technique's main drawbacks rely on its limited detection capabilities as it cannot detect, for example, when the processor wrongly executes another instruction inside the same block.

YACCA [92] attempts a similar approach as ECCA but with an optimized signature generation and checking function. Their results show less memory and power overheads than ECCA and CFCSS, with similar CFE coverage rates. Venkatasubramanian et al. [91] proposed a new approach, called ACFC, that is not dependent on predecessor-successor relationships between basic blocks. It instead uses the execution parity of a basic block for fault detection. This technique adds one extra instruction to a subset of all BB in the flow. It also presented similar but lower fault coverage than ECCA and CFCSS with significantly less memory and performance overheads.

One of the more advanced software-based techniques is CEDA [7]. It adds signature verifications at the start and end of every basic block. It devises two signatures for every block and uses a global signature register updated in runtime. Newer approaches such as the ones presented by Chielle et al. [6] build on top of CEDA and focus on further code analysis to identify which are the most critical basic blocks of a program. They implement signature monitoring on these blocks, not achieving maximum error detection but optimizing the overhead cost imposed on the code.

3.3.2 Weaknesses of Software-Based Techniques

Aggregate studies on the efficacy of such techniques found that the additional instructions and overheads inflicted seriously diminish any previously estimated reliability gains [9, 10]. These incurred overheads are more harmful than what is first thought. They show that the evaluations performed until now on such methods fail to qualify their reliability gains accurately through independent approaches. Standard fault injection methods for validating CFC techniques overestimate their protection results. Usual validation schemes

only consider simple residual failure rate or CFE coverage rates, i.e., the rate between errors detected by the monitoring methods and the total number of faults injected that resulted in errors. These validations are performed through software or simulated fault injection campaigns and usually focus on the exact faults against which the system is designed to protect.

Schuster et al. [9] tested the techniques mentioned in Section 3.3.1, performing an *absolute failure count* analysis. This approach weighs the vulnerability of faults that resulted in CFEs according to the system's critical bits' lifespan. After this analysis, they found an overall degradation in the application's reliability due to a widening of data liveness intervals from each technique's overheads. They show that signature-based CFC schemes perform lower than advertised reliability gains. Their re-evaluated techniques include CFCSS [90] and YACCA [92]. Instead of approximating $P(F)$ from the number of injected faults by generated failures, they weigh the faults according to the life-time of data elements in memory/registers [93]. This way, they give more importance to the faults that would occur more often on the system. In essence, they use an analysis based on AVF, described in Section 2.3.4, to consider the actual risk associated with each injected fault. ACE bits that only exist in the system for a brief amount of time may be over-counted if injected during a stochastic fault injection campaign. They refer to the typical approach as residual failure rate and their weighed fault injection as absolute failure count. Overall, their study disclosed various latent deficiencies of both the residual failure rate metric and software-implemented CFC, potentially compromising their widespread use.

Rhisheekesan et al. [10] chose to avoid fault injections altogether due to their elevated computational cost and developed a method that directly computed the system's AVF through simulations of each described CFC method. They directly calculate all vulnerable bits before and after applying the protection schemes, therefore identifying the difference in vulnerable bits and estimating the methods' overall strength. Ultimately, they identify the same underlying problem as the previous study.

To calculate the AVF of different pipeline components, they used description models of the Amber Core [94] combined with instruction-level system simulations. They then measure the normalized vulnerability, calculating the AVF rate found for each method running different benchmarks against a baseline without protection mechanisms. The study performed this analysis on several well-known software techniques such as CFCSS [90], CEDA [7], and also one

hardware-based technique called CFCET [65]. Their reported results show that all software-based CFC implementations had average increases in their practical vulnerability factors between 18% and 21%. They also note that the hardware-based approach correctly detected errors without increasing the system AVF due to its isolation and minimal overheads.

The original CFCET uses an old Intel Pentium platform and trace subsystem, while Rhisheekesan et al. use an ARM-based instruction-level simulator [10]. They did not disclose their re-implementation of the hardware-based approach. Unlike software-based approaches, hardware-based strategies are not easily portable and are even harder to realize in simulated systems, not fully described in hardware.

In general, they find that the additional code overhead in implementing the advanced CFC techniques renders them worse than earlier software CFC techniques, as the residency of additional CFC code instructions in different microarchitectural processor components tends to increase vulnerability. While the reliability improvements in terms of residual failure rates were significant and in line with the literature, the transition to absolute failure counts, which accounted for memory and execution time overheads, resulted in moderate improvements at best and even degraded reliability at worst. This disparity between the two metrics suggests that ignoring overheads has led to ill-guided optimizations and previous CFC scheme evaluations.

Finally, they also note that experimental investigations with control-flow fault injections suggest that prevailing hardware protection mechanisms already catch the vast majority of CFEs, with only a small fraction of less than 6% manifesting themselves as failures [9]. However, we usually require hypervisor or supervisor software systems to control these mechanisms, and it has been shown that operating systems significantly increase the overall system vulnerability if left unprotected [95, 96].

3.3.3 General Hardware-Based Methods

Instead of relying on structural redundancy like other hardware approaches, hardware-based CFC techniques deploy dedicated monitoring devices that analyze specific processor interfaces in search of errors. Some techniques may further implement software component modifications to acquire enough

information about a program's execution state. These methods correspond to a hybrid approach that combines hardware monitoring with additional software instructions.

Broadly, auxiliary hardware units verify control-flow by comparing instruction addresses with expected target addresses or comparing runtime signatures with reference values saved in local memory. Therefore, these WDPs implement two primary units: one checker unit implemented in an external general-purpose processor or microcontroller running a single dedicated monitoring algorithm; a local verification memory used to hold the reference information corresponding to a correctly running system.

Such memories can be local, only accessible by the monitoring unit, or shared with the system, resulting in performance overheads as the main buses are continuously used by checking modules. The checker units compare the information stored in the system with runtime data that the application processor generates. However, the processor's monitored interfaces may not provide full access or observability of the required data. Information in registers, buses, cache memories, and other internal hardware interfaces may not be accessible. These constraints limit the use of watchdogs depending on the architecture of the target processor. Another debilitating constraint is the necessity to modify the SoC or to integrate the watchdog into a communication bus close to the execution unit. Many proposed techniques assume that the system's full hardware description is available and modifiable, either as part of the processor development in ASIC or as a soft-core component implemented in FPGA.

Although hardware-based techniques provide high-reliability increases, they can introduce significant overheads, like an increase in area and power consumption, if directly built on top of the processor architecture. Hardware-based techniques also present additional design and production costs for ASIC implementations [97]. Using COTS with embedded FPGAs, we can benefit from a design already optimized for power consumption and their inherently lower integration costs than a custom-built ASIC. Finding monitoring techniques that best use available FPGA-Processor interfaces is necessary for their use as part of critical systems.

Several hardware-based monitoring techniques have been proposed throughout the years. They differ in several aspects and details, but they can be compared in regards to the interfaces used to gather information about the running system.

Therefore, such techniques can be non-exhaustively classified with respect to the following groups:

- **Software Interfaces:** [8, 78, 98, 99]
  These approaches form hybrid techniques built upon the exchange of information between software and hardware components. The hardware monitors receive information about the state of execution directly from the running application, while the checking logic and decision-making are accelerated in hardware. This information exchange can be done through modified ISAs, creating new instructions used solely by watchdog processors, or combined with data interfaces to send software-generated signatures to the watchdog. These methods provide more comprehensive fault coverage due to increased collected information but maintain the same drawbacks of software-based implementation regarding memory and performance overheads.

- **Data Memory Interfaces:** [100–103]
  These approaches rely on monitoring bus activity and access to critical variables or code segments. Therefore, these architectures are less useful for CFC at the machine instruction level. More information might be gathered from the system if combined with software signature checking algorithms. However, it can further increase data bus bandwidth and possibly cause significant performance overheads.

- **Performance Counters:** [104, 105]
  These schemes rely on using integrated performance measuring units inside the processor to evaluate its correct runtime behavior. Depending on the architecture, performance counters may provide valuable information, including the number of executed branches, total committed instructions, power consumption, and total memory accesses. We may use performance counters in different ways to verify the correct execution of a system. For example, specific elements can be directly observed such as total committed branches per code segment, or many different elements combined into a single runtime signature later matched to a previously calculated pattern.

- **Instruction Memory Interfaces:** [71, 79, 102, 106]
  These methods correlate the progression of fetched instructions to the application processor's control-flow decisions at every cycle. They usu-

ally use instruction cache interfaces that deliver both current fetched instruction addresses and opcode values. Although they do not require modification of the internal processor architecture, they can only be implemented if a system-level description of the hardware is available. Therefore, only hardware manufacturers can integrate such methods, and they are much less portable to COTS systems.

- **Direct Pipeline Monitoring:** [98, 107–109]
  These schemes depend on direct reading access to the processor's pipeline. They extract the highest amount of information available by the hardware, as the monitor system can verify in detail all execution steps and operations performed by the system. Such techniques can only be implemented if the complete processor description is available, for example, in soft-core processors or the design of an entire SoC system.

- **Debug & Trace Interfaces:** [65–68]
  Debug interfaces combine the benefits of the large amount of information given by pipeline or instruction cache monitoring with data memory interfaces' flexibility. They provide information about the system's execution state but do not require any extensions to the internal hardware architecture, as most modern processors already provide program instruction tracing capabilities.

These possible CFC strategies make trade-offs between system access level assumptions and the amount of information we use to track the execution flow efficiently, e.g., interfaces closer to the processor core provide more information but are harder to integrate. These techniques may also combine multiple interfaces to extract more information from the system, such as simultaneously checking data and instruction interface transfers [79].

The chosen interface imposes constraints and limits on what platforms these techniques can be applied. More detailed information is gathered with direct pipeline monitoring or instruction memory and cache interfaces, but we must have the complete hardware description of the execution unit to implement such monitoring units. Therefore, such techniques can only be used in an entirely FPGA-based platform with soft-core processors or an SoC components' development phase. We depend on access to information from interfaces such as data buses, performance counters, and debug & trace units if we wish to support COTS devices. Overheads may also be unavoidable if this access is

dependent on a shared memory bus, as constant read operations may increase usual access times.

Lu et al. [78] showed one of the first hardware-based software control-flow error detection approaches. Their technique relies on a dedicated second processor running a single monitoring algorithm that receives signatures from executed basic blocks and compares them with memory reference values. The Watchdog Direct Processing method [79] expanded the monitoring interfaces by adopting a custom interface that monitors the processor's fetch stage as well as the memory bus, monitoring both executed instructions and accessed data memory regions.

OSLC [107] and ASIS [99] propose hardware architectures that rely on signature generator components directly coupled to each monitored application processor. The signatures are then sent to the checker unit through the internal bus. OSLC further proposes using online signatures generated during the test phase, avoiding the requirement of static code analysis for the pre-calculation of stored signatures. These approaches allow for monitoring multiple processors with a single architecture but add significant area and latency overheads to detect faults in runtime.

Modern trace interfaces can typically be configured for both on-chip access through the memory bus or off-chip access through dedicated implementation-specific interfaces. Therefore, they provide overhead-free and detailed information gathering means that can be used efficiently by the proper hardware. Some of the more recent CFC methods propose reading instruction opcodes and address interfaces from the program traces or instruction caches [66, 106]. However, as shown in Chapter 4, not all trace protocols are built the same, and even similar processors may present differences in implemented features. Therefore, the primary concern of this work lies in identifying precisely which relevant features are shared across different processor families.

3.4 Summary

CFC architectures are usually implemented in two ways, either by hardware or software modifications. Hardware techniques directly modify the CPU, memory bus, or implement watchdogs, while software-based methods rely

on replicating or modifying instructions and instrumenting the application code. Due to additional instructions, software-based methods present high overheads, increasing the overall execution time and program memory usage. Recent studies show that overheads can introduce new regions susceptible to errors and that they significantly increase system vulnerability, undermining any actual safety gains [9, 10]. Therefore, overheads in execution time, code size, and data usage significantly increase the system's chances of having a fault.

This chapter presented the current state of art of control-flow protection mechanisms. It first presented the newest hardware monitor architectures of past years, concentrating on the newest methods that focus on using program traces to verify the correct execution of the system in real-time. The analysis of these methods demonstrated how every proposed trace monitoring mechanism was restricted to its target processor architecture or use case. While the results show how trace-based methods are a powerful tool for detecting errors, they do not concern themselves with expanding their methods beyond the proposed use-cases.

It then presented the most current software-based methods for protecting the control-flow and detecting CFEs. These methods rely on software transformation and new instructions that calculate running signatures and identify if a branch was correctly executed. These methods typically cannot detect all types of CFE, specifically errors that modify the target to a point inside another valid block. They also add significant execution and program size overheads due to the need to execute as many as two times as many branch operations instructions. These high overheads are due to additional instructions, increasing the execution time and program memory usage. Recent findings show how these overheads are a critical limiting factor to performance and the actual final reliability of the system. The additional instructions generate new points of failure and increase the total effective critical bits of the program. However, hardware-based techniques have been shown not to suffer these same problems.

Therefore, the two key unanswered questions in the domain of trace-based CFC are:

1. can we reliably use control-flow trace across different processor architectures?

2. can we use trace to detect errors under real-time constraints? Moreover, what timing limitations can we expect from the online decoding of trace packets?

The first of these questions is the focal point of Chapter 4, where different trace protocols are analyzed together with the architecture of COTS processors. These findings are later used to develop the portable CFC monitor presented in Chapter 6. This work proposes a two-stage architecture, where it first translates the actual target trace packets to an independent architecture interface before evaluating them in a dedicated verification unit. This approach allows the verification stage to be reused across different target systems, as long as designers provide the appropriate translation stage.

The second question is answered with the evaluations and collected results presented in Chapter 7. It is shown how application characteristics such as average time between branches and speed variations between monitor and processor may cause high latencies and large occupancy of input buffers. To perform this evaluation, we combine fine-grained static binary analysis, presented in Chapter 5, with the limits in decoding speed given by the translation stage architecture shown in Chapter 6.

4 Debug and Trace Architectures

Modern processors integrate additional components that increase the observability of executed operations. They are used to help software developers debug and test their systems, and together these components are called the Trace Subsystem, shown in Fig. 4.1. They form an independent subsystem whose generated data helps follow or trace the execution path of the processor. Usually, trace data is sent outside of the integrated system through a JTAG connection. However, some SoCs allow users to connect to and access a trace's output interface directly. Therefore, it is possible to implement a dedicated monitor component, e.g., in an FPGA, that continuously processes the trace data in search for errors and can quickly stop the processor if a wrong state is identified. All this without interfering with the normal execution of the program. Most CFC methods rely on direct access to these components, which is not always possible in hard-core processors or is very dependent on the architecture. The solution proposed in this work accesses only the available system interfaces, working with the same data used by debugging environments.

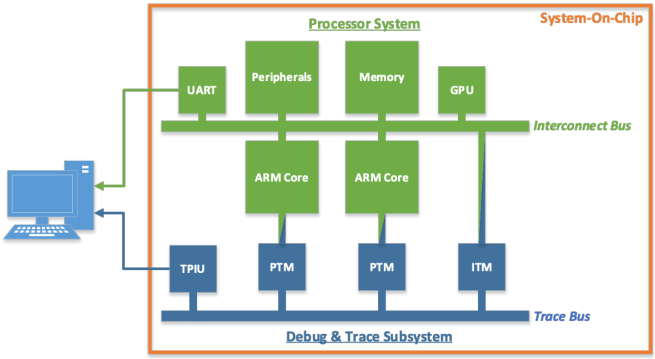


Figure 4.1: Detailed trace subsystem architecture based off of the Zynq-7000 platform.

This chapter analyzes and characterizes the trace subsystems of some of the leading processors of semiconductor manufacturers currently in the market, especially those used in safety-critical systems. These include our main target architectures as well as their safety-critical usage domains. To systematically and adequately perform system partitioning and isolation in modern complex systems, we require hypervisors, OSs, and other self-monitoring software. These software system components are also responsible for error detection and isolation of failures. Therefore, we must ensure that these components and application codes run correctly and without failures. A failure in software-based monitoring functions can lead to a violation of technical safety requirements. Integrated hardware monitors are a strong candidate for the reliable monitoring of critical software and identifying errors in processing units. They can be easily isolated from the system and supervise the application without impacting or changing its characteristics.

The available COTS multi-core processors do not usually provide sufficient protection for safety-critical environments. Hence, we must be able to adapt and extend them for such domains. Most modern processors share a prevalent feature: non-invasive debug and trace subsystems. These subsystems can provide precise information about committed instruction branches and serviced exceptions. There are many common objectives among all existing test and debug architectures, not least the desire to simplify and speed up data extraction at runtime. Many standards and proprietary solutions aim to address these same needs [110–114].

Concerning FPGA multi-core architectures, processing units can be classified into two main groups: soft-core or hard-core processors. Soft-cores are flexible processor architectures fully defined in hardware description languages and implemented through FPGA fabrics. They are usually available in Hardware Description Language (HDL) files (e.g., VHDL, Verilog, or SystemVerilog) and can be freely modified or configured by developers. On the other hand, hard-cores are implemented on silicon as part of the overall IC chip and have bus and other direct interface connections with the FPGA fabric. Both categories can use the integrated programmable hardware logic to implement accelerators. Hard-cores generally achieve much faster operating frequencies, up to gigahertz speeds. They achieve this because the fabric's additional routing and architectural constraints do not affect them. Soft-cores can be

easily modified and tuned to specific requirements, and many parallel cores can be implemented in a single system, given enough resources.

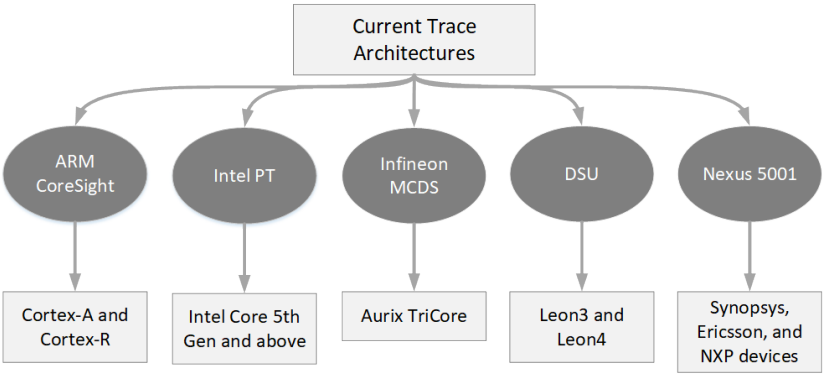


Figure 4.2: Diagram showing relevant trace architectures and the processors or companies that implement them.

Current solutions for FPGA SoCs with hard-core processors include Xilinx's Zynq-7000 [32] and Zynq-UltraScale+ [115], and Intel's Cyclone, Arria, Stratix and Agilex [116] device families. They are all based on ARM dual-core Cortex-A9 and quad-core Cortex-A53 general-purpose processors. All of these solutions have also implemented ARM's CoreSight Trace, and Debug architecture [117]. One example of a widely used soft-core processor in safety-critical aerospace applications is Gaisler's Leon3, Leon3FT (Fault-Tolerant), and Leon4 SPARC V8 based CPUs [76]. The fault-tolerant variant of the Leon family of processors comes with built-in fault detection and correction techniques. At the same time, the non-FT version is distributed open-source and widely used in academia. The Leon cores also implement a non-invasive tracing unit called the Debug Support Unit (DSU). This chapter aims to identify standard features across different trace architectures for control-flow monitoring purposes. For this comparison, four architectures previously targeted by both CFC and CFI implementations were chosen (ARM, Intel, Infineon, and Gaisler's Leon) and one international trace standard, the IEEE-5001 Nexus specification. Figure 4.2 shows the connection between the trace architectures explored in this chapter and the processors and commercial architectures that implement them.

4.1 Common Trace Architecture Features

Table 4.1 presents the evaluated architectures' features side by side. The Leon family of processors presents the least compression, as it outputs full opcode and PC values for every executed instruction. The Nexus and Intel PT standards present a heavy compression scheme that outputs only 1 bit of data for conditional branches, informing if taken or not, and do not automatically generate packets for direct branch operations. Except for Leon DSU, all architectures compress the target address information, only informing target address bits that have changed since the last packet. Designers of trace architectures heavily optimize trace data to minimize the use of internal memory buffers. This compression creates overheads for real-time analysis.

While Nexus 5001 and Intel PT do not directly support full target address tracking of direct branches, we can achieve it if we perform a transformation over the compiled program branches. Direct branches can be replaced by equivalent indirect branches, with direct target addresses stored in an available general-purpose register. Finally, all architectures present similar features and codification schemes, and techniques used to decode one can be similarly used for other implementations.

To fully monitor a given program, we require the ability to know the result of all control-flow and branching operations. Any executed branch also implies executing all sequential instructions in the program between it and the last branch. With each branch, we need to know what is its target address and, in

Interface	*Trace Compression*	*Target Address Tracking*	*Packet Sizes*
LEON DSU	Low	Full	16 Bytes
NEXUS 5001	High	Partial	2 to 8 Bytes
MCDS	Medium	Full	5 Bytes
Intel PT	High	Partial	1 to 9 Bytes
ARM CoreSight	High	Full	1 to 6 Bytes

Table 4.1: Relevant features for online monitoring in each of the evaluated Trace architectures.

the case of conditional instructions, if the branch was taken or not. The target address defines the next sequential block of instructions. In a typical program, this block will also end in a branch. Once a branch instruction is executed, and the following instruction address is resolved, we know the next branch to be executed if information about the running code is available.

The address of the next instruction from a conditional branch is not required in case the branch is not taken since it will always be the next sequential instruction in the code. However, more information is required to account for processor interruptions and other exceptions, as they can happen anywhere in the program code. If the address where an interruption happens is always knowable, the return address of its exception routine is also known. Some CFC techniques that focus on violating real-time requirements may also rely on timing information from the executed code. Table 4.2 describes how an instruction trace data packet of a protocol that follows these requirements might look.

Table 4.2: Description of an abstract instruction trace packet sufficient for general control-flow monitoring.

Data Component	*Description*
Current Address	Program address of the executed branch or at which an exception happens.
Target Address	Target address of a branch instruction or exception.
Branch Not Taken	Flag that is set only if a conditional branch was not taken.
Exception	Flag that is set if this packet corresponds to an exception.
Time Count (optional)	Amount of elapsed time since the last executed branch or received exception.

4.2 Debugging and Tracing Software Systems

The main use-cases of current real-time trace analysis are in debugging and verification [118]. In most commonly used embedded trace solutions, incoming trace data is first stored in temporary buffer memories before being analyzed offline. At the end of a traced execution run, a trace analyzer must

reconstruct the program flow and calculate the execution's structural coverage. This procedure's limits are the observation time bounded by the buffer memory's size and the additional computing time required for the program flow's offline reconstruction. Typical use-cases of hardware program tracing do not include online or real-time analysis. The information retrieved through tracing is not intended to be used by the actual running system but only during the development's design and verification phases.

Online program analysis can help solve these problems but has its own technical challenges. First, the new real-time analyzer must decode the compressed trace data stream before reconstructing the CPU's control-flow. Second, the generated data must be sufficient to reconstruct the program's actual flow path without assuming that target addresses are statically known, i.e., the trace data must contain the exact target address of executed branch operations. This last requirement is essential as an online trace monitor's primary objective is to identify dynamic anomalies in the program's control flow.

Hardware accelerators for online trace monitors have been used as solutions for both safety and security problems. In the safety domain, CFC techniques using program tracing have been proposed to identify control fault errors originating from hardware faults and random electromagnetic effects [65–67]. CFI security techniques have also been considered to prevent malware attacks from redirecting a program's control flow [119, 120]. However, these different techniques usually focus on a single target processor and specific trace features.

4.2.1 Differences between Tracing and Profiling

Profiling is a method for collecting events by sampling or taking snapshots of an entire object (software component, process, or event of the whole system) at specific intervals. If the sample rate is fast enough, all the events executed on the system can be essentially caught. Otherwise, any events executed in the time between sampling operations may be missed. The Linux OS implements the `perf_event_open` syscall, which can help users and designers to perform profiling at the system level. One of its primary functions is saving the state of the CPU's registers at every sampling action. Other tools like `bpftrace` also provide profiling capability, using the same datasource backend as perf (from `perf_event_output`). Profiling can help us have a broad view of a

system, but not a complete full picture. This lower observability comes from the nature of profiling and its dependence on sampling the system state. The sampling rate may not be able to observe all system changes and may also impact the observed system's performance. Sampling and reading operations can significantly impact the CPU performance or disk IO.

As its name implies, event tracing relies on trace data generation for specific and required events. Modern processors implement program and data tracing accelerators in hardware to mitigate the drawbacks of standard sampling profiling approaches. These accelerators enable users to configure filters to select which events are to be traced. In data tracing, these filters can be configured, for example, to generate trace packets for specific memory accesses or messages in the data bus. For program tracing, these trace and debug accelerators generate packets to track changes in the internal state of the target CPU. The easiest way to follow internal changes in the CPU is to trace changes to its control-flow. From the running software's point of view, all changes directly correlate to executed control-flow operations in the binary and imply the execution of any preceding sequential instruction. Whenever the program changes its control-flow, i.e., performs a branch operation, an event trace is generated, sent, and stored by the hardware's tracing subsystem. The importance of profiling and tracing modern high-performance systems has made program trace architectures extraordinarily complex and diverse. However, they are all solutions to the same problem. All trace solutions strive to provide a low-overhead fine-granular analysis of running applications. Thus, they ultimately implement similar functionalities, such as the crucial ability to know when branch operations are executed and the operation's target address destination. This chapter explores existing solutions implemented for hardware trace acceleration in some of the most well-known and commercially available processors.

4.3 ARM CoreSight Debug and Trace Architecture

The ARM CoreSight architecture comprises tools and components that offer debug and trace solutions for SoC designs [117]. The standard outlines components like trace macrocells, which allow for continuous collection of system

information [121]. The macrocell of processing elements allows for a complete tracing of indirect and conditional branches. They trace branch events from the most basic level, where only the execution of branches is informed, to higher levels that allow for full target address tracking. This last configuration is called Branch Broadcast and issues full target address information of all executed branches, including unconditional direct branches [122]. This feature facilitates the implementation of more powerful online trace monitors, but as shown in Chapter 6, trace data compression still poses difficulties for the decoder. The proliferation of modern ARM Cores in modern embedded systems and the presence of hybrid SoCs with integrated ARM cores makes the CoreSight architecture a "de facto" standard among modern systems.

The implementation detailed in this work focuses on this architecture, given the ubiquity of ARM systems and its list of overlapping features to other trace systems. This work utilizes leading development platforms to test this system, such as Digilent's Zynq Development Board (ZedBoard) [123] implementing a Zynq-7000 with a dual-core ARM Cortex-A9 processor. This platform implements a CoreSight architecture subsystem that provides full program trace information from the two CPUs to the FPGA through a dedicated interface. This interface is unique and not shared by other devices, meaning that its use does not impact the use of other peripherals, causing unwanted overheads.

A typical CoreSight system structure presents four main component types [124, 125]:

1. Control and Access components: these modules configure, access, and control the trace generation. They do not deal with the generation or the processing of data. Access components may include access ports (AP) such as the JTAG-AP or the Debug Access Port (DAP).

2. Sources: They are responsible for generating all trace data and providing it through a dedicated standard bus infrastructure called AMBA trace bus (ATB).

3. Links: They manage trace data flow throughout the ATB interconnect. They are the communication infrastructure between trace generating components and interfaces that communicate outside the subsystem.

4. Sinks: They are the endpoints for trace data inside the system. Some sinks can also aggregate the packets of multiple trace sources into a

single stream that can be read off-chip, stored in a dedicated internal buffer, or routed into a shared memory system.

The components of most interest to us include the trace *sources*, *links* and *sinks*. Trace sources are hardware modules that enable the tracing of software applications. They either provide low-level instruction traces from executing units or higher-level trace messages from specific instrumented code. This instrumentation is achieved through instrumentation trace units, called either ITM or Instrumentation Trace Macrocell (STM), that depend on the running program's more invasive modifications. Processor trace units focus on low-level non-invasive tracing and implement a dedicated interface that acquires information directly from each core's execution pipeline. These units are the Embedded Trace Macrocell (ETM) or PTM, depending on the target processor. Each Cortex-A53 core has a corresponding ETM module [126] while each Cortex-A9 core has its own PTM [121]. Each macrocell is specific to its designed processor, and its feature set may vary depending on the use cases of each architecture. This work focuses on the features shared by both version 4 of the ETM architecture and the PTM unit, including the trace of direct, indirect, and conditional branch instructions and software and hardware exceptions and interruptions.

Trace links include all interconnect components that provide specific features to transport, share, and filter data packets from trace Sources to Sinks. These components include bridges, clock and power domain crossing, packet replicators, and funnels, combining data streams and buffer components. They can also stall a given trace source based on the downstream components' ability to collect data and share synchronization requests through the bus.

Trace sinks are the final components in the trace bus chain. A single bus may contain more than one trace sink, where each sink is configured to collect distinct trace streams and overlapping data. They may cover a wide range of latency and bandwidth capabilities, depending on how they are implemented, configured, or on the destination of their trace packets, be it on-chip or off-chip. Essential sinks include the Embedded Trace Buffer (ETB), the Embedded Trace Router (ETR), and the Trace Port Interface Unit (TPIU). ETBs and ETRs are implemented as specific configurations of a more general component called Trace Memory Controller (TMC). They are responsible for storing trace data either in a dedicated circular buffer or routing it over an AXI bus directly to the system memory or any memory-mapped AXI slave. The TMC is usually

responsible for on-chip trace data storage, while the TPIU is used for off-chip trace data collection.

In Zynq-7000 and Zynq-UltraScale+ devices, the TPIU can be directly connected to specific external pins on the board or routed through the FPGA fabric to other interfaces. This second configuration can be used to connect the TPIU with a dedicated monitoring unit that will capture, decode, and process any incoming packets from ATB sources directly on FPGA. This direct FPGA connection gives us much greater flexibility than trace monitors that depend on external debug interfaces. The CoreSight architectures implemented for the Zynq-7000 and Zynq-UltraScale+ are shown respectively in Figs. 4.3 and 4.4.

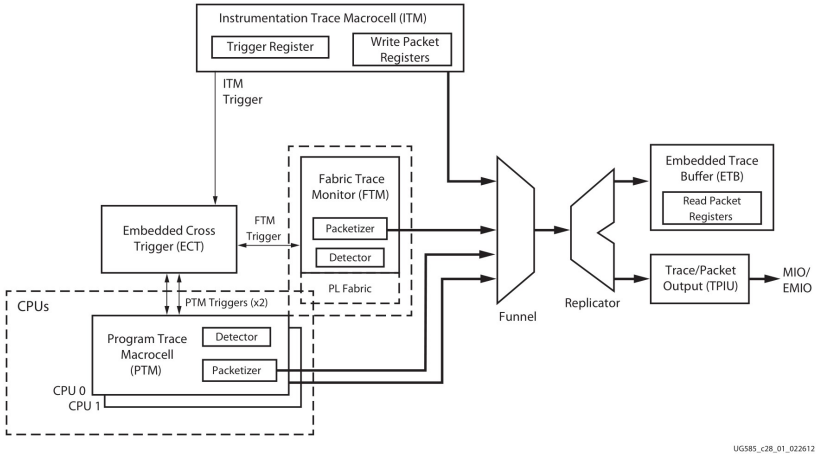


Figure 4.3: Zynq-7000 CoreSight system block diagram [32].

Implemented trace sinks include an ETB component in both architectures and one ETR present in the UltraScale+. ETBs are controlled and accessed by the processor running the traced application and are thus not suitable for the CFC monitor proposed here. The ETR, on the other hand, can be configured to continuously write trace data to an internal memory-mapped component implemented in FPGA. The disadvantage of accessing the ETR is that access to trace data must go through the main memory-mapped bus. All trace data will flow through the main system bus that the CPUs and application memory share, increasing bus bandwidth. This increase can cause bus contention for

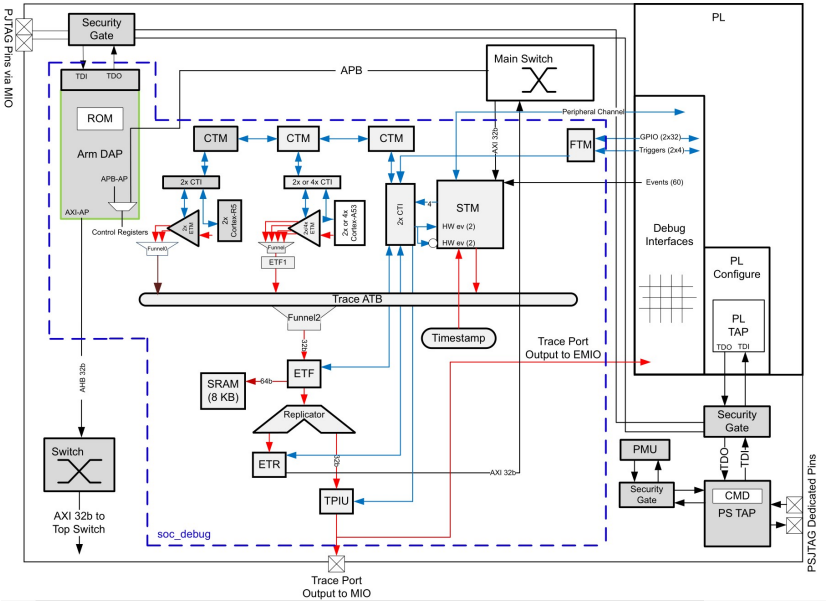


Figure 4.4: Zynq UltraSCale+ CoreSight system block diagram [115].

data-dependent applications, increasing the delay to program memory accesses and increasing timing overheads. Therefore, TPIU remains the best choice interface to collect all incoming packets without affecting the running system.

4.3.1 Program Tracing with PTM

The program tracing protocol implemented in the Program Trace Macrocell (PTM) units of Cortex-A9 cores follow what ARM calls the Program Flow Trace (PFT) architecture [122]. This protocol focuses on reducing trace bandwidth, relying on a minimal set of traced instructions and data compression. It assumes that the trace decompressor analyzing the data keeps a copy of the program or its relevant control-flow information. It only outputs enough information so that an analyzer can later reconstruct the program sequence. The architecture also supports tracking exceptions serviced by the CPU and

synchronization with other trace-generating sources through timestamps. It provides cycle count information regarding the number of CPU clock cycles elapsed between the execution of two traced instructions or events.

The PFT protocol names trace generating instructions as *waypoints*. The protocol defines a waypoint as all points in the program execution that may cause a change in its flow. PFT waypoints, or trace generating events, include:

- all indirect branches;
- conditional and unconditional direct branches;
- software and hardware exceptions;
- instructions that change the ISA state or Security state of the running core.

The ARMv7-A implements two main instruction sets, the ARM and Thumb instruction sets, also called A32 and T32 in 64-bit ARMv8-A architectures [36, 127]. All ARM instructions correspond to 32-bit instructions and correspond to the main ISA of the system. Thumb instructions have varying instruction lengths, either 32-bits or 16-bits, and can optimize code areas in some implementations. Table 4.3 shows a non-exhaustive list of relevant waypoint instructions from the ARM ISA. Any of the instructions shown may also be coded as a conditional instruction if the instruction operation code, i.e., `B`, `BL`, is followed by a condition code. In this case, a conditional direct branch is taken when the CPU state flags match the code modifier, e.g., `BEQ` executes a direct branch only if a previous instruction's side effects set the "equal to" flag to one. These branch instructions are commonly used for simple control-flow operations, procedure calls, and exception returns. Besides the branches listed above, any other operation that loads or modifies the PC will cause a change in the control-flow. These include operations that have the `PC` as a target register such as load from memory (`LDM`), or arithmetic operation, e.g., addition (`ADD`). Newer specifications deprecate this use of the PC, enforcing the strict use of branch operations to perform jumps in code execution [128]. The same is true for the general use of the LR as a target register in load or arithmetic operations [128].

The Cortex-A9 implements an 8-stage pipeline with out-of-order speculative execution [39]. It contains two instruction decoding units and 3+1 integer

Table 4.3: ARMv7-A operations relevant for control flow monitoring [121].

Instructions	*Details*
`B <addr>`	Unconditional Direct Branch to <addr>.
`BL <label>`	Call a subroutine to to a PC-relative address.
`BLX <label>`	Call a subroutine to to a PC-relative address and changes the instruction set to the one used by the subroutine.
`BX LR`	Return from subroutine. Branches to the instruction block pointed by the Link Register (LR) and changing instruction set.
`POP {..., PC}`	Load values from the stack to a list of registers that include the PC. Return from subroutine if a previous LR value is loaded to the PC.
`SUBS PC, LR, #4`	Used as a return from exception. Branches to the previous word aligned instruction from the one pointed to by LR.

execution units. Therefore, any executed waypoint instruction is only available to the PTM for trace generation once the CPU commits the instruction at the end of the pipeline on the write-back stage. The main functional blocks of the PTM are shown in Fig. 4.5. The PTM collects every waypoint instruction on the Input first in, first out (FIFO) and sends them to the packet generator when the CPU commits them. The Trace generator compresses the instruction signals and passes them to the main FIFO buffer that manages the ATB interface transfer. The Filtering and Triggering resources are used to define the start and stop points for trace generation, PFT configuration parameters, and synchronization with other CoreSight components.

The PFT architecture consists of a byte-aligned trace data stream and defines the following packets [122]:

- *Alignment Synchronization (Async)*: It identifies a boundary on the data stream, with the first byte of data after an Async packet being the header of a new packet.

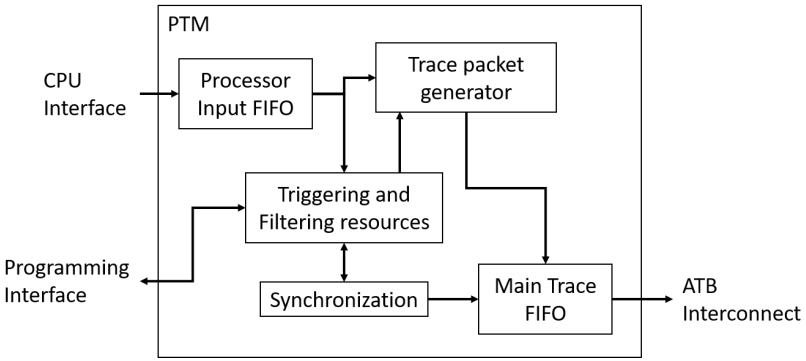


Figure 4.5: Cortex-A9 PTM functional block diagram [121].

- *Instruction Synchronization (Isync)*: This packet specifies the complete processor state for the next instruction to be executed.
- *Atom*: An Atom packet informs whether the branch instruction passed its condition code check. This packet has different encodings if cycle-accurate tracing is enabled or not. If cycle-counting is enabled, each packet holds the information of a single branch. If it is not enabled, each packet may inform the execution of up to 5 branches.
- *Branch Address*: This packet can imply the execution of two different control-flow operations. It can either imply the execution of a branch that passed its condition code check with the branch's target address or indicate an incoming exception with the corresponding exception vector address.
- *Waypoint Update*: It contains the address information of a non-waypoint instruction after it is updated to a waypoint. The PTM generates a Waypoint Update packet to indicate either the last executed instruction before an exception occurred or after the execution of more than 1024 sequential instructions.
- *Trigger*: It indicates the occurrence of a configured trigger inside the CoreSight subsystem.

- *Context ID*: This packet indicates a change in the Context ID register and signals that every instruction executed after it is run with this new ID.
- *VMID*: This packet indicates a change in the virtual machine ID (VMID) of the processor for architectures that implement virtualization extensions.
- *Timestamp*: A Timestamp packet enables correlation between multiple trace streams through an absolute timestamp clock shared among trace sources.
- *Exception Return*: It requires a single byte and indicates the return from an exception handler.
- *Ignore*: This packet has no effect. It is used to complete any unused bytes of a trace port if the PTM must output data.

According to PTM configurations and the amount of information that changed between consecutive processor states, these packets may vary in length. For all packets other than Isync, address information is compressed so that the packet only contains the differing least significant bits (LSB) since the last state. Our chosen PTM configuration does not focus on maximum trace compression but on outputting new control-flow information as soon as possible and minimizing decoder complexity. CFTC has the PTM broadcast all conditional or direct branches and configures cycle-accurate trace packets when possible. Branch broadcasting configurations force the PTM to generate Branch Address packets for all taken for all branches. The PTM will only use Atom packets if a conditional branch instruction is not taken with this configuration enabled.

Cycle-accurate tracing only affects the format of I-sync, Atom, and Branch packets. When active, this option includes in each packet an explicit cycle count. This cycle count always indicates the number of CPU cycles since the last packet output by the PTM. With this information, the precise execution time between any two branch instructions can be determined, i.e., the execution time of all sequential code blocks.

4.3.2 Program Tracing with ETMv4

At its core, the A53 retains similar execution characteristics to the A9. It comprises a 64-bit dual-issue in-order CPU with an 8-stage pipeline and 2+1 integer execution units [129]. The differences in in-order execution and smaller integer execution stage mean that each core will have a similar average number of executed instructions per cycle (IPC). However, it has more efficient floating-point units (FPUs) and can typically run at much faster frequencies, 1GHz in the ZCU102 instead of a maximum of 660MHz ZedBoard [123, 130]. The ARMv8-A ISA used in this processor implements two different architectures, the 32-bit AArch32 architecture compatible with the ARMv7 instructions with its A32 (previously ARM) and T32 (previously Thumb) instruction sets, and the 64-bit AArch64 architecture with its A64 instruction set [127]. Table 4.4 shows some of the existing branching operations implemented in the A64 ISA. Some significant differences between the 32-bit and 64-bit ISAs include the presence of system calls for virtualization, the requirement that only direct branches can be conditional, and the locking of the PC register for almost all instructions that are not direct branches.

The Cortex-A53 implements the latest version of the ETM trace source component, called ETMv4 [126]. This unit includes instruction tracing similar to the one presented in Section 4.3.1, with a its functional architecture presented in Fig. 4.6. The diagram shows many blocks similar to those available at Fig. 4.5. The main difference is the presence of an extra Processor Interface Block. In the ETMv4, this block is responsible for generating and encoding the instruction trace packet for executed instructions. The Trace Out block controls the Output FIFO to ATB interconnect and makes the clock crossing interface between the CPU clock and Bus clock domains.

One big difference from the trace protocol implemented by Cortex-A9 devices is that Cortex-A53's trace source module also traces speculative instructions. The trace generator sends data about the speculative execution of instructions and branches. It then uses "Commit" and "Cancel" data elements to either confirm or not the actual execution of an operation. The ETMv4 architecture defines trace messages called *instruction trace elements*. The information transmitted in these elements corresponds in part to the information of PFT packets. The instruction trace elements defined by this protocol and implemented in the Cortex-A53 ETM unit are [126]:

Table 4.4: ARMv8 A64 operations relevant for control-flow monitoring [126].

Instructions	*Details*
`B <label>`	Unconditional Direct Branch to <label>.
`BR <Xn>`	Unconditional Direct Branch to the address at register <Xn>.
`B.<cond> <label>`	Conditional Direct Branch to <label>.
`BL <label>`	Call the subroutine at <label>.
`BLR <Xn>`	Call a subroutine at the address pointed to by the register <Xn>.
`CBZ {<Xn>}, <label>`	Compare and branch to <label> if register <Xn> is zero.
`CBNZ {<Xn>}, <label>`	Compare and branch to <label> if register <Xn> is not zero.
`RET`	Return from subroutine to the address pointed to by LR.
`SVC #<imm>`	Causes a Supervisor Call exception defined by the 16-bit <imm> value.
`HVC #<imm>`	Causes a Supervisor Call exception defined by the 16-bit <imm> value.
`ERET`	Return from exception using the Exception Link Register (ELR).

- *Trace Info*: It provides a synchronization point for the trace analyzer and decoder to start interpreting the data stream.
- *Trace On*: Is used to indicate a discontinuity in the trace stream, either after the ETM is first enabled, when it re-enters a tracing code region after being idle, or when an information is lost after a trace buffer overflow.
- *Discard*: A Discard instruction trace element is generated if uncommitted instruction elements remain when the trace unit enters a state where it is no longer able to generate trace.

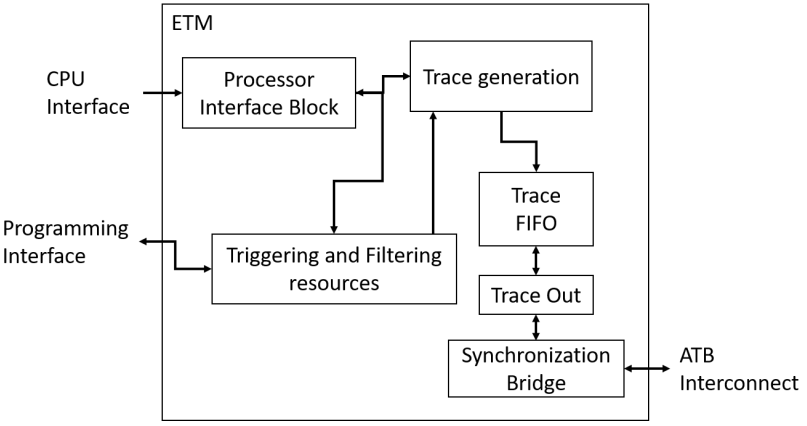


Figure 4.6: Cortex-A53 ETM functional block diagram. [129]

- *Overflow*: This element indicates an overflow of the internal trace buffer of the trace Source. Therefore some of the trace data might have been lost.
- *Atom*: An Atom element is used to trace branch instructions. For conditional branches, it informs if the branch was taken or not, while unconditional branches are always traced as being taken.
- *Q element*: These elements are used to indicate that at least one instruction, up to 'M', has been executed, including possible branches and other synchronization instructions.
- *Exception*: A Source generates this element whenever an exception occurs. It informs the type of exception, e.g., Reset, IRQ, or Hardware Faults, and the exception return address.
- *Exception Return*: This element indicates that an exception return occurred, with the previous and most recent Atom or Q element indicating the corresponding instruction that caused the return from an exception.
- *Address*: This element indicates the instruction set and instruction address of the next instruction to be executed. An address element follows any other trace element that informs or updates the current execu-

tion state. This includes: Trace Info and Trace On elements, indirect branches, direct branches on branch broadcast mode, exceptions, Q elements, and mis-speculation of instruction that must be updated.

- *Context*: This element defines the current execution context of the CPU and includes information about: the Context ID, Virtual Context Identifier, security state, exception level, executing ISA (either AArch32 or AArch64). This element is usually output whenever a change in the context state occurs.
- *Timestamp*: This element inserts a global timestamp into the instruction trace stream. A timestamp value corresponds to the most recently generated element among Atom, Exception, Event, or Q elements.
- *Cycle Count*: Cycle Count elements enable cycle-accurate tracing and are only associated to Commit elements. It indicates the number of processor clock cycles between two of the most recent Commit elements. Only cycle counts larger than a programmed threshold are generated to minimize trace bandwidth. The minimum threshold for the A53 is four clock cycles.
- *Event*: This element indicates when a programmed system event occurs. It also carries a value that identifies which event occurred.
- *Commit*: The Commit element is generated to indicate that Atom, Exception, or Q elements have been committed for execution. This trace element also indicates how many uncommitted elements must be committed.
- *Cancel*: The Cancel element indicates the number of most recent uncommitted instructions to be canceled. The ETM may cancel elements because a branch is wrongly speculated or an exception occurs before the CPU has time to commit the branch instruction. Any element that must be associated with a Commit element can instead be associated with a Cancel element.
- *Mispredict*: It indicates that the most recent Atom element has the incorrect taken or not-taken status. For direct branches, this element also indicates that the target address for the branch is recalculated.

Compared to the PFT protocol implemented on A9 processors, this architecture is much more complex. Mainly, it requires that the trace decoder handles which instructions are committed by the core and discarded instead of only outputting committed instructions. Nonetheless, ETMv4 implements all the features most important for our control-flow monitoring system. These include tracing all direct, indirect, and conditional branches and exceptions with explicit target address propagation.

4.4 Leon3 and Leon4 Debug Support Unit (DSU)

The LEON3 and LEON4 are synthesizable VHDL 32-bit processors compliant with the SPARC V8 architecture and developed by Cobham Gaisler AB [76]. They are also available as dedicated fault-tolerant devices focusing on aerospace applications available as LEON3FT and LEON4FT. Both architectures implement a trace and debug interface through their DSU. They implement a circular trace buffer that stores information about executed instructions.

Each entry of the trace buffer is 128 bits wide and gives detailed information about the executed instructions. They inform the full state of the program counter and the full opcode of the last executed instruction. Therefore, all executed instructions and their linear addresses can be identified. Once a branch instruction's opcode is recognized in the trace, the following entry's PC value will correspond to the branch's executed target.

Gaisler's Leon3 and Leon4 processors are SPARC V8 architectures with a 7 stage integer pipeline, with the Leon4 version also implementing branch prediction units [131]. Leon3 processor architectures are resilient and valuable for many safety-critical environments [132]. These processors can be found in radiation-hardened hard-core chipsets as well as in programmable logic architectures implemented as soft-core processors [133]. One example architecture is the InvasIC Network-on-Chip (NoC) [134, 135]. This platform promises a novel paradigm for the design and resource-aware programming of future parallel computing systems. For systems with several hundred cores on a chip, resource-aware programming is of utmost importance to obtain high utilization and computational and energy efficiency. Each processing node in this NoC can contain up to four configured Leon3 processors. The trace-based

method proposed here can be used as an error-detection solution in this or other performance-focused platforms.

The Leon3 and Leon4 cores implement each a version of a trace generator component called the DSU. The DSUs for each Leon3 and Leon4 are the DSU3 and DSU4. They both implement the same trace data information format, described in Table 4.5. They also implement circular buffers that collect all instruction and processor state information from the last stage of the pipeline, storing it before being sent to an external debugger. The lack of speculative or super-scalar execution makes this a good and straightforward architecture for non-invasive tracing. However, this approach creates a considerable bandwidth as each completed instruction generates 128-bit data packets, 1.6 Gbit/s for a processor running at 100 MHz.

Table 4.5: DSU3/4 - Leon3/4 Hardware Debug Support Unit trace data format [76].

Bits	*Name*	*Description*
127	-	Unused.
126	Multi-Cycle Instruction	Set on the second instance of a multi-cycle instruction such as load or stores.
125:96	Time Tag	Value of the 30-bit cycle counter implemented in the DSU.
95:64	Load/Store Value/Param.	Value of the loaded data or parameters of store operation.
63:34	Program Counter	Value of the PC register for the executed instruction.
33	Instruction Trap	Set if the instruction caused a Trap exception.
32	Processor Error Mode	Set if the instruction caused the processor to go into a Error Mode.
31:0	Operand	Opcode of the last executed instruction.

The DSU Time Tag generates complete cycle-accurate information for each executed instruction. It uses the CPU clock frequency and keeps counting as long as the processor runs and the DSU is enabled. The time tag value of two

consecutive branches can be compared to quickly obtain the number of CPU clock cycles between two control-flow points. The DSU also implements filters that can collect trace packets for instructions that cause control-flow changes. The possible filtered instructions include all subroutine or system routine calls and branch and trap instructions, including branch target instructions containing a branch's target addresses. These filtering options imply that conditional branches' results are always known after verifying the branch target packets' PC value.

Other similar works on CFC have already been implemented on Leon3 based processors [133, 136]. However, these control-flow monitoring techniques rely on the ability to modify the soft-core processor description. They require access to the instruction fetch stage of the CPU pipeline and be reliant on the particular Opcode and PC information given by the DSU instruction trace data as shown in Table 4.5. Therefore, these techniques are constrained to the particular implementation of the Leon architecture and cannot be easily applied to our target ARM processors.

4.5 IEEE-ISTO NEXUS 5001 Standard

Nexus 5001 is a debug standards initiative based on the IEEE ISTO 5001 debug specification that addresses the diverse challenges for embedded-processor and digital-system debug interfaces. Developers require more comprehensive debug features and will benefit from more standardized interfaces to address the constant increasing complexities of modern embedded applications (data communication, automotive powertrain, computer peripherals, wireless systems, and other control applications)

The initial 5001 Nexus specification was developed in 1999. It is an IEEE standard for debug interfaces of embedded systems and processors. It has since been implemented in numerous devices, such as Synopsys, Freescale, and NXP processors and microcontrollers [114]. It defines four device implementation classes with different compliance requirements. Classes 2 and above require real-time program tracing features such as what is needed to enable monitoring. The standard outlines program trace packets that provide branch target address information only for indirect branches. On the other hand, direct branch packets

provide only the number of executed instructions since the last branch, while the target is assumed to be statically known.

This architecture is based on a packet-based messaging scheme, which supports debugging complex multi-core systems. Control of the multi-core debug processes based on a transaction protocol (TCODE) that allows data to be sent in packets, using a packet header to provide information on the source and assumed destination of the data on-chip components as well as information on the subsequent data packets containing trace or other information. This simplifies the interleaving of multiple trace sources and concurrent communication with multiple Nexus instruments. Each message is made up of a 6-bit TCODE header followed by a variable number of packets. The Nexus specification defines a standard set of TCODEs for common identification and trace operations, which is also extensible to user-defined debug commands. The standard defines fixes a value of 4 for the TCODE that precedes program trace packets, see Table 4.6.

Table 4.6: Indirect Branch packet format for Nexus protocol [114].

Program Trace Packets		
Min Size (bits)	*Name*	*Description*
6	TCODE	Value = 4.
0	TSTAMP	Number of cycles message was held in the buffer or the full timestamp value.
1	U-ADDR	The unique portion of the branch target address for a taken indirect branch or exception.
1	I-CNT	Number of instruction units executed since the last taken branch.
0	B-TYPE	Branch type.
0	SRC	Client (Processor) that is source of message.

Nexus program trace is limited to instruction discontinuities, e.g., branches, conditional jumps, and interrupts. By mapping these values to an assembled

program, trace analyzers can interpolate branch locations in the program and reconstruct the instruction flow.

4.6 Aurix Multi-Core Debug Solution (MCDS)

Infineon MCDS is a multi-core debug solution developed for their Aurix processor devices. Similar to other debug solutions, it consists of configurable IP building blocks, which provide trace compression, trace qualification, timestamping, and complex cross-target triggering. It also enables simultaneous measurement of different performance indicators with timestamped trace results.

Figure 4.7 shows the MCDS subsystem architecture example consisting of a trace kernel and on-chip trace memory (TMEM). In this example, communication between the on-chip debug environment and the debug tool is implemented based on a JTAG TAP with optional data trace interfaces. As a previous member of the consortium behind the Nexus 5001 standard, its interfaces are partially compliant with Nexus ports, see Section 4.5. Each debug target, i.e., processor core, is connected to the MCDS through an adaptation logic block. However, the design of its trace component block may be target-specific.

Each block adapts the target's custom interface to a generic, standardized interface that MCDS uses. It also synchronizes signals from the target side to the clock domain of the MCDS in case they are in different clock domains. The architecture of the MCDS kernel depends on the number and type of debug targets and consists of so-called observation blocks (OB), a multi-core cross-connect (MCX), and a debug memory controller (DMC), with MCX being connected to all OBs and the DMC. It is responsible for distributing programmable cross-triggers and provides a central timestamp for all trace messages. Additionally, MCX provides many counters, which can count events and trigger an action after an event has occurred N times or a specific period has elapsed. MCX provides the functionality to observe a system with multiple processor cores. Interactions between multiple cores are also traceable, and complex conditions can be evaluated to recognize a specific event. Each target signal within the SoC is connected to its dedicated OB. Within this block, trace qualification and trace message generation take place. Each OB may contain several custom trace units of different types.

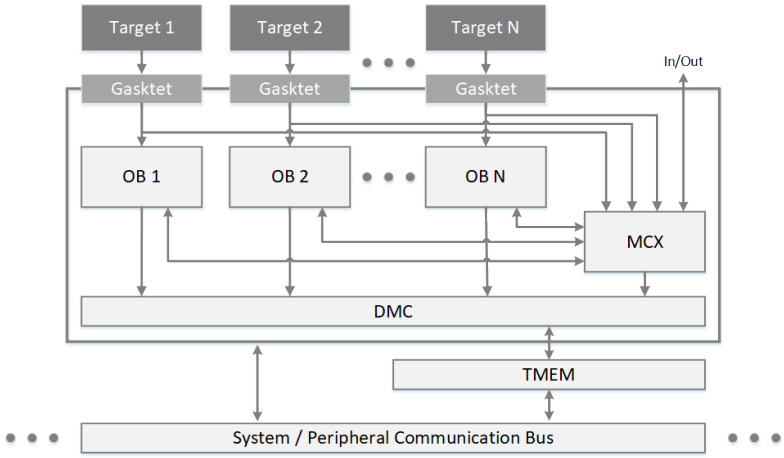


Figure 4.7: MCDS subsytem architecture [137].

The primary trace interface provided by MCDS is a synchronous tagged data protocol without handshakes. This type of interface is similar to ARM's TPIU, requiring a reader interface that must match the interface's frequency. The sender places the data in well-defined packets on the data port and concurrently indicates which kind of packet is present through the mode port. The mode port must express at least two different values: *Idle* and *Valid*. The smallest common implementation of this interface for program branch tracing has two parts:

- **Base Address:** After power on and after each discontinuity of the control–flow, the trace logic needs to know the exact and complete current PC value.
- **PC Increment:** Once the base address is known, only incremental updates are sent to keep the local copy in sync with the original PC.

The interface defines discontinuities as either:

- **Direct Branches**: They are caused by jump instructions in the executed program. In such cases, the target address is a constant in the source

code and can be easily obtained. A branch of this kind is indicated by *Forget*(see Table 4.7) on the increment port.

- **Indirect Branches**: They occur from jump instructions with calculated target–address, e.g., a return from a subroutine, or by exceptions, e.g., interrupts and traps. The target address must be contained in the trace memory in these cases. *Forget* on the base port is used to indicate such a branch. Each *Forget* received on any of the two ports invalidates the current base address. The exact protocol definition is given in Table 4.7.

Table 4.7: MCDS program trace protocol with Mode and Data port pair meanings [137].

Mode Port Encoding	*Data Port Description*
	Base Address Protocol
`Idle`	Data port holds no value.
`Valid`	Target address after a preceding discontinuity, optionally the current instruction pointer otherwise.
`Forget`	Target address after a preceding discontinuity. This base must only be used for one clock cycle and discarded thereafter. Don't care otherwise.
	PC Increment Protocol
`Idle`	Data port holds no value.
`Valid`	Instruction pointer increment. This is the number of bytes the instruction pointer was advanced since the last time the Mode Port was not `Idle`.
`Forget`	Instruction pointer increment. This is the number of bytes the instruction pointer was advanced linearly since the last time the Mode Port was not `Idle`. In case of a taken branch this includes the branch instruction.

4.7 Intel Processor Trace (PT)

The Intel Processor Trace (Intel PT) is an extension of Intel processor architectures that captures information about software execution using dedicated hardware facilities that cause minimal performance perturbation to the software [138]. Intel PT's implementations offer control-flow tracing, which generates a variety of packets to be processed by a software decoder. The packets include timing, program flow information, e.g., branch targets, branch taken/not taken indications, and program-induced mode related information, e.g., Intel TSX state transitions, CR3 changes. For monitoring purposes, this architecture generates two main types of control-flow data packets: one for conditional direct branches, called Taken Not-Taken (TNT) packets, and one for indirect branches, called Target IP (TIP) packets, from instruction pointer (IP) the given for the PC or instruction address counting register inside the Intel architecture. An additional packet type called Flow Update Packet (FUP) is also available. FUPs provide the source IP addresses for asynchronous events (interrupt and exceptions), as well as other cases where the source address cannot be determined from the binary.

TNT are analogous to CoreSight Atom packets. They track the *direction* of conditional branches, i.e., the outcome of the branch signaling if it was taken (executed branch) or not taken (branch not executed). The format used for TNT packet is shown in Fig. 4.8. This packet has two sub-format: a short TNT and a long TNT. Short TNTs packets can hold information related to between 1 and 6 executed conditional branches. In contrast, long TNTs packets can hold between 1 and 47 bits, each related to a single branch's execution condition. The packet's payload bits are interpreted as:

- `1` indicates a taken conditional branch, or a compressed `RET` instruction (procedure return).
- `0` indicates a not-taken conditional branch.

TIP packets are the Intel version of the Branch Address packets present in ARM's CoreSight architecture. The processor generates this packet type whenever the execution flow encounters an indirect branch (including un-compressed return instructions), far branch, interrupt, and exception. It contains a header with value `0xD`, `01101` in binary, and up to 8 bytes containing the new least

	7	6	5	4	3	2	1	0	
0	1	B_1	B_2	B_3	B_4	B_5	B_6	0	Short TNT

(a) Short TNT

	7	6	5	4	3	2	1	0	
0	0	0	0	0	0	0	1	0	Long TNT
1	1	0	1	0	0	0	1	1	
2	B_{40}	B_{41}	B_{42}	B_{43}	B_{44}	B_{45}	B_{46}	B_{47}	
3	B_{32}	B_{33}	B_{34}	B_{35}	B_{36}	B_{37}	B_{38}	B_{39}	
4	B_{24}	B_{25}	B_{26}	B_{27}	B_{28}	B_{29}	B_{30}	B_{31}	
5	B_{16}	B_{17}	B_{18}	B_{19}	B_{20}	B_{21}	B_{22}	B_{23}	
6	B_8	B_9	B_{10}	B_{11}	B_{12}	B_{13}	B_{14}	B_{15}	
7	1	B_1	B_2	B_3	B_4	B_5	B_6	B_7	

(b) Long TNT

Figure 4.8: Intel PT TNT packet format [138].

significant bytes of IP since the last transmitted packet, see Fig. 4.9. Anytime a TIP is encountered, it indicates that the CPU transferred control to the IP provided in the payload. The source of this control-flow change, and hence the IP or instruction to which it binds, depends on the packets that precede it. If this packet is encountered in the trace stream and all preceding packets have already been bound, it will apply to the upcoming indirect branch or similar jump instruction.

	7	6	5	4	3	2	1	0
0	IPBytes			0	1	1	0	1
1	TargetIP[7:0]							
2	TargetIP[15:8]							
3	TargetIP[23:16]							
4	TargetIP[31:24]							
5	TargetIP[39:32]							
6	TargetIP[47:40]							
7	TargetIP[55:48]							
8	TargetIP[63:56]							

Figure 4.9: Intel PT TIP packet format [138].

FUPs are analogous to CoreSight Waypoint Update Packets, its format is shown in Fig. 4.10. It is generated by asynchronous Events, such as interrupts and

exceptions, and provides the address where the asynchronous event occurred. It is never sent alone, always sent with an associated TIP, and potentially other packets. If there were a preceding unbound FUP, it would bind to the next TIP packet. Therefore, the TIP provides the target of the corresponding asynchronous event or abort that occurred at the IP given in the FUP payload.

	7	6	5	4	3	2	1	0
0	IPBytes			1	1	1	0	1
1	IP[7:0]							
2	IP[15:8]							
3	IP[23:16]							
4	IP[31:24]							
5	IP[39:32]							
6	IP[47:40]							
7	IP[55:48]							
8	IP[63:56]							

Figure 4.10: Intel PT FUP packet format [138].

FUPs header value corresponds to `0x1D`, `11101` in binary, and its payload follows the same format as TIP packets, compressing the IP value to only the bytes that changes since the last sent packet.

4.8 Summary

This chapter presented the functionalities and protocol specifications of different commercial off-the-shelf (COTS) processor architectures. One central point of interest is the similarities between many of these architectures, such as between ARM and Intel program traces and between Infineon MCDS and the Nexus Standard. Each one implements the same functionalities in different ways.

Once having analyzed the trace architectures of Cortex-A9, Cortex-A53, and Leon3 processors, we can now look at their similarities, focusing on the minimum required data that could be used for a proper control flow monitoring implementation. As such, this chapter proposed an abstracted and general instruction trace protocol able to give enough information for monitoring or trace

analysis tools to recreate all control-flow sequences. This definition is later used in Chapter 6 to implement an architecture-independent CFC monitor.

Comparing the information required by the generic interface defined here and the trace protocols in the previous sections, we see that they all contain the required features. CoreSight based protocols do not output the current instruction address as the DSU based protocol does, but it generates special packets before exceptions that serve the same purpose. On the other hand, while CoreSight protocols are always able to output the destination address of taken branches, the DSU does not. It overcomes this by later outputting a packet for the next executed instruction after any branch operation. Therefore, the taken target address is known, as DSU packets always contain the PC's value. This analysis shows that these instruction trace protocols are equivalent, and it is possible to create a component that transforms the information acquired through these three protocols into a single generic interface that follows the layout described in Table 4.2. As a result, any control flow monitoring mechanism based on this generic interface will be usable by any architecture accompanied by a proper protocol translating module.

A distinction is sometimes made in these systems between direct and indirect branches, in which cases only indirect branch targets are explicitly given in the trace packet. Quick transformation can be performed at the binary level, transforming direct branch operations into indirect ones and storing its branch target in a register or dedicated memory region near the function space. This transformation causes little to no overheads to the execution and allows the generation of complete trace packets by any of the protocols described here. Our proposed functional safety concept for CFE detection can be employed in any system that contains branch target address propagation for all taken branches. Analyzing traces from simultaneous execution units can provide a more comprehensive temporal and logical monitoring of a program's sequence.

5 Binary Program Analysis for Embedded Trace Monitoring

Most programs are written in high-level languages like C or C++, which processing systems cannot run directly. However, to be admissible, each transformation requires a set of static guarantees, in this case, for global control-flow analysis and refinement. As the *de facto* standard in the embedded domain, the C language is somewhat tricky in this sense. Its permissive, low-level memory model and type system emphasize aggressive local performance optimizations over accurate global analysis. Consequently, either some language features have to be omitted (e.g., function pointers, computed goto's) to again facilitate global analysis, or one can resort to type-safe languages, such as Java, in the first place.

5.1 Generating the Binary Application Code

Binary executable files are produced through the *compilation* of text files that conform to specific language syntax and structure. This process translates human-readable source codes, such as C or C++, into machine code that processing units can execute. The steps involved in the compilation process of a typical C code are shown in Fig. 5.1. The full compilation process involves four phases: *pre-processing*, *compilation*, *assembly*, and *linking*. In modern compilers, some or all of these phases may be merged. Each step nonetheless corresponds to a required and defined output or state in the chain.

The *pre-processing* stage is responsible for combining the header and source files that make up the design and resolving all pre-processor instructions in the code. In this phase, any `#define` and `#include` directives are resolved and expanded so that the resulting output is pure C code ready to be compiled. After this phase is complete, the source is ready to be compiled.

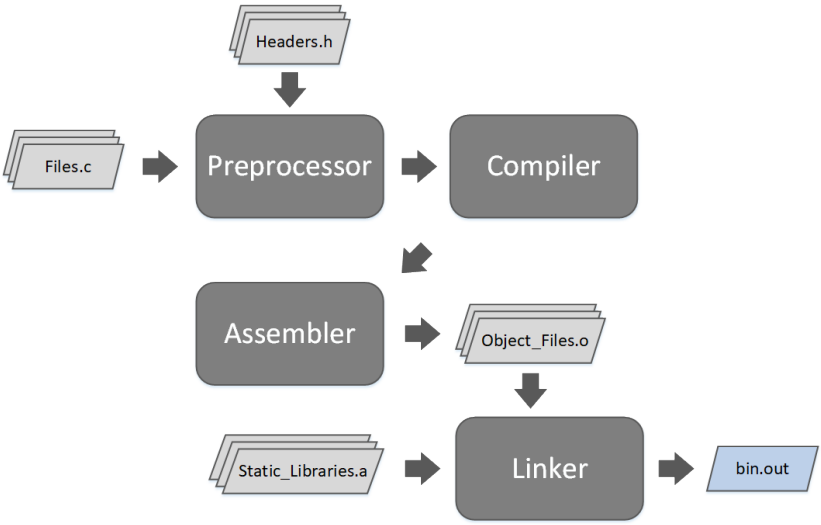


Figure 5.1: C compilation process from source to executable binary.

The compilation phase translates the pre-processed code to the target processor's assembly language, e.g., ARM's AArch32, AArch64, or Intel's x86_64. This step is where most compilers typically perform their hardest code optimizations. GCC for example allows for the configuration of *optimization levels*, such as `-o0` and `-03`, in this phase. Section 5.1.2 explains the effects of some of the most relevant options.

The *assembly* is tied to the final generation of machine code from the still human-readable assembly. This phase's inputs are the set of processor-specific assembly language files, and its output is a set of *object files*, or *modules*. These files contain, in theory, machine-executable code but just from their corresponding source code files. Usually, one source file will generate one assembly file, later turned into an object file. Therefore any links and function calls, external libraries, and the entry point to start the code execution may be missing in object files.

The final phase of this process is the *linking* phase, performed by a program called a *linker*, usually separate from the primary compiler tool. All machine code object files are linked and put together to create a single executable binary.

Some modern compilers introduce at this stage additional optimization passes, called link time optimization (LTO). Object files are generated independently, without assumptions of their memory placements or particular base addresses. The linker's job is to take all the object files belonging to a program and merge them into a single and coherent executable, typically intended to be loaded at a particular memory address. The linker is also responsible for resolving all symbolic references, i.e., all references to data or code addresses unknown to the specific object.

Depending on the program's types of libraries, the linker may also possibly resolve any library calls present. Static libraries are merged into the final binary executable, allowing any references to be completely resolved. Dynamic or shared libraries are loaded in memory separately from the program and are shared among all programs that run inside the same system.

After the compilation process is over, the generated executable file is loaded into memory before the processor executes it. This object usually follows a file format depending on the underlying OS or program loader, in the case of embedded systems with no underlying OS. In 32 and 64-bit Microsoft Windows systems, this is the portable executable (PE) file format, while executable and linkable format (ELF) files are used in most Unix, Unix-like, or embedded systems. Our analysis is performed by retrieving assembly level data from a finalized ELF executable file in a process called *disassembly*.

5.1.1 Binary Execution and Memory Maps

Executable binary files have to follow their own standards depending on system and processor architecture. Any system must conform to an application binary interface (ABI). An ABI defines how data structures or computational routines are accessed in machine code, which is a low-level, hardware-dependent format. These standards typically describe the different processor instruction details, such as the register file structure, heap and stack organizations, and memory access types. Similarly, an embedded-application binary interface (EABI) specifies standard conventions for file formats, data types, register usage, stack frame organization, and function parameter passing of an embedded software program for use within an embedded system environment.

Typically, in systems with separate virtual and physical address spaces, the loader utility will load the program to its physical range and configure the MMU accordingly. Inside the user application, all memory accesses are done using virtual memory addresss (VMAs). These addresses are directly encoded in the program's binary. A program's corresponding physical memory addresss (PMAs) are only known after MMU translation, allowing complex OSs to reallocate all or parts of a program to manage a small shared memory space better. This translation scheme means that a processor always deals internally with VMAs. Therefore, the target of branch operations as executed by the processor will match the same addresses used inside a program's binary. From the processor core's point-of-view, as it fetches and executes instructions, these addresses are the only available, as the translation step is invisible. Program trace mechanisms also typically use this virtual representation when operating, directly matching the binary machine code.

Figure 5.2 present an example of a loaded ELF binary inside a Linux-based platform. While stack organization and region of growth depend on the specific ABI or EABI, all modern systems implement a stack architecture variation. The stack holds function local values and is used to save the CPU's state before a procedure call. When the current function comes to an end, the previous function's state is restored. For trace monitoring purposes, the only value of interest saved inside the stack is the LR. This value corresponds to the return target address of the currently running function and must be saved to the stack. Otherwise, when nested functions are called, the values would be lost.

When procedure calls are identified inside the binary, the following sequential address corresponds to the return address. While the exact function may have multiple allowed return addresses, a single calling instance of a function has only one possible return point. Our analysis model focuses on these calling instances, identifying a function's return address dynamically rather than statically. This dynamic value identification reduces the total number of outgoing paths in the CFG of a function, simplifying the data that need to be stored and used by the monitor.

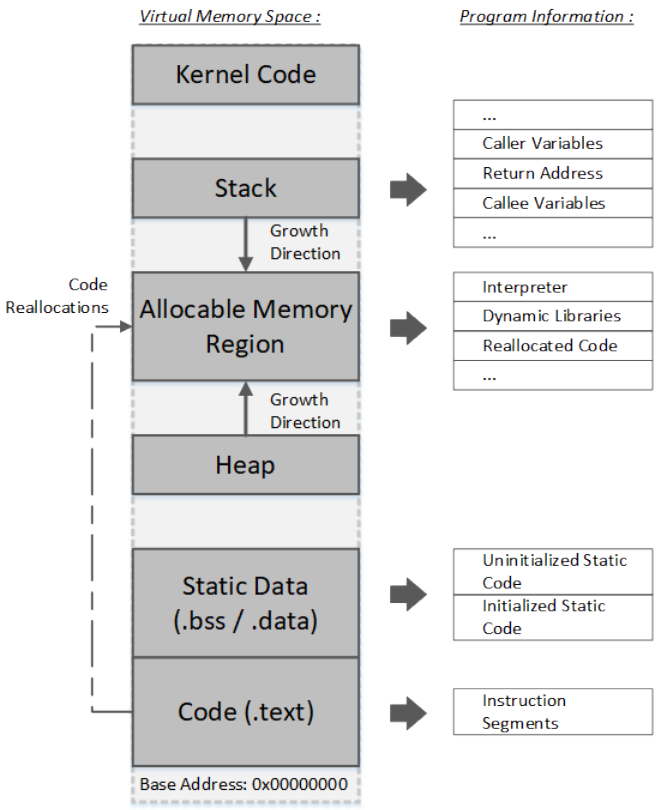


Figure 5.2: Memory map and load scheme of an ELF binary inside a Linux-based system.

5.1.2 Control-Flow Optimizations

In compilation tool suites such as GCC, parameters can be passed before starting the compilation process to optimize the resulting binary code. These optimizations can target different measures such as speed performance or total code area. Therefore, a single source code program can have multiple valid machine code representations in practice. GCC was our compiler suite of choice, it implements 5 optimization levels: no optimization (`-O0`), "optimize"

(`-O1`), "optimize more" (`-O2`), "optimize yet more" (`-O3`), and optimize for size (`-Os`).

Each optimization level, from 1 to 3, adds different optimization techniques so that `-O3` implements all techniques in `-O2` plus additional ones. Chapter 7 presents tests of multiple benchmarks in two optimization configurations: `-O0` and `-O3`. Using `-O3` can drastically change the resulting binary control-flow. Most control-flow optimizations only change internal loop structure or branch targets to minimize code or reduce the number of required branches for a function. These effects are transparent when generating the CFG from the binary code. However, the hardware monitor's use of a dynamic function return identification required that all function start and end blocks be well defined. The most important optimization techniques that can complicate the control-flow analysis are tail call and tail merging optimizations and function inlining.

When a function F_1 returns the value of a call to a second function F_2, this is a *tail call*, see Fig. 5.3. When the compiler optimizes a tail call, it removes the return block of the function F_1 and the preamble of the function F_2 and substitutes it for a direct branch operation. Therefore, when F_2 ends, it can return directly to the caller of function F_1, as if F_1 and F_2 were a single function.

Tail merging, also called cross-jumping optimizations, applies to basic blocks whose last few instructions are identical and continue to the same location. It replaces the matching instructions of one block with a branch to the corresponding point in the other, see Fig. 5.4. Inline optimization (Fig. 5.4) work similarly by optimizing away function calls and returns. Instead, the function's CFG is completely inserted into the calling point as if it were a part of the caller's code. These optimizations make it harder to identify where a function starts and another function end. This identification is essential for our monitor because procedure calls and procedure returns are treated differently by the CFC hardware.

5.1.3 Coding Guidelines for Critical Systems

Current standards outline many coding practices and rules that should be followed by software components so that the design and implementation shall

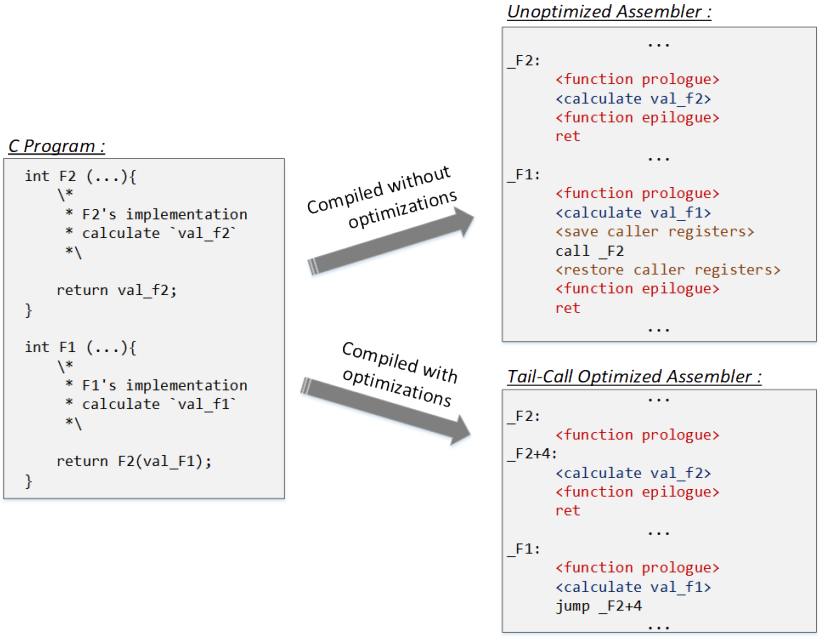


Figure 5.3: Example of function tail call optimization in binary code.

avoid unnecessary complexity and achieve testability and maintainability. Given the importance of code verification and branch coverage and the dangers of errors in the control-flow, the standards outline guiding principles for what types of calling conventions and code structures are allowed. The most important of these guidelines is MISRA-C:2012, the third edition of the MISRA-C rule set made in compliance with the C99 C language standard. This standard is the base for most coding rules for C/C++ systems, like those described in the ISO-26262 standard and the AUTOSAR guidelines [139].

The described rules focus on code manageability, clarity, and avoidance of structures that facilitate errors. Usage of function and instruction pointers, for example, is one of the major concerns. Indirect branches, such as those that target a function or other instruction address locations, make it much more challenging to analyze and statically generate a binary control-flow graph.

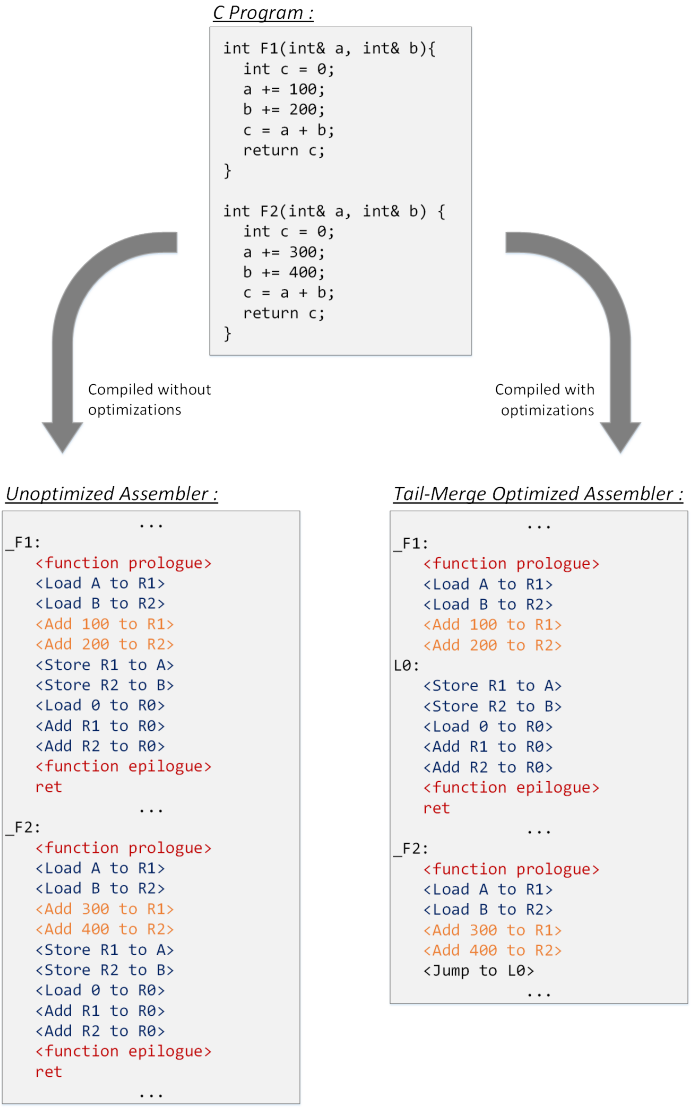


Figure 5.4: Example of function tail merge optimization in binary code.

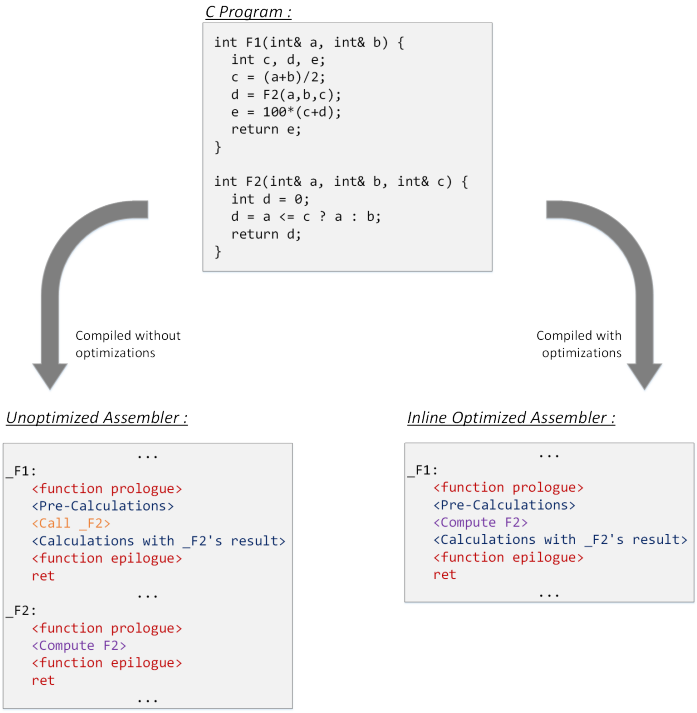


Figure 5.5: Example of function inline optimization in binary code.

These software guidelines clarify that every instruction pointer must not be part of pointer arithmetic calculations and must be explicitly stored in memory. Therefore, every register associated with an indirect branch or call operation target address should have its range of possible or allowed values immediately known through static code analysis.

5.2 Binary Control-Flow Analysis

The binary analysis framework uses a code structure called a control-flow graph (CFG) (Fig. 5.6) to reconstruct the program's internal structure and its func-

tions after disassembly. This structure reduces the program to only its basic branch control-flow information. It has been demonstrated that any program can be described through a directed control-flow graph, where all program branches correspond to edges and where its vertices are the sections of sequential code between two branches [87]. A walk in this graph from its input vertex to its output codifies a complete and valid program execution. Therefore, only the information about which edges have been taken is required, i.e., its program branches, to follow any program execution in its entirety. By acquiring a program's binary CFG, we have all the information needed to verify if a sequence of program trace packets is correct or not. However, this model best monitors single tasks with no dynamic scheduling and no asynchronous servicing of interruptions. Our program's final representation is that of a disjoint set of connected CFGs. Each fully connected CFG corresponds to a single function or interruption handler. Whenever an asynchronous interruption occurs, the current CFG is switched with that of the calling interruption handler until it is finished or a new higher priority interruption is called. After it is over, the last CFG resumes its execution.

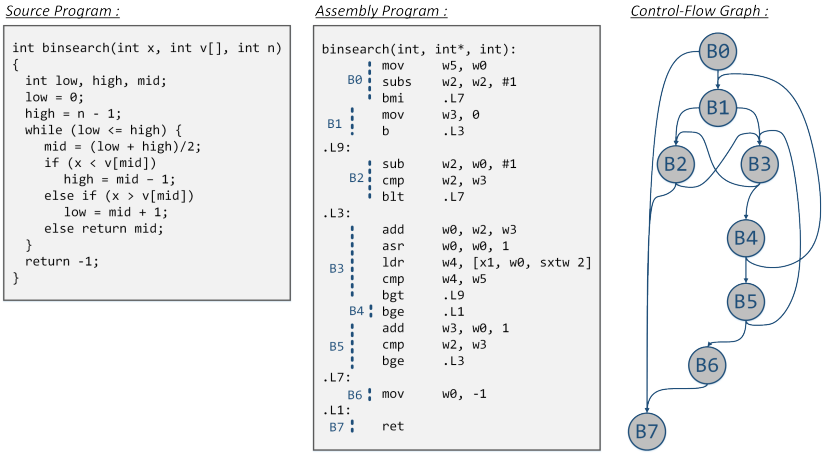


Figure 5.6: Control-Flow Graph model of an application.

The process used to perform the code disassembly and generate the final CFG is shown in Fig. 5.7. First, the machine code is transformed back to its assembly representation with a utility such as GCC's *objdump* tool. Symbolic

information contained in the executable is used to identify labels and variable names. After that, a single sequential pass is performed on the assembly code, breaking down new blocks for every identified branch and procedure call operation. When performing indirect call analysis, context-sensitive must be taken into account. Therefore the second pass is used over the blocks to identify any missing targets that the first sequential pass could not directly identify.

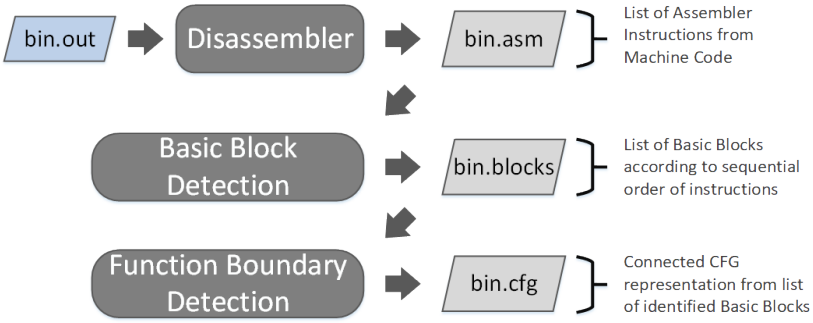


Figure 5.7: CFG generation process from the executable binary code.

Once the tool performs the second pass, all basic block vertices of the CFG are fully connected, and all functions are identified. An example CFG representation is shown in Fig. 5.8, based on the C code presented in Listing 5.1. Each block can represent three main types of nodes: a standard direct branch, procedure call, or procedure return. Their main attributes besides branch type are the pointers to the *target* and *next* nodes. Target nodes represent those blocks containing the branch operation's target address that defines the end of the basic block. The Next nodes are those that start with the first instruction that follows the end of the block.

Listing 5.1: Example Quicksort C code.

```
1 void swap (int* a, int* b) {
2   int t = *a;
3   *a = *b;
4   *b = t;
5 }
6
```

```
7  int partition (int arr[], int low, int high) {
8    int pivot = arr[high];
9    int i = (low - 1);
10   for (int j = low; j <= high- 1; j++) {
11     if (arr[j] <= pivot) {
12       i++;
13       swap(&arr[i], &arr[j]);
14     }
15   }
16   swap(&arr[i + 1], &arr[high]);
17   return (i + 1);
18 }
19
20 void quicksort (int arr[], int low, int high) {
21   if (low < high) {
22     int pi = partition(arr, low, high);
23     quickSort(arr, low, pi - 1);
24     quickSort(arr, pi + 1, high);
25   }
26 }
```

Direct branches are the most common and always connect to another block through their branch target address. Procedure call blocks are similar to direct blocks, but the monitor must execute additional operations given their block type signals. If the monitor identifies a procedure call in the trace stream, the Next block of this Procedure Call should be saved to its LR stack copy. It uses this value to verify the target of the corresponding Procedure Return block of this call. Any block can also contain a conditional type, meaning its next block is an allowed possible jump target. As trace mechanisms usually do not provide the address of a resulting not taken branch, this address must be part of the CFG so the monitor is always aware of the current running basic block and its corresponding branch operation.

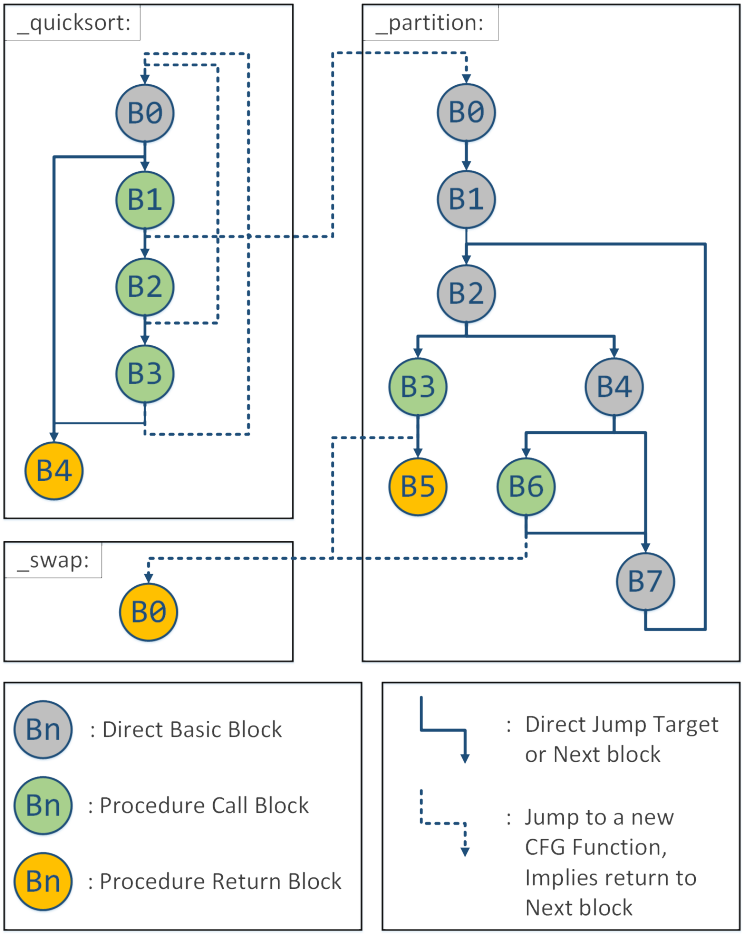


Figure 5.8: Resulting CFG generated from example recursive Quicksort implementation.

5.3 Program Trace and the Control-Flow Graph

As presented in this chapter, a CFG is generated from the application's low-level code to codify its allowed control-flow state changes. Signatures and check operations are executed for each basic block and CFG transitions. A standard CFG is composed of basic blocks that only have one entry point and one exit point. Therefore not all transitions between blocks correspond to branching instructions. A block may end while another may start in the middle of a sequence of non-branch instructions if one of those operations targets a branch somewhere else in the code.

However, instruction branch tracing only provides information for executed branch instructions, so we cannot follow all transitions in a traditional CFG. Therefore, a reduced CFG representation must be used. This new reduced graph (G_R) is defined as the set of reduced vertices (V_R) and edges (E_R) where $G_R = \{V_R, E_R\}$. Each reduced edge in this representation corresponds directly to a branch operation obtained from our binary analysis of Section 5.2. Therefore, this graph has no transitions that do not relate to a binary branch operation in the program's code. All implicit block transitions between two sequential instructions are removed with this new definition, affecting the meaning of basic blocks. Traditionally, basic blocks only contain one entry point and one exit point. By removing implicit transitions, our BBs can have multiple entry points but maintain a single exit point. This exit point will always correspond to one or two edges, one for the target of its branch operation and one for its sequential block, in case of conditional branches that are not taken. Finally, a reduced vertex is a sequence of instructions that execute sequentially with possible branches coming into the sequence and always has a branch at the last instruction.

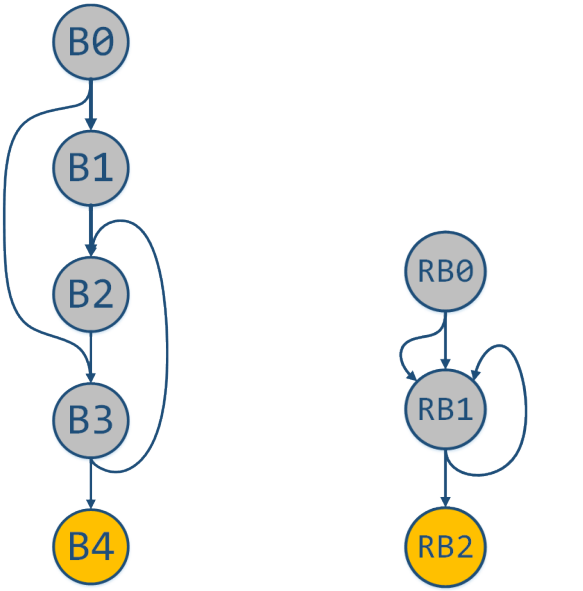
Essentially, this corresponds to an edge contraction operation over the classical definition of CFG as showed in Section 3.3. The graph's resulting blocks are defined as a chain of sequential instructions that end at a single branch, with the first instruction of each block directly succeeding a branch. Figures 5.9a and 5.9b show the corresponding classic and reduced CFGs of an example ARMv7 assembly code presented in the Listing 5.2. In the reduced graph shown, the original blocks $B1$ to $B3$ merged to form $RB1$, comprising the instructions from lines 10 to 21 of the code. Every new block of this reduced representation can have many different entry points but has only one exit

point. Branch tracing allows us to check that all exit points are correctly taken, ensuring at the same time that all preceding instructions, i.e., the sequential instructions that are part of that block, have been correctly executed in order.

Listing 5.2: Example ARMv7 Assembler code.

```
1   _func1:
2   push {fp, lr}
3   add fp, sp, #4
4   sub sp, sp, #8
5   mov r1, #10
6   mov r0, #2
7   add r4, r2, r3
8   cmp r4, #100
9   blt .L0
10  sub r4, r4, #3
11  str r3, [fp, #-16]
12  .L1:
13  sub r1, #1
14  add r4, r4, #3
15  mov r3, #0
16  str r2, [fp, #-4]
17  .L0:
18  str r4, [fp, #-8]
19  sub r2, #1
20  cmp r1, #0
21  bne .L1
22  pop {fp, pc}
```

CFC techniques typically only check the correct execution of transitions inside the target application's control-flow graph They usually generate all possible branches statically, including all possible return targets from a single function. They also typically do not encode information on the specific types of branches. In this approach, as shown in Section 5.2, blocks ending in functional calls and returns are considered unique types. While common direct branches have only one possible destination and conditional branches have two, the same procedure may be called from many different program points. Therefore, a subroutine's first and last blocks have several predecessors and successor



(a) Example of a classically defined CFG (b) Example of a reduced CFG

Figure 5.9: Example control-flow graph generated from the code in **??**.

vertices. Typically, this creates several edges into and from the first and last blocks of a function. This is simplified by identifying procedure calls statically but only acquiring the correct procedure return value dynamically.

Once the processor dynamically executes a procedure call, we know its corresponding return address, i.e., the following address from the call instruction. This address is the Next block starting after the function call instruction's address. Therefore, whenever the CPU calls a function, we store its corresponding return address, monitor its execution, and verify its return address when it ends with the stored value. A similar approach is used to monitor incoming interruption signals. The only difference is that asynchronous interruptions may occur at any point of the code and are not tied to specific static operations. An interruption's sub-graph corresponds to its ISR like a regular function, with a single start block and one or more return blocks. Monitoring the trace facilitates our ability to monitor asynchronous interruptions. All

modern trace protocols provide ways to identify a program's preemption by an interruption and the address where the interruption occurred. After it handles the interruption with its ISR, the processor must resume the execution state from this exact address. Therefore, this address directly corresponds to the interruption's return address and is stored in a similar structure used to store the function return addresses. The hardware monitor must store and analyze dynamic flow information if incoming instruction traces are to be properly checked.

5.4 Summary

This chapter described and analyzed the compilation stages fundamental to the generation of a binary application. Some of the most critical aspects of this process were outlined regarding the final resulting executable binary file and its structure. Mainly, function optimization and calling conventions are fundamental in identifying function bounds. The chapter showed how the iterative disassembly flow is performed from the machine code and how the executable control-flow graph can be retrieved. This generated CFG structure contains the minimum required information to decide if an incoming trace packet corresponds to a valid branch or not. In other words, the CFG stores the entire program control-flow, and the single traced execution corresponds to a unique path in this graph. Our hardware monitor uses the trace to identify the program's active vertex, i.e., its current basic block. The currently active block, in turn, uniquely identifies all allowed ensuing branch operations, depending on the CFG edges leaving from this vertex.

6 Hardware Control-Flow Monitoring Architecture

This chapter presents the proposed CFTC monitoring platform. CFTC expands earlier works on CFC by proposing an abstracted architectural model for trace-based hardware CFC techniques. It breaks down differences in control-flow complexity of embedded systems and how they influence the program's proper monitoring. The general idea behind this technique is exemplified by Fig. 6.1. The data stream coming out of the trace subsystem through the TPIU is rerouted from the typical JTAG path and sent instead to a dedicated monitor unit, in this case residing in the FPGA. The data corresponds to the payload of the JTAG unit before any encoding is done.

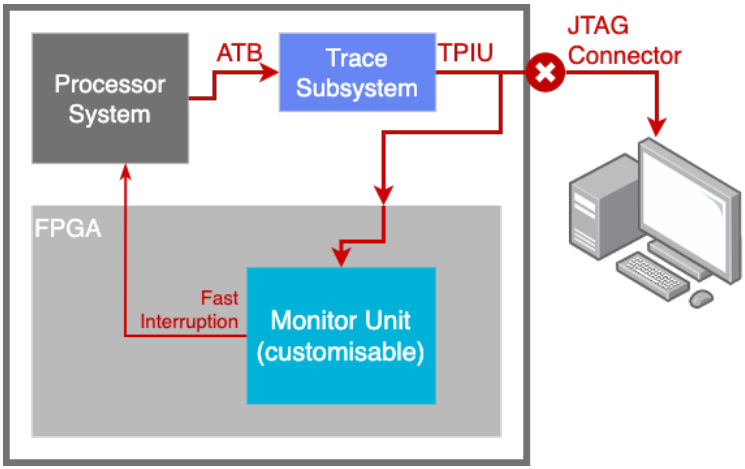


Figure 6.1: Diagram showing trace data rerouted away from the JTAG Connector and instead to the FPGA.

This chapter describes how this data is parsed and used by the monitor. The two main stages of this architecture are the trace translation and the verification stages, see Fig. 6.2. First, the translation stage includes the architectural dependent components, i.e., the *trace parser* and *trace decoders*, used to separate and decode trace packets. Then, the verification stage implements the portable architecture independent components, i.e., the checker and configuration memory, that correspond to the monitoring module's heart. Architecture dependence is associated with each system's particular trace protocol and its internal component mapping. At this stage, we cannot untie the specifics of the protocol used and component topology to identify the source of generated trace packets. Independent components can be used throughout different architectures. They only depend on the information given by the trace and use a generic interface. At the end of the Translation Stage, trace protocol encoding and topology are abstracted away, and we are left with one interface for each trace source that provides the control-flow changes encoded in the trace stream.

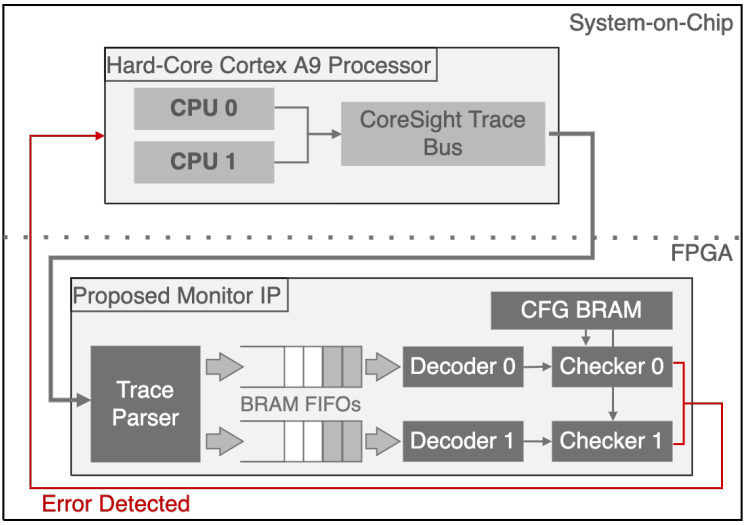


Figure 6.2: High-level diagram presenting the major components of the proposed CFC architecture.

This chapter describes in detail the implementation of the CFTC monitoring architecture, one of the central contributions of this work, the first portable and scalable trace-based hardware monitor architecture for modern multicore

processors. This chapter's proposed contributions and subsections are further schematized as follows. Sections 6.1.1 and 6.1.2 present the trace decoupling logic between the monitor detection units and the target ARM platform. Section 6.1.1 focuses on the trace parser of Fig. 6.2, while Section 6.1.2 explains the necessary decoders. Section 6.1.3 presents the checker architecture responsible for monitoring the state of each individual CPU core and which allows for easy customization for both fine- and coarse-granular monitoring. Section 6.1.4 describes the necessary control-flow graph configuration memory. It provides parameters for ease of implementation of different configuration representations and the size of monitored code. Finally, Section 6.2 clarifies the detectable errors and events possible with CFTC. This architecture is able to monitor not only intra-process control-flow events but also inter-process communications and synchronization events.

6.1 Monitor Architecture

This section describes the proposed scalable architecture for trace-based hardware monitoring. This thesis focuses on one target, the ARM CoreSight trace system, among the trace platforms presented until now. Embedded systems heavily use ARM processors, and many SoC devices further integrate ARM cores with FPGA fabrics, which allows for easy access to the integrated trace port of the SoC. The test-case platform used here is Xilinx's Zynq-7000 FPGA with an embedded ARM Cortex-A9 dual-core processor. This work introduces a real-time trace decoding hardware unit and interface that can be used by CFC and CFI monitors.

To fully track the control-flow, the interface must: inform whenever a new branch is executed and if it was taken or not in case of conditional instructions; notify when an asynchronous interruption occurs; and give instruction address information to identify branch targets, current PC values, and interruption address vectors; While this allows for full target address tracking, static analysis of the executing binary is required to identify the types of branches the application executes, e.g., procedure calls, returns, or conditional, as this information is not available within the trace.

Before the trace can be decoded, the processor core responsible for generating the packets must be identified. In Nexus-5001, this can be easily achieved

through the SRC field of its Program Trace Messages. Intel PT and ARM CoreSight apply a second layer of formatting over their trace packets to identify data from different sources over the same system trace bus. In Intel platforms, this is done through the Intel Trace Hub (TH), while in CoreSight, this is done in trace formatter units. These components combine trace data and id into frames before they are sent through serial buses or stored in memory.

These frames are composed of 16 bytes that interlace data and mixed-use bytes. These mixed-use bytes contain either data or a new source id value, and the bit 0 of each byte defines the value it contains. The protocol complements their content with a final auxiliary byte (AUX) at the end of the frame, as shown in Fig. 6.3. The direct approach to parse such a frame is to serialize all bytes into a buffer, verifying which data bytes belong to which ID. However, this is non-optimal in hardware, creating a bottleneck as a frame would need at least 16 cycles to be completely read.

This thesis focuses on one target, the ARM CoreSight trace system, among the trace platforms presented until now. CFC techniques have historically focused only on single CPU systems. Multiple concurrent processing units affect significantly affect the hardware. The monitor must deal with multiple incoming trace streams, with at least one for every processing unit.

Figure 6.2 presents an overview of our major implemented hardware components. The first two component types on top are the *Source Parser* and *Trace Decoder*. According to their source, the parser separates the different trace packets on the trace bus or interfaces. It then provides a dedicated interface for each monitored trace source. Each interface connects to one trace decoder component. This unit is bound to interpret all incoming trace packets and reconstruct the processor's internal state.

The following two main blocks are the *Checker*, responsible for verifying the correctness of received CPU states, and the *Configuration Memory*, responsible for holding the copy of the CFG and verification information used by the monitor. These components are the core of the monitoring system and implement the structures required to perform the different checking operations. They are described in detail in Section 6.1.3. A detailed implementation of the first single task monitor for an ARM Cortex-A9 processor architecture is presented in the later sections. This first monitor architecture is an example of

the separation between components that depend on the architecture and those reusable across different hardware systems.

6.1.1 Source Parser

The trace monitoring pipeline's first two components are responsible for collecting, separating, and decoding trace data from each monitored processing unit. The Source Parser component must be generated for the correct number of trace sources and configured with their respective IDs to parse these frames correctly. This component has three main functions. It parses and extracts the data from the first protocol layer of ARM's trace architecture. It aggregates multiple payload sizes into a standard 128bit bus that the decoder can more easily handle. Finally, it also works as a switch, driving trace payloads from different sources to the correct decoder and checkers units for each CPU.

This unit rebuilds the trace stream corresponding to each source present in the system. It receives and routes the packets to the correct output interface. After the decoder handles the data, it is available in a generic interface independent of the underlying trace architecture. This abstract interface builds on the analysis performed and results presented from Chapter 4. Each trace packet originates from a single core or processing unit. These packets cross through a system-wide trace bus to the dedicated trace analyzer or capture device. To differentiate each packet's source when inside the bus, plus other information that might be available regarding system-wide events, each data packet is associated with a unique identification (ID) value.

This work's implementation test-case is ARM's CoreSight trace architecture with the ATB interconnect [125]. ATB belongs to and is a part of the Advanced Microcontroller Bus Architecture (AMBA), an open-standard, on-chip interconnect specification developed and maintained by ARM. To send all packets and their source information to a source external to this interconnect, usually serial interfaces are used. The packets are then combined into larger frames, composed of trace data and identification values. In the ARM trace system, each frame has 16-bytes that interlace data and mixed-use data or source ID bytes. The first bit of each mixed-use byte tells if it is being used for data or a new source value. The content of mixed-use bytes is complemented by a final auxiliary byte (AUX) at the end of the frame (Fig. 6.3). This extra byte holds

one additional bit of information for each mixed-use byte. It holds bit 0 for mixed-use data bytes and the update delay for mixed-use ID bytes, i.e., if the ID is immediately updated or delayed for one cycle. All data values inside a frame and subsequent frames always relate to the source ID value of the last active value informed. Using this or similar frame formats to encode source IDs increases the trace port bandwidth. The CoreSight frame protocol shown here adds an overhead of 6% to the bandwidth requirement of a trace port.

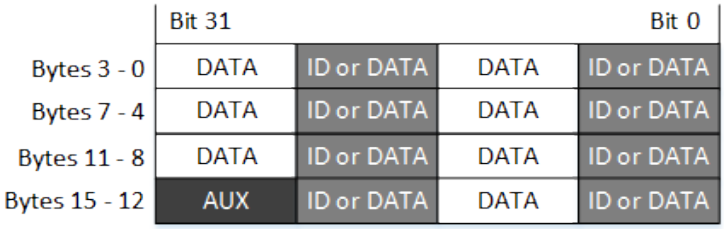


Figure 6.3: CoreSight frame format definition [117].

The CoreSight frame has a regular size and format, which favors an optimized hardware implementation. However, its payload contents' irregularity makes it hard to implement a parser that can offer high-output bandwidth. Figure 6.4 shows the straight finite state-machine (FSM) implementation to parse such a frame and its accompanying hardware diagram. This approach serializes all payload bytes into two buffers, one for data and one for the accompanying source ID. While easy to implement, this straightforward implementation requires 15 cycles to read a frame wholly.

This approach's reading latency creates a significant input bottleneck and significantly increases the system's detection time. Therefore the Parser needed to be optimized to solve this problem. The possible payload configurations and how to decrease the latency to decode a frame were analyzed. While not detailed in the protocol's documentation, the frame generation unit prioritizes the formation of frames with a single ID. Two typical cases were clear: a new frame with no new ID packets, meaning the payload is only composed of data packets from the previously identified source ID; and a packet with a single ID change on the first payload byte. Thus the Parser could immediately parse the frame and move the payload to the correct decoding unit within a single cycle.

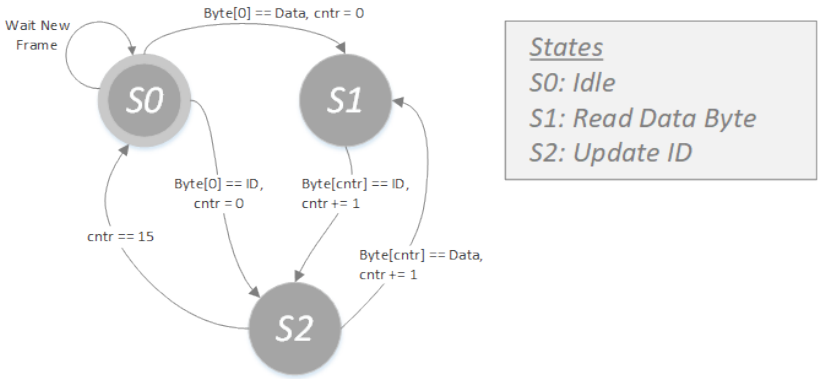


Figure 6.4: Direct state-machine description to parse a CoreSight Frame.

After optimizing the Parser for single ID payloads, the advantages in the similarity between frame rows were used to identify other common cases. Table 6.1 presents the possible source ID payloads for a single row of a CoreSight Frame. Each row can have a maximum of 2 new source IDs, and each new ID present can be immediately applicable or delayed by one data byte. A row is composed of four bytes, from which bytes 0 and 2 can either encode a new ID or a data byte, depending on the Auxiliary frame byte (frame byte 15). Whenever byte 0 encodes a new source ID, it is defined as the first ID of the row. Similarly, byte 2 is the second ID of the row if it encodes a new ID, independent of the presence of a first ID or not.

Given the possible resulting row payloads of Table 6.1, we see that in the best-case scenario, there is one cycle to output the payload contained in the row and two cycles in the worst case, one cycle for each ID and corresponding data. Two cases are not present in the table, the row payload configuration where two new IDs are present, and the first ID is delayed. This possible configuration never happens in practice, as a second immediate ID would overwrite the first delayed ID. Similarly, two new delayed IDs in the same line can be optimized away by one immediate ID in the current row and one in the next row. Further, these two configurations were never observed during testing.

The optimized frame parser requires, on average, between 1 and 4 cycles to parse such a frame completely. Figure 6.5 shows the internal architecture for

Table 6.1: Possible row payload's ID configuration for a CoreSight Frame.

New IDs	*Applicability*	*Resulting Payload*
No new ID	-	4 bytes with last ID
First ID	Immediate	3 bytes with First ID
First ID	Delayed	1 byte with last ID, 2 bytes with First ID
Second ID	Immediate	2 bytes with last ID, 1 byte with Second ID
Second ID	Delayed	3 bytes with last ID and update ID to Second
First and Second IDs	1^{st} Immediate, 2^{nd} Immediate	1 byte with First ID, 1 byte with Second ID
First and Second IDs	1^{st} Immediate, 2^{nd} Delayed	2 bytes with First ID and update ID to Second

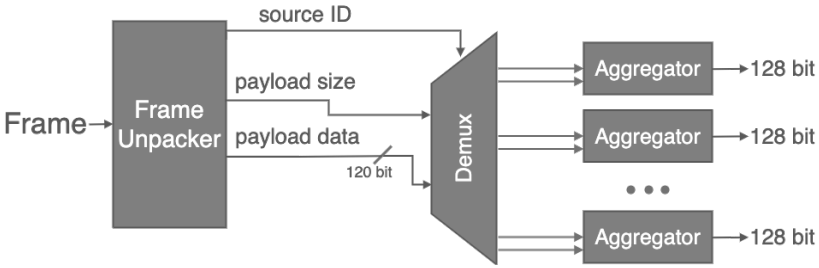


Figure 6.5: The Source Parser internal components with frame Unpacker and data aggregator units.

optimized frame parser. The first subunit is the frame unpacking stage of the optimized Parser. After it receives a new frame, it outputs a new source ID every cycle, the payload data for that ID in the frame, and the payload size in bytes. The payload sizes can range from 15 for a frame with no new IDs or 1 for a frame line with 2 ID changes. First, it identifies the total number of new source IDs present in the packet. If no new IDs are present or all data bytes share the same ID, all 15 to 14 data bytes can be output at once, accelerating

the process. Otherwise, the frames are analyzed line by line using the cases defined in Table 6.1. This approach takes 4 to 6 cycles to retrieve all data, depending on where the new ID source is. After analyzing the typical output of execution traces, it was noticed that the trace formatter tends to group data from the same sources into a single frame. Therefore most frames usually have a single ID change, and frame decoding takes, on average, one to two cycles.

After parsing the data payload with its corresponding source ID, this information must move to the Trace Decoder. However, to facilitate the decoding pipeline's implementation, the irregular payload outputs must be aggregated inside the parser, and the data is combined to form regular-sized chunks. After parsing, the output can have the following sizes: 15 bytes, 14 bytes, 4 bytes, 3 bytes, 2 bytes, or 1 byte. Each of these payload sizes can be given in a cycle as input to a single trace packet stream that must be written to the decoder's input buffer in order. Therefore, a *Data Aggregator* subunit is used after the Unpacker unit. This unit provides a standard interface of 16 bytes to the decoder, as shown in Fig. 6.6.

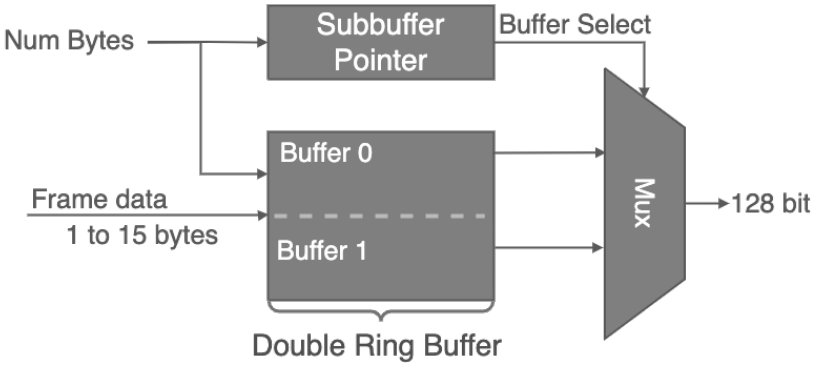


Figure 6.6: Architecture for a single data aggregator subunit.

The frame parser provides the number of acquired data bytes at each parsing cycle. The aggregator implements a ping-pong-buffer architecture where the input payload is shifted and written to a double ring buffer with two output lines for either half of the complete buffer. When one half of the ring buffer is being written to, the other can be read by the decoder. Each buffer is 16 bytes wide, and whenever one is complete, the aggregator transmits the data to the decoder.

This highly parallel architecture enables us to quickly separate and send the packet data contained in the frames. However, as shown in Chapter 7, the aggregator requires many resources to accommodate its parallel-shift scheme. The Source Parser is not needed if only a single-core is monitored, but for multicore systems, such a fast component is crucial. However, for multicore systems, such a fast component is crucial.

6.1.2 Trace Decoder

The trace decoder component is responsible for unpacking all trace packets for a given trace source and providing the CPU state interface required by the monitoring component. This interface should contain the necessary information to support our defined control flow monitoring operations. The output interface implemented for our trace decoders is shown in Table 6.2. Through this interface, the monitor can know all events that affect the control flow of the processor. Some extra signals help distinguish between system interruptions related to detected faults and programmed interruptions.

System aborts are sent when the program tries to access an invalid memory region or execute a wrong operation. Update events are sent right before an interruption. They are used to update the trace decoder and inform from which section the program was last executing. With this, the monitor knows what the exact point inside the block the ISR must return to is. Cycle count information is helpful to make sure the system conforms to real-time constraints. Each retrieved cycle count can be accumulated from sequential elements to obtain time measures of certain code regions or function sequences. The deadlines of such functions or events can be pre-programmed to detect any time violations of the program. Not all information provided by this interface needs to be used by the Checker. Which information is used depends on the granularity or complexity of control flow monitoring performed.

After identifying the origin of trace data packets, one decoding unit for each system core is generated. For this work's target system, a real-time decoding unit conforming to the PFTv1.1 standard [122] is created. The Decoder processes the trace packets and makes the CPU state available through a generic interface. Once the state information is retrieved, it can be used by control flow monitor architectures.

Table 6.2: Signal output interface of a trace decoder unit.

Signal	*Definition*
`new_trace`	Informs the monitor that a new trace information is available to be read on this cycle.
`address[31:0]`	Either informs the target address of a taken branch or the last executed address before an interruption.
`branch_event`	The trace packet corresponds to a normal branch.
`cycle_count[31:0]`	(Optional) Unsigned integer informing the number of processor cycles executed since the last event.
`cond_taken`	If `branch_event` is set, it informs that the branch corresponds to a conditional branch and that it was taken.
`cond_not_taken`	If `branch_event` is set, it informs that the branch corresponds to a conditional branch and that it was not taken.
`update_event`	The trace packets is used to update the last address in the PC. Usually sent before interruptions.
`irq_event`	The trace packet corresponds to a received interruption request or software exception.
`abort_event`	The trace packet informed of a taken hardware interruption or abort. They are sent when the system detects an internal error.

The two most essential packets for online monitoring are Branch Packets and Atom Packets, corresponding to TIP and TNT in the Intel PT architecture. Their formats are given in Fig. 6.7. The total number of bytes in each packet is only known during execution. The protocol uses C bits in every byte to only output as much data as it needs to inform which states have changed in the CPU. If the C bit is set, the next byte in the stream belongs to the same packet. Atom packets use an F bit that is set in case a conditional branch fails its check. These packets can optionally give cycle count information: the number of CPU clock cycles since the last branch. These counts help monitor techniques that incorporate timing analysis, but as shown in Section 7.4.5, increasing the packet size can add considerable increases to trace bandwidth.

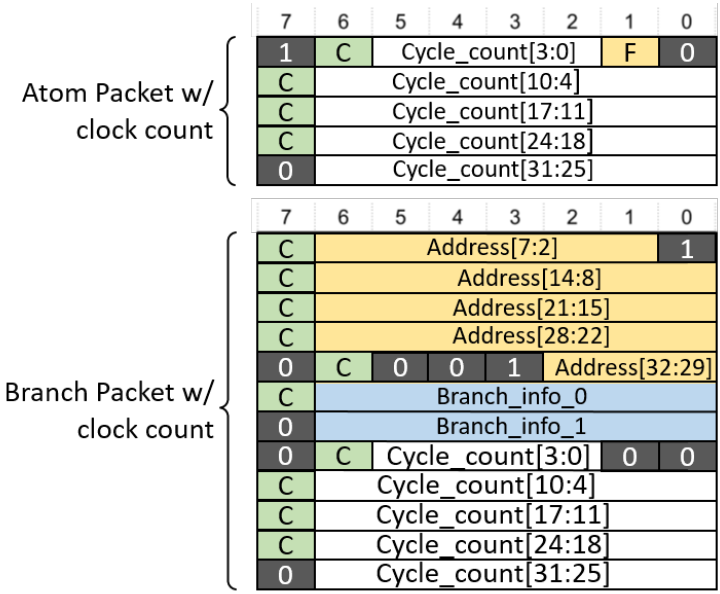


Figure 6.7: Data formats for Atom and Branch CoreSight PFTv1.1 traces.

This type of packet format, also similarly implemented in Intel PT and Nexus-5001, makes it challenging to implement an optimized parallel decoding scheme, as it would require complex distributed shift registers and comparison networks. However, by analyzing the actual average output of an execution trace, it was noticed that most Branch Packets output only 1 to 2 bytes of address data, with most branches having relative target addresses to near areas of code. Similarly, most cycle count data has only a single byte, as the average size of basic execution blocks in a program is relatively small. These factors indicate that serializing the packet stream would not add large overheads to the decoding time and could help reduce the hardware accelerator's resource usage.

The hardware decoder architecture is presented in Fig. 6.8. Every byte of the trace packet stream is read and scheduled to an appropriate decoder FSM depending on the current packet header. A dedicated state-machine pipeline

is implemented for every standard packet type to decode incoming traces optimally. A set of global state flags are used with a global FSM to synchronize the start of the packet stream and every decoder state-machines simultaneously.

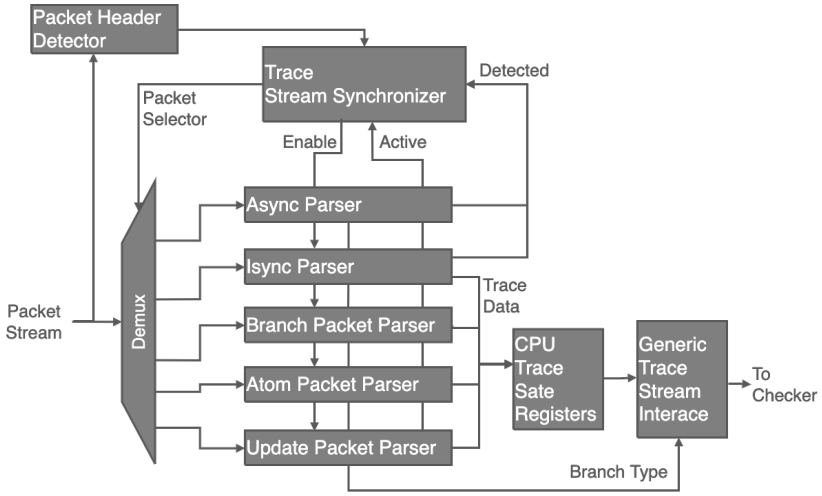


Figure 6.8: Internal architecture of the trace decoder hardware unit.

The main synchronization state-machine is shown in Fig. 6.9. It follows the ARM CoreSight specification PFTv1.1. It enables all packet state-machines once an A-Sync packet followed by an I-Sync packet is correctly decoded. The A-Sync packet informs the data stream's boundary and where the following packet header lies, Fig. 6.10. The I-Sync packet provides all the information regarding the current CPU state, Fig. 6.11. This information includes the current PC address, security state, hypervisor and virtualization modes, and active ISA.

The state-machines responsible for branch and atom packet decoding are shown in Figs. 6.12 and 6.13. Both FSMs can update the decoder's output interface according to the last incoming packet's new state information. Due to its compressed encoding, a single atom packet may represent multiple sequential not-taken branches (up to 5 in a row). If a single atom packet contains multiple control-flow state changes, one cycle is required to output every state. In

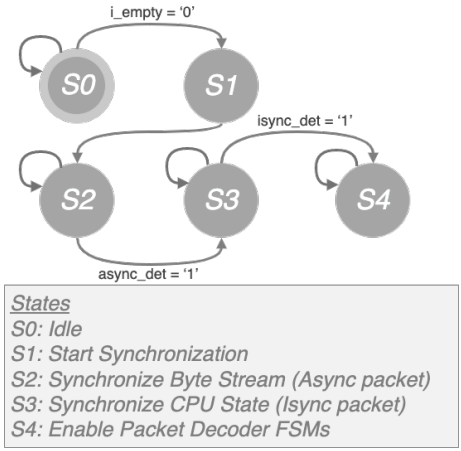


Figure 6.9: Decoder synchronization state machine for the Cortex-A9 use case.

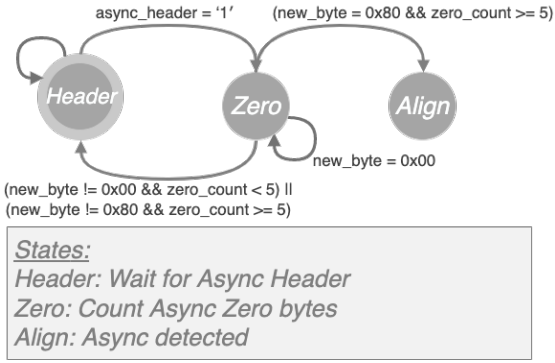
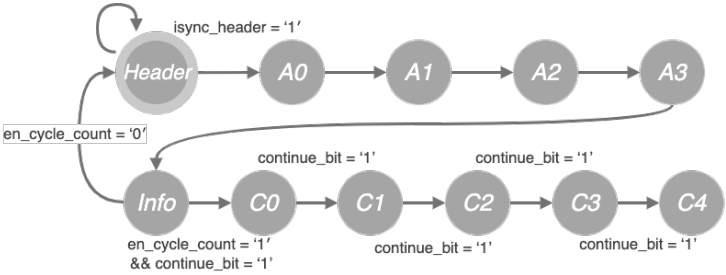


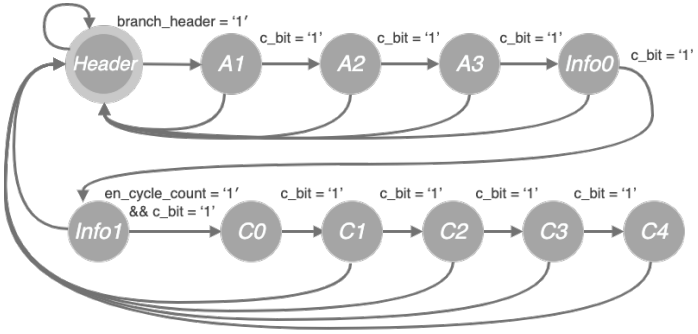
Figure 6.10: Packet stream synchronization with A-Sync packets.

such a scenario, the atom decoder generates back-pressure to the input stream controller, which stalls other decoder units unit it has finished translating its current packet. If the processing core is asynchronously interrupted, the trace system sends an update packet to inform the current PC address before a branch



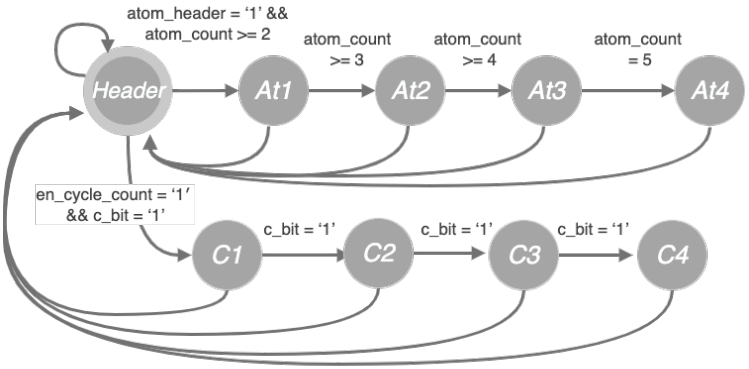
State:
Header: Wait for Isync Header.
A0-A3: Read Current PC Address bytes.
Info: Read reason for packet synchronization, e.g. trace start.
C0-C4: Read cycle count bytes, if configured.

Figure 6.11: CPU state synchronization with I-Sync packets.



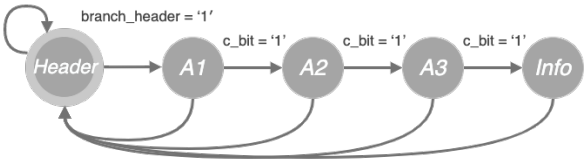
State:
Header: Wait for Branch Header. Read Address byte 0.
A0-A3: Read Target PC Address bytes (only changes from last address).
Info0-1: Read CPU mode changes and branch reason, e.g. branch or IRQ.
C0-C4: Read cycle count bytes, if configured.

Figure 6.12: PFT Branch Packet decoder diagram.



State:
Header: Wait for Atom Header. Detect single branch atoms.
At1-At4: Decode additional encoded branches.
C1-C4: Read cycle count bytes, if configured.

Figure 6.13: PFT Atom Packet decoder diagram.



State:
Header: Wait for Branch Header. Read Address byte 0.
A0-A3: Read Target PC Address bytes (only changes from last address).
Info: Read if instruction set architecture (ISA) changed since last packet.

Figure 6.14: PFT Update Packet decoder diagram.

packet notifies the type of interruption and target interruption vector address. Figure 6.14 shows the corresponding FSM used to decode Update packets.

The decoder component as a whole holds the last state of the CPU. The decoder has to hold these previous values to complement the partial address values given by each packet. It also uses this information to correctly decompress the instruction address according to the processor's operation mode and current ISA. When it identifies a branch packet, its address bytes only inform the least-significant bit changes since the last jump. The ISA state informs the expected alignment of incoming addresses. The Checker can use context ID, Hypervisor, and security state information to identify illegal operations or changes in the control, depending on the implemented basic software model.

The decoder can require an extremely complex architecture due to the trace protocol's nature. Its implementation can change considerably depending on the target processor architecture. The simplicity of the Leon 4 processor's trace packet, for example, only requires a single cycle to parse a packet that already convenes all the information needed by the decoder output interface (Section 4.4).

6.1.3 Control-Flow Checker Core

The architecture-independent components correspond to those that can be used across different processors and trace hardware architectures. They only require the implementation of the generic trace interface provided by the Decoder. The monitor's verification stage is separated into two units, one tasked with performing the correctness checking operations and another responsible for storing and providing access to encoded control-flow information. They correspond to the control-flow Checker and the configuration memory units.

The Checker is the central unit of the error detection stage. It receives the current CPU state changes through the trace and compares these changes to the latter possible states with the configuration memory. The Checker's general execution flow and checking operations are shown in the diagram in Fig. 6.15. A target address check is performed depending on the type of branch executed. The component's internal structures are updated accordingly if the branch is correct. For example, if the program makes a function call, its return address is pushed to a call stack implemented in hardware. When it executes a return instruction, it is dynamically compared to the top of this stack to check for errors. An error signal is sent to the processor if any CFEs are detected.

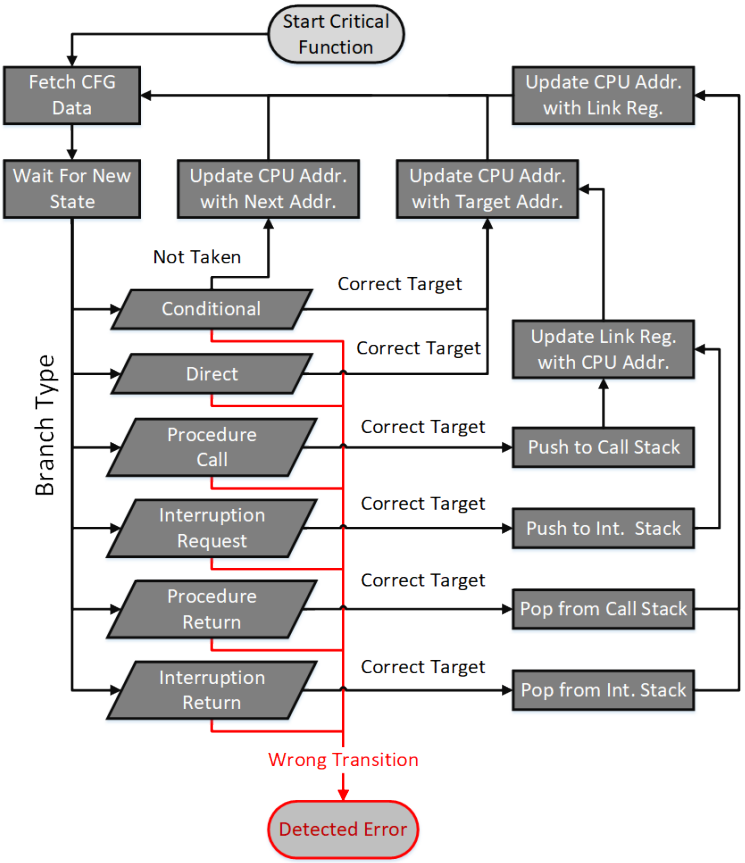


Figure 6.15: Operational diagram of control-flow checker after a new CPU state.

A pipelined FSM architecture and hardware structures are required to implement this checking mechanism. They hold replicated software data for sub-routines and interruption stacks, as shown in Fig. 6.16. In the configuration memory, the CFG corresponding to the different tasks and interruption routines of the current binary is stored. Given an address query, this dedicated

memory must inform the program checker of all related information of the next sequential branch from the input address.

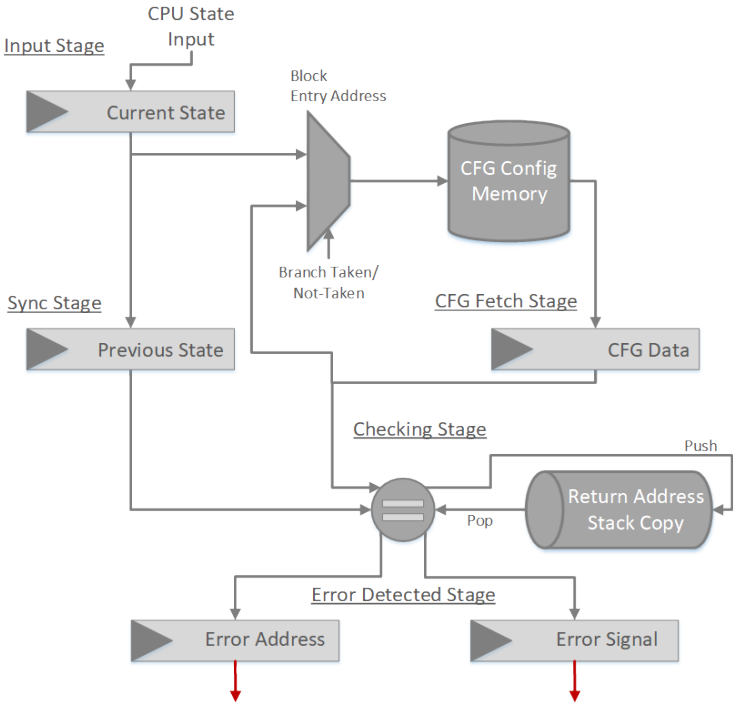


Figure 6.16: Hardware structure of main control-flow checker component.

This Checker holds hardware structures that replicate software systems and information about the processor's current execution state. The first of such structures are the last state registers, which hold the last known PC state of the processor, current nesting level, i.e., the number of nested calls that have not returned yet, as well as secure execution state information, e.g., if the processor is running in a privileged execution mode or similar. This PC state is updated after every branch with the value of its destination. When the CPU handles interruptions, it must first update the PC with the interrupted instruction's address. This address is acquired with Update packets and saved in the stack

to be checked at the interruption's return. Soon after this Update, a Branch packet informs the interruption's vector address that will be executed.

The main structures handled by the Checker are the replicated stacks for procedure calls and interruption requests. Each stack saves the LR values needed to know return branches' targets dynamically. These stacks must be updated at every branch that corresponds to a procedure call or procedure return instruction. The type of branch is retrieved from the configuration memory using the current state's address. Whenever the processor is executing a block related to a procedure call or return, the program checker knows that the next incoming branch must be checked against a function address or the topmost address on the stack.

Using the current PC state, the Checker retrieves the configuration memory data on the current basic block. Each block holds the information of its corresponding branch, executed as the last instruction in the block. This information is used to know what type of verification and update operation must be performed. Our single task checker must perform the following operations depending on the current block:

1. **Direct Branch Check:** The trace packet must correspond to a direct branch and its address must match the current block target address. The PC copy is then updated with the new address.

2. **Conditional Branch Check:** The next trace packet must correspond to a conditional branch, if it is marked as taken, the address and block target address must match. The PC copy is updated with the target address if the branch is marked as taken or with the sequential next block if marked as not taken.

3. **Procedure Call Check:** The verification is done the same as for a Direct Branch Check. The update operation however must update the PC copy and push the next sequential block address to the stack.

4. **Procedure Return Check:** A return from a subroutine must be implemented as a direct branch, it is only valid if the target address of the packet corresponds to the value on top of the stack. This value is removed from the stack and used to update the internal PC.

The Checker uses different pipeline stages to manage each of its essential operations. To perform all the described operations with minimal stalling or hindering of the input trace interface. The pipeline architecture is presented in Fig. 6.17. It comprises five execution stages and one error stage, only reached if a control-flow error is identified. The five stages correspond to an input stage, a stage to request data to the CFG memory, one synchronization stage, one memory read stage, and a final checking stage. If the checking stage finds a discrepancy, the error detection stage is triggered where an interruption is sent with the precise error resulting information.

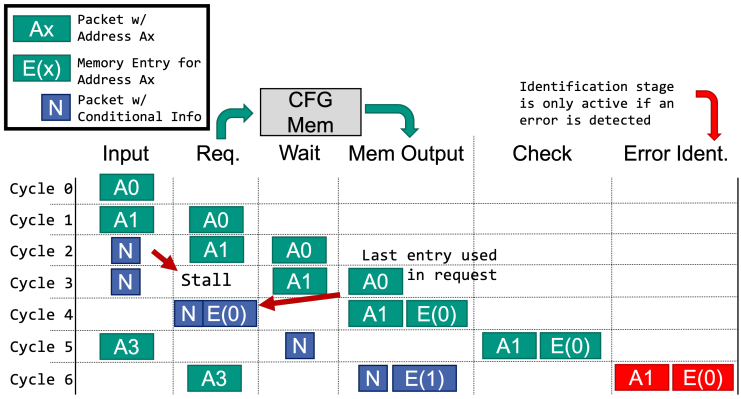


Figure 6.17: Example of the information flow through each pipeline stage.

The input stage is responsible for accessing and communicating to the decoder interface to get incoming trace information. It determines if the new CPU address is off-bounds from the limits of the current function, quickly informing if a jump to a forbidden region has occurred. In the example in Fig. 6.17 we see the arrival of different packets between cycles 0 through 5. Packets that give address information are shown as A0-3 in the pipeline. That is the current address pointer by the PC. Atom packets provide a packet with a single N in our implementation. That is a signal that a conditional branch was not taken.

The configuration memory read request stage uses the input information to fetch the current block information combined with the current internal CPU state. In the request stage, the address in the trace packet combined with

the Checker's copy of the CPU state is used to get the data about the next branch following from this address. Meanwhile, new packets with branch trace information arrive. In the example, address A0 will be used to find the information needed to check if address A1 is correct. The CFG entry for A0, shown as E(A0), contains the data for the next branch that will be executed following sequentially from A0. All entries E(x) are determined through static analysis and stored in the configuration memory.

With this setup, the first branch is always assumed to be correct. The Checker can keep treating incoming packets as correct so long as the checking stage does not trigger an error. The final comparison check stage will trigger an error at the very first mismatch between a CFG entry trace and its corresponding packet data. The data used to fetch the entry is one cycle before the currently checked packet. Therefore the retrieved CFG information corresponds to a correct state. If an error is detected, it will have been the first error in the pipeline sequence, and all following packets are discarded. Finally, the check stage compares this read CFG information with the trace data and the top of its internal stacks.

N packets complicate the process since they only implicitly inform where the current PC is. To find out the actual current address, we need the instruction address value of the last branch. The N packet tells us that the PC now points to the following address from it, i.e., the last branch's instruction address plus four. The pipeline must stall for one cycle so that the entry fetched from the last packet can be used for the current request stage.

This architecture detects all deviations from a binary application's control-flow operation targets. These deviations are, for example, the creation or deletion of branch instructions, stack overflows or underflows, the modification of any target addresses, and the execution of unallowed interruptions. The correct execution of loops and the correctness of conditional branches could also be verified with a more complex binary analysis. For example, monitoring how many times a conditional loop is taken before its condition is not taken. This work's fined-grained yet straightforward analysis does not require complex signature generations and calculations to identify incoming errors. The complexity of what types of errors are identifiable lies with the trace architecture and the available information given by the CFG configuration memory.

6.1.4 Configuration Memory

The configuration memory is responsible for providing the results of block queries made by the program checker. The Checker uses this to evaluate new incoming trace packets. The process used to acquire the CFG information that fills this memory is described in Chapter 5. The memory must give the corresponding block information for every instruction address, i.e., the block's branch information. Trace data inherently works with instruction address values regarding currently executed instructions or target branch addresses. Another constraint imposed by branch tracing is the need to use a reduced CFG model, contracting all edges of the graph that do not correspond to a transition due to a branch instruction. This contraction results in blocks having multiple input points and multiple addresses as valid incoming branches.

The CFG memory also has to be as fast as possible to minimize any detection delays. Therefore, incoming addresses must be mapped with their respective block information, optimizing the memory for total size and access times. The solution proposed in this work is based on an associative memory with variable block sizes. Therefore, this configuration memory takes the form of a type of content-addressable memory (CAM). The general architecture of our memory is shown in Fig. 6.18.

Many optimized architectures exist for this type of associative memory. This implementation uses a two-stage access method. First, the current CPU instruction address is used to retrieve its corresponding block's key. This key is 12 bits wide and is then used to retrieve the basic block entry from the CFG memory. This intermediary key table has one entry for every instruction of the monitored function, similar to an inverted list. First, the instruction address is normalized by subtracting the monitored function's base address. Bits 15 to 2 of the instruction address are used as a tag to access the Key values of the instruction's block. We discard the address' least significant two bits because instructions are 32-bit aligned in memory. A tag with 14 bits allows us to uniquely identify instructions in a contiguous region of 64kB ($14b \rightarrow 2^{14} = 64\text{kB}$).

The key table memory contains one entry for every monitored branch instruction. The block key size k retrieved by this table can be set according to the number of blocks in the program ($k = log_2(N_B)$). Using a 12-bit key allows us up to 4096 blocks, more than any of our tested benchmarks. The total

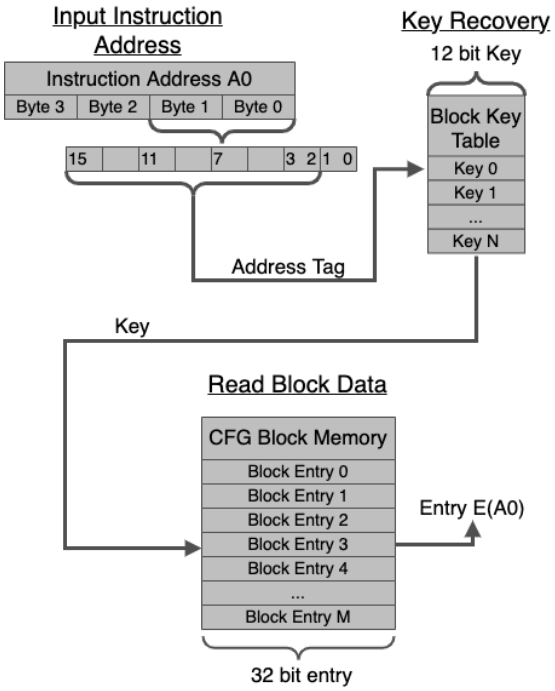


Figure 6.18: Configuration memory architecture with key access and entry retrieval stages.

table size is then 32bit Afterward, the memory controller immediately uses the block Key to retrieve the final CFG entry that holds the desired basic block information. Figure 6.19 shows the layout of a single entry in this memory.

Each block entry is a 32-bit value that holds the next sequential block's target address, the next sequential address, and the type of the corresponding branch. Each stored address uses the reduced tag format, so the Checker can directly request the key and entry for current PC addresses. The remaining four most significant bits correspond to the block's branch type information. It is used to perform the correct branch monitoring operations in the Checker. They correspond to the type of control-flow operation associated with the block. Bit 0 identified if the branch is conditional or not, signaling that the next address

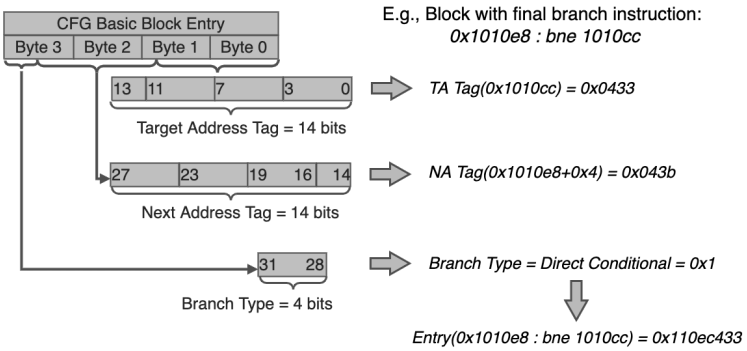


Figure 6.19: Entry format of the CFG configuration memory.

field is a possible target of the branching operation. The remaining 3 bits inform the remaining complementary eight possible block types:

- `4'b000x`: Direct Branch
- `4'b001x`: Procedure Call
- `4'b010x`: Procedure Return
- `4'b011x`: Interruption
- `4'b100x`: Interruption Return
- `4'b101x`: Unused
- `4'b110x`: Unused
- `4'b111x`: Unused

Besides blocks having multiple input points, their sizes are also not uniform. Although basic blocks have an average of 10 instructions for a program, their actual sizes present a large variability, with many blocks having only one instruction and some as much as 385. The irregularity of basic block sizes and base addresses makes it extremely hard to implement a suitable comparison mechanism or hash function that retrieves a block key from an arbitrary instruction address. Our reduced block configuration allows a block to have multiple

input points. Therefore any instruction address that belongs to a specific block can be a possible entry point. Dealing with diverse instruction addresses is even more significant if the program implements asynchronous interruptions. When the interruption ends, the Checker must restore the state previous to the interruption. The interrupted address can correspond to any arbitrary point inside the program, and the Checker needs to be able to retrieve the basic block information related to that point.

Equation (6.1) gives the total memory size S_{mem} of our implementation, where N_I corresponds to the total number of contiguous monitored instructions, N_B the total number of basic blocks in the monitored function, t is the tag length, and k the key length.

$$\begin{aligned} S_{mem} &= 2^{log_2(N_I)-2} * log_2(N_B) + 2^{log_2(N_B)} * (4+2*(log_2(N_I)-2)) \\ &= 2^t * k + 2^k * (4+2*(t)) \\ &= 2^{14} * 12 + 2^{12} * 32 \\ &= 320\,\text{kbit} = 40\,\text{kB} \end{aligned} \tag{6.1}$$

This configuration memory is implemented in the FPGA test-case platform as a read-only memory (ROM) in through dedicated Block RAM (BRAM) components. The final memory size was 40 kB, or about 10% of the available BRAMs resources of a common Zynq-7000 device, and 1% of BRAM resources in a Zynq UltraScale+ device. Accessing this memory does not degrade the application's performance, as it does not share the main memory bus with the device data and program memory. The configuration data is generated from the binary executable corresponding to the monitored program. This flexible scheme allows the system to configure the FPGA before executing a critical function with its corresponding configuration data. Although this work uses a 12-bit key in its implementations, this value can be changed to accommodate smaller or bigger program functions at the cost of larger configuration memory requirements.

6.2 Detectable Control-Flow Errors

As the primary information-gathering interface used in this architecture, instruction branch tracing defines what errors can be detected. The pipeline

region used to trigger the creation of trace packets also influences the errors that might be detected. The architecture presented here, called CFTC, can detect any errors during a branch instruction execution if trace packets are created after the pipeline commits the operations. If packets are transmitted during the fetch stage, it can only verify that the control-flow was correctly executed up to and including the last instruction before the trace, but not the trace instruction itself. Trace packets from ARM's PTM and ETMv4 modules are generated during the fetch stage but transmitted differently. The PTM unit only transmits packets after the corresponding instruction has been committed. ETMv4 transmits them immediately and uses an additional commit or discard packet to tell if possibly speculative instructions were committed or not.

Chapter 2 explains the major types of CFEs that can occur during program execution. With trace data, any *Incorrect Targets* and the *Branch Creation* of control instruction in non-control code sections can be easily identified. Their detection delays are constant and determined by the total trace bus latency and monitor unit latencies, which will be around the same order of magnitude as the detection unit's clock period. *Branch Deletion* CFEs can be detected, but with a longer latency. They are detected the moment the next branch, which sequentially follows the deleted one, is executed. A hardware monitoring unit can detect and sign the wrong branch order, with the added time latency of the next basic block's execution time. We can add to these cases all other CFEs that do not directly delete or create a branch but modify a conditional branch to an unconditional or change any other part of the instruction opcode, causing it to behave incorrectly.

Direct PC Errors can be similarly detected. Changing the PC's value may not cause the generation of a trace packet. Therefore, it cannot be directly observed but inferred by the wrong sequence of instructions that the processor will fetch. This specific type of error can cause three types of wrong fetches. It may cause the processor to fetch from a memory region outside of the one allowed by the program, in which case it will be directly detected by memory management and protection mechanisms. Alternatively, it will cause a branch to a region inside the program's code. Once the processor starts to fetch from the program region after the CFE, the first branch fetched and executed from a code section not corresponding to the following allowed edges in the CFG will trigger an error. Therefore, the only possible PC errors not detected by our technique are those that cause the PC to point to an address inside the same

basic block to which it was already pointing. If such a fault occurs, the next fetched branch will still be the expected one. In such a scenario, the processor will have executed more or fewer instructions of the same basic block, possibly causing data errors.

Finally, to detect *Incorrect Condition Checks*, the method used must either implement NMR for branches and the comparison instructions that precede them in a software-based approach. Hardware-based methods must use a pre-computed reference execution of the correct comparison results or have the means for a parallel runtime replication of the same operations using the current input data. However, in both cases, the protection goes only as far as the replicated instruction chain, with all condition codes and internal state flags of the processor remaining unprotected. These schemes still propagate any errors occurring to the compared data if it happens before the duplicated comparisons. Therefore this data remains unprotected. The incorrect check will be done both by the processor and the monitor. Hardware monitors must know which kind of structure these conditional control-flow instructions belong to. Loops that execute a fixed number of iterations are the easiest to verify, and if-else statements with generic data checks are the hardest. Due to these factors, detecting condition check errors is not currently in the scope of this work.

6.3 Summary

This chapter presented the detailed architecture of the principal components of the hardware monitor. It first presented the translation stage components composed of the trace source parser and trace packet decoder. The objectives of most trace protocols lie in reducing bandwidth and memory storage requirements. This focus on data compression means that decoding such data can be very demanding. The most important aspect is that each packet is processed in a single cycle or as fast as possible, e.g., all the packet's data is consumed and disseminated over the Decoder's pipeline in a single cycle. It showed an architecture that performs as desired for trace frames, separating the data into different source streams. The chapter also showed how, for ARM CoreSight traces, trace packets are so irregular that a single cycle process would cost many hardware resources. However, the small size of packets allows for an

optimal byte-processing that can be more easily implemented in an FPGA and depend only on the average packet size. In Chapter 7 it is shown that packets are usually between 1 and 2 bytes, meaning that they can still be quickly processed through an FPGA accelerator.

The portable architecture for both the central Checker and Configuration Memory components was also presented here. These components can function independently of the branch architecture used. They are optimized for FPGA resources and can detect a wide range of different CFEs. CFTC's CFG configuration memory format can be changed according to the number of monitored instructions and basic blocks. A modular architecture approach such as this is ideal for fault tolerance and error detection methods. This technique opens the space to focus on different optimization aspects such as trace decoding speeds, configuration memory size, and detectable errors. These methods also allow for extensions that can enable, for example, the detection of inter-process synchronization errors by determining which executed basic blocks are responsible for synchronization operations and implementing suitable hardware structures that can replicate current resource allocation chains.

7 Evaluation and Fault Injection Results

This chapter evaluates the proposed control-flow monitor for an ARM Cortex-A9 architecture case study implemented in a Zynq-7000 FPGA. First, the platform is analyzed in terms of its hardware implementation results, examining its resource sub-component usage and comparing it to the protected CPU estimated area in gate counts. One of the main contributions of this work is also detailed here: a new error detection latency model for this and future trace-based monitoring techniques to evaluate these systems' usability in real-time systems. Most safety-critical domains must also adhere to soft and hard real-time constraints for executing software and system functionalities. Here it is demonstrated how to use static binary analysis to estimate monitor utilization pressure using Queue Theory models. This work also proposes a new way to estimate the AVF of a black-box program-trace subsystem such as the CoreSight, presenting collected measures for the case of the Cortex-A9 CoreSight.

Finally, this model and the proposed hardware architecture are validated by collecting different benchmark program traces in single- and dual-core setups under fault injections. The chosen benchmarks are initially evaluated running in a single-core configuration through static fault injections based on a binary modification scheme presented in Section 7.4.3. Later, the same benchmarks are verified under an asynchronous dual-core configuration through runtime fault injections based on an interruption scheme presented in Section 7.4.4. The first injection analyzes this monitor's ability to identify branch-opcode-related faults, including almost 74% of masked faults. Runtime injections allow for observing different latencies between trace packets according to how far in its execution each program was. It shows the possible effects visible to an isolated monitor from the trace interface and other opportunities in using hardware tracing for fault tolerance and program monitoring designs with these evaluations. Through the model and evaluations presented here, the results of

this thesis can precisely identify metrics and constraints on the employment of such monitoring techniques for real-time systems.

7.1 Hardware Implementation Results

This section presents the resource usage of the proposed CFTC architecture in comparison to different alternative solutions. Table 7.1 shows the implementation results of the full architecture described in Chapter 6, in the test-case FPGA. It has an integrated dual-core Cortex-A9 application processor. The proposed CFC scheme only required a total of 693 LUTs and 706 Registers, accounting for 1.3% and 0.66% of available resources of the FPGA. Our monitor required a total of 3654 LUTs and 2493 flip-flopss (FFs) in the multi-core configuration, accounting for 6.9% and 2.3% of the available resources of the Zynq platform. Meanwhile, the monitor's single-core configuration required 4.3% and 1.9% of available LUTs and flip-flops. The additional overhead is needed for the implementation of the trace parser and source aggregator, described in Section 6.1.1. These components parse and unpack trace frame information used to combine multiple packets with their execution unit's identification number. An extensive pipeline structure is required to avoid bottlenecks in parsing this frame, with many required comparison and shift operations.

Table 7.1: Total (and Percent) of used resources for our CFTC test-case implementation.

Module	*LUTs*	*FFs*	*Muxes*	*BRAMs*
Source Parser	1016 (1.91%)	729 (0.69%)	8 (0.02%)	1 (1.43%)
Source Aggregator	1341 (2.52%)	422 (0.40%)	0	0
Trace Decoder	860 (1.62%)	731 (0.69%)	37 (0.09%)	16 (11.43%)
Checker	435 (0.82%)	610 (0.57%)	0	6 (4.29%)
Control Memory	2 (0%)	1 (0%)	0	14 (10%)
Total	3654 (6.87%)	2493 (2.34%)	45 (0.11%)	37 (26.43%)
Total (Single-Core)	1297 (2.44%)	1342 (1.26%)	37 (0.09%)	36 (25.71%)

It is standard for hardware monitors and complex FPGA components to be implemented using a dedicated soft-core processor. Soft-core processors provide flexibility, are usually small, and implement enough computation capabilities for most tasks unrelated to high-bandwidth data processing. Table 7.2 presents a comparison of resource requirements between different soft-core processors implemented in the Zyqn-7000 and our monitor architecture. Rocket is a 5-stage in-order core implementing the Risc-V 32-bit and 64-bit general architectures [140]. Taiga is a similar but optimized RISC-V 32-bit based processor [141]. Meanwhile, Leon3 is a Spark V8-based soft-core [76, 131] and MicroBlaze is a Xilinx soft-core processor optimized for FPGAs such as Zynq. For this comparison, the Real-Time implementation of the MicroBlaze processor was used. It provides timing guarantees and an average best case between resource usage and performance.

The only soft-core that achieved a similar resource utilization to CFTC's configurations was Xilinx's MicroBlaze, targeted and optimized for the Zynq FPGA boards. In this comparison, the BRAMs used for the input FIFO buffers of the decoder stage are omitted. Such soft-core processors as replacement architectures would still require the same, or even more significant, input buffers. The same applies to the memory used for the control memory. However, its usage would depend on the configuration adopted for the control data, which can change with design and implementation. Using soft-core for both decoder and checking stages does not remove the difficulties inherent in processing the incoming trace information and makes it harder to implement a meaningfully portable and scalable system.

As discussed in Chapter 3, any additional overhead in the hardware area creates new points where faults may occur. Due to the design of CFTC and its focus in isolation from the central processor, errors due to SEUs in monitor components cannot influence or damage the executing software. However, any such faults can cause de detection of false positives, as a particle hit causes the corruption of information conveyed by the trace. The gate counts of the Cortex-A9 were compared to the estimated gate count of CFTC given its total resources to evaluate the size between the secured processor and monitor. Comparing gate counts gives a good estimation likelihood of soft-errors, given two devices implemented in the same technology and operating under the same conditions. The Cortex-A9 has a total of 20 M transistors or around 7 M gates. Our CFTC monitor estimate has 93 K gates, based on Xilinx's proposed average of 15

Table 7.2: Comparison of resource usage between the CFTC monitor and dedicated soft-core implementations.

Resources	*LUTs*	*Flip-Flops*	*BRAMs**	*Total Resources*
Rocket	17144	9058	10	26212
Taiga	3998	2942	10	6950
Leon3	6704	3640	2	9648
MicroBlaze	2465	2120	6	4591
CFTC	3654	2493	21	6168
CFTC Single-Core	1297	1342	20	2659
Ratio to CFTC Multi-Core Resources				
Rocket	4.69	3.63	0.48	4.25
Taiga	1.09	1.18	0.48	1.13
Leon3	1.83	1.46	0.10	1.56
MicroBlaze	0.67	0.85	0.29	0.74
CFTC	1.00	1.00	1.00	1.00
CFTC Single-Core	0.35	0.54	0.95	0.43
Ratio to CFTC Single-Core Resources				
Rocket	13.22	6.75	0.50	9.86
Taiga	3.08	2.19	0.50	2.61
Leon3	5.17	2.71	0.10	3.89
MicroBlaze	1.90	1.58	0.30	1.73
CFTC	2.82	1.86	1.05	2.32
CFTC Single-Core	1.00	1.00	1.00	1.00

gates per logical FPGA resource [142]. Therefore, our implemented multi- and single-core configurations correspond respectively to 1.3% and 0.6% of the total size of the hard-core CPU, further showing the minimal requirements used for a dedicated trace monitor system.

Besides its advantages in size and resources, this work combines the use of such a component with tested and validated protection techniques for hardware components and FPGAs. Methods such as TMR, memory scrubbing, and periodic resets can be used to improve the reliability of such components significantly [143].

Table 7.3: Approximate gate count in relation to implemented Cortex-A9 processor.

Module	*Gates*	*Ratio to CPU Size*
CFTC	92880	1.33%
CFTC (Single-Core)	40140	0.57%
Cortex-A9	2000000	100%

7.2 Models for Trace-Based Monitoring

This section explains a new model for trace profiling and identifies latencies related to trace-based monitoring architectures. The trace mechanisms available in the market can provide precise timing measurements of processor events. This work uses these measurements from trace packets to identify transmission and error detection latencies in the proposed architecture. Some of these latencies are intrinsic to the trace and debug subsystem. The model and measurements show how the different latencies and timing parameters relate to the detection latency and error response time. The two most important measurements for this monitor architecture are:

- The detection latency, i.e., the time from the moment an error occurs until the Checker identifies it.
- The response time, i.e., the detection time combined with the latency taken by the CPU to respond or handle this arising error.

Most safety-critical environments have soft or hard real-time requirements for executing different functionalities. These requirements also apply to any safety mechanism therein. To properly use program branch traces with real-time online monitors, we must be able to estimate the trace data bandwidth. This bandwidth directly influences the response and detection times of the system and monitoring unit. This estimation will shape the choice of hardware component for the monitor's implementation, e.g., if FPGA or ASIC, and the target processor's configuration.

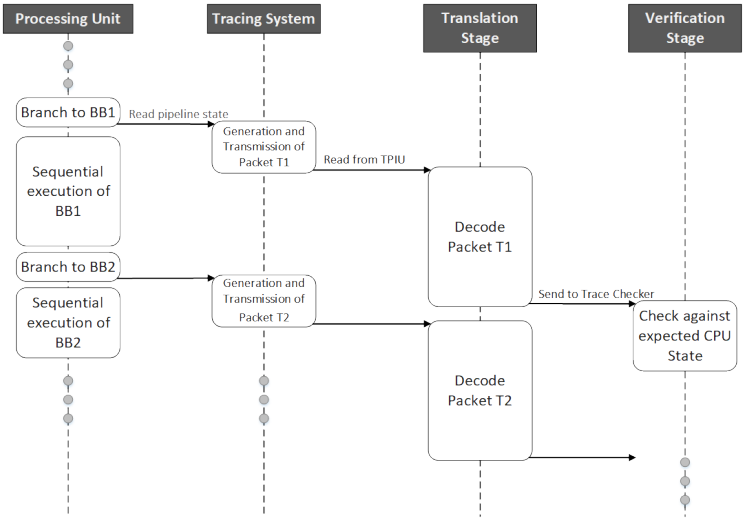


Figure 7.1: Timing diagram for the generation, transmission, decodification, and verification of a trace packet.

7.2.1 Error Detection and Error Response Times

The most precise way to profile and measure the time of different events inside the processor and SoC is through tracing. In offline profilers and tracers, this data is saved and later analyzed. All recorded events show an instantaneous timestamp relative to the CPU clock and can be directly related to one another as they all occur inside the central execution unit. In online tracing and, more specifically, online monitoring, multiple parallel components can create events in response to the processor's actions. However, these trace events are not instantaneously related to their causes. Figure 7.1 shows a timing diagram of a sample execution of a general program and the steps between executing a branch and its observation by the monitor. In this diagram, the monitor is represented by its two main processing stages, i.e., the translation and verification stages.

We can achieve increased observation capabilities into internal processor events through tracing. However, the observability functionalities only exist for the processor and not the trace subsystem. Therefore we can only correlate and

measure the time between different processor events. In the scenario presented in Fig. 7.1, we only have the information on how many cycles transpired between the 1^{st} and 2^{nd} branches. There are no direct means to measure the timings between branches and monitor events directly. As a result, it is only feasible to compare events that manifest inside the processor.

We must adequately characterize the error detection time of the system proposed here for its use in real-time critical systems. The detection time (L_{det}) can be expressed, as shown in Eq. (7.1) and Fig. 7.2, as the sum of the generation and transmission latency (L_T), the decoder latency (L_D), and the latency to check and verify the branch information (L_C). For a system with multiple clock domains, such as the test case where the monitor is implemented in an FPGA, we must consider the relation between CPU and monitor frequencies ($f = F_{CPU}/F_{MON}$). This way, all presented times can be expressed in units of processor clock cycles.

$$L_{det} = L_T + \frac{F_{CPU}}{F_{MON}}(L_D + L_C) = L_T + f(L_D + L_C) \tag{7.1}$$

These latencies are directly related to the internal structure of each hardware subsystem. The architecture proposed in this work can achieve decoding and detection times on the order of 10s of cycles, However, L_T is intrinsic to the SoC tracing system, i.e., the Cortex-A9 CoreSight, and we cannot precisely calculate it without access to the design descriptions. L_{det} cannot directly be measured as its endpoint lies outside the processor. Nevertheless, it is possible to measure the processor's total error response time (L_R), shown in the diagram of Fig. 7.2. The difference is that this reaction or response time has an additional term corresponding to the latency of the interruption service routine (ISR) (L_{ISR}).

All examined trace mechanisms from ARM to Intel provide a relative time in cycles between two elapsed branches. Each branch packet can inform how many cycles elapsed since the last executed branch. Therefore, it is viable to combine standard trace logging and profiling with the online monitor and measure the relative times of different events in the processor, including the execution of erroneous branches and the start of response routines.

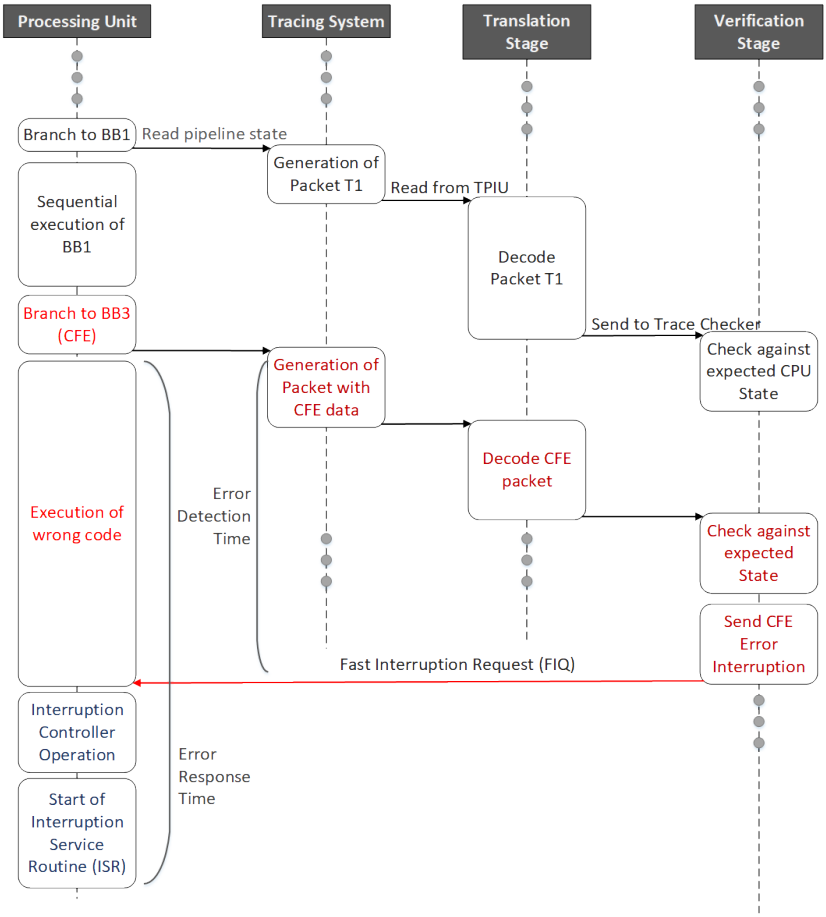


Figure 7.2: Timing diagram of error response time.

To characterize all latencies, the monitor generates an fast interruption request (FIQ) to the processor in two situations, when a CFE occurs and when a branch to a specific target address is taken. An external microcontroller configures these two modes before the benchmark executes. The FIQ is a fast, low-latency, and high-priority interruption that will cause the program control-flow to jump to a dedicated ISR. It is possible to know how long the trace packet took

between being generated and being received by the hardware unit if the setup collects branch execution times before the ISR is serviced, i.e., the time it took traversing the bus system. After the processor executes the target branch instruction, but before the FIQ is sent, the system will continue running and sending trace packets until it receives and handles the FIQ. Thus, only the cycle counts of packets between the interruption-triggering branch and the first jump to ISR are required to know the amount of time between both events. By performing this measurement several times at different execution points, one can minimize other factors that could account for the latency, such as packet sizes or execution time variance.

7.2.2 Trace Generation and Transmission Latency

The latency L_T to generate and transmit the trace packets is unavailable in any documentation. Therefore the system must be treated as a black box and work around the error response time to try and estimate L_T, as shown in Fig. 7.2. Therefore, we must minimize all other latencies that make up the measurement when running transmission latency investigations. The monitor is configured to send an interruption only when a specific branch occurs. The hardware waits for the program to execute B_1 and immediately sends an interruption once the corresponding trace packet is received. Hence, the Monitor verification stage is bypassed and only the necessary time to decode the packet needs to be considered. In these experiments, the processor runs at a frequency of 600 MHz, and the monitor at 100 MHz, corresponding to a factor $f = 6$.

The triggering branch is chosen to be a jump to a nearby area of memory. This choice reduces the total time required by the decoder, minimizing its latency. Following the format described in Section 4.3, the final packet has only 2 bytes, requiring a total of 12 cycles to be processed. The code for the corresponding ISR function is locked in the L2 Cache to minimize the interruption handling mechanism's latency. Thus, it can be optimistically assumed that the function will be ready for execution in the next couple of cycles following the interruption's arrival. The estimation for this latency is finally given by Eq. (7.2).

$$L_T = L_{resp} - f(L_D + L_C) - L_{ISR} = L_{resp} - 12 - L_{ISR} \approx L_{resp} \qquad (7.2)$$

For this equation to be a reasonable estimate, the measurement setup must also ensure that the triggering branch is the first branch traced by the system. This requirement guarantees that the trace bus and the monitor are idle at the moment of decoding and interruption generation, avoiding any additional delays from previous packets' execution. These extra difficulties are explored in the coming sections of this chapter. A total of 1000 measurements were executed to estimate the variable L_T, finally obtaining an average or expected value of:

$$E[L_T] = 346\,\text{cycles} = 577.7\,\text{ns}$$

At a time scale of this magnitude, the effects of L_{ISR} become highly significant. Therefore, L_{ISR} was also independently estimated for the same L2 Cache locked configuration. However, the measurements of this variable relied on the internal Core timers and the software-generated interrupt feature of the processor's GIC. A command is issued to the GIC to cause an interruption that is immediately handled. A timer is simultaneously started and then stopped at the beginning of the ISR. Through these tests, final estimates of the desired L_{ISR} variable were obtained:

$$E[L_{ISR}] = 250\,\text{cycles} \implies E[L_T] \simeq 100\,\text{cycles}$$

7.2.3 Real-Time Trace Processing Model

As first described in Chapter 6 and presented in Eq. (7.1), the latencies intrinsic to our monitor are relatively small but are dependent on the operation speed factor f between components and on the size of incoming packets. Real-time trace processing poses a problem. The system must balance the rate of packets generation to the monitor's processing speed. If a packet's data is not fully consumed before the next arrives, it generates a backlog of packets in the input buffer.

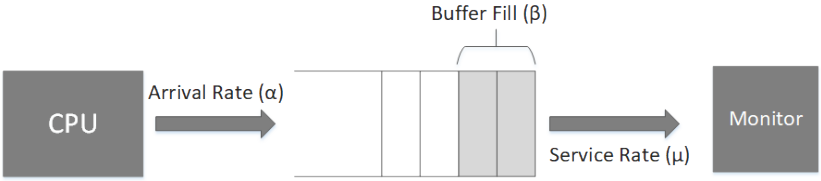


Figure 7.3: Queue model of an online real-time trace monitor for a single-core processor.

CFTC is analyzed in terms of a producer-consumer queue model such as the example of Fig. 7.3 to calculate and understand better the trade-offs and design constraints of the system. The objective is to model the evolution of the input buffer fill size (β) and the waiting time (W) experienced by a recently arriving packet. These two values are directly proportional and related through the trace service time (S) or its inverse, the service rate (μ), more commonly used in Queue Theory, see Eq. (7.3).

$$W = S\beta = \frac{\beta}{\mu} \tag{7.3}$$

The service time corresponds to the time necessary to process a complete packet, i.e., its decoding time. Due to all current trace architectures' irregularities and focus on data compression, multiple cycles are required to process a packet. As shown in Chapter 6, to process an entire packet in a single cycle requires many parallel comparators and shift-register networks for realignments. The best cost-effective approach is implementing a pipelined decoder such as the one implemented in our CoreSight use case. In this case, the service time is directly equal to the packet's size or payload in bytes (P_{pkt}) times the frequency factor f. The payload of a trace packet can vary greatly, but it stays on an average of 1 to 2 bytes for a minimal configuration. Figure 7.4 shows the distribution of packets sizes for all our test-case benchmarks. The benchmarks used and these results are further explained in Section 7.4.5.

$$S = fP_{pkt} \tag{7.4}$$

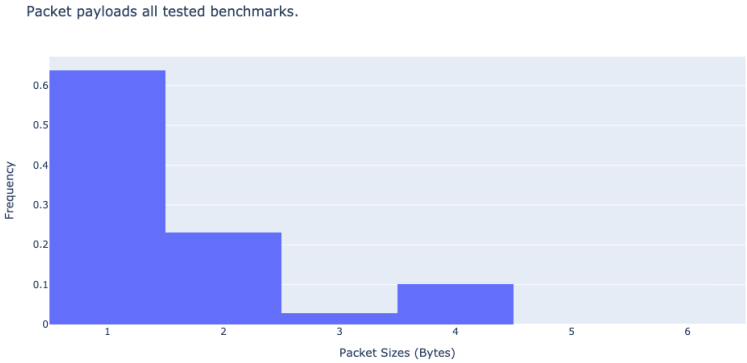


Figure 7.4: Distribution of packet sizes for all tested benchmarks.

The main concern is in determining the packet generation rate. We must look at the running software and its branch execution rate. The packets' arrival rate (α) is defined as the inverse of the packets' inter-arrival time (Λ). The inter-arrival time is the time between the execution of two branch operations, which is the execution time of the individual basic blocks of the program. This block execution time is a random variable. Given delays in memory access or variations in the core pipeline state, the execution time of individual blocks can change. The whole program's inter-arrival time ($\Lambda(t)$) can be modelled as a mixture distribution of its individual basic block execution times ($\Lambda_{BB_i}(t)$), shown in Eq. (7.5), by considering the inter-arrival time, or execution time of any basic block in the program (Λ_{BB})

$$\Lambda(t) = \sum_{i=0}^{N} p_i \Lambda_{BB_i}(t) = \sum_{k=0}^{N} \frac{f_i}{N_{BB}} \frac{I_{BB_i}}{IPC(t)} = \frac{1}{\alpha} \tag{7.5}$$

This distribution combines the distributions of all N blocks in the program, weighted by their probability of execution (p_i). This probability can be estimated by the frequency with which the block is executed over the length of an execution (f_i), divided by the total number of executed basic blocks (N_{BB}). It is also the total number of branches into a block i divided by the sum of all ex-

ecuted branches. The execution time of a block i can be further approximated by its number of instructions (I_{BB_i}) and the processor's instructions per cycle (IPC), which can be seen as the random, varying variable over the program's execution.

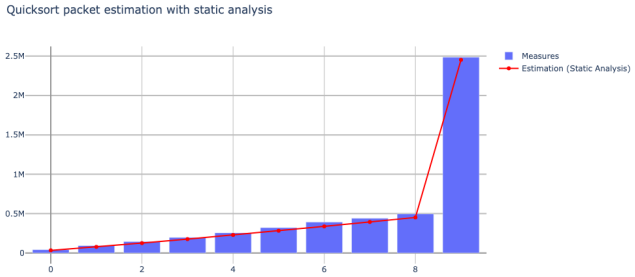
Both static and dynamic binary analysis techniques were combined to find all these values. A program's structure and input dictate the number and frequency of branches' and basic blocks' execution. Three benchmark algorithms were taken, Quicksort, Dijkstra's Shortest Path, and AES-256, and used branch analysis techniques described in Chapter 5 to estimate the total number of branches given their input sizes. The total number of estimated and measured branches and the model errors are shown in Figs. 7.5 to 7.7. The models' error tended to zero in all algorithms as the input size increased. As the input size increases, each algorithm's main processing sections are stressed, minimizing the effect of other program blocks.

By combining typical worst-case execution time (WCET) analysis with branch count analysis, it is possible to obtain the overall average time between branches statically. This analysis also shows how the algorithm's most executed basic block can estimate their overall branch arrival rate. So, given the arrival rate of the most executed basic block ($\Lambda_{BB_{max}}$), the following relation is obtained:

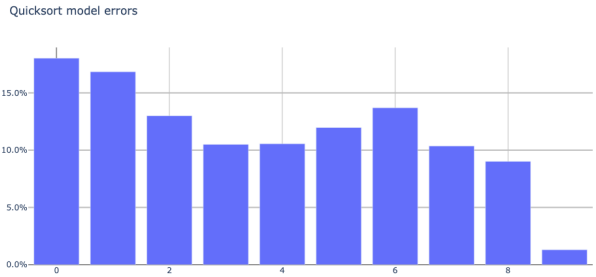
$$\Lambda(t) \approx \Lambda_{BB_{max}}$$

The test-case algorithms were profiled, and their overall branch inter-arrival time distribution was compared to an exponential gamma distribution to verify this approximation. The profiled distributions and fits for three of the algorithms in two different compiler optimizations are presented in Figs. 7.8 and 7.9 for Dijkstra, Figs. 7.10 and 7.11 for fast fourrier transform (FFT), and Figs. 7.8 and 7.9 for Quicksort. Except for Fig. 7.8, all fits properly represented the probability distributions, even if in a worst-case distribution scenario, of the tested algorithms. The fit for the Dijkstra algorithm without optimization failed the basic assumption of an exponential distribution. Due to the algorithm's longer execution time, the inter-arrival time distribution is skewed to the right and more closely determined by a binomial rather than exponential gamma fit.

A reasonable worst-case approximation value can be achieved after finding proper arrival and service rates models. The service time can be approximated to the average packet size times the frequency factor f. This approximation



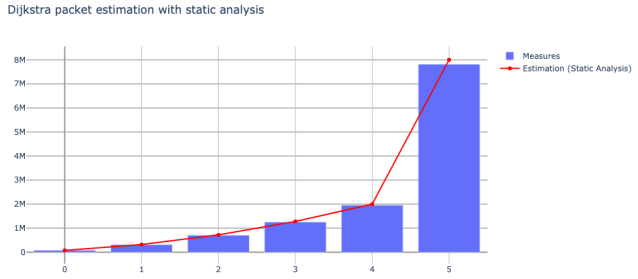
(a) Measured and estimated number of branches



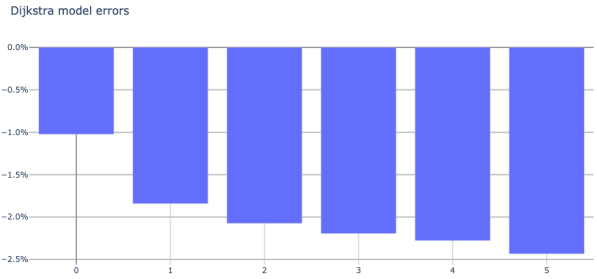
(b) Error percent

Figure 7.5: Estimation and model error of trace data generated for Quicksort algorithm according to input size.

is used to calculate the final average waiting time of incoming trace packets. The arrival rate can also be similarly approximated through a constant by the inverse of our fitted function mode, i.e., the most common inter-arrival time. This approximation simplifies the buffer fill rate calculation, leading us to the derivation of Eq. (7.6) for the buffer fill over time.



(a) Measured and estimated number of branches

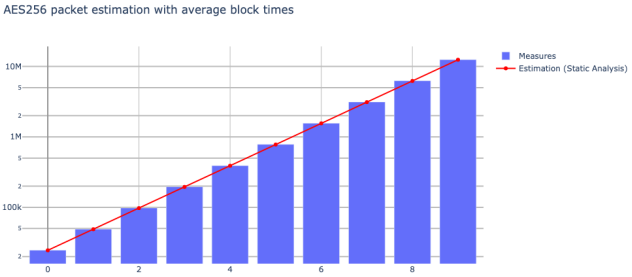


(b) Error percent

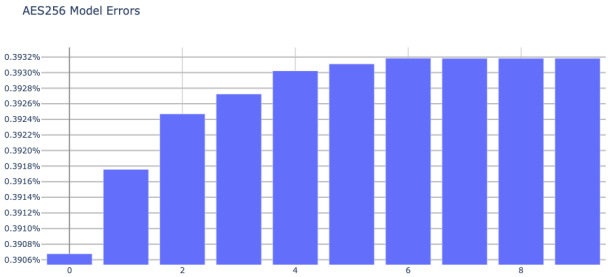
Figure 7.6: Estimation and sampling of trace data generated for Dijkstra algorithm according to input size.

$$\begin{aligned}\beta(t) &= \alpha t - \mu t \\ &= (\alpha - \mu)t \\ &= \mu(\frac{\alpha}{\mu} - 1)t \\ &= \mu(\rho - 1)t, \quad t > 0, \quad \rho > 1\end{aligned} \tag{7.6}$$

As long the arrival rate is lower than the service rate, all packets can be processed in time, and the buffer will remain bounded. The ratio $\rho = \frac{\alpha}{\mu}$ is defined as the buffer utilization factor. As long as the utilization remains



(a) Measured and estimated number of branches



(b) Error percent

Figure 7.7: Estimation and sampling of trace data generated for AES encryption algorithm according to input size.

below one, the backlog of packets will not increase. However, if ρ is greater than one, the buffer will start to fill. Section 7.4.5 presents an analysis of how the instantaneous utilization factor influences the buffer fill and, ultimately, the waiting times of packets in the queue before they are verified. This increase in buffer fill, in turn, can lead to a considerable increase in error detection times (L_{det}) of our or any trace-based monitor architecture.

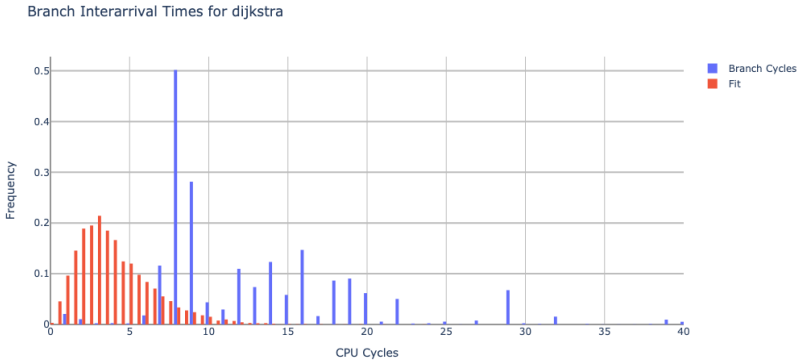


Figure 7.8: Branch Inter-arrival distribution for Dijkstra (compiled with o0).

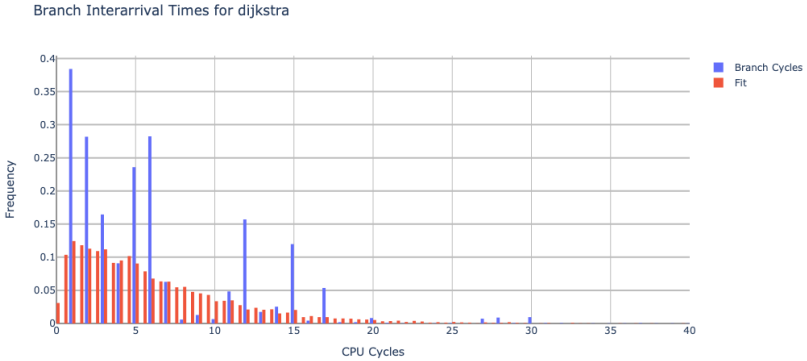


Figure 7.9: Branch Inter-arrival distribution for Dijkstra (compiled with o3).

7.3 Vulnerability Analysis of Hardware Monitor

A big problem of complex safety mechanisms is the lack of an additional vulnerability analysis [9, 10]. Most Techniques end up not meeting their desired or announced goals due to unforeseen overheads in program execution, the

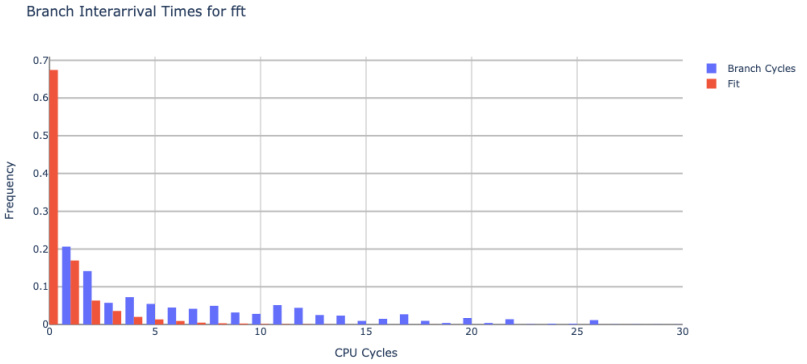


Figure 7.10: Branch Inter-arrival distribution for FFT (compiled with o0).

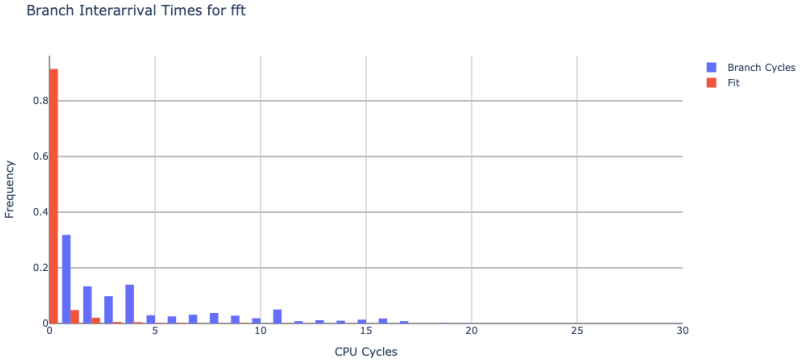


Figure 7.11: Branch Inter-arrival distribution for FFT (compiled with o3).

liveness of critical memory elements, or additional memory components. In Rhisheekesan et. al. [10] the vulnerability of software CFC schemes was studied, including one hardware CFC, the CFCET technique [65]. This technique was one of the principal ones to explicitly compare trace branch targets against a CFG generated in compile time. Their analysis shows how using program tracing gives a significant advantage for real-time monitoring as it has low vul-

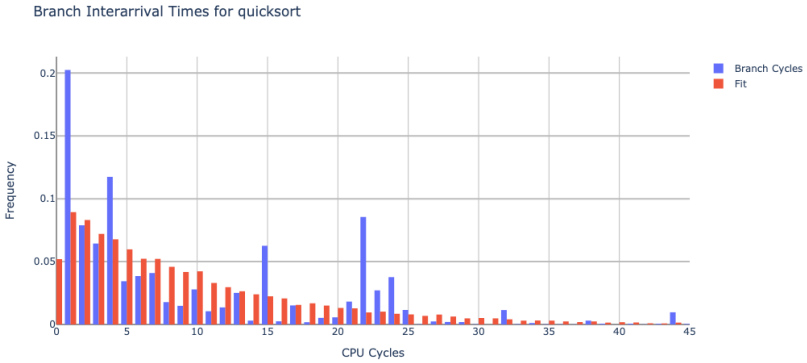


Figure 7.12: Branch Inter-arrival distribution for Quicksort (compiled with o0).

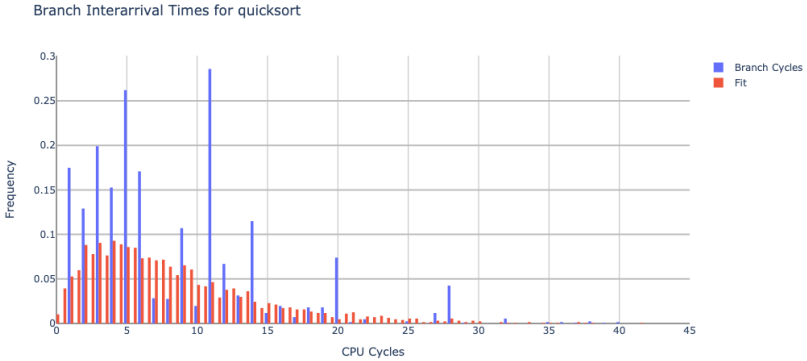


Figure 7.13: Branch Inter-arrival distribution for Quicksort (compiled with o3).

nerability costs [10]. It is thus a necessity of CFC techniques to also explore new trace architectures such as CoreSight and Nexus that do not incur time overheads to the running software [114, 125].

As these studies show, correctly understanding the added complexity and vulnerability of fault tolerance techniques is extremely important [9, 10]. The final

efficacy of techniques that aim to protect against faults depends on their not failing or lowering the architectural vulnerability of the final system beyond their added reliability benefits. Hardware monitors already have an advantage against software-based approaches in which they do not require code modifications and thus are isolated from the running application. However, this independence between monitor and application is only valid if the monitor unit does not share any system or memory buses with the processing units. It might cause bus contention as it consumes bandwidth and increases the overall execution time due to increased memory access times.

However, hardware-based methods present an extra system complexity from the additional trace and checker hardware. Unlike other techniques, this increase in complexity does not directly correspond to an increase in program or system vulnerability as these additional components do not impair the software execution if they fail. Failures in such components may generate, at worst false negatives if an error is undetected or false positives if they identify an error that did not occur. Calculating the components' vulnerability is also crucial in this case. If the monitor stops being operational or due to errors in the trace data, it fails to detect an error in the control flow. By implementing the monitoring components inside an FPGA we can profit from additional FPGA protection mechanisms such as scrubbing, or architectural fault tolerance methods such as TMR [80, 144].

One of the drawbacks of working with hard-cores is the lack of micro-architectural information about the implemented processors and their subsystems. A direct architectural vulnerability factor (AVF) analysis of the whole hardware platform cannot be performed. However, the AVF of the monitor can be calculated by considering the monitoring system as composed of two main parts: the trace bus and the CFC unit itself. Calculating the trace bus's average bandwidth allows us to estimate the average probability that a fault may hit one of the packets. All trace data used by the monitoring system must be counted as part of the vulnerable architecture. If a packet does not contain the correct information, the monitor will not detect processor faults, or it may accuse false CFEs to the system. With the complete architectural description of the CFC unit, performing its AVF calculation is very straightforward. At the same time, vulnerability analysis of its programmable FPGA description can be performed based on the size of its configuration memory.

The vulnerability of error detection mechanisms can play a significant role in the final system reliability [9, 10]. Radiation experiments on SRAM FPGAs have shown that proper fault tolerance techniques such as TMR and scrubbing can significantly increase the reliability of FPGA designs, also suggesting that FPGAs may have ceased to be the weakest link in the overall system reliability [145]. As it has been shown, the overall FPGA area and resources required by this work's CFTC technique are minimal. Therefore it can be easily combined with other FPGA protection techniques to safely monitor the processing system without significant risk of failure to the CFC unit itself. However, the internal CPU trace bus is typically used only for test and validation during software development and does not have any additional safety features.

The size and vulnerability of an FPGA design are associated with its configuration memory size and the number of critical bits in that design, i.e., the design's ACE bits configuration [146]. It has been shown that the number of critical bits of programmable hardware components is very low, often not more significant than 15% of the total configuration memory [147]. The critical bits correspond to the ACE bits of the FPGA configuration, any modification of such bits will cause the underlying FPGA architecture to incorrectly route signals across modules or incorrectly calculate the result of a function.

An analytical AVF calculation of the tracing subsystem can be performed by using Little's Law and knowing the average number of trace bits generated for each branch packet and the average duration of a program's basic block [47]. Although the trace bus is part of the hard-core processor architecture and generally no information about its implementation is available, the AVF of the trace bus can be computed by knowing, on average, the amount of trace packet data that stays on the bus. Assuming that every bit in a program trace packet is an ACE bit, the residency of packets in the bus needs to be calculated before they reach our hardware monitor, as this will be related to the probability that a fault can corrupt this data before it is parsed. Little's Law, defined in Eq. (7.7), states that given a constant and steady system, the long-term average number of elements inside the system (E) will be equal to the long-term average arrival rate of elements in the system (α) times the average waiting time, or latency, they spend in the system (L).

$$E = \alpha L \tag{7.7}$$

The rate α, defined by the branch inter-arrival time, depends primarily on the basic block's size since a new packet is always generated at the end of a block. The latency required here was also previously calculated in Section 7.2.2 and found to be on the order of 100 cycles ($L_T \approx 100$) Since a fault in the trace bus will prevent the correct reconstruction of the CPU state, the reliability of the incoming data must be identified. To calculate its AVF, the number of average vulnerable memory elements over time must be known [43]. Following the definition laid out in Section 2.3.4, the expression for the internal trace system takes the form of Eq. (7.8).

$$AVF_T = \frac{E_T}{N_T} = \frac{\alpha L_T}{N_T} = \frac{P_{pkt} \frac{IPC}{I_{BB}} L_T}{N_T} \tag{7.8}$$

Where P_{pkt} is the average branch trace packet payload in bytes, I_{BB} is the average number of instructions for executed basic blocks, and IPC is the estimated or the given number of instructions per clock cycle for a single processor core. The AVF represents the fraction of bits in the hardware vulnerable to errors. We assume that all bits in a trace packet must be correct for this calculation. AVF is approximated by the average arrival rate of trace data (α) times the average time required to traverse the bus (L_T). Finally, this is divided by the bus's total internal memory (N_T) to reach the final factor.

Although N_T is unknown, Eq. (7.8) can be used to find the average amount trace data being sent E_T. Section 7.4 delineates the performed fault injection tests and also characterizes the benchmarks used for those tests. Tables 7.5 and 7.6 present the collected measurements for all benchmarks. With tested values of $IPC = 1.1$, $P_{pkt} = 2\,\mathrm{B}$, $I_{BB} = 7$, and $L_T = 100$, the total average number of critical data elements inside trace bus is estimated as:

$$E_T = \frac{2 * 1.1 * 100}{7} \approx 32\,\mathrm{B}$$

This value is smaller than the total amount of bytes present inside a single core's pipeline. Even using the maximum possible IPC of the Cortex-A9 processor only gives an average number of elements of:

$$E_T^{max} = \frac{2 * 2.5 * 100}{7} \approx 72\,\mathrm{B}$$

Caches of modern processors can store data in the order of kilobytes or megabytes. Thus, the trace system does not significantly add to the overall number of vulnerable bits in the CPU architecture. Our theoretical results also match with radiation experiments performed by Peña-Fernandez et al. [148] where the trace mechanism was used to detect faults through their PDTC architecture. To correct errors in the programmable logic that may affect their component, a Xilinx's soft-error mitigation (SEM) controller could correct and detect faults in the FPGA. Thus, even a complex processor architecture such as the Cortex-A9 CoreSight can reliably be operated in a dangerous environment.

7.4 Fault Injection Tests

This section describes the methodology for evaluating the CFTC error-monitoring system under a fault injection environment. First, the methodology behind fault injection tests is explained, and the possible types of tests available are outlined. The section then explains the setups for the chosen types of tests that best fit the current test-case scenarios. Finally, the error detection and timing results are presented. They show a definite set of metrics and design choices that can be used for the final application of this and similar monitors in actual industry implementations.

7.4.1 Methods for Evaluating Fault Tolerance Techniques

One of the most common methods to evaluate the efficiency of fault protection and detection mechanisms is through hardware or software fault injection. A fault injection campaign consists of running the chosen application and hardware system multiple times, either in an irradiated environment, such as radiation-based FI, or in a controlled setup that inserts a single fault or bit-flip during an execution, called simulated or software-based fault injection.

Such campaigns intend to approximate the system's probability of failure, $P(F)$, depending on the used fault model, such as described in section 2.3.3. During a fault injection run, if the system stops responding or reaches a state from which it cannot recover to keep its functionality, it is said to have failed. To calculate $P(F)$, the total number of failure events, Ω_F, is counted against

the total number of executions or number of faults injected, N. Their ratio is then used to predict the probability that a fault in the system causes a failure:

$$P(F) = \frac{\Omega_F}{N} \tag{7.9}$$

For radiation injection campaigns, where faults occur randomly and uncontrollably, N is the number of total executions under radiation, and $P(F)$ is the execution rate of application runs that did not execute correctly. In targeted and controlled fault injections, where faults are inserted in specific simulated hardware elements or specific software instructions, exactly one fault is injected per execution, and N is the total number of injected faults. In such cases, $P(F)$ gives the rate of single faults, causing a system's failure. These campaigns are performed both for the unprotected application, called the baseline or unhardened configuration, and for the one protected through the method we wish to validate, called the hardened version. If failure probability approximations are run for both the baseline, $P_B(F)$, and hardened cases, $P_H(F)$, we can calculate the reliability gain given by the technique:

$$r = \frac{1 - P_H(F)}{1 - P_B(F)} = \frac{R_H}{R_B} \tag{7.10}$$

Where $R = 1 - P(F)$ is called the system reliability, or the probability that the system executed correctly and did not fail. For a sufficiently large N with well-distributed faults, r will equal the real gain given by the technique. Different approaches have different properties and correspond to trade-offs in cost, accessibility, and controllability. Table 7.4 summarizes 5 common approaches to fault injection.

Accelerated radiation testing, or heavy-ion radiation, uses radiation facilities and particle accelerators to simulate the exact effects of environmental radiation on the board. The tested device is placed in a sealed vacuum environment and then irradiated with particles while the test application is executed. This approach can test the whole device and stress all possible faults inside the device. However, test facilities are costly, and depending on the radiation source, the tested chips must be modified and decapsulated so that the particles can affect the individual transistor gates.

Table 7.4: Summary and properties of different fault injection approaches [31, 149].

Property	*Accelerated Radiation*	*Simulated Hardware*	*Runtime Software*	*Static Software*
Accessibility	Very High	Medium	Low	Low
Controllability	Low	High	High	High
Intrusiveness	None	Low	High	Low
Repeatability	None to Low	High	High	High
Injection Speeds	Medium	Variable	Low	Medium
Cost	High	Medium	Low	Low

Simulated hardware requires a system description and a tool to execute the software application. They are highly dependent on the type and level of detail the hardware description gives. Cycle accurate and bit accurate simulations usually require a complete VHDL description of all hardware components and their connections, usually producing simulation times several times that the application requires to run on the actual hardware. Instruction accurate simulation can be run much faster than the application requires, but at the component detail and accessibility cost. It must also be noted that simulation environments for complex systems, such as multi-core FPGA SoCs, usually must be built from scratch and that access to model descriptions is very costly and not easily accessible.

Software implemented fault injection presents the most negligible costs and has limited access to injection targets. These approaches depend on defined interruption routines that stop the application at specific time points and insert faults at specific memory addresses or registers. Other approaches work by creating a modified execution binary with the fault pre-inserted in a specific instruction. These different binaries are then executed, and the modified instruction simulates a possible upset in pipeline or register file memory components. Although very cheap and reproducible, these last approaches overestimate the effect of faults in instruction registers compared to other system elements.

The use of branch tracing hardware modules brings difficulties regarding evaluation procedures. Current modern simulators such as Gem5 and OVP lack a proper simulation of architectural instruction trace mechanisms, opting to pro-

vide debugging mechanisms as features of the simulators themselves. Available simulators are optimized for instruction level and architectural simulation. They rarely provide the means to simulate a complete SoC or FPGA integrated system. The best option for software-implemented hardware simulation would be a completely customized and dedicated Register or Transaction Level system simulator for each specific platform. Implementing such a simulator runs outside of the scope of this work and could be a different topic of interest for future extensions and analyses of this work.

For software fault injection, two approaches are possible: either directly modifying the program at the binary level or using unique interruption routines that modify the architectural register values to insert a fault before returning to the application. Both approaches have trade-offs that must be analyzed. Using interruption-based fault injection creates overheads and modifies system behavior as all cores in a multi-core setup must be simultaneously stopped while the injection routine is running.

In binary injections, measures must be set to ensure that the faulty instruction will execute only in one core. For asymmetric multi-processing (AMP) setups, where each processing unit works on its independent binary code, this can be easily done, but for symmetric multi-processing (SMP) setups, most commonly used in OS-based systems, this is not the case. One solution for SMP setups is creating similar but separate functions for processing units. Like this, faults can be easily injected, but the advantages of SMP such as shared synchronization data structures used by operating systems are still kept.

Many different fault injection approaches have been proposed in the literature with trade-offs in cost, accessibility, and controllability. Due to trace, hardware simulation approaches and injections through software interruptions are challenging to adopt. Therefore, this work opts for static software injections for the performed tests, i.e., modifying the code's executed binary before each execution. This chapter presents results for both runtime and static software fault injection campaigns.

7.4.2 Fault Classification

After defining the proper fault-injection campaign setup, each run's outcome must be classified. In the campaigns performed here, the outcome of each execution is classified according to the following categories:

- **Masked**: represents fault injection runs that ended correctly and with the correct output.
- **SDC**: for silent data corruption (SDC), characterizes fault injections that caused errors in the output data.
- **Hang**: indicates the runs that presented abnormal program termination.
- **Timeout**: for program runs that failed to terminate on time.

The resulting errors can be classified once the fault is injected and the application finishes executing. The definitions from Chapter 2 are used and classified as ACE bits or un-ACE bits, as first defined by Mukherjee et al. [43]. For a narrower classification, the errors generated by ACE bit faults can be separated in SDC or Hangs. If the application finishes normally, but its output differs from the expected value, it is classified as an SDC. If the application fails to meet its deadline or is aborted due to an internal error, it is classified as a Timeout or Hang.

Timeouts are defined by all cases where the application did not complete its function and the Host or control server stopped it. If the application stopped responding for more than 1 s, it is considered unrecoverable and a Timeout. Hangs were categorized whenever the processor detected an incorrect operation and signaled it with a hardware abort. These are internal errors flagged by the processor that generates a hardware exception interruption. The Cortex-A9 generates exceptions for:

- *Undefined Instructions*, for when the CPU tries to decode an invalid Opcode;
- *Data Aborts*, flagged by the MMU when a data transfer instruction attempts to load or store data at an illegal address; and

- *Prefetch Aborts*, also thrown by the MMU when the system attempts to execute an instruction pre-fetched from an illegal address.

Injected faults were similarly classified for the system running under our CFTC monitor. The decision flow diagram for the classification of injected faults is shown in Figs. 7.14 and 7.15. Figure 7.14 shows the classification tree for fault injection runs without the CFTC protection mechanism. When the monitor is present and a fault is undetected by the monitor, the same scheme is used to classify it. Otherwise, if a fault was detected the scheme shown in Fig. 7.15 was used.

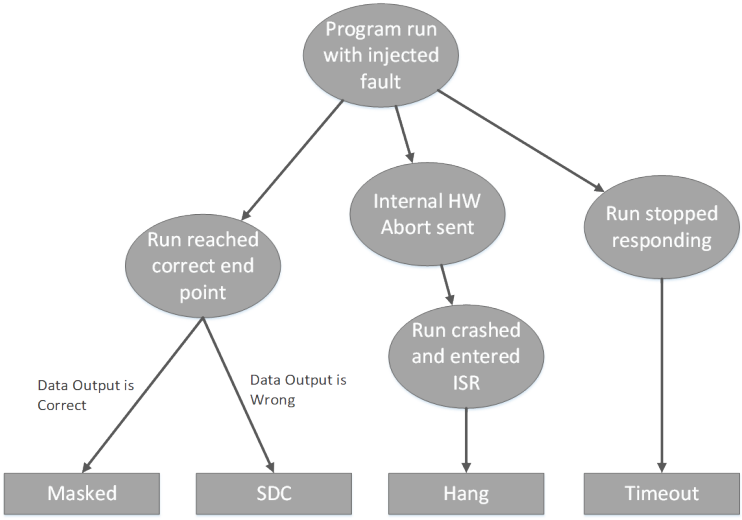


Figure 7.14: Fault classification diagram for unhardened injections and undetected faults.

Whenever the system detected a fault, an FIQ signal was generated to the offending core. The program was then left to handle the interruption and continue the execution of the application. All detected Hangs generated an interruption back to the processor. After handling the interruption, the processor was left to finish the execution of its function. If the CPU could respond to this interruption and the application finished correctly, it was counted as a Masked fault. If it finished with corrupted data, it was marked as a SDC, or as a Hang if it was unable to finish. However, if the processor was unable to respond to

the interruption under 1 ms, it was counted as a Timeout. This classification allowed us to identify the differences in outcome for the same injected faults and the reduction in Timeouts through our error detection monitor.

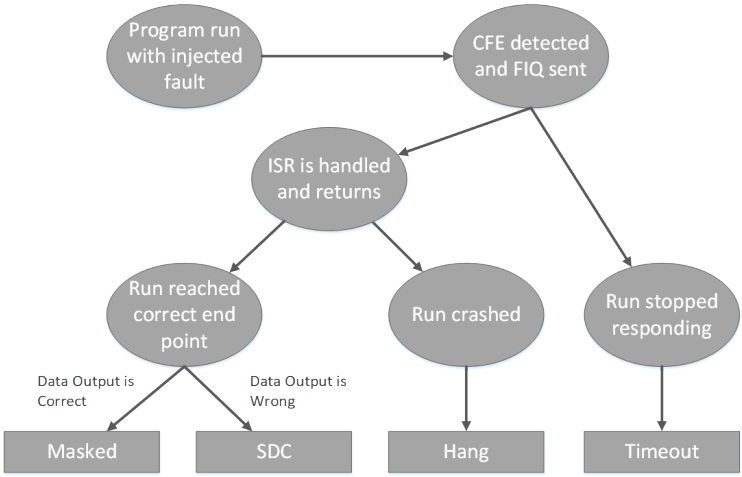


Figure 7.15: Fault classification diagram for detected faults.

7.4.3 Static Fault Injections

In this first injection setup, the instruction opcodes of each of the benchmarks are targeted. These fault injections are implemented through instrumented binaries. The execution flow of an injection run is shown in Fig. 7.16. A power-on reset was performed with an electrical switch to guarantee total isolation of system states between runs between each execution. All detected CFEs generated an FIQ signal back to the system. The monitor also provided back to the system CFE information such as the instruction address which it occurred and the expected and erroneously executed target branch addresses. In injections without CFTC, Hangs were detected directly by the processor and MMU whenever the application tried to access or execute an invalid memory region or invalid operations. In all cases, if the CPU entered an infinite loop and did not finish, the run was considered a Timeout.

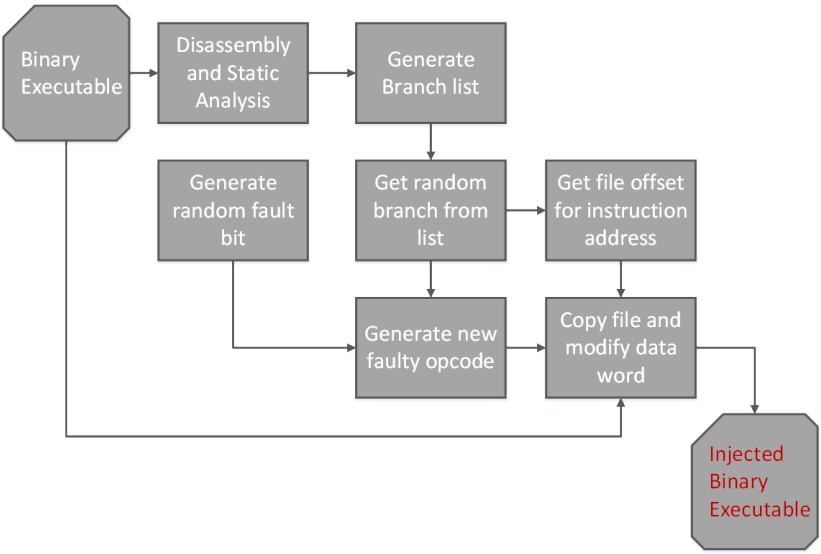


Figure 7.16: Injection interruption scheme for static injection.

Static tests allowed us to evaluate a more extensive set of possible CFEs with a finer granularity. By directly changing branch opcodes of the target binary, all the types of errors described in Section 2.3.3 can be evaluated. Even a single bit-flip can drastically change the resulting operation. The opcode representation of branch operations is shown in Fig. 7.17. Any changes to bits 0 to 23 of both branch and branch-and-link (procedure call) operations will result in a false target address jump. For the `POP` operation, bits 0 to 16 select which registers and how many values will be popped from the stack, pointed by the SP register. Bit 0 selects register `R0` as a destination, bit 1 `R1`, and so forth. Bit 15 corresponds to the PC. If selected, we are loading a previous LR value saved in the stack. This sequence corresponds to the standard operation of function returns in the ARM ABI.

This type of injection also allows the evaluations to precisely target all regions inside the code. However, its main drawback is the difficulty in adequately evaluating the error detection time's growth from the buffer fill (L_{det}). Because all faults are decided and injected statically, they are activated on the

faulty branch's first execution before the application begins to run. This static modification implies that branches part of long loop structures can only be evaluated under faults at the beginning of the loop sequence and never at the end. Dynamic injections are also performed to compensate for this, where instead of branches, general-purpose registers were targeted and changed at specific points throughout the program's execution.

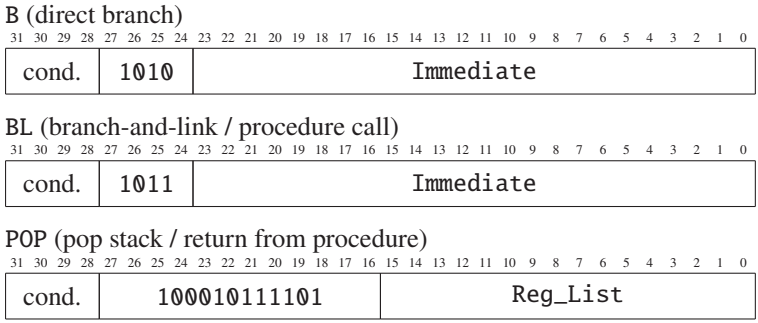


Figure 7.17: Opcode representations for branch, procedure call, and return operations.

7.4.4 Runtime Fault Injections

The main objective behind runtime fault injection tests is to exercise our monitor's response to faults at different points across the application's execution. It analyzes the vulnerability of the register file of the Cortex-A9 processor running sequential benchmarks in a single-core processor. For this runtime injection, a SEU fault model is assumed, affecting the architectural register file component of a single core of the processor. For each injection, precisely one bit is flipped during a randomly chosen point of the program's execution. The SEUs are generated through a specific injection interruption routine. A timer is configured and used to generate an interruption that will modify the registers saved inside the program stack, see Fig. 7.18 At the beginning of the ISR, registers R0 through R12 are saved in the stack together with the LR. By modifying the LR, PC faults can be simulated on the return from function calls.

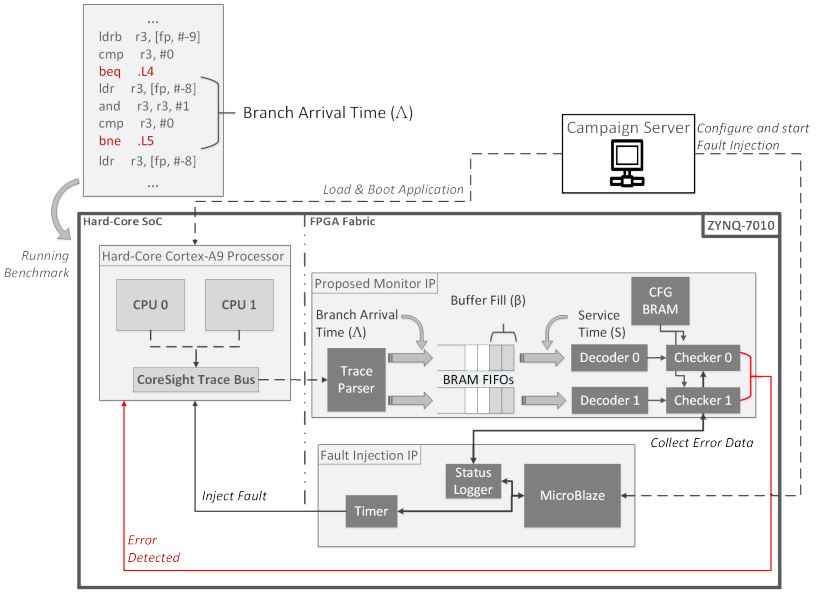


Figure 7.18: Hardware architecture configuration for the dynamic fault injection setup.

Once all registers are saved in the stack, the injection ISR starts execution, and it modifies the selected bit of the targeted register directly in the stack. When the handler routine ends, and the values are popped back to the registers, the targeted register will now hold its old value plus the flipped bit simulating an SEU. When the interruption ends, the modified stack values are re-loaded to the physical registers, and the benchmark resumes from the same context except for the newly injected fault. Figure 7.19 presents a diagram showing the use of the dedicated interruption routine to insert the faults. When the program starts, before the execution of the benchmark function, three random values are generated: the timer's timeout, the register number to be injected, and the mask value to choose which bit to flip. These values are logged on the Host and used to correlate results extracted from trace with the register affected.

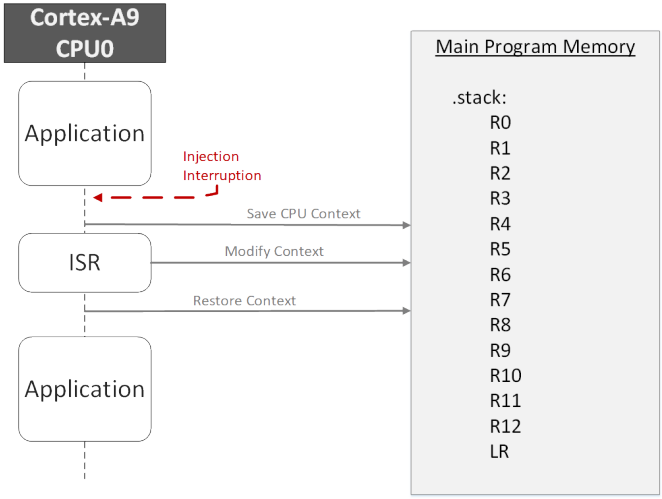


Figure 7.19: Injection interruption scheme showing the use of the stack.

7.4.5 Fault Injection Results

The objective of performing these fault injection experiments was to verify the proposed architecture's ability in detecting a specific set of processor errors, CFEs. It is not to fully estimate the soft-error rate (SER) of the design but to show its strengths, drawbacks, and ability to detect control-flow errors when they occur correctly. Five thousand (5000) fault injections were executed per benchmark use case through each explained fault injection method to evaluate the final monitoring platform. Six different benchmarks from the embedded MiBench suite were used, combined with two compiler optimizations for a total of 12 different use-cases [150]. The six algorithms selected as use cases were chosen due to their relatively complex or intricate control-flow. They were selected according to the following domains: Cryptography and Security (AES, LZO), data processing (FFT, Matrix Multiplication (MM)), and data analysis and management (Dijkstra Shortest Path, Quicksort).

All benchmarks were implemented in the C language in a baremetal environment and compiled with Linaro's version of the Gnu Compiler Collection (GCC) for embedded ARM (GCC version 8.2.0). Each algorithm was compiled

with and without optimizations through the use of *-o0* and *-o3* compilations flags. These flags focus on structural optimizations of the code, which was the primary influence of interest. The optimization *-o3* flag of the *gcc* compiler was detailed more extensively in Section 5.1.2. Also, SIMD and FPU optimization flags were explicitly not used, as our main objective was to see the effects of different control-flow compiler optimizations. Each resulting object binary was statically linked to an optimized version (*-o3*) of the C library provided by GCC to generate the final application executables. Table 7.5 presents a summary of each use-case's binary structure. These results match other reported CFG structures from common control-flow errors related studies for average basic block sizes and percentages of branch instructions [9, 41, 66].

Table 7.5: Static analysis results from tested use-cases.

Benchmarks without optimizations (-o0)							
Measures		*Dijkstra*	*FFT*	*Quicksort*	*AES*	*LZO*	*MM*
Instruction Count		319	8077	567	842	2682	685
Function Calls		2	479	10	18	76	6
Basic Blocks (BB)		29	1438	76	66	493	108
Instructions per BB (max/avg)		28/11	60/5.6	26/7.5	87/12.76	74/5.44	36/6.34
Benchmarks with optimization(-o3)							
Instruction Count		280	4259	283	835	968	797
Function Calls		3	253	2	4	10	25
Basic Blocks (BB)		54	804	47	21	146	142
Instructions per BB (max/avg)		18/5.2	89/5.3	20/6	385/39.76	50/6.63	21/5.61

Trace-based profiling was performed over the benchmarks under the same configurations used for the fault injections to evaluate the model described in Section 7.2. Each benchmark was executed with a randomly generated input of a defined size. The trace data was collected inside the board's BRAMs buffers and sent through a dedicated ethernet connection back to the host pc. The choice of input size had to be kept small so that the available buffers could store the complete execution trace history. In previous tests, the data was parsed by a dedicated external FPGA component to collect information such as the total number of branches or the average branch inter-arrival time.

However, obtaining more detailed information required an analysis of all the execution trace data of an execution. The obtained profiling results are given in Table 7.6.

Table 7.6: Dynamic analysis results from profiling of full program trace.

Benchmarks without optimizations (-o0)						
Measures	*Dijkstra*	*FFT*	*Quicksort*	*AES*	*LZO*	*MM*
Input/Sample Size	30x30	30	150	64	8kB	9x9
Branch Inter-arrival (Λ) (avg)	13.18	7.23	9.84	25.01	13.80	8.38
Branch Inter-arrival (Λ) (std)	7.23	7.39	9.64	18.58	3.92	8.90
Execution Time (cycles)	99038	79680	93567	130956	83039	122547
IPC (avg)	0.85	1.13	0.98	0.88	1.13	1.27
IPC (std)	0.37	0.90	0.75	0.49	0.51	0.99
Taken Branches	5970	5843	7106	4644	3826	8047
Not-Taken Branches	1334	2819	1776	490	1127	3754
Packet Size (B_{pkt}) (avg/std)	1.5/0.5	1.9/1.0	1.5/0.6	1.5/0.5	1.0/0.3	1.9/1.3
Benchmarks with optimizations (-o3)						
Input/Sample Size	30x30	30	150	64	8kB	9x9
Branch Inter-arrival (Λ) (avg)	6.17	5.03	8.32	53.81	5.83	5.40
Branch Inter-arrival (Λ) (std)	5.66	6.95	6.69	103.09	5.42	5.51
Execution Time (cycles)	37530	11031	21148	18566	34616	78221
IPC (avg)	1.13	1.07	0.83	1.30	1.02	1.35
IPC (std)	1.0	0.71	0.85	0.91	0.39	0.95
Taken Branches	3913	1391	1229	199	3791	7194
Not-Taken Branches	1751	497	727	78	1092	4420
Packet Size (B_{pkt}) (avg/std)	1.2/0.4	2.4/1.3	1.3/0.5	1.3/0.7	1.0/0.2	1.9/1.3

An increase in speed was observed from all benchmarks between the regular and compiler-optimized binaries. However, the performance gains were primarily due to fewer taken branches, with IPC varying very little between each version. The average number of instructions per cycle was reduced in many cases, such as FFT, Quicksort, and LZO algorithms. Average trace packet sizes remained very similar, around 1.5, which corresponds to branches to very close

code regions, as only the trace only produces the difference in bits between the current and last branch target addresses. For non-optimized code, the average inter-arrival roughly corresponded to the type of implemented algorithms, with more significant arrival times for more data-bound applications. Also, after optimization, a reduction in the average inter-arrival time was seen. Interestingly it was also noted that almost all algorithms had the same average of about five cycles between branches with full optimizations. The only exception was the AES algorithm, which more than doubled its time between branches, which could be explained by loop-unrolling directives considering the drastic reduction in branch operation between both compilation configurations.

Each of the 12 benchmarks was tested under the two fault injection setups, static and runtime. All use-cases were analyzed unprotected and under CFTC monitoring protection. The frequency of the CPU was also scaled to verify the effects of different frequency factors (f). During testing, the monitor was run at 100 MHz. For the processor, three frequencies were chosen from the highest configuration to the smallest, 600 MHz, 300 MHz, and 100 MHz. This corresponds to factors of $f = 6$, $f = 3$, and $f = 1$ respectively. Ten thousand (10000) random faults were generated for each use case, injecting the same faults in each execution configuration. Only one fault is evaluated during an application run for 360000 injections.

The static injection results are shown in Tables 7.7 and 7.8. CFTC was able to detect 100% of CFEs that caused SDCs, Hangs, and Timeouts, while 74% of Masked faults were also detected. Masked faults do not necessarily mean that no data was corrupted, only that the execution reached the correct ending point. A branch could have jumped to a nearby region, re-executing or skipping some instructions. At the same time, they also correspond to modifications of opcodes that do not change the execution. With the use of CFTC, a significant decrease in the relative number of Timeouts was identified. One crucial aspect observed was the reduction of system timeouts from $\approx 7\%$ to $\approx 2\%$.

The first results from dynamic fault injection campaigns relating to this work showed a strong correlation between register usage and fault location. These results were published in the 14th International Symposium on Applied Reconfigurable Computing (ARC) [1]. Without any optimization, the compiler will choose the most straightforward method for memory access and data handling. It will use R3 and FP to read from and write values to the stack, avoiding other registers' use to keep intermediary values. Figure 7.20 shows the error rates

for each register from our injection campaigns in our non-optimized injection setup. Figure 7.20(a) shows that 70-80% of injections on the R3 register caused SDCs, while Hangs were proportionally distributed between R11/FP and PC. These distributions validate what is expected, given the functionalities associated with each register. R3 keeps the operational data while frame pointer (FP) stores the addresses used to access application memory.

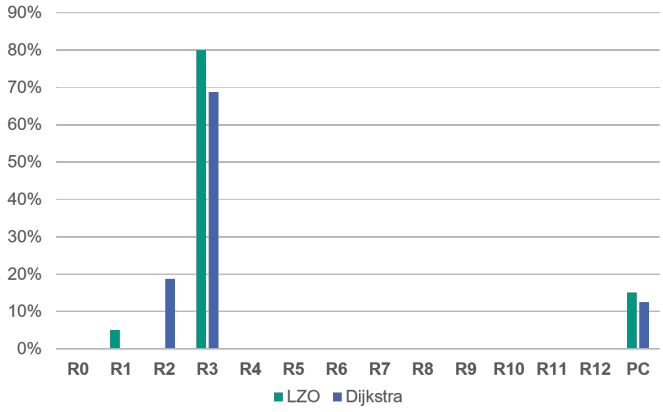
During the first experiments, homogeneous injections across all architectural registers were performed, Fig. 7.21(a). However, a strong dependence of optimization options on the use of registers was seen, Fig. 7.21(b). Except for the PC, without optimizations (O0), the compiler favors the use of R0-R3. The register usage in both applications was tightly coupled to the ABI and calling conventions used by the compiler. According to the "ARM Architecture Procedure Calling Standard (AAPCS)" [151], registers R0 to R3 are caller-save registers used to hold temporary values that need not be preserved across calls.

Table 7.7: Fault injection results per benchmark.

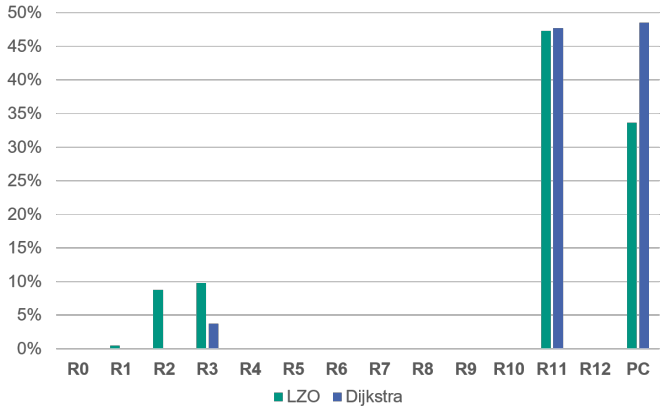
Benchmark	*Masked*	*SDC*	*Recovered*	*Unrecovered*	*Total Injections*
Dijkstra	2676	6527	20031	766	60000
FFT	2749	6550	20256	445	60000
Quicksort	2522	6768	20274	436	60000
AES	2251	6905	20029	815	60000
LZO	2436	6442	20283	839	60000
Matrix M.	2016	6643	20724	617	60000

Table 7.8: Detected errors and total identified fault injection results.

FI Result	*Injections (% of Total)*	*Detected (%)*
Masked	27447 (7.6%)	20360 (74.2%)
SDC	79865 (22.2%)	79865 (100%)
Recovered	244809 (68%)	2448098 (100%)
Unrecovered	7879 (2.2%)	7879 (100%)
Total	360000 (100%)	3529132 (98%)



(a) SDCs



(b) Hangs

Figure 7.20: Rate of SDCs and Hangs per targeted register.

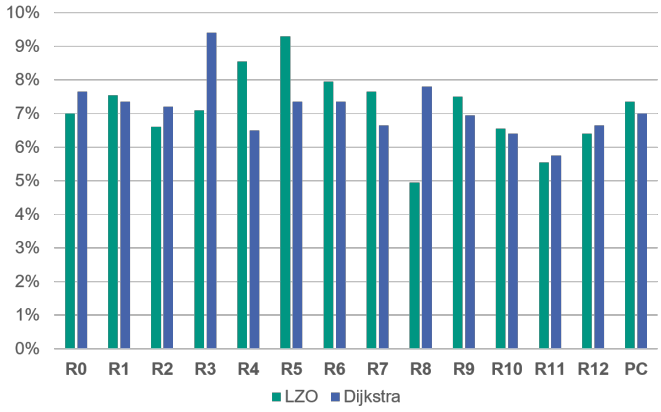
While R0-R2 are the preferred registers to hold arguments through function calls, R3 is the preferred working register. The standard also states that while not enforced, R11 is set as the default FP. The FP is responsible for pointing to the section of the stack allocated to the currently executing function.

The MMU page bits caused Timeout failures instead of generating any measurable latency for both applications. The memory pages outside of the current one used by the program never generated the execution of incorrect instructions. This effect is due to the faults leading the program to jump to empty memory regions, whose "instructions" are evaluated as `nops`. In these first published tests, PC errors were more strongly associated with Timeouts than hardware aborts, with all jumps to different memory sections generating Timeouts. Such errors can be detected by re-configuring the MMU; however, many errors generated by jumps inside the same memory section would still be seen and would not be detected. It is worth noting that all PC-related Aborts were faults in bits that left it still pointing to the program's memory section. Therefore, implementing a CFTC module can quickly detect problems and return the system to a functioning state.

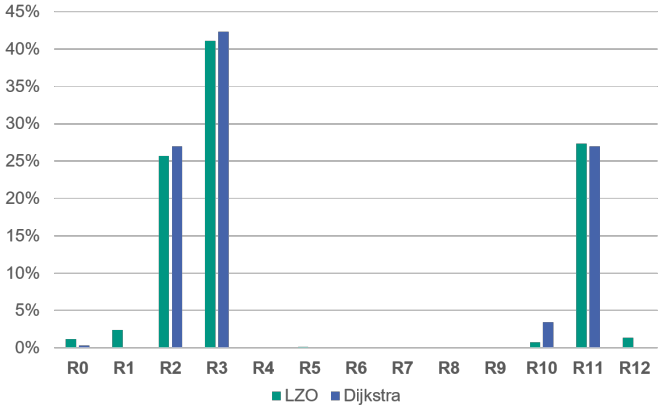
Figure 7.22 presents the measured error response times (L_{resp}) for all benchmarks for a low buffer utilization ($\rho < 1$). One drawback CFTC presents is waiting for trace packets to be processed. For $\rho > 1$, much higher response times were observed, in some cases as high as 1 Mcycles. Most of these higher utilization rates were directly related to higher frequency factors, reducing the effective packet service rate of the monitor. If buffers are empty, the trace is quickly processed, and the interruption is sent, but if they are full, all previous packets must be processed, corresponding to a high waiting time W.

The timer responsible for triggering and initiating the fault injection was configured to send a random interruption in times multiple of 5 kcycles. Faults were injected in cycles spaced in such a way to equally evaluate the buffer fills at different execution points of the code. It allows better observation of the evolution of the programs. Appendix A presents each individual graph in more detail. Figures A.1 to A.6 show the evolution in time of the optimized benchmarks for $f = 6$. While Figs. A.7 to A.12 and Figs. A.13 to A.18 show the evolution of the same benchmarks for $f = 3$ and $f = 1$, respectively.

Each graph represents the instantaneous buffer utilization over time (left Y-axis) and the packets' instantaneous waiting time evolution (right Y-axis). Both graphs correspond to the normal execution of each benchmark without any fault effects or control-flow errors present. Overlapped with these graphs are the error response time points collected for every one of our fault injection runs (right). The X axis for the error response times corresponds to the point during the execution when the fault injection interruption happened. The 5 kcycles



(a) Injection rates per register



(b) Register usage rates for *o0*.

Figure 7.21: Register injection rates (a) and usage rates (b)

spacing between the collection of points correspond to the timer only injecting a fault in multiples of elapsed 5 thousand cycles. In each of the graphs, the error response time almost directly takes the form of Eq. (7.11).

$$L_{resp} = L_{det} + L_{ISR} \cong W + L_{ISR} = (\rho - 1)t + L_{ISR} \tag{7.11}$$

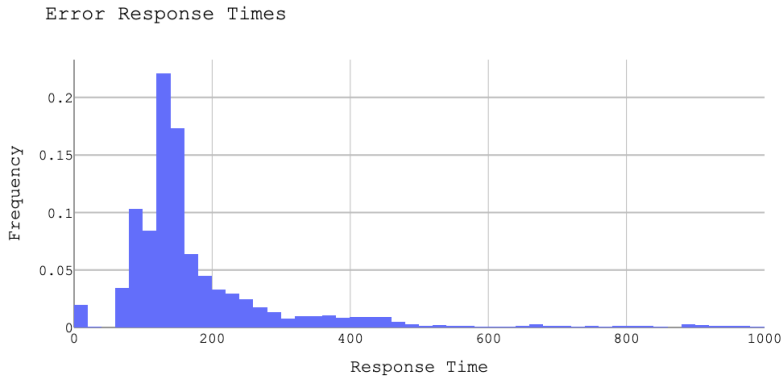


Figure 7.22: Error response times for low utilization tests ($\rho < 1$).

To calculate the utilization shown in the graphs, we divided the service time necessary for the arriving block, i.e., $S = f \times P_{pft}$, by the expected inter-arrival time for every block

$$\rho = \frac{\alpha}{\mu} = \frac{S_{BB}}{\Lambda_{BB}}$$

To obtain the expected inter-arrival time (Λ_{BB}) for each block, the trace configured for time stamping of branches was again used. Once every block was profiled and an average execution time was obtained, the benchmarks were again profiled without timestamping packet payloads. The presence of time stamping increases the payload size significantly, and therefore, the service time for incoming packets increases their waiting time in the buffer queue. The evolution of the utilization was plotted as a moving average to obtain a curve, using a window of 100 values.

For large enough buffer fills, a packet's waiting time in the buffer dominates, with the evolution or derivative of this latency being given by $(\rho - 1)$. Otherwise, for small ρ, the dominant factor is the processor's interruption handling latency. Therefore, by properly evaluating the utilization rate of a program, either through profiling or an in-depth static analysis, the expected trace decoding times during the program's execution can be assessed. This estimation

gives us a defined and proper metric to work with, which can then be used in future real-time architectures.

7.5 Summary

This chapter showed how to model and identify the intrinsic timings of our proposed hardware monitor architecture. Here it was shown through different targeted fault injection campaigns how CFTC's monitoring components detect 98.03% of all injected faults and 100% of all CFEs. The chapter also analyzed the system under different load scenarios, through different data-bound and control-bound benchmarks, under specific optimizations and frequency scaling configurations. The proposed architecture is general and usable in any modern processor that provides a comprehensive program trace interface such as the ones described in Chapter 4. It was shown here how to characterize the whole of the trace system, from the black-box trace bus subsystem provided by the COTS processor to our monitor IP. The chapter also showed how even as an FPGA component, it requires minimal resource usage, even for a simple SoC such as the Zynq-7000.

This chapter presented the essential metrics to be aware of in implementing such monitors. Until now, no studies have shown the correlation of an application's control-flow execution, i.e., the rate at which branches are executed and the bounds for detecting errors on the incoming trace. As nothing is presupposed about the specific monitor architecture, the models should apply to any current trace-based hardware monitoring architecture, be they CFC error detectors or CFI security monitors against attacks. The adjustable variable required to minimize the need for a buffer was also precisely shown, i.e., the utilization factor ρ. Evidently, the faster the monitor runs, the better. However, it is shown here precisely how fast the monitor must operate to verify the correctness of an embedded program's execution. To reduce these delays, we must rely on design time trade-offs. Decreasing the CPU frequency or deactivating caches can make the application run slower, providing the monitor enough time to process each packet. More performance-driven FPGA architectures can also be used that better match the processor's speed or implement the CFC modules as dedicated ASIC components.

8 Summary, Conclusions and Outlook

As we depend on embedded devices throughout our world, we must develop more robust and quantifiable mechanisms to detect and protect them against failures. Even after years of studies and proposed architectures, control-flow errors are still a concern. Among the many different techniques proposed over the years, two main types have emerged: software-based and hardware-based. Hardware-based techniques are usually costly. They depend on modifications to the platform and SoC, which results in the need to build new devices from scratch. Nowadays, we call processors specifically built with such techniques as Radiation Hardened (RadHard) processors. These processors are extremely robust and reliable in critical and fault-prone environments. However, they tend to have a much lower performance than standard commercial off-the-shelf (COTS) processor.

The need to profit from the high-performance of multi-core COTS devices has led to the emergence of new software-based protection and fault-tolerance approaches. These fault detection and tolerance methods rely solely on modifications to the underlying code to achieve higher reliability. They usually apply code transformation to achieve a certain measure of protection to certain aspects of the application's data or control path. Nevertheless, it has been recently shown that software-based techniques can detect faults in certain code regions, increasing the processing unit's total vulnerability overall. The primary established problem is the increase in execution time and code area required by these techniques. The new transformed code, in effect, increased the total number of possible fault locations that generate errors.

Therefore, protection mechanisms should both avoid adding overheads to the running application, not to increase the AVF of CPU components, and be portable to different COTS architecture, to take advantage of their increased performance. A new type of hardware-based technique tries to bridge this gap.

Trace-based control-flow checking (CFC) relies on the presence of a dedicated debug & trace interface to monitor in real-time changes to CPU state. These changes can be followed with the help of the application's control-flow graph (CFG) representation. Any mismatches between CPU state and CFG imply an execution by the processor of an unexpected operation not originally codified. Using these interfaces, such external monitors do not need to modify the existing processor architecture, thus being implemented much more economically. The re-emergence of fast FPGA architectures with embedded application processors further allow such components to be dynamically implemented without fabricating custom ASIC devices.

While some techniques were proposed based on online program trace analysis, they assumed a specific trace interface used by their target platforms and did not expand on ways to port such monitors to other systems [66, 148]. With this in mind, this thesis proposes a novel trace-based technique with portability as the primary focus. The analysis executed here started in Chapter 4 where the trace architectures implemented in the most widespread COTS processors were explored. These trace systems' standard features were identified with the information they provide about the control-flow's current state.

Chapter 5 looked at the processor's binary application and how it correlates back to the generated trace packet information. It identified how the relevant data could be retrieved to generate the corresponding binary CFG. This information was then used as the basis of the checker and configuration memory architecture presented in Chapter 6. The CFTC architecture is separated into two stages, a translation stage that takes the architecture-specific trace packets sent by the SoC and transmits only the information required to build the CPU's state. This independent state representation is then used to perform the final error checking. The final monitor developed in this work was smaller and faster than previous techniques and was demonstrated to be portable to other systems as long as a suitable translation stage is implemented.

This method can reliably identify control-flow errors, as shown by the fault injection campaign results in this work and those of other similar techniques. A central point of struggle not explored by any trace-based methods thus far was the feasibility of using program tracing in real-time constrained scenarios. For this purpose, Chapter 7 develops a new execution model for binary applications able to detect bottlenecks in online trace generation and analysis. A fault-injection analysis was performed over different sets of benchmarks and

frequency scales. This study reliably demonstrated the limits of online tracing, e.g., the minimum performance required by any hardware trace monitor to achieve and maintain bounds on its detection latencies. Applications that run too fast to the monitor generate a higher trace bandwidth than processed. This distinction in times causes congestion of input buffers and increases the time to process later trace packets. The models presented here can reliably predict the trace congestion and possible increases in the latencies' final error detection.

The global economy is seeing enormous growth in "smart mobility" domains such as autonomous driving. The autonomous car sector could add as much as $7 trillion to the world economy alone by 2050, as published by Intel and the research company Strategy Analytics [152]. Over time, service and application revenue generated by mobility-as-a-service will supplant the value of vehicle sales as core sources of value. Inevitably, autonomous cars will have a massive effect on our individual driving experiences, with comfort, reliability, and safety all key factors. More importantly, autonomous cars are an essential consideration in how our cities and towns of the future will be designed and structured. Arguably, the future of our cities and communities is inherently linked to the future of automobiles. We need to be taking a positive and progressive approach toward this.

This thesis showed that trace-based error detection could bridge the gap between the need for software isolation given by hardware-based methods and portability across different systems given by software-based methods. However, there is no silver bullet, and if the application generates considerable trace bandwidth while the monitor lacks the required frequency or performance to process it, stalls can happen, and significant detection times are observed. Control-flow errors still represent a majority of different collected errors in embedded systems. The results presented here show the actual metrics to be observed if essential monitoring mechanisms are to be implemented in existing systems. Therefore, designers who wish to use such methods can now identify acceptable trace bandwidth constraints and how to achieve them. With this information in hand, they can decide which monitor implementation best suits such bandwidth and what latencies are expected to detect common CFEs.

List of Figures

List of Tables

A Response Time Analysis Graphs

This appendix presents the collected results from the response time analysis evaluations performed in Chapter 7.

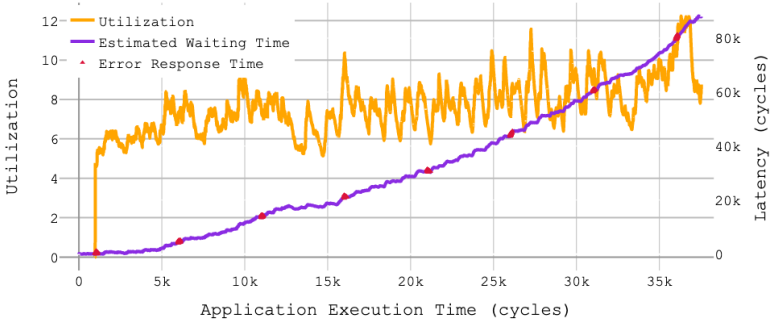


Figure A.1: Packet waiting time and error response time over the execution time of Dijkstra (-o3), $f = 6$.

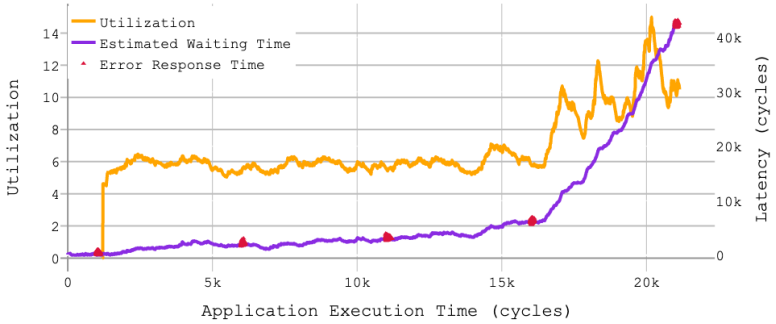


Figure A.2: Packet waiting time and error response time over the execution time of Quicksort (-o3), $f = 6$.

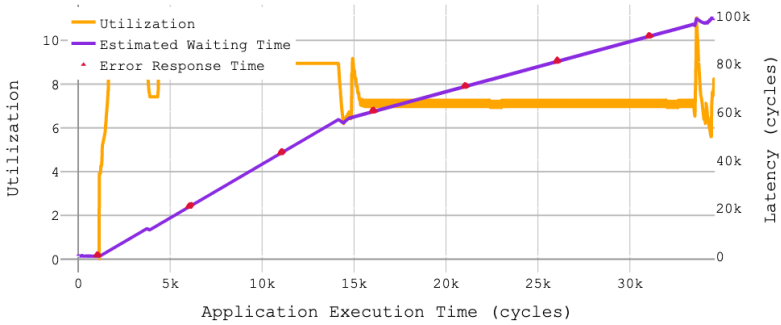


Figure A.3: Packet waiting time and error response time over the execution time of LZO (-o3), $f = 6$.

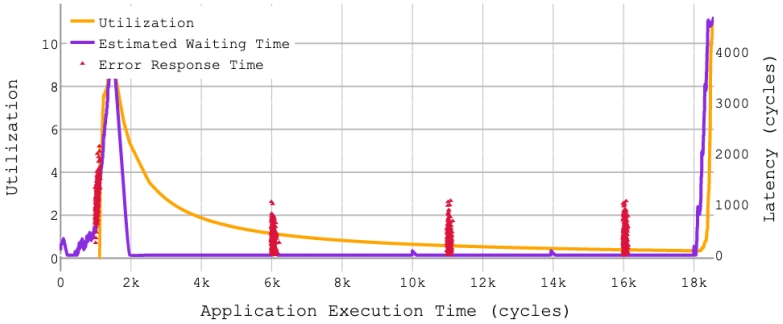


Figure A.4: Packet waiting time and error response time over the execution time of AES (-o3), $f = 6$.

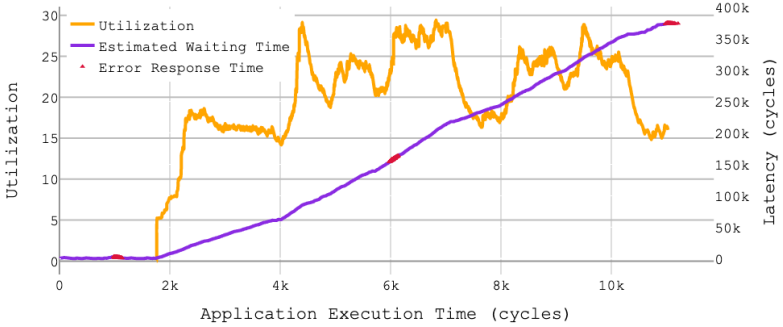


Figure A.5: Packet waiting time and error response time over the execution time of FFT (-o3), $f = 6$.

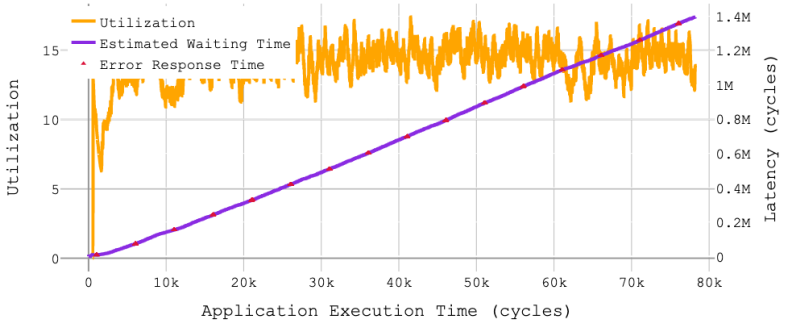


Figure A.6: Packet waiting time and error response time over the execution time of MM (-o3), $f = 6$.

Figure A.7: Packet waiting time and error response time over the execution time of Dijkstra (-o3), $f = 3$.

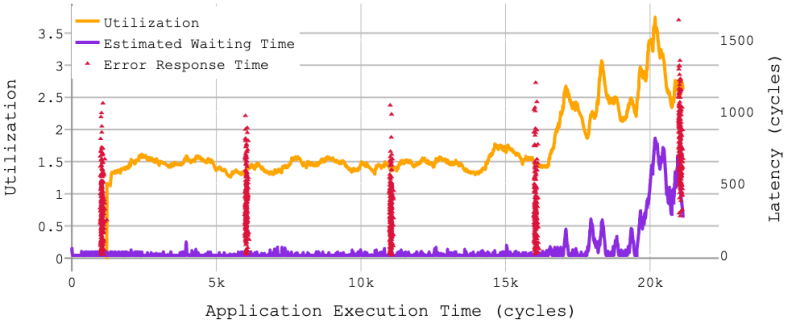


Figure A.8: Packet waiting time and error response time over the execution time of Quicksort (-o3), $f = 3$.

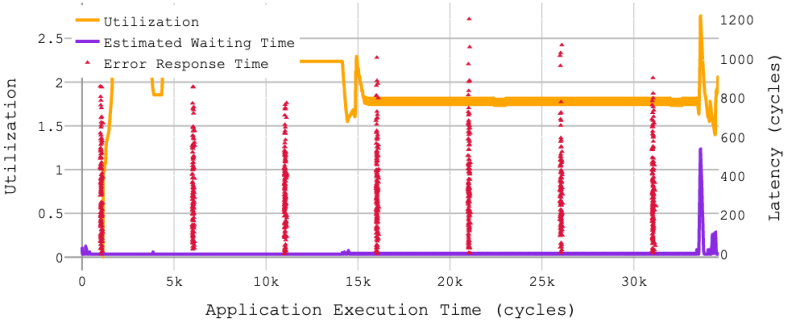


Figure A.9: Packet waiting time and error response time over the execution time of LZO (-o3), $f = 3$.

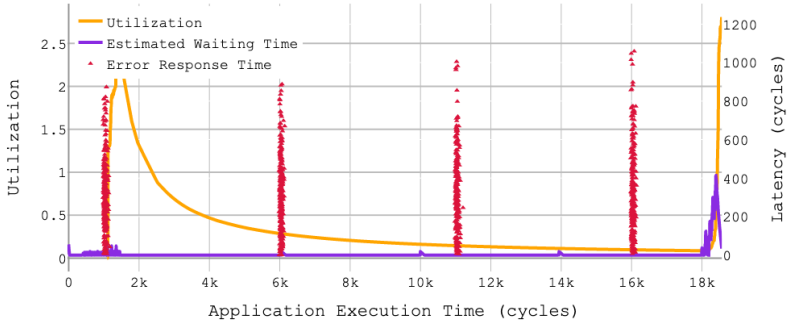


Figure A.10: Packet waiting time and error response time over the execution time of AES (-o3), $f = 3$.

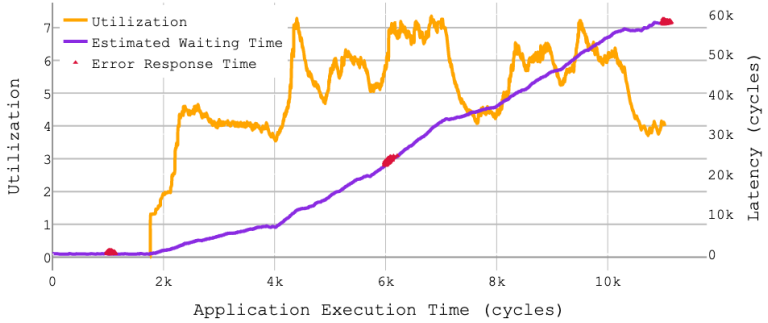


Figure A.11: Packet waiting time and error response time over the execution time of FFT (-o3), $f = 3$.

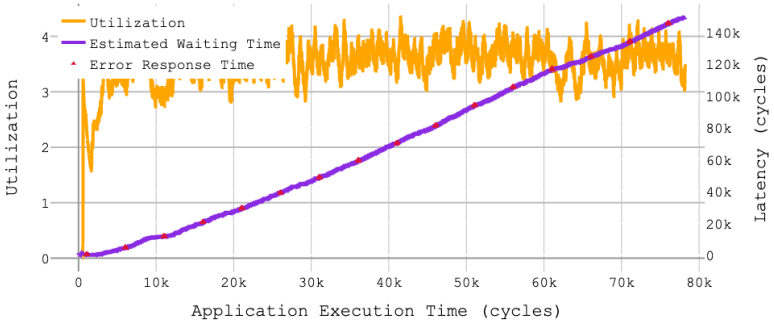


Figure A.12: Packet waiting time and error response time over the execution time of MM (-o3), $f = 3$.

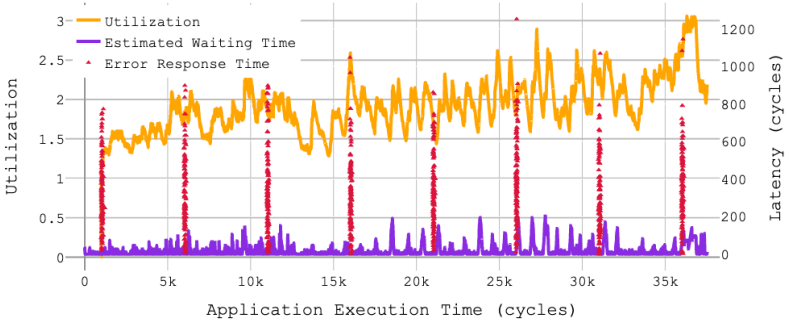


Figure A.13: Packet waiting time and error response time over the execution time of Dijkstra (-o3), $f = 3$.

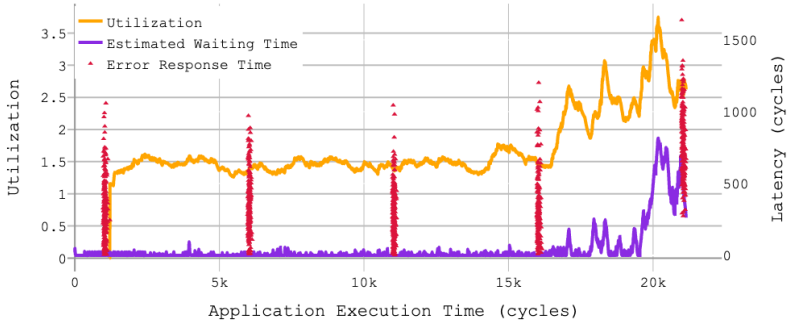


Figure A.14: Packet waiting time and error response time over the execution time of Quicksort (-o3), $f = 3$.

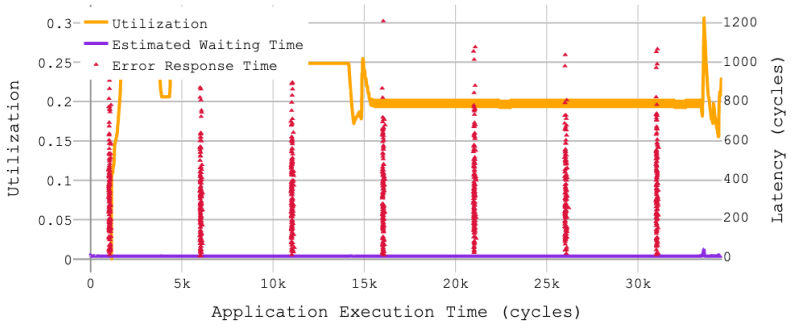


Figure A.15: Packet waiting time and error response time over the execution time of LZO (-o3), $f = 1$.

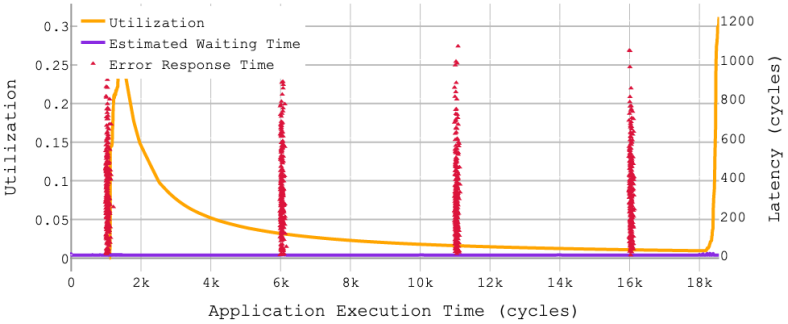


Figure A.16: Packet waiting time and error response time over the execution time of AES (-o3), $f = 1$.

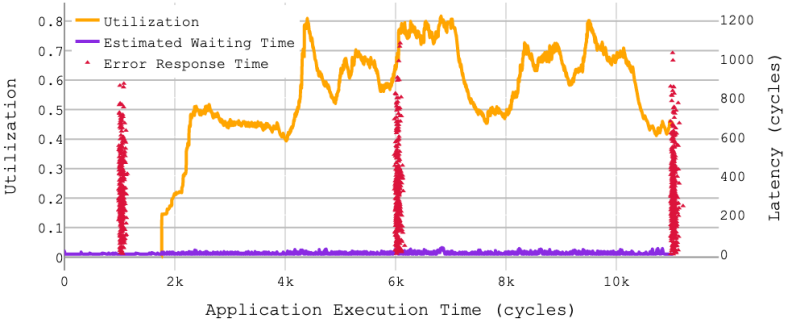


Figure A.17: Packet waiting time and error response time over the execution time of FFT (-o3), $f = 1$.

Figure A.18: Packet waiting time and error response time over the execution time of MM (-o3), $f = 1$.

Publications

[1] A. W. Hoppe, F. Kastensmidt, and J. Becker, “Control Flow Analysis for Embedded Multi-Core Hybrid Systems,” *Lecture Notes in Computer Science book series (LNCS)*, vol. 10824, pp. 485–496, 2018.

[2] A. Hoppe, F. Kastensmidt, and J. Becker, “Fine Grained Control Flow Checking with Dedicated FPGA Monitors,” in *2020 33rd IEEE International System-on-Chip Conference (SOCC)*, vol. 10824, 2020, pp. 485–496.

[3] A. W. Hoppe, F. L. Kastensmidt, and J. Becker, “High-speed Hardware Accelerator for Trace Decoding in Real-Time Program Monitoring,” in *2021 IEEE 12th Latin American Symposium on Circuits and Systems (LASCAS)*, vol. 10824, 2021, pp. 485–496.

[4] A. Hoppe, F. L. Kastensmidt, and J. Becker, “Investigating real-time control-flow error detection in hardware: How fast can we detect errors and take action?” *Microelectronics Reliability*, vol. 126, p. 114264, 2021.

Bibliography

[5] J. Ohlsson, M. Rimen, and U. Gunneflo, "A study of the effects of transient fault injection into a 32-bit RISC with built-in watchdog," in *[1992] Digest of Papers. FTCS-22: The Twenty-Second International Symposium on Fault-Tolerant Computing*. IEEE, 1992, pp. 316–325.

[6] E. Chielle, G. S. Rodrigues, F. Kastensmidt, S. Cuenca-Asensi, L. A. Tambara, P. Rech, and H. Quinn, "S-SETA: Selective Software-Only Error-Detection Technique Using Assertions," *IEEE Transactions on Nuclear Science*, vol. 62, no. 6, pp. 3088–3095, 2015.

[7] R. Vemu and J. Abraham, "CEDA: Control-flow error detection using assertions," *IEEE Transactions on Computers*, vol. 60, no. 9, pp. 1233–1245, 2011.

[8] J. R. Azambuja, M. Altieri, J. Becker, and F. L. Kastensmidt, "HETA: Hybrid error-detection technique using assertions," *IEEE Transactions on Nuclear Science*, vol. 60, no. 4, pp. 2805–2812, 2013.

[9] S. Schuster, P. Ulbrich, I. Stilkerich, C. Dietrich, and W. SchröDer-Preikschat, "Demystifying Soft-Error Mitigation by Control-Flow Checking – A New Perspective on its Effectiveness," *ACM Transactions on Embedded Computing Systems*, vol. 16, no. 5s, pp. 1–19, 2017.

[10] A. Rhisheekesan, R. Jeyapaul, and A. Shrivastava, "Control Flow Checking or Not ? (for Soft Errors)," *ACM Transactions on Embedded Computing Systems*, vol. 18, no. 1, p. 25, 2019.

[11] N. S. Kim, T. Austin, D. Blaauw, T. Mudge, K. Flautner, J. S. Hu, M. J. Irwin, M. Kandemir, and V. Narayanan, "Leakage current: Moore's law meets static power," *Computer*, vol. 36, no. 12, pp. 68–75, dec 2003.

[12] S. Thompson, R. Chau, T. Ghani, K. Mistry, S. Tyagi, and M. Bohr, "In Search of "Forever," Continued Transistor Scaling One New Material at a Time," *IEEE Transactions on Semiconductor Manufacturing*, vol. 18, no. 1, pp. 26–36, feb 2005.

[13] R. Baumann, “Soft Errors in Advanced Computer Systems,” *IEEE Design and Test of Computers*, vol. 22, no. 3, pp. 258–266, may 2005.

[14] TSMC, “TSMC: 5nm Technology,” 2021. [Online]. Available: https://www.tsmc.com/english/dedicatedFoundry/technology/logic/l_5nm

[15] “Technology Quarterly after Moore’s Law - Double, double, toil and trouble.” The Economist, 2016. [Online]. Available: https://www.economist.com/technology-quarterly/2016-03-12/after-moores-law

[16] G. M. Amdahl, “Validity of the single processor approach to achieving large scale computing capabilities,” in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, ser. AFIPS ’67 (Spring). New York, NY, USA: Association for Computing Machinery, 1967, p. 483–485.

[17] J. L. Gustafson, “Reevaluating amdahl’s law,” *Commun. ACM*, vol. 31, no. 5, p. 532–533, May 1988.

[18] P. Leteinturier, S. Brewerton, and K. Scheibert, “MultiCore benefits & challenges for automotive applications,” in *SAE Technical Paper Series*. SAE International, Apr 2008.

[19] “A model of soft error effects in generic IP processors,” in *20th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT’05)*. IEEE Comput. Soc, 2005, pp. 334–342.

[20] J. R. Azambuja, S. Pagliarini, L. Rosa, and F. L. Kastensmidt, “Exploring the limitations of software-based techniques in SEE fault coverage,” *Journal of Electronic Testing*, vol. 27, no. 4, pp. 541–550, Apr 2011.

[21] T. Rains, M. Miller, and D. Weston, “Exploitation Trends: From Potential Risk to Actual Risk,” in *Proceedings of the RSA Conference 2015*, San Francisco, California, USA, 2015.

[22] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, “StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks,” in *Proceedings of the 7th USENIX Security Symposium*, 1998.

[23] M. Prasad and T.-c. Chiueh, “A Binary Rewriting Defense against Stack based Buffer Overflow Attacks,” in *Proceedings of the USENIX Annual Technical Conference*, 2003, pp. 211–224.

[24] C. Cowan, S. Beattie, J. Johansen, and P. Wagle, “PointGuard: Protecting pointers from buffer overflow vulnerabilities.” in *Proceedings*

of the USENIX Security Symposium. USENIX, Berkeley, CA, 2003, pp. 91–104.

[25] X. Iturbe, B. Venu, and E. Ozer, "Soft error vulnerability assessment of the real-time safety-related ARM Cortex-R5 CPU," in *2016 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. IEEE, sep 2016, pp. 91–96.

[26] J. Maiz, S. Hareland, K. Zhang, and P. Armstrong, "Characterization of multi-bit soft error events in advanced SRAMs," in *IEEE International Electron Devices Meeting 2003*. IEEE, 2004, pp. 21.4.1–21.4.4.

[27] X. Li, K. Shen, and M. Huang, "A memory soft error measurement on production systems," *2007 USENIX Annual Technical*, pp. 275–280, 2007.

[28] V. Sridharan, N. DeBardeleben, S. Blanchard, K. B. Ferreira, J. Stearley, J. Shalf, and S. Gurumurthi, "Memory Errors in Modern Systems: The Good, The Bad, and The Ugly," *ACM SIGARCH Computer Architecture News*, vol. 43, no. 1, pp. 297–310, mar 2015.

[29] N. Seifert, S. Jahinuzzaman, J. Velamala, R. Ascazubi, N. Patel, B. Gill, J. Basile, and J. Hicks, "Soft Error Rate Improvements in 14-nm Technology Featuring Second-Generation 3D Tri-Gate Transistors," *IEEE Transactions on Nuclear Science*, vol. 62, no. 6, pp. 2570–2577, Dec. 2015.

[30] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, jan 2004.

[31] I. Koren and M. C. Krishna, *Fault-tolerant systems*. San Francisco, CA, USA: Elsevier/Morgan Kaufmann, 2007.

[32] *Zynq-7000 All Programmable SoC - Technical Reference Manual*, Xilinx, Oct 2017, version 1.12.

[33] W. J. Dally, J. Balfour, D. Black-Shaffer, J. Chen, R. C. Harting, V. Parikh, J. Park, and D. Sheffield, "Efficient embedded computing," *Computer*, vol. 41, no. 7, pp. 27–32, 2008.

[34] V. W. Lee, P. Hammarlund, R. Singhal, P. Dubey, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, and S. Chennupaty, "Debunking the 100X GPU vs. CPU myth," in *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3, 2012, p. 451.

[35] I. Kuon and J. Rose, "Measuring the gap between FPGAs and ASICs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 203–215, 2007.

[36] *ARM Architecture Reference Manual – ARMv7-A and ARMv7-R edition*, ARM, May 2014.

[37] A. Holdings, "ARM Limited Roadshow Slides Q2 2020," 2018. [Online]. Available: https://group.softbank/system/files/pdf/ir/presentations/2020/arm-roadshow-slides_q2fy2020_01_en.pdf

[38] *ARM Cortex-A Series – Programmer's Guide*, ARM, Jan 2014, version: 4.0.

[39] *Cortex-A9 – Technical Reference Manual*, ARM, Jul 2011, revision r3p0.

[40] M. Manoochehri, M. Annavaram, and M. Dubois, "CPPC," in *Proceeding of the 38th annual international symposium on Computer architecture - ISCA '11*. ACM Press, 2011. [Online]. Available: https://doi.org/10.1145/2000064.2000091

[41] M. A. Rouf and Soontae Kim, "Low-Cost Control Flow Protection via Available Redundancies in the Microprocessor Pipeline," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 23, no. 1, pp. 131–141, jan 2015.

[42] "ISO 26262:2018 – Road Vehicles – Functional Safety," Geneva, CH, Standard, 2018.

[43] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor," in *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, 2003, pp. 29–40.

[44] S. Raasch, A. Biswas, J. Stephan, P. Racunas, and J. Emer, "A fast and accurate analytical technique to compute the AVF of sequential bits in a processor," in *Proceedings of the 48th International Symposium on Microarchitecture - MICRO-48*. New York, New York, USA: ACM Press, 2015, pp. 738–749.

[45] A. Biswas, N. Soundararajan, S. S. Mukherjee, and S. Gurumurthi, "Quantized AVF : A Means of Capturing Vulnerability Variations over Small Windows of Time," in *In IEEE Workshop on Silicon Errors in Logic - System Effects*, 2009.

[46] X. Li, S. V. Adve, P. Bose, and J. A. Rivers, "Online Estimation of Architectural Vulnerability Factor for Soft Errors," in *2008 International Symposium on Computer Architecture*. IEEE, jun 2008, pp. 341–352.

[47] J. D. C. Little, "A Proof for the Queuing Formula: $L = \lambda W$," *Operations Research*, vol. 9, no. 3, pp. 383–387, Jun 1961.

[48] N. J. Wang, A. Mahesri, and S. J. Patel, "Examining ACE analysis reliability estimates using fault-injection," in *ISCA '07 Proceedings of the 34th annual international symposium on Computer architecture*, 2007, pp. 460–469.

[49] A. Biswas, P. Racunas, J. Emer, and S. S. Mukherjee, "Computing Accurate AVFs using ACE Analysis on Performance Models : a Rebuttal," *IEEE Computer Architecture Letters*, vol. 7, no. 1, pp. 21–24, 2008.

[50] "IEC 61508:2010 Functional safety of electrical/electronic/programmable electronic safety- related systems," Standard, 2010.

[51] "DO-178C: Software Consideration in Airborne Systems and Equipment Certification," Washington, D.C., USA, Standard, 2011.

[52] "DO-254: Design Assurance Guidance For Airborne Electronic Hardware," Washington, D.C., USA, Standard, 2000.

[53] V. Lefftz, J. Bertrand, H. Casse, C. Clienti, P. Coussy, L. Maillet-Contoz, P. Mercier, P. Moreau, L. Pierre, and E. Vaumorin, "A design flow for critical embedded systems," in *International Symposium on Industrial Embedded System (SIES)*. IEEE, jul 2010, pp. 229–233.

[54] R. Berger, L. Burcin, D. Hutcheson, J. Koehler, M. Lassa, M. Milliser, D. Moser, D. Stanley, R. Zeger, B. Blalock, and M. Hale, "The RAD6000MC System-on-Chip Microcontroller for Spacecraft Avionics and Instrument Control," in *2008 IEEE Aerospace Conference*. IEEE, mar 2008, pp. 1–14.

[55] L. Gagea and I. Rajkovic, "Designing devices for avionics applications and the DO-254 guideline," in *2008 International Semiconductor Conference*, vol. 2. IEEE, 2008, pp. 377–380.

[56] L. R. Eveleens, "Integrated Modular Avionics Development Guidances and Certification Considerations," in *Mission Systems Engineering 2*, Nov 2006.

[57] H. Shacham, "The geometry of innocent flesh on the bone," in *Proceedings of the 14th ACM conference on Computer and communications*

security - CCS '07. New York, New York, USA: ACM Press, 2007, p. 552.

[58] N. Carlini and D. Wagner, "Rop is still dangerous: Breaking modern defenses," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, ser. SEC'14. USA: USENIX Association, 2014, p. 385–399.

[59] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM Conference on Computer and Communications Security*, ser. CCS '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 340–353. [Online]. Available: https://doi.org/10.1145/1102120.1102165

[60] "Fine-Grained Control-Flow Integrity for Kernel Software," in *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, mar 2016, pp. 179–194.

[61] N. Carlini, A. Barresi, E. T. H. Zürich, M. Payer, D. Wagner, T. R. Gross, E. T. H. Zürich, N. Carlini, A. Barresi, D. Wagner, and T. R. Gross, "Control-Flow Bending : On the Effectiveness of Control-Flow Integrity This paper is included in the Proceedings of the," in *USENIX Security Symposium*, 2015, pp. 161–176. [Online]. Available: http://dblp.uni-trier.de/db/conf/uss/uss2015.html{#}CarliniBPWG15

[62] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Transactions on Information and System Security*, vol. 13, no. 1, pp. 1–40, 2009.

[63] "Web of Science Core Collection," Clarivate, 2015, Accessed: 2020-12-13. [Online]. Available: https://www.webofknowledge.com/

[64] R. Baumann, "The impact of technology scaling on soft error rate performance and limits to the efficacy of error correction," in *Digest. International Electron Devices Meeting*. IEEE, 2003, pp. 329–332.

[65] A. Rajabzadeh and S. G. Miremadi, "CFCET: A hardware-based control flow checking technique in COTS processors using execution tracing," *Microelectronics Reliability*, vol. 46, no. 5-6, pp. 959–972, 2006.

[66] B. Du, M. Sonza Reorda, L. Sterpone, L. Parra, M. Portela-García, A. Lindoso, and L. Entrena, "Online Test of Control Flow Errors: A New Debug Interface-Based Approach," *IEEE Transactions on Computers*, vol. 65, no. 6, pp. 1846–1855, 2016.

[67] M. Pena-Fernandez, A. Lindoso, L. Entrena, M. Garcia-Valderas, Y. Morilla, and P. Martin-Holgado, "Online Error Detection Through Trace Infrastructure in ARM Microprocessors," *IEEE Transactions on Nuclear Science*, vol. 66, no. 7, pp. 1457–1464, jul 2019.

[68] M. Fazeli, R. Farivar, and S. G. Miremadi, "A software-based concurrent error detection technique for powerPC processor-based embedded systems," in *Proceedings - IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, 2005.

[69] R. G. Ragel and S. Parameswaran, "A hybrid hardware–software technique to improve reliability in embedded processors," *ACM Transactions on Embedded Computing Systems*, vol. 10, no. 3, pp. 1–16, 2011.

[70] R. Nathan and D. J. Sorin, "Nostradamus: Low-cost hardware-only error detection for processor cores," in *Proceedings -Design, Automation and Test in Europe, DATE*. EDAA, 2014, pp. 1–6.

[71] S. Bergaoui, P. Vanhauwaert, and R. Leveugle, "IDSM: An improved disjoint signature monitoring scheme for processor behavioral checking," *LATW 2014 - 15th IEEE Latin-American Test Workshop*, vol. 6, pp. 5–10, 2014.

[72] B. Du, M. Sonza Reorda, L. Sterpone, L. Parra, M. Portela-Garcia, A. Lindoso, and L. Entrena, "Exploiting the debug interface to support on-line test of control flow errors," in *2013 IEEE 19th International On-Line Testing Symposium (IOLTS)*. IEEE, jul 2013, pp. 98–103.

[73] B. Du, M. S. Reorda, L. Sterpone, L. Parra, M. Portela-Garcia, A. Lindoso, and L. Entrena, "A new solution to on-line detection of Control Flow Errors," in *2014 IEEE 20th International On-Line Testing Symposium (IOLTS)*, vol. 675. IEEE, jul 2014, pp. 105–110.

[74] M. Peña-Fernandez, A. Lindoso, L. Entrena, M. Garcia-Valderas, S. Philippe, Y. Morilla, and P. Martin-Holgado, "PTM-based hybrid error-detection architecture for ARM microprocessors," *Microelectronics Reliability*, vol. 88-90, no. May, pp. 925–930, sep 2018.

[75] OpenCores, "miniMIPS – Overview," 2004, available at https://opencores.org/projects/minimips, accessed 26th Apr. 2019.

[76] *GRLIB IP Core User's Manual*, Gaisler, Dec 2017.

[77] F. L. Kastensmidt, J. Tonfat, T. Both, P. Rech, G. Wirth, R. Reis, F. Bruguier, P. Benoit, L. Torres, and C. Frost, "Aging and voltage

scaling impacts under neutron-induced soft error rate in SRAM-based FPGAs," in *2014 19th IEEE European Test Symposium (ETS)*. IEEE, may 2014, pp. 1–2.

[78] D. J. Lu, "Watchdog Processors and Structural Integrity Checking," *IEEE Transactions on Computers*, vol. C, no. 7, pp. 681–685, 1982.

[79] T. Michel, R. Leveugle, and G. Saucier, "A new approach to control flow checking without program modification," in *[1991] Digest of Papers. Fault-Tolerant Computing: The Twenty-First International Symposium*. IEEE, 1991.

[80] J. R. Azambuja, C. Pilotto, and F. L. Kastensmidt, "Mitigating soft errors in SRAM-based FPGAs by using large grain TMR with selective partial reconfiguration," in *Proceedings of the European Conference on Radiation and its Effects on Components and Systems, RADECS*. IEEE, sep 2008, pp. 288–293.

[81] P. Ezhilchelvan, I. Mitrani, and S. Shrivastava, "A performance evaluation study of pipeline TMR systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, no. 4, pp. 442–456, 1990.

[82] A. S. Keys, J. H. Adams, J. D. Cressler, R. C. Darty, M. A. Johnson, M. C. Patrick, and M. S. El-Genk, "High-Performance, Radiation-Hardened Electronics for Space and Lunar Environments," in *AIP Conference Proceedings*, vol. 969. AIP, 2008, pp. 749–756.

[83] T. M. Lovelly and A. D. George, "Comparative Analysis of Present and Future Space-Grade Processors with Device Metrics," *Journal of Aerospace Information Systems*, vol. 14, no. 3, pp. 184–197, 2017.

[84] A. Mahmood and E. McCluskey, "Concurrent error detection using watchdog processors-a survey," *IEEE Transactions on Computers*, vol. 37, no. 2, pp. 160–174, 1988.

[85] T. Austin, "DIVA: a reliable substrate for deep submicron microarchitecture design," in *MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE Comput. Soc, 2003, pp. 196–207.

[86] O. Goloubeva, M. Rebaudengo, M. S. Reorda, and M. Violante, *Software-Implemented Hardware Fault Tolerance*, 1st ed. Springer US, 2006.

[87] F. E. Allen, “Control flow analysis,” in *Proceedings of a symposium on Compiler optimization -*. New York, New York, USA: ACM Press, 1970, pp. 1–19.

[88] G. Kanawati, V. Nair, N. Krishnamurthy, and J. Abraham, “Evaluation of integrated system-level checks for on-line error detection,” in *Proceedings of IEEE International Computer Performance and Dependability Symposium*. IEEE Comput. Soc. Press, 1996, pp. 292–301.

[89] Z. Alkhalifa, V. S. Nair, N. Krishnamurthy, and J. A. Abraham, “Design and evaluation of system-level checks for on-line control flow error detection,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 6, pp. 627–641, 1999.

[90] N. Oh, P. P. Shirvani, and E. J. McCluskey, “Control-flow checking by software signatures,” *IEEE Transactions on Reliability*, vol. 51, no. 1, pp. 111–122, 2002.

[91] R. Venkatasubramanian, J. P. Hayes, and B. T. Murray, “Low-cost on-line fault detection using control flow assertions,” in *Proceedings - 9th IEEE International On-Line Testing Symposium, IOLTS 2003*, 2003, pp. 137–143.

[92] O. Goloubeva, M. Rebaudengo, M. S. Reorda, and M. Violante, “Soft-error detection using control flow assertions,” in *Proceedings - IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, vol. 2003-Janua, 2003, pp. 581–588.

[93] H. Schirmeier, C. Borchert, and O. Spinczyk, “Avoiding Pitfalls in Fault-Injection Based Comparison of Program Susceptibility to Soft Errors,” in *Proceedings of the International Conference on Dependable Systems and Networks*, vol. 2015-Septe, 2015, pp. 319–330.

[94] OpenCores, “Amber ARM-compatible core – Overview,” 2010, available at https://opencores.org/projects/amber, accessed 29th Apr. 2019.

[95] G. Rodrigues, F. ROSA, A. de Oliveira, F. Lima Kastensmidt, L. Ost, and R. Reis, “Analyzing the Impact of Fault Tolerance Methods in ARM Processors under Soft Errors running Linux and Parallelization APIs,” *IEEE Transactions on Nuclear Science*, vol. 64, no. 8, pp. 1–1, 2017.

[96] T. Santini, L. Carro, F. R. Wagner, and P. Rech, “Reliability analysis of operating systems for embedded SoC,” *Proceedings of the European Conference on Radiation and its Effects on Components and Systems, RADECS*, vol. 2015-Decem, pp. 2–6, 2015.

[97] S. Cuenca-Asensi, A. Martinez-Alvarez, F. Restrepo-Calle, F. R. Palomo, H. Guzman-Miranda, and M. A. Aguirre, "A Novel Co-Design Approach for Soft Errors Mitigation in Embedded Systems," *IEEE Transactions on Nuclear Science*, vol. 58, no. 3, pp. 1059–1065, jun 2011.

[98] M. A. Schuette and J. P. Shen, "Processor Control Flow Monitoring Using Signatured Instruction Streams," *IEEE Transactions on Computers*, vol. C-36, no. 3, pp. 264–276, 1987.

[99] J. B. Eifert and J. P. Shen, "Processor Monitoring Using Asynchronous Signatures Instruction Streams," in *Proceedings of FTCS-25*, vol. III. IEEE, 1995, pp. 394–399.

[100] K. Wilken and J. P. Shen, "Continuous Signature Monitoring: Efficient Concurrent-detection of Processor Control Errors," in *Proceedings of the 1988 International Conference on Test: New Frontiers in Testing*. IEEE, 1988, pp. 914–925.

[101] N. R. Saxena and E. J. Mccluskey, "Control-flow checking using watchdog assists and extended-precision checksums," *IEEE Transactions on Computers*, vol. 39, no. 4, pp. 554–559, 1990.

[102] G. Giaconia, A. Di Stefano, and G. Capponi, "FPGA-based concurrent watchdog for real-time control systems," *Electronics Letters*, vol. 39, no. 10, p. 769, 2003.

[103] S. A. Asghari, H. Taheri, H. Pedram, and O. Kaynak, "Software-based control flow checking against transient faults in industrial environments," *IEEE Transactions on Industrial Informatics*, vol. 10, no. 1, pp. 481–490, 2014.

[104] A. Rajabzadeh and S. G. Miremadi, "Transient detection in COTS processors using software approach," *Microelectronics Reliability*, vol. 46, no. 1, pp. 124–133, jan 2006.

[105] J. Freitag and S. Uhrig, "Dynamic interference quantification for multicore processors," *AIAA/IEEE Digital Avionics Systems Conference - Proceedings*, vol. 2017-Septe, 2017.

[106] F. Khosravi, H. Farbeh, M. Fazeli, and S. G. Miremadi, "Low cost concurrent error detection for on-chip memory based embedded processors," *Proceedings - 2011 IFIP 9th International Conference on Embedded and Ubiquitous Computing, EUC 2011*, pp. 114–119, 2011.

[107] H. Madeira and J. G. Silva, "On-line signature learning and checking: experimental evaluation," in *[1991] Proceedings, Advanced Computer Technology, Reliable Systems and Applications*. IEEE, 1991.

[108] N. J. Wang and S. J. Patel, "ReStore: Symptom-based soft error detection in microprocessors," *IEEE Transactions on Dependable and Secure Computing*, vol. 3, no. 3, pp. 188–201, 2006.

[109] N. Farazmand, M. Fazeli, and S. G. Miremadi, "FEDC: Control flow error detection and correction for embedded systems without program interruption," in *ARES 2008 - 3rd International Conference on Availability, Security, and Reliability, Proceedings*, 2008, pp. 33–38.

[110] A. B. T. Hopkins and K. D. McDonald-Maier, "Debug support for complex systems on-chip: a review," in *IEE Proceedings - Computers and Digital Techniques*, vol. 153, no. 4. IET, 2006, p. 197.

[111] A. Mayer, H. Siebert, and K. D. McDonald-Maier, "Boosting debugging support for complex systems on chip," *Computer*, vol. 40, no. 4, pp. 76–81, 2007.

[112] B. Vermeulen, R. Kühnis, J. Rearick, N. Stollon, and G. Swoboda, "Overview of debug standardization activities," *IEEE Design and Test of Computers*, vol. 25, no. 3, pp. 258–267, 2008.

[113] R. Backasch, C. Hochberger, A. Weiss, M. Leucker, and R. Lasslop, "Runtime verification for multicore SoC with high-quality trace data," *ACM Transactions on Design Automation of Electronic Systems*, vol. 18, no. 2, pp. 1–26, mar 2013.

[114] "IEEE-ISTO 5001-2012 the nexus 5001 forum – standard for a global embedded processor debug interface," Standard, Jun 2012, available at https://nexus5001.org/, accessed 18th Apr. 2019.

[115] *Zynq UltraScale+ Device - Technical Reference Manual*, Xilinx, Nov 2017, version 1.6.

[116] Intel Corporation, "Intel SoC FPGAs," 2019, accessed 18th Apr. 2019. [Online]. Available: https://www.intel.com/content/www/us/en/products/programmable/soc.html

[117] *ARM CoreSight Architecture Specification*, ARM, Feb 2017, version 3.0.

[118] N. Decker, P. Gottschling, C. Hochberger, M. Leucker, T. Scheffel, M. Schmitz, and A. Weiss, "Rapidly adjustable non-intrusive online

monitoring for multi-core systems," in *Formal Methods: Foundations and Applications*. Springer International Publishing, 2017, pp. 179–196.

[119] Y. Lee, J. Lee, I. Heo, D. Hwang, and Y. Paek, "Using CoreSight PTM to Integrate CRA Monitoring IPs in an ARM-Based SoC," *ACM Transactions on Design Automation of Electronic Systems*, vol. 22, no. 3, pp. 1–25, apr 2017.

[120] W. He, S. Das, W. Zhang, and Y. Liu, "BBB-CFI: Lightweight CFI Approach Against Code-Reuse Attacks Using Basic Block Information," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 19, no. 1, pp. 1–22, feb 2020.

[121] *CoreSight PTM-A9 – Technical Reference Manual*, ARM, Jul 2011, revision r1p0.

[122] *CoreSight Program Flow Trace – PFTv1.0 and PFTv1.1 – Architecture Specification*, ARM, Mar 2011.

[123] *ZedBoard (Zynq Evaluation and Development) - Hardware User's Guide*, Digilent, Jan 2014, version 2.2.

[124] *ARM CoreSight Components – Technical Reference Manual*, ARM, Jul 2009.

[125] *ARM CoreSight SoC-400 – Technical Reference Manual*, ARM, Sept 2013, revision r3p1.

[126] *ARM Embedded Trace Macrocell Architecture Specification – ETMv4.0 to ETMv4.3*, ARM, Apr 2017.

[127] *ARM Architecture Reference Manual – ARMv8, for ARMv8-A architecture profile*, ARM, Dec 2017.

[128] *ARM Compiler armasm User Guide*, ARM, Nov 2016, version 5.06.

[129] *ARM Cortex-A53 MPCore Processor – Technical Reference Manual*, ARM, Jul 2011, revision r0p4.

[130] *ZCU102 Evaluation Board - User Guide*, Xilinx, Jan 2019, version 1.5.

[131] *Leon3 / Leon3-FT – SPARC V8 32-Bit Processor Companion Core Data Sheet*, Gaisler, Mar 2010, accessed 21st Apr. 2019. [Online]. Available: http://www.actel.com/ipdocs/LEON3_DS.pdf

[132] A. M. Keller and M. J. Wirthlin, "Benefits of Complementary SEU Mitigation for the LEON3 Soft Processor on SRAM-Based FPGAs,"

IEEE Transactions on Nuclear Science, vol. 64, no. 1, pp. 519–528, 2017.

[133] A. Lindoso, L. Entrena, M. Garcia-Valderas, and L. Parra, "A Hybrid Fault-Tolerant LEON3 Soft Core Processor Implemented in Low-End SRAM FPGA," *IEEE Transactions on Nuclear Science*, vol. 64, no. 1, pp. 374–381, jan 2017.

[134] S. Clerc, F. Abouzeid, G. Gasiot, J.-M. Daveau, C. Bottoni, M. Glorieux, J.-L. Autran, F. Cacho, V. Huard, L. Dugoujon, R. Weigand, F. Malou, L. Hili, and P. Roche, "Space radiation and reliability qualifications on 65nm CMOS 600MHz microprocessors," in *2013 IEEE International Reliability Physics Symposium (IRPS)*. IEEE, apr 2013, pp. 6C.1.1–6C.1.7.

[135] J. Henkel, A. Herkersdorf, L. Bauer, T. Wild, M. Hubner, R. K. Pujari, A. Grudnitsky, J. Heisswolf, A. Zaib, B. Vogel, V. Lari, and S. Kobbe, "Invasive manycore architectures," in *17th Asia and South Pacific Design Automation Conference*. IEEE, jan 2012, pp. 193–200.

[136] L. Parra, A. Lindoso, F. Restrepo-Calle, M. Portela, S. Cuenca-Asensi, L. Entrena, and A. Martinez-Alvarez, "Efficient Mitigation of Data and Control Flow Errors in Microprocessors," *IEEE Transactions on Nuclear Science*, vol. 61, no. 4, pp. 1590–1596, 2014.

[137] N. Stollon, *On-Chip Instrumentation: Design and Debug for Systems on Chip*, 1st ed. Springer US, 2011.

[138] *Intel® 64 and IA-32 Architectures Software Developer's Manual*, Intel, Oct 2019.

[139] "Guidelines for the use of thec++14 language in critical andsafety-related systems."

[140] K. Asanović, R. Avizienis, and J. a. Bachrach, "The rocket chip generator," Tech. Rep., Apr 2016, Accessed: 2021-01-31. [Online]. Available: https://people.eecs.berkeley.edu/~krste/papers/EECS-2016-17.pdf

[141] E. Matthews and L. Shannon, "TAIGA: A new RISC-V soft-processor framework enabling high performance CPU architectural features," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, sep 2017, pp. 1–4. [Online]. Available: http://ieeexplore.ieee.org/document/8056766/

[142] "XA Zynq-7000 SoC Data Sheet: Overview," Xilinx, 2018, Accessed: 2019-11-18. [Online]. Available: https://www.xilinx.com/support/documentation/data_sheets/ds188-XA-Zynq-7000-Overview.pdf

[143] A. B. de Oliveira, L. A. Tambara, F. Benevenuti, L. A. C. Benites, N. Added, V. A. P. Aguiar, N. H. Medina, M. A. G. Silveira, and F. L. Kastensmidt, "Evaluating Soft Core RISC-V Processor in SRAM-Based FPGA Under Radiation Effects," *IEEE Transactions on Nuclear Science*, vol. 67, no. 7, pp. 1503–1510, jul 2020.

[144] J. Tonfat, F. Lima Kastensmidt, P. Rech, R. Reis, and H. M. Quinn, "Analyzing the Effectiveness of a Frame-Level Redundancy Scrubbing Technique for SRAM-based FPGAs," *IEEE Transactions on Nuclear Science*, vol. 62, no. 6, pp. 3080–3087, dec 2015.

[145] L. A. C. Benites, F. Benevenuti, A. B. De Oliveira, F. L. Kastensmidt, N. Added, V. A. P. Aguiar, N. H. Medina, and M. A. Guazzelli, "Reliability Calculation With Respect to Functional Failures Induced by Radiation in TMR Arm Cortex-M0 Soft-Core Embedded Into SRAM-Based FPGA," *IEEE Transactions on Nuclear Science*, vol. 66, no. 7, pp. 1433–1440, jul 2019.

[146] R. Lee, "Soft Error Mitigation Using Prioritized Essential Bits," Application Note, 2012, available at https://www.xilinx.com/support/documentation/application_notes/xapp538-soft-error-mitigation-essential-bits.pdf, accessed 6th May. 2019.

[147] L. A. Tambara, "Analyzing the impact of radiation-induced failures in all programmable system-on-chip devices," Ph.D. dissertation, Universidade Federal do Rio Grande do Sul (UFRGS), 2017.

[148] M. Pena-Fernandez, A. Lindoso, L. Entrena, and M. Garcia-Valderas, "Error Detection and Mitigation of Data-Intensive Microprocessor Applications Using SIMD and Trace Monitoring," *IEEE Transactions on Nuclear Science*, vol. 67, no. 7, pp. 1452–1460, jul 2020.

[149] J. Arlat, Y. Crouzet, J. Karlsson, P. Folkesson, E. Fuchs, and G. H. Leber, "Comparison of physical and software-implemented fault injection techniques," *IEEE Transactions on Computers*, vol. 52, no. 9, pp. 1115–1133, 2003.

[150] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proceedings of the Fourth Annual IEEE International*

Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538). IEEE, 2001, pp. 3–14.

[151] *Procedure Call Standard for the ARM Architecture*, ARM, Nov 2015.

[152] S. Analytics, "Accelerating the future: The economic impact of the emerging passenger economy," 2017. [Online]. Available: https://newsroom.intel.com/newsroom/wp-content/uploads/sites/11/2017/05/passenger-economy.pdf

[153] E. W. Dijkstra, "Solution of a problem in concurrent programming control," *Communications of the ACM*, vol. 8, no. 9, p. 569, sep 1965.

[154] J. Clark and D. Pradhan, "Fault injection: a method for validating computer-system dependability," *Computer*, vol. 28, no. 6, pp. 47–56, jun 1995.

[155] R. Vemu, S. Gurumurthy, and J. A. Abraham, "ACCE: Automatic correction of control-flow errors," in *Proceedings - International Test Conference*, 2007, pp. 1–10.

[156] W. Chao, F. Zhongchuan, C. Hongsong, B. Wei, L. Bin, C. Lin, Z. Zexu, W. Yuying, and C. Gang, "CFCSS without aliasing for SPARC architecture," *Proceedings - 10th IEEE International Conference on Computer and Information Technology, CIT-2010, 7th IEEE International Conference on Embedded Software and Systems, ICESS-2010, ScalCom-2010*, no. Cit, pp. 2094–2100, 2010.

[157] S. Mitra, K. Brelsford, Y. M. Kim, H. H. K. Lee, and Y. Li, "Robust system design to overcome CMOS reliability challenges," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 1, no. 1, pp. 30–41, 2011.

[158] A. Barenghi, L. Breveglieri, I. Koren, and D. Naccache, "Fault Injection Attacks on Cryptographic Devices: Theory, Practice, and Countermeasures," *Proceedings of the IEEE*, vol. 100, no. 11, pp. 3056–3076, nov 2012.

[159] J. Zhang, R. Hou, W. Song, S. A. Mckee, Z. Jia, C. Zheng, M. Chen, L. Zhang, and D. Meng, "RAGuard: An Efficient and User-Transparent HardwareMechanism against ROP Attacks," *ACM Transactions on Architecture and Code Optimization*, vol. 15, no. 4, pp. 1–21, jan 2019.

[160] P. Shivakumar, M. Kistler, S. Keckler, D. Burger, and L. Alvisi, "Modeling the effect of technology trends on the soft error rate of combinational

logic," in *Proceedings International Conference on Dependable Systems and Networks*. IEEE Comput. Soc, 2002, pp. 389–398.

[161] G. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. August, "SWIFT: Software Implemented Fault Tolerance," in *International Symposium on Code Generation and Optimization*. IEEE, 2005, pp. 243–254.

[162] R. G. Ragel and S. Parameswaran, "Hardware assisted pre-emptive control flow checking for embedded processors to improve reliability," in *Proceedings of the 4th international conference on Hardware/software codesign and system synthesis - CODES+ISSS '06*. New York, New York, USA: ACM Press, 2006, p. 100.

[163] F. Wang and V. D. Agrawal, "Single Event Upset: An Embedded Tutorial," in *21st International Conference on VLSI Design (VLSID 2008)*. IEEE, 2008, pp. 429–434.

[164] M. Grosso, M. S. Reorda, M. Portela-Garcia, M. Garcia-Valderas, C. Lopez-Ongil, and L. Entrena, "An on-line fault detection technique based on embedded debug features," in *2010 IEEE 16th International On-Line Testing Symposium*. IEEE, jul 2010, pp. 167–172.

[165] A. Shrivastava, A. Rhisheekesan, R. Jeyapaul, and C.-J. Wu, "Quantitative Analysis of Control Flow Checking Mechanisms for Soft Errors," in *Proceedings of the The 51st Annual Design Automation Conference on Design Automation Conference - DAC '14*. New York, New York, USA: ACM Press, 2014, pp. 1–6.

[166] M. Duricek and T. Krajčovič, "Interactive hybrid Control-flow checking method," in *International Conference on Applied Electronics*, no. January. IEEE, 2014, pp. 79–82.

[167] Y. Lee, I. Heo, D. Hwang, K. Kim, and Y. Paek, "Towards a practical solution to detect code reuse attacks on ARM mobile devices," in *Proceedings of the Fourth Workshop on Hardware and Architectural Support for Security and Privacy - HASP '15*, vol. 14-June-20. New York, New York, USA: ACM Press, 2015, pp. 1–8.

[168] E. Chielle, B. Du, F. L. Kastensmidt, S. Cuenca-Asensi, L. Sterpone, and M. S. Reorda, "Hybrid soft error mitigation techniques for COTS processor-based systems," in *LATS 2016 - 17th IEEE Latin-American Test Symposium*, 2016, pp. 99–104.

[169] Y. Lee, J. Lee, I. Heo, D. Hwang, and Y. Paek, "Integration of ROP/JOP monitoring IPs in an ARM-based SoC," in *Proceedings of the 2016*

Design, Automation and Test in Europe Conference and Exhibition, DATE 2016. EDAA, 2016, pp. 331–336.

[170] I. Agirre, J. Abella, M. Azkarate-Askasua, and F. J. Cazorla, "On the tailoring of CAST-32A certification guidance to real COTS multicore architectures," in *2017 12th IEEE International Symposium on Industrial Embedded Systems, SIES 2017 - Proceedings*, vol. 2017-June, 2017, pp. 1–8.

[171] N. Decker, B. Dreyer, P. Gottschling, C. Hochberger, A. Lange, M. Leucker, T. Scheffel, S. Wegener, and A. Weiss, "Online analysis of debug trace data for embedded systems," in *Proceedings of the 2018 Design, Automation and Test in Europe Conference and Exhibition, DATE 2018*. EDAA, 2018, pp. 851–856.

[172] P. Muntean, M. Neumayer, Z. Lin, G. Tan, J. Grossklags, and C. Eckert, "Analyzing control flow integrity with LLVM-CFI," in *Proceedings of the 35th Annual Computer Security Applications Conference on - ACSAC '19*. New York, New York, USA: ACM Press, 2019, pp. 584–597.

[173] S. Canakci, L. Delshadtehrani, B. Zhou, A. Joshi, and M. Egele, "Efficient context-sensitive cfi enforcement through a hardware monitor," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, C. Maurice, L. Bilge, G. Stringhini, and N. Neves, Eds. Springer International Publishing, 2020, pp. 259–279.

[174] ARAMIS-II, "ARAMIS-II Project Homepage," https://www.aramis2.com/, 2019, Accessed: 2017-11-12.

[175] Oberhumer, "LZO Compression Library," http://www.oberhumer.com/opensource/lzo/, 2017, Accessed: 2017-11-12.

[176] AWS, "The FreeRTOS™ Kernel," https://freertos.org/, 2017, Accessed: 2019-11-12.

[177] S. Segars, "ARM Processor Evolution," Hot Chips (HC23) Keynote, Standford University, 2011, Accessed: 2019-11-18. [Online]. Available: https://www.hotchips.org/wp-content/uploads/hc_archives/hc23/HC23.18.2-zkeynote1/HC23.18.keynote1.Arm-Hot%20Chips%202011%20SS%20Final.pdf

[178] M. Wolfe, "Compilers and More: Is Amdahl's Law Still Relevant?" 2015. [Online]. Available: https://www.hpcwire.com/2015/01/22/compilers-amdahls-law-still-relevant/

[179] *MicroBlaze Processor Reference Guide*, Xilinx, Jun 2020, v22020.1. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_1/ug984-vivado-microblaze-ref.pdf

[180] *Cortex-A9 MPCore – Technical Reference Manual*, ARM, Jun 2012, revision r4p1.

[181] "Position Paper CAST-32A: Multi-core Processors," Tech. Rep., 2016, accessed 18th Apr. 2019. [Online]. Available: https://www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/cast_papers/media/cast-32A.pdf