
Concepts for run-time and error-resilient control flow checking of embedded RISC CPUs

Daniel Ziener* and Jürgen Teich

Department of Computer Science,
University of Erlangen-Nuremberg,
Am Weichselgarten 3,
Erlangen 91058, Germany
E-mail: daniel.ziener@cs.fau.de
E-mail: teich@cs.fau.de
*Corresponding author

Abstract: In this paper, we introduce new concepts and methods for checking the correctness of control flow instructions (CFI) issued during the execution of programs for embedded RISC CPUs. Our proposed methodology is able to detect at run-time any error of illegal or faulty direct jump and branch instruction as well as call and return from subroutine for a given program code. Furthermore, two different hardware concepts and implementations of generic *control flow (CF) checker units* which may be tightly attached to a given CPU are proposed. These implementations can detect and even avoid the execution of faulty CFI at very low area and usually no latency penalty. Other benefits of this novel approach are that the application code must not be changed or augmented by signatures or additional instructions at all. The presented approach is, thus, completely transparent to the program developer.

Keywords: autonomous elements; control flow checker; CFI method; control flow instruction method; CF method; control flow method; embedded CPUs; error-resilient control flow checking; Leon core; monitoring.

Reference to this paper should be made as follows: Ziener, D. and Teich, J. (2009) 'Concepts for run-time and error-resilient control flow checking of embedded RISC CPUs', *Int. J. Autonomous and Adaptive Communications Systems*, Vol. 2, No. 3, pp.256–275.

Biographical notes: Daniel Ziener received his diploma degree (Dipl.-Ing. (FH)) in Electrical Engineering from the University of Applied Science, Aschaffenburg, Germany, in August 2002. In 2003, he joined the Fraunhofer Institute of Integrated Circuits (IIS) in Erlangen, Germany as a Research Staff in electronic imaging group and the Chair of Hardware-Software-Co-Design at the University of Erlangen-Nuremberg headed by Prof. Jürgen Teich as PhD student. His main research interests are IP core watermarking, the efficient usage of the FPGA structures, design of signal processing FPGA cores and reliable and fault tolerant processors.

Jürgen Teich received his Masters degree (Dipl.-Ing.) in 1989 from the University of Kaiserslautern. From 1989 to 1993, he was a PhD student at the University of Saarland, Saarbrücken, Germany from where he received his PhD (Dr.-Ing.) degree. In 1994, he was PostDoc at UC Berkeley. Habilitation in 1996 from ETH Zurich. From 1998 to 2002, he was a Full Professor in the Department of Electrical Engineering and Information Technology at the University of Paderborn, Germany, holding a Chair in Computer Engineering. Since 2003, he is appointed as a Full Professor in the Department of Computer

Science at the University of Erlangen-Nuremberg, holding a Chair on Hardware/Software Co-Design. He is senior member of the IEEE and has been program-chair of many EDA-related conferences such as CODES+ISSS 2007, FPL'08 and ASAP'10. He has published more than 300 papers in peer-reviewed conference proceedings and journals.

1 Introduction

Modern electronic systems are integrated more and more together with communication devices. In the past, aero planes were steered by cables, axes and hydro pneumatic systems. Now, planes become fitted with 'fly-by-wire' systems without any direct mechanical coupling between the pilot's control elements and the actuators. Clearly, such systems require a very high standard of reliability. The Airbus A380, for instance, has reached a new dimension on integration of wire-based and wireless communication components (Ziegler and Benz, 2005).

Thus, from the researcher's point of view, the focus is constantly shifting from the integration of new technologies to the effort to increase the reliability and security of existing systems. The manifold use of electronic systems raises the market of reliable and comfortable products.

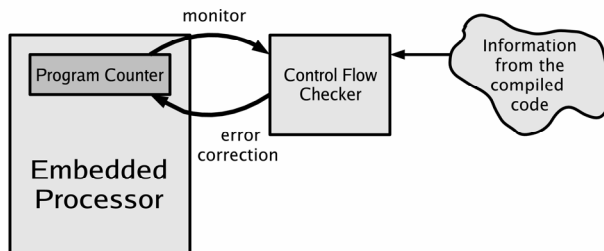
Robustness, reliability and security are essential requirements of today's systems on chips (SoCs). Modules and their integration in the system have to be designed to be still operational also in difficult and inference-prone areas as well as insecure environments (Stechele et al., 2007).

In this paper, our goal is to investigate methods to recognise, analyse and correct sporadic and/or permanent errors occurring in the control paths of embedded RISC CPUs. Our vision thereby is to define autonomously behaving elements that resolve functional errors of a RISC CPU Core locally inside the core at run-time (during the program execution) and preventing false instructions to be executed at all which means that no faulty branch or jump instruction shall be executed finally leading to a vulnerable or wrong program state (error-resilience).

Corresponding autonomous elements are called *control flow (CF) checkers*. They are supposed to recognise, evaluate and even correct errors during the program execution of the processor. In particular, soft errors (Lee et al., 2006) as well as malign security attacks (US-CERT, 2008) are in the focus of this paper. Detecting and avoiding the execution of faulty or corrupted CFI, such as branches and jumps are a problem of growing importance w.r.t. reliability and security. On the other hand, hardware overhead and an easy integration into processor design and application development flows are of utmost important in many cost-sensitive, particularly, in embedded systems.

In the following, we propose such concepts and an implementation of a corresponding CF checker hardware (see Figure 1). The main task of the control logic in a CPU is to control the program flow. The actual state of execution of a program is, in general, given by the value of the program counter and the CPU registers. Usually, the next instruction to fetch is given by an increment of the program counter, but also branches and jumps may occur.

Figure 1 Concepts of autonomously interacting control flow checker that can monitor and correct the program counter



Note: Control flow checker that can monitor the program counter and detect false jumps or branches based on information in the compiled code. Moreover, the checker should also be able to correct false jumps and branches.

Sources of errors that we want to autonomously detect and correct include accidental sporadic, permanent or intended errors caused by local attacks that try to manipulate the program execution. These errors can effect a wrong program counter value. Errors may also be caused by pure software means like buffer overflows. Here, a wrong jump destination or a wrong return address from a subroutine might cause the execution of infiltrated code. If an error is detected, the CF checker should be able to initiate the reexecution of the control flow instruction (CFI). A quite general definition of CF checking may be given as follows:

Definition: *CF checking denotes the task to test whether a sequence of program counter values is correct with respect to a given program specification.*

In the following, we give a summary of the structure of this paper. In Section 2, a general overview of related work is given. In Section 3, we present a classification of CFI. Subsequently, in Section 4, two different methodological concepts for CF checking are introduced. Architectural concepts for implementing these ideas present the focus of Section 5. Finally, in Section 6, we present an implementation of an autonomous CF checker for a given real *Leon3* (Gaisler Research) CPU and analyse the corresponding overheads for CF checking in Section 7. Section 8 concludes the work.

2 Related work

Error detection and correction methods have important roots in the area of fault-tolerance. Here, one of the most familiar method for error detection is the duplication of a given processor core with subsequent comparison of the results (Mueller et al., 1999). Duplication of processing units is, however, most of the times too cost-intensive and, thus, prohibitive due to cost (area) and power consumption. Hence, these approaches are only used in safety-critical systems with a high demand on reliability.

In the following, we first define relevant criteria for comparing different methodologies for CF checking quantitatively. Here, the following criteria will be used: *fault coverage* denotes the degree of faults that can be detected by a method. For example, some methods for CF checking discussed in the following can only detect a certain type of CFI (e.g. direct branches and jumps). Some may detect not 100% of all CF

errors. Section 4.4 presents the fault coverage of our methods. Another criterion for comparison is the *detection latency*. The detection latency denotes the time between occurrence of a fault and its detection. This time is important to prevent a system failure. Only if an error is detected with a low latency, the error handling can react to transfer the system into a secure state or to trigger error correction measures. Our approach is able to detect a faulty instruction prior to finally executing it. It is thus preventing faulty branch and jump instructions to be executed at all and thus guaranteeing error-resilience. On the other hand, the following overheads may be caused: *execution time overhead* (CPU time), *memory overhead* and *area overhead* (hardware cost overhead).

Related work on CF checking can be divided into approaches using an additional hardware checker unit or a watchdog processor (Arora et al., 2006; Lu, 1982; Majzik et al., 1994; Michel, Leveugle and Saucier, 1991; Ragel, 2006; Schuette and Shen, 1987), and approaches which are completely software-based (Goloubeva et al., 2003). In these approaches, the program code is first structured into basic blocks.

A basic block is a sequence of code which is executed successively without any jumps or branches except, possibly, at the end. The basic block can only be left at the end of a block and can only be entered at the beginning. Only the last instruction can be a jump or branch and only the first instruction can be a jump or branch destination (see Section 1).

CF checking using assertions (CCA) (Goloubeva et al., 2003) denotes a software-based approach. After creating a basic block graph, a sequence of special control instructions is inserted into the program code at the beginning as well as at the end of each basic block. These additional instructions verify that only legal branch or jump destinations according to the specification, given by the basic block graph is taken. The advantage is that no additional hardware (area overhead) is required, but this approach has an obvious impact on the performance of the program code (execution time overhead). Finally, undesired jumps caused by faults occurring on instructions inside a basic block cannot be detected at all, and are also not prevented from being executed.

A good overview over software methods for CF checking for security and fault tolerance is given in Abadi et al. (2005).

To check all types of instructions, a signature (hash or a CRC value) of all instructions of a basic block can be calculated offline (at compile-time). At run-time, a hardware checker can calculate the signature of the executed instruction in a basic block. When leaving a basic block, the signatures can be compared and errors inside the basic block can be found. Signature methods can be divided into two groups, namely *embedded signature monitoring* (ESM) (Lu, 1982; Majzik et al., 1994; Schuette and Shen, 1987) and *autonomous signature monitoring* (ASM) (Arora et al., 2006; Michel, Leveugle and Saucier, 1991).

In the ESM methods, the offline calculated signature (golden signature) is stored in the program code with additionally inserted instructions at the end of each basic block. These instructions read out the calculated signature of the executed instructions from the checker unit and compare it to the golden signature. The advantage of these methods is that all types of instructions can be checked and a new program contains already the corresponding signature. The disadvantages are a significant performance impact (execution time overhead) and that a fault can only be detected at the end of a basic block which may be too late to prevent a system failure (detection latency). Also, a single event upset (SEU) during the execution of the additionally inserted instruction can lead to a false detection or spoofing of an error.

In the ASM methods, the golden signature is stored in a separate memory belonging to the checker unit. Also, the comparator for the golden and the calculated signature is implemented in hardware. The information of the basic block graph is mapped into micro-instructions located inside the instruction memory of the checker. Jumps and branch destinations can thus also be checked. The advantages are that the program code must not be altered and that there is no performance impact. Also, all types of instructions can be monitored. The disadvantages are that an extra memory for the checker unit is required (memory overhead) and that synchronisation between the CPU and the checker unit is difficult. So, interrupts, multithreading and indirect jumps cannot be completely covered.

An ASM approach for security applications is described in Arora et al. (2006). The *intra-procedural CF checking* methodology is similar to ours. The advantages of ours, however, is that we are

1. more flexible in using memories to store the CF (instruction) graph instead of synthesising a dedicated finite state machine (FSM) in logic for each program to be executed
2. moreover, we have no performance impact in the error-free case
3. our checker unit is simpler and thus requires less resources.

Finally, the Diva approach (Austin, 1999) describes a pipeline which accepts only checked results for the further processing after the commit phase. A redundant second pipeline is a simple rudimentary pipeline, where the results of arithmetic functions are recalculated with a separate checker and memory items are refetched. The weakness is that the second pipeline is assumed to be fault-free, which might not be the reality in today's deep submicron designs. The area overhead of Diva is lower than in the case of fully redundant units, but also performance reduction exists due to a longer pipeline.

3 Branches and jumps

CFI can be categorised into conditional branches and unconditional jumps. Conditional branches depend on the result of a logical or arithmetic operation.

Both groups of CFI can be further subdivided into direct (static) and indirect (dynamic) jumps or branches. The destination of direct branches or jumps is fixed at compile-time and is encoded into the jump or branch instruction in an absolute or relative address. For indirect jumps or branches, the destination address is determined during program execution. The destination address is given by either a register value or as the result of an operation with registers or the result of an operation with a register and a constant value which is encoded into the instruction. Absolute or relative addressing modes can be used there.

Summarising, four types of CFI exist: (*Unconditional*) *direct jumps* (e.g. `call`, `goto`), (*Conditional*) *direct branches* (e.g. `if .. then .. else`), (*Unconditional*) *indirect jumps* (e.g. `return` from subroutine) and (*Conditional*) *indirect branches*.

Furthermore, the class of unconditional indirect jumps can be subdivided into *returns from subroutine*, *register indirect calls* and *other jumps*. A return from subroutine is an example of an indirect jump because the program counter jumps to the address where the routine is called from, and this address is only known at run-time. Register indirect calls are calls where the address of the called subroutine is determined at run-time.

Table 1 Accumulated number of CFI of benchmarks of the SPEC CINT2000 test suite (SPEC) when compiled to the SPARC (SPARC) architecture

<i>SPEC program</i>	<i>Direct</i>		<i>Indirect</i>		
	<i>Branches</i>	<i>Jumps</i>	<i>Returns</i>	<i>Calls</i>	<i>Other jumps</i>
<i>gzip</i>	1426	599	111	4	0
<i>gcc</i>	54,791	22,446	2236	140	273
<i>vpr</i>	2764	2012	269	2	7
<i>mcf</i>	288	82	26	0	0
<i>crafty</i>	4814	4074	108	0	13
<i>parser</i>	3189	1701	320	0	2
<i>gap</i>	18,733	4158	828	1262	5
<i>vortex</i>	12,537	8491	913	15	21
<i>bzip2</i>	2106	1835	189	27	17
<i>twolf</i>	5701	2060	189	0	2

Finally, also jumps which are not triggered by an instruction can occur such as *interrupts* and *traps*. The destinations of interrupts are typically given by the start address of the main interrupt service routine, and so, interrupts are direct jumps. Traps occur on exception conditions (like divide by zero). Here, the program jumps to the address of an exception handler, and so, traps can be treated as direct jumps.

Table 1 shows an analysis of the occurrence rates of these different types of branches and jumps on the SPARC architecture for the SPEC CINT2000 benchmark (Standard Performance Evaluation Corporation (SPEC), 2000) for a given list of programs. As it can be seen, indirect calls and jumps occur relatively rarely as opposed to direct branches and jumps, too.

4 Methods for autonomous CF checking

In SoCs, a CPU often executes only a few specified programs over its lifetime. This holds true, particularly, for embedded applications where the system is often only programmed once, and the code is never changed during the lifetime of the product, except for the update of the SoC with a new firmware and software. Furthermore, it is well-known that in many computational intensive problems, most of the execution time is spent in only few subroutines. So, it is beneficial to analyse these subroutines for branches and jumps statically.

If we may assume that only direct jumps and branches exist in a given code segment, we are able to check the CF of this code by verifying the correct execution of each direct CFI as well as the (successively) linear execution of all the other instructions (the program counter value is incremented by one word address after each instruction).

To check the correct execution of CFI, we need to check the correct address of the CFI and the correct target address. The program counter value before and after the execution of a CFI can be compared to these addresses. If there is a mismatch, an error signal may be raised.

In the following, we propose two alternative methods to obtain the correct addresses of CFI of a given machine program and the corresponding targets.

The first method is called *basic block* or *CF method*. The second method is called *CFI method*.

4.1 CF method

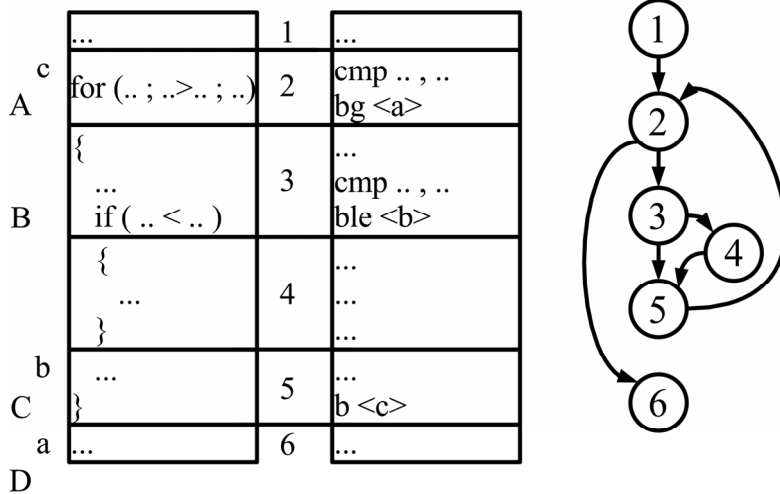
First, a given compiled machine code is separated into a set of basic blocks BB. The following instructions define the begin of a basic block:

1. the first instruction in a program or segment
2. the instruction following a CFI
3. instructions which are destinations of CFI.

From this information, the control flow graph CFG (BB, T) is built: each node $BB_i \in BB$ of the control flow graph represents a basic block. The nodes are sorted with increasing start address of the corresponding basic block in ascending address order. Each edge $t \in T$ represents a transition of the CF from one basic block to another. If the last instruction of a basic block BB_i is a direct branch instruction, the basic block has two successors. One is the basic block next in the list BB_{i+1} (if the branch is not taken) and to a basic block where the first instruction is the branch destination (if the branch is taken). Jumps have only one successor, and if the last instruction is not a CFI, the successor basic block is always the next basic block BB_{i+1} . An example program is shown in Figure 2 which is separated into basic blocks. Also, the corresponding CFG is shown.

With the given CFG, we have all information to check a sequence of program counter values for correctness as follows: the information of the CFG can be either used to directly define a FSM to check the correctness of CFI. Alternatively, an implementation using micro-instructions of a micro-programmed circuit can be deduced from the CFG.

Figure 2 An example program code is given on the left hand side together with the corresponding assembler code and on the right hand side, the corresponding CFG is shown



Note: The CFIs are denoted *A* to *C* and the CFI destinations with *a* to *c*. *D* denotes the end of the program or segment to be checked. Furthermore, the code is divided into basic blocks, denoted with 1–6.

For an implementation of a micro-programmed circuit, the information of the CFG can be stored inside memories. For each basic block, we need the start and the end address and also the indices of the successor basic blocks. The start address of a basic block is the end address of the previous basic block incremented by one. To minimise the memory overhead, we only need the end address and a global start address. Also, we need to store only one successor of a basic block for branches because if the branch should not be taken, always the basic block with the next index (BB_{i+1}) shall be executed.

The correct CFI address may directly be stored inside the memory (basic block end address). The corresponding target address is the start address of the successor basic block, given by its index. To get this address, the end address of the basic block with the previous index is fetched and the address is incremented ($BB_{i-1} + 1$). Having both addresses, the CFI can be verified.

Note that, we can extend this method to check the integrity of all instruction sequences inside a basic block using a CRC or hash value. This value can be calculated at run-time from the executed instructions and can be compared at the end of a basic block with a precalculated value (Michel, Leveugle and Saucier, 1991).

For example, Arora et al. (2006) describe a CF method, where the CFG is implemented in hardware by a FSM and a lookup table for resolving the CFI addresses and indices (in memory).

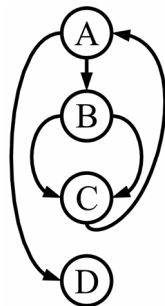
4.2 CFI method

In case of direct branches and jumps, the start and target address is known at compile-time. So, it is possible to extract this information from the binary or the disassembled program code by decoding the instructions. The CFI are then sorted by increasing addresses in ascending address order.

Then, the CFI *graph* (CFG (CFI, T)) is built: here, each CFI in the code which should be checked represents a node ($CFI_i \in CFI$). Directed edges $t \in T$ of the CFG denote transitions to the following CFI.

Like in a CFG, each node can have a maximum of two successors: two for a branch instruction and one in case of a jump instruction. For a branch instruction CFI_i , one successor is CFI_{i+1} (branch is not taken). The other successor of a direct branch and jump instruction is CFI_n , which is the next CFI in the program code after the branch destination (branch is taken). The CFG, of the example, program code is shown in Figure 3.

Figure 3 For the example program code in Figure 2, the corresponding CFG is shown



Note: Each node corresponds to a CFI and each edge denotes a transition.

Like in the CF method, the information of the CFG can be used as a specification of a CF checker unit and implemented either directly by a FSM or by micro-instructions of a micro-programmed circuit. For the micro-programmed circuit, we store for each CFI the start and the target address in memory. Also, the index of the successor CFI must be stored inside this memory. For direct branches, we store the successor CFI for taken branches. If the branch is not taken, the successor CFI is CFI_{i+1} .

4.3 Memory overhead discussion

The example program in Figure 2 has six nodes in the CFG and four nodes in the CFG. For the CF method, we need to store only one address for a CFG node (basic block) and the index of the successor block. In the CFI method, we need to store two addresses (CFI address and target address) and the index of the successor block for each CFI node. Usually, the index needs fewer bits as the addresses of instructions, so the CF method uses less memory than the CFI method for this example.

For measuring, the memory overhead for standard user programs, we are using the programs from the SPEC CINT2000 (SPEC) benchmark in the following (see Section 3). Table 2 shows the memory overhead caused to implement the CF and CFI method for the SPEC CINT2000 benchmark when compiled to the 32 bit SPARC architecture. The smallest index bit width is chosen for the given program to calculate the memory overhead in bits.

The results, in Table 2, show that the CF method usually produces a lower memory overhead than the CFI method.

The CFI method, on the other hand has no execution time overhead at all. The CF method must stall the processor for one clock cycle, if a basic block consists only of one instruction.

Fortunately, basic blocks with only one instruction are very rare. If an error occurs and the CFI must be reexecuted, four extra pipeline cycles occur in both methods.

Table 2 Required memory overhead of the programs of the SPEC CINT2000 benchmark in bits for the CF and CFI method. Also, the number of basic blocks and CFI and the corresponding index width is shown

<i>SPEC program</i>	<i>CF method</i>			<i>CFI method</i>		
	<i>Number of basic blocks</i>	<i>Index w. [bits]</i>	<i>Overhead [bits]</i>	<i>Number of CFIs</i>	<i>Index w. [bits]</i>	<i>Overhead [bits]</i>
gzip	2615	12	109,830	2140	12	154,080
gcc	90,819	17	4,268,493	79,886	17	6,151,222
vpr	6029	13	259,247	5054	13	368,942
mcf	514	10	20,560	398	9	27,324
crafty	10,205	14	449,020	9009	14	666,666
parser	6384	13	274,512	5212	13	380,476
gap	30,484	15	1,371,780	24,986	15	1,873,950
vortex	24,978	15	1,124,010	21,977	15	1,648,275
bzip2	5061	13	217,623	4174	13	304,702
twolf	9827	14	432,388	7952	13	580,496

4.4 Fault coverage

The faults inside a SoC can be categorised into *degenerations faults, manufacturing faults, design faults, attacks and single event transient (SET) or SEU*.

Degeneration faults are, for example, caused by the hot-carrier effect (Guerin et al., 2007), by electromigration (Clarke et al., 1990) or by time-dependent dielectric breakdown (TDDB) (Crook, 1979). All these faults are permanent chance run-time faults which lead first to timing errors and later, particularly electromigration, to value errors like open or short circuits.

Manufacturing faults are, for example, stuck at 0, 1 or open faults as well as bridge faults which lead to value errors. Also, signals which are after manufacturing too slow to meet timing constraints are manufacturing faults which cause timing errors. These faults are also permanent chance faults but emerge during the manufacturing process.

SET and SEU are sporadic transient chance run-time faults which may be caused by different types of radiation. A common fault is the impact of an alpha particle. These faults usually cause value errors.

Design faults are permanent chance development faults which are caused by an incorrect specification or implementation of the developer. But, also design tools can cause design faults.

Security attacks are typically transient or permanent intended run-time faults which cause usually value errors. Here, an unauthorised person tries to get access to the system by breaking the security facilities.

Obviously, which different types of faults can be detected depends heavily on the proposed error detection method. *Fault coverage* refers to the percentage of faults which can be detected from a defined set of faults.

The CFI method described in Section 4.2 is able to detect the following types of faults:

- Permanent and transient faults in instruction memory, memory bus, memory controller, instruction cache instruction register and logic between different pipelined instruction registers, if the instruction (the correct or the falsified) affects the CF (CFI).
- All permanent and transient faults in program counter register, logic between different pipelined program counter registers, as well as the logic of the calculation of the next program counter.
- Attacks which affect the CF (e.g. stack and heap smashing).
- Design faults in hardware which can be detected through diversity of the processor core and the checker unit, and in software caused by diversity between the compiled program and the checker unit entries (e.g. unauthorised software update).

Using the CF method presented in Section 4.1 with the additional instruction integrity checker extension we can further detect the following types of faults besides the faults mentioned above:

- Permanent and transient faults in instruction memory, memory bus, memory controller, instruction cache instruction register and logic between different pipelined instruction registers, until the pipeline step, where the instruction integrity checker is instantiated.

4.5 Methods conclusions

Both introduced methods can only check direct branches and jumps, where start and destination address can be extracted from the compiled code. For indirect CFI, we will present extensions for both methods. Some of these extensions are discussed later.

The advantage of the CF method is that in most cases, fewer additional memory resources are needed than for the CFI method.

The disadvantage of the CF method is that we need two times access to the memory for each CFI. One access for the end address of the basic block and one for the start address of the successor basic block. To ensure that on a branch or jump the correct start and destination address is available, we might pre-read both values. But this pre-read can only be done if the basic block consists of more than one instruction. If a basic block consists only of one instruction, we must stall the processor pipeline to verify the CFI to prevent possible erroneous behaviour.

The advantages of the CFI method are that the checker unit is very simple and uses only few logic resources. Also, we have no performance impact because the correct CFI address and target address may be loaded from the memory in a single clock cycle. The disadvantages are that usually more memory resources are needed as for the CF method, and that we are not able to check the integrity of non-CFIs.

5 Architectures for CF checking

In the following, we introduce hardware architectures for lightweight CF checking to monitor and to correct the executed CFI of a given RISC CPU. Our approaches can monitor direct jumps and branches as well as call and returns from subroutine. To achieve a correction of a corrupt program, a detected incorrect jump or branch can be re-executed. Our implementations may be called *lightweight* because they produce only little area overhead, and we can detect and correct many though not all errors. Our architecture concepts are modular in the sense that coverage aspects can be traded off for implementation overhead.

5.1 Handling of direct jumps/branches

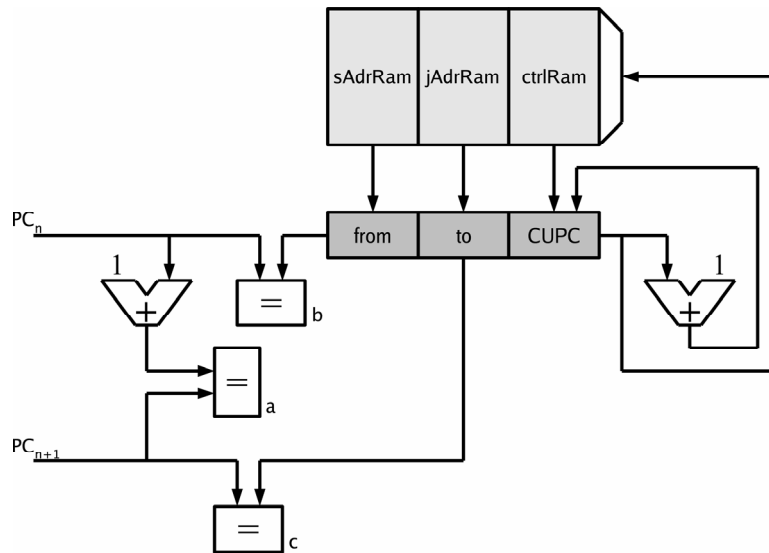
In this section, we describe architectures for the CF and CFI method introduced in Sections 4.1 and 4.2 to check direct jumps and branches. In particular, the CFG/CFIG information is mapped to dedicated memories as will be shown next.

5.1.1 An Architecture for the CFI method

The checker must know the instruction's program address and the address of the next instruction to execute. Since most CPU architectures today are pipelined, these addresses can easily be taken from successive pipeline stages of the program counter.

If no jump or branch instruction occurs, the next instruction address PC_{n+1} is typically one instruction word higher than the value of the current program counter PC_n . So, an incremented instruction address can be compared to the address after the instruction (see comparator a in Figure 4). If the current instruction is a direct jump or branch instruction, the next program counter is the jump destination, or, in the case of a branch, the branch destination. Or, if the branch is not taken, the next address in the program code.

Figure 4 Architecture for CF checker with three rams and three comparators and also, the CUPC is shown



Each pair of CFI address and target address is stored in two RAMs of the checker unit, one for the start and one for the target address (see Figure 4). The addresses of the branch or jump instructions are stored successively in the start address RAM (*sAdrRam*) and the corresponding targets in the jump address RAM (*jAdrRam*). Also, a checker unit program counter (CUPC) is needed which points in these RAMs to the cell, where the address of the next direct branch or jump is stored. This start address is compared to the current program counter to determine whether a branch or jump instruction is executed (comparator *b*). In this case, the following program counter value is compared to the address of the jump address RAM to verify the correct execution of the branch or the jump (comparator *c*). Now, the CUPC must point to the next branch or jump address. This can be achieved by introducing a third RAM (*ctrlRam*) where the next CUPC is stored for each branch or jump. In the case of a branch, it must also be determined if the branch is taken or not. If the branch is taken, the next CUPC has the value which is stored in the *ctrlRam*. If the branch is not taken, then the CUPC can be incremented. The CUPC and the *ctrlRam* presents a micro-programmed architecture which implements the CFIF. The CUPC can be compared with the index of the CFI. The transitions of the CFIF are stored in the *ctrlRam*.

Also, control flags are stored in the *ctrlRam*. So, the checker unit can distinguish between jumps or branches or can activate or deactivate the checker unit based on specific program addresses. This can be done by storing the checking start or end address in the *sAdrRam* and setting the checking start or end flag in the corresponding cell in the *ctrlRam*. If the program flow reaches the starting address, the checker unit will be activated, or, if the checking end address is reached, the checker unit deactivates itself. Finally, parts of the program flow, for example, non-critical sections or sections which cannot be checked due to not supported indirect jumps, might be excluded from the checking process by setting the checking start and end flags.

Usually, the processors stall in many wait cycles due to cache misses or pipeline stalls. However, an additional CPU wait cycle will only be produced for each basic block which consists of only one single instruction and if no other wait cycles which are not caused from the checker unit exists. So, it is not even sure that a basic block with only one instruction will cause an additional wait cycle when applying the CF method.

5.2 Handling of calls and returns

The most frequent use of indirect jumps occurs in the form of returns from subroutine, see for example, Table 1. By executing a *return from subroutine instruction*, the program counter jumps to the next address after the instruction from where the subroutine was called from. The return address is typically stored in a CPU register, so the return instruction is a special indirect jump. Returns can be verified also in our approach by introducing an additional *hardware stack*. Upon a call (direct or indirect), the return address is stored in the stack. Once the return instruction is executed, the correctness of the target address can be verified.

5.3 Handling of interrupts and traps

A jump may also occur in case of a trap or an interrupt. If there is an exception during the execution of an instruction (e.g. a division by zero), a trap is triggered. Then, a jump to a fixed or predefined address where the trap table is stored, is typically induced. The different kinds of traps are distinguished in the trap table, and the right trap service routine is called. After the successful execution of the *trap service routine*, a back jump to the instruction following the trap is initialised. Interrupts are handled similarly with a fixed or predefined address for the interrupt table.

It is important to note that a trap or an interrupt can occur asynchronously with respect to program execution. So, direct jumps to the base address for the trap and interrupt table are always allowed. Now, this base address for the trap and interrupt table can also be stored in the checker, and therefore, these jumps may be verified, too. Also, back jumps may be checked because the return address is automatically stored on the return stack (see Section 5.2).

5.4 Correction by reexecution

If a faulty jump or branch instruction occurs, this instruction will be reexecuted as follows: The error can be detected fast enough to ensure that the state of the CPU is not altered by the faulty instruction execution. To guarantee this, the checker must monitor the program counter in the first pipeline stage of a given RISC CPU. Unfortunately, in most architectures, the jump or branch instructions need more than one cycle to execute. So, until the error is detected, some other instructions after the jump might be executed already. After error detection, the program counter is reset to a value previous the error occurs by looping back the program counter value from a subsequent pipeline step. The details of the reexecution process depends highly on the processor architecture and design.

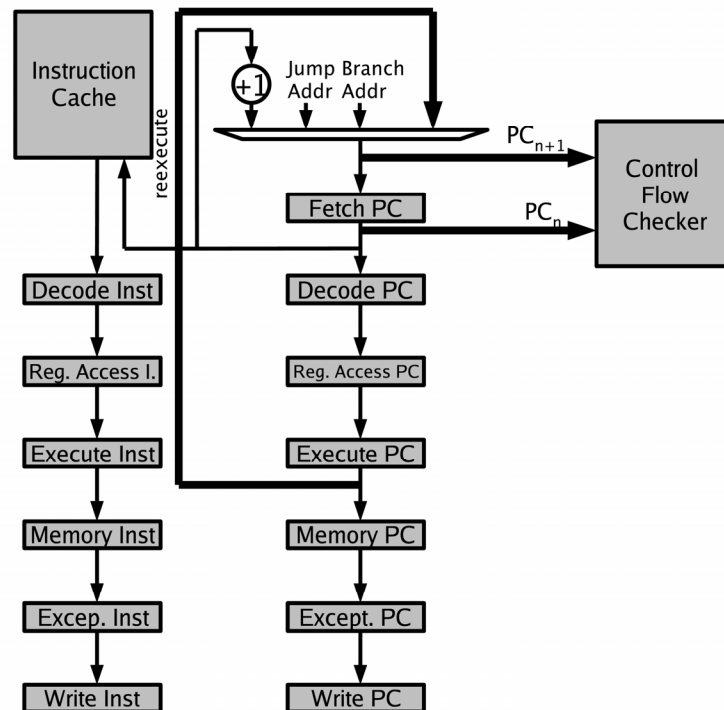
The SPARC architecture allows to execute one instruction after a branch instruction or two instructions after a jump instruction before the branch or jump is performed (see Sparc Architecture Manual (SPARC), 1992). If an error is detected

and the jump or branch instruction must be reexecuted, also these following instructions must be reexecuted. It must also be ensured that these instructions cannot alter the state (e.g. register content or memory operations) of the CPU before reexecution (see Section 6).

6 Example implementation

We prototyped and analysed the architecture implementing the CFI method (see Section 5.1.1) for the open source SPARC CPU *Leon3* from Gaisler Research (2008) in a *Virtex 4* FPGA from Xilinx. The checker can monitor direct branches, jumps and calls as well as indirect returns and has also the possibility to reexecute a corrupted jump or branch instruction by fetching it again from the memory. Other features of the checker are the support of activate and deactivate procedures described in Section 5.1.1. The complete methodology for CF checking consists of the concept of checking of direct jumps and branches (Section 5.1.1), the return stack (Section 5.2) and the repair mechanism (Section 5.4). To minimise the resource overhead, some features can be disabled (see Section 7). Indirect jumps which are not returns, are not supported so far, but many application programs or routines in embedded systems have none of these instructions, or should avoid their use.

Figure 6 The checker unit is placed in between the first pipeline stages of the *Leon3* core



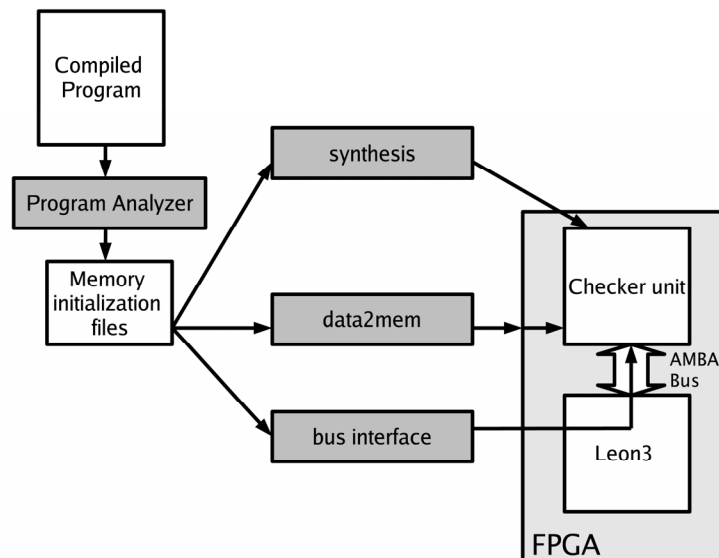
Note: All bold lines denote new signal paths needed for monitoring and reexecution of jump and branch instructions.

The checker is placed between the first pipeline stages of the *Leon3* core (see Figure 6). The current program counter for the checker is the program counter in the decode pipeline step and the next program counter is the program counter of the fetch pipeline step. For reexecuting a jump or branch, the program counter of the execution step is looped back to the program counter generation (a step prior to the fetch step), and the instructions are annulled after the execution step, so the incorrect instructions are not executed and no registers or memories are written. Because of the loop back, the jump is executed again, and if this execution is correct, the program is continued normally.

To prepare an application, the compiled code is analysed by a program which decodes the instructions and searches for jumps, branches, calls and returns (see Figure 7). The addresses of these instructions are stored in the *sAdrRam* initialisation file and the destination address, except for the return instruction, is stored in *jAdrRam* initialisation file. Also, the initialisation file for the control Ram (*ctrlRam*) is generated, and the corresponding activate and deactivate instructions for the checker unit are inserted. The original program of the application remains completely unchanged.

The memory initialisation files can be used for the synthesis of the checker unit, or the content of the rams can be initialised directly in the bitfile of the FPGA with the Xilinx tool '*data2mem*'. We also implemented an *AMBA Bus* interface for the checker unit to have access to the checker rams from the processor side. Using the bus interface, the memories can also be initialised at run-time over the memory bus or the processor.

Figure 7 From the compiled code, the program analyser extracts all branches, calls and returns and generates the memory initialisation files for the *sAdrRam*, *jAdrRam* and *ctrlRam*. The checker rams can be initialised during synthesis, later in the bitfile with the *data2mem* tool, or at run-time using a bus interface



7 Overhead analysis

Finally, we provide an overhead analysis for different versions of the checker, which support different jump instructions and error detection features, thus resulting in different area overheads.

The smallest version of the checker (version A) can only monitor direct jumps or branches. All indirect jumps are not supported and not allowed in the code, but it is allowed that indirect jumps can occur in the unchecked code. This includes also returns from subroutine, so this technique can only be used for a single procedure or function. But many of these procedures and functions can be checked if the checker unit is deactivated at calls and returns, and activated inside the function.

The second version (version B) is version A with an additional 32 entry return stack. With this version, we can also monitor calls and returns, so most application programs can be fully monitored.

The last version (version C) has the additional capability of repeating an incorrect jump or branch instruction as described in Sections 5.4 and 6.

Table 3 shows the overhead of different versions, synthesised and implemented on a *Virtex 4* with ISE 8.2 with different checker ram sizes. The results show that the area overhead for logic (lookup tables and flip flops) is very small relative to the amount of resources needed by the processor core. If more CFI shall be monitored and more checker ram entries are needed, only the overhead of necessary block rams (BRAMs) increases. The bus interface for the *AMBA bus* creates an additional area overhead of 68 *LUTs* and 5 *flip flops*.

Table 3 Area overheads of different checker unit versions (without the bus interface) with respect to a *Leon3* core without PCI and ethernet

Overhead	<i>Leon3</i>	<i>Version A</i>		<i>Version B</i>		<i>Version C</i>	
<i>Checker rams with 512 entries</i>							
<i>LUTs</i>	17031	106	0.62%	242	1.42%	259	1.52%
<i>Flip Flops</i>	5412	11	0.20%	17	0.31%	20	0.37%
<i>BRAMs</i>	50	3	6%	3	6%	3	6%
<i>Checker rams with 1024 entries</i>							
<i>LUTs</i>	17031	111	0.65%	248	1.46%	265	1.56%
<i>Flip Flops</i>	5412	12	0.22%	18	0.33%	21	0.39%
<i>BRAMs</i>	50	5	10%	5	10%	5	10%
<i>Checker rams with 2048 entries</i>							
<i>LUTs</i>	17031	112	0.66%	250	1.47%	267	1.57%
<i>Flip Flops</i>	5412	13	0.24%	19	0.35%	22	0.41%
<i>BRAMs</i>	50	8	16%	8	16%	8	16%
<i>Checker rams with 4096 entries</i>							
<i>LUTs</i>	17031	105	0.62%	251	1.47%	268	1.57%
<i>Flip Flops</i>	5412	14	0.26%	20	0.37%	23	0.42%
<i>BRAMs</i>	50	10	20%	10	20%	10	20%

Note: The area (*LUTs*) and memory overheads (*flip flops* and *BRAMs*) of the full version C and the reduced versions (A and B) and for different checker ram sizes are shown in absolute numbers and in percent of the resources needed by the *Leon3* core.

The verification of the checker has been performed by simulation and the Leon in-circuit debugger. Instruction faults are simulated between the instruction cache and the integer unit by an XOR with an error mask, read in from a file. With the in-circuit debugger, CFI can be altered inside the memory (e.g. the jump destination encoded inside the instruction) or the entries of the checker rams over the bus interface of the checker unit. With these techniques, the checker and the correction of incorrect jump instruction has been verified.

8 Conclusions and future work

We introduced a systematic methodology for autonomous CF checking for embedded RISC CPUs which can monitor direct jump and branches as well as returns from subroutine. Experimental results show that the additional hardware overhead is quite small. In particular, lookup tables and flip flops overhead amount to an overhead of less than 2% of the CPU core requirements in all cases. So, the only overhead results from the additional memory needed to monitor the CFI. A modular concept for generation of generic micro-programmable checker units has been proposed, so the area overhead can be further reduced, by removing some functions. The detection of errors happens during the execution of the erroneous instruction, so we have the possibility to react immediately and prevent those incorrect instructions from being finally executed. Furthermore, an incorrect jump or branch instruction can be refetched and reexecuted. With this technique, we therefore have no performance impact on the CPU and the compiled program code remains unchanged.

In our implementations, an independent program counter CUPC with its own state machine and own micro-code with own micro-instructions (*ctrlRam*) was introduced. The checker micro-program code is based on the extracted branch and jump instructions from the program code. This reduced code covers only direct branches or jumps without the instructions between two branch points. With this technique, we enhanced the CPU with a reduced second independent program counter and instruction unit at minimum additional hardware cost and full control of the program flow.

Further extensions can be the support of indirect jumps, multi-threading and the check of the correct computation of conditionals of branches. Also, an interface to the operation system could be usefully to count the errors and to report the reliability of the CPU.

Finally, using the bus interface of the checker unit, the contents or part of the content of the checker rams might also be stored in the system memory instead of dedicated memories. Only the content for checking the current part of program (e.g. the current function or a set of functions which are current in use) may be held in the local checker rams. If the checker needs information which is not stored inside the local checker rams, the checker can generate a page-fault-like event to signal the operating system to reload the checker rams with the needed contents. This concept of caching can reduce the memory overhead significantly.

Acknowledgement

This work has been supported by BMBF project 01 M 3083 ‘Autonome Integrierte Systeme’.

References

- Abadi, M., Budi, M., Erlingsson, U. and Ligatti, J. (2005) ‘Control-flow integrity’, *CCS '05: Proceedings of the 12th ACM Conference on Computer and Communications Security*, New York, NY: ACM Press, pp.340–353, ISBN 1-59593-226-7. doi, Available at: <http://doi.acm.org/10.1145/1102120.1102165>.
- Arora, D., Ravi, S., Raghunathan, A. and Jha, N.K. (2006) ‘Hardware-assisted run-time monitoring for secure program execution on embedded processors’, *IEEE Transactions on VLSI Systems*, Vol. 14, No. 12, pp.1295–1308.
- Austin, T.M. (1999) ‘DIVA: a reliable substrate for deep submicron microarchitecture design’, *International Symposium on Microarchitecture*, pp.196–207, Available at: citeseer.ist.psu.edu/austin99diva.html.
- Clarke, P.J., Ray, A.K. and Hogarth, C.A. (1990) ‘Electromigration-a tutorial introduction’, *International Journal of Electronics*, Vol. 69, No. 3, pp.333–338.
- Crook, D.L. (1979) ‘Method of determining reliability screens for time dependent dielectric breakdown’, *Reliability Physics Symposium, 1979, 17th Annual*, pp.1–7.
- Gaisler Research (2008) *LEON3 SPARC V8 Processor core*, Available at: <http://www.gaisler.com>.
- Goloubeva, O., Rebaudengo, M., Reorda, M.S. and Violante, M. (2003) ‘Soft-error detection using control flow assertions’, *DFT '03: Proceedings of the 18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, Washington, DC: IEEE Computer Society, p.581, ISBN 0-7695-2042-1.
- Guerin, C., Huard, V. and Bravaix, A. (2007) ‘The energy-driven hot-carrier degradation modes of nMOSFETs’, *IEEE Transaction on Device and Materials Reliability*, Vol. 7, No. 2, p.225.
- Lee, K., Shrivastava, A., Issenin, I., Dutt, N. and Venkatasubramanian, N. (2006) ‘Mitigating soft error failures for multimedia applications by selective data protection’, *CASES '06: Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, New York, NY: ACM, pp.411–420, ISBN 1-59593-543-6. doi, Available at: <http://doi.acm.org/10.1145/1176760.1176810>.
- Lu, D.J. (1982) ‘Watchdog processors and structural integrity checking’, *IEEE Transactions on Computers*, Vol. 31, No. 7, pp.681–685.
- Majzik, I., Pataricza, A., Cin, M.D., Hohl, W., Hönig, J. and Sieh, V. (1994) ‘Hierarchical checking of multiprocessors using watchdog processors’, *European Dependable Computing Conference*, pp.386–403, Available at: citeseer.ist.psu.edu/majzik94hierarchical.html.
- Michel, T., Leveugle, R. and Saucier, G. (1991) ‘A new approach to control flow checking without program modification’, *Fault-Tolerant Computing, 1991. FTCS-21. Digest of Papers, Twenty-First International Symposium*, pp.334–343.
- Mueller, M., Alves, L.C., Fischer, W., Fair, M.L. and Modi, I. (1999) ‘RAS strategy for IBM S/390 G5 and G6’, *IBM Journal of Research Development*, Vol. 43, Nos. 5/6.
- Ragel, R.G. (2006) ‘Architectural support for security and reliability in embedded processors’, PhD Thesis, University of New South Wales, Sydney, Australia.
- Schuette, M.A. and Shen, J.P. (1987) ‘Processor control flow monitoring using signed instruction streams’, *IEEE Transaction on Computer*, Vol. 36, No. 3, pp.264–277, ISSN 0018-9340.
- SPARC (1992) *The SPARC Architecture Manual V8*, Available at: <http://gaisler.com/doc/sparcv8.pdf>.

- Standard Performance Evaluation Corporation (SPEC) (2000) *SPEC CPU2000 V1.3*, Available at: <http://www.spec.org>.
- Stechele, W., Bringmann, O., Rolf, E., Herkersdorf, A., Hojenski, K., Janacik, P., Rammig, F., Teich, J., Wehn, N., Zeppenfeld, J. and Ziener, D. (2007) 'Concepts for autonomic integrated systems', *Proceedings of edaWorkshop07*, Munich, Germany.
- US-CERT (2008) *Vulnerability Notes Database CERT Coordination Center*, Available at: <http://www.kb.cert.org/vuls/>.
- Ziegler, P. and Benz, B. (2005) *Fliegendes Rechnernetz*, In *c't (magazin fur computertechnik)*, Heise Verlag.