Off-Chip Control Flow Checking of On-Chip Processor-Cache Instruction Stream*

Federico Rota, Shantanu Dutt and Sahithi Krishna
Dept. of ECE, University of Illinois-Chicago

Abstract

Control flow checking (CFC) is a well known concurrent checking technique for ensuring that a program's instruction execution sequence follows permissible paths. Almost all CFC techniques require direct access to the CPU-cache bus, meaning that the checking hardware (generally called a watchdog processor (WP)) has to be on-chip. However, an on-chip WP directly accessing the CPU-cache bus has a few disadvantages chief among them being that it will use up appreciable chip real estate of a commodity processor, but may be unnecessary in most environments that do not have significant transient error rates. On the other hand, if an off-chip CFC technique can be developed that imposes minor hardware overheads on the processor chip, then such a WP can be plugged onto the external system bus when needed for concurrent checking, and will have very little of the disadvantages of on-chip WPs. Such an off-chip WP, however, will not generally be able to monitor all instructions due to the bandwidth difference between the CPU bus and the system or memory bus. We present techniques that allow generally effective off-chip CFC using partial access to the instruction execution stream that respects the CPU/system bus bandwidth factor (ratio) K, and still achieve reasonable block-level instruction error coverage ranging from 70-80% for $K = 5$ to about 94% for a $K = 2$. Furthermore, our experimental results show that the program-level error coverage is almost 100% even for $K = 5$ (i.e., we will almost always detect the presence of an instruction error in a program sooner or later before it completes execution, which is useful for fail-safe operation), underscoring the efficacy of our methods.

1: Introduction

Computers and processor chips are ubiquitous in almost all aspects of human endeavor, from everyday living (e.g., embedded systems in VCDs and PDAs) to business transactions to engineering and scientific applications to mission- and life-critical systems. At the same time, the complexity of processor chips are increasing every few months and their components are reducing in feature sizes as dramatically. Current feature sizes have dropped below the 100 nanometer level and significant efforts are being undertaken in research on nano-electronics, which will take feature sizes to the next level of compaction—nanometers. Voltage levels for these systems have also dropped significantly from the classical 5V to to its current levels of 1.1V and lower. These developments augur well for more complex and faster processing of processor chips. However, as feature sizes decrease, the likelihood significantly increases of radiation disturbances such as alpha-particle or electromagnetic (EM) pulses changing voltage values in memory cells or interconnects and/or affecting transistor operation. Reduction of voltage levels also means that digital circuits are more susceptible to external or internal noise like crosstalk causing logic or delay faults. These fault sources mainly cause *transient faults* that remain for the duration of the radiation source. The majority of computer failures originates in transient faults, and a high portion of these faults is manifested as disturbance in the program control flow [7]. An efficient concurrent check of the correct control-flow execution of a program is thus an important requirement for *fault-tolerant* (correction of errors or recovery from errors in a manner enabling correct system performance) or *fail-safe* (shutting down safely in the presence of irrecoverable errors) operation of computer systems.

*This work was supported by the U.S. Department of Defense under MURI grant F49620-01-1-0436.

Proceedings of the 21st IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT'06)
0-7695-2706-X/06 $20.00 © 2006 IEEE

Control flow checking (CFC) is a well-known concurrent checking technique for ensuring that program flow traverses permissible paths. It has been extensively studied for applications in fault tolerance [4, 5, 8, 9, 10, 11, 12, 13, 14, 16] and computer security [1, 2, 18]. CFC techniques are either implemented in hardware using an on-chip monitor or watchdog processor (WP) [2, 4, 5, 8, 9, 10, 11, 13, 14, 18] (or equivalently, an off-chip WP in systems without on-chip caches) or in software [1, 12, 16]. Hardware implemented CFC using on-chip WPs have the advantages of potentially low performance overheads, but also the disadvantage of imposing appreciable processor chip-area overhead that can indirectly lower performance (some high-performance enabling hardware has to be left out when the die area is a constraint) or result in higher cost (due to increased die area). Software-based CFCs can work with any commodity processor but have the disadvantage of high performance overheads, as well as vulnerability to security attacks [18]. In this paper, we develop techniques for off-chip hardware-based CFC that have little of the above disadvantages of on-chip and software techniques, but have most of their advantages. However, in basic off-chip CFC, there is some reduction in fine-granularity error coverage, though program-level error coverage is almost 100%. We also propose additional techniques that are able to alleviate the decrease in fine-granularity error coverage, and establish theoretical results on this issue. To the best of our knowledge, this is the first time that off-chip CFC techniques are being proposed for systems with on-chip caches.

The rest of this paper is organized as follows. Sec. 2 discusses relevant background material in CFC, and Sec. 3 briefly describes our implementation of an on-chip WP for performing CFC for a Motorola 68040. In Sec. 4 we present novel techniques for performing CFC with an off-chip WP. In Sec. 5, the simulation platform and experimental results are discussed, and Sec. 6 concludes the paper.

2: Basics of Control Flow Checking

A *watchdog processor (WP)* is a relatively small and simple co-processor used to perform concurrent system-level error detection by monitoring the behavior of program execution on the main processor (CPU). Being much simpler than the processor being tested, the WP approach is significantly more inexpensive than system level redundancy techniques like duplication. A WP can also be added incrementally to an existing processor system without requiring any significant modification to it. The general configuration of a system with a WP is shown in Fig. 1.

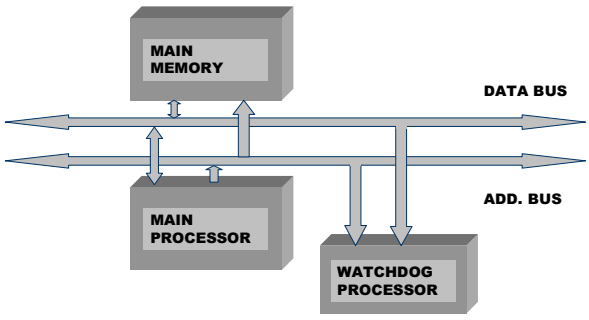


Figure 1. *General structure of a system with a watchdog processor.*

One behavioral aspect of the CPU that can be concurrently monitored by the WP snooping onto the instruction-fetch bus (the internal CPU-cache bus in the case of on-chip caches or the external memory or system bus in the case of cache-less or external-only cache systems) is whether control flows of programs being executed are along permissible paths. The control flow of a program can be represented by a control flow graph (CFG) in which the *nodes* or *blocks* represent a program unit with only one branch/jump instruction at the end, and only one entry point at the beginning. while *arcs* represent the syntactically allowed flow of control between blocks. An example of a CFG of a program is shown in Fig. 2(a). In this figure, valid control flows are $N_1 \rightarrow N2 \rightarrow N4$ or $N_1 \rightarrow N2 \rightarrow N3 \rightarrow N2$. A correct control flow is a fundamental part of the correct execution of computer programs. Monitoring the control flow of a program to ensure that flows between the blocks is in accordance to syntactically allowed sequences is called control-flow checking (CFC). A large variety of faults and their corresponding errors can be detected by CFC. These include memory and cache faults, system-bus and cache-bus faults, processor faults (ALU, register file and control unit faults that affect computation of instruction addresses).

Proceedings of the 21st IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT'06)
0-7695-2706-X/06 $20.00 © 2006 IEEE

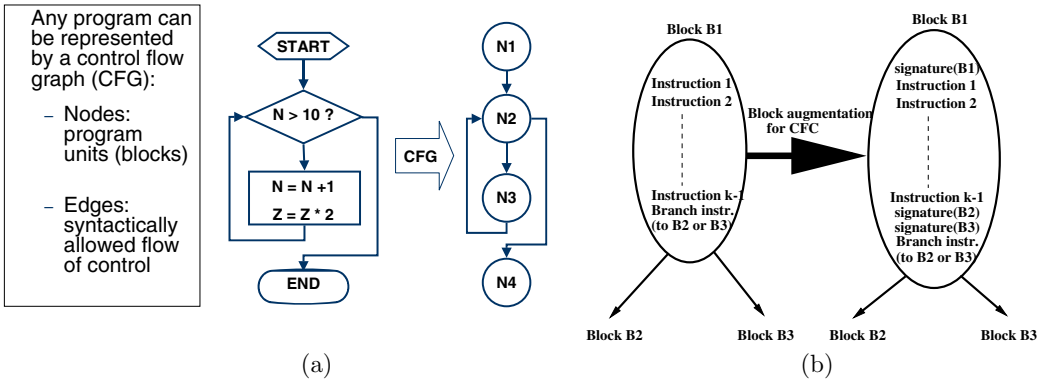


Figure 2. *(a) The control flow graph (CFG) of a simple program. (b) A block in a control flow graph and its possible augmentation for the purpose of control-flow checking.*

The most widely used techniques determine *signatures* as labels of the nodes of the CFG to determine the control flow. The WP is provided with the reference signatures and the relationships between them (represented by the CFG). During run-time, the WP monitors the program run and concurrently computes the signatures which are then compared to the reference ones provided earlier. If a discrepancy is detected then an error is signaled. From this basic idea, different methods have been developed that differ from each other in the definition of a node, in the representation of the reference information and in the derivation of the run-time information. Figure 2(b) shows a possible augmentation of a basic block of a program for the purpose of CFC. Each "signature instruction" can be one or two NOPs containing the signature in the non-opcode fields or some other unused instruction with an immediate data field that can be occupied by the signature.

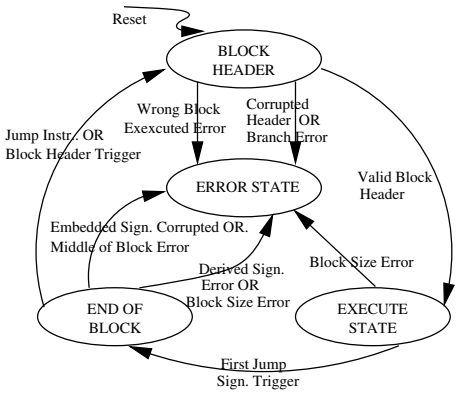


Figure 3. *Main state transition graph of the on-chip WP.*

The reference CFG can be provided to the WP in three different ways [10]. The approach we use is similar to one of these methods: The control flow graph with the signature of each block is stored in the local memory of the WP. If a run-time signature is received or computed by the WP then the stored reference is searched to determine whether the signature is a valid successor to the previous one. There are also two different ways of determining the signatures of each block of the CFG, *assigned signatures* and *derived signatures* [10]. In assigned signature techniques, the signatures of the blocks are assigned arbitrarily using prime numbers or random selection. Assigned signature based methods check only that the blocks are executed in an allowed sequence. In derived signature techniques, the block signatures are derived from the instruction sequence of the block by encoding schemes like mod-2 addition, checksum or linear feedback shift register (LSFR). The run-time signatures are also computed concurrently by the WP by access to instructions fetched by the CPU. The WP-computed signatures are compared to its reference information to perform the checking. A major advantage of derived signatures is that besides performing CFC, they also allow the checking of the integrity of the instructions themselves within a block.

3: Basic On-Chip WP for CFC and Address Range Checking

We have developed a basic WP using run-time derived signatures for CFC that directly monitors the instructions executed by a Motorola 68040 CPU. This basic WP can be thought of as either directly monitoring the off-chip CPU-cache bus of a processor without on-chip caches or equivalently as an

Proceedings of the 21st IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT'06)
0-7695-2706-X/06 $20.00 © 2006 IEEE

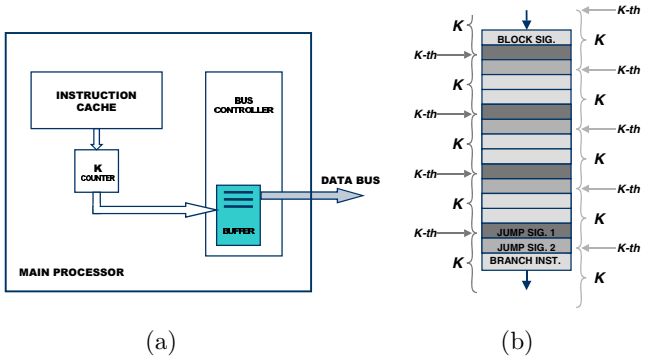
on-chip WP directly monitoring the bus between the CPU and the first-level on-chip cache—in both scenarios the CPU instruction execution stream is directly monitored by the WP. We will assume the latter configuration in the rest of the paper.

To improve error coverage beyond that provided by classical CFC, we also implemented address range and sequence checking features in the WP. We assume that the program is allocated such that the code, made of instructions and immediate data, is stored in an area of memory that is disjoint from the area for data. Under this assumption, every instruction fetch address should be in the code range. Further, instruction fetches, including immediate-data fetches, within a block should occur at consecutive addresses. Our WP also checks that all accesses (instruction and data) are within the used memory range. Figure 3 shows the main state transition diagram of our WP; see [15] for further details.

4: Off-Chip CFC for CPUs with On-Chip Caches

There are several reasons for preferring or even needing a CFC monitoring WP to be off-chip:

1. Designers do not want to use chip real-estate for concurrent testing or fault tolerance features.
2. It may not be necessary to use concurrent testing in "normal" environments where external EMI and alpha particle radiations are extremely low. Concurrent testing would be desirable, indeed, necessary, in significant radiation environments like space or military installations. However, it would not be cost-effective to have chip real estate devoted to concurrent testing on commodity processors when a majority of them will not be used in radiation-hazard environments.



(a) (b)

Figure 4. *(a) The small on-chip hardware for sending 1 of K fetched instructions to the off-chip WP. (b) Partition of a block into K subsets of regular instructions, one of which is ent to the WP based on the counter value at the beginning of the block.*

Having an off-chip WP to monitor the CPU-cache instruction fetch traffic is thus a viable design option for CFC in computer systems using minorly-modified commodity processors. We assume that the CPU is non-pipelined—in a pipelined CPU, our techniques can be combined with those of [5] to provide a solution to both issues.

In our technique, the cache controller explicitly sends to the WP information about the instructions that are being fetched by the CPU from cache. These instructions can be sent to the WP over the system bus when it is not being used. It may also be possible to shield from radiation as well as dynamically isolate the (possibly short) portion of the system bus between the CPU and the WP so that no further errors are introduced in this communication; we consider this option in our simulations.

In general, it will not be possible to send all the instructions from the processor chip to the WP because the system bus over which the instructions are sent to the WP has a much slower clock than the CPU clock. E.g., a 3.06 GHz P4 processor has a system clock of 133 MHz, and with "quad-pumping"—accessing 4 data items in one cycle—its effective bus speed becomes 533 MHz, giving a bandwidth factor (ratio) of $3060/533 \approx 5.5$. In this context, we define the *Bandwidth Factor (K)* as the ratio of the bandwidths of the 1st-level cache bus to the system bus:

$$K = \frac{\text{(Processor Clock frequency)} \times \text{(Cache Bus Width)}}{\text{(Bus Clock frequency)} \times \text{(System Bus Width)} \times \text{(Cycle BW)}}$$

where the "Cycle BW (Bandwidth)", also called the "pump factor", is the number of data transfers that can be effected over one bus cc—current Intel P4 and AMD processors have cycle BW's of two to four; see, e.g., [17].

Due to the system-bus bandwidth constraint, we send only one out of every K "regular" instructions fetched by the CPU to the WP plus all signature instructions in each executed block; the concept is

illustrated in Fig. 4(b). Note that besides the 1 out of K regular instructions, it is strictly necessary to send the signature instructions in order for the WP to perform basic CFC. Thus the instructions are sent to the WP at a rate slightly higher than the system bus bandwidth, and we need a small buffer to absorb this mismatch. The high-level block diagram of the design is shown in Fig. 4(a). However, for simplicity, in our current implementation, instead of having a buffer, we adjust the ratio of the CPU and system clock bandwidths to be lower than K (the real ratio) to accommodate sending to the WP 1 out of K instructions plus the three signatures for each block. The effect is similar to having an actual buffer. We also have a mod$K$ counter that counts from 1 to K every instruction fetch, and whenever the counter value is 1 (i.e., whenever $i \bmod K + 1 = 1$, where i is the instruction count of fetched instructions starting from the beginning of the program), the cache controller sends the currently fetched instruction to the WP.

4.1: Hybrid Signatures

As described above, the WP will always receive the three basic signatures in each block that signal a block's arrival and the two possible successor blocks. However, for off-chip CFC, these signatures can no longer be derived, since which subset of instructions are sent to the WP is dynamic, based on different counter alignments for different visits to the block. Thus the basic signatures of each block are assigned. However, it is still possible to check the integrity of the sent instructions within a block by performing a type of 1-out-of-K derived signature checking. Though we do not know a-priori which subset of instructions will be sent to the WP when a block is visited, we do know the K possible subsets. We thus have K possible derived signatures, and the computed signature based on the sent instructions for any instance of a block visit is compared for a match to any of the K derived signatures. A match indicates instruction integrity with high probability (assuming that the Hamming distance between the K derived signatures is large), while a mismatch indicates the presence of erroneous instruction(s).

4.2: Loop Lengths, Relative Primality of K and Error Coverage

Most programs and hence their CFGs have directed loops; see Fig. 2(a). Loops afford a means to ameliorating the decreased instruction checking in off-chip CFC, since revisiting a block B_i means that it may be possible that a different set of instructions of B_i is sent to the WP compared to the previous visit to B_i. This increases the number of instructions monitored for B_i and thus increases error coverage albeit with increased latency. In fact, if the number of instructions in a loop is relative prime w.r.t. K, then it can be shown that in K consecutive iterations of this loop, all instructions in it will be checked by the off-chip WP.

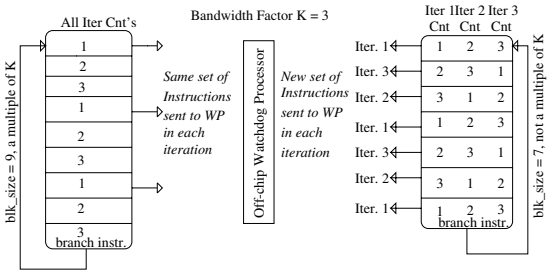


Figure 5. *Instruction monitoring by the WP for two self-looping blocks (simple loops), one of length a multiple of the bandwidth factor $K = 3$ and another of length relatively prime to K.*

Theorem 1 *Let $\mathcal{L} = B_1, B_2, \ldots, B_t, B_1$ be a loop of blocks in the CFG of a program and let $l = \sum_{i=1}^{l} |B_i|$ be the length (size) of $\mathcal{L}$, where $|B_i|$ is the number of instructions and signatures in B_i. If either (a) K is prime and $l \neq mK$ for any integer $m \geq 1$ or (b) more generally, $\gcd(K, l) = 1$ [1] (i.e., K and l are relatively prime), then in K consecutive executions of this loop, all instructions in $\mathcal{L}$ will be sent to the WP. Thus if none of the blocks in $\mathcal{L}$ have been replaced from the (1st level) cache any time during the K iterations, then any error(s) in any instruction or signature in $\mathcal{L}$ will be detected by the WP barring fault-masking situations (which have extremely low probabilities since each bit error has to line up with another bit error in a unique instruction).*

Proof: We give the proof for the general condition (b) which includes condition (a) as a special case. Let $l = mK + r$, $m \geq 0, 0 < r < K$ and without loss of generality let the counter value when the header signature of B_1 at the top of the loop $\mathcal{L}$ is fetched in the 1st iteration be 1. Thus instructions of $\mathcal{L}$ at a distance of jK, $\forall j, 1 \leq j \leq m$ from this header signature will be sent to the WP. In the beginning

[1] gcd stands for *greatest common divisor*.

of subsequent iterations i, $2 \leq i \leq K$ the value of the counter when the header signature of B_1 is reached, the counter value will be $((i-1)mK + (i-1)r) \bmod K + 1 = (i-1)r \bmod K + 1$. We claim that all $[(i-1)r \bmod K] + 1$ values will be unique. If some two different iterations p, q $2 \leq p, q \leq K$, $p > q$ produce identical counter values, it means $(p-1)r \bmod K = (q-1)r \bmod K$, which implies that $(p-q)r = sK$ for some $s \geq 1$. Now $\gcd(l, K) = 1 \Rightarrow \gcd(mK + r, K) = 1 \Rightarrow \gcd(r, K) = 1$. Thus, since r and K have no factors in common, $(p-q)r = sK$ implies that $(p-q) = xK$ for some $x \geq 1$. However, since $p, q \leq K$, $p - q < K$, and thus $p - q = xK$ is impossible, and we reach a contradiction. Thus all $[(i-1)r \bmod K] + 1$ values are unique, and by the pigeon hole principle, these values will cover all integers in the range $[2 \ldots K]$. This means that all instructions in $\mathcal{L}$ will be sent to the WP over K iterations of $\mathcal{L}$. $\diamondsuit$.

Figure 5 shows a simple loop of a single block on the right of length 7 that satisfies the condition of the theorem, and as shown, over $K = 3$ iterations every instruction in it gets a counter value of 1 and is thus sent to the WP. The single-block loop on the left, on the other hand, has a a length of 9, a multiple of 3. As a result, every time the loop is executed the counter value remains the same for each instruction. Thus the same instructions are checked by the WP every iteration, and there is no increase in instruction error coverage. This problem can be alleviated by adding a NOP to the block, thereby making its length relatively prime to K. The above discussion suggests a NOP insertion approach for increasing off-chip CFC error coverage, an example of which is shown in Fig. 8 for matrix multiplication; it helps increase error coverage significantly as we discuss in Sec. 5.

5: Simulation Environment and Experimental Results

All hardware modules in our system were described in VHDL with a mixture of structural and behavioral descriptions; the core Motorola 68040 VHDL description was obtained from [3].

F. Frequency f_{fault}	F. Period T_{fault}	F. duration t_{fault}
500 Hz	20000 cc	5/10/15 cc
1 KHz	10000 cc	5/10/15 cc
2 KHz	5000 cc	5/10/15 cc
80 KHz	125 cc	32 cc
160 KHz	63 cc	16 cc
320 KHz	32 cc	8 cc

Table 1. *Fault frequencies and durations*

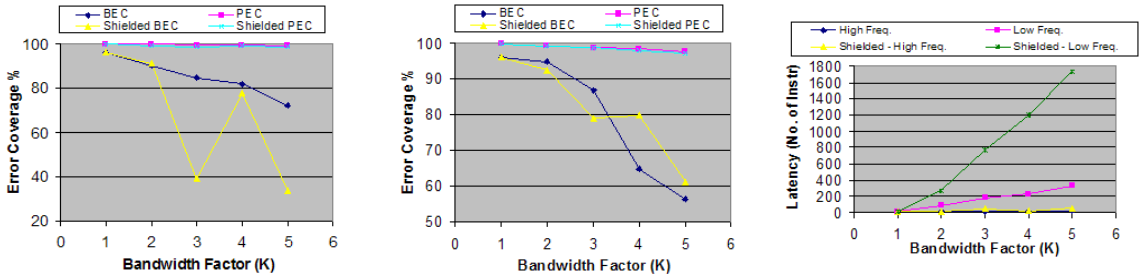
A major goal of our experiments is determining the effect of deliberate electromagnetic interference (EMI) on digital and computer systems; the interference pattern is expected to be a train of EM pulses for some duration followed by a silent period, with the pattern repeating at some frequency dependent on the power capability of the interfering source. We thus have fault injection parameters f_{fault}—rate at which new faults are periodically injected and *fault duration* t_{fault}—the time for which an injected fault is present; see Table 1. Off chip wiring like system buses are especially susceptible to EMI. We thus focus our fault injections primarily on the system data bus[2]. Our fault injector injects the following patterns of stuck-at-0/1 faults: i) two random faults, ii) one cluster of 2 faults, iii) four random faults, iv) two clusters of 2 faults, v) one cluster of 4 faults, vi) six random faults, and vii) one cluster of 6 faults. For each of these fault patterns, in our simulations we sweep through the different values of fault frequencies and durations reported in Table 1. We classify the first three frequencies as low and the others as high.

All our experiments were conducted for a matrix multiplication program that multiplies a 3×2 with a 2×5 matrix, has 9 blocks and executes 1353 instructions (including signature instructions) for a successful computation. We report *fault latency*, which is the average delay, in units of average instruction execution time, across all runs between the first instruction error and the detection of an error in each run, and two different *error coverage metrics*, versus different values of the bandwidth ratio K. Note that the results for $K = 1$ for an off-chip WP are equivalent to the results for an on-chip WP, as both WPs will see all the instructions executed by the CPU. The two different error coverage (EC) metrics are:

1. *Block level error coverage (BEC)*: is the weighted % of erroneous blocks that are detected by the WP (using CFC and address-range checking). For a *simulation run*[3] j, let $B_1^j, B_2^j, ..., B_k^j$ be

[2] Address bus errors are very easily caught and thus not very interesting.

[3] A simulation run is an execution of an application program one or more times until an error in it is detected. We performed 100's of simulation runs for each parameter combination (fault pattern, frequency and duration) to obtain reliable results.



(a) (b) (c)

Figure 6. *(a-b) Average error coverages and (c) average latencies over all fault numbers and patterns versus the bandwidth factor K for both unshielded and shielded bus communication betwen CPU and WP for (a) high fault frequencies, and (b) low fault frequencies.*

the sequence of blocks (not necessarily unique) executed before an error is reported in block B_k^j. Then the coverage definition is:

$$\text{BEC} = \sum_{j=1}^{r} e_k^j / \sum_{j=1}^{r} (e_1^j + e_2^j + ... + e_k^j)$$

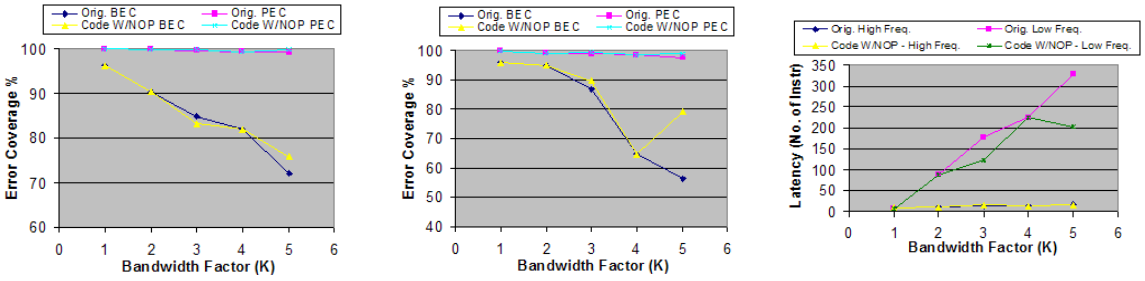
where r is the number of runs for the parameter combination of interest and e_i^j is the number of erroneous instructions in block B_i^j.

2. *Program level error coverage (PEC)*: is the % of program executions that have errors in any instruction field including immediate data that are detected as erroneous by the WP at some point in the program execution.

PEC is a coarse granularity error coverage metric, and is mainly useful to estimate how effective our techniques are in at least having a *fail-safe* termination of an erroneous program (assuming its output data is used only after it terminates successfully). BEC is a finer granularity metric and measures the accuracy of error detection at the block level.

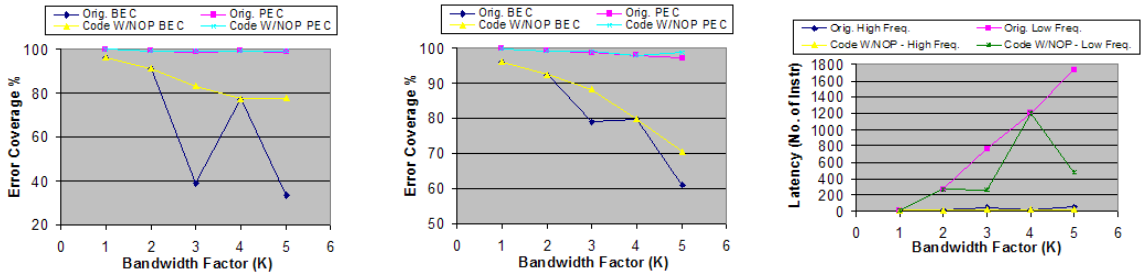
i) Results for an unshielded bus without program modification: Figure 6(a-b) demonstrates how our basic off-chip WP techniques (without any code modifications for obtaining relative primality of loop lengths w.r.t. K) perform for different values of the bandwidth factor K. For a CPU $\rightarrow$ WP communication over an unshielded system bus, BEC ranges between 70-93% for high frequencies and 56-94% for low frequencies, with the coverages decreasing with increasing K. This phenomenon can be explained as follows. The probability that the set of e erroneous instructions in a block will not be sent to the WP (leading to non-detection of basic instructions errors in the block) is $(1 - \frac{1}{K})^e$ (the *skipping probability*) and it increases with K; we call this the *skipping effect*. Furthermore, fault duration being equal, e is proportional to the fault frequency. Thus at low fault frequencies the skipping probability is higher than at high frequencies. The effect of this is seen in Figs. 6(a-b), which show that the BEC decrease with K is sharper for low frequencies (Fig. 6(b)) than for high frequencies (Fig. 6(a)), barring the jarring BEC decrease in Fig. 6(a) for a shielded bus for $K = 3, 5$ (which is discussed later). A good news is that PEC is close to 100% for all Ks, meaning that our off-chip CFC techniques are very effective for the fail-safe goal. Our basic techniques are reasonably effective for the BEC metric, but will be rendered more effective in combination with program modification methods, as we shortly discuss.

ii) Results for a shielded bus without program modification: Figure 6(a-b) also show the BECs for a shielded bus—note that the shielded bus is assumed to be an electrically isolatable and small part of the unshielded system bus. A shielded bus has the advantage of not having additional errors in instructions that can *mask* already present errors in the same or other instructions of the block. An unshielded bus, on the other hand, has the advantage of getting around the skipping effect by having the possibility of errors introduced in instructions sent to the WP, so that an originally erroneous block gets detected even if the originally erroneous instructions are not sent to the WP (on the flip side there can be false alarms—errors detected in blocks that are not originally erroneous, though such events are rare). We call this the *anti-skipping effect*. In an unshielded bus, the anti-skipping effect is more prominent than masking, especially at high fault frequencies. These trends are borne out in Figs. 6(a-



(a) (b) (c)

Figure 7. *Results for an unshielded bus between the CPU and WP: (a-b) Error coverage changes and (c) latency changes, after block modification via NOP insertion for obtaining loop sizes for which K is relatively prime, for (a) high fault frequencies and (b) low fault frequencies.*



(a) (b) (c)

Figure 9. *Results for a shielded bus between the CPU and WP: (a-b) Error coverage changes and (c) latency changes, after block modification via NOP insertion for obtaining loop sizes for which K is relatively prime, for (a) high fault frequencies and (b) low fault frequencies.*

b), which show that the shielded-bus BEC is generally lower that the unshielded-bus BEC, especially in Fig. 6(a). The dramatic decrease in BEC for a shielded bus for $K = 3, 5$ in Fig. 6(a) is addressed further below. The PEC for the shielded bus is almost the same as the unshielded case and close to 100%.

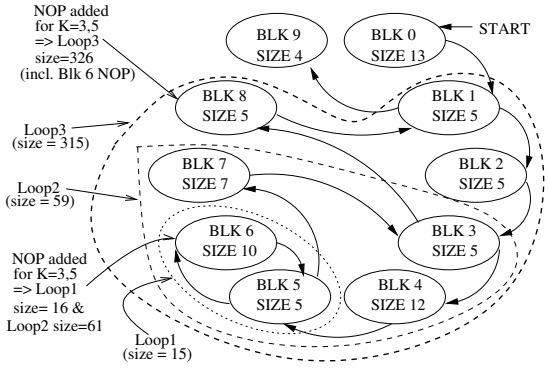


Figure 8. *The CFG, block and loop sizes of the tested matrix multiplication program, and NOP insertions for obtaining relative primality of loop sizes w.r.t. $K = 3, 5$.*

iii) Results for program modification via NOP insertion: The significant BEC loss for a shielded bus for $K = 3, 5$ shown in Fig. 6(a) is the result of errors in the loop-control instructions in the inner loop of our matrix multiplication code (blocks 5 and 6 in Fig. 8), which resulted in an infinite loop situation. Further, this loop's size is 15, a multiple of both 3 and 5, and thus for these K values, the corrupted instructions were never sent to the WP in spite of a very large number of erroneous iterations (up to 600) of this loop (the correct number is two), thus leading to a significant decrease in BEC[4]. When we insert a NOP instruction in block 6, the new inner loop size, 16, is relatively prime w.r.t. $K = 3, 5$. So in each iteration of the modified inner loop, a new subset of instructions is sent to the WP (see Fig. 5) and an error in the corrupted block is detected within K iterations. This dramatically improves error coverage as depicted in Figure 9(a).

Figure 8 shows the CFG of the matrix multiplication program and its block and loop sizes for multiplication of a 3×2 with a 2×5 matrix. The figure also shows the two block in which NOPs

[4]In the case of an unshielded bus, this situation was obviated by the the anti-skipping effect.

are inserted to make loop sizes relatively prime for $K = 3, 5$. The original loop sizes are already relatively prime for $K = 2, 4$; thus no NOP insertions are needed in these cases and the coverage is high. Figure 9(a-b) emphatically shows the dramatic improvement in error coverages that about double after NOP insertions for $K = 3, 5$ for a shielded bus. Figure 7(a-b) also shows appreciable improvement in BEC for the unshielded bus for $K = 3, 5$ except for the small glitch for $K = 3$ in Fig. 7(a). The improvements in the unshielded bus case is smaller than for a shielded bus, as the anti-skipping effect of the former had already prevented significant deterioration in BEC, so there was not that much room for improvement via NOP insertion. The BEC increase in the shielded bus case ranges from 11% to 132%. These results give empirical support to Theorem 1 and to the concept of making loop sizes relative-prime w.r.t. K.

6: Conclusions

We have developed novel techniques for off-chip CFC of a processor with on-chip caches—the first time such techniques have been developed to the best of our knowledge. Our techniques include: a) sending a subset of instructions in a block from the CPU to the off-chip WP—the subset size depends on the bandwidth mismatch K between the CPU-cache bus and the system bus, b) the concept of K alternate derived-signatures per block, and c) the concept of NOP insertion for obtaining relative primality of loop sizes with respect to K—this facilitates checking by the WP of almost all instructions of each loop over multiple loop iterations. The results show effective block-level error coverage (BEC) of 80-93% for high fault frequencies and of 70-94% for low fault frequencies, and almost 100% program level error coverage (meaning that error(s) in the program are detected at some point in its execution, even if each erroneous block is not detected during the block's execution), which facilitates reliable fail-safe operation. These results underscore the overall efficacy of our techniques, and thus makes it viable for commodity processor systems (with minor overheads for enabling off-chip CFC) to be plugged in with an external WP when concurrent error checking is desired.

References

[1] M. Abadi, et al., "Control-flow integrity", *Proc. 12th ACM Conf. on Computer & Comm. Security*, pp. 340-353, 2005.

[2] S. Arora et al., "Secure embedded processing through hardware-assisted run-time monitoring", *Proc. Design Automation & Test in Europe*, 2005.

[3] A. Benso, et al., " Model of the MC 68040 developed at Politecnico di Torino", http://www.polito.it.

[4] Y-Y. Chen, "Concurrent Detection of Control Flow Errors by Hybrid Signature Monitoring", *IEEE Trans. Comput.*, Oct. 2005 pp. 1298-1313.

[5] X. Delord and G. Saucier, "Formalizing signature analysis for control flow testing of pipelined RISC microprocessors", *Int'l. Test Conference*, pp. 936-945, 1991.

[6] D. Gil, P. Radeva, J. Saludes and J. Mauri, "A study of the effects of transient fault injection into the VHDL model of a fault-tolerant microcomputer system", *On-Line Testing Workshop*, pp.73-79, 2000.

[7] U. Gunneflo, J. Karlsson, J. T.: Evaluation of error detection schemes using fault injection by heavy-ion radiation. *19'th Int'l. Fault Tolerant Computing Symp.*, 1989.

[8] X. Li, J-L. Gaudiot, "A Compiler-Assisted On-Chip Assigned-Signature Control Flow Checking", *Proc. Asia-Pacific Comp. Systems Arch. Conf.*, 2004, pp. 554-567.

[9] D. J. Lu, "Watchdog processors and structural integrity checking", *IEEE Trans. on Comp.*, Vol. 31, pp. 681-685, 1982.

[10] A. Mahmood and E. J. McCluskey, "Concurrent Error Detection Using Watchdog Processors - A Survey", *IEEE Trans. on Comp.*, Vol. 37(2), pp. 160-174, 1988.

[11] T. Michel, R. Leveugle and G. Saucier, "A New Approach to Control Flow Checking without Program Modification", *21'st Int'l. Fault Tolerant Computing Symp.* pp. 334-341, 1991

[12] N. Oh, P.P. Shirvani, and E.J. McCluskey, "Control-flow checking by software signatures", *IEEE Trans. Reliab.*, pp. 111-122, March 2002.

[13] J. Ohlsson and M. Rimen, "Implicit Signature Checking," ftcs, *Proc. 25'th Int'l. Fault-Tolerant Computing Symp.*, 1995.

[14] 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," *22'nd Int. Fault-Tolerant Computing Symp.* pp. 316-325, 1991.

[15] F. Rota, *Control Flow Checking Using Main Memory Bus Monitoring in an Internal Cache Environment*, M.S. Thesis, University of Illinois at Chicago, Dec. 2002.
Available at: http://www.ece.uic.edu/~dutt/theses/federico-thesis-03-03.pdf

[16] R. Venkatasubramanian, J. P. Hayes, and B. T. Murray, "Low-cost on-line fault detection using control flow assertions", *Proc. 9th IEEE Int'l On-Line Testing Symposium*, 2003.

[17] Article on Pentium 4 in *Wikipedia—The Free Encyclopedia*, http://en.wikipedia.org/wiki/Pentium_4.

[18] T. Zhang et al., "Anomalous path detection with hardware support", *Proc. Int'l Conf. Compilers, architectures and synthesis for embedded systems*, pp. 43 - 54, 2005.