Dynamic Code Duplication with Vulnerability Awareness for Soft Error Detection on VLIW Architectures

JONGWON LEE, Seoul National University

YOHAN KO and KYOUNGWOOL LEE, Yonsei University

JONGHEE M. YOUN and YUNHEUNG PAEK, Seoul National University

Soft errors are becoming a critical concern in embedded system designs. Code duplication techniques have been proposed to increase the reliability in multi-issue embedded systems such as VLIW by exploiting empty slots for duplicated instructions. However, they increase code size, another important concern, and ignore vulnerability differences in instructions, causing unnecessary or inefficient protection when selecting instructions to be duplicated under constraints. In this article, we propose a compiler-assisted dynamic code duplication method to minimize the code size overhead, and present vulnerability-aware duplication algorithms to maximize the effectiveness of instruction duplication with least overheads for VLIW architecture. Our experimental results with SoarGen and Synopsys simulation environments demonstrate that our proposals can reduce the code size by up to 40% and detect more soft errors by up to 82% via fault injection experiments over benchmarks from DSPstone and Livermore Loops as compared to the previously proposed instruction duplication technique.

Categories and Subject Descriptors: B.8.1 [Performance and Reliability]: Reliability, Testing, and Fault-Tolerance; C.4 [Computer Systems Organization]: Performance of Systems—*Fault tolerance*; D.3.4 [Programming Languages]: Processors—*Code generation; Compilers; Optimization*

General Terms: Algorithms, Design, Performance, Reliability

Additional Key Words and Phrases: Embedded systems, VLIW architecture, soft errors, code size, compiler-assisted dynamic code duplication scheme, vulnerability-aware duplication algorithm, temporal vulnerability, physical vulnerability

ACM Reference Format:

Lee, J., Ko, Y., Lee, K., Youn, J. M., and Paek, Y. 2013. Dynamic code duplication with vulnerability awareness for soft error detection on vliw architectures. *ACM Trans. Architec. Code Optim.* 9, 4, Article 48 (January 2013), 24 pages.

DOI = 10.1145/2400682.2400707 <http://doi.acm.org/10.1145/2400682.2400707>

This work was supported in part by the Korea Science and Engineering Foundation (KOSEF) NRL Program grant funded by the Korea government (MEST) no. 0421-2012-0047, the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology (MEST)/Korea Science and Engineering Foundation (KOSEF) grant 2012-0000470, IDEC, the Center for Integrated Smart Sensors funded by the MEST as Global Frontier Project CISS-0543-20110012, and in part by the Basic Science Research Program through NRF funded by MEST 2012-015421 and by the MKE (the Ministry of Knowledge Economy) under the Global Collaborative R&D program supervised by the KIAT M002300089.

Authors' addresses: J. Lee, Department of Electrical and Computer Engineering, Seoul National University, Korea; email: jwlee@sor.snu.ac.kr; Y. Ko, K. Lee (corresponding author), Department of Computer Science, Yonsei University, Korea; email: kyoungwoo.lee@gmail.com; J. M. Youn, Y. Paek, Department of Electrical and Computer Engineering, Seoul National University, Korea.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2013 ACM 1544-3566/2013/01-ART48 \$15.00

DOI 10.1145/2400682.2400707 <http://doi.acm.org/10.1145/2400682.2400707>

1. INTRODUCTION

Several constraints such as performance, code size, area, and power have been posed in designing embedded systems. Besides these constraints, reliability is becoming an important concern for the design of embedded systems [Hu et al. 2009]. This is because technology scaling, which incurs shrunk feature size, decreased voltage level, and reduced noise margin, makes systems more susceptible to transient faults [Baumann 2005; Hazucha and Svensson 2000; Wrobel et al. 2001]. Transient faults, also known as *soft errors*, mainly caused by energetic particles such as alpha particles and neutrons, may result in erroneous program states, incorrect outputs, and eventually system crashes. Unless soft errors are detected, even though they are not permanent and nondestructive, the reliability of a system cannot be ensured any longer. Especially in resource-constrained embedded systems used for medical, financial, and security applications, requiring reliable information, it is extremely important to deliver high reliability by detecting soft errors with least overheads in terms of code size, area, performance, and power [Shrivastava et al. 2010; Mukherjee 2008].

VLIW (Very Long Instruction Word) architectures are popular in embedded systems since they offer the potential for high-performance processing at a relatively low cost and energy usage [Zhong et al. 2005]. Consequently, techniques to improve the reliability of application execution in VLIW processors are of interest [Hu et al. 2005]. Several approaches have been proposed to detect soft errors in VLIW architectures. One of the promising techniques is to duplicate instructions at compile time. The idea of duplicating instructions exploits available resources in VLIW architectures. The lack of instruction-level parallelism in applications unavoidably makes an amount of issue slots unused. Indeed, a number of slots are unused on the average in 4-way VLIW processors [Hu et al. 2009]. These unused issue slots are called *empty slots*. By allocating the duplicated instructions to these empty slots possibly at compile time, comparing the result of an original instruction and that of its duplicate at runtime, and flagging error detection if they are not identical, the reliability of VLIW architectures can be improved [Bolchini 2003; Holm and Banerjee 1992; Hu et al. 2005, 2009].

Unfortunately, all duplicated instructions cannot be allocated to empty slots, which enforces generating additional VLIW packets to include the duplicated instructions. Thus, the increase of code size, another important design concern in embedded systems, due to the extra VLIW packets, is necessarily accompanied with the enhanced reliability. For example, Jie Hu et al. [Hu et al. 2005, 2009] have recently proposed a constraint-induced instruction duplication technique for VLIW architectures. However, their technique can increase the code size by up to 90% for complete instruction duplication. The increase of code size has a negative impact not only on the design constraint but also on system reliability. This is because a large size of code makes more bits present in the system, which leads to a higher soft error rate since the larger exposed, the more vulnerable [Reis et al. 2005c]. Further, their duplication algorithm does not consider different degrees of vulnerability in instructions and thus it might lose the effectiveness of duplicating instructions by unnecessarily duplicating unimportant instructions in terms of reliability. The main reason is because they try to duplicate instructions in a sequential manner without any awareness of instruction vulnerability. Thus, their approach first duplicates early-located instructions of the code especially when the performance or power constraint is relatively small.

In order to minimize code size and to effectively increase reliability with least overheads for duplicating instructions, we propose a novel approach, *compiler-assisted dynamic code duplication method* and *vulnerability-aware duplication algorithms* for VLIW processors. Our proposed VLIW architecture accepts an assembly code composed of only original instructions as an input, and generates duplicated instructions

at runtime with the help of encoded information attached to original instructions. When the compiler generates the assembly code, it is determined whether an original instruction will be duplicated or not at runtime, and then the result of the decision is included in the encoding space of the original instruction. Since the duplicates of original instructions are not explicitly present in the assembly code, the increase of code size due to the duplicated instructions can be avoided in our proposed technique. Also, our compiler-assisted duplication algorithms provide mechanisms considering vulnerability of each instruction so that our approach can offer selective protection under limited budget of power and performance so they can provide higher reliability than the previously proposed techniques unaware of different vulnerability levels of instructions. Our vulnerability-aware duplication algorithms take into account two metrics: (i) *temporal vulnerability* based on the more often executed, the more vulnerable, and (ii) *physical vulnerability* based on the larger cell area (more number of transistors), the more vulnerable.

The contributions and results of this work include the following.

- We present a compiler-assisted dynamic code duplication scheme for VLIW architectures which can reduce the code size significantly.
- We propose vulnerability-aware duplication algorithms which can improve the reliability effectively with minimal costs.
- Our experimental results show that our proposed VLIW architecture is implemented with 3.2% area overhead and no clock cycle penalty as compared to an existing technique.
- Our experimental results demonstrate that our proposals can reduce the code size by up to 40% and detect more soft errors by up to 82% via fault injection experiments over a suite of benchmarks as compared to the previously proposed technique.

2. RELATED WORK

With technology scaling, soft errors are becoming an important design concern in embedded systems. Soft errors have already been revealed to cause fiscal damages [Wang et al. 2004]. For example, Sun blamed soft errors for the crash of their million-dollar line SUN flagship servers in November 2000 [Lyons 2000]. In one incident, soft errors crashed an interleaved system farm. In another incident, soft errors brought a billion-dollar automotive factory to halt every month [Tremblay and Tamir 1989]. Further, highly integrated chip equipped reliability-sensitive embedded devices such as mobile healthcare systems and anti-lock braking systems (ABS) in automotive engine control units (ECU) are significantly threatened by exponentially increasing soft error rates with technology scaled. Thus, it is a necessity to combat soft errors for embedded systems in both emerging and traditional computing environments.

Previous works for coping with soft errors have been based on redundancy. Redundancy has been applied at different levels of granularity, such as hardware level, thread level, and instruction level, etc. Techniques for exploiting n -modular redundancy (nMR) [Mitra et al. 2005] check soft errors with redundant hardware components and thus incur high overheads in terms of area and power consumption [Austin 1999; Meixner et al. 2007]. The appearance of simultaneous multithreading (SMT) capabilities in modern processors gives an opportunity for soft error detection by running two copies of one thread and comparing their outcome [Gomaa and Vijaykumar 2005; Reddy et al. 2006; Reinhardt and Mukherjee 2000]. The drawbacks of these approaches include substantial performance degradation, hardware cost, and power consumption increase.

Several researches have investigated redundancy techniques at instruction level for soft error detection. Unlike aforementioned techniques highly dependent on hardware features, they achieve redundant execution by relying on software techniques, with

little or no hardware cost. As one of the promising compiler-based software approaches for soft error detection, SWIFT [Reis et al. 2005a] duplicates program's instructions, schedules the original and duplicated instruction sequences together in the same thread of control, and inserts explicit validation codes to compare the results from the original instructions and their corresponding duplicates. CRAFT [Reis et al. 2005b] and PROFIT [Reis et al. 2005b] enhance SWIFT approach by leveraging extra hardware structures and applying partial protection based on AVF (Architectural Vulnerability Factor) analysis [Mukherjee et al. 2003], respectively. These approaches provide complete fault coverage with minimal area cost. However, they incur significant performance overhead since the number of instructions can be easily doubled mainly due to full duplication of instructions.

In the context of redundancy at instruction level, Jie Hu et al. [Hu et al. 2005, 2009] propose techniques to mitigate the impact of soft errors on reliability by duplicating instructions in VLIW architectures, which are of our interest. The main idea behind their approach is to fill empty slots (NOP instructions) with duplicated instructions without performance penalty if there exist available empty slots. Otherwise, it copies duplicated instructions at new instruction cycles. As compared to the previous instruction duplication studies [Reis et al. 2005a, 2005b], this approach can also increase the reliability since it can detect soft errors by comparing the output of an original instruction with that of the duplicated instruction. Interestingly, this approach can improve the reliability under constraints of power consumption, performance, and code size by static analysis at compile time. It can trade off reliability at the cost of performance by adjusting the rate of duplicate instructions as opposed to full duplication of instructions in the previous works [Reis et al. 2005a, 2005b]. Therefore, this approach can be exploited in various forms of application requirements from reliability-sensitive to performance-sensitive ones. Although this approach, called *static code duplication scheme* in this article, is very promising in the area of the instruction-level redundancy, it has two primary drawbacks: (i) *the increase of code size* and (ii) *unawareness of different importance among instructions*. In contrast, our approach does not incur the code size overhead and considers different levels of vulnerability in instructions when duplicating instructions.

Recently, several selective protections have been proposed to increase the reliability. Rehman et al. [2011] propose reliability-aware code transformation techniques to duplicate instructions under the performance constraint. Their techniques are very promising to increase the reliability with least performance overhead by transforming the codes while our approach presents a dynamic code duplication to reduce the code size and to consider instruction vulnerability for duplication. Note that their techniques are orthogonal to our dynamic code duplication and thus theirs can be applied for our proposed architectures. Nakka et al. [2007] present processor-level selective replication instead of entire replication. This methodology can improve the performance because their compiler can ignore benign errors. Their scheme is exploited at processor-level selective replication while ours is at the instruction-level selective duplication. Borodin and Juurlink [2010] propose efficient instruction duplication scheme by exploiting pre-computation and memoization. It improves the performance and fault coverage of permanent faults. However, their technique deals with permanent faults while our technique is presented for transient faults such as soft errors in VLIW architectures. There have also been several researches for out-of-order processors to dynamically generate code against soft errors [Wang and Patel 2006; Soundararajan et al. 2007; Vera et al. 2009]. Among these approaches, Vera et al. [2009] propose a selective replication scheme close to our approach with the perspective of considering the vulnerability of instructions. They protect only a subset of instructions, the most vulnerable ones, by selectively replicating those with higher vulnerabilities than a predefined vulnerability

Fig. 1. Our VLIW architecture for dynamic code duplication.

threshold. In their scheme, vulnerability of instructions is estimated based on the cell area they occupy and the time they spend in the issue queue, a part of the dynamic scheduler for out-of-order execution. In other words, this scheme exploits a dynamic scheduler for out-of-order processors to duplicate and schedule instructions. However, this approach also has two drawbacks: (i) *significant hardware cost due to the dynamic scheduler* and (ii) *inflexibility as a hardware-based approach*.

In contrast, our approach incurs the least area overhead since ours exploits VLIW architectures which do not need the dynamic scheduler. Further, as a software-based approach, our scheme has advantages in that ours can exploit global information available at compile time such as a loop structure and adjust the threshold value at compile time without redesigning the hardware architecture. Also, our method proposes methodology to combine physical or spatial vulnerability and temporal vulnerability by offline profiling. Therefore, our approach can improve reliability from various angles. In our experiments, we will present the effectiveness of our software-based scheme as compared to the hardware-based selective replication scheme.

3. COMPILER-ASSISTED DYNAMIC CODE DUPLICATION

We propose a *compiler-assisted dynamic code duplication scheme* for VLIW architectures. Our purpose of instruction duplication is to mitigate soft error impacts on datapath, in particular, ALU (IALU and FALU) and LSU (Load/Store Unit). All the other components are assumed protected in an appropriate way. For instance, instruction cache, data cache, and general-purpose register files can be protected with parity. Also we assume that buses, queues, comparators, and other registers are protected as well. For our proposed scheme, our compiler can generate scheduling information embedded in the code and help our modified VLIW architecture duplicate instructions at runtime rather than at compile time. This is why our approach can resolve the issue of increased code size in the static code duplication scheme since duplicated instructions are not explicitly present in our code before runtime while they are present in the code of the static code duplication scheme.

We have implemented our VLIW architecture for dynamic code duplication scheme by modifying that of the static code duplication scheme [Hu et al. 2009]. Figure 1 shows the datapath of both VLIW architectures where the modified part is highlighted in shade. Indeed, the fetch stage only needs to be modified mainly because our scheme

needs to decode and use the embedded information for instruction duplication generated by our compiler. At the fetch stage, a sequence of consecutive instructions, called a *fetch packet*, is read from the program memory, and each instruction of the fetch packet is sent to the decode stage according to the functionality of each issue slot. The bundle of instructions sent to the decode stage is called an *execute packet*, which is identical to the fetch packet in the case of the static scheme in general and in the static code duplication scheme as well. On the other hand, in our scheme, the execute packet could be different from the fetch packet. The difference results from that the fetch packet does not have a duplicated instruction explicitly within it, but the execute packet could include duplicated instructions, which are newly generated at the fetch stage. In other words, instructions are duplicated dynamically at the fetch stage in our scheme. Thus, we separate the fetch stage into FEF (Front-End Fetch) and BEF (Back-End Fetch), introduce two pipeline register sets between FEF and BEF (FEF/BEF Register) and between BEF and Decode (BEF/DC Register), and add MUXes between them to selectively duplicate instructions at different cycles as shown in Figure 1. Note that the pipeline architectures from our decode stage are identical to that of the static scheme and our dynamic code duplication approach can exploit the features of the static scheme [Hu et al. 2005, 2009]. To compare the outputs of both original and duplicate instructions for validation, the static scheme proposes and maintains architectural components such as integer/floating-point register value queue (IRVQ/FRVQ) and load/store address queue (LSAQ) whereby all value comparisons are accomplished without explicit checking instructions (see Figure 1). For instance, when an original instruction completes, it writes the value into the output register as well as in the RVQ. When the duplicate instruction of this instruction completes, its output is compared with the content of the entry in the RVQ associated with the original instruction. Therefore, our dynamic code duplication scheme also eliminates the need of checking instructions for validation by taking advantage of IRVQ/FRVQ and LSAQ since our dynamic scheme also exploits these features. In both the static scheme and our dynamic scheme, an original instruction and its duplicate one identically behave except that RVQ and LSAQ are only written by the original. Other hardware components, such as arithmetic logic unit, address generation unit, bypassing unit, etc., are equivalently exploited for both original and duplicate instructions. One thing we need to keep in mind is that the identical parts between two schemes are the *pipeline architectures* after the fetch stages, not the *execution behaviors*, since we only modified the fetch stage while maintaining the other stages (DC, EX, MEM, and WB stages) unchanged from the static scheme. Even though the pipeline stages except for the fetch stages are identical, the execution behaviors of two schemes should not be the same since the code generations of two schemes are different from each other. In the following subsections, we will describe our dynamic code duplication mechanism and its modified fetch stage in more detail.

3.1. ISA Design

When a fetch packet is converted to an execute packet at the fetch stage, configuration information for the execute packet should be given in a certain mechanism. To minimize the hardware overhead for duplicate instructions, our dynamic scheme considers three possible duplication cases in addition to no duplication. For this, we designate two bits, *D0* and *D1*, in each instruction as summarized in Table I. They indicate whether an original instruction is duplicated or not at runtime. They also indicate whether it is scheduled at the current cycle or at the next cycle if it is duplicated. Further, they indicate whether in a new packet or not if it is at the next cycle. First, a duplicate is generated at the same cycle as its original instruction. As shown in Figure 2(a), the duplicate of an instruction *A* at slot 0 is generated at slot 1 where there is a

(a) duplicated to replace a NOP at the same cycle: $(D0,D1)=(0,1)$ (b) duplicated to replace a NOP at the next cycle: $(D0,D1)=(1,0)$

slot 0, 1 : ALU instructions

slot 2 : multiply, floating point operations

slot 3 : memory access instructions (load, store)

(c) duplicated within a new VLIW packet at the next cycle: $(D0,D1)=(1,1)$ Fig. 2. Three possible cases for generating instruction duplication (a shaded one represents a duplicate instruction, $(D0,D1)=(0,0)$ means no duplication).

Table I. The Meaning of D0 and D1 to Duplicate Instructions Dynamically

D0	D1	Implication
0	0	Not Duplicated
0	1	Duplicated to replace a NOP at the same cycle
1	0	Duplicated to replace a NOP at the next cycle
1	1	Duplicated within a new VLIW packet at the next cycle

NOP available. Second, a duplicate is generated to replace a NOP at the next cycle. Figure 2(b) shows that the duplicate of A cannot be generated at cycle t since there is not a NOP but B at slot 1. Even though there are NOPs at slot 2 and slot 3, the duplicate of A cannot be executed at those slots due to the constraints of issue slots. In this case, if there is a NOP at the same issue slot of the next cycle, a duplicate can be generated to replace the NOP. Otherwise, a new VLIW packet should be generated for duplicates, as shown in Figure 2(c). This is the third case.

Note that it is definitely not hard to find out two bits unused space in general in 32-bit instruction set architecture [Lee et al. 2012] and D0 and D1 can be assigned into existing encoding space of instructions without overheads of space and loss of instructions [Hu et al. 2009].

To distinguish each case at runtime, the configuration information should be embedded in the instruction encoding space at compile time. Otherwise, the hardware cannot

avoid being more complicated because it should dynamically check data dependencies among instructions. Also, we consider scheduling the duplicate instruction by the next cycle, not further, to minimize the complexity overhead of the hardware implementation. Thus, the duplication range, the range of cycles where a duplicate instruction can be scheduled, is two cycles in our scheme. However, this limitation of the duplication range does not incur performance overhead in our experimental results as will be presented in Section 5.2. Note that our compiler-assisted dynamic code duplication scheme is orthogonal to further complicated VLIW architectures and there should be interesting trade off space between complexity and performance, which is definitely a topic for our future work.

3.2. Modified Fetch Stage

Our approach needs to separate the fetch stage into two stages, FEF and BEF. It could have increased clock cycle time if decoding $D0$ and $D1$ would be merged into the fetch stage or into the decode stage, which would have made a negative impact on the performance at that stage. Figure 3 depicts behavior flowcharts for two separate FEF and BEF stages. Global registers such as R , $prev_pc$, $N1$, and $N2$, and local registers used for each issue slot such as $D[i]$ and $prev_inst[i]$ are newly introduced. The global registers are for the third case. R notifies that the fetch packet of the next cycle should be the same as that of the current cycle. For this, $prev_pc$ stores the address of the fetch packet of the current cycle. $N1$ indicates that the current fetch packet at FEF stage requires a new VLIW packet for its duplicate, and $N2$ represents whether the current execute packet at BEF stage is an original or a duplicate. The local registers are for the second and the third cases. $prev_inst[i]$ stores the original instruction and delivers it to the decode stage by writing it into BEF/DC pipeline register if $D[i]$ is set to 1, which means the duplication of this instruction. Using these registers, FEF and BEF stages work in a combined manner to support dynamic instruction duplication. As shown in Figure 3, FEF stage behaves differently according to the values of R and $N1$, and processes the first case, i.e., $(D0, D1) = (0, 1)$. BEF stage also processes the second and third cases according to the values of $D[i]$, $N2$ and $(D0, D1)$ of each instruction in the execute packet as shown in Figure 3(b).

To help understand our scheme, Figure 4 presents an example how FEF and BEF stages work for each case of generating duplicated instructions. Figure 4(a) shows an example code and initial states of pipeline registers and other registers. Figure 4(b) shows the architectural state after execution at cycle 0. $I0'$, a duplicate of $I0$, is generated at FEF stage and is placed in slot 1 since $(D0, D1)$ of $I0$ equals $(0, 1)$. Thus, Figure 4(b) shows the first case. Next, Figure 4(c) shows the second case since $(D0, D1)$ equals to $(1, 0)$, respectively, in each of $I2$ and $I3$ in $P1$. Duplicated instructions, $I2'$ and $I3'$, are stored in $prev_inst[0]$ and $prev_inst[1]$, respectively, at cycle 2. Also, both $D[0]$ and $D[1]$ are set to 1. Then, BEF stage delivers $I2'$ and $I3'$ to the next pipeline stage at cycle 3 since both $D[0]$ and $D[1]$ are equal to 1 (see Figure 3(b) and Figure 4(e)). Figure 4(e) shows that $I2'$ and $I3'$ are placed in the NOP slots, slot0 and slot1, of $P2$. Thus, $I2'$, $I3'$, $I5$, and $I6$ are executed at the same cycle. Note that we need to avoid a data dependency violation among $I2'$, $I3'$, $I5$, and $I6$, which is guaranteed with the help of an instruction duplication algorithm at compile time.

The last case for duplicating instructions is the third, where both $D0$ and $D1$ are equal to 1. Duplicated instructions, $I5'$ and $I6'$, are stored in $prev_inst[2]$ and $prev_inst[3]$, respectively, at cycle 3 as shown in Figure 4(e). Also, both $D[2]$ and $D[3]$ are set to 1. In this case, $P3$ does not have NOPs that could be replaced with duplicated instructions, so a new VLIW packet is generated for to the decode stage at cycle 4 as shown in Figure 4(f). the duplicate of $P2$ and then the new packet is delivered. This new packet generation is processed at BEF stage as the behavior represented in

(a) Front-End Fetch (FEF) stage

(b) Back-End Fetch (BEF) stage

Fig. 3. Behavior flowchart of modified fetch stage.

Figure 3(b). Since $D[2]$ and $D[3]$ are equal to 1, $I5'$ and $I6'$ are delivered to the next pipeline stage at cycle 4 as shown in Figure 4(f). In the case of slot 0 and slot 1, NOPs should be delivered to the next pipeline stage at cycle 4. For this, $N2$ is set to 1 at cycle 3 in Figure 4(e). Otherwise, $I7$ and $I8$ could be delivered to the next pipeline stage with $I5'$ and $I6'$, which incurs an incorrect result of the program. Due to the generation of the new packet, $P3$ should be delivered to the decode stage at cycle 5 instead of cycle 4. For this, FEF stage at cycle 4 needs to read the same fetch packet at cycle 3, i.e., $P4$ should be identical to $P3$. Otherwise, $P3$ cannot be delivered to the decode stage, and the program would not work properly. To read the same fetch packet in consecutive two cycles as shown in Figure 4(e) and Figure 4(f), R and $N1$ are used as flags. At cycle 2, our architecture understands the necessity of a new packet to generate the duplicate of $P2$ since $(D0, D1)$ are equal to $(1, 1)$ for $I5$ and $I6$. Thus, $N1$ is set to 1 at cycle 2

Fig. 4. An example: Behaviors of modified fetch stage according to each case.

(see Figure 4(d)) and therefore R is also set to 1 at cycle 3 (see Figure 4(e)). Since R is 1 at the start of cycle 4, a fetch packet is read from $prev_pc$ (see Figure 3(a)) and it becomes possible to read the same fetch packet at both cycles 3 and 4 (see Figure 4(f)).

In this section, we describe our architecture based on previously proposed VLIW architecture for compiler-assisted dynamic code duplication. The following will present our compilation and instruction duplication techniques for our VLIW architecture.

4. COMPILATION TECHNIQUES

Our proposal resolves two issues: (i) code size reduction and (ii) vulnerability-aware instruction duplication.

Our compiler-assisted dynamic code duplication scheme is able to reduce the code size by dynamically duplicating instructions in the modified fetch stage in VLIW architectures as described in the previous section and to effectively increase reliability by taking the different degrees of instruction vulnerabilities into account in duplicating instructions. In the following subsections, we will talk about the previously proposed static code duplication scheme and our dynamic code duplication scheme which is aware of vulnerability to determine which instructions are to be duplicated at compile time.

4.1. Static Code Duplication Algorithm

As stated in Section 2, the static code duplication scheme [Hu et al. 2009] can trade off performance loss with required degree of reliability by adjusting the amount of duplicated instructions while other previous works fully duplicate instructions at the expense of maximum performance loss. Their approach allocates duplicated instructions if NOP is available or increases the schedule length for duplicating instructions within a duplication range which is determined by the level of allowable performance degradation.

However, their code duplication algorithm duplicates instructions in a sequential manner and its fault-tolerant coverage is limited to the earlier examined instructions, especially if the power or performance constraints are limited. For instance, their duplication algorithm under no performance overhead duplicates 17 instructions out of duplicable 70 ones and they are all located within the first half in the scheduled code for benchmark *complex_multiply* when we implement their duplication algorithm and run a simple experiment. This unbalanced duplication of instructions is effective enough to increase the reliability under the performance bound if each instruction has equal impact on the reliability to others. However, several researchers [Reis et al. 2005a, 2007; Mukherjee et al. 2005; Vera et al. 2009; Lee et al. 2006, 2009] have shown that not all data or instructions are equally important in terms of reliability. Thus, our code duplication algorithm introduces the different degrees of instruction vulnerabilities when selecting instructions for duplication to increase the reliability with minimal performance overhead, which will be described in the following subsection.

4.2. Vulnerability-Aware Duplication Algorithm

We present three duplication algorithms considering different degrees of instruction vulnerabilities for duplicating instructions to effectively maximize reliability.

Our first approach is *temporal-vulnerability-aware duplication (TVAD)* algorithm. Temporal vulnerability has been presented and exploited in several previous works, especially works for cache and memory protection against soft errors [Asadi et al. 2005; Wang et al. 2006]. They estimate the time period of data such as program variables in caches and protect selectively those which have higher vulnerability in terms of time above the threshold value. However, it is extremely hard to estimate which instruction executes more often than others. We suppose that estimating the vulnerability of an instruction in a datapath is beyond our interest in this work and it will be definitely our future work. In our study, the first simple proposal exploits this concept of temporal vulnerability and considers instructions in the loop more important than others in terms of reliability since they have higher chance to be executed more often, which implies more chances to be exposed to soft errors. Indeed, compilation techniques suppose 10 times of execution for instructions in the loop [Muchnick 1997] so our

Table II. Cell Areas of Instructions Normalized to That of an Instruction *ldc.ri*

Instruction	Area	Instruction	Area	Instruction	Area
mul	271.72	add + shift	219.40	sub + shift	213.58
xor + shift	178.06	add	170.36	or + shift	168.06
and + shift	164.41	sub	155.05	xor	129.16
or	114.83	and	109.93	store	87.03
load	69.72	cmp_rr	59.96	branch	48.77
cmp_ri	48.26	lui_ri	1.00	ldc_ri	1

approach puts 10 times more vulnerability for instructions in the loop, i.e., importance in terms of reliability, than ones out of the loop when our approach selects instructions for duplication. Thus, our TVAD algorithm defines the vulnerability of instructions such that $V_I = 10 \times v$ if I is in the loop or $V_I = 1 \times v$ otherwise where V_I is the vulnerability of an instruction I and v is a vulnerability unit.

Our second approach is *physical-vulnerability-aware duplication (PVAD)* algorithm. Note that the more exposed, the more vulnerable [Kim 2006; Vera et al. 2009]. If a combinational logic consists of more number of transistors and takes up larger portion in the chipset than another logic, it is more vulnerable since it is more largely exposed to energetic particles inducing soft errors. To estimate physical vulnerabilities of instructions, we have run a simple experiment in a compiler-simulator-synthesizer framework (see Section 5.1) and estimated the cell areas of instructions. Table II samples the normalized cell areas of several instructions to that of an instruction *ldc.ri*. The *mul* instruction takes up more than 270 times in cell area than *ldc.ri* instruction, which can be translated into 270 times higher vulnerability of *mul* instruction than that of *ldc.ri*. Thus, it makes better sense in terms of reliability to duplicate *mul* instruction rather than *ldc.ri* if we can only select one instruction out of these two instructions due to the performance bound. Note that the critical path has been already determined from the longest delay of an instruction in the pipeline design and therefore the selection with larger cell area does not affect the performance negatively. Our PVAD algorithm defines the vulnerability of instructions such that $V_I = n_I \times V_{ldc.ri}$ where V_I is the vulnerability of an instruction I , n_I is the normalized cell area of I to that of the instruction *ldc.ri*, and $V_{ldc.ri}$ is the vulnerability of *ldc.ri*.

Our last approach is *temporal and physical vulnerability-aware duplication (TPVAD)* algorithm combining TVAD, PVAD, and a basic instruction scheduling algorithm in VLIW architecture. Thus, TPVAD is a vulnerability-aware duplication algorithm considering both temporal and physical vulnerability under performance constraint. Our TPVAD can improve reliability more effectively under constrained performance as compared to a previously proposed static code duplication scheme by duplicating instructions with higher vulnerability in terms of both temporal and physical vulnerabilities. TPVAD defines the vulnerability of instructions such as $V_I = 10 \times n_I \times V_{ldc.ri}$ if I is in the loop or $V_I = n_I \times V_{ldc.ri}$ otherwise where V_I is the vulnerability of an instruction I , n_I is the normalized cell area of I to that of the instruction *ldc.ri*, and $V_{ldc.ri}$ is the vulnerability of *ldc.ri*.

Figure 5 describes our TPVAD algorithm. The first loop (lines 01–11) makes a priority list of instructions considering temporal and physical vulnerability simultaneously. The second loop (lines 13–32) attempts to schedule duplicate instructions in priority order with perspective of its vulnerability. For each instruction in the list, L , our TPVAD duplicates the instruction, and gets the information needed for instruction duplication (lines 14–18). And it attempts to schedule the duplicate instruction within the next cycle (lines 19–25). When the schedule fails, TPVAD inserts a new cycle and schedules

TPVADuplication

Input : A VLIW code sequence without duplicate instructions in a function

Output : A VLIW code sequence with information for duplicate instructions in a function

A function has a region graph G Let $G = (N, E)$ where $N = \{n_i \mid n_i \text{ is a region, which is normally a basic block}\}$ and $E = \{e_{ij} \mid \exists e_{ij} \text{ if there is a control flow } n_i \rightarrow n_j\}$ *phy_vul.table* T : a table with each instruction and its physical vulnerability*priority.list* L : a list of instructions sorted in a descending order of the vulnerability

```

01: for (each  $n_i \in G$ )
02:   if ( $n_i$  is a part of loop) then
03:     temp_vul_factor = 10
04:   else
05:     temp_vul_factor = 1
06:   endif
07:   for (each instruction  $inst \in n_i$ )
08:     phy_vul =  $T.get\_phy\_vul(inst)$ 
09:     assign_vul( $inst, phy\_vul \times temp\_vul\_factor$ )
10:      $L.insert(inst)$ 
11:   endfor
12: endfor
13: for (each instruction  $inst \in L$ )
14:   dup_inst = create_duplicate( $inst$ )
15:   region = get_region( $inst$ )
16:   code_increase_margin = get_code_increase_margin(region)
17:   sched_cycle = get_sched_cycle( $inst$ )
18:   possible_cycle = sched_cycle
19:   while (possible_cycle  $\leq$  sched_cycle + 1) do
20:     sched_success = try_to_schedule(dup_inst, possible_cycle)
21:     if (sched_success == true) then
22:       break
23:     endif
24:     possible_cycle ++
25:   endwhile
26:   if (sched_success == false) && (code_increase_margin > 0) then
27:     insert_new_cycle(sched_cycle + 1)
28:     try_to_schedule(dup_inst, sched_cycle + 1)
29:     code_increase_margin --
30:     update_code_increase_margin(region, code_increase_margin)
31:   endif
32: endfor

```

Fig. 5. TPVADuplication– Temporal and Physical Vulnerability-Aware duplication algorithm.

the instruction at the cycle if the code increase margin is larger than 0. Otherwise, it gives up duplicating the instruction (lines 26–31).

5. EXPERIMENTS

5.1. Experimental Setup

To evaluate the effectiveness of our proposals, we have implemented a compiler-simulator-synthesizer framework as shown in Figure 6. Our proposed VLIW architecture has been implemented in Processor Designer of Synopsys [Synopsys Inc. 2001]. It generates software tools such as assembler, linker, and simulator. Further, it generates HDL (Hardware Description Language) code based on an architecture description language LISA 2.0 [Synopsys Inc. 2001]. The software tools are used to estimate code size and execution time, and the HDL code is used as an input to Synopsys Design Compiler [Synopsys Inc. 2001] to retrieve the information in terms of hardware costs such as clock cycle time and cell area as shown in Figure 6.

In our experiments, we have selected one of the diverse processor models offered by Synopsys Processor Designer as a baseline architecture, which is composed of 4-issue slots, i.e., 4-way VLIW architecture, with two integer ALUs, one floating-point ALU, one load/store unit, and one branch unit. The baseline architecture has a typical

Fig. 6. Our compiler-simulator-synthesizer framework.

RISC-style ISA such as MIPS ISA [MIPS Technology, Inc. 2001]. For comparison, both the static code duplication architecture and our proposed architecture have been modeled and implemented upon this baseline architecture. After implementing the static code duplication architecture, our proposed architecture has been modeled by modifying the instruction fetch logic to support our compiler-assisted dynamic code duplication scheme as described in Section 3.2. Architectures for our compiler-assisted dynamic code duplication scheme and the static code duplication scheme are shown in Figure 1.

The compiler for the proposed VLIW architecture is generated by a retargetable compiler platform, SoarGen [Ahn and Paek 2009]. Given the ISA of a target processor architecture written in an architecture description language, SoarDL [Ahn and Paek 2009], SoarGen can generate the compiler for the target processor. The proposed vulnerability-aware duplication algorithm explained in Section 4.2 has been implemented in the compiler for our proposed VLIW architecture.

For extensive simulations, we have used two suites of benchmarks, DSPstone [Zivojnovic et al. 1994] and Livermore Loops [McMahon 1986], to evaluate the effectiveness of our approach. DSPstone is a suite of kernel benchmarks consisting of code fragments or functions which are commonly used in DSP algorithms. Livermore Loops is a benchmark suite for parallel architectures and consists of a set of loop kernels in numerically intensive applications. To estimate the overhead of our proposed architecture, cell area, power, and clock cycle time are estimated. To show the efficiency of our proposed architecture, the analysis result in terms of code size and execution time will be presented in Section 5.2. Also, the effectiveness of the proposed vulnerability-aware duplication algorithm in terms of vulnerability will be presented in Section 5.3.

5.2. Effectiveness of Compiler-Assisted Dynamic Code Duplication

To see the area cost, our first analysis has synthesized HDL codes for both architectures of the static code duplication scheme and our compiler-assisted dynamic code duplication scheme. Table III shows the logic synthesis results in terms of cell areas from Synopsys Design Compiler with the input of the HDL code generated by Processor Designer in Synopsys [Synopsys Inc. 2001]. Total cell area includes combinational area and noncombinational area. Compared to that of the static code duplication scheme,

Table III. Cell Area Comparisons (unit: μm^2)

Duplication Scheme	Static Code Duplication Scheme	Compiler-assisted Dynamic Code Duplication Scheme
Combinational Area	426635.84	437481.09
Non-combinational Area	136997.53	144327.28
Total Area	563633.37	581808.37

Fig. 7. The effectiveness in code size reduction.

the total cell area of our scheme increased by about 3.2%. This overhead results from adding a new pipeline stage due to the split of the fetch stage.

The power consumption overhead caused by adding an extra pipeline stage is negligible at least from our experimental analysis, since it does not incur much power overhead in case of the number of pipeline stages 5 to 6 where our architecture has been designed. In our preliminary experiments with McPAT [Li et al. 2009], we have estimated power consumptions as the number of pipeline stage increases from pipeline depth 4 to 10 by 1 for all available architectures such as Niagara, Alpha, and X86. We also estimate the power consumption with the regression line from the number of pipeline stages 4 to 10, and it also shows nearly linear regression in the range of 3.3% as one pipeline stage is added. In summary, adding one stage to the pipeline is not a big concern in terms of power consumption in this analysis.

We have adopted the energy consumption estimation model from that of the previously proposed static scheme [Hu et al. 2009]. Our experimental results clearly show that our approach does not incur much overhead in terms of energy consumption as compared to the static scheme. The energy consumption model in the static scheme only considers those of functional units to estimate the impact of duplicating instructions on energy consumption. Our energy consumption estimation is identical to the static scheme except that an additional pipeline stage increases the power consumption in our scheme. Since power consumption overhead for our new architecture is significantly small, the energy consumption overhead is also small. Our experiments show minimal energy consumption overhead (less than 3% on average). In summary, our dynamic duplication scheme can be implemented with minimal energy consumption overhead as compared to the static scheme.

Our first set of experiments is to evaluate code size and execution time when duplicating instructions. Figure 7 clearly shows the effectiveness of our scheme in terms of code size. Our dynamic code duplication scheme can reduce code size overhead by allocating duplicated instructions at runtime at the cost of an acceptable hardware

Fig. 8. The effective code size reduction over benchmarks.

Fig. 9. The competitiveness in performance.

overhead (about 3.2% area overhead). It is significantly efficient since it does not incur any overhead in terms of memory space for the code. Figure 7 shows the code size of our dynamic code duplication scheme as compared to that of a previously proposed static code duplication scheme [Hu et al. 2009] with the *complex_multiply* benchmark of the DSPstone suite [Zivojnovic et al. 1994]. Clearly, our approach can achieve the reduction of code size, i.e., it does not incur any duplicated code at compile time while the static code duplication scheme increases the code size by duplicating more instructions. With the increase of allowable performance degradation, ours can keep reducing the code size and achieve code size reduction by up to about 40% under 100% performance constraint as compared to the static code duplication scheme. This is because our method does not increase code size, while the static code duplication scheme increases code size as more performance degradation is allowed. We can observe similar results with other benchmarks as shown in Figure 8. Our scheme can reduce the code size by about 30% on average over the benchmark suite as compared to the static code duplication scheme in case of full instruction duplication under the 100% performance bound. The main reason of this code size reduction is that our scheme can duplicate instructions at runtime rather than at compile time.

Figure 9 represents the result of the performance comparison between our approach and the static scheme in case of the *fir2dim* benchmark from the DSPstone suite. Our scheme achieves comparable performance with the static code duplication scheme when duplicating the same number of instructions. To estimate performance, the execution time is measured by a cycle-accurate simulator from Synopsys Processor

Designer [Synopsys Inc. 2001]. As explained in Section 3.1, the duplication range of our scheme is limited to only two cycles to reduce the complexity overhead of hardware implementation while that of the static scheme is the end of a basic block. Since the static scheme can definitely browse a larger interesting space than ours, we also speculated that duplicate instructions of the static scheme would be located to the code with smaller amount of overhead in cycles than our scheme. However, our limited duplication range does not incur significant performance overhead as compared to the static scheme mainly because most duplicate instructions are allocated within the next cycles of their originals. This is mainly due to data dependency even with the static scheme. We believe that the data dependency prevents the large duplication range of the static scheme from being fully utilized. Indeed, in the static scheme, most duplicate instructions are allocated within the next or a few further cycles of their originals, not the end of the basic block, due to the data dependency. Our experimental results over benchmark suites show that our scheme incurs performance overhead by up to 3% (in case of *fir2dim* benchmark as shown in Figure 9) and 1% performance overhead on average, as compared to the static scheme in terms of the execution time. As a result, a small duplication range of our approach incurs negligible performance overhead, which provides a solid foundation for our scheme in terms of performance.

In summary, our scheme can effectively reduce the code size with little performance overhead since our compiler can generate duplication information and our modified fetch stage can utilize this information for instruction duplication at runtime.

5.3. Effectiveness of Vulnerability-Aware Duplication Algorithm

Our second set of experiments is to evaluate the effectiveness of our approach in terms of vulnerability reduction and error detection. To estimate the amount of vulnerability reduction, our heuristic method quantifies the vulnerability of an instruction using the cell area computed by Synopsys Design Compiler as stated in Section 4.2, based on the observation that the larger area occupied by the instruction, the more vulnerable it is [Vera et al. 2009; Rehman et al. 2011]. This quantification enables us to reflect *physical vulnerability*. Total amount of vulnerability reduction in a program is the sum of the vulnerability values of all duplicated instructions. The total vulnerability reduction is measured at runtime, not compile time, to take into account actual loop execution count. This runtime vulnerability measurement makes it possible to reflect *temporal vulnerability* in our study. To validate the effectiveness of our approach, fault injection experiments have been performed. We have injected soft errors, i.e., random bit errors, into instructions in our simulation framework. A fault rate is one fault per 100 cycles on average with a random time interval between consecutive faults. We have run 1,000 times of each benchmark and calculated the error detection rate which is the number of detected errors over 1,000 experiments. Note that vulnerability reduction experiments show the relative reliability based on a heuristic method and our fault injection experiments show the practical effects of our approach in reliability.

Figure 10 clearly shows the effectiveness of considering temporal vulnerability in our approach. For convenience, we call an instruction in a loop, a *loop instruction*, and otherwise, a *normal instruction*. Figure 10(a) shows the rate of vulnerability reduction according to the change of the number of duplicated instructions for each approach. The rate of vulnerability reduction (R) is calculated as $R = 100 \times (V_{Dup} / V_{All})$ where V_{Dup} is the sum of vulnerabilities of duplicated instructions and V_{All} is the sum of vulnerabilities of all duplicable instructions. When a loop instruction is selected for duplication, its vulnerability is accumulated repeatedly as much as dynamic loop execution count in computing V_{Dup} . For a fair comparison, the rate of vulnerability reduction is estimated under the constraint that both approaches have identical numbers of duplicated instructions. Our approach achieves more vulnerability reduction than the static code

(a) vulnerability reduction

(b) error detection

Fig. 10. Effectiveness of our dynamic code duplication with TVAD in terms of vulnerability and error detection.

duplication scheme since a loop instruction has higher priority for duplication than a normal one and therefore more loop instructions can be duplicated in our approach than the static code duplication scheme if the same number of instructions are duplicated. On the other hand, it is likely that more normal instructions are duplicated in the static code duplication scheme since it duplicates instructions in the order of their locations in the code. Thus, the static code duplication scheme achieves less vulnerability reduction than our approach. In *convolution* of DSPstone benchmark, our approach can achieve more vulnerability reduction by up to 70% and by on average 43% than the static code duplication scheme. Figure 10(b) shows the error detection rate according to the change of the number of duplicated instructions for each approach. The error detection rate is calculated as the ratio in percentage of the number of detected errors to the total number of the injected errors in our simulation framework. Figure 10(b) shows that the error detection rate follows the trend close to the vulnerability reduction rate with a benchmark, *convolution*. Our approach can achieve more error detection by up to 22% and by on average 13% than the static code duplication scheme. Note that our approach seems less effective in terms of error detection than in terms of vulnerability reduction and it is mainly because an error can be easily injected on instructions without their duplications and an injected error does not always result in different outputs at the comparison.

Figure 11 clearly shows the effectiveness of considering physical vulnerability in our approach. By our heuristic method, each instruction has its own physical vulnerability based on the cell area occupied by itself as explained before. Figure 11(a) shows the

Fig. 11. Effectiveness of our dynamic code duplication scheme with PVAD in terms of vulnerability and error detection.

rate of vulnerability reduction according to the change of performance degradation. Note that the performance degradation is a knob to adjust the reliability by duplicating instructions under that performance bound. In case of the maximal performance degradation, both approaches achieve very close vulnerability reduction since most instructions are duplicated. However, when performance degradation is limited, i.e., a small number of instructions could be duplicated, our approach can achieve more vulnerability reduction than the static code duplication scheme. This is because the instructions with higher physical vulnerability can be preferentially duplicated in our approach while the static code duplication scheme cannot. In *convolution* of DSPstone benchmark, our approach can achieve more vulnerability reduction by up to 23% and by on average 5% than the static code duplication scheme. Figure 11(b) shows the error detection rate according to the change of performance degradation. In *convolution* of the DSPstone benchmark, our approach can detect more errors by up to 5% and by on average 3% than the static code duplication scheme.

Figure 12 clearly shows the effectiveness of our approach in terms of vulnerability reduction and error detection when jointly considering both temporal and physical vulnerability. Figure 12(a) shows the rate of vulnerability reduction for both approaches when the difference between the rates of vulnerability reduction of both approaches is maximum for each benchmark. In common with the case of considering only the temporal vulnerability, the rate of vulnerability reduction is estimated under the constraint

Fig. 12. Effectiveness of our dynamic code duplication with TPVAD in terms of vulnerability and error detection.

that both approaches duplicate the same number of instructions for each benchmark. Obviously, the effectiveness of considering two types of vulnerabilities simultaneously is better than that of considering solely one of those vulnerabilities. Over benchmarks, our approach can achieve more vulnerability reduction by up to 91% and by on average 60% than the static code duplication scheme. Figure 12(b) shows the error detection rate for both approaches when the difference between the error detection rates of both approaches is maximum over benchmarks. Our approach can detect more errors by up to 82% and by on average 23% than the static code duplication scheme.

Note that our vulnerability reduction measurement is an approximate estimation. To achieve more accurate estimation of vulnerability reduction, AVF (Architectural Vulnerability Factor) can be used as a metric. However, since our proposed approach is orthogonal to methodologies for measuring the vulnerability reduction, our experimental results are enough to show the relative efficiency of our approach even if there might exist some degree of incorrectness. This is why fault injection experiments have been performed and the error detection rate has been calculated.

In summary, our approach can reduce the vulnerability and increase error detection rate, i.e., increase reliability effectively since ours can consider the importance of instructions when selecting instructions to be duplicated under the performance constraint.

Our third set of experiments is to show the efficacy of our compiler-based dynamic code duplication scheme in terms of vulnerability as compared to the hardware-based

(a) vulnerability reduction

(b) error detection

Fig. 13. The effectiveness of our compiler-assisted dynamic code duplication as compared to hardware-based approach.

dynamic code duplication scheme. Figure 13 clearly shows the effectiveness of considering the instruction vulnerability at software level. Unlike the software-based approach where the instruction vulnerability is considered at the software level such as our proposal, Vera et al. [2009] present a hardware-based approach that considers the instruction vulnerability at hardware level as explained in Section 2. Figure 13(a) shows that there is an explicit trade-off between the rate of vulnerability reduction and the performance in the hardware-based approach. However, our approach can achieve more vulnerability reduction when a small performance overhead is allowed as compared to the hardware-based approach. Similarly, Figure 13(b) shows that our approach can detect more errors than the hardware-based approach when a small performance is allowed. This is because our approach can control the threshold value according to the characteristics of instructions such as whether they exist in a loop or not, and thus achieve as much vulnerability reduction as possible at compile time.

In summary, our approach can present a more flexible solution for improving the system reliability compared to the hardware-based approach.

6. CONCLUSION

Embedded systems designers are paying more and more attention to soft errors to increase reliability as well as other constraints such as power, performance, area, and code size. In VLIW, researchers proposed techniques to increase reliability by duplicating instructions. However, these techniques in general incur an increase of code size and ignore different vulnerability levels of instructions. In this work, we propose compiler-assisted dynamic code duplication for VLIW architectures and present vulnerability-aware duplication algorithms to effectively increase reliability with minimal code size overhead.

Our future work includes the vulnerability analysis of instructions at other abstraction levels such as the gate level and further selective protections to achieve high reliability against soft errors with minimal overheads in terms of power, performance, and area for other architectures.

REFERENCES

- AHN, M. AND PAK, Y. 2009. Fast code generation for embedded processors with aliased heterogeneous registers. *Trans. High Perform. Embed. Archit. Compilers*. 149–172.
- ASADI, G., SRIDHARAN, V., TAHOORI, M., AND KAEI, D. 2005. Balancing performance and reliability in the memory hierarchy. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'05)*. 269–279.
- AUSTIN, T. 1999. DIVA: A reliable substrate for deep submicron microarchitecture design. In *Proceedings of the 32nd Annual International Symposium on Microarchitecture (MICRO'99)*. IEEE, 196–207.
- BAUMANN, R. 2005. Soft errors in advanced computer systems. *IEEE Des. Test Comput.* 22, 3, 258–266.
- BOLCHINI, C. 2003. A software methodology for detecting hardware faults in vliw data paths. *IEEE Trans. Reliabil.* 52, 4, 458–468.
- BORODIN, D. AND JUURLINK, B. H. H. B. 2010. Instruction precomputation with memorization for fault detection. In *Proceedings of the Conference on Design, Automation and Test in Europe*. 1665–1668.
- GOMAA, M. AND VIJAYKUMAR, T. 2005. Opportunistic transient-fault detection. In *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA'05)*. IEEE, 172–183.
- HAZUCHA, P. AND SVENSSON, C. 2000. Impact of cmos technology scaling on the atmospheric neutron soft error rate. *IEEE Trans. Nucl. Sci.* 47, 6, 2586–2594.
- HOLM, J. AND BANERJEE, P. 1992. Low cost concurrent error detection in a vliw architecture using replicated instructions. In *Proceedings of the International Conference on Parallel Processing*. 192–195.
- HU, J., LI, F., DEGALAHAL, V., KANDEMIR, M., VIJAYKRISHNAN, N., AND IRWIN, M. 2005. Compiler-Directed instruction duplication for soft error detection. In *Proceedings of the Conference on Design, Automation and Test in Europe*. 1056–1057.
- HU, J., LI, F., DEGALAHAL, V., KANDEMIR, M., VIJAYKRISHNAN, N., AND IRWIN, M. 2009. Compiler-Assisted soft error detection under performance and energy constraints in embedded systems. *ACM Trans. Embed. Comput. Syst.* 8, 4, 27.
- KIM, S. 2006. Area-Efficient error protection for caches. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'06)*. 1282–1287.
- LEE, J., YOUN, J., LEE, J., AHN, M., AND PAK, Y. 2012. Dynamic operands insertion for vliw architecture with a reduced bit-width instruction set. In *Proceedings of the 26th International IEEE Parallel Distributed Processing Symposium (IPDPS'12)*. 119–130.
- LEE, K., SHRIVASTAVA, A., ISSENIN, I., DUTT, N., AND VENKATASUBRAMANIAN, N. 2006. Mitigating soft error failures for multimedia applications by selective data protection. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems*. ACM Press, New York, 411–420.
- LEE, K., SHRIVASTAVA, A., ISSENIN, I., DUTT, N., AND VENKATASUBRAMANIAN, N. 2009. Partially protected caches to reduce failures due to soft errors in multimedia applications. *IEEE Trans. VLSI Syst.* 17, 9, 1343–1347.
- LI, S., AHN, J., STRONG, R., BROCKMAN, J., TULLSEN, D., AND JOUPPI, N. 2009. Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'09)*. IEEE, 469–480.
- LYONS, D. 2000. Sun screen: Soft error issue in sun enterprise servers. members.forbes.com/global/2000/1113/0323026a.html.

- McMAHON, F. H. 1986. The Livermore fortran kernels: A computer test of the numerical performance range. Tech. rep. UCRL-53745, Lawrence Livermore National Laboratory.
- MIEXNER, A., BAUER, M., AND SORIN, D. 2007. Argus: Low-Cost, comprehensive error detection in simple cores. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'07)*. IEEE, 210–222.
- MIPS TECHNOLOGY, INC. 2001. *MIPS32 Architecture for Programmers*. Vol. 1: Introduction to the MIPS32 Architecture. MIPS Technology, Inc., Mountain View, CA.
- MITRA, S., SEIFERT, N., ZHANG, M., SHI, Q., AND KIM, K. 2005. Robust system design with built-in soft-error resilience. *Comput.* 38, 2, 43–52.
- MUCHNICK, S. S. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, CA.
- MUKHERJEE, S. 2008. *Architecture Design for Soft Errors*. Morgan Kaufmann, San Francisco, CA.
- MUKHERJEE, S., EMER, J., AND REINHARDT, S. 2005. The soft error problem: An architectural perspective. In *Proceedings of the 11th International Symposium on High Performance Computer Architecture (HPCA'05)*. IEEE, 243–247.
- MUKHERJEE, S., WEAVER, C., EMER, J., REINHARDT, S., AND AUSTIN, T. 2003. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'03)*. IEEE, 29–40.
- NAKKA, N., PATTABIRAMAN, K., AND IYER, R. 2007. Processor-Level selective replication. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. 544–553.
- REDDY, V., ROTENBERG, E., AND PARTHASARATHY, S. 2006. Understanding prediction-based partial redundant threading for low-overhead, high-coverage fault tolerance. *SIGOPS Oper. Syst. Rev.* 40, 83–94.
- REHMAN, S., SHAFIQUE, M., KRIEBEL, F., AND HENKEL, J. 2011. Reliable software for unreliable hardware: Embedded code generation aiming at reliability. In *Proceedings of the 7th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. 237–246.
- REINHARDT, S. AND MUKHERJEE, S. 2000. Transient fault detection via simultaneous multithreading. *SIGARCH Comput. Archit. News* 28, 2, 25–36.
- REIS, G., CHANG, J., AND AUGUST, D. 2007. Automatic instruction-level software-only recovery. *IEEE Micro* 27, 1, 36–47.
- REIS, G., CHANG, J., VACHHARAJANI, N., RANGAN, R., AND AUGUST, D. 2005a. Swift: Software implemented fault tolerance. In *Proceedings of the International Symposium on Code Generation and Optimization*. IEEE, 243–254.
- REIS, G., CHANG, J., VACHHARAJANI, N., RANGAN, R., AUGUST, D., AND MUKHERJEE, S. 2005b. Software-Controlled fault tolerance. *ACM Trans. Archit. Code Optim.* 2, 4, 366–396.
- REIS, G. A., CHANG, J., VACHHARAJANI, N., RANGAN, R., AUGUST, D. I., AND MUKHERJEE, S. S. 2005c. Design and evaluation of hybrid fault-detection systems. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*. 148–159.
- SHRIVASTAVA, A., LEE, J., AND JEYAPPAUL, R. 2010. Cache vulnerability equations for protecting data in embedded processor caches from soft errors. *SIGPLAN Not.* 45, 143–152.
- SOUNDARARAJAN, N. K., PARASHAR, A., AND SIVASUBRAMANIAM, A. 2007. Mechanisms for bounding vulnerabilities of processor structures. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA'07)*. ACM Press, New York, 506–515.
- SYNOPSYS INC. 2001. *Design Compiler Reference Manual*. Synopsys Inc., Mountain View, CA.
- TREMBLAY, M. AND TAMIR, Y. 1989. Support for fault tolerance in vlsi processors. In *Proceedings of the IEEE International Symposium on Circuits and Systems*. IEEE, 388–392.
- VERA, X., ABELLA, J., CARRETERO, J., AND GONZALEZ, A. 2009. Selective replication: A lightweight technique for soft errors. *ACM Trans. Comput. Syst.* 27, 4, 8.
- WANG, N. AND PATEL, S. 2006. Restore: Symptom-Based soft error detection in microprocessors. *IEEE Trans. Depend. Secur. Comput.* 3, 3, 188–201.
- WANG, N., QUEK, J., RAFACZ, T., AND PATEL, S. 2004. Characterizing the effects of transient faults on a high-performance processor pipeline. In *Proceedings of the International Conference on Dependable Systems and Networks*. IEEE, 61–70.
- WANG, S., HU, J., AND ZIAVRAS, S. 2006. On the characterization of data cache vulnerability in high-performance embedded microprocessors. In *Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (IC-SAMOS'06)*. IEEE, 14–20.
- WROBEL, F., PALAU, J., CALVET, M., BERSILLON, O., AND DUARTE, H. 2001. Simulation of nucleon-induced nuclear reactions in a simplified sram structure: Scaling effects on seu and mbu cross sections. *IEEE Trans. Nucl. Sci.* 48, 6, 1946–1952.

- ZHONG, H., FAN, K., MAHLKE, S., AND SCHLANSKER, M. 2005. A distributed control path architecture for vliw processors. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*. IEEE, 197–206.
- ZIVOJNOVIC, V., VELARDE, J., SCHLAGER, C., AND MEYR, H. 1994. DSPstone: A dsp-oriented benchmarking methodology. In *Proceedings of the International Conference on Signal Processing and Technology*.

Received June 2012; revised November 2012; accepted November 2012