Trade-offs in Execution Signature Compression for Reliable Processor Systems

Jonah Caplan*, Maria Isabel Mera†, Peter Milder†, and Brett H. Meyer*
* Electrical and Computer Engineering, McGill University, Montréal, Québec, Canada
{jonah.caplan@mail.mcgill.ca, brett.meyer@mcgill.ca}
† Electrical and Computer Engineering, Stony Brook University, Stony Brook, New York, USA
{maria.mera, peter.milder}@stonybrook.edu

Abstract—**As semiconductor processes scale, making transistors more vulnerable to transient upset, a wide variety of microarchitectural and system-level strategies are emerging to perform efficient error detection and correction computer systems. While these approaches often target various application domains and address error detection and correction at different granularities and with different overheads, an emerging trend is the use of state compression, e.g., cyclic redundancy check (CRC), to reduce the cost of redundancy checking. Prior work in the literature has shown that Fletcher's checksum (FC), while less effective where error detection probability is concerned, is less computationally complex when implemented in software than the more-effective CRC. In this paper, we reexamine the suitability of CRC and FC as compression algorithms when implemented in hardware for embedded safety-critical systems. We have developed and evaluated parameterizable implementations of CRC and FC in FPGA, and we observe that what was true for software implementations does not hold in hardware: CRC is more efficient than FC across a wide variety of target input bandwidths and compression strengths.**

I. Introduction

While advances in semiconductor manufacturing have made it possible to fabricate ever smaller, faster, and lower-power transistors, these trends result in an increase in transistor density and reduction in the critical charge Q_c required to disrupt a transistor. The transistors in mission- and safety-critical systems have thus become more vulnerable to errors introduced by cosmic rays, packaging radiation and thermal neutrons [1]. These errors are called single event upsets (SEU)—they occur once due to external stimuli, and are not repeatable—and may lead to complete system failure.

SEU have been addressed before through the use of sophisticated data redundancy such as error detecting codes (EDC) and error correcting codes (ECC) [2], and redundant computation, such as lockstep dual-modular redundancy (DMR) [3]. However, EDC and ECC come at high computational cost and area overhead [4]. Furthermore, lockstep is wasteful of compute resources when the DMR pair is not fully utilized executing safety-critical tasks, as is increasingly the case for mixed-criticality systems that devote some of their resources to the execution of non-critical tasks.

As a result, a number of techniques have emerged to make it easier to employ redundancy when needed but allow cores to execute in isolation otherwise, such as single- and multi-processor redundant threading [5], [6], and execution fingerprinting [7], [8]. These techniques utilize two or more threads executing on one or more processors for the purpose of redundancy checking, with an important caveat: unlike under lockstep execution, the threads need not execute at the same time, often yielding efficiencies. To support this, many such techniques compress the changes in processor state—such as store addresses and data—that must be compared to ensure coherent execution of the redundant threads. This compression reduces comparison bandwidth requirements at the expense of a small, tunable chance of aliasing and undetected error.

There are many design choices and trade-offs to consider when implementing hardware to compress execution state, such as: what state to compress; the compression algorithm used; and, the hardware implementation of the targeted compression algorithm, and other implementation details such as the size and organization of buffers.

In this paper, we focus on the selection and implementation of compression algorithms for this purpose. The selection of a compression algorithm and the microarchitectural details of how it is built is crucial—especially in the case of low-cost, low-power embedded safety-critical system. Beyond affecting traditional system design metrics (such as performance and power), the compression algorithm contributes to critical design metrics such as error detection latency (EDL) and error detection probability (EDP). Maxino and Koopman [9] investigated the tradeoffs of software performance and error detection probability for different algorithms, such as cyclic redundancy check (CRC) and Fletcher's checksum (FC). Given a generator polynomial, CRC takes a sequence of bits (the message) and performs polynomial division in the Galois Field GF(2); the remainder of this division is used as the *signature* characterizing the aggregate changes in state. Alternatively, FC divides the message into blocks and performs two different accumulations over the blocks; the resulting sums are concatenated and used as the signature. In the context of software implementations, Maxino and Koopman found that while implementations of FC are more computationally efficient, they are much less robust than CRC in terms of EDP.

We revisit the issue of algorithm selection in the context of hardware-based signature compression and observe that prior findings about the relative costs of Fletcher's checksum and CRC do not hold when implemented in hardware. Hardware-based compression operates under very different constraints than software-based approaches: (a) awkward arithmetic or bit-wise operations can be cost-effectively realized, but (b) storage is relatively expensive. In this paper, we focus on FPGA implementations of compression logic. Investigating

978-3-9815370-2-4/DATE14/©2014 EDAA

these trade-offs in the context of an ASIC design flow is the subject of future work; however, the importance of FPGAs in embedded systems cannot be understated, especially in the aerospace domain where manufacturing volumes remain low, and therefore ASICS are rare.

We design, synthesize, and evaluate a wide variety of CRC and FC signature compression circuits. For each, we consider a variety of implementation options, varying the checksum width as well as the data input rate. Our CRC implementations are based on the flexible design in [10], and our FC implementations are based on a novel parallelization technique we have developed, which supports scaling the input data rate with a linear increase in cost; the Verilog/VHDL generation scripts we developed are available at http://bhm.ece.mcgill.ca/~resc. Our evaluation shows that when implemented on an FPGA, CRC implementations are much more efficient than FC in terms of data throughput per unit cost.

II. Related Work

Surveys in the literature provide a valuable overview of the general area of fault tolerance. Overviews of the fundamental structures of fault-tolerant computing are available in [11]–[13], while more recent work [14] covers transient errors and architectures to mitigate them.

More specifically, fault-tolerant hardware can be classified by how redundancy is achieved; of relevance to the present work are approaches that employ either (a) microarchitectural or (b) system-level redundancy. Generally, microarchitectural techniques, such as error-correcting codes, achieve moderate coverage at high cost [4]. A variety of system-level techniques have been proposed to address this cost and take advantage of opportunities to share resources in the multi-core era.

One body of research employs redundant multi-threading to achieve redundancy [5]. Subramanyan, *et al.* reduce throughput losses in a multiprocessor when a redundant thread lags the leading thread by forwarding loaded values and branch outcomes [6]. Sloan and Kumar developed a framework that distributes voting logic to support efficient, dynamic nMR group formation in chip multiprocessors (CMPs) [15].

Another body of emerging research compares compressed state during redundancy checking. Argus employs CRC-6, amongst other things, to achieve low-cost single-core fault tolerance [16]. CRC is also used by Dynamic and Scalable DMR (DDMR, CRC-16) and Dynamic Core Coupling (DCC, 2x CRC-32) to compress state to support techniques that dynamically form pairs of redundant processors [17], [18]. Distributed temporal redundancy (DTR) also proposes to use CRC for fingerprint compression to allow statically scheduled safety-critical tasks to execute out of lockstep [8].

While [9] goes a long way towards characterizing the EDP of FC and CRC, their recommendations on which to use based on implementation cost do not carry over into the embedded hardware domain. Furthermore, [19] and [10] describe CRC designs that sacrifice circuit area to maximize the operating frequency by introducing pipelining and extremely wide circuits into the parallel design. Designers considering how best to include compression hardware in their systems may consequently reach the conclusion that CRC is prohibitively complex and expensive. To the best of our knowledge, ours is the first work developing a flexible parallel hardware implementation of Fletcher's checksum, allowing us to reassess the tradeoffs between the two compression algorithms in this context.

An evaluation of forward error-correcting codes, such as low-density parity check codes, which are generally more complex than backward error-correcting codes, is left for future work. Cryptographic hashes are also of interest, but are also out of scope due to their increased complexity relative to CRC and FC in the cost-constrained environment of embedded systems.

III. Execution Signature Compression

An execution signature compression system (ESCS) uses a hash function (*e.g.* cyclic redundancy check or Fletcher's checksum) to compress state changes (*e.g.* store addresses and data) or other information (*e.g.* checkpoints) into a single, fixed-width word called a signature. Often, this signature is then buffered for comparison against the signatures generated by a redundant thread executing on the same core at a later time or on another core (*a la* DCC [17] and DDMR [18]), or matched against a pre-computed signature (Argus [16]). If the signatures match, the redundancy check passes, and execution continues. If not, corrective action is taken, either in the form of executing an additional copy of a thread [8] or performing a rollback-and-recovery operation.

A. ESCS Design Space

Three key parameters affect the performance and cost of an ESCS: *input bandwidth*, *signature width*, and *algorithm*. Input bandwidth, the number of bits compressed in a cycle, is often related to the error detection latency (EDL) of an ESCS. Compressing more bits of state information in a single cycle allows erroneous state to enter the compression stream more quickly, reducing EDL. We will use the variable W to represent input bandwidth (in bits per clock cycle).

Increasing input bandwidth also increases ESCS cost (area); however, reducing input bandwidth beneath the number of bits that can potentially arrive for compression in a single cycle implies the use of input buffers, which are themselves costly. Prior work has shown that this buffering can vary from as little as a few kB to hundreds of kB, depending on the characteristics of the bitstream being compressed [20].

The signature width determines the compression strength; increasing it increases error detection probability (EDP) by reducing the chances that two different bitstreams alias. Increasing signature width generally increases the cost of an ESCS, as (a) more registers are needed to store the portion of the bitstream being operated on, and (b) maintaining the same bitstream throughput requires additional logic. We will use the variable M to represent signature width (in bits). To the first order, the error detection probability of a compression approach with an M-bit signature is $1 - 2^{-M}$.

Different algorithms behave differently in response to changes in the length of the compressed message or signature width. For example, 32-bit Fletcher's checksum achieves an error Hamming distance (EHD)—the minimum difference between two messages (in bits) that is not guaranteed to be detected—of 3 for messages up to and including 1,048,951 bits

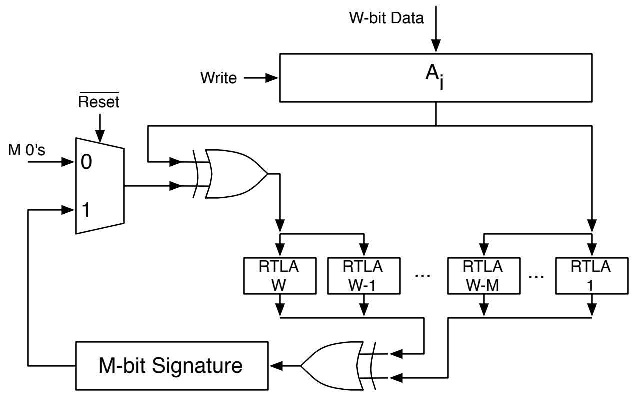

Fig. 1. Reduced lookup table algorithm (RTLA) implementation of CRC [10].

long [9]. Several 32-bit CRC polynomials, on the other hand, achieve an EHD of 6 for messages up to 32,738 bits long [21]. Safety-critical systems such as X-by-wire require an EHD of 6 [22]; a properly designed ESCS must meet the reliability requirements of a given application while minimizing cost overhead. In this work, we aim to provide designers with an understanding of the microarchitectural tradeoffs related to the implementation of signature compression units. A designer can then combine this understanding with an application-specific evaluation of error detection probability for his or her system (or rely upon general-purpose evaluations such as [9]).

B. Signature Compression with CRC

Cyclic redundancy check (CRC) is a coding technique that is often used for ensuring that errors—burst errors in particular—introduced to a bitstream can be detected. Given a generator polynomial, CRC performs polynomial division over GF(2), dividing a bitstream message by the generator, to compute the remainder. This remainder is used as the *signature*. After transmission, the division on the bitstream is performed again; if the remainder matches, the message was transmitted error-free. Otherwise, an error occurred, requiring that the message be retransmitted. A more detailed description of CRC can be found in the literature [22].

We implement a fast, parallel CRC circuit based on prior work [10], illustrated in Figure 1. This circuit has the dual advantages of performing the CRC operation in parallel on input data and supporting a variety of input data widths. The circuit takes an N-bit word to compress into an M-bit signature at a rate of W bits per clock cycle. Compressing an N-bit word therefore takes N/W clock cycles. When the first input block is presented to the circuit, the reset signal is asserted, clearing the previous signature and restarting the CRC calculation for the new data. The bits of the block A_i, which we will refer to as a_j, are used as select lines for Reduced Table Lookup Algorithm (RTLA) multiplexers. The RTLA multiplexers output zeros when $a_j = 0, j \in \{0, ..., m-1\}$. When $a_j = 1$, the multiplexers output the CRC of 2^j (which is pre-calculated and stored). All the RTLA signals are XORed together and then stored in the signature register. When the next W bits of data are presented to the circuit, reset is de-asserted, and the result of previous calculation is incorporated in the next stage by XORing it with the highest M bits of the new block. In our implementation we assume $W \geq M$. More detail on this technique is available in [10].

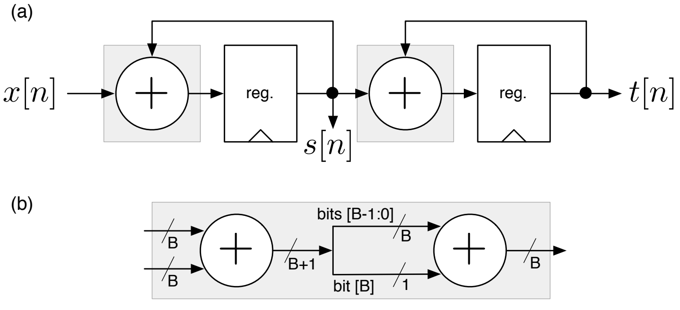

Fig. 2. (a) Serial implementation of Fletcher's checksum. (b) Structure of a B-bit ones' complement adder.

Based on this design, we have written a flexible hardware generation script that takes as input parameters W, M, and the generator polynomial, and produces the corresponding description as synthesizable VHDL. With this tool we can easily produce a variety of different implementations that exhibit different trade-offs. For instance, as W and M increase the circuit grows in size and its maximum frequency is reduced. Other published CRC designs such as [19] exhibit similar architectures that can be pipelined to reach higher frequencies, but require several times more hardware than our technique.

C. Signature Compression with Fletcher's Checksum

The Fletcher's checksum (FC) [23] of size M is a ones' complement integer algorithm that produces an M bit checksum from a series of $B = M/2$ bit inputs. Typically, FC is computed on serially-arriving data (where one B-bit input block arrives per clock cycle). Figure 2(a) illustrates such a serial implementation. During clock cycle n, one B-bit word ($x[n]$) enters the system, and two sums ($s[n]$ and $t[n]$) are calculated. First, $s[n]$ calculates the B-bit ones' complement sum of all current and prior values of $x[n]$. Then $t[n]$ calculates the B-bit ones' complement sum of all prior and current values of $s[n]$ (that is, the sum of sums). The M-bit checksum is the concatenation of $s[n]$ and $t[n]$.

B-bit ones' complement arithmetic can be viewed as unsigned arithmetic modulo $2^B - 1$, so the required operation is implemented (as shown in Figure 2(b)) with a B-bit adder, where the carry bit is added to the lower B bits of the sum. (We use a similar technique for ones' complement multiplication.)

The computation of FC maps naturally to a serial implementation. However, a designer may want to calculate an M-bit checksum at a rate faster than B input bits per clock cycle. In this case, it is desirable to parallelize the checksum computation so that more than one input word can be processed concurrently. Tight data dependencies make it appear difficult to parallelize this algorithm (for example, $s[k]$ cannot be computed until $s[k-1]$ is known). However, we are able to address this by computing several independent checksums in parallel on independent streams of data, and then merge the results such that the final checksum is equivalent to performing a single checksum serially on the same data. To the best of our knowledge, this technique has not been previously shown.

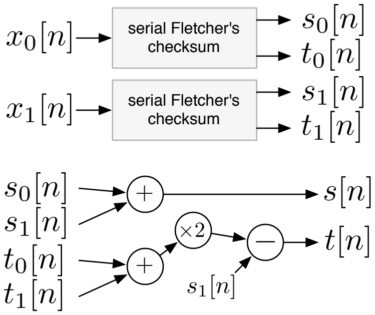

Fig. 3. $P = 2$ implementation of Fletcher's checksum.

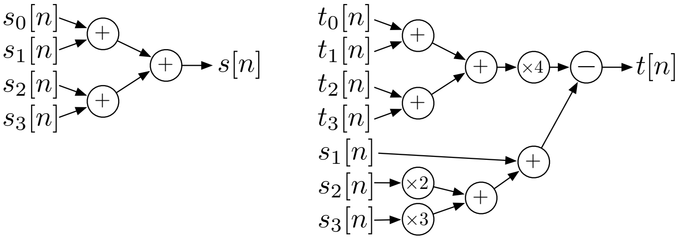

Fig. 4. Final merge steps of $P = 4$ implementation of Fletcher's checksum.

We wish to perform an M-bit checksum at a rate of W bits per clock cycle ($W \geq B$, where $B = M/2$), so we define parameter $P = W/B$ as the desired parallelism. Figure 3 illustrates our parallel approach for $P = 2$. First, we decompose our input stream $x[n]$ into even ($x_0[n]$) and odd ($x_1[n]$) parts, with resulting checksums ($s_0[n], t_0[n]$) and ($s_1[n], t_1[n]$). Then, we use these intermediate checksums to calculate our final checksum values as $s[n] = s_0[n] + s_1[n]$ and $t[n] = 2(t_0[n] + t_1[n]) - s_1[n]$. (All arithmetic operations are ones' complement.) The merging step of the $P = 2$ implementation requires three additional ones' complement adders and one constant multiplier (with constant 2).

As P increases, computation is performed in a similar way, as follows. First, the input stream is decomposed into P parallel streams (so that $x[0]$ goes into stream 0, $x[1]$ goes into stream 1, and so on). Then, the FC is computed for each of the P streams. We will label the P individual checksums as $(s_a[n], t_a[n])$, where $0 \leq a < P$. Then, the final $s[n]$ and $t[n]$ are computed from these terms as

$$s[n] = \sum_{k=0}^{P-1} s_k[n], \quad t[n] = P \sum_{\ell=0}^{P-1} t_\ell[n] - \sum_{m=1}^{P-1} m s_m[n]. \quad (1)$$

So, the final merging steps must (a) sum up all the intermediate values of $s_k[n]$, (b) sum up the intermediate values of $t_\ell[n]$ and scale them by P, (c) sum up scaled versions of $s_m[n]$, and lastly (d) subtract the result of (c) from the result of (b). These steps are illustrated for $P = 4$ in Figure 4. Straightforward evaluation of (1) confirms the correctness of the decomposition, and we have additionally verified the correctness in simulation.

The P-parallel FC has complexity $O(P)$, where we measure cost as the number of arithmetic units and registers required. More specifically, the P parallel serial checksums require $2P$ additions; the merging step requires $3P - 3$ additions and $P - 1$ multiplications by small constants (with constants $\leq P$). This parallel design allows a high amount of flexibility in the amount of pipelining to be used. The three summations in (1) each are implemented as adder trees with depth $\lceil \log_2 P \rceil$. They may be highly pipelined or implemented fully combinationally. If enough pipelining is used, the critical path will eventually become the feedback path within the serial modules (as in Figure 2(a)).

We have built a hardware generator that takes as input the parameters M and W, as well as parameter to control the pipelining, and produces a synthesizable Verilog implementation based on (1). Our generator currently assumes that W is a power-of-two multiple of B, although this restriction can be relaxed to any integer multiple.

IV. Experimental Setup

We conducted experiments to compare the relative cost and performance of different implementations of CRC and FC. In each case, we explored how parameters W (input bandwidth) and M (signature width) affect the system's implementation costs and throughput. For each set of design parameters, we use our generation scripts to produce a design, synthesize it targeting an Altera FPGA, and determine the resource consumption, and the maximum clock frequency.

A. CRC Polynomials

The generator polynomial used with CRC is crucially important to determining the probability of error detection. For signature widths 8, 16, and 32, there are several polynomials known to be be good choices for general purposes [21], [24]. However there is no similar peer-reviewed research on polynomials of other lengths. Further complications occur because the properties of a given polynomial depend on the length of messages to be compressed. That is, some polynomials give higher Hamming distance at smaller message lengths but perform poorly compared to alternatives at higher lengths. However, the specific polynomial has no effect on the throughput rate achieved, and only a very small effect on the system's logic cost. For example, we evaluated a range of 32-bit polynomials, implemented with $M = 32$, and observed less than 5% difference in the amount of logic required. For this reason, we have chosen not to focus on the impact of polynomial selection on circuit cost. Our current designs are built and customized for a single polynomial given at design time, although they could easily be modified to produce a system that allows runtime selection from among several polynomials (with an increase in design cost). In this experiment, we used the following polynomials: 0x8e [24], 0xb75 [24], 0xcbe5 [24], CRC-24, 0xf4acfb13 [21], and CRC-64-ECMA-182.

B. Design Simulation and Synthesis

In order to validate the correctness of our CRC and FC designs, we simulated them with random inputs, and compared the results with software-computed checksums. To evaluate implementation costs and performance, we synthesized each design using Altera Quartus II targeting an Altera Arria II GX FPGA (EP2AGX45DF29C5). After using Quartus II to run the procedures for analysis, synthesis, fitting (place and route), and static timing analysis, we determined: (i) the number of adaptive LUTs (ALUTS) used (ii) the number of registers used,

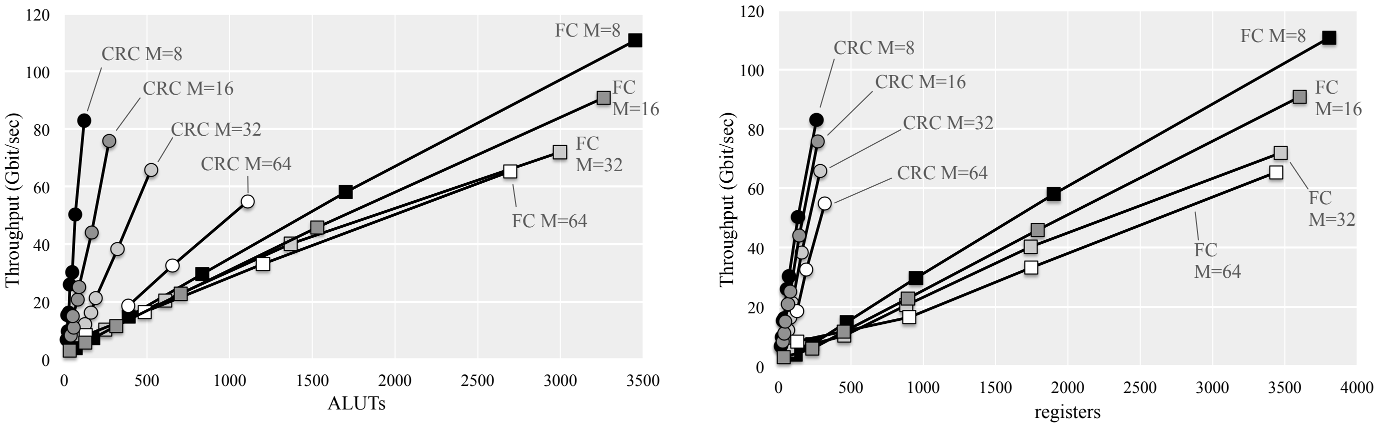

Fig. 5. Comparison of throughput versus ALUTs (left) and throughput versus registers (right) for $M = 8, 16, 32$, and 64 for CRC and FC.

and (iii) the maximum clock frequency. Then, we determined the system's throughput by multiplying the clock frequency by W, the number of input bits per clock cycle.

V. Results

The results of our experiment are summarized in the two graphs shown in Figure 5. In both graphs the y-axis shows the throughput (input data rate, equal to W times the clock frequency) in gigabits per second. The x-axis of each graph shows an area cost metric—the left graph illustrates the logic cost in the number of Adaptive Look-Up Tables (ALUTs), which are combinational logic blocks, and the right graph's x-axis indicates the number of registers required. Separate lines are shown for signature widths $M = 8, 16, 32$, and 64, and for both CRC and Fletcher's checksum (FC). For both checksums, we evaluate all legal values of W (the number of input bits processed per cycle) in $(4, 8, 12, 16, 24, 32, 48, 64, 128, 256)$. (Recall that our CRC implementation can process any value of $W \geq M$ and our FC implementation requires $2W/M$ to be equal to a power of two ≥ 1.) In each line, the smallest value of W is located in the lower-left corner, and increasing values of W proceed above and to the right.

We observe that a CRC design has much lower cost than its FC equivalent. One can quantify the efficiency of designs in terms of throughput per ALUT and throughput per register. Within the range of parameters considered here, the CRC units are 1.4–22× more efficient in terms of throughput per ALUT than their equivalent FC design, and they are between 8.0–15× more efficient in terms of throughput per register.

Given a checksum method and a value of M (that is, a line on the graph), as W increases, the designs exhibit a roughly linear relationship between throughput and cost (matching our expectations). The line's slope provides a visual indication of the relative area cost needed to increase throughput; the steeper the slope is, the less expensive it is to reach higher data rates. The slope of a line depends on the checksum method and the signature size (M). Naturally, larger signatures require more operations to be performed (XORs and lookups in CRC, and wider additions in FC), and thus designs with larger M require more resources to reach a given throughput. However, the choice of checksum (CRC or FC) is much more important than M in determining the cost and performance of the design.

TABLE I. Number of ALUTs, number of registers, and throughput, averaged over design space, normalized to CRC.

	ALUTs	registers	throughput
CRC	1	1	1
FC (minimal pipelining)	4.9	2.5	0.22
FC (intermediate pipelining)	4.9	5.4	0.63
FC (full pipelining)	5.5	11.8	0.88

In fact, the line for CRC with $M = 64$ has a more efficient slope than even the $M = 8$ FC design (though its maximum throughput is more limited).

Although it is shown only indirectly in these graphs, clock frequency is a crucial factor in determining system throughput. For CRC implementations, the clock frequency in this design space varies from 214 MHz (for $M = 64$, $W = 256$) to 844 MHz (for $M = 8$, $W = 8$). For FC, the clock frequency varies from 255 MHz (for $M = 64$, $W = 256$) to 489 MHz (for $M = 8$, $W = 8$).

Our FC implementation is flexible with respect to the amount of pipelining. The values in Figure 5 use the maximum number of pipeline stages. However, when we set our generation tool to use fewer pipeline stages, we obtain designs with roughly the same number of ALUTs, but varying values of throughput and register count. To quantify this, Table I gives average number of ALUTs and registers and the average throughput of FC designs with minimum, maximum, and an intermediate level of pipelining; all values are normalized with respect to CRC for ease of comparison. These comparisons are based only on the design parameters where both our CRC and FC implementations are possible. (Recall, our FC implementations can support a smaller W than our CRC implementations, but they are more restricted in supported values as W grows.) We observe that even with reduced pipelining, the ALUT and register cost of FC remains several times higher than that of CRC.

When implemented in software, FC is typically considered to be a computationally cheaper alternative to CRC that is appropriate when less robust error detection properties are acceptable [9]. It is intuitive why FC is a faster algorithm when implemented in software—its basic operation (addition) is well suited to run on a processor when the addition's data width is compatible with the processor's arithmetic unit. Meanwhile, the XOR operations and the dataflow required by CRC lead to significant overheads. Our results show that when implemented in an FPGA, the situation is reversed: CRC achieves much higher efficiency than FC. The same XOR operations that are problematic in software map well to look-up tables in an FPGA, while adders are relatively more expensive.

VI. Conclusions and Future Work

System-level redundancy techniques are emerging as cost-effective solutions for multi-core systems under growing vulnerability to transient upset. Such systems often employ signature compression to facilitate rapid redundancy checking. However, designers often fail to consider the tradeoffs inherent in embedded implementation of signature compression. We advocate a systematic approach to evaluating the relevant design space, taking into account how parameters such as the compression algorithm chosen, the width of its signature, and microarchitectural options affect cost and performance.

Using our generation tools, we evaluated a wide variety of CRC and FC implementations targeting an Altera Arria II GX FPGA. For each resulting design we evaluated (a) logic element usage, (b) register usage, and (c) compression throughput. As shown in Figure 5 and Table I, with respect to the above metrics, CRC greatly outperforms FC for the parameters evaluated (signature width M from 8 to 64 bits and input bandwidth W from 8 to 256 bits per clock cycle).

Typically CRC is viewed (in the context of software) as a more-expensive alternative to FC that results in stronger error detection. Our results therefore reexamined this well-studied trade-off, and show that when implemented as embedded hardware, the relative costs of these alternatives do not follow conventional wisdom. The ease with which XORs and lookup tables can be implemented in hardware results in a clear efficiency advantage for CRC in terms of throughput per unit cost; in our results CRC implementations were 1.4–22× more efficient than FC in terms of throughput per ALUT, and 8.0–15× more efficient in terms of throughput per register.

Our future work aims to improve understanding of the benefits and costs of execution signature compression systems. To do so we will explore in two directions.

First, we will consider a wider variety of compression algorithms and implementations and examine how trade-offs (including energy) differ when implemented in other technologies such as ASIC. For example, in an ASIC-based CRC design, run-time polynomial selection will likely become more important, leading to additional costs. Further, our CRC implementation's lookup-table-based structure fits extremely well into an FPGA's reconfigurable logic elements; implementation in an ASIC design flow may reverse this, leading to further interesting and important considerations.

Second, we will study at a system-level how other options (such as the choice of which aspects of system state to compress) affect design issues such as aliasing probability, detection latency, and system cost. By creating a framework to ease evaluation and implementation of these high level decisions, we will allow designers to intelligently optimize and understand the implications of these choices on system reliability, performance, and cost.

Acknowledgments

This work is supported by: the Semiconductor Research Corporation (SRC) through gift 2013-RJ-2371G; the Natural Science and Engineering Research Council of Canada (NSERC) through grant RGPIN 418639-12 and the NSERC Undergraduate Student Research Award Program; the McGill University Faculty of Engineering Chwang-Seto Faculty Scholar Research Award.

References

[1] R. C. Baumann, "Radiation-induced soft errors in advanced semiconductor technologies," *IEEE Trans. Device Mater. Rel.*, vol. 5, no. 3, 2005.

[2] J. Kim, N. Hardavellas, *et al.*, "Multi-bit error tolerant caches using two-dimensional error coding," in *ISCA-40*, 2007.

[3] M. Baleani, A. Ferrari, *et al.*, "Fault-tolerant platforms for automotive safety-critical applications," in *CASES'03*, 2003.

[4] L. G. Szafaryn, B. H. Meyer, *et al.*, "Evaluating overheads of multibit soft-error protection in the processor core," *IEEE Micro*, 2013.

[5] E. Rotenberg, "AR-SMT: a microarchitectural approach to fault tolerance in microprocessors," in *FTCS'99*, 1999.

[6] P. Subramanyan, V. Singh, *et al.*, "Multiplexed redundant execution: A technique for efficient fault tolerance in chip multiprocessors," in *DATE'10*, 2010.

[7] J. C. Smolens, B. T. Gold, *et al.*, "Fingerprinting: bounding soft-error detection latency and bandwidth," in *ASPLOS'04*, 2004.

[8] B. H. Meyer, B. H. Calhoun, *et al.*, "Cost-effective safety and fault localization using distributed temporal redundancy," in *CASES'11*, 2011.

[9] T. C. Maxino and P. J. Koopman, "The effectiveness of checksums for embedded control networks," *Dependable and Secure Computing, IEEE Transactions on*, vol. 6, no. 1, pp. 59–72, 2009.

[10] H. F. A. Hamed, F. A. Elmisery, *et al.*, "Implementation of low area and high data throughput CRC design on FPGA," *IJARCSEE*, vol. 1, no. 9, 2012.

[11] V. Nelson, "Fault-tolerant computing: fundamental concepts," *Computer*, vol. 23, no. 7, July 1990.

[12] V. Prasad, "Fault tolerant digital systems," *IEEE Potentials*, vol. 8, no. 1, February 1989.

[13] F. Cristian, "Understanding fault-tolerant distributed systems," *Commun. ACM*, vol. 34, no. 2, 1991.

[14] S. Mukherjee, *Architecture Design for Soft Errors*. Morgan-Kaufmann, 2008.

[15] J. Sloan and R. Kumar, "Towards scalable reliability frameworks for error prone CMPs," in *CASES'09*, 2009.

[16] A. Meixner, M. E. Bauer, *et al.*, "Argus: Low-cost, comprehensive error detection in simple cores," in *MICRO-40*, 2007.

[17] C. LaFrieda, E. Ipek, *et al.*, "Utilizing dynamically coupled cores to form a resilient chip multiprocessor," in *DSN'07*, 2007.

[18] A. Golander, S. Weiss, *et al.*, "DDMR: Dynamic and scalable dual modular redundancy with short validation intervals," *IEEE Computer Architecture Letters*, vol. 7, no. 2, 2008.

[19] M. Walma, "Pipelined cyclic redundancy check (CRC) calculation," in *ICCCN'07*, 2007.

[20] B. H. Meyer, M. Liu, *et al.*, "Rapid, tunable error detection with execution fingerprinting," in *SAE 2013 AeroTech*, 2013.

[21] P. Koopman, "32-bit cyclic redundancy codes for internet applications," in *DSN'02*, 2002.

[22] J. Ray and P. Koopman, "Efficient high hamming distance CRCs for embedded networks," in *DSN'06*, 2006, pp. 3–12.

[23] J. G. Fletcher, "An arithmetic checksum for serial transmissions," *IEEE Transactions on Communications*, vol. COM-30, no. 1, January 1982.

[24] P. Koopman, "CRC selection for embedded network messages," https://www.ece.cmu.edu/~koopman/crc/, 2004.