Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2014, Article ID 724585, 8 pages
http://dx.doi.org/10.1155/2014/724585

Research Article

Using Statistical Assertions to Guide Self-Adaptive Systems

Tim Todman,[1] Stephan Stilkerich,[2] and Wayne Luk[1]

[1] *Department of Computing, Imperial College London, 180 Queen's Gate, London SW7 2AZ, UK*
[2] *Software Engineering, EADS Innovation Works, Willy-Messerschmitt Street 1, 85521 Ottobrunn, Germany*

Correspondence should be addressed to Tim Todman; tjt97@doc.ic.ac.uk

Received 7 January 2014; Accepted 4 March 2014; Published 13 April 2014

Academic Editor: Marco D. Santambrogio

Copyright © 2014 Tim Todman et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Self-adaptive systems need to monitor themselves, to check their internal behaviour and design assumptions about runtime inputs and conditions. This kind of monitoring for self-adaptive systems can include collecting statistics about such systems themselves which can be computationally intensive (for detailed statistics) and hence time consuming, with possible negative impact on self-adaptive response time. To mitigate this limitation, we extend the technique of in-circuit runtime assertions to cover statistical assertions in hardware. The presented designs implement several statistical operators that can be exploited by self-adaptive systems; a novel optimization is developed for reducing the number of pairwise operators from $O(N)$ to $O(\log(N))$. To illustrate the practicability and industrial relevance of our proposed approach, we evaluate our designs, chosen from a class of possible application scenarios, for their resource usage and the tradeoffs between hardware and software implementations.

1. Introduction

Self-adaptive systems can configure themselves to flexibly deal with changing environments after they are deployed. The configuration itself is systematically guided by means of system self-monitoring to aid decisions about changing modes or to check design assumptions about runtime data and conditions or their internal operation. Such monitoring could check elementary Boolean conditions or, more generally, could process collected runtime system data, feeding a process of deciding whether or how the system can be adapted. The response time to adaptation is a fundamental feature characterizing self-adaptive systems. For the class of applications from the avionics domain we investigate, a fast response time to adaptation is crucial and motivates our advocated approach, presented in the rest of the paper. Gathered system data can be used for many purposes; for example, design assumptions about ranges of input values, used to optimize operator bit-widths, can be checked by assertions about the standard deviation of the input.

In this paper, we propose *in-circuit statistical assertions*, compiled into the hardware part of a software-hardware design as a dedicated self-monitoring facility for self-adaptive systems, with a fast response time to adaptation. Compared to the proposed in-circuit assertions that can compute in parallel with the rest of the design, assertions based on sequential software need to wait until the hardware has finished computing its results before they can begin their own tasks. Moreover, efficient hardware designs are often deeply pipelined, operating on large batches of data, further prolonging the time until software assertions can start processing. Additionally, by preprocessing potentially large amounts of data, in-circuit data gathering can improve use of limited bandwidth between hardware and software of the self-adaptive system triggering and controlling system adaptation. In summary, in-circuit assertions are the necessary precondition to realize fast response times to adaptation not achievable by sequential software assertions.

Figure 1 provides a structural overview of our approach. A hardware datapath is instrumented by in-circuit statistical operators which compute relevant statistics about the design. These are then sent back to a software engine running a self-adaptive system. The software builds up the self-adaptive representation which is used to control reconfiguration of the system. It should be mentioned that whilst we target a software-hardware system setting, our approach is not limited to this setting at the outset. The software could

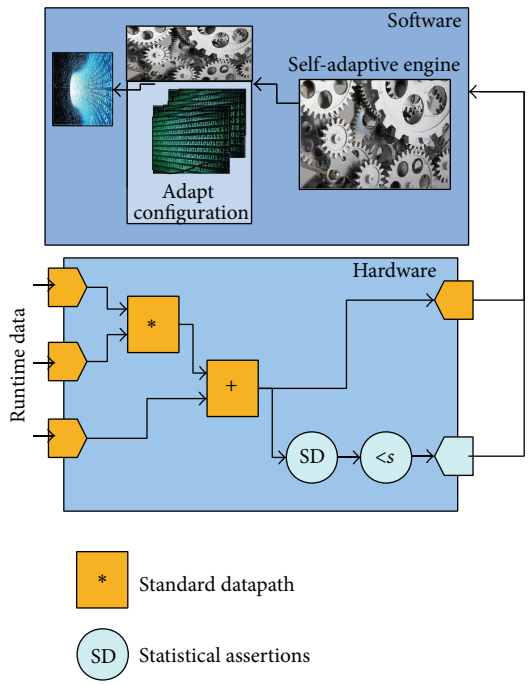


Figure 1: Our approach: hardware datapath augmented with in-circuit statistical assertions feeding an engine running a self-adaptive algorithm in software.

likewise run, for example, on a soft processor within a field programmable gate array (FPGA) fabric.

This paper makes the following contributions:

(i) the design and optimized implementation of in-circuit statistical assertions, which can be used by self-adaptive systems to monitor themselves and control system adaptation;

(ii) a case study on avionics systems, showing the potential of in-circuit statistical assertions;

(iii) evaluation of tradeoffs between assertion implementations in software and in hardware, showing the advantages of our proposed in-circuit assertions.

Compared to our previous work [1], this paper adds two new architectures for statistical assertions and extends the avionics case study with an implementation of true airspeed, an important instrument for many avionics applications; we apply our statistical assertions to this implementation.

The rest of the paper is organized as follows. Section 2 describes related work. Section 3 shows our designs for assertions and implementations for Maxeler systems. Section 4 is a brief case study for avionics. Section 5 evaluates our implementation. Section 6 concludes and suggests future research.

2. Background

Runtime Verification. Several researchers have used temporal logic for runtime verification; for example, Reinbacher et al. [3] implement hardware temporal logic monitors for a software system running on a soft processor on the same device. Calinescu et al. [4] propose that self-adaptive software needs quantitative runtime verification; our statistical in-circuit assertions could complement such approaches.

Assertion-based verification allows the use of Boolean and temporal assertions for debugging designs in simulation [5]; it is extended to in-circuit assertions by Curreri et al. [6]. This paper introduces in-circuit assertions with statistical operators. In our approach, failed assertions do not necessarily indicate errors but may be the trigger for a self-adaptive system to adapt or reconfigure itself.

Statistical assertions have been proposed by Dinh et al. [7] to allow users to reason about large parallel programs (at debug time) using derived metrics, rather than raw program output. The assertions are implemented efficiently using a map-reduce style computation. We use statistical assertions for runtime monitoring of reconfigurable hardware-accelerated systems.

3. In-Circuit Statistical Assertions

This section presents our approach to in-circuit statistical assertions and their implementation.

3.1. Assertion Language. Our assertion language comprises C language style Boolean operators, augmented by statistical primitives. We choose the C language as it is familiar to many designers. The set of statistical primitives is as follows:

(i) mean(e), the mean value of expression e;

(ii) stdev(e), the standard deviation of expression e;

(iii) variance(e), the variance of expression e.

We choose these primitives as a useful set for expressing statistical conditions; future work could add further statistical operators such as covariance, skewness, and kurtosis or limit the number of cycles over which the statistics are calculated, potentially reducing hardware resources.

The following shows the grammar of our statistical assertions language in extended Backus-Naur form:

```
e   =  a
     |  e  bop  e
     |  uop  e
     |  mean(e)
     |  stdev(e)
     |  variance(e)
bop  =  ==  |  !=  |  <  |  >  |  ···
uop  =  +  |  −  |  !  |  ¯,
```
(1)

where bop represents any C binary operator, uop any C unary operator, and a any atomic expression (literals, variables, and

constants). This language allows the user to combine both Boolean and statistical operators.

Notation. We adopt the notation of Chan et al. [2] given data points x_i to x_j calculate the sum $T_{i,j}$, the mean $M_{i,j}$, and the sum of square differences from the mean $S_{i,j}$ as:

$$T_{i,j} = \sum_{k=i}^{j} x_k$$
$$M_{i,j} = \frac{1}{(j-i+1)} T_{ij} \tag{2}$$
$$S_{i,j} = \sum_{k=i}^{j} \left(x_k - M_{ij}\right)^2.$$

3.2. Architectures for Assertion Operators. We propose four architectures suitable for streaming systems: both *feedforward* and *feedback* architectures of *online* and *pairwise* algorithms. Each design has different properties which can be useful for different applications of statistical operators. All of the designs are based on statistical building blocks which we call *S-operators*; the same design can be used with different *S*-operators to build different statistical operations such as sum, mean, and variance.

Online algorithms for calculation of statistical metrics such as mean, variance, and standard deviation are known [8, 9]. They involve a single pass over the input data, using an accumulator and the current input element. While such designs may seem suitable for streaming implementations, they contain feedback owing to the accumulator, requiring the design to use techniques such as C-slowing and unrolling to achieve a reasonable clock speed.

Online algorithms can be expressed as a set of recurrence equations calculating the sum, mean, and sum of square differences in terms of the current input x_n and the sum, mean, or sum of square differences for the previous $n-1$ inputs [2]:

$$T_{1,0} = 0$$
$$T_{1,n} = x_n + T_{1,n-1}$$
$$M_{1,0} = 0$$
$$M_{1,j} = M_{1,j-1} + \frac{1}{j}\left(x_j - M_{1,j-1}\right) \tag{3}$$
$$S_{1,0} = 0$$
$$S_{1,j} = S_{1,j-1} + \left(j-1\right)\left(x_j - M_{1,j-1}\right)\left(\frac{x_j - M_{1,j-1}}{j}\right).$$

We design both feedforward and feedback architectures for the online algorithms. Figure 2 shows a feedforward implementation for part of the calculation. The design can calculate sum, mean, and sum of square differences by changing the *S*-operator. Note that the design calculates a large majority of the result; the rest can be calculated in software. The design consists of W repeating units, costing $O(W)$ area, and reducing output volume by factor $W+1$.

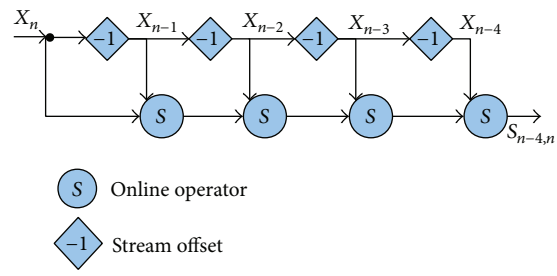


Figure 2: Partial calculation of statistics using feedforward online operators. There are W repeating units; in this diagram, $W = 4$.

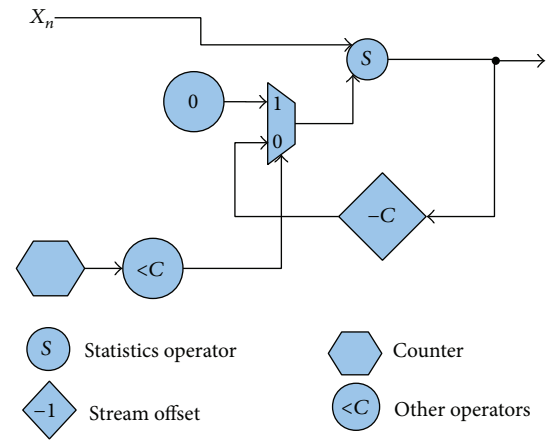


Figure 3: Partial calculation of statistics using feedback online operators.

Figure 3 shows a feedback version of the online algorithm. Feedback feeds partial results from the *S*-operator back into its input, reducing the total output size to C (the pipeline length of the *S*-operator), at an area cost of $O(1)$ (a single *S*-operator), given that the $-C$ and $< C$ operators are of negligible size.

Chan et al. developed pairwise algorithms for sum, mean, and variance [2] which can be parallelized; for W input elements, naively implementing this algorithm on streaming systems would require $O(W)$ hardware, as shown in Figure 4. Unlike the online algorithms, pairwise algorithms use a divide-and-conquer approach which Chan et al. show to give better numeric stability than online algorithms. For example, their pairwise algorithm calculates the sum of square differences as

$$S_{1,2m} = S_{1,m} + S_{m+1,2m} + \frac{1}{2m}\left(T_{1,m} - T_{m+1,2m}\right)^2. \tag{4}$$

Figure 4 shows the datapath for a straightforward feedforward design, combining stream offsets with Chan's pairwise operators for calculating variance or mean; for clarity, we omit the calculation of $T_{i,j}$, which has the same pattern.

We optimize the feedforward pairwise design using the observation that, in a streaming system, iterating through the input data in order, sums of neighbouring elements

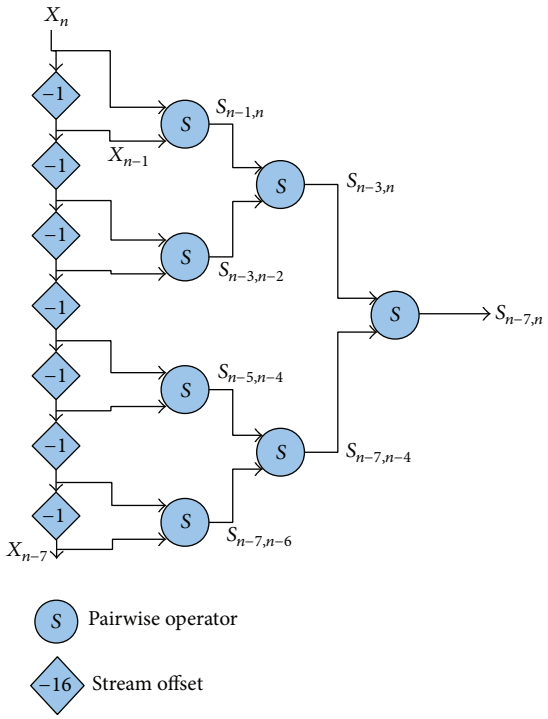


FIGURE 4: Partial calculation of statistics using feedforward pairwise operators: naive implementation of Chan et al.'s algorithm [2], in hardware.

can be accessed by stream offsets, which are mathematically equivalent to

$$\text{offset}\left(T_{m,m+b}, -o\right) = T_{m-o,m+b-o}. \tag{5}$$

In Figure 4, note that $S_{n-3,n-2}$ is simply $S_{n-1,n}$ delayed by two cycles. Adapting Chan et al.'s notation,

$$\begin{aligned} T_{1,2^k} &= T_{1,2^{k-1}} + T_{2^{k-1}+1,2^k} \\ &= T_{1,2^{k-1}} + \text{offset}\left(T_{1,2^{k-1}}, -2^k\right), \end{aligned} \tag{6}$$

where $\text{offset}(e, n)$ means the value of expression e sampled n cycles in the past; so, for example, $T_{i-3,i-2} = \text{offset}(T_{i-1,i}, -2)$; $S_{1,2^k}$ is calculated in the same way; Figure 5 shows our optimized design. Unlike the straightforward implementation of Chan et al.'s algorithm, which requires $O(W)$ hardware, our optimized design requires only $O(\log_2 W)$ statistical operators plus $O(W)$ delay elements used to implement the offset operation.

Note that the above only calculates part of the variance, specifically the local variance around each sample; however, it greatly reduces the amount of data sent back to software. The design consists of repeating units of the pairwise operator and stream offsets to delay the input. Each repeating unit reduces by half both the output data and the remaining calculations to be done in software, so U units reduce it 2^U-fold. Note furthermore that the leftmost operator can be optimized, because $S_{i,i} = 0$ (the variance of a single point is zero) and $M_{i,i} = x_i$.

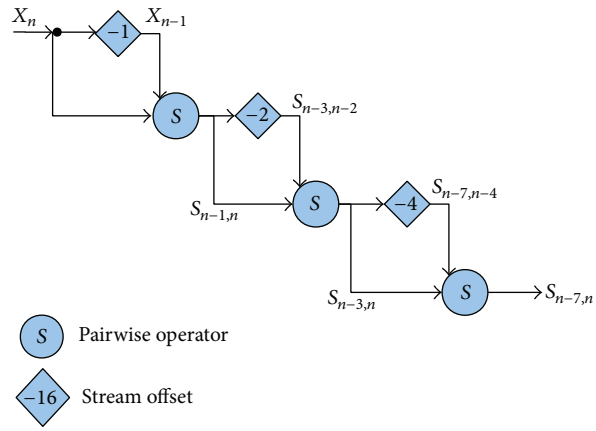


FIGURE 5: Partial calculation of statistics using feedforward pairwise operators, optimized for streaming systems. Compared to Figure 4, this design uses $O(\log N)$ instead of $O(N)$ S-operators.

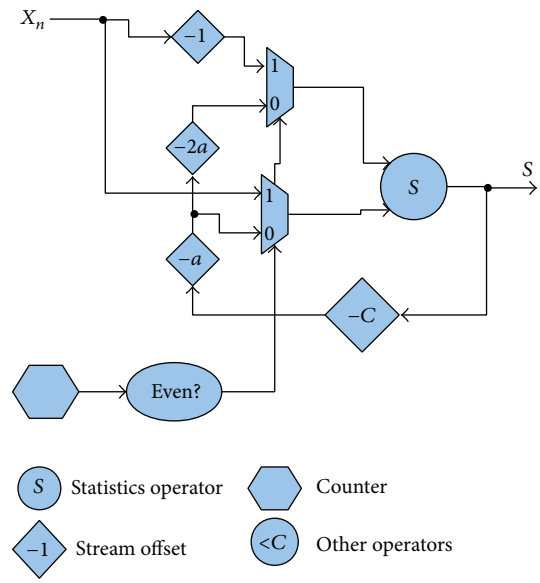


FIGURE 6: Partial calculation of statistics using feedback pairwise operators. The two multiplexors share a control input; stream offsets labelled $-a$ and $-2a$ are variable.

In addition, we extend the optimized pairwise algorithm to a feedback architecture (Figure 6); this essentially realizes the spatial structure of Figure 4 in time. Consequently, this requires more buffering than the feedback online architecture, $O(W)$ for summarizing W inputs. The amount of buffering also has to vary with time, as shown in Figure 6.

Table 1 summarizes the design properties of the different operator designs.

3.3. Implementation Targeting Maxeler Designs. We choose Maxeler streaming systems to implement our designs, though the approach is not Maxeler specific and can be ported to other design descriptions such as Verilog and VHDL. We focus on a systematic approach to translating assertions into Maxeler designs; future work includes developing a

Table 1: Summary of different operator properties.

Architecture	Algorithm	Storage	Compute	Outputs
Feedback	Online	$O(1)$	$O(1)$	C
Feedback	Pairwise	$O(\mathrm{W})$	$O(1)$	N/W
Feedforward	Online	$O(W)$	$O(W)$	N/W
Feedforward	Pairwise	$O(W)$	$O(\log_2 W)$	N/W

tool for compiling Maxeler designs extended with statistical assertions into the current base language.

The Maxeler system generates *streaming* designs, where inputs and outputs are large arrays used as streams. Each output element is calculated from corresponding elements in one or more input streams; offsets allow reading from neighbourhood stream elements. The user programmatically builds a datapath using a domain-specific language based on Java. The control path can involve counters or state machines generated from another domain-specific language.

Maxeler tools compile designs into hardware description languages and control FPGA vendor tools to build the corresponding bitstream for a specific FPGA device. Software can interact with the generated hardware using a Maxeler application programming interface to configure the FPGA device with the bitstream and run on user data stored in C arrays. The Maxeler tools automatically pipeline the datapath, resulting in deeply pipelined operators at a high clock rate. This works well for feedforward designs, but feedback requires some manual intervention and reordering or duplicating of input data.

4. Case Study: Avionics Systems

Avionics systems are electronic systems used for control or information in the aviation or aerospace industries [10].

Self-adaptive systems with a fast response to adaptation (where fast means quicker than 500 ms) are promising architectures for dedicated application scenarios in the avionics and space-flight industry. Systems that profit from architectures with fast response time to adaptation include

(i) autonomous flying systems,

(ii) special satellites,

(iii) deep-space mission systems,

(iv) exploratory space mission systems.

All these systems operate in environments that cannot be fully described at design time and hence such systems cannot be statically designed to cover and handle all environmental settings. Furthermore, these systems have strong constraints on power consumption, weight, and packaging volume. Additionally, these systems may never be physically reachable after deployment.

We choose a 500 ms limit as this duration meets the requirements of many processing and control loops of the systems and application scenarios mentioned in the paper. Hence, if we realize our self-adaptation and self-expression with the configuration within this limit, it would fit into our proposed self-adaptive systems, applications, and current systems that can benefit from self-monitoring.

We analyze the processing structure of these systems for the functionality of guidance, navigation, and orientation, revealing that the processing is commonly composed of different blocks/kernels with inputs and outputs. Determining the adequate bit widths and hence precision for the inputs and outputs is difficult and is often based on worst case assumptions involving unnecessary resources. An alternative is to start with an initial, more optimistic design assumption about the input/output value range, used to optimize operator bit widths. Such assumptions can be checked by assertions about the standard deviation of the input and adapted by another kernel version accordingly if required. Obviously, fast response time to adaptation is to avoid compromising system functionality, while simultaneously optimizing the system at runtime with respect to performance, energy efficiency, and environmental adaptability.

Our case study involves true airspeed calculation. Calculating the true airspeed from external sensor signals (Pitot tubes) and continuously providing a correct value of the true airspeed to the avionic computers is of critical importance for safe flying, navigation, and air-traffic operation. This calculation affects aircraft and helicopters operated manually as well as autonomous unmanned aerial vehicles (UAVs). Due to the safety relevance of this particular calculation and the overall unalterable structure of today's avionic systems, it is common practice to redundantly (triple) realize the calculation of the true airspeed, including the sensors. With respect to the sensors, the redundant approach is obvious and well justified, but for the calculation of the true airspeed a redundant approach is, in most flight phases, an overdesign and consequently a waste of computing resources. For civilian and military aircraft and helicopters, this approach can currently be tolerated and technically realized. However, for UAVs with envisaged long operation times, a predefined and fixed architecture wastes computing resources, directly impacts on weight and fuel consumption, and consequently negatively influences operation time.

Our proposed approach offers a twofold strategy and technical solution to systematically address the calculation, with respect to the true airspeed, especially providing a technically attractive solution for UAVs. In detail, hardware assertions will allow for single realisation of the true airspeed calculation in flight phases with stable and predictable weather conditions. Possible failures in the calculation can be identified by the assertions with counteractions of using the old value or starting reconfiguration actions if the failure remains permanent. Statistical assertions evaluation can be utilized to predict upcoming problems and to start counteractions before the failure appears and affects the system. Reconfiguration itself, within our defined timeframe of 500 ms, enables flexible adjustment of system to adapt to identified failures or simply to different flight phases, where, for instance, failures of the true airspeed are not acceptable and double or triple modular redundancy is, for that situation in time, the best solution. Consequently, our proposed approach can adjust our calculation of the true

airspeed to suit different flight situations and can utilize the computing resources efficiently.

For example, given inputs of static and impact pressures, the formula for calculating true airspeed V_t is as follows (if the Mach number is not known) [11]:

$$V_t = a_0 \cdot \sqrt{5\left[\left(\frac{q_c}{P}+1\right)^{2/7}-1\right]\cdot\frac{T}{T_0}}, \quad (7)$$

where

(i) a_0 is the speed of sound at standard sea level;

(ii) q_c is the impact pressure;

(iii) P is the static pressure;

(iv) T is the temperature;

(v) T_0 is the standard sea level temperature.

Note the units of measurement are governed by the units of a_0; for navigation purposes the units of speed are knots.

Implementation. We implement the true airspeed as a streaming Maxeler design, using 32-bit floating point arithmetic for internal calculation precision. Since the Maxeler library does not provide general exponentiation operators, we substitute the equivalent calculation using logarithms.

Figure 7 shows our implementation of the datapath for the true airspeed calculation. Statistical assertions can be added to each variable input (q_c, P, and T) and to each connector within the datapath.

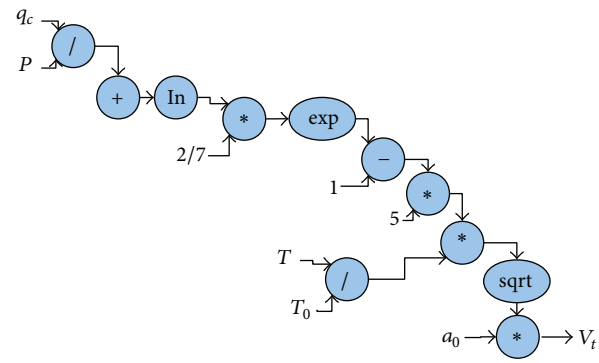


Figure 7: Datapath of streaming implementation of true airspeed. Each connector is a potential point to add statistical assertions.

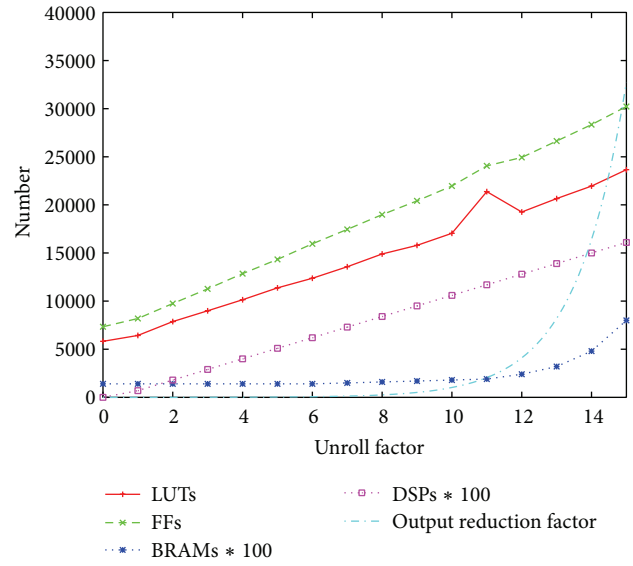


Figure 8: Area usage and output reduction versus unroll factor for the feedforward pairwise variance operator.

5. Evaluation

We evaluate our implementations of on-chip statistical assertions showing the tradeoff between hardware and software implementations. We compare

(i) scalability: operator size versus hardware size;

(ii) software versus hardware-assisted results: speed, bandwidth.

Experimental Setup. We implement our designs using Maxeler compiler version 2012.1 and Xilinx ISE 13.1. Designs target a MAX3 system, containing a Xilinx xc6vsx475t FPGA, with a speed goal of 200 MHz. We implement one or more variance assertions, with 32-bit input data in IEEE single-precision (SP) floating-point format, one data element per cycle. The variance assertion is the largest and most computationally-intensive of all the operators we design and implement.

Figure 8 shows the effect of unroll factor on the area resources for the feedforward pairwise variance operator; the area is measured in numbers of look-up tables (LUTs), flip flops (FFs), and digital signal processing (DSP) blocks. The area cost is linearly proportional to the unroll factor (for LUTs and FFs), but the output data reduction factor is exponential: increasing the unroll factor by one halves the overall output volume. For unroll factor $U = 15$, the data reduce by 2^{15} and the variance takes about 5% of flip flops and 8% of other resources. For LUTs and FFs there is also a small fixed cost which is due to the Maxeler runtime system used to communicate with the host. The cost in block RAMs (BRAMs) is exponential in the unroll factor, as they are used to store delayed stream elements used for calculating the offset expressions; however, the cost is still modest even for large unroll factors.

The feedforward online operator performs less well than the feedforward pairwise operator, as shown in Figure 9. For unroll factor U, the output reduces by factor U.

Figure 10 shows that the feedback online design uses a small fixed area per assertion (about 3.5% of LUTs, 2% of FFs for 32-bit SP variance). For 32-bit SP data, the pipeline is 85 stages long, padded to 128 stages. The data are reduced to 128 partial variances, which can be further reduced to a single variance by Chan et al.'s method [2]. The design runs at 200 MHz.

Case Study. We assume a hard 0.5 s limit for avionics hardware runtime. Figure 11 shows estimated runtimes versus

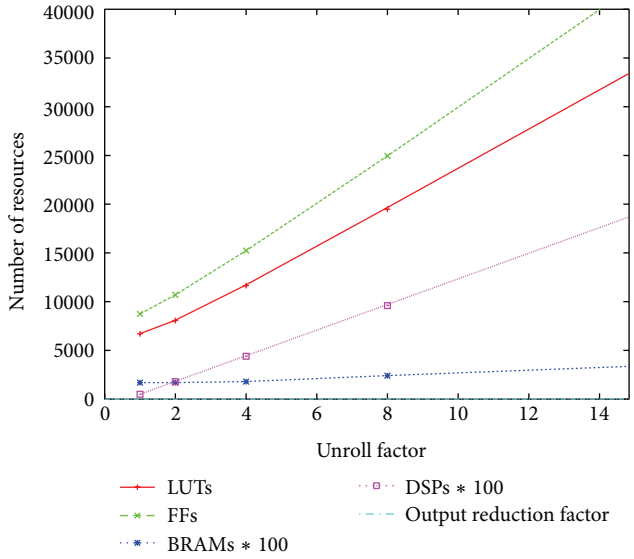


FIGURE 9: Area usage versus unroll factor for feedforward online variance operator.

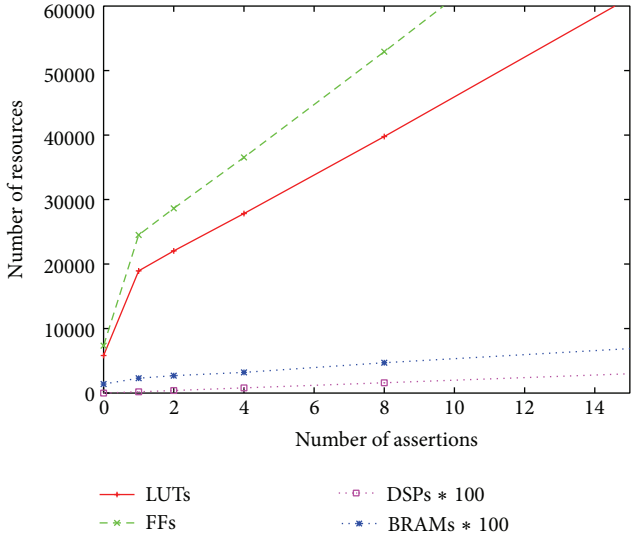


FIGURE 10: Area usage versus number of variance assertions for feedback online variance operator.

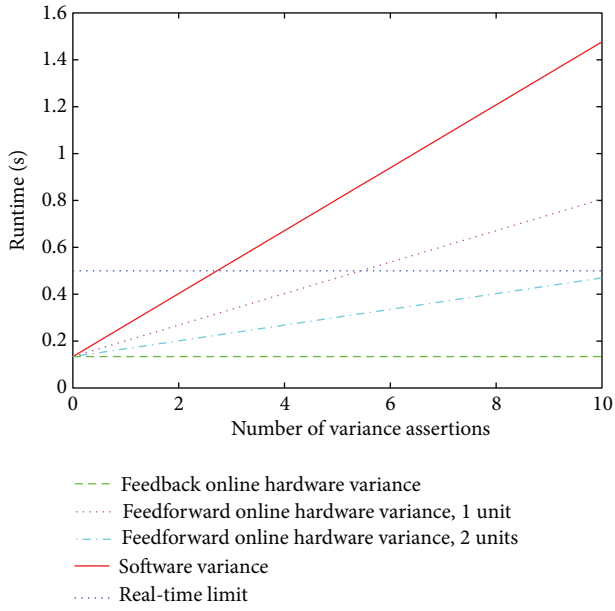


FIGURE 11: Avionics case study: estimated time taken by software and feedback hardware variance assertions versus number of assertions.

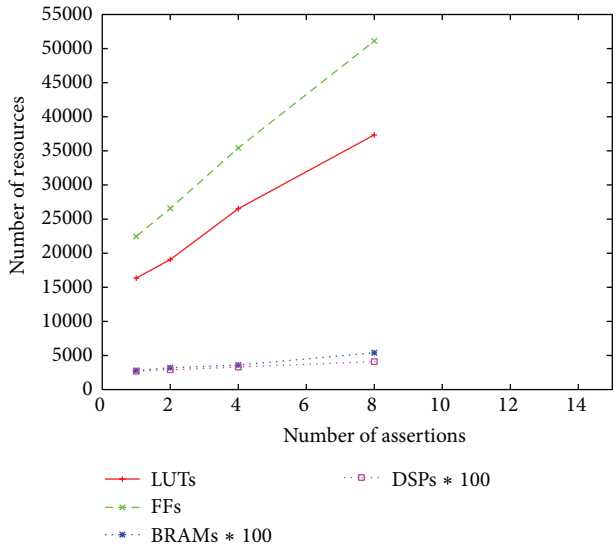


FIGURE 12: Avionics case study: resources used by the true airspeed datapath (Figure 7) versus number of feedback online variance operators.

the number of statistical assertions for both software and hardware implementations. We assume the design is limited by the bandwidth between software and hardware (MAX3 has 2 GB/s maximum speed); stream length is 2^{26}, and each output is 4 bytes wide, so the runtime with no assertions is 0.15 seconds. Software calculations are limited to two assertions within the time limit, because all 2^{26} values must be streamed across the bus for each exception. In contrast, the feedback design summarizes 2^{26} data to 128 values, and hence the time cost of each exception is much lower. The feedforward hardware designs allow the number of assertions to be traded for hardware area. Note we do not include time to calculate the variance on the host.

True Airspeed Calculation. We augment the true airspeed datapath (Figure 7) with several in-circuit, feedback online variance assertions. As expected, there is a modest linear cost per assertion (Figure 12). If we assume the same time constraints as in Figure 11, the number of feedback online assertions is limited by area, not by time.

6. Conclusion

To enable efficient monitoring for self-adaptive systems, we design and implement in-circuit statistical assertions, allowing designs to use several frequently occurring statistical operators to capture desired runtime properties of design inputs, outputs, and internal signals. Results show that response time can be greatly reduced at a modest cost in hardware area per exception.

Current and future work includes enlarging the set of statistical primitives to allow more general assertions on the state of the design. We would also like to explore the interaction of the statistical operators with runtime reconfiguration, since statistical conditions can be used to decide when to reconfigure. More generally, the statistics operators themselves can be reconfigured, allowing the system to alter the balance of configurable hardware between assertions and computation depending on runtime conditions. Furthermore, runtime statistics for large datapath operators can provide information about their adequacy, enabling continuous runtime optimization of system performance and energy efficiency.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

This work was supported in part by the UK EPSRC, by the European Union Seventh Framework Programme under Grant agreement nos. 257906 and 318521, by the HiPEAC NoE, by the Maxeler University Program, and by Xilinx.

References

[1] T. Todman, S. Stilkerich, and W. Luk, "Using statistical assertions to guide self-adaptive systems," in *Proceedings of the 2nd Workshop on Self-Awareness in Reconfigurable Computing Systems (SRCS'13)*, 2013.

[2] T. F. Chan, G. H. Golub, and R. J. LeVeque, "Updating formulae and a pairwise algorithm for computing sample variances," Tech. Rep. STAN-CS-79-773, Department of Computer Science, School of Humanities and Sciences, Stanford University, 1979.

[3] T. Reinbacher, M. Függer, and J. Brauer, "Real-time runtime verification on chip," in *Runtime Verification*, S. Qadeer and S. Tasiran, Eds., vol. 7687 of *Lecture Notes in Computer Science*, pp. 110–125, Springer, Berlin, Germany, 2013.

[4] R. Calinescu, C. Ghezzi, M. Kwiatkowska, and R. Mirandola, "Self-adaptive software needs quantitative verification at runtime," *Communications of the ACM*, vol. 55, no. 9, pp. 69–77, 2012.

[5] S. Vasudevan, "What is assertion-based verification?" *SIGDA E-News*, vol. 42, no. 12, 2012.

[6] J. Curreri, G. Stitt, and A. D. George, "High-level synthesis of in-circuit assertions for verification, debugging, and timing analysis," *International Journal of Reconfigurable Computing*, vol. 2011, Article ID 406857, 17 pages, 2011.

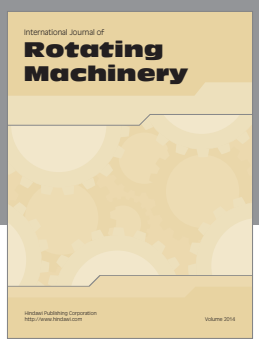
[7] M. N. Dinh, D. Abramson, J. Chao et al., "Debugging scientific applications with statistical assertions," in *Proceedings of the International Conference on Computational Science (ICCS '12)*, vol. 9, pp. 1940–1949, Procedia Computer Science, 2012.

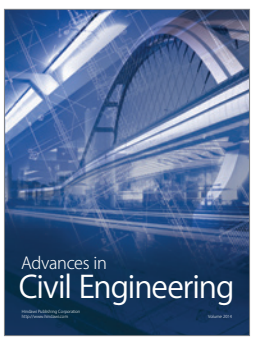
[8] D. E. Knuth, "The art of computer programming," in *Seminumerical Algorithms*, vol. 2, Addison-Wesley, 3rd edition, 1998.

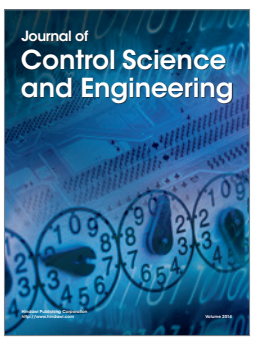
[9] B. P. Welford, "Note on a method for calculating corrected sums of squares and products," *Technometrics*, vol. 4, no. 3, pp. 419–420, 1962.

[10] R. P. G. Collinson, *Introduction to Avionics Systems*, Kluwer, South Holland, The Netherlands, 2nd edition, 2003.

[11] H. W. Liepmann and A. E. Puckett, *Introduction to Aerodynamics of a Compressible Fluid*, John Wiley & Sons, New York, NY, USA, 1947.







Hindawi
Submit your manuscripts at
http://www.hindawi.com