Information and Communication Technologies (ICT) Programme

Project N°: FP7-ICT-287804

D3.3: Hardware support for Efficient Dynamic Reconfiguration

Author(s): Georgi Gaydadjiev (CHT), Tobias Becker (IMP), Tim Todman (IMP)

Status -Version: Version 1.0

Date: 28 August, 2013

Distribution - Confidentiality: Confidential

Code: FASTER_D3_3_IMP_FF_20130831

Abstract:

This document constitutes deliverable D3.3 *Hardware support for efficient run-time verification*, the outcome of task T3.4, detailing the necessary support for efficient run-time verification. The document shows ways to support low overhead (in terms of performance and area) run-time verification, with adequate, light-weight hardware (architectural) support. In particular, the document includes developed by IMP for in-circuit assertions and exceptions as well as statistical in-circuit assertions, which allow low-overhead run-time verification.

© Copyright by the FASTER Consortium

Disclaimer

This document may contain material that is copyright of certain FASTER beneficiaries, and may not be reproduced or copied without permission. All FASTER consortium partners have agreed to the full publication of this document. The commercial use of any information contained in this document may require a license from the proprietor of that information.
The FASTER Consortium is the following:

Beneficiary Number	Beneficiary name	Beneficiary short name	Country
1(coordinator)	Foundation for Research and Technology – Hellas	FOR	Greece
2	Chalmers University of Technology	CHT	Sweden
3	Imperial College London	IMP	UK
4	Politecnico di Milano	PDM	Italy
5	Ghent University	GNT	Belgium
6	Maxeler	MAX	U.K.
7	ST	STM	Italy
8	Synelixis	SYN	Greece

The information in this document is provided "as is" and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

Document Revision History

Date	Issue	Author/Editor/ Contributor	Summary of main changes
June 18, 2013	0.1	Georgi Gaydadjiev	Initial table of contents
August 9, 2013	0.2	Tim Todman	Contributions from IMP
August 28, 2013	0.3	Georgi Gaydadjiev	First complete version
August 31, 2013	1.0	Tim Todman	Final version after QC

Table of contents

1. Introduction **4**
1.1. Task objectives 4
1.2. Document overview 4

2. Technical overview **5**
2.1. Background 5

3. In-circuit assertions and exceptions **7**
3.1. Abstract approach 7
3.2. Implementation for Maxeler designs 9
3.3. Evaluation 12

4. In-circuit statistical assertions **13**
4.1. In-circuit statistical assertions 13
4.2. Evaluation 15

5. Conclusion **17**

1. Introduction

This document constitutes deliverable D3.3 *Hardware support for efficient run-time verification*, the outcome of task T3.4, detailing the necessary support for efficient run-time verification.

1.1. Task objectives

This deliverable meets the second and third objectives of task 3.4: *exploring techniques for verifying selected static and dynamic aspects of a reconfigurable design at run-time*, and *adequate, light-weight hardware (architectural) support for run-time verification.*

Figure 1 shows how verification fits into the overall FASTER tool flow. We develop low-overhead, portable designs for in-circuit assertions (including statistical assertions) and exceptions which can be easily ported to multiple back-end flows. While our designs are currently implemented for Maxeler systems, they can be easily ported to other platforms and tool flows.

In particular, IMP has developed approaches for in-circuit assertions and exceptions, which meet both task objectives:

- Firstly, run-time verification of static and dynamic aspects of a design can be achieved by writing in-circuit assertions which, when true, indicate that the design is working correctly. We show that our designs allow users to carry out run-time verification without significant impact on speed, area or power consumption.
- Secondly, the in-circuit assertions also provide lightweight hardware support for run-time verification; we later show that the overhead of these assertions is low. Whilst our in-circuit assertions are implemented using reconfigurable fabric, future architectures could harden statistical operators, allowing even lower overhead circuit monitoring. We further show that assertions can be implemented in hardware or software, making the hardware/software boundary more flexible, which can benefit some application scenarios.

1.2. Document overview

The rest of this document is organized as follows: Section 2 gives a technical overview of the work, showing the approach used; Section 3 details our approach to in-circuit assertions and exceptions; Section 4 shows our approach to in-circuit statistical assertions. Finally, Section 5 concludes and shows planned future work.

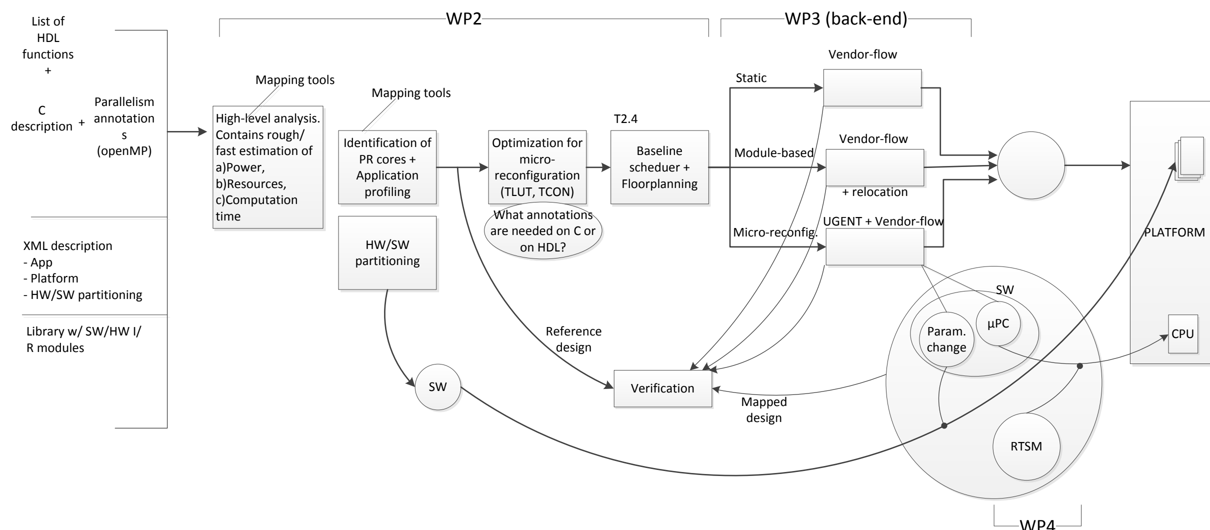

Figure 1: How verification fits into the FASTER tool flow.

2. Technical overview

Our approach for run-time verification uses in-circuit assertions and exceptions. The user takes the design specifications to derive design properties, which can be divided into static and dynamic sets. *Static* properties do not not depend on run-time data, and can be verified using the approach developed by IMP for verifying static and dynamic properties at compile-time, detailed in FASTER project deliverable D3.2. By contrast, *dynamic* properties do depend on run-time data, and can be verified using the approach detailed in this document.

Whilst our implementations use Maxeler design inputs, the approach is general and can extend to other design inputs, such as the more traditional Verilog and VHDL languages.

We define an *assertion* as any run-time Boolean expression which, when false, indicates an error of some kind, such as an input value out of range, or an intermediate result that will cause overflow. An *exception* is part of the control or data path that runs only when a corresponding assertion is false; if no assertions are false, no exception paths are active.

Assertions and exceptions separate error-handling code from normal operation, when no errors have been detected. Other language constructs could be used, but separating normal and error-handling code makes both easier to reason about. Assertions used in development may be removed for deployment; some criticize this as like “a sailing enthusiast who wears his lifejacket when training on dry land, but takes it off as soon as he goes to sea” [1].

2.1. Background

Software assertions and exceptions: assertions are part of the C standard and by default print a message on the console before aborting. C has no built-in support for exceptions, but can emulate them using calls to jump back to functions deeper in the stack. Some languages have extensive support for exceptions, notably Ada and Eiffel [2].

Hardware exceptions: the IEEE754 standard for floating-point arithmetic [3] includes exceptions, recommending that exceptions be resumeable, allowing user programs to fix problems. Exception handling in pipelined or out-of-order processors is difficult because exceptions from later instructions may occur before earlier instructions finish.

Hardware debugging: Debugging circuits can correct a design after deployment, whereas exceptions are included from the beginning. Hung and Wilton [4] monitor signals in FPGA (Field Programmable Gate Array) designs by reclaiming unused routing resources; conditions causing errors can be observed but not corrected in-place.

Assertion-based verification lets designers add assertions to their designs, written in Boolean and temporal logic [5]. Approaches include PSL [6] and SVA [7]. These approaches only apply to simulation, not real hardware, and only to hardware parts of designs, not corresponding host software. Assertion-based verification has been extended to in-circuit assertions by Curreri [8], who extend ANSI-C assertions to streaming FPGA designs. This approach may catch some bugs caused by mismatches between software and hardware. However, there is no exception mechanism; user programs cannot recover from exceptions in hardware, only report errors back to software.

Statistical assertions have been proposed by Dinh et al. [9], as a debug-time method to reason about large parallel programs – users can reason with derived metrics, rather than raw program output. The assertions are implemented efficiently using a map-reduce style computation. We use statistical assertions for run-time monitoring of reconfigurable hardware-accelerated systems.

Hardware redundancy: fault tolerance can be achieved using extra hardware, for example in the work of Siozios et al. [10], where redundancy is selectively applied to areas of the hardware most likely to experience faults. These approaches are orthogonal to our work.

3. In-circuit assertions and exceptions

This section shows techniques developed by IMP for run-time verification by in-circuit assertions and exceptions.

3.1. Abstract approach

We now describe our abstract approach to runtime assertions and exceptions for streaming hardware designs. The approach does not depend on any particular tool, but could adapt to several available streaming hardware design tools.

We choose streaming hardware designs because they are increasingly used to implement reconfigurable hardware designs, particularly for high-performance applications. Much of our approach could also apply to other hardware design languages such as VHDL and Verilog.

Figure 2 shows our verification flow. The flow starts with a design to verify and the properties to be verified. First, the user divides the properties into static or compile-time properties, and dynamic or run-time properties, dependent on run-time data. Second, the user separates the properties into assertions and exceptions; assertions encoding design properties, exceptions labelling error conditions. Static properties can be handled by existing static verification approaches. Third, the user writes run-time assertions to encode design assumptions which can only be checked at run-time, for example input variable ranges. For some exceptions, the user writes handlers to catch the exception and substitute a replacement value for the expression causing the exception: for example, an overflow exception might result in the value being clamped at the maximum value for that variable, resulting in a saturating arithmetic. Other exceptions may have no sensible replacement value and are propagated to software, where they can be used for debugging. Finally, the user runs the design including assertions and exceptions. If no assertions are raised, and any exceptions are handled, the design is verified as correct for the input and assertions used. Currently, the user manually separates the static and dynamic properties and adds assertions and exceptions, but future work could automate this, perhaps requiring the user to write the properties using a specification language, then detecting which are dynamic, and compiling from specifications into assertion conditions.

When multiple designs implement the same specifications, for example straightforward and optimized implementations, the same assertions and exceptions can be reused for both, to check requirements are met. This way design efforts are reduced. Other assertions can be added to each design to check design-specific properties.

Hardware exceptions differ from assertions in that they can be handled, meaning that a value is substituted for the expression which raised the exception. This allows designs to handle errors in place rather than relying on host software to fix the problem, potentially reducing bus traffic between software and hardware hosts (the control and the accelerator sub-systems). Users can explore a tradeoff: handling more errors in hardware, costing more resources versus handling more errors in software, at a cost of more bandwidth require between hardware and software hosts.

The grammar of our abstract stream language follows:

```
d = ...                                                                  1
```

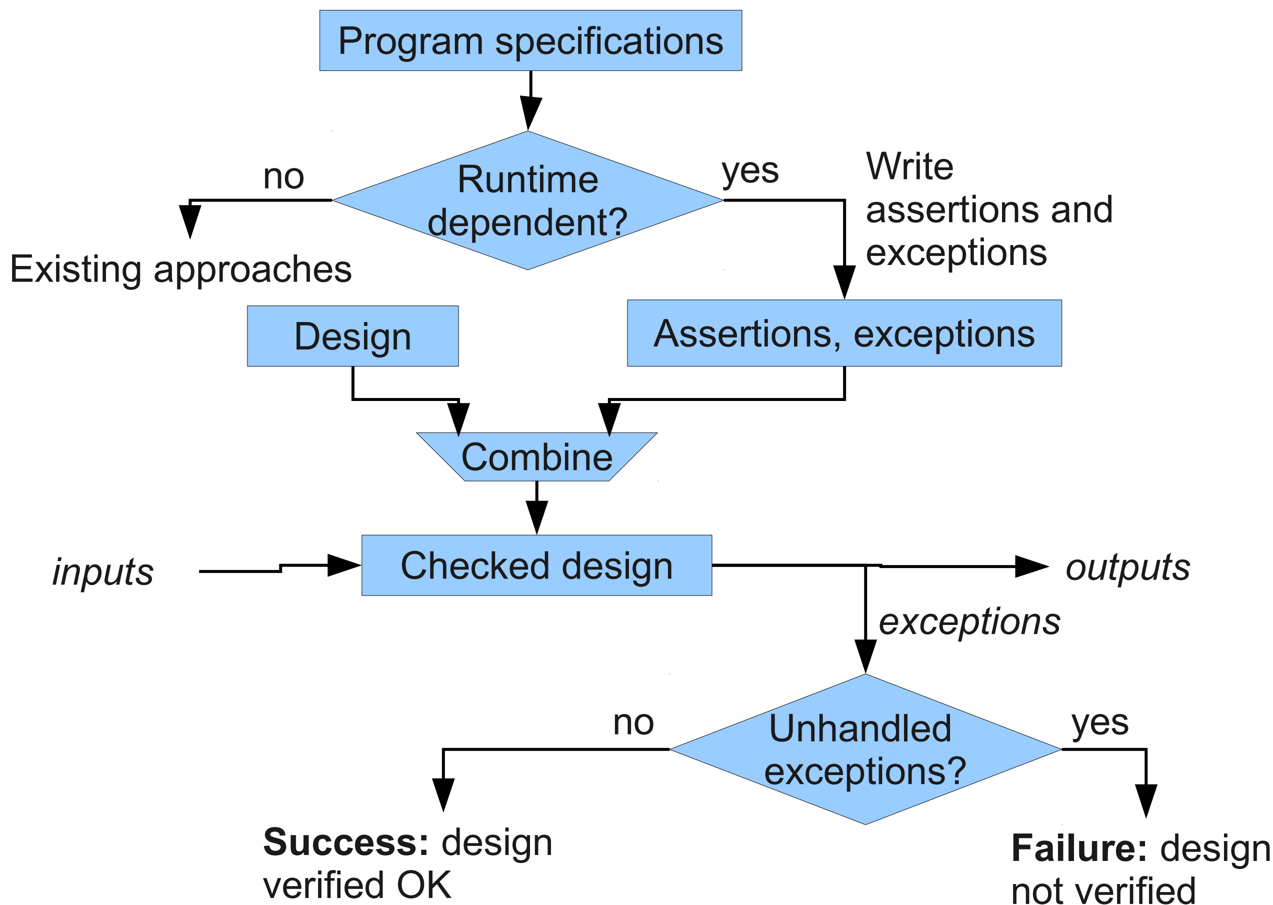

Figure 2: Verification flow of our abstract approach.

```
  | "exception" ID ";"                                   2
s = lval "=" expr                                        3
  | "if" "(" e ")" s "else" s                            4
  | "while" "(" e ")" s                                  5
  | "assert" "(" e ")"                                   6
e = e bop e                                              7
  | INT                                                  8
  | FLOAT                                                9
  | ID                                                   10
  | "(" e ")"                                            11
  | uop e                                                12
  | "raise" ID                                           13
  | "try" e "with" ( ID "->" e )*                        14
bop = "+" | "-" | "*" | "/" | ...                        15
uop = "+" | "-" | "~" | ...                              16
```

where d, s and e are declarations, statements and expressions respectively. Extensions for assertions and exceptions comprise: 1. a declaration to define possible exceptions in this program; only declared exceptions can be used; 2. a statement to assert a condition: if false, an exception is raised; 3. an expression to raise an exception; 4. an expression to allow raised exceptions to be handled. Given an expression e, its result is e if no exceptions are raised in e,

otherwise the optional list of exception handlers is consulted. If a handler matches the raised expression, the corresponding value is the result of the expression, otherwise the exception propagates to the surrounding program.

The assert statement is directly taken from C99; many designers will already be familiar with this. Since C has no support for exceptions, we base our design on OCaml, which allows exceptions to be declared, raised and handled within both expressions and statements.

An informal semantics of our assertions and exceptions is:

1. a failed assertion is recorded in a buffer showing which assertion failed, on which cycle;
2. raising an undeclared exception is a compile-time error;
3. raising an exception propagates it out to the enclosing expression;
4. an exception raised within a try expression is matched against the list of handlers; if a handler matches, the corresponding expression results, otherwise the exception propagates to the surrounding expression;
5. if an exception propagates to a statement, it is unhandled and recorded like a failed assertion.

3.2. Implementation for Maxeler designs

We implement our abstract approach for Maxeler streaming systems. In the Maxeler system, users describe hardware designs as Java programs, using a Java class library and language extensions. Prior to execution, the compiler builds a dataflow graph of the program, translates the graph into an HDL (Hardware Description Language) implementation, and calls FPGA vendor tools to compile the HDL into a bitstream. The design consists of a data path reading from one or more stream inputs, one element per stream per cycle, and producing one or more stream outputs (one element per cycle). State machines or counters control the design.

We systematically translate designs using Maxeler kernels extended with the proposed assertions and exceptions into regular Maxeler designs. Currently our translation is manual, but future work could automate it.

Extensions for runtime assertions and exceptions: we extend the Maxeler kernel description language, based on Java, with the our abstract language features for runtime assertions and exceptions. We extend the grammar as follows:

```
d = ...                                              1
  | "__exception" ID ";"                             2
s = ...                                              3
  | "__try" s ( "__catch" "(" ID ")" s )*            4
  | "__assert" "(" e ")"                             5
  | "__raise" e ";"                                  6
e = ...                                              7
  | "__try" e ( "__when" ID "->" e )*                8
  | "__raise" e                                      9
```

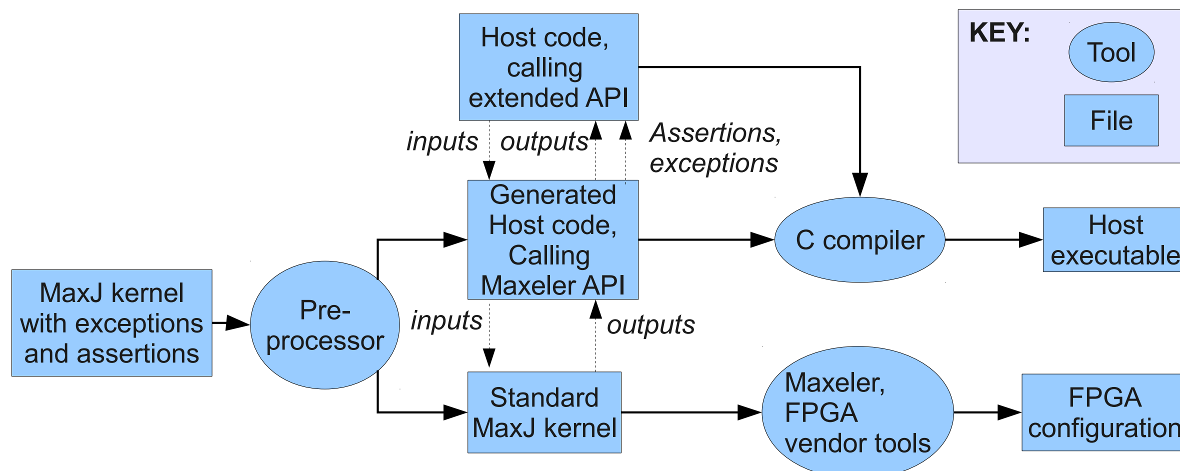

Figure 3: Design flow targeting Maxeler designs.

where existing grammar for declarations (`d`), statements (`s`), and expressions (`e`) is represented by ellipses (...). We allow exceptions to be raised and handled in both statements and expressions; this gives designers additional freedom about where to put error-handling code: one `__try ... __catch` block can handle any exceptions raised in the entire block.

Figure 3 shows the design flow for Maxeler systems. The user writes their design as a software program using our extended version of Maxeler's API (Application Programming Interface) for controlling a hardware design written in our extended version of Maxeler's MaxJ kernel description language. Our API extensions allow (a) assertions in hardware designs to be reflected into software designs; (b) exceptions to be declared, raised and handled in hardware designs. Unhandled exceptions similarly reflect into software.

Figure 4 shows how exceptions are supported by wrapping Maxeler hardware and software APIs. Each exception which can escape from the hardware becomes another streaming output, which must be passed using standard Maxeler APIs. In software, our tool adds a loop which performs a C software assertion for each exception output added.

Case study: the following shows a basic C implementation of a 32-bit integer moving average filter, which we use as a basis for our experiments. The design is parametrised for stream length N and filter radius W; we use arbitrary stream lengths and radius $W = 64$. This code reads from input array `inp` and writes to output array `outp`.

```
const size_t N=16*1024*1024;                1
int inp[N], outp[N];                        2
for (i=0;i<N;++i) {                         3
  sum=0;                                    4
  for (j=0;j<W;++j) {                       5
    sum += inp[i-W/2+j];                    6
  }                                         7
  outp[i] = sum/W;                          8
}                                           9
```

For space reasons we omit the code that prevents reading outside the input array. A Maxeler implementation is:

```
__exception OutOfRange; //declare exception    1
```

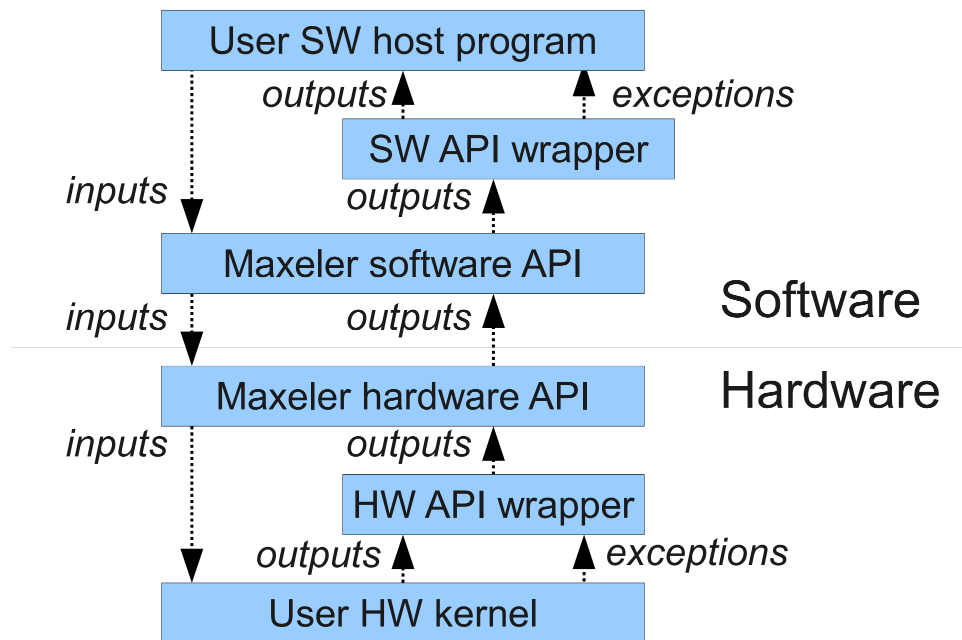

Figure 4: Wrapping Maxeler hardware and software APIs.

```
HWVar inp = io.input("inp", hwInt(W));                                    2
__try {                                                                    3
  HWVar sum = constant.var(0);                                             4
  for (j=0;j<W;++j) {                                                      5
    sum += stream.offset(inp,-(W/2)+j);                                    6
    if (sum<0) __throw OutOfRange;                                         7
  }                                                                        8
__catch (OutOfRange) {                                                     9
  sum = MAX;                                                               10
  }                                                                        11
}                                                                          12
io.output("outp",sum/W,hwInt(W));                                          13
```

where: line 1 declares an exception; line 2 declares a stream input `inp` of 32-bit, unsigned integer type; lines 3 to 8 comprise a runtime exception-handling block: an OutOfRange exception raised in this block is handled by the corresponding catch block; line 4 declares a variable `sum` to store intermediate results; lines 5 to 8 implement the filter; this loop runs at compile-time (a fully-unrolled implementation); line 8 raises the OutOfRange exception if sum is negative (indicating overflow); lines 9 to 11 handle the exception from lines 4 to 8: if caught, sum is set to `MAX`; finally, line 12 declares output stream `outp`.

We augment Maxeler API calls interacting with the hardware to read back assertions and exception outputs, and generate one C assertion for each failed hardware assertion or unhandled exception. While C does not support exceptions, our approach could adapt to languages which do, so unhandled hardware exceptions lead to software exceptions.

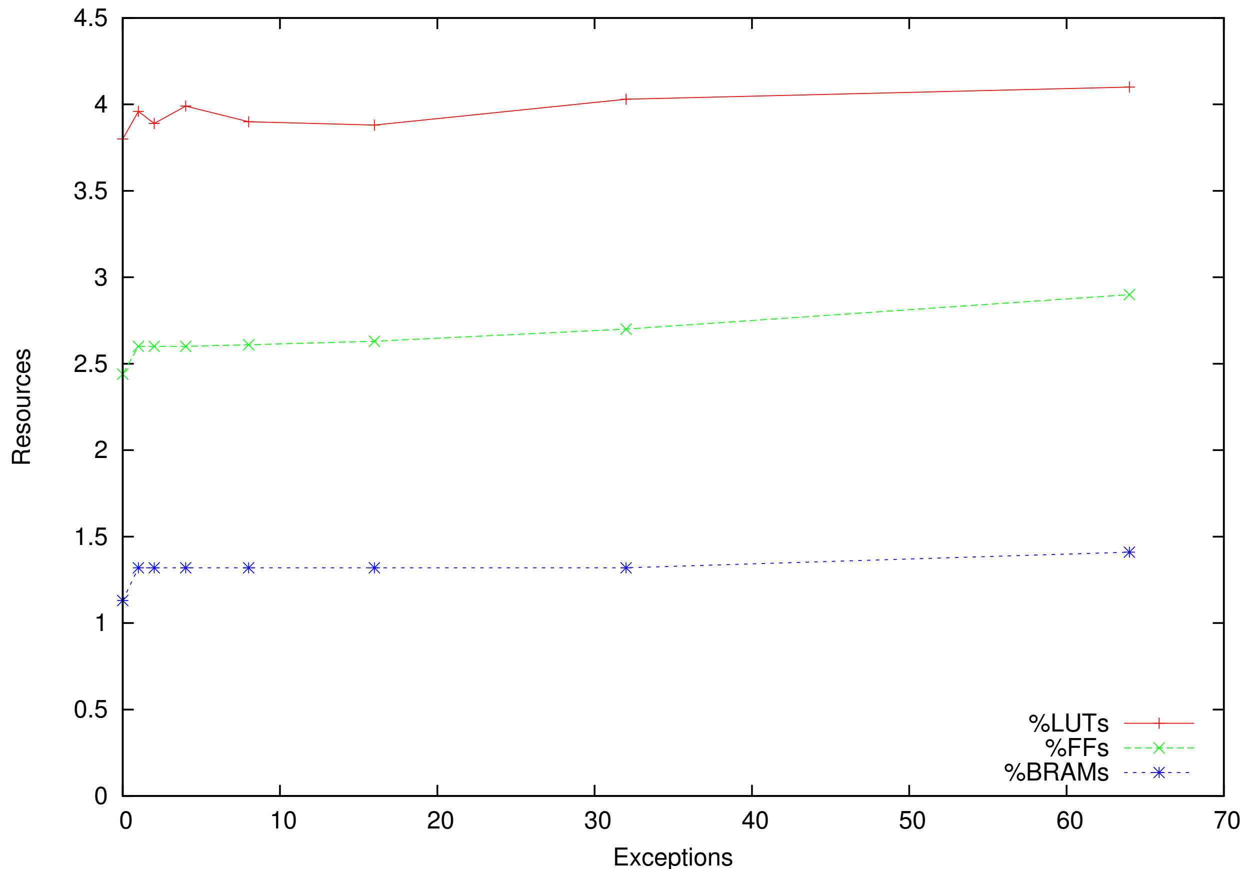

Figure 5: Area results: % area versus no. exceptions for a 64-wide, 32-bit moving average filter.

3.3. Evaluation

We evaluate using a moving average filter as a case study; though simple, similar tradeoffs in terms of area versus speed, and number of exceptions and assertions are needed in larger designs. Experiments measure the cost (reconfigurable hardware resources) to add assertions.

Experimental setup: hardware is compiled using Maxeler MaxCompiler version 2012.1 and Xilinx ISE 13.1, targeting the Maxeler MAX3 board (Xilinx Virtex-6 xc6vsx475t device). Each design targets a clock rate of 300MHz.

Area results: to measure assertion costs, we add an assertion to the loop that variable `sum` is always positive (a negative number indicates overflow). We add A assertions, where $1 < \mathtt{A} < W$ by inserting the line: `if (j<N) assert(sum>0);` after the accumulation in the loop body.

Figure 5 shows area resources used (LUTs and Flip Flops) versus number of exceptions for the moving average application The cost of adding assertions lies between 4.5% (LUTs) and 1.5% (BRAMs) due to the logic used to implement assertion conditions, and buffers used to store assertion results. Beyond that, there is a linear area cost per assertion added; since each exception is a Boolean stream output, adding an exception has a small area penalty. Designers may thus add many exceptions without much concern over area costs.

4. In-circuit statistical assertions

This section shows techniques developed by IMP for run-time verification by in-circuit statistical assertions. This expands the work of the previous section by allowing assertion conditions to use the statistics of in-circuit signals, rather than simple Boolean conditions.

4.1. In-circuit statistical assertions

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:

- $mean(e1)$, the mean value of expression $e1$;
- $stdev(e1)$, the standard deviation of expression $e1$;
- $variance(e1)$, the variance of expression $e1$.

We choose these 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 [11]:

```
e = a                                   1
  | e bop e                             2
  | uop e                               3
  | "mean" "(" e ")"                    4
  | "stdev" "(" e ")"                   5
  | "variance" "(" e ")"                6
bop = "==" | "!=" | "<" | ">" | ...     7
uop = "+" | "-" | "!" | "~"             8
```

where *bop* represents any C binary operator, *uop* any C unary operator and a any atomic expression (literals, variables, constants). This language allows the user to combine both Boolean and statistical operators.

Online algorithms for calculation of statistical metrics such as mean, variance and standard deviation are known [12] [13], which involve a single pass over the input data, using an accumulator and the current input element. Whilst this may seem suitable for streaming implementations, they contain feedback owing to the accumulator. Chan et al. developed a pairwise algorithm for variance [14] which can be parallelized; for N input elements, naively implementing this algorithm on streaming systems requires $NlogN$ hardware. Chan et al's algorithm denotes the sum and mean of data points x_i as T_{ij} and M_{ij} respectively:

$$T_{ij} = \sum_{k=i}^{j} x_k \qquad M_{ij} = \frac{1}{j-i+1} T_{ij}$$

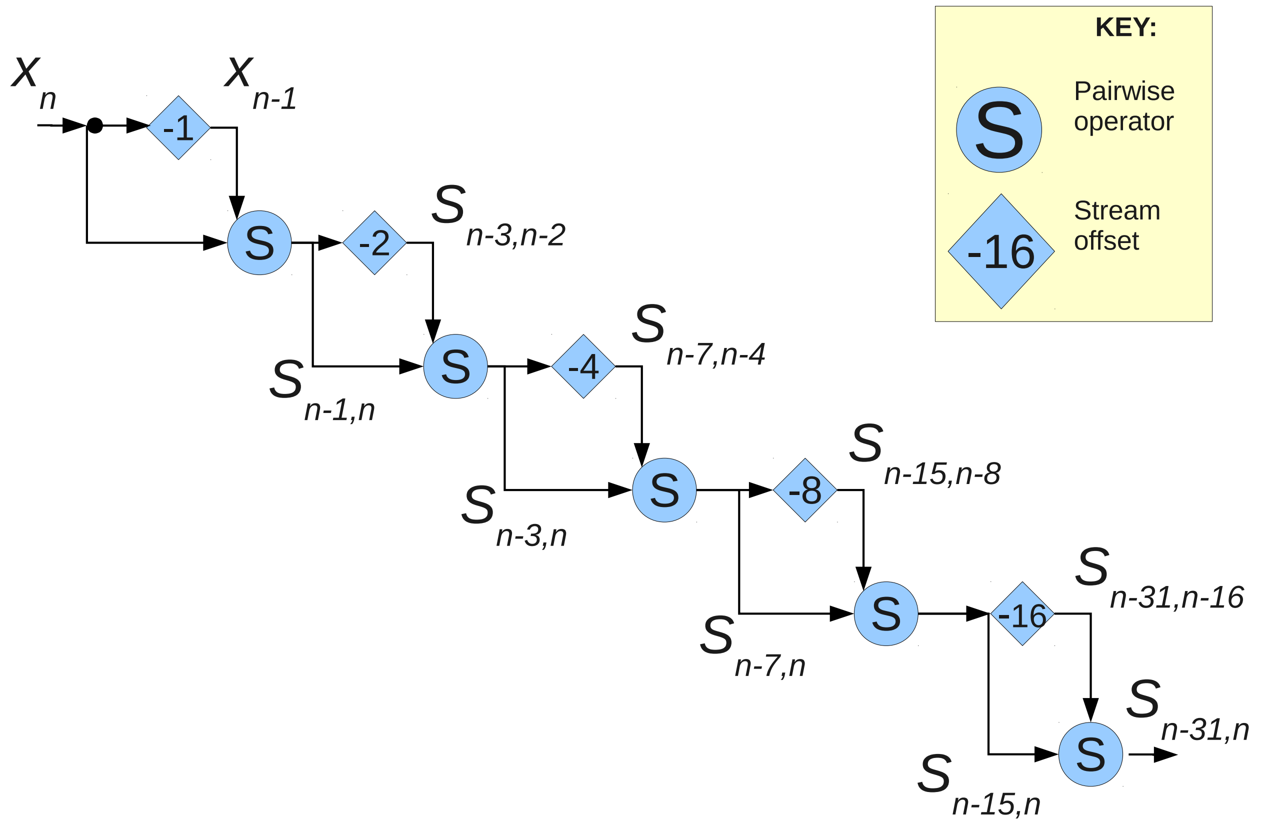

Figure 6: Systolic partial calculation of variance of input x_n using pairwise operators.

and the sum of squares S_{ij}:

$$S_{ij} = \sum_{k=i}^{j} (x_k - M_{ij})^2$$

calculated by their pairwise algorithm (here $i = 1$ and $j = 2m$):

$$S_{1,2m} = S_{1,m} + S_{m+1,2m} + \frac{1}{2m}(T_{1,m} - T_{m+1,2m})^2$$

We propose two designs suitable for streaming systems: a systolic design adapted from Chan's parallel algorithm, and a C-slowed variant of the online algorithm. Figure 2 shows the datapath for the systolic 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. Note that the leftmost operator can be optimized, because $S_{i,i} = 0$ (the variance of a single point is zero). The systolic design uses the observation that in a streaming system, iterating through the input data in order, sums of neighbouring elements can be accessed by stream offsets, so using Chan et al's notation:

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

and $S_{1,2^k}$ is calculated in the same way.

Note that this 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 both the output data and the remaining calculations to be done in software by half, so K units reduces it 2^K-fold.

Implementation targeting Maxeler streaming 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 could implement a compiler from Maxeler designs extended with statistical assertions into the 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 programatically builds a datapath using a domain-specific language based on Java. The control path may be 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 a reconfigurable device bitstream implementing a design. 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 feed-forward designs, but feedback requires some manual intervention and reordering or replicating of input data.

4.2. Evaluation

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

1. scalability: operator size versus hardware size;
2. software statistics versus hardware-assisted: 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 200MHz. We implement a single variance assertion, with 32-bit input data in IEEE Single-Precision (SP) floating-point format, one data element per cycle.

Figure 7 shows the effect of unroll factor on the area resources for the systolic variance operator; the area is measured in Look-Up Tables (LUTs), Flip-Flops (FFs), Digital Signal Processing (DSP) and Block RAM (BRAM) blocks. The area cost is linearly proportional to the unroll factor (for LUTs, FFs and DSPs), but the output data reduction factor is exponential: increasing the unroll factor by one halves the output volume. For unroll factor 15, the data reduces by 2^{15} and the variance takes about 5% of flip-flops, 8% of other resources. For BRAMs, the cost is proportional to the number of elements summarized in the variance. There

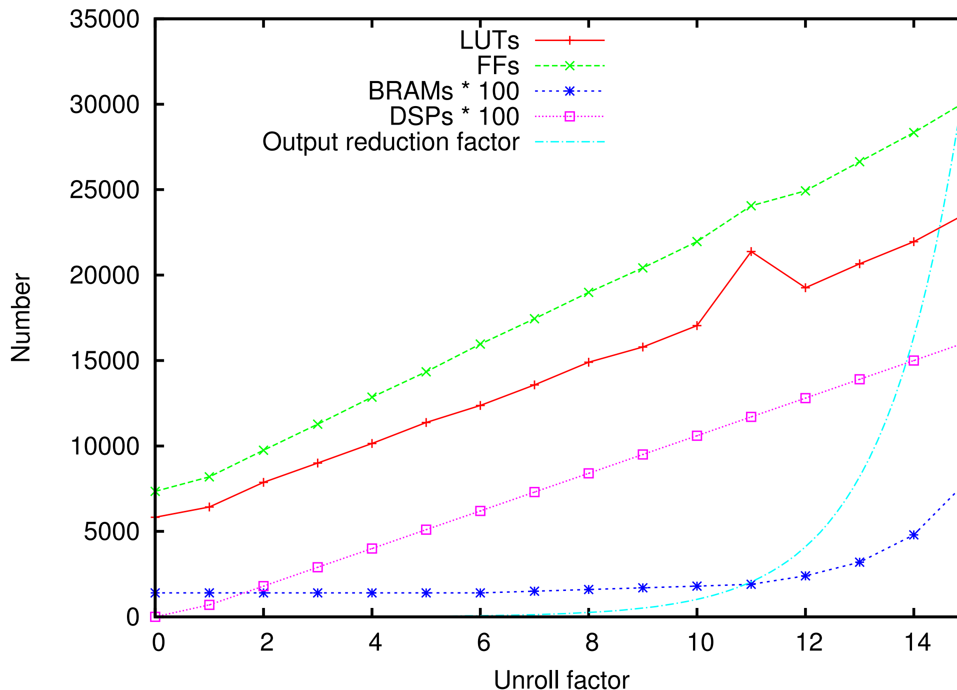

Figure 7: Area usage and output reduction versus unroll factor for systolic variance operator.

is a small anomaly in circuit area for unroll factor 11 which we believe is due to pseudorandom variance in the Xilinx place and route tools.

The C-slowed 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. Note that the C-slowed design is smaller than the unrolled systolic designs, and can run at up to 300MHz.

Case study: time limit: we assume a hard 0.5s limit for hardware run time. Figure 8 shows estimated run times versus 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 2GB/s maximum speed for the host-FPGA connection across the PCIe bus); stream length is 2^{26}, each output is 4 bytes wide, so without any assertions, the runtime is given by the time to transfer 2^{26} elements across the bus: $2^{26} \times 4/(2 \times 10^9) = 0.134$ 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 C-slowed design summarizes 2^{26} data to 128 values, meaning the time cost of each exception is much lower. Systolic hardware designs allow the number of assertions to be traded for hardware area. Note we do not include time to complete the variance calculation on the host, but this is very small (only 128 inputs instead of 2^{26}).

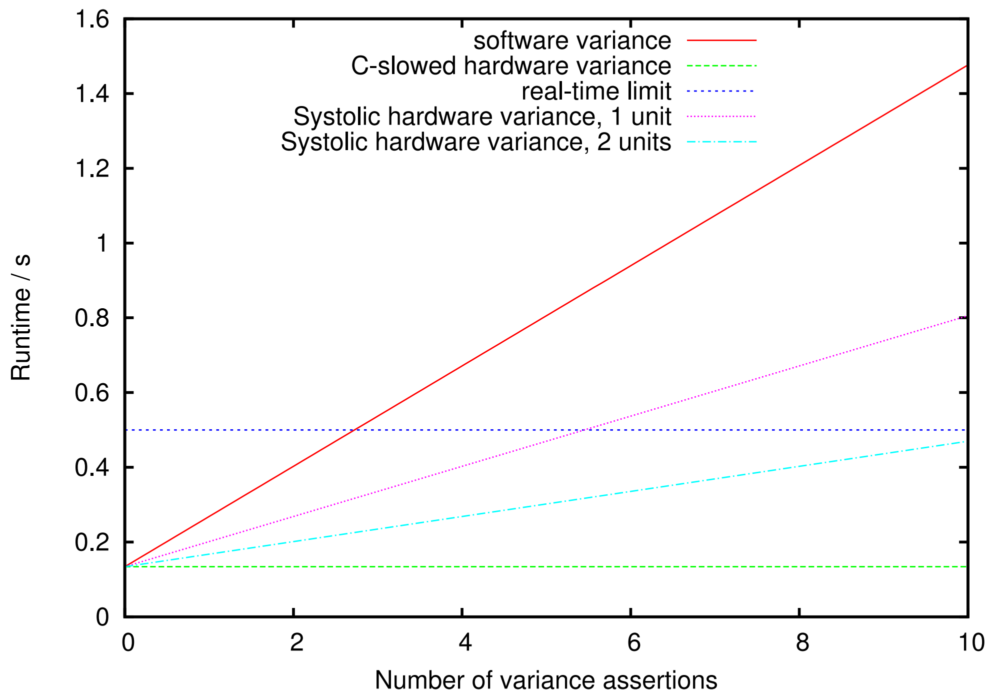

Figure 8: Estimated time taken by software and C-slowed hardware variance assertions versus number of assertions.

5. Conclusion

We present an abstract approach for adding in-circuit assertions and exceptions to hardware designs, and a concrete implementation for Maxeler systems. Results show that our assertions and exceptions add little area and execution latency costs.

To allow efficient monitoring for self-adaptive systems, we design and implement in-circuit statistical assertions, allowing designs to use several frequently-occurring statistical operators to express 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, firstly, integrating our approach with temporal logic, allowing a more formal basis for the error handling. Secondly, we would like to add support for run-time reconfiguration. Designs could reconfigure to add exception handlers if many errors are detected, or running circuits could dynamically change exception handlers and assertions, without changing the rest of the design.

Thirdly, we would like to enlarge 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 run-time reconfiguration. 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.

References

[1] C. A. R. Hoare, "Hints on programming language design.," Tech. Rep. STAN-CS-73-403, Stanford, CA, USA, 1973.

[2] M. Scott, *Programming Language Pragmatics.* Morgan Kaufman, 3 ed., 2009.

[3] "IEEE standard for floating-point arithmetic," *IEEE Std 754-2008*, pp. 1–58, 2008.

[4] E. Hung and S. J. E. Wilton, "Towards simulator-like observability for fpgas: a virtual overlay network for trace-buffers," in *FPGA '13*, 2013.

[5] S. Vasudevan, "What is assertion-based verification?," *SIGDA E-News*, vol. 42, December 2012.

[6] "IEEE standard for property specification language (PSL)," *IEEE Std 1850-2010 (Revision of IEEE Std 1850-2005)*, pp. 1–182, 2010.

[7] D. Bustan, D. Korchemny, E. Seligman, and J. Yang, "Systemverilog assertions: Past, present, and future sva standardization experience," *Design Test of Computers, IEEE*, vol. 29, no. 2, pp. 23–31, 2012.

[8] 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, 2011.

[9] M. N. Dinh, D. Abramson, J. Chao, D. Kurniawan, A. Gontarek, B. Moench, and L. DeRose, "Debugging scientific applications with statistical assertions," *Procedia Computer Science*, vol. 9, no. 0, pp. 1940 – 1949, 2012. Proceedings of the International Conference on Computational Science, (ICCS) 2012.

[10] K. Siozios, D. Soudris, and D. Pnevmatikatos, "A framework for enabling fault tolerance in reconfigurable architectures," in *Reconfigurable Computing: Architectures, Tools and Applications* (P. Sirisuk, F. Morgan, T. El-Ghazawi, and H. Amano, eds.), vol. 5992 of *Lecture Notes in Computer Science*, pp. 257–268, Springer Berlin Heidelberg, 2010.

[11] N. Wirth, "What can we do about the unnecessary diversity of notation for syntactic definitions?," *Commun. ACM*, vol. 20, pp. 822–823, Nov. 1977.

[12] D. E. Knuth, *The Art of Computer Programming, volume 2: Seminumerical Algorithms.* Addison-Wesley, 3rd ed., 1998.

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

[14] 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, November 1979.