RUNTIME ASSERTIONS AND EXCEPTIONS FOR STREAMING SYSTEMS

Tim Todman, Wayne Luk

Department of Computing
Imperial College London
180 Queen's Gate, London SW7 2AZ
email: {timothy.todman, w.luk}@imperial.ac.uk

ABSTRACT

We present an approach to enable run-time, in-circuit assertions and exceptions in reconfigurable hardware designs. Static, compile-time checking, including formal verification, can catch many errors before a reconfigurable design is implemented. However, many other errors cannot be caught by static approaches, including those due to run-time data. Our approach allows users to add run-time assertions and exceptions to a design, giving multiple ways to handle run-time errors. Our work includes an abstract approach to adding assertions and exceptions to a design, a concrete implementation for Maxeler streaming designs, and an evaluation. Results show low overhead for adding exceptions to a design.

1. INTRODUCTION

As the size of reconfigurable hardware devices increases, they are used to implement increasingly large and complex designs. This leads to a challenge: verification, ensuring that designs implement their intended behaviour. There are many approaches to static, compile-time checking of designs, including formal verification, but static approaches cannot in general hope to catch all errors that can occur at run-time, particularly those caused by run-time input data.

Traditionally, simulation is used to catch run-time errors, but designs are now so large that simulation cannot hope to catch them all. Assertion-based verification is increasingly popular; examples include Property Specification Language (PSL) [1] and System Verilog Assertions (SVA) [2]. In-circuit assertions [3] detect errors in hardware and report them to software. We extend to in-circuit exceptions, handling errors in the circuit where they are detected.

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" [4].

This paper makes the following contributions:

- An abstract approach to enabling runtime assertions and exceptions in hardware designs, with a language of assertion conditions and user-customizable policies for actions when assertions are violated;
- An implementation of our abstract approach for Maxeler streaming hardware designs, showing how the abstract approach maps into streaming hardware;
- An evaluation of our approach on a case study.

The rest of the paper is organized as follows: the next section outlines related work. Section 3 describes our abstract approach to runtime assertions for reconfigurable hardware designs; section 4 details the implementation for Maxeler designs; section 5 evaluates the approach, while section 6 concludes and outlines future work.

2. 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 [5].

Hardware exceptions: the IEEE754 standard for floating-point arithmetic [6] 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

978-1-4799-0004-6/13/$31.00

from the beginning. Hung and Wilton [7] 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 [8]. Approaches include PSL [1] and SVA [2]. 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 [3], 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.

3. 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 1 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. 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.

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, saving design effort. Other assertions check design-specific properties.

Hardware exceptions differ from assertions in that they

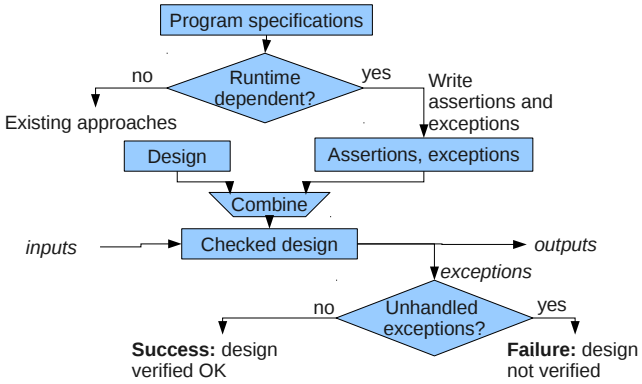

Fig. 1. Verification flow of our abstract approach.

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. 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 host.

The grammar of our abstract stream language follows:

```
d = ...                                                    1
  | '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 declare 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.

4. 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. When run, the programs build a dataflow graph of the, compile the graph into an HDL (Hardware Description Language) implementation, and call 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 per cycle, and producing one or more stream outputs, one per cycle. State machines or counters control the design.

We systematically translate designs using Maxeler kernels extended with 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
```

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 more choice about where to put error-handling code: one `__try ... __catch` block can handle any exceptions raised in the entire block.

Figure 2 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

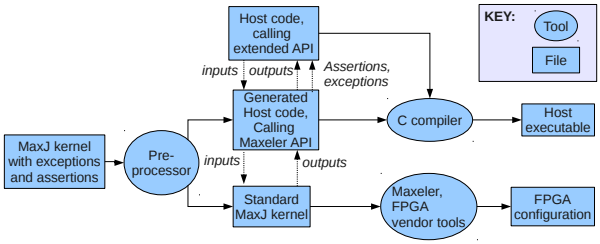

Fig. 2. Design flow targeting Maxeler designs.

to be declared, raised and handled in hardware designs. Unhandled exceptions similarly reflect into software.

Figure 3 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 parameterised 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 code to stop reading outside the input array. A Maxeler implementation is:

```
__exception OutOfRange; //declare exception               1
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
io.output("outp",sum/W,hwInt(W));                          12
```

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

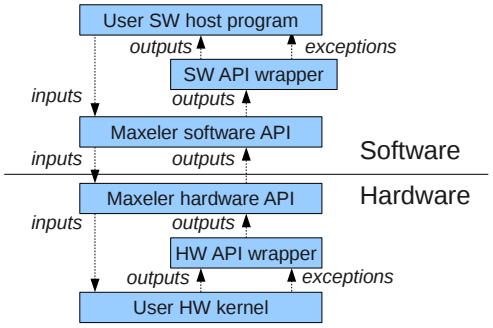

Fig. 3. Wrapping Maxeler hardware and software APIs.

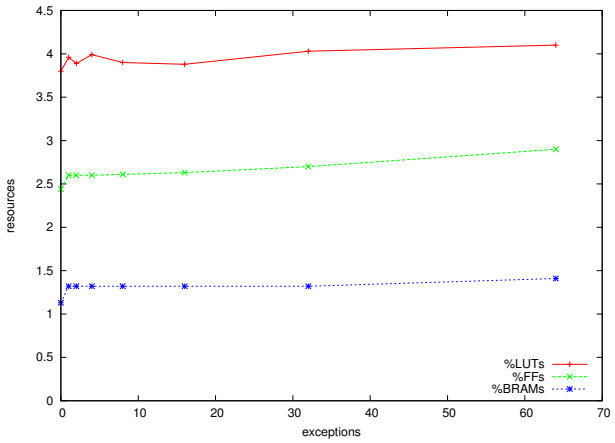

Fig. 4. Area results: % area versus no. exceptions for a 64-wide, 32-bit moving average filter.

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.

5. 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 $1 < \texttt{A} < W$ by inserting the line: `if (j<N) assert(sum>0);` after the accumulation in the loop body.

Figure 4 shows area resources used (LUTs and Flip Flops) versus number of exceptions for the moving average application The cost of adding assertions lies between 5% (LUTs) and 15% (BRAMs), due to 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.

6. 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 speed cost.

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.

Acknowledgements: Thanks to the reviewers for their comments. This work is supported in part by UK EPSRC, the European Union Seventh Framework Programme under Grant agreement numbers 257906, 287804 and 318521, the HiPEAC NoE, and Xilinx.

7. REFERENCES

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

[2] 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.

[3] 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.

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

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

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

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

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