Verification of streaming hardware and software codesigns

Tim Todman, Peter Boehm, Wayne Luk
Department of Computing, Imperial College London, UK
timothy.todman@imperial.ac.uk

***Abstract*—We present an approach to verifying the codesign of software and hardware. Our approach verifies that a reference design, perhaps a straightforward software implementation, is equivalent to a design combining software and reconfigurable hardware, possibly using runtime reconfiguration. Our approach combines symbolic simulation with equivalence checking to compare symbolic output expressions. Whilst our implementation uses C-style software and streaming hardware based on Maxeler designs, our approach is modular and could generalize to other software or hardware inputs. We evaluate our approach by applying it to several kernels, including one used for geoengineering.**

I. Introduction

To meet demanding application requirements, designers increasingly choose complex systems combining hardware and software, the idea being to combine the performance and power-efficiency of dedicated hardware with the flexibility of software. Software parts may be easier to develop and rewrite, while dedicated hardware is usually faster and often more power-efficient. This combination may occur at board level, or within a single device such as a system-on-chip (SoC), which may combine custom and standard hardware and software.

Additionally, designers may use reconfigurable hardware, which blends hardware efficiency with software flexibility, but adds to the design complexity, particularly if using run-time reconfiguration (RTR), which allows large designs to fit onto smaller reconfigurable hardware, or reduces power requirements by replacing complex variable-input elements, such as multipliers, with more efficient constant-input elements.

While complex systems with multiple software and hardware elements can meet application demands, their very complexity makes them hard to design and debug. Traditionally, such systems are tested by extensive simulation with test inputs; however, it is well-known that the complexity of today's systems makes exhaustive simulation infeasible.

Therefore, formal methods and verification are often used to complement simulation and testing, especially in safety critical application areas. The formal verification of hardware and software are both well-studied fields with long histories in academic research going back at least to Hoare [1], Floyd [2], and Dijkstra [3]. Traditionally hardware and software verification research are separate; different challenges result from the complexity of Boolean algebra (for hardware) and the undecidability and non-deterministic concurrency of software.

978-1-4673-2845-6/12/$31.00 © 2012 IEEE

However, for embedded systems or SoCs, for example, verifying hardware and software independently may not be helpful: hardware without low-level software may simply be non-functional, while low-level software may depend on underlying hardware so crucially that a hardware abstraction is insufficient to specify the software's intended behaviour. Thus, these systems usually have to be considered as hardware/software codesigns and methodologies for their formal verification is still an open research challenge.

Recent work [4] verifies FPGA-based streaming hardware by combining symbolic simulation with equivalence checking. However, when considering a software and runtime reconfiguration, new verification challenges arise: (i) modelling reconfiguration is a non-trivial and may significant increase the complexity of the formal model as many possible configurations may have to be considered; (ii) the interface between hardware and software may be subject to change because of reconfiguration, amplifying the problems.

We propose a methodology to verify the equivalence of combined software and runtime reconfigurable hardware with golden reference software. Our approach could extend to compare a design with a specification; we do not implement this since designers often have a software version, not a specification. Our work makes the following main contributions:

- an abstract approach allowing verification of both hardware and software designs, including hardware-software codesigns and runtime reconfiguration;
- a prototype implementation for verifying designs combining C software and Maxeler streaming hardware;
- case studies demonstrating the flexibility and the usability of the approach and prototype, including reverse time migration, a computationally-intensive finite difference computation from the geoengineering application domain.

To the best of our knowledge, this is the first effort verifying the compliance of runtime reconfigurable hardware/software codesigns with a software specification. The automation of our approach and the integration of well-established tools into the prototype highlight the applicability of our work.

The rest of this paper is organized as follows: the next section details related work. Section III shows our abstract approach to verifying software and reconfigurable hardware, while section IV shows a prototype implementation targeting reconfigurable codesigns of C software and Maxeler hardware

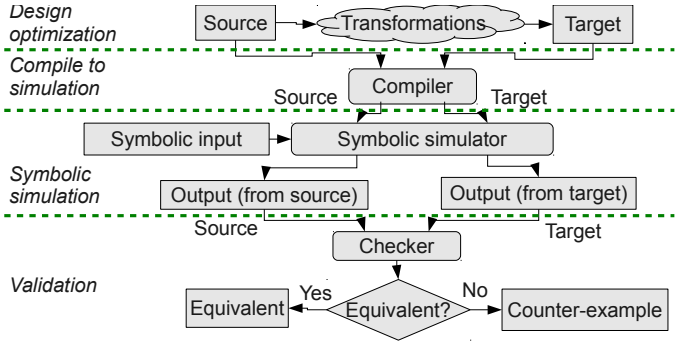

Fig. 1. Abstract verification design flow.

descriptions; section V evaluates our approach on several case studies. Section VI concludes and discusses future work.

II. Background

There is rich literature on software verification and on system-level specification of hardware/low-level software designs using high-level languages. Ball [5], Clarke *et al.* [6], and Lahiri *et al.* [7] present verification approaches for software combining predicate abstraction [8] and model checking [9] which have proven successful for verifying C programs. Several tools based on a *abstract-check-refine* paradigm [10] such as SLAM [11] or CBMC [12] have shown great results.

Domain specific variations of general-purpose languages such as SystemC have gained popularity for specifying systems with hardware and software parts. Existing work on SystemC verification includes: Habibi and Tahar [13] on design and verification of SystemC models at transaction level; Kroening and Sharygina [14] explore a verification approach based on automatic partitioning of hardware and software; Vardi [15] discusses formal techniques for SystemC verification.

Language-independent approaches to codesign verification have been studied, among others, by Chiodo *et al.* [16]. Their approach is based on communicating finite state machine (FSM) models and hardware/software synthesis. While FSMs are well suited for formal correctness argumentation, using them for specifying large complex systems can be hard, error-prone task. Edwards *et al.* [17] and Baleani *et al.* [18] present extensions and enhancements to that work which also consider FPGAs.

Other FPGA-related verification efforts include work by Singh and Lilleroth [19]. The authors verify the equivalence of FPGA cores using a model checker, and outline their initial ideas for runtime verification using a model checker at runtime. This approach is heavily performance limited and only applicable to simple components.

Our approach resembles work on design validation of imaging operations using symbolic simulation and equivalence checking [20]. However, we target streaming Maxeler hardware instead of Handel-C, we use our own symbolic simulator rather than ACL2, and we support runtime reconfiguration.

III. Hardware and software verification

Figure 1 shows our general verification flow. We refer to the reference model as *source model* and to the optimised model, which is being checked for equivalence, as *target model*. Either design could be software, hardware, or a combination.

Given source and target designs, our approach has four modular phases: (i) *Design Translation*: source and target designs are translated to an intermediate representation, keeping application/input-specific parts as small as possible. (ii) *Symbolic Simulation*: both designs are executed symbolically using the same inputs using a symbolic simulator for our intermediate language. (iii) *Output Combination*: to support weaker notions of equivalence than strict bisimulation-equivalence, such as stutter-equivalence, we pre-process the two output traces of the symbolic simulator in an output combiner before passing to the equivalence checker. (iv) *Validation*: an equivalence checker compares the symbolic outputs from source and target designs. The result is either successful verification, or a counterexample with symbolic inputs leading to different outputs.
Partitioning the workflow in these four steps, keeping the design translation small and straightforward, increases the modularity and generality of the approach. While we target C software and Maxeler hardware, the system can extend to other hardware input languages, such as Verilog.

To handle large systems, we model at word level. This limits the state-space explosion problem, but assumes correct operator implementations. We can still use bit-level correctness results for components in our system using a modular verification approach as operator implementations are combinatorial.

Our approach has some restrictions, which can be mitigated: (i) we assume synchronous hardware with one global clock; (ii) some data such as array sizes must be numeric, not symbolic; (iii) data-dependent control flow can still cause state-space explosion. Using numeric values for data that cannot be abstracted to symbolic values rarely poses a problem in practice: for example there is only a limited set of useful image sizes which need to be verified. Data-dependent control flows can be addressed by verifying different modes separately.

Finally, our approach can symbolically compare hardware and software implementations of same algorithm—often the translation between the two is a source of bugs.

A. The Model for Codesign

Our model for software-hardware codesign is a software host with an application programming interface (API) for loading and running streaming hardware designs. Our model and the API calls are *synchronous*, meaning that the software halts until the hardware returns. While this limits parallelism, this can verify simple hardware-software partitioning which may help to eliminate some bugs in a more general, asynchronous design (here asynchronous means the hardware and software run in parallel). The API contains the following calls: (i) `load`: loads a streaming hardware design compiled with our hardware compiler; (ii) `run`: runs a previously-loaded hardware design for a given numeric cycle count, with one or more input or output arrays, which must match stream inputs and outputs on the hardware design; (iii) `set_scalar`: sets a scalar hardware input to a given value, which will apply to the hardware design on the next call to `run`.

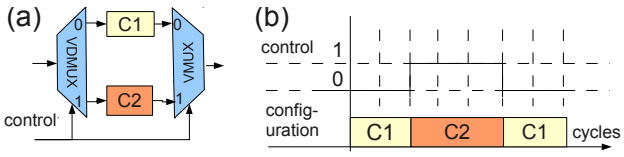

Fig. 2. (a) two mutually-exclusive configurations connected by a multiplexer (VMUX)-demultiplixer (VDMUX) pair; (b) Timing of reconfiguration modelled by virtual multiplexers; when the control line of the multiplexer-demultiplexer pair changes, the corresponding design is configured on the same cycle.

B. Runtime Reconfiguration

We extend our approach to support run-time reconfiguration by adding to our API a call that allows the user to load multiple streaming hardware designs and switch between them by writing to a particular scalar input. Our approach compiles reconfiguration to symbolic simulation using the concept of *virtual multiplexers* to represent different configurations [21].

Figure 2 shows (a) how mutually exclusive configurations are bracketed by virtual multiplexer-demultiplexer pairs, and (b) the timing of our reconfiguration model.

IV. Prototype System

We implement our approach for two design inputs: C software and Maxeler hardware; To verify software-hardware codesign, the source design is a C program, and the target design is a C program driving a Maxeler hardware design. The approach is not resticted to this scenario; other combinations of software and hardware could be used as source and target designs.

In our prototype, compilation to hardware happens in two phases: first the design is interpreted to compile away any metaprogramming features, such as conditional compilation (for C) or arbitrary Java programs which, when run, produce the final design (for Maxeler hardware). Our compiler then compiles into design input for our symbolic simulator.

Next, the user applies symbolic inputs to the design; the user can define these manually, or the simulator can automatically generate symbolic inputs based on a template. Our output combiner translates both symbolic outputs for the Yices SMT (Satisfiability Modulo Theories) solver [22], which checks whether symbolic output expressions are actually equivalent. Since different implementations may produce the same output on different cycles, or at different rates, the user can parameterize the output combiner to ignore some initial cycles (allowing different startup latencies), or to filter some results from either input (allowing different rates).

A. The Software Input Language

Our approach allows software designs to be described in a C-like language with restrictions: there is no standard library and no dynamic memory allocation; such restrictions allow useful software from some application domains, such as embedded software or digital signal processing, to be verified. We could allow dynamic memory allocation, but with numeric allocation sizes only. We extend C with a `write` statement and `read` expression to respectively allow symbolic inputs to and outputs from the design. Practical implementations could extend existing C input/output facilities to symbolic input/output.

Our approach compiles from software to symbolic simulation using a syntax-directed scheme, translating expressions to combinatorial hardware and statements to a one-hot encoded state machine. This means the simulation is effectively scheduled, with one assignment statement per simulation cycle, but makes it easy for the designer to calculate when the outputs corresponding to a symbolic input will emerge.

B. The Hardware Input Language

Our hardware design description is a simplified version of the Maxeler MaxJ streaming hardware description language, which allows the user to describe streaming hardware designs as Java programs written using their Java class library and Java language extensions. When run, the program builds the dataflow graph of the streaming design, compiles the graph into a HDL (Hardware Description Language) implementation, and calls FPGA vendor tools to compile the HDL into a bitstream.

The streaming hardware design consists of a data path reading from one or more stream inputs, and writing to one or more stream outputs, one per cycle. Scalar inputs are set once per run of the stream and are constant for each run. A control path controls the design.

Our hardware input language comprises the data path of a Maxeler description, plus the control path using counters; there are restrictions: the control path is not symbolic, and models numeric counters only. This means the verification can cover all numeric states but that if the bounds of the counters are set at compile time, the design must be verified for each parameter value. We do not currently model Maxeler's state machines; these could be modelled like the counters.

C. Runtime Reconfiguration.

The Maxeler tools currently allow whole-chip reconfiguration but not partial reconfiguration. Unlike real devices, our reconfiguration model does not allow devices to run during reconfiguration, but does allow designs which preserve state (for example, intermediate results from the previous configuration).

Our extension adds a statement to Maxeler's Java dialect grammar: `reconfigure_if ( e ) s else s` where `s` and `e` are non-terminals for statements and expressions, respectively, and `reconfigure_if` is a new keyword. Unlike the if-statement, this leads to run-time decisions, rather than compile-time decisions in Maxeler kernels. The meaning is: `e` is a run-time boolean expression; when `e` is true, the FPGA is configured to perform `s1`, otherwise it is configured to perform `s2`. The tools must (a) compile the expression into a reconfiguration controller, such as a soft processor on a Xilinx device, (b) compile `s1` and `s2` into separate bitstreams that can load on demand depending on `e`. Several `reconfigure_if` statements can be composed to model multiple configurations. Figure 3 shows the datapath of the run-time reconfigurable version, where the scalar input switches between constant multiplier sets; we factor out common parts of the configurations for clarity. To swap between configurations, we modify the software to to write to the scalar input `conf`.

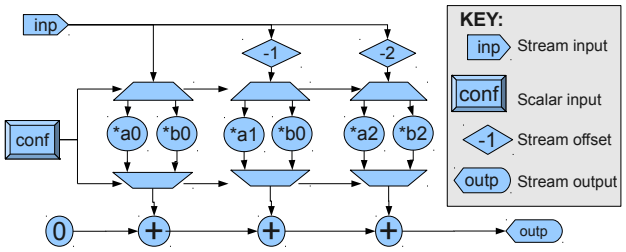

Fig. 3. Run-time reconfigurable 1D convolver design. The multiplexers and demultiplexers are virtual, switching between partial configurations. Note we gang the multiplexers: they share a control input. Writing to scalar input `conf` reconfigures from one coefficient set to the other.

V. Results and evaluation

We evaluate our approach by applying it to (a) 1D convolution, (b) Reverse time migration.

Experimental setup: we develop compilers from streaming hardware and software to symbolic simulator inputs, an output combiner and use Yices version 1.0.34.

1D convolution: we compare various 1D convolver implementations, with window size $W = 3$ and N stream inputs, comparing software with: *(1)* streaming hardware; *(2)* software plus hardware; *(3)* software plus reconfiguring hardware.

(1): software versus hardware: the software produces one result every 9 cycles, the hardware one per cycle. The extra latency of the software comes from scheduling one assignment per cycle. While the software takes 9 cycles to produce its first result, the hardware produces results immediately, but the first $W - 1$ are partial, using stream offsets before the start of the stream. To verify the designs as equivalent, we (i) parameterize the output combiner to ignore the first 9 software results and 8 of every 9 thereafter; (ii) duplicate software inputs 9-fold.

(2): software versus software driving one hardware configuration. The software-hardware design takes $3N$ cycles: N to load data, N to run and N to copy back to software.

(3): we modify both designs to run the first $N/2$ inputs with one coefficient set, the remaining $N/2$ with another. Applying the same skipping and duplication to the inputs and outputs as above, the designs are verified equivalent.

Reverse time migration (RTM): we implement a one-dimensional core of the RTM algorithm, which iterates over a grid, summing corresponding neighbours of a grid element, multiplying by a coefficient array. We compare two versions: one where the coefficients are scalar inputs, and one where the design has two configurations, one per coefficient set. Our verification approach shows that the two designs are equivalent.

VI. Conclusion

We present our approach to verifying the codesign of software and streaming hardware, which allows reference software to be verified as equivalent to designs using software and hardware together, and extends to run-time reconfigurable hardware. We apply our approach to several examples.

Current and future work includes adapting our approach to other input languages such as Verilog. Second, we would like to verify compile-time parametrisable designs once, rather than once per parameter value; while the space of useful parameter values is typically small, highly parameterised designs could still take a long time to verify.

Acknowledgements: Thanks to the reviewers for their comments. The research leading to these results has received funding from European Union Seventh Framework Programme under grant agreement number 287804, 248976 and 257906. The support by UK EPSRC, the HiPEAC NoE, the Maxeler University Program, and Xilinx is gratefully acknowledged.

References

[1] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, pp. 576–580, Oct. 1969.
[2] R. W. Floyd, "Assigning Meaning to Programs," in *Proceedings of the Symposium on Applied Maths*, vol. 19. AMS, 1967, pp. 19–32.
[3] E. W. Dijkstra, *A Discipline of Programming*. Prentice-Hall, 1976.
[4] T. Todman and W. Luk, "Verification of streaming designs by combining symbolic simulation and equivalence checking," in *FPL '12*, 2012.
[5] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani, "Automatic predicate abstraction of C programs," in *PLDI '01*. New York, NY, USA: ACM, 2001, pp. 203–213.
[6] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav, "Predicate Abstraction of ANSI-C Programs Using SAT," *Form. Methods Syst. Des.*, vol. 25, no. 2-3, pp. 105–127, Sep. 2004.
[7] S. Lahiri, R. Bryant, and B. Cook, "A Symbolic Approach to Predicate Abstraction," in *CAV '03*, ser. LNCS, W. Hunt and F. Somenzi, Eds. Springer Berlin, 2003, vol. 2742, pp. 141–153.
[8] S. Graf and H. Saidi, "Construction of abstract state graphs with PVS," in *CAV '97*, ser. LNCS, O. Grumberg, Ed. Springer Berlin / Heidelberg, 1997, vol. 1254, pp. 72–83.
[9] E. M. Clarke, O. Grumberg, and D. Peled, *Model checking*. MIT Press, 2001.
[10] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Lazy abstraction," in *POPL '02*. New York, NY, USA: ACM, 2002, pp. 58–70.
[11] T. Ball and S. Rajamani, "The SLAM Toolkit," in *CAV '01*, ser. LNCS, G. Berry, H. Comon, and A. Finkel, Eds. Springer Berlin / Heidelberg, 2001, vol. 2102, pp. 260–264.
[12] E. Clarke, D. Kroening, and F. Lerda, "A Tool for Checking ANSI-C Programs," in *TACAS '04*, ser. LNCS, K. Jensen and A. Podelski, Eds. Springer Berlin / Heidelberg, 2004, vol. 2988, pp. 168–176.
[13] A. Habibi and S. Tahar, "Design and verification of SystemC transaction-level models," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 14, no. 1, pp. 57–68, Jan. 2006.
[14] D. Kroening and N. Sharygina, "Formal verification of SystemC by automatic hardware/software partitioning," in *MEMOCODE '05*, Jul. 2005, pp. 101–110.
[15] M. Y. Vardi, "Formal Techniques for SystemC Verification; Position Paper," in *DAC '07*, Jun. 2007, pp. 188–192.
[16] M. Chiodo, P. Giusto, A. Jurecska, H. C. Hsieh, A. Sangiovanni-Vincentelli, and L. Lavagno, "Hardware-software codesign of embedded systems," *Micro, IEEE*, vol. 14, no. 4, pp. 26–36, Aug. 1994.
[17] S. Edwards, L. Lavagno, E. A. Lee, and A. Sangiovanni-Vincentelli, "Design of embedded systems: formal models, validation, and synthesis," *Proceedings of the IEEE*, vol. 85, no. 3, pp. 366–390, Mar. 1997.
[18] M. Baleani, F. Gennari, Y. Jiang, Y. Patel, R. K. Brayton, and A. Sangiovanni-Vincentelli, "HW/SW partitioning and code generation of embedded control applications on a reconfigurable architecture platform," in *CODES '02*. New York, NY, USA: ACM, 2002, pp. 151–156.
[19] S. Singh and C. J. Lillieroth, "Formal Verification of Reconfigurable Cores," in *FCCM*. IEEE Computer Society, 1999, pp. 25–32.
[20] K. W. Susanto, T. Todman, J. G. F. Coutinho, and W. Luk, "Design Validation by Symbolic Simulation and Equivalence Checking: A Case Study in Memory Optimization for Image Manipulation," in *SOFSEM*, ser. LNCS, vol. 5404. Springer, 2009, pp. 509–520.
[21] P. Lysaght and J. Stockwood, "A simulation tool for dynamically reconfigurable field programmable gate arrays," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 4, no. 3, pp. 381–390, Sep. 1996.
[22] B. Dutertre and L. de Moura, "The YICES SMT Solver," Computer Science Laboratory, SRI International, 333 Ravenswood Avenue, Menlo Park, CA 94025 - USA, Tech. Rep., 2006.