In-circuit temporal monitors for runtime verification of reconfigurable designs

Tim Todman
Department of Computing
Imperial College London
180 Queen's Gate, London
SW7 2AZ, UK
timothy.todman@imperial.ac.uk

Stephan Stilkerich
Airbus Group Innovations
Willy-Messerschmitt Str. 1,
85521 Ottobrun
stephan.stilkerich@airbus.com

Wayne Luk
Department of Computing
Imperial College London
180 Queen's Gate, London
SW7 2AZ, UK
w.luk@imperial.ac.uk

ABSTRACT

We present designs for in-circuit monitoring of custom hardware designs implemented in reconfigurable hardware. The monitors check hardware designs against temporal logic specifications. Compared to previous work, which uses custom hardware to monitor software, our designs can run at higher speeds and make better use of hardware resources, such as shift registers and embedded memory blocks. We evaluate our monitor circuits on example hardware designs targeting FPGA implementation, showing that they have low overhead in terms of circuit area, and can run at the same speed as the circuits they monitor.

1. INTRODUCTION

As hardware technologies in general, and reconfigurable hardware devices such as field-programmable gate arrays (FPGAs) in particular continue to improve, designs implemented on them become larger and more complex, and hence harder to verify. Many approaches have been tried to verify such designs, from traditional exhaustive simulation, to formal verification.

A promising approach to run-time verification is to add *in-circuit assertions* – circuits monitoring Boolean expressions which must be true if the design is running correctly [1] – to a reconfigurable design [2], allowing running hardware designs to be monitored. Although useful, simple Boolean conditions cannot capture all correctness conditions for a circuit; in particular, such conditions cannot describe time-dependent behaviour, such as asserting that when one signal becomes true, another signal must be asserted within a bounded number of cycles. Such assertions are useful for designing circuits such as state machines and memory controllers.

This paper proposes *in-circuit, temporal logic-based monitors* for verifying time-dependent behaviour of circuits. Temporal logic allows properties involving time to be specified. Although similar circuits have been proposed before, these targeted the verification of programs running on soft processors implemented on an FPGA fabric [3]. In contrast, our designs target verification of hardware designs (such as the soft processor itself), meaning that they must run at the full circuit speed. We develop multiple architectures for our monitors, including one derived from a novel transformation reducing the amount of computation from $O(N)$ to $O(\log N)$.

While our monitor designs must be added to the design under test at compile-time, recent work has shown how to accelerate this process, allowing monitoring hardware to be added incrementally without requiring completely rerunning FPGA place-and-route tools. Such techniques allow our in-circuit monitors to be added to an existing design [4, 5].

This paper makes the following contributions:

- An approach to runtime verification of hardware designs using in-circuit monitors for temporal logic specifications;
- Feedforward and feedback architectures for temporal logic monitors;
- Proof of an $O(N)$ to $O(\log N)$ optimization of such temporal monitors;
- Evaluation, showing resources used and speeds of circuits with our monitor designs.

The rest of the paper is organized as follows. Section 2 reviews background and related work. Section 3 shows our approach to in-circuit temporal-logic based monitors. Section 4 describes hardware architectures for our monitors. Section 5 outlines a proof of correctness for the optimized architecture. Section 6 evaluates our monitor designs using various examples. Finally, Section 7 concludes and suggests future work.

2. BACKGROUND

In-circuit assertions: Traditionally, hardware designs are validated by extensive simulation, but as designs become larger, the simulation space becomes impractically large. Assertions – Boolean expressions which must be true if the design is functioning correctly – have been proposed for hardware designs [2].

Temporal logic monitors for software and hardware: Several researchers implement temporal logic monitors. Backash et al [6] verify multicore system-on-chip (MPSoC) designs using linear temporal logic monitors compiled from a higher-level specification. Their work targets software designs implemented on the MPSoC, whereas we target hardware designs;

 ACM 978-1-4503-3520-1/15/06...$15.00
http://dx.doi.org/10.1145/2744769.2744856

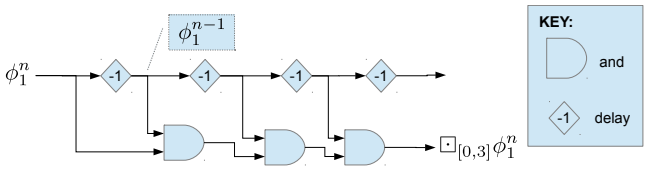

Figure 1: Straightforward feedforward *invariant* operator $\boxdot_{[0,3]}\phi_1$.

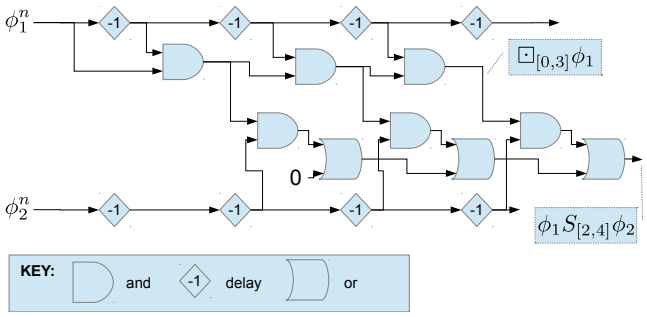

Figure 2: Straightforward feedforward *since* operator $\phi_1 S_{[2,4]}\phi_2$.

their monitors can run on FPGAs up to 50MHz whereas our feedforward monitor architectures can be arbitrarily pipelined, and hence run at high clock rates, up to 300MHz..

The Property Specification Language (PSL) is based on linear temporal logic and used in both static and runtime hardware verification. Several researchers have studied synthesizing hardware monitors from PSL expressions, such as work by Borrione et al [7]. The area used by the monitors "increases gracefully" with the observation window size; for our optimized feedforward design, registers increase linearly, but logic gates increase logarithmically.

Thati and Roşu [8] develop algorithms for monitoring MTL specifications which break MTL formulae into subformulae; we also use subformulae to develop our monitor architectures. The total number of memory bits needed to monitor is similar to our feedforward architectures, but our optimized architecture uses only a logarithmic amount of hardware resources.

The closest work to ours is by Reinbacher et al [3], in which the system under test is a software program running on a soft processor implemented on a reconfigurable hardware device. The soft processor is augmented with hardware monitors for temporal properties written in past-time metric temporal logic. Unlike this work, we monitor streaming hardware, not software, meaning that our monitors must run at the same rate as the rest of the hardware design. Their monitors are memory efficient but complex; we trade complexity for throughput. Although our monitors can have higher asymptotic complexity, we show that they make better use of hardware resources such as shift registers and embedded memory blocks.

3. RUNTIME VERIFICATION MONITORS

This section shows our approach to runtime verification, and details our abstract specification language for runtime hardware monitors for hardware designs.

In our approach, we verify that design meets a specification. The user separates the design specifications into compile-time and run-time properties, where run-time properties cannot be verified at compile-time, because they depend on run-time data. Compile-time properties are checked by compile-time methods such as symbolic simulation.

The user captures run-time design properties to be checked as temporal logic specifications. The specifications are written in past-time MTL (ptMTL), the past-time fragment of Metric Temporal Logic (MTL). We choose ptMTL to specify run-time properties as specifications that target synchronous hardware designs, with a single global clock.

Metric Temporal Logic extends linear temporal logic with operators expressing time bounds [9]. In our approach, time is represented by the global system clock applied to the hardware, so temporal properties are expressed in terms of intervals of clock cycles ($[t,t'] = \{i \in \mathbb{N}_0 | t \leq i \leq t'\}$) .

Specification grammar: Given a set of atomic propositions AP, the formal grammar of a ptMTL specification η is as follows:

$$\eta ::= true | false | \Sigma | \neg\eta | \eta \bullet \eta | \eta S_J \eta$$

where $\Sigma \in AP$, $\bullet \in \{\wedge, \vee, \rightarrow\}$, and interval $J = [t,t']$ for $t, t' \in \mathbb{N}_0$, and $t' \geq t$. Essentially, a ptMTL formula is a Boolean formula augmented by one temporal monitoring operator, S_J.

An execution e is a sequence of system states s_t, where $t \in \mathbb{N}_0$. For a ptMTL formula η, time $n \in \mathbb{N}_0$ and execution e, we inductively define $e^n \models \eta$, meaning η holds at time n of execution e as:

$e^n \models true$	is true
$e^n \models false$	is false
$e^n \models \Sigma, \text{where} \Sigma \in AP$	iff Σ holds in state s_n
$e^n \models \neg\eta$	iff $e^n \not\models \eta$
$e^n \models \eta_1 \bullet \eta_2$	iff $e^n \models \eta_1 \bullet e^n \models \eta_2$, where $\bullet \in \{\wedge, \vee, \rightarrow\}$
$e^n \models \eta_1 S_J \eta_2$	iff $\exists i \in [0,n] : (n-i \in J \wedge e^i \models \eta_2 \wedge \forall j \in [i+1, n] : e^j \models \eta_1)$
$e^n \models D(\eta)$	iff $\begin{cases} e^{n-1} \models \eta & \text{if } n > 0 \\ e^0 \models \eta & \text{otherwise} \end{cases}$

We denote repeatedly applying the delay operator D by a superscript, such that $D^2(\phi^n) = D(D(\phi^n)) = D(\phi^{n-1}) = \phi^{n-2}$ and $D^0(\phi^n) = \phi^n$.

Informally, the temporal *since* operator $\eta_1 S_J \eta_2$ means that η_2 was true at some time in the past (within range J), and since then, η_1 has been true. For hardware designs, this can be used to express useful properties that must hold for correct execution, for example that if signal s_1 becomes true, then within a bounded number of cycles, another signal s_2 must become true.

Other useful operators can be derived from the *since* operator S_J: $\boxdot_J \eta$ (*invariant within interval* J) and $\diamond_J \eta$ (*exists within interval* J). In terms of the *since* operator, these can be written as $\diamond_J \eta \equiv true S_J \eta$ and $\boxdot_J \eta \equiv \neg(true S_J (\neg\eta))$ respectively.

Useful properties of temporal operators: The temporal operator $S_{[a,b]}$ and operators derived from it ($\diamond$ and $\boxdot$)

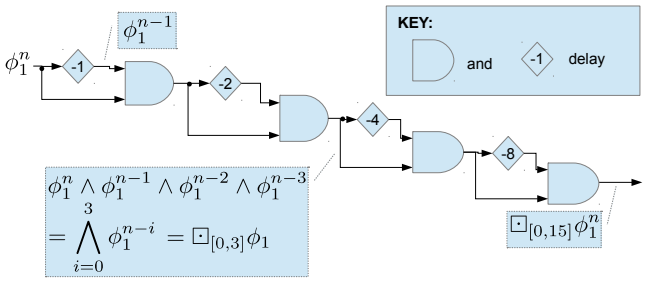

Figure 3: Optimized feedforward invariant operator$\boxdot_{[0,15]}\phi_1$, for 2-input LUTs (k=2).

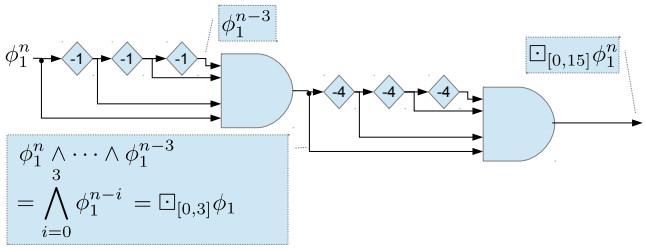

Figure 4: Optimized feedforward invariant operator$\boxdot_{[0,15]}\phi_1$, for 4-input LUTs (k=4).

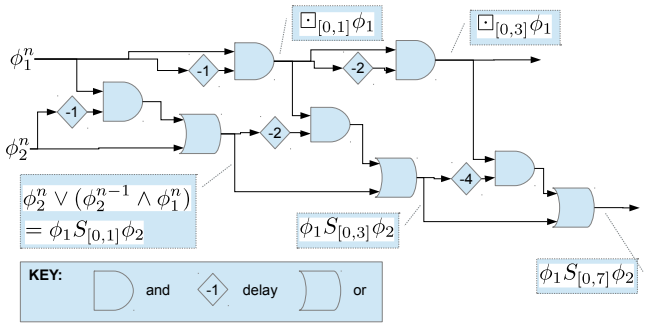

Figure 5: Optimized *since* operator architecture for $\phi_1 S_{[0,7]}\phi_2$. Intermediate results are labelled, showing how the result is built from $\phi_1 S_{[0,3]}\phi_2$, $\phi_1 S_{[0,1]}\phi_2$, $\boxdot_{[0,3]}\phi_1$ and $\boxdot_{[0,1]}\phi_1$.

have several useful properties which we use to derive circuit architectures for them.

Given non-empty interval $[a, b]$, current cycle $n \in \mathbb{N}_0$ and subformula ϕ, the *invariant within interval* operator can be written as:

$$\begin{aligned}\boxdot_{[a,b]}\phi &\equiv \neg(\mathit{true} S_{[a,b]}(\neg\phi)) \\ &= \neg\Big(\bigvee_{i=a}^{b}(\neg\phi^{n-i} \wedge \bigwedge_{j=0}^{i-1}\text{true})\Big) = \neg\Big(\bigvee_{i=a}^{b}(\neg\phi^{n-i})\Big) \\ &= \bigwedge_{i=a}^{b}\phi^{n-i} = \phi^{n-b} \wedge \phi^{n-b+1} \wedge \cdots \wedge \phi^{n-a} \qquad (1)\end{aligned}$$

where ϕ^i means the value of ϕ on cycle i.

An *invariant* operator over a range, $\boxdot_{[a,b]}$, can be split into subranges:

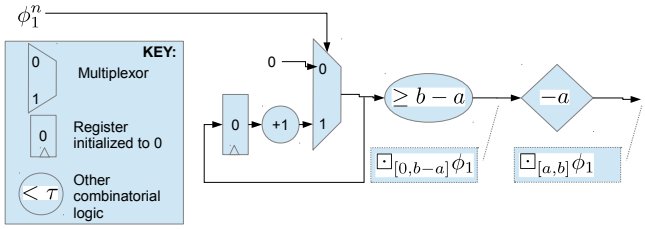

Figure 6: Feedback architecture for *invariant* operator $\boxdot_{[0,\tau]}\phi$.

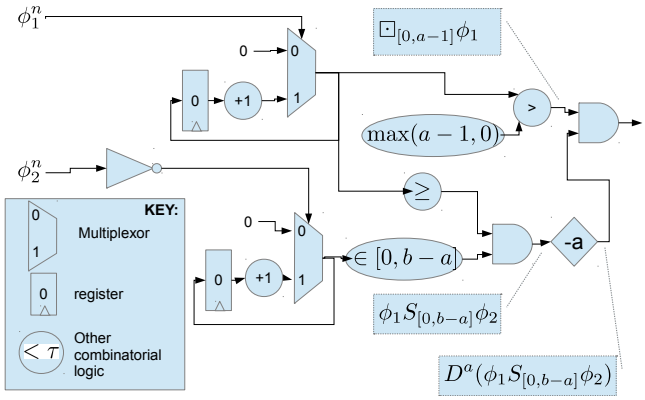

Figure 7: Feedback architecture for *since* operator $\phi_1 S_{[a,b]}\phi_2$.

$$\boxdot_{[a,b]}\phi = \boxdot_{[a,t]}\phi \wedge \boxdot_{[t,b]}\phi \qquad (2)$$

where $a \leq t \leq b$. Moreover, these subranges can overlap, so that: $\boxdot_{[a,b]}\phi = \boxdot_{[a,t1]}\phi \wedge \boxdot_{[t2,b]}\phi$ where $a \leq t_2 < t_1 \leq b$. This is possible because of the idempotency of logical conjunction, i.e. $\eta \wedge \eta = \eta$. Note that $\boxdot_{[a,b]}\phi = D^a(\boxdot_{[0,b-a]}\phi)$.

The temporal logic *since* operator $S_{[a,b]}$ also has useful properties. It can be written as:

$$\begin{aligned}\phi_1 S_{[a,b]}\phi_2 \equiv\ & \phi_2^{n-b} \wedge \phi_1^{n-b+1} \wedge \cdots \wedge \phi_1^n \\ \vee\ & \phi_2^{n-b+1} \wedge \phi_1^{n-b+2} \wedge \cdots \wedge \phi_1^n \\ & \cdots \\ \vee\ & \phi_2^{n-a} \wedge \phi_1^{n-a+1} \wedge \cdots \wedge \phi_1^n\end{aligned}$$

so the *since* operator over a range $S_{[a,b]}$ can be normalized into an operator over the range $[0, b-a]$:

$$\phi_1 S_{[a,b]}\phi_2 = D^a(\phi_1 S_{[0,b-a]}\phi_2) \wedge \boxdot_{[0,a-1]}\phi_1 \qquad (3)$$

4. IN-CIRCUIT MONITOR ARCHITECTURES

This section shows our designs for in-circuit monitors for temporal logic specifications. We devise both feedforward and feedback architectures implementing our operators; feedforward architectures do not cycle intermediate results back to the inputs, and can achieve high throughput at the expense of area. In contrast, feedback architectures can be more compact, but can have lower maximum clock speeds.

Architecture	Compute	Storage
Straightforward	$O(b)$	$O(b)$
Optimized	$O(log_2 b)$	$O(b)$

Table 1: Summary of different architectures for temporal logic monitors; both architectures can be used for both *since* and *invariant within interval* operators. The optimized version saves compute resources compared to the straightforward version.

4.1 Feedforward architectures

Straightforward architectures: We derive straightforward architectures for temporal logic operators directly from their specification.

Invariant within interval: We use equation 1 to derive an architecture for an *invariant* operator of the form $\boxdot_{[0,\tau]}\phi$; figure 1 shows an example for $\boxdot_{[0,3]}\phi_1$. To implement the more general form $\boxdot_{[a,b]}\phi$, we use equation 2.

Note that this design uses a linear amount of resources: b 1-bit registers and $b-a$ 2-input gates.

Since *within interval*: Similarly, given non-empty interval $[a,b]$, current cycle $n \in \mathbb{N}_0$ and subformulae ϕ_1 and ϕ_2, the *since* operator can be written as:

$$\phi_1 S_{[a,b]} \phi_2 = \bigvee_{i=a}^{b} \left(\phi_2^i \wedge \bigwedge_{j=0}^{i-1} \phi_1^{n-j} \right)$$

meaning that at some time i in the interval $[n-b, n-a]$, ϕ_2 was true, and for all cycles j, $i < j \leq n$ since then, ϕ_1 has been true. We derive a straightforward architecture in the same manner as for the *invariant within interval*, as shown in figure 2.

Again, the total hardware resources are of order $O(b)$: $2b$ 1-bit registers and $b + 2(b-a)$ 2-input gates.

Optimized architectures: We use the properties of the temporal operators to derive optimized architectures which use $O(\log b)$ computational area, rather than $O(b)$.

Invariant within interval: Using equation 2, we can implement a power-of-2 length range as:

$$\boxdot_{[0,2^l-1]}\phi = D^{2^{l-1}}(\boxdot_{[0,2^{l-1}-1]}\phi) \wedge \boxdot_{[0,2^{l-1}-1]}\phi$$

Applying this equation recursively leads to the architecture in figure 3. Note that while the total number of register bits is still b, the number of 2-input gates is reduced to $O(\log b)$.

Moreover, this architecture can take advantage of wider combinatorial operators afforded by multi-input lookup tables (LUTs) in many FPGAs: figure 4 shows an example for LUTs with $k = 4$ inputs, where the total LUT usage is $log_k(\tau)$.

Since *within interval*: Similarly, the same approach can apply to the *since* operator, using equation 3 to recursively implement the operator in terms of smaller *since* and *invariant* operators; figure 5 shows an example. Again, the computational resources are reduced to $O(\log b)$.

Table 1 summarizes the different architectures we propose for temporal logic monitors.

4.2 Feedback architectures

We derive feedback architectures for the *since* and *invariant* operators based on counters. Firstly, an *invariant previously* operator can be derived directly from its natural language specification: $\boxdot_{[0,\tau]}\phi$ is true iff ϕ has been true for the last τ cycles, including the current cycle. This can be implemented using a counter enabled when ϕ is asserted, and reset otherwise, plus logic to output true when the counter value exceeds τ; Figure 6 shows this design. To implement the *invariant* operator $\boxdot_{[a,b]}$, this design can be used to implement $\boxdot_{[0,b-a]}$, with shift registers to delay the result by a cycles.

Total resources used by this design are w_t bits for the register plus a bits for the shift register totalling $w_t + a = O(a)$; total compute resources are $O(1)$ (does not depend on a or b).

Similarly, a feedback architecture for the *since* operator can be derived from its specification and the property $\phi_1 S_{[a,b]} \phi_2 = D^a(\phi_1 S_{[0,b-a]} \phi_2) \wedge \boxdot_{[0,a-1]}\phi_1$. Figure 7 shows an implementation directly derived from the property. The top counter implements $\boxdot_{[0,a-1]}\phi_1$, while the bottom counter checks that the last time ϕ_2 was not asserted lies within $[0, b-a]$, and that ϕ_1 has been asserted at least since then.

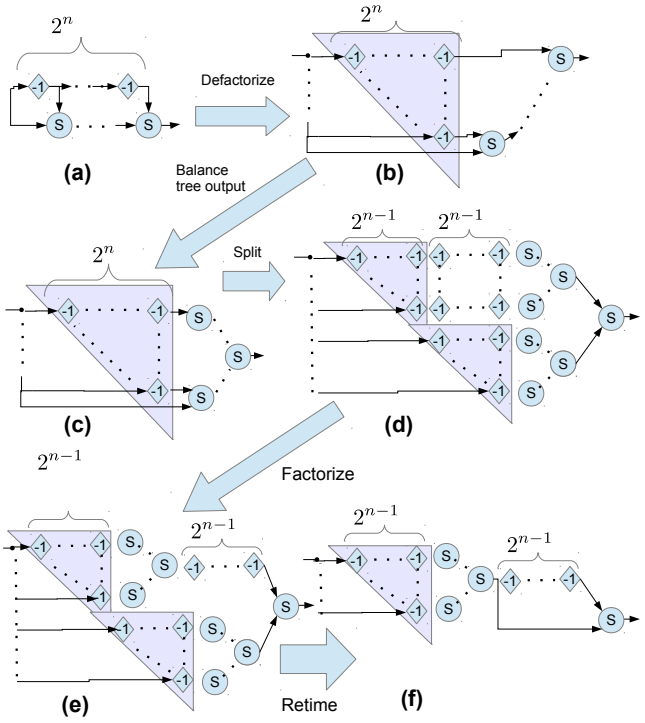

Figure 8: Proof of $O(b)$ to $O(\log b)$ optimization.

5. PROOF OF OPTIMIZATION

This section outlines a proof that our optimized designs ($O(\log b)$ computation, where b is the maximum number of cycles that the temporal operator looks into the past) are equivalent to the straightforward design with ($O(b)$) computation. The proof is not limited to temporal logic assertions; it applies to any associative operator.

Although we have implemented the steps of this proof using the Ruby hardware description language [10], we outline the proof graphically here, to avoid having to introduce a language in the space available.

Figure 8 shows the main proof steps: (1) First, start with the straightforward implementation of $\boxdot_{[0,\tau]}$ (figure 8(a)). The operators are labelled S because the proof applies to any

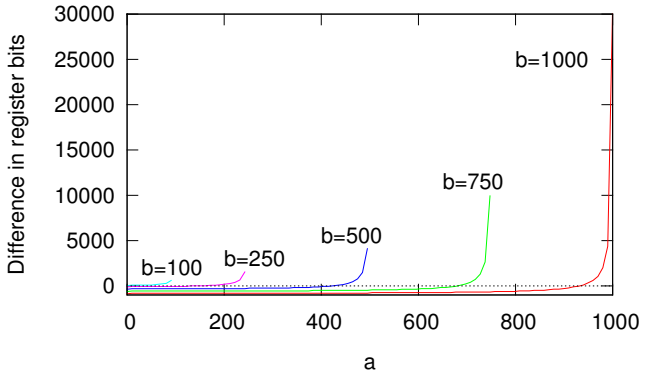

Figure 9: Differences in numbers of register bits used versus parameter a, for various values of b, comparing our design and that of Reinbacher et al.

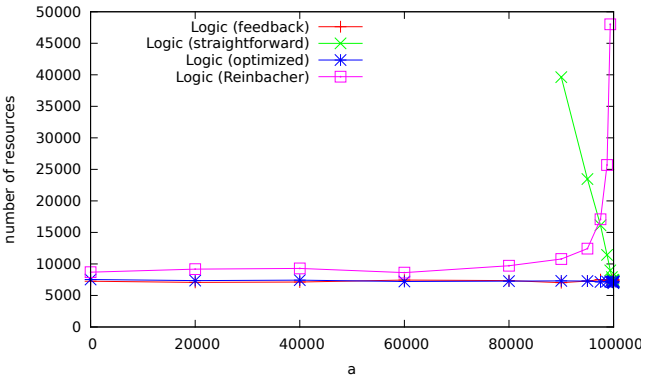

Figure 10: Resources (LUTs) used for FPGA implementation of various *invariant* monitor designs versus a, for $b = 10^5$.

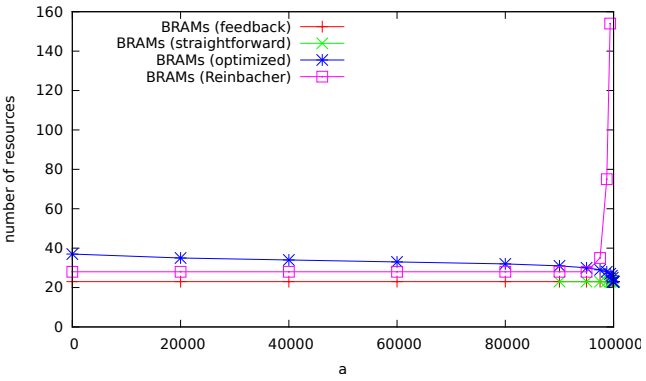

Figure 11: Resources (BRAMs) used for FPGA implementation of various *invariant* monitor designs versus a, for $b = 10^5$.

associative operator S; (2) Unfactorize the delay elements into a triangle of delays (figure 8(b)); (3) Since S is associative, replace the unbalanced tree of S operators with a balanced tree (figure 8(c)); (4) Split the delay triangle vertically into two halves: at the bottom, a delay triangle half the size of the input; at top, the same size of delay triangle followed by a set of delay chains; similarly, split the balanced tree

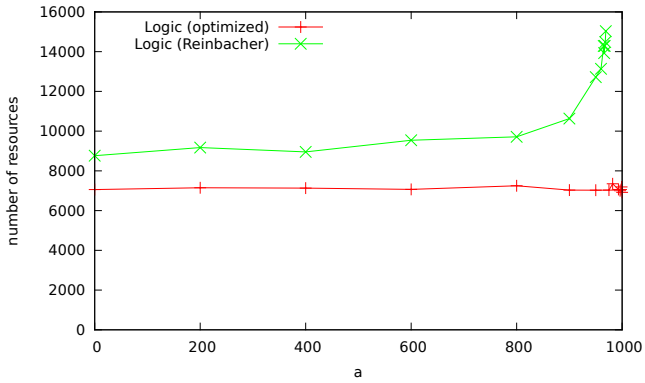

Figure 12: Avionics case study: logic resources used for FPGA monitors, for $b = 10^3$.

of S operators into two halves (figure 8(d)); (5) Retime the delay chains in the upper half to move them after the upper part of the balanced tree of S operators (figure 8(e)); (6) Finally, factorize the two identical delay triangles followed by balanced trees of S operators (figure 8(f)). (7) If we assume we have already proved the transformation for the left-hand side of figure 8(f) (the inductive hypothesis), then figure 8(c) reduces to figure 3.

6. EVALUATION

In this section we evaluate our runtime monitor designs against previously published work, comparing: 1. Our implementation of straightforward monitors; 2. Our improved monitors, using feedforward and feedback designs; 3. Our implementation of Reinbacher et al's monitors [3], which trade complexity for memory size.

Experimental setup: we implement designs using Maxeler's compiler (version 2013.2.2), aided by Xilinx ISE version 13.3, targeting a Maxeler MAX3 board with a Xilinx Virtex xc6vsx475t FPGA. Designs target a clock speed of 100MHz.

Asymptotic complexity: We compare resources used by Reinbacher et al's monitors to ours. Reinbacher's monitors use timestamps to record intervals where the monitored signal or signals are true. Each timestamp is of w_t bits, where the size is chosen to match the maximum length needed; we choose $w_t = 32$ to match their presentation. The total number of bits used by Reinbacher et al's monitors to monitor a range $[a, b]$ is given as [3]:

$$2w_t \left\lceil \frac{2\max(a,b) - \min(a,b) + 2}{\max(a,b) - \min(a,b) + 2} \right\rceil \quad (4)$$

which compares to b bits used by our designs for *invariant*, and $b + \lfloor b/2 \rfloor$ bits used for *since* operator.

Figure 9 plots the difference between the total number of register bits used by Reinbacher et al's designs and ours versus parameter a, for several values of b. We observe that although Reinbacher et al's designs use a lower number of register bits across most of the parameter space, the number is not much lower than for our design. For example, for $[a, b] = [0, 1000]$, our designs use about 1000 bits, whereas theirs use a minimum of 128. For $b < 200$, our designs always use fewer bits.

Moreover, where our designs use fewer register bits, they often use significantly fewer, due to the demoninator in

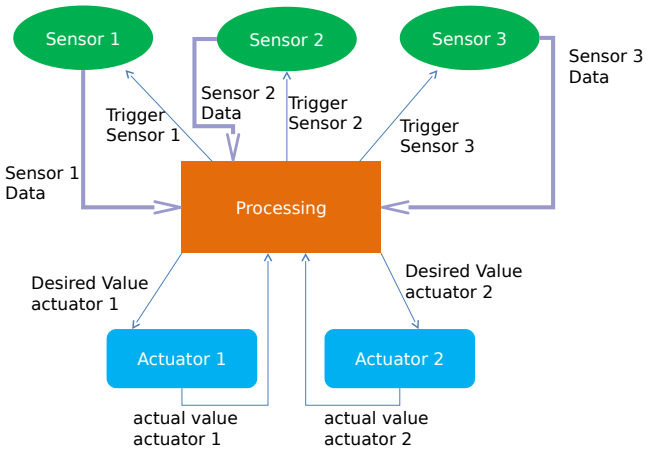

Figure 13: Avionics case study.

equation 4. The worst case is always along the line $a = b$; for example, for $[a, b] = [1000, 1000]$, our design uses about 1000 bits, whereas their design uses $32,064$ bits.

Hardware speed and area versus monitor size: We implement our designs and our implementation of Reinbacher's designs on FPGA hardware, using Maxeler's dataflow and state machine compilers. We implement designs for $b = 10^5$ and various values of a between 0 and b. Figure 10 plots FPGA logic resources (LUTs) used versus parameter a. We observe that both our optimized feedforward and feedback designs use an almost constant number of LUTs, a little lower than for Reinbacher et al's design for low values of a; for values of a approaching b, Reinbacher et al's design uses many more LUTs, due to the denominator of equation 4. The straightforward design uses many more LUTs as the registers are interleaved between logic gates, making it impossible for the tools to infer shift registers; for $a < 90,000$, the FPGA build does not complete. Figure 11 plots the number of Block random access memories (BRAMs) used versus a; again, the Reinbacher design uses many more as a approaches b.

Avionics case study: We consider a straightforward monitoring requirement from the avionics application domain. Figure 13 shows a schematic of an embedded avionics system. The design repeatedly requests input from its sensors; when the sensors reply, it requests changes from its actuators, which reply with the actual value they achieved. Informally, the specification is that each sensor or actuator must respond to requests within a bounded number of cycles (< 1000). We write this as a ptMTL specification: $\text{reply}_i \rightarrow \boxdot_{[t_1,t_2]}\text{request}_i$; every reply implies the corresponding request must have been made within the range $[t_1, t_2]$ cycles ago. Figure 12 shows our design uses fewer resources than the rival design.

7. CONCLUSION

We present techniques for in-circuit monitors for reconfigurable hardware, which can be used to verify that hardware circuits meet temporal logic specifications. Compared to previous work, our monitors map better to hardware resources such as shift registers and embedded memories, and can run at high clock rates.

Our current monitor designs assume a single global clock, but many practical FPGA designs have multiple clocks. We would like to examine how to monitor temporal properties of such designs, possibly by using a richer temporal logic to write specifications. We would also like to extend our work to future-time MTL. Streaming languages already allow forward stream offset operations, to look ahead in a stream.

Current and future work includes: extension to runtime reconfigurable designs, where the circuit can change, possibly while other parts of the design are still running. The possibilities include adaptively adding more monitors to a design, altering balance between monitors and design, monitoring a runtime reconfigurable design, and monitoring correctness of reconfiguration.

8. ACKNOWLEDGEMENTS

This work was supported by UK EPSRC, Maxeler, and Xilinx. The research leading to these results has received funding from the European Union Seventh framework programme (grant agreement number 318521) and the UK Engineering and Physical Sciences Research Council (award number EP/I012036/1).

9. REFERENCES

[1] Vasudevan, S.: What is assertion-based verification? SIGDA E-News **42**(12) (December 2012)

[2] Curreri, J., Stitt, G., George, A.D.: High-level synthesis of in-circuit assertions for verification, debugging, and timing analysis. International Journal of Reconfigurable Computing **2011** (2011)

[3] Reinbacher, T., Függer, M., Brauer, J.: Real-time runtime verification on chip. In Qadeer, S., Tasiran, S., eds.: Runtime Verification. Volume 7687 of Lecture Notes in Computer Science., Springer Berlin Heidelberg (2013) 110–125

[4] Hung, E., Wilton, S.: Incremental trace-buffer insertion for FPGA debug. IEEE Trans. on VLSI **22**(4) (April 2014)

[5] Hung, E., Todman, T., Luk, W.: Transparent insertion of latency-oblivious logic onto FPGAs. In: Field Programmable Logic and Applications (FPL), 2014 24th International Conference on, IEEE (2014) 1–8

[6] Backasch, R., Hochberger, C., Weiss, A., Leucker, M., Lasslop, R.: Runtime verification for multicore SoC with high-quality trace data. ACM Trans. Des. Autom. Electron. Syst. **18**(2) (April 2013) 18:1–18:26

[7] Borrione, D., Liu, M., Morin-Allory, K., Ostier, P., Fesquet, L.: On-line assertion-based verification with proven correct monitors. In: Information and Communications Technology, 2005. Enabling Technologies for the New Knowledge Society: ITI 3rd International Conference on. (Dec 2005) 125–143

[8] Thati, P., Roşu, G.: Monitoring algorithms for metric temporal logic specifications. Electronic Notes in Theoretical Computer Science **113**(0) (2005) 145 – 162 Proceedings of the Fourth Workshop on Runtime Verification (RV 2004) Fourth Workshop on Runtime Verification 2004.

[9] Alur, R., Henzinger, T.A.: Real-time logics: complexity and expressiveness. Information and Computation **104**(1) (1993) 35–77

[10] Sheeran, M.: Describing and reasoning about circuits using relations. In McEvoy, K., Tucker, J.V., eds.: Theoretical Foundations of VLSI Design. Cambridge University Press (1990) 263–298 Cambridge Books Online.