DHOOM: Reusing Design-for-Debug Hardware for Online Monitoring

Neetu Jindal
IIT Delhi
neetu@cse.iitd.ac.in

Sandeep Chandran
IIT Palakkad
sandeepchandran@iitpkd.ac.in

Preeti Ranjan Panda
IIT Delhi
panda@cse.iitd.ac.in

Sanjiva Prasad
IIT Delhi
sanjiva@cse.iitd.ac.in

Abhay Mitra
IIT Delhi
abhaymitra2007@gmail.com

Kunal Singhal
IIT Delhi
knsn1994@gmail.com

Shubham Gupta
IIT Delhi
shubh.dec93@gmail.com

Shikhar Tuli
IIT Delhi
shikhartuli98@gmail.com

ABSTRACT

Runtime verification employs dedicated hardware or software monitors to check whether program properties hold at runtime. However, these monitors often incur high area and performance overheads depending on whether they are implemented in hardware or software. In this work, we propose DHOOM, an architectural framework for runtime monitoring of program assertions, which exploits the combination of a reconfigurable fabric present alongside a processor core with the vestigial on-chip Design-for-Debug hardware. This combination of hardware features allows DHOOM to minimize the overall performance overhead of runtime verification, even when subject to a given area constraint. We present an algorithm for dynamically selecting an effective subset of assertion monitors that can be accommodated in the available programmable fabric, while instrumenting the remaining assertions in software. We show that our proposed strategy, while respecting area constraints, reduces the performance overhead of runtime verification by up to 32% when compared with a baseline of software-only monitors.

KEYWORDS

Runtime Monitoring, Design-for-Debug Hardware

1 INTRODUCTION

Runtime verification is a lightweight formal verification technique that monitors *only one* (current) run, instead of considering all possible runs, of the system for possible violations of a set of (safety and invariant) properties specified as assertions [1]. The major factors contributing to a performance overhead in runtime monitoring are: (i) *instrumentation* of the program to generate events for the monitor; and (ii) *execution analysis*, performed either in lock step with program execution or *post facto*. Earlier approaches have reduced the overhead of instrumentation by using the existing on-chip tracing infrastructure such as ARM CoreSight, and the overhead of execution analysis by using dedicated hardware features [4]. However, these solutions are resource intensive, and require fixing at *design time* the available hardware resources.

In contrast, we propose and develop an *architectural framework* for runtime verification, targeting a feature already being explored in modern processors such as the Xilinx Zynq (usually for custom accelerators), namely a reconfigurable fabric adjacent to a core. The merits of our approach are that it is: (a) *resource efficient* – requiring very little space for runtime monitoring; (b) *resource aware* – implementing in software those assertions which cannot be accommodated in the available space on the reconfigurable fabric; and (c) *versatile* – working well with general (block-structured) programs, and where the monitors for assertions are synthesized at *compile time*.

We avoid the resource-intensive trace reconstruction step of [1] by a compile-time mapping of software events which are relevant to the assertion-monitors to low-level events on execution traces. Another contribution of our solution is a compile-time $O(n^2)$ algorithm (in the number of assertions) to dynamically select an effective set of assertions to be implemented in the reconfigurable hardware at different program points, when operating under an area constraint.

Our monitoring framework DHOOM uses a combination of the Design-for-Debug (DFD) hardware and the on-chip reconfigurable fabric attached to a RISC processor to offload the monitor computation from the main processor. We propose using a slim communication interface between the processor pipeline and the reconfigurable fabric by leveraging the trace buffer present in the DFD infrastructure. Complicated code instrumentation and communication costs between the processor and the monitors are avoided, *with few architectural changes*. The framework integrates with a standard compiler back-end since the information required for monitoring the program is limited to the variable-to-register mapping, and the range of Program Counter (PC) values that delimit the scope of an assertion.

Figure 1 illustrates the high-level architecture for implementing DHOOM. In RISC architectures, we can *continuously* monitor assertions related to register-allocated variables at a very fine granularity, with very low communication overhead, by observing the destination register values of instructions. This mechanism works efficiently for scoped invariant assertions (of the form $\Box((L \leq pc \leq U) \rightarrow \varphi)$), not merely assertions at program points.

Partly supported by DST-JST project "Security in the IoT Space" and Semiconductor Research Corporation project 2014-TJ-2528.

DAC '19, June 2–6, 2019, Las Vegas, NV, USA

ACM ISBN 978-1-4503-6725-7/19/06…$15.00
https://doi.org/10.1145/3316781.3317799

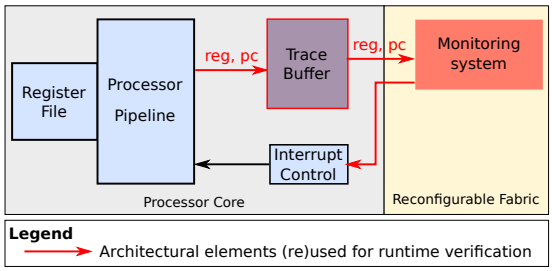

Figure 1: High-level schematic of DHOOM.

Moreover, the approach lets us specify program-specific assertions at *compile-time*, in contrast with earlier approaches involving hardware monitors, e.g., for embedded systems, where the assertions had to be fixed at *design-time*. The framework currently addresses a structured programming model with the usual control constructs of sequencing, choice, iteration and function call, including recursion.

The rest of the paper is organized as follows. In §2, we provide a brief overview of various approaches to runtime monitoring, especially in hardware. The DHOOM system architecture and work-flow is presented in §3. Space constraints on the reconfigurable fabric may not permit all monitors to be realizable in hardware. Accordingly, we present a selection algorithm that picks a maximal subset of assertions at different program points that can fit in the space available on the reconfigurable fabric, while minimizing the performance overhead (§4). The details of the prototype implementation of DHOOM, and case-studies highlighting its benefits are discussed in §5. The example programs are familiar illustrative algorithms from a data structures course. We conclude in §6 with a discussion on possible future work.

2 RELATED WORK

Several works have proposed customizing the architecture to make it amenable to online runtime verification [14]. Program properties are usually specified using a temporal logic, and these specifications are then synthesized into monitoring circuits, usually at design time. Better performance is at the expense of versatility.

Several other proposals exhibit versatility by exploiting the presence of multiple cores on chip [7, 15, 18]. Here, the execution results of one processor core are passed to the monitoring core through on-chip buffers. These proposals are resource intensive when compared with an unmonitored system.

Other approaches propose using a FPGA to analyze the execution traces generated by the DFD hardware [1, 4], or monitoring the pin activity of various devices (on the board) [9]. However, since execution traces do not contain any information on program features such as function calls and variable accesses, elaborate trace reconstruction hardware is used. We avoid such an extensive trace reconstruction step by translating the scope of each monitor into PC ranges, and *a priori* defining program behavior features in terms of events on execution traces. This helps us process the trace stream in a resource-efficient manner.

The integration of reconfigurable fabric with the main processor has been the subject of researchers' attention to accelerate the main computation [8, 17]. There are several works that propose using the reconfigurable fabric for implementing specific monitoring tasks, and book-keeping functions [5]. Runtime verification for such processor-FPGA systems was specifically considered in [16] where the traffic over the system bus was monitored for execution analysis. Since we can monitor register traffic, we are able to support finer-grained runtime verification. Our framework also supports selective implementation of monitors in the presence of other accelerators on the reconfigurable fabric. Our work also resembles other works which reuse DFD hardware for various purposes such as security [2] and additional functional memory [10, 12].

3 OVERVIEW OF DHOOM

3.1 Architecture

Figure 2 shows our proposed DHOOM architecture. In our framework, the processor core and its associated DFD hardware require no modifications. However, we assume that the updates to the register file are available through the instruction trace generated as suggested in [13]. A standard compiler such as LLVM is used to compile the source program. The resulting binary is examined to determine the following information: (i) the scope of assertions in terms of the PC ranges, and (ii) the variable to register mapping. The PC ranges thus identified are used by the monitoring system to configure the DFD hardware such that only execution traces of just those regions of execution where the specified assertions are active are stored into the trace buffer for further analysis, using existing trace conditioning logic [3]. The trace buffer acts as a FIFO where these execution traces reside before being read by the monitoring system present on the reconfigurable fabric. It is through the judicious use of the DFD hardware that we manage to decouple the processor core from the monitoring system.

Our architecture uses a slim communication interface between the processor and the monitoring system: It includes the PC, updates to the register file, and a control signal for interrupting the processor. This communication interface exploits the simplicity of the RISC architecture, where each instruction uses at most three operands: two reads and one write. Changes to the system state—consisting primarily of the register file and memory—are uniquely identified through the PC of the instruction causing the change. Another benefit of the RISC architecture is that memory accesses flow through the register file and are visible to the monitor. The centrality of the register file in such architectures makes monitoring simple, and aids in fine-grained monitoring. Thus we are able to continuously monitor scoped invariant properties efficiently.

In scenarios where the resources on DFD hardware such as the number of event triggers provided is less than the number of PC ranges to be tracked, we merge the PC ranges of the active assertions that are minimally separated. Since the PC ranges of interest to the monitors are known *a priori* and do not change during the lifetime of the application, this guarantees that few unnecessary execution traces are stored into the trace buffer. The PC ranges are passed into the reconfigurable fabric along with the register values so that the monitors can filter out any unnecessary traces. We reuse the dumping logic associated with the DFD hardware to stall the processor when the trace buffer is full, and to send its contents to the reconfigurable fabric. We refrain from using the existing core-fabric bus to send execution traces to the monitoring system so as

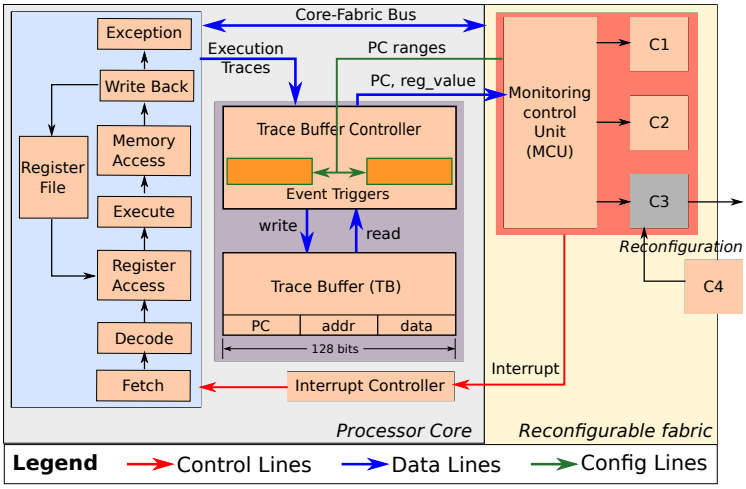

Figure 2: Detailed architecture of DHOOM

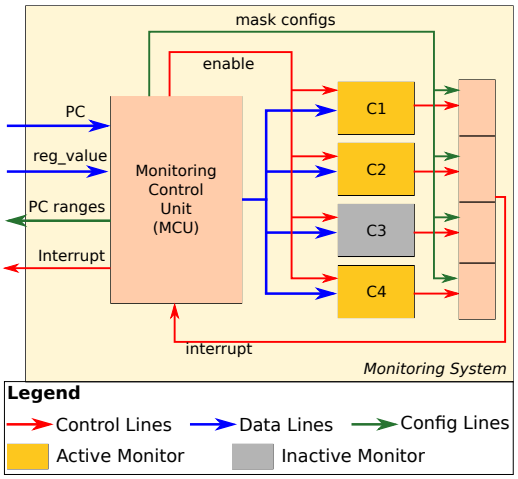

Figure 3: Monitoring system

to not interfere with the functioning of other on-fabric accelerators. Since not all instructions executed by the processor would update the variables being monitored, the monitoring system can mostly keep up with the processor (as shown by our experiments).

Figure 3 shows the internal details of the monitoring system. Each property to be monitored is implemented as a separate circuit module (labeled C1 to C4) in the reconfigurable fabric, which raises an interrupt if a violation is detected. An interrupt mask is used to either pass an interrupt to the processor, or suppress it, based on the scope of each assertion. The Monitoring Control Unit (MCU) receives the program counter and the updated register value from the trace buffer, and determines the modules that should be activated. The register value is ignored if it is not relevant to any module.

3.2 DHOOM Flow

We discuss the DHOOM flow through the example listing shown below. The programmer begins by appropriately annotating the source code with assertions that are to be monitored.

```
#ifdef C1
  bool goingLeft = false; int parent = -1;
#endif

void preOrder(struct node *root) {
#ifdef C1
  assert(((goingLeft) && (root->key <= parent)) ||
    ((!goingLeft) && (root->key >= parent)))
#endif
  if(root != NULL) {
    // C1 goes out of scope inside printf()
    printf("%d(%d) ", root->key, root->count);
#ifdef C1
    goingLeft = true; parent = root->key;
#endif
    preOrder(root->left);
#ifdef C1
    goingLeft = false; parent = root->key;
#endif
    preOrder(root->right); }}
```

In the above listing, if flag C1 is on, the assertion whether the Binary Search Tree condition (where the left subchild is less than or equal to, and the right subchild is greater than or equal to the parent) is checked, as it traverses the tree when printing the elements in pre-order. The assertion monitor comes into scope every time preOrder() is called, but goes out of scope on entering the printf() function.

The program is first compiled by disabling the macro C1, and the running time of the resulting binary is measured. Then the run-time of the binary generated by enabling the macro C1 is measured. The difference in the run-times gives the "baseline" performance cost of C1. Next, assertion C1 is translated into VHDL code and then synthesized to get its area overhead if implemented in hardware. These overheads are then examined to determine the benefits of implementing C1 in hardware vis-a-vis in software.

Let us consider a scenario where the monitor C1 is implemented in hardware. As the program executable obtained after disabling C1 is loaded on the processor for execution, the DFD hardware is configured and the reconfigurable fabric of the monitoring platform is simultaneously loaded and initialized with the monitor for C1. If the verification circuit reports a violation, an interrupt is generated for the core, which results either in termination of the program, or initiation of a recovery routine. Assertions within recursive functions pose no complication provided the variable-to-register mapping remains unchanged — the monitoring circuits can be reused across recursive calls because block-structured scoping rules prevent assertions from simultaneously viewing the state of local variables/parameters across different recursive function call instances.

3.3 Design complexities

In processors with complex design elements such as superscalar issue and out-of-order execution where multiple registers can be updated simultaneously and registers can hold speculative values respectively, the MCU would have to maintain the map of architectural registers along lines similar to those of [16]. The PC of retiring instructions would then have to be used for verification. The communication interface in this scenario would require a few

other internal signals to be part of the execution trace, and software events such as function calls and variable updates to the low-level events on trace stream would have to be mapped suitably. But essentially, the DHOOM flow would not change.

4 ASSERTION MAPPING ALGORITHM

Since the reconfigurable fabric is area-limited, we exploit partial reconfiguration to efficiently "time-share" assertions in hardware. For example, in Figure 2, a reconfiguration permits C4 to displace C3. Algorithm 1 (AssertMap) takes the following input parameters: a set of assertions A; the area cost c_i, of mapping an assertion A_i in hardware, the performance cost b_i, when A_i is implemented in software; and the reconfiguration time (R clock cycles) of the fabric. It selects a maximal subset of assertions $S \subseteq A$ that should be mapped to the reconfigurable fabric at different program points by associating with them a configuration W such that:

- $\sum_{i \in W} c_i \leq M$ where M is the maximum allowed area for implementing assertions in hardware, and
- The total execution time overhead O is minimal, where $O = mR + \sum_{i \in A-S} b_i$, and m is the number of times the fabric is reconfigured to modify the set of assertions mapped to hardware.

The above problem is NP-hard, being a general version of the register allocation problem, and AssertMap is a simple heuristic. Similar strategies have been employed in coverage and signal selection algorithms [6, 11].

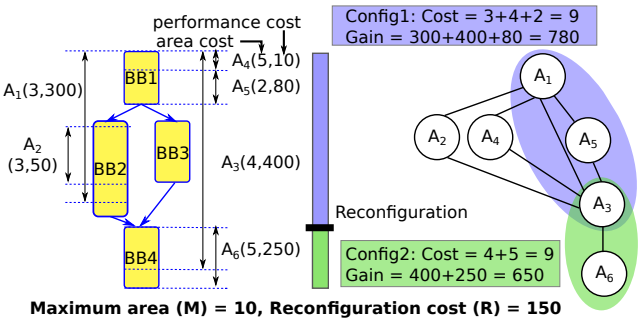

Figure 4: AssertMap Illustration

We maintain a subset $S \subseteq A$ of assertions that will be implemented in hardware, and create *Config*s consisting of groups of assertions that represent distinct configurations of the fabric. We first construct a graph $G(V, E)$ with nodes representing assertions and the edge e_{ij} representing the *temporal overlap* of nodes i and j. In the *for*-loop (lines 6-17) of procedure AssertMap we select the node (assertion v) that maximizes the cycles b_v saved if it is converted to hardware along with its temporal neighbours N that have been mapped to hardware, with the saving adjusted by R if a new reconfiguration is required ($k = 1$ in line 11). The total area occupied by them together has to be within M. If this node fits within an existing Config(W) then it is assigned to W, otherwise, a new config C is generated. It is possible that two or more assertions can share some logic if implemented in hardware, thereby the area

Algorithm 1 Assertion Mapping Algorithm

```
1:  procedure AssertMap
2:      Construct G(V, E)
3:      S = φ; V' = V
4:      while V' ≠ φ do
5:          max = 0; n = ϵ; W = φ;
6:          for all v ∈ V' do
7:              N = {u ∈ S | e_uv ∈ E}
8:              if c_v + Σ_{q∈N} c_q > M OR
                  multiple configs present in N then
9:                  V' = V' − {v}
10:             else
11:                 sav_v = b_v − kR
12:                 (where k = 1 if N = φ, else k = 0)
13:                 if sav_v ≥ max then
14:                     max = sav_v; n = v; W = N;
15:                 end if
16:             end if
17:         end for
18:         if n ≠ ϵ then
19:             S = S ∪ {n}
20:             if Config(W) exists AND
                  c_n + area(Config(W)) < M then
21:                 Assign n to Config(W)
22:             else
23:                 Create new config C
24:                 Assign n and W to C.
25:             end if
26:         else
27:             return
28:         end if
29:         V' = V' − {n}
30:     end while
31: end procedure
```

required to implement them would be less than the sum of individual areas and possibly result in smaller reconfiguration time. We use pessimistic estimates to simplify the analysis.

Figure 4 shows an illustration of the algorithm with $M = 10$ and a control flow structure with four basic blocks and assertion ranges as indicated, with assertions annotated with their (c_i, b_i) values. The algorithm begins by constructing the graph G as shown in the figure. A_3 is selected for hardware mapping in the first iteration of the while-loop because its gain ($b_3 = 400$) is maximum. A new configuration Config 1 begins. A_1 is selected in the second iteration ($b_1 = 300$), as its combined cost, with A_3, is still within the limit M ($c_3 + c_1 = 3 + 4 \leq 10$). A_1 is added to Config 1. A_6 is selected in the next iteration, but cannot be added to Config 1 because the total area would exceed M ($3 + 4 + 5 > M$), so a new Config 2 is started, consisting of A_3 and A_6. A_5 is selected in the next iteration and added to Config 1. A_2 and A_4 are not mapped to hardware because their area cost is too large.

The *while*-loop iterates a maximum of n times where n is the number of assertions, and the *for*-loop iterates over the number of remaining assertions. The overall complexity is $O(n^2)$, with the cost

of computing $area_v$ absorbed into the updates performed when an assertion is mapped to hardware. The above algorithm targets a general reconfigurable architecture, but is also applicable in the alternative formulation where a fixed subset out of a given set of assertions is to be selected for hardware implementation, as a degenerate case in which only one Config is used.

5 EXPERIMENTS

5.1 Setup

Our experimental setup consists of a single-core LEON3 processor in which DFD structures such as a trace buffer of size 1KB, trace conditioning and dumping logic similar to [3], were implemented. The instruction trace from standard LEON3 captures parameters such as program counter, opcode, instruction trap and time tag. We have modified it to capture register addresses, register values and the PC values instead. Each modified trace message continues to be 128-bits wide (as in standard LEON3 processor). This modified LEON3 processor, the monitoring control unit, and all the monitors were implemented in VHDL, synthesized using Xilinx ISE 14.1, and downloaded to a Xilinx Virtex 5 board (XC5VLX110T). This setup is on lines similar to [5]. The application executables were transferred onto the board and executed through GRMON. The profiling of the application was carried out through the performance monitor counters (PMCs) present within the LEON3 core and were programmed and read using the `l3stat` module of GRMON.

We emulate reconfiguration by implementing all the monitors in the Virtex 5 board, and activating/deactivating them as necessary. We count the number of times the 'Config' identified by our mapping algorithm changes over the application's execution, and add a fixed reconfiguration cost of 100000 cycles towards each such change. We study the working of the mapping algorithm for different area constraints of 500, 600, and 700 slices available on the reconfigurable fabric. We demonstrate the benefits of our proposed scheme through four applications of varying complexity, that cover several real-world verification scenarios.

5.2 Case Studies

Dijkstra's shortest path algorithm (DSP): The first application is an implementation of Dijkstra's shortest path algorithm, with assertions such as: (i) edge weights should not be negative and (ii) the accumulated path cost should be monotonically increasing. These assertions are checked on each vertex visited during the execution of the program, and therefore come and go out of scope frequently. Since these checks are live only at specific program points, and do not have to save any state between successive calls, they together occupied less than 500 slices on the reconfigurable fabric, and hence the MCU only had to activate and deactivate the assertions based on the PC ranges. However, the large number of monitors required the event triggers in the DFD hardware to merge PC ranges. The assertions, when implemented on the reconfigurable fabric, resulted in an 18% reduction in the execution time of DSP. We used this application to illustrate the effect of CPU stalls on the execution time while varying the trace buffer sizes and operating frequency of reconfigurable fabric (Figure 5). Note that the CPU stalls occur when the trace buffer is full and the processor waits for the monitors on the reconfigurable fabric to read the contents of the trace buffer as the two may be operating at different frequencies. We observe a reduction in performance improvement from 18% to 12% and 9% when the operating frequency decreases to half and quarter respectively as the processor core halts more often and for longer durations. However, the reduction at a particular frequency is not as severe when the size of the trace buffer increases to 2KB and 3KB as a larger trace buffer can hold the traces for longer, thereby resulting in fewer stalls of the processor core.

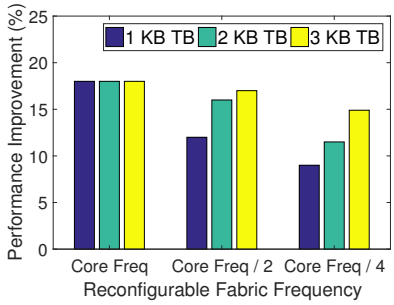

Figure 5: Performance overhead versus hardware features

Binary Search (BS): We considered an implementation of Binary Search over an input array of 1000 integers, where we monitor the invariants at each program iteration, together with some "hygiene" assertions regarding the binary search tree property and array boundaries. Figure 6 shows the area cost and performance overhead of the different applications under consideration. The overall runtime of the application under each hardware-software combination of monitors normalized to the runtime of the software-only implementation is shown with the line associated with each bar. The overall runtime of the application captures the delay arising due to stalls as a result of the trace buffer being full, the reconfiguration cost, and the performance overheads due to implementing some monitors in software. We observe an improvement of 32.5% when the area for monitors is restricted to 700 slices as compared with the software-only implementation.

AVL Tree (AVL): We implemented an AVL tree that supports insertion and deletion of duplicate values. We also implemented a pre-order traversal over the tree as shown in the listing in Section 3. This implementation has some assertions that are common to all the three operations, and some assertions that are specific to each function. This presents a case where the set of assertions that are live at any given point in time depends on the control path. Moreover, since these assertions have to maintain some state across function calls, all the monitors do not fit together on the reconfigurable fabric. Thus, based on the path taken, we have to reconfigure certain monitors to restrict the overall area overhead to the specified limits. We observe that two out of five assertions can be implemented in the reconfigurable fabric when up to 600 slices are available. However, when 700 slices are available, a different configuration of assertions is chosen, with 'A1' and 'A5' being partially reconfigured at runtime. The benefit of such dynamic reconfiguration is observed from the decrease in the overall runtime when going from 'AL2' to 'AL3'. The overall runtime overhead for the application reduces from 33.97% in case of all-software based monitors, to 18.74% under our proposed scheme.

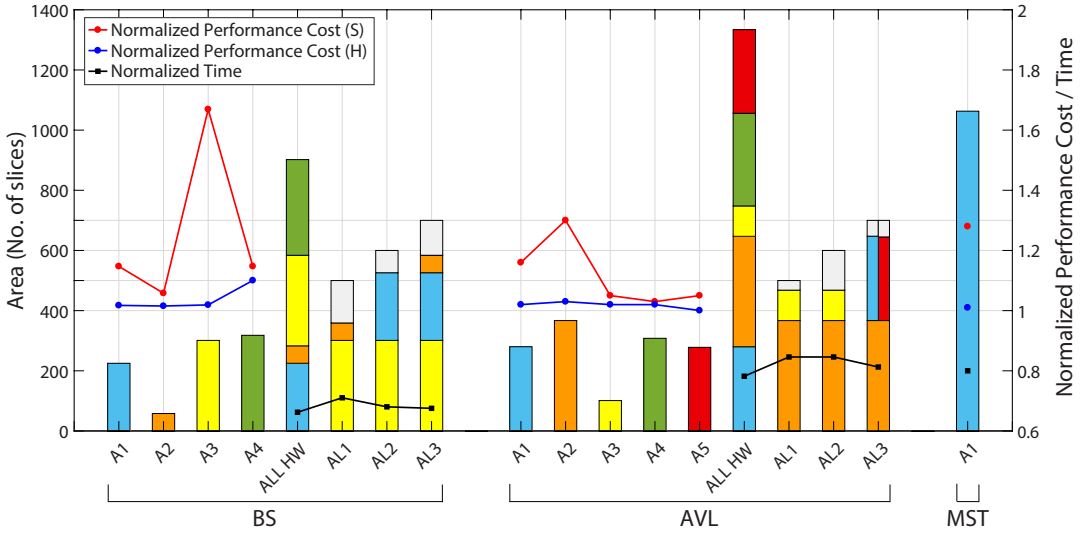

Figure 6: Impact of DHOOM architecture on area cost and performance overhead (normalized to base system with no monitors). The bars marked 'A1' to 'A5' are the number of slices required to implement the monitor in the reconfigurable fabric. The performance cost of each monitor in both hardware and software is shown with lines above the respective bars. The bars 'ALLHW', 'AL1', 'AL2', and 'AL3' show the area cost when all the monitors are implemented in hardware, and the subset of assertions implemented on the reconfigurable fabric when 500, 600, and 700 slices are available on the reconfigurable fabric.

Minimum Spanning Tree (MST): We considered a program to compute the MST of an input graph with over 1000 nodes using Prim's algorithm, monitoring only a single assertion that verifies that the input graph is connected. The monitor associated with this check is implemented on the reconfigurable fabric, and can proceed entirely in parallel to the computation of the MST in software. In this case, we observe no performance overhead of runtime verification because the monitor is implemented in hardware. The same monitor, if implemented in software, incurs an overhead of 19.5%.

6 CONCLUSION

We proposed and implemented a novel architectural framework that supports runtime verification of software while incurring minimal performance overhead under a given reconfigurable fabric area constraint. We presented a mapping algorithm that selects a subset of assertions to be implemented in hardware. We demonstrated the flexibility and benefits of our proposed scheme through four case studies that are representative of real-world applications and libraries. Our experiments reveal that the runtime overhead incurred due to runtime verification can reduce significantly under our proposed flow of selectively and judiciously moving assertions into hardware without exceeding the specified area constraints. In the future we intend to explore generalizations to monitoring multi-threaded workloads, context switching, and using more expressive logics.

REFERENCES

[1] R. Backasch, C. Hochberger, A. Weiss, M. Leucker, and R. Lasslop. 2013. Runtime Verification for Multicore SoC with High-quality Trace Data. *ACM TODAES* 18, 2 (2013).

[2] A. Basak, S. Bhunia, and S. Ray. 2016. Exploiting design-for-debug for flexible SoC security architecture. In *DAC*. ACM.

[3] S. Chandran, P. R. Panda, S. R. Sarangi, A. Bhattacharyya, D. Chauhan, and S. Kumar. 2017. Managing Trace Summaries to Minimize Stalls During Postsilicon Validation. *IEEE TVLSI* 25, 6 (2017).

[4] N. Decker, P. Gottschling, C. Hochberger, M. Leucker, T. Scheffel, M. Schmitz, and A. Weiss. 2017. Rapidly Adjustable Non-intrusive Online Monitoring for Multi-core Systems. In *SBMF*. Springer.

[5] D. Y Deng, D. Lo, G. Malysa, S. Schneider, and G E. Suh. 2010. Flexible and efficient instruction-grained run-time monitoring using on-chip reconfigurable fabric. In *MICRO*. IEEE.

[6] F. Farahmandi, R. Morad, A. Ziv, Z. Nevo, and P. Mishra. 2017. Cost-effective analysis of post-silicon functional coverage events. In *DATE*. IEEE.

[7] P. Fogarty, C. MacNamee, and D. Heffernan. 2013. On-chip support for software verification and debug in multi-core embedded systems. *IET Software* 7, 1 (2013).

[8] J. R Hauser and J. Wawrzynek. 1997. Garp: A MIPS processor with a reconfigurable coprocessor. In *IEEE FCCM*.

[9] S. Jakšić, E. Bartocci, R. Grosu, R. Kloibhofer, T. Nguyen, and D. Ničković. 2015. From signal temporal logic to FPGA monitors. In *MEMOCODE*. IEEE.

[10] N. Jindal, P. R. Panda, and S. R Sarangi. 2018. Reusing Trace Buffers as Victim Caches. *IEEE TVLSI* 26, 9 (2018).

[11] H. F. Ko and N. Nicolici. 2010. Automated trace signals selection using the RTL descriptions. In *ITC*. IEEE.

[12] C. Lai, Y. Yang, and I. Huang. 2014. A Versatile Data Cache for Trace Buffer Support. *IEEE TCSI* 61, 11 (2014).

[13] H. Lu and A. Forin. 2008. Automatic processor customization for zero-overhead online software verification. *IEEE TVLSI* 16, 10 (2008).

[14] A. Nassar, F. J Kurdahi, and W. Elsharkasy. 2015. NUVA: architectural support for runtime verification of parametric specifications over multicores. In *CASES*. IEEE.

[15] W. Shi, H. S Lee, L. Falk, and M. Ghosh. 2006. An integrated framework for dependable and revivable architectures using multicore processors. *ACM SIGARCH CA News* 34, 2.

[16] D. Solet, J. Béchennec, M. Briday, S. Faucou, and S. Pillement. 2016. Hardware runtime verification of embedded software in SoPC. In *SIES*. IEEE.

[17] G. Stitt, B. Grattan, J. Villarreal, and F. Vahid. 2002. Using on-chip configurable logic to reduce embedded system software energy. In *FCCM*. IEEE.

[18] E. Vlachos, M. L Goodstein, M. A Kozuch, S. Chen, B. Falsafi, P. B Gibbons, and T. C Mowry. 2010. ParaLog: Enabling and accelerating online parallel monitoring of multithreaded applications. *ACM SIGARCH CA News* 38, 1.