Semi-Formal Cycle-Accurate Temporal Execution Traces Reconstruction*

Rehab Massoud[1], Jannis Stoppe[1,2], Daniel Große[1,2], and Rolf Drechsler[1,2]

[1] Group of Computer Architecture, University of Bremen, 28359 Bremen, Germany
[2] Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
{massoud,jstoppe,grosse,drechsle}@cs.uni-bremen.de

Abstract. Today's Real-Time Systems' (RTSs) increasing speed and complexity make debugging of timing related faults one of the most challenging engineering tasks. Debugging starts with capturing the fault symptoms, which requires continuous cycle-accurate execution traces. However, due to limitations of on-chip buffers' area and output ports' throughput, these cannot be obtained easily.
This paper introduces an approach that divides the tracing into two tasks, monitoring on-chip execution to retrieve accurate timing information and high level functional simulation to retrieve signal contents. A semi-formal cycle-accurate reconstruction method uses these two sources to retrieve a complete, cycle-accurate trace of a given signal. An experiment illustrates how this method allows the cycle-accurate reconstruction of on-chip traces of a Real-Time Autonomous-Guided-Vehicle software.

1 Introduction

Locating errors is a crucial part of the Systems-on-Chip (SoC) development process. In order to be able to pinpoint bugs in the design, sophisticated logging and monitoring techniques are used. Usually, designers have to decide between: 1) much information from potentially slow simulations, 2) formal approaches that often limit the model's timing (if considered it at all) to a given upper accuracy and/or duration bounds or 3) limited data from on-chip runs.

Simulation-based techniques may be used to analyze a given system as soon as there is an executable prototype down to the end of the development process. While simulators supposedly provide an exact model of the given design, they inherently only offer 1) an abstraction of the real fabricated final hardware and 2) a fraction of the performance of it (running on general purpose host systems, a full accurate simulation of a single input-output-combination takes much longer than on the final SoC). While this is not much of an issue for a wide variety of use cases, it is for the location of timing-related errors in today's SoC's. Correct SoC's timing simulations require more details; hence excessive computations

* This work was supported by the University of Bremen's graduate school SyDe funded by the German Excellence Initiative, the German Federal Ministry of Education and Research (BMBF) within the project 01IW16001 (SELFIE), the German Research Foundation (DFG) within the Reinhart Koselleck project DR 287/23-1 and the German Academic Exchange Service (DAAD).

causing prohibitive slow-down in the simulation performance. Simulations can count only for those *apriori* known and modeled effects, so they can not cover all possible sporadic executions of the actual system.

Model-based approaches utilize functional hardware models that are e.g. provided to the software developers for functional testing. These models are based on hardware specifications like an instruction set architecture (ISA) [16]. Approaches such as worst case execution time (WCET) analyses [21] and abstract interpretations [19] are using this concept to give some guarantees about the behavior of the software when it operates on hardware. However, the more reliable and formal these methods, the more computation they require to account for every possibility and aspect in reality. The multitude of environment effects and variations of input/output interactions makes these model-based verification techniques very challenging – if not downright impossible. The execution on the fabricated hardware can still differ from its model-specifications due to possible unexpected (and hence non-modeled) process variations or other environmental operating conditions.

On-chip debugging requires stopping the system to get a scan-out of the current chip registers or state. Traditionally, scan-chains, Multi-Input Shift Registers (MISR) and Test Access Points (TAP) are used for post silicon validation [11], whereas specialized trace buffers and debug support units are mainly used in embedded processors [5]. This run/stop approach is inherently unsuitable for temporal behavior debugging, and requires many reruns until the root-cause is identified – which may result in it missing the sporadic behavior. To support continuous logging in embedded processors domain, current solutions are very customized (they use on-chip debugging modules and/or depend on compiler's generated meta data [2, 4]), that they cannot be extended to any SoC. Current on-chip techniques are often intrusive, i.e. they alter the temporal behavior itself, potentially affecting the timing that may be causing the error in the first place. Therefore, post-silicon timing aspects are usually addressed by different methods to avoid expensive continuous or time-accurate logging. These, however, focus only on capturing specified timing constraints violations and do not provide further means to detect a violation's root-cause, as in [10] and [15].

Methods for determining which signals to log or monitor to accurately reflect the system state at a specific instance (enhancing logic visibility) have been investigated [18]. However logging such signals continuously on a temporal accurate base was not considered so far.

Assuming that relevant signals have been identified beforehand to provide the best coverage of possible root-causes, obtaining a temporally accurate access to their evolution over time is still limited by factors such as the trace-buffers' area (if they would be stored on-chip) or the output ports' capabilities (when they are to be logged on-line). For SoCs in general, on-chip area can not accommodate continuous (theoretically) infinite traces; and on-chip signals/transactions speeds are orders of magnitudes greater than current logging ports capabilities [20].

Each of these techniques thus has its specific but severe issues for spotting timing-related errors. To address these shortcomings, this work shifts the focus from full-scale on-chip tracing to only log the temporal behavior accurately, omitting the functional content, which is provided via an off-chip functional simulation. To realize the reduced on-chip logging functionality, the idea of signature-summaries previously used in [8] and [14] is reformulated and generalized to be applied continuously to any on-chip traced signal. This altered usage of signatures is introduced as the continuous logging of *"footprints"* to denote their light weight and periodic nature. Non-temporal information of the erroneous run is obtained via traces from running a high level functional simulation of the specific scenario. While the logged simulation data lacks precise timing information (due to its potential high-level nature, which may sacrifice timing accuracy to improve the performance), it provides significantly more detail concerning the order and changes in value of the traced signal. This data (logged temporal execution footprints containing timing information and detailed off-chip simulation logs) is combined and used to reconstruct the accurate on-chip behavior.

The contributions of this work are:

1. a novel yet simple consistent methodology for continuous accurate temporal execution tracing and
2. a semi-formal offline Cycle-Accurate Temporal Reconstruction Algorithm (CATRA).

A proof-of-concept implementation for efficiently logging footprints from a running LEON3 processor[3], using functional Transaction Level Model (TLM) simulation traces from the SoCRocket simulator[17], is provided to illustrate how the approach may be applied and used to capture sporadic timing related bugs.

2 Methodology

The core goal of retrieving cycle accurate traces of on-chip temporal behavior drives the ideas and design decisions that are taken for the presented approach. First, an overview of the approach is presented that explains both, the methodology itself and the structures of the implementation. Two major parts of the given approach – the trace logging itself and the merging of on-chip and off-chip (simulation) traces – are discussed afterwards, providing the details of the approach.

2.1 Overview

As sporadic timing-related faults are hard to reproduce, precise information concerning the time *and* the data of erroneous transactions is required to enable the designer to identify the cause of the problem. In order to provide both, the task is split in two parts:

1. Logging precise timing information of the chip's behavior, i.e. storing information concerning *when* something happened and
2. logging the behavioral information itself, i.e. storing information concerning *what* happened.

The first part is needed to properly capture the temporal on-chip behavior, and is required to avoid altering the timing and thus changing what is – by definition – part of the cause of this *timing*-related fault. Thus, this part is explicitly logged from the traced on-chip execution. On the other hand, having an already functioning system removes the burden of logging the exact state *or signal* value itself every clock cycle, so only a data-parity-check is logged.

The second part – the data itself – is calculated off-chip in a functional simulation. Correct abstract functionality is enough to simulate the transitions of states – or signals values changes – irrespective of their timing, which depends on architectural and environmental particularities. In practice, SystemC Transaction Level Modeling (TLM) models are executed to calculate the behavioral data of the design. SystemC itself is a C++ library that allows designing hardware systems using high level language constructs, sacrificing synthesizeability for the sake of being able to quickly develop prototypes, with the TLM additions providing improved simulation performance at the cost of reduced timing accuracy. Notice that while the given example relies on SystemC, any functional simulation framework providing the required data may be used.

These two sources of information are then mapped onto each other, providing designers with a comprehensive continuous capture of the system's behavior. While the hardware is executed, the temporal behavior information (first part) is logged continuously. When a fault becomes visible, the scenario that was run on the hardware and lead to the faulty behavior, is used to start a functional simulation to provide the basic data, of which its temporal behavior was logged from the hardware execution.

2.2 Definitions

A trace τ is defined as a consecutive traced values of a signal over time. Hence, a trace can be represented by an ordered vector $\tau^{u,l} = \{\rho_0, \rho_1, ..., \rho_N\}$ if for the duration l, N different values were traced. Traced values are samples of the signal's continuous value, sampled every clock-cycle. A trace is either timed $\tau^{t,l}$, (it contains a value for every time instance), or un-timed $\tau^{u,l}$ (it only contains the consecutive ordered different values, appearing after each other). An infinite or continuous execution trace is denoted without a period l, i.e. τ^t or τ^u. When a trace is timed, elements of the trace ordered vector $\tau^{t,l}$, namely $\{\rho_{t_0}, \rho_{t_1}, ..., \rho_{t_{l-1}}\}$, represent the value ρ of a signal S at times $t_0, t_1, ..., t_{l-1}$. Due to the time being discrete and the system running on an internal clock, we can state that $t_i = t_{i-1}+1$. Thus, if a value ρ_x remained for two clock-cycles, starting from t_i, then two consecutive values ρ_{t_i} and $\rho_{t_{i+1}}$ would be equal. On the other hand if the trace is un-timed, one value ρ_i which corresponds to both ρ_{t_i} and $\rho_{t_{i+1}}$ is added to the trace ordered vector τ^u.

Traces could be obtained by continuously logging the values of on-chip signal. Such a complete trace is called *Actual Timed Trace* τ_a^t and represents the ideal goal. For cycle accurate tracing, such actual timed trace τ_a^t needs to be either stored on-chip or logged off-chip, the former being not possible due to limited on-chip storage, the latter due to the limited bandwidth of available ports. An alternative is to obtain the information from the simulation, with such a result being called *Simulation Trace* τ_s, or recovered from logged footprints, called *Reconstructed Trace* τ_r. Each of those traces could be either timed or un-timed.

2.3 Footprints Logging

In this work, we choose to generate and then log the temporal footprints periodically. The actual on-chip trace τ_a^t of the signal S is first divided into equal M long *Trace-cycles* $\tau_{r_i}^{t_M}$, where i is the trace-cycle number. M's actual value is a matter of the designer's preferences. It is a trade-off between the time required to decode the information, as shall be seen later, the logging bandwidth being used and the required on-chip storage. The logging is then limited to three distinct types of information:

- The timing information is encoded using periodic signatures. Each clock-cycle within the trace-cycle T_a^t is marked with a unique *time-stamp* TS_n, where $0 \leqslant n < M$ indicates the clock cycle within a trace-cycle. For the given implementation, w_{TS} bits (denoting bit-width of the time-stamps) are used to encode each clock-cycle within the trace-cycle. The traced signal S is sampled/monitored in a clock-cycle accurate basis. The old value of the signal is kept in a register, and is compared to the current value of S, raising a *Temporal Check* TC when it detects a change, as shown in Fig. 1. Time-stamps marking the cycles at which the given signal changes are aggregated (in the suggested implementation using XOR operations) into a single $Trace-Cycle$'s signature called *Temporal Cyclic Footprint* TCF. In Fig. 1, TC can be seen to invoke the aggregation of time-stamps TS 2, TS 6, TS 8, TS 13, and TS 14 (when the traced signal changes its value) to generate a TCF. Only this generated signature is logged to express change instances. To reduce the amount of data of this TCF to a size that fits through any potential bottlenecks, the time-stamp bit width can be reduced as desired – at the cost of potentially creating ambiguous footprints.
- A similar technique is applied for the considered signal itself: Each change in the observed signal's value at any cycle during the interval, contributes to creating a signature from the signal, called *Functional-Check (FC)*. (In the suggested implementation a simple parity check of the consecutive signal values during the $Trace-Cycle$ is used, also in the form of XOR of these values). This functional check is later used to match the un-timed simulation and actual traces.
- Finally, the number of signal changes N is stored and transferred as well.

This data (i.e. the timing and data signatures, and how many signal toggles occurred within the trace-cycle) is encompassed in a structure, that is logged and transferred to the host computer each trace-cycle. This set is called a *footprint* FP_s=$<s.FC, s.TCF, s.N>$ of signal s, describing how the signal s leaves a series of these distinct traits that are unique to the events that happened (or a set of possible events that could have happened) but do not represent the information

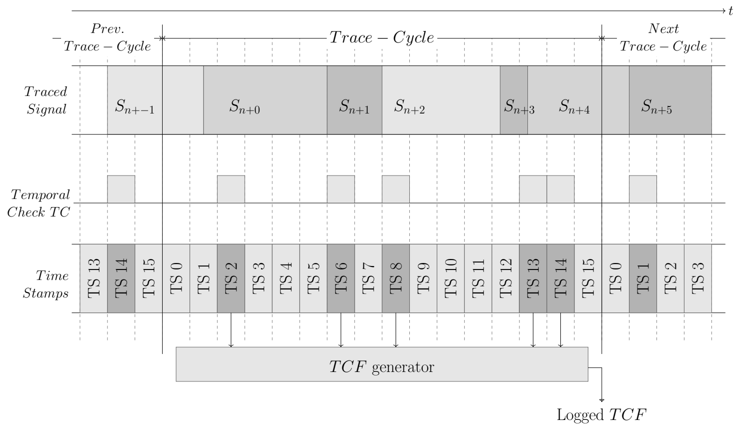

Fig. 1. Time-stamps involved in generating the Temporal Cyclic Footprint TCF

itself. For a 16-clock-cycles long trace-cycle, a 16 bits-wide footprint gives exactly one solution, which is equivalent to logging one bit every clock cycle. Such footprint of width $w_{TCF} = M$ enables the full recovery of the trace temporal check TC (and thus the times at which the signal was altered), irrespective of the number of changes N. To reduce the required bandwidth, the time-stamps' width w_{TCF} is reduced, and N is used afterwards to narrow the possibilities down. These footprints do not contain any explicit information about the behavior. However, the missing information is generated using a high-level functional simulation.

2.4 Functional Simulation

Techniques such as TLM allow designers to run simulations that sacrifice accurate timing information to gain performance. The assumption is thus that the semantics of the functional simulation are identical to the chip's behavior but the timing may be inaccurate and that the simulation can be executed when the on-chip execution reports an error that needs to be investigated. The data that can be retrieved from the simulation thus complements the footprints, which provide the timing information that the simulation's trace is lacking.

The functional simulation is executed in a controlled environment, so a signal's simulated functional values (constituting an un-timed functional simulation trace τ_f^u) can easily be generated and stored on a host system. Although the simulation is conducted on higher granularity and might differ in some details, it still provides a baseline from which the actual timed trace τ_a^t can be reconstructed. In literature, the different flow possibilities of interrupts and threads of executions of the simulated scenarios can be obtained via methods like [13]. When there is a set of known flow possibilities that could be short-listed for matching, the process becomes easier as shall be seen in the experiments section. In general, the complexity of such dynamic behavior matching was addressed in [9].

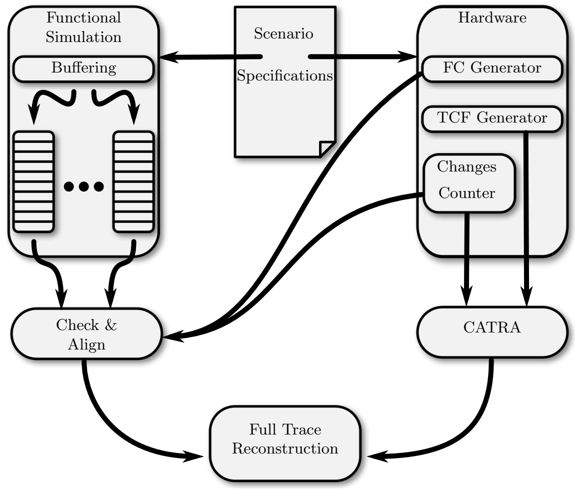

Fig. 2. Methodology for Footprints Logging and Traces Reconstruction

The process of retrieving complete traces is illustrated in Fig. 2. For the functional part (left-hand side of the figure), basic scenario specifications (such as e.g. inputs with their schedule, a software image and the set of interrupts to be fed to the system with their periodicity and/or estimated/planned occurrence instances) are needed to execute the functional simulation. From such simulation, the monitored signal values are also stored and then buffered: i.e. repetitions are eliminated to obtain un-timed functional traces (as stated in the definitions section); also the basic trace segments are identified. Segments are those groups of values of the trace known to be consecutive even if other segments came in between. Extracting the trace segments can be done with different granularity levels, in our experiment for example as the SoCRocket simulation already supports interrupts injection, we considered the whole main program as a single segment as obtained from simulation and interrupts service routines each as a segment. These two operations (eliminating repetitions and extracting segments) are called buffering in the figure. Hence, a potential candidate un-timed trace τ_f^u -or group of traces as a result from composition of segments- is obtained.

Then, using N and FC obtained from the hardware (right-hand side of the figure), the un-timed simulation trace τ_s^u can be mapped to trace-cycles $T_{s_i}^u$, of N changes each. Comparing a trace-cycle's logged FC to the simulated N values' generated FC is a parity check of $\tau_{s_i}^u = \tau_{a_i}^u$; i.e. the simulation values matches the actual values, providing a safeguard for the assumption about the simulation's correctness. It can also help amending discrepancies between simulation and actual traces if they existed; but only when the difference's root-cause can be speculated (i.e. correcting functional simulation trace/scenario to match the reality from a set of possibilities that can be tried by the designer until the logged FC matches the values checked in the simulation trace).

After applying this mapping, we have an *M-Cycles-Accurate* reconstructed trace $\tau_r^{t_M}$, where t_M denotes that the timing accuracy is within M clock cycles. This mapping does not need to be done separately for every trace-cycle. Instead, the number of changes N can be added all along the execution, until reaching the suspected trace-cycles. The simulation of long executions can also be projected into repeating periodic patterns. Still, obtaining the exact change clock-cycles, for a complete cycle accurate reconstructed trace is not trivial.

2.5 Cycle Accurate Trace Reconstruction Algorithm (CATRA)

To reach single-cycle-accuracy, the on-chip timing information of particular trace-cycles is reconstructed from the collected timing-part of the footprint TCF. This is done only for trace-cycles that are suspected to be of special interest and require cycle-accurate (i.e. timed) reconstructed trace-cycles $\tau_r^{t,M}$. This allows designers to pick any arbitrary trace cycle to inspect without having to process the whole execution log to get the exact cycle accurate data of a particular part.

In the Trace-Cycle mapping, it is possible that discrepancies in the values could go undetected if the suggested parity-check based functional check FC cannot detect it. For example, if the footprint was generated from the signal S from Fig.1 and the FC was generated using the suggested XOR-aggregation, as shown in Fig. 3 below, the two identical values S_{n+1} and S_{n+3} would cancel each other out. If $S^c_{n+1} = S^c_{n+3}$ on chip were *both* different from the simulation's $S^s_{n+1} = S^s_{n+3}$, the footprint's FC would not indicate any problem.

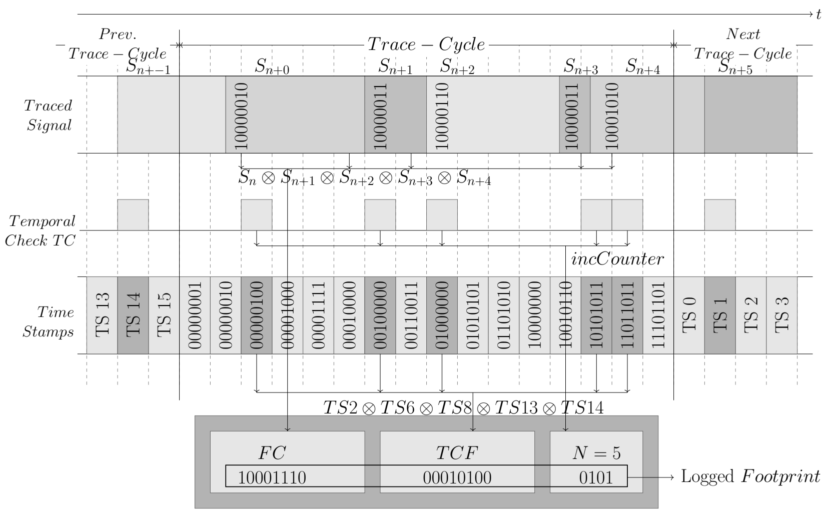

Fig. 3. Example of Footprint Generation

Fig. 3 shows a set of time-stamps that are aggregated (here using XOR) into the temporal cyclic footprint parts TCF. If a given trace cycle should be analyzed, all possible combinations that could lead to this specific TCF are obtained, with the actual combination that was calculated on-chip being among them. The number of possibilities may be large, though: for the example, when time-stamps of width $w_{TS} = 8$ are used (as indicated in the Fig. 3), there are 256 possible combinations of TSs that could have led to this logged TCF. Of these 256, only five[3] combinations contain 5 changes, which is the number N in our trace-cycle. In this case, the exact cycle accurate reconstruction is one of these five combinations. Notice that having more than one possible result makes determining which one exactly is what happened on hardware probabilistic (as the designer can assign probabilities to the obtained solutions).

Reconstruction using Formal Methods. The footprints contain a set of constraints describing characteristics of on-chip execution. From them, searching for solutions using established formal methods to deduce the actual on-chip trace is a viable approach. The TCF is created by merging all relevant time step signatures into a single footprint; now this process needs to be reversed. In order to quickly retrieve all possible combinations of time steps that result in a given footprint, the relation between footprint and time steps is formulated as a problem for (established) satisfiability solvers. The reconstruction of M cycles from an w-bits-wide signature (footprint) can be formulated as a simple Satisfiability Modulo Theory (SMT) problem as shown in Algorithm 1. The algorithm first initializes the value of the footprint TCF_0 to ρ_0 (which is user defined in reset -for the first trace-cycle- and the previously logged footprint afterwards) in line 1. It then builds a set of M consecutive `if-then-else` (*ite*) statements to be given to the solver in lines 3 and 4 that instruct the solver how to build the footprint: if the i^{th} bit in the Temporal Check $TC[i]$ indicates a change, the corresponding time-stamp TS is XORed. The solver is then constrained to finding a solution that matches one that has been retrieved from the hardware ($loggedTCF$) in line 6, thus giving a possible solution to when the signal was altered in line 7.

The SMT solver Boolector[7] was used to solve Algorithm 1, reconstructing TC in times shown in Table 1. In the case of smaller time-stamps bit-width w_{TS}, Algorithm 1 is used incrementally. In accordance to the number of possible solutions, the amount of time needed to compute all possible reconstructions of TC grows exponentially, which can be seen in the columns of Table 1, with different w_{TS}.

Improving Results using Available Information. To improve the scalability, the fact that the number of solutions can be reduced by N (which is the

3 in the original published paper only "one" combination was reported –instead of 5 here– because of a bug in our counting script; which we discovered only when we compared the number of solutions to the one we get with our new SAT reduction, presented in our next paper. It is worth mentioning that we can have less combinations if we used different set of timestamps. For example, using the set of timestamps used in our next paper, only 3 combinations contain 5 changes.

Algorithm 1: TC Reconstruction from Temporal Cyclic Footprints

```
Data: ρ0, TS, loggedTCF
TCF_0 = ρ0
bitvector[M] TC /* where M is the width of the bit vector variable
    TC                                                            */
foreach i in 1 ⟶ M do
    TCF_i = ite(TC[i], TCF_{i−1} ⊕ TS_i, TCF_{i−1})
    /* where TC[i] is the ith bit of TC                           */
end
AddConstraint(TCF_M = loggedTCF)
Solve_SAT ⇒ TC
```

Table 1. Average run-time in Seconds of Alg.1 for different M and $w = w_{TCF}$

M	w=M	w=4	w=8	w=16	w=32
8	Direct mapping	0.02	-	-	-
16	Direct mapping	0.3	1.9	-	-
32	Direct mapping	1.05	1.7	13.9	-
512	Direct mapping	-	-	-	3576

number of changes in the given trace-cycle) can be utilized to exclude all solutions containing number of changes that does not equal N during the solving process itself. N is required to map the functional trace vector's elements to the trace-cycles (and thus is logged anyway), so utilizing it to improve reconstruction performance does not cause any additional overhead. Excluding the solutions obtained by Algorithm 1 that do not match the given amount of changes N reduces the number of possible solutions but not the time required to obtain them. So in Algorithm 2 below, N is used as input to the solver.

Algorithm 2 uses N to reduce the amount of possible solutions *and* the time required to obtain them as follows. The algorithm relies on solving for a list of N indices, each indicating the time (inside the trace cycle) where a change occurred instead of a list of bits TC, where each indicates whether a change happened at the given index or not. Table 2 shows the average run times of the modified algorithm. Reductions in computation time are significant if few changes occur within a trace-cycle. It still needs to be applied iteratively to locate all possible (ambiguous) solutions. This algorithm relies on a list of indices, stored in the *change_index* bitvector that is declared in line 2. This set references the timestamps that should be used to calculate the resulting footprint. Table 2 shows the average run times of Algorithm 2 for different N and M. The reduction in computation time by algorithm 2 is remarkably significant in the two extreme cases: where there are very few and (as explained next) too much changes in a trace-cycle.

For signals that change frequently, the logged footprint may be first XORed with an all-time-stamps-XOR value, hence resulting in a new footprint that carries only the XOR of the remaining instances that were not XORed in the logged

Algorithm 2: Bounded to N Changes (No-changes) Trace Reconstruction

Data: ρ_0, TS, N
$FP_0 = \rho_0$
$bitvector[N][\lceil log(M) \rceil]$ $change\_index$
$AddConstraint(TCF_N = loggedTCF)$
foreach j in $1 \longrightarrow 2^{M-w_{TCf}}$ **do**
  `/* where` M `is the trace length after which we log the footprint; and` w_{TCf} `is the footprint's bit-width */`
  foreach i in $1 \longrightarrow N$ **do**
    $TCF_i = TCF_{i-1} \oplus TS[change\_index[i]]$ `/*` $change\_index[i]$ `is index of the clock-cycle in which the` i^{th} `change happened */`
  end
  $Solve\_SAT \Rightarrow change\_index_j$
  if $UNSAT$ break
  AddConstraint($change\_index \neq change\_index_j$)
end

footprint; then the algorithm is used to locate those $M - N$ time-stamps that indicate the instances of no change. This reduces the reconstruction complexity for larger N, allowing the algorithm to have an upper worst case for the reconstruction algorithm, which is $N = \frac{M}{2}$. So the algorithm shall be reconstructing either (changing or stagnating) change instances.

Table 2. Algorithm bounded by N, average run-time in m minutes and s seconds, for different trace lengths M and number of changes N. For N=1, it's just a direct mapping, i.e. the TCF is the single change's time-stamp.

$M \setminus N$	1	2	3	4	5	6
8	0	~ 0	~ 0	~ 0	~ 0	~ 0
16	0	~ 0	0.1s	0.2s	0.3s	0.4s
32	0	~ 0	0.5s	1.6s	2.1s	5.4s
512	0	1m16s	7m10s	43m65s	-	-
1024	0	6m42s	37m46s	-	-	-

This bounded by N-changes algorithm can result in only one solution if the time-stamps are designed to provide unique TCF for each different combination of N aggregated time-stamps. As the time-stamps are set prior to the execution, they may be generated to specifically satisfy this criterion.This is particularly useful, if a given N is assumed to be problematic. For only one change ($N = 1$), the uniqueness of time-stamps is enough for cycle-accurate trace reconstruction; as a logged time-stamp can then be directly mapped to its respective instance.

For $N = 2$ and using XOR gates to merge the time-stamps, the condition is:

$$\begin{aligned} \forall i, j, k, l, &[TS_i \neq TS_j] \\ &\cap [TS_i \oplus TS_j \neq TS_k \oplus TS_l], \end{aligned} \tag{1}$$

where

$$\begin{array}{rcl} & (0 < i, j, k, l \leqslant M) \cap i \neq j \cap k \neq l & \\ \cap(i = k \Rightarrow j \neq l) & \cap & (j = l \Rightarrow i \neq k) \\ \cap(i = l \Rightarrow j \neq k) & \cap & (j = k \Rightarrow i \neq l) \end{array}$$

Similar conditions can be derived for higher N.

In summary, using a combination of on-chip traced footprints, off-chip functional simulation data, and the reconstruction and mapping of this information, a cycle accurate reconstruction of on-chip behavior is possible. The next section illustrates the applicability of the method in practice, showing how the reduction in the amount of logged data allows the approach to be used in continuous logging. This in turn allows the designer to efficiently capture timing related sporadic faults and assists in finding their root-causes.

3 Experiments

As a case study, the presented methodology was used to continuously capture the temporal behavior of a toy software, which contains an integrated safe-zone calculation module for mobile robots from the SAMS project[4]. In the given design, the current angle of the moving robot is updated via an interrupt service routine (ISR), which checks for differences from the previous value $\Delta\theta$ as shown in Fig. 4. If the difference is below an accepted limit θ_1, it continues the previously executed task. Otherwise it restarts the safety zone calculation algorithm with the new values if there is enough time to finish it before the deadline. If restarting the algorithm will result in missing the deadline, the ISR checks whether the difference is less than another value θ_2 where $\theta_2 > \theta_1$. If it is, it adds a margin to the current calculation. The value to be added depends on the time difference between the last two time readings. If the difference $\Delta\theta \geq \theta_2$, it activates a worst case algorithm with the updated values. The maximum interrupt rate is $100ms$. The generated software image was run on a LEON3 processor implemented using Xilinx zync7020 FPGA. The same software image was run on SoCRocket, a LEON3 SystemC TLM simulator to get the functional execution traces.

A trace-cycle with $M = 1023$ clock cycles, given the 83 MHz (12 $nSec$) input clock of the Zynq FPGA, would make 12.276 μSec duration of each trace-cycle; during which no interrupt can occur twice (the quickest is the timer interrupt with periodicity $10ms$). Including the watchdog interrupt, the maximum number of interrupts we can have in one trace-cycle is three corresponding to the 3

[4] www.sams-project.org, the module is certified for use in safety systems up to SIL-3 according to IEC EN 61508.

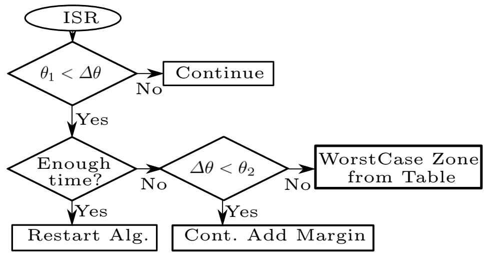

Fig. 4. Interrupt-service-routine

interrupts occurring at the same trace-cycle. The signals chosen for tracing were the program counter (PC, w_s=32 bits) and two interrupts lines ($IRQ\_l_{1,2}$, w_s=2 bits). One of those 2 bits of IRQ_l is our ISR's IRQ line and the other is timer interrupt line. So, here $IRQ\_l.FC$ is not only a check, but it also indicates which interrupts occur; for $IRQ\_l.N$, we took 3 interrupt lines as shall be seen later. At which clock cycle exactly an interrupt that has occurred starts to be served is not usually known because of the pipeline mechanics and interrupt masking (if used). In our case, there is no masking, no critical sections where interrupts are disabled and the different interrupts are allowed to be nested according to their priorities. So within a trace-cycle at which the interrupt line's footprint indicates an interrupt request, the exact instance of interrupt occurrence is obtained using CATRA and the exact instance where the interrupt starts executing lies within the maximum detailed architectural delay (cache miss, pipeline, interrupt priority ...etc).

A hardware module (implementing the hardware-box in Fig. 2 containing the generators and counter) was implemented on-chip to generate and log FC, TCF and N for both PC and IRQ separately every trace-cycle (for $M = 511$ and $M = 1023$ as in Table. 3 below). Our implementation does not cause any system slow-down, as we used continuous EXORing with previous FC and TCF. So the changes at the borders between trace-cycles do not require any special handling.

Additionally, the values of those two signals (PC and IRQ_l) are logged during the SoCRocket simulation and buffered to eliminate consecutive similar values. SoCRocket enables injecting interrupts via timers and given certain delay from the start time. The exact time when interrupts occurred can be obtained from applying CATRA to the interrupt line footprint. Still during simulation, the actual time in which the interrupt occurs may be not exactly the time the interrupt was fired in the simulation (because the model is not cycle accurate).

The direct way to map changes to their respective trace-cycles is to start from reset where the initial values of both simulation and hardware are similar. Each trace-cycle, the logged $PC.N$ is used to pick N values from the simulation trace and assigning them to a trace-cycle. Then the generated FC for these values is compared to the logged footprint's FC as a check. It is possible to skip this step (when there is high confidence in the functional simulation results) and jump to the suspected trace-cycle ($K^{th} trace - cycle$, where more than one interrupt occur), get the sum of all previous changes ($N_{sum} = \Sigma_{i=0}^{k-1} N_i$) and then get the start of the traced signal from the simulated trace τ_s^u values as the N_{sum}^{th} value.

Within a specific trace-cycle, if there was an interrupt, how many among the N changes in the trace-cycle belong to the interrupt and how many belong to the interrupted segment are initially unknown. We start by assuming that the actual interrupt occurrence instance obtained from the logged IRQ_l footprint via CATRA is the exact instance in which the PC value has switched to the interrupt segment; then from the $PC.TCF$ via CATRA as well, we determine the first change instance appearing at or after actual interrupt occurrence instance. We then assume PC values before this instance belong to the interrupted segment, and from the instance on belong to the interrupt. Then we calculate the FC by EXORing these PC simulation trace values and check if it matches the logged footprint's FC. We increment the PC-switch-to-interrupt instance to the next possibility by considering one more PC value from the interrupted segment and one less from the interrupt. We repeat this to consider the range of possible maximum architectural delay. As a result, candidate traces that match the logged FC for further investigation are collected. If no FC matches were found, then the previous assumption leading to the start value obtained from the simulation is probably wrong, hence earlier trace-cycles are investigated.

Two scenarios in which we used the above mechanism to debug sporadic faults that did not appear consistently are shown in Table. 4. In both cases we started our analysis from the last trace-cycles that had more than one interrupt before the fault becomes visible. Then the above described flow was used to get when exactly (after which instruction) the interrupts were executed and arrived to the conclusions in Table. 4 about the faults' root-causes. [5]

Table 3. The number of bits logged every trace cycle, and the required bit-rate for logging, in the implementation $w_{FC} = 32 + 2$ and $w_{TCF} = 32$ for both PC and IRQ

Trace-cycle length	Naive logging	Required bit-rate	TC, FC (1 bit per clk)	Required bit-rate	TCF, FC, N (+$CATRA$)	Required bit-rate
M= 511	17374	2.92$Gbps$	1056	171.81$Mbps$	109	17.73$Mbps$
M= 1023	34782	2.92$Gbps$	2080	169.37$Mbps$	110	8.95$Mbps$

Using naive logging, $M * (w_s(PC) + w_s(IRQ\_l))$ bits are logged per trace-cycle, i.e. 34782 bits for $M = 1023$. Using the proposed logging scheme and CATRA, only: $w_{s.FC}(PC) \mid_{32} + w_{s.FC}(IRQ) \mid_2 + w_{s.TCF}(PC) \mid_{32}$

$+ w_{s.TCF}(IRQ) \mid_{32} + w_{s.N}(PC) \mid_{10} + w_{s.N}(IRQ)) \mid_2$ bits are logged, i.e. 110 bits for the same setup. Table. 3 shows the reduction in the required logging bit-rate. So, instead of logging the signal every clock cycle, a set of footprints

[5] Note that using interrupts to alter the execution is not recommended for safety critical software in general. However, it could be unavoidable to fulfill a hard requirement of responding to external changes instantaneously not via pulling.

are logged periodically. Using the proposed approach cycle-accurate details of the exact on-chip execution trace are captured.

Table 4. Scenarios that encountered sporadic faults and their symptoms in the second column, root-cause analysis and its computation effort in the 3rd and 4th columns

#	Scenario	Symptoms	Root-cause Analysis
1	$\theta_1 < \Delta\theta < \theta_2$ and ISR comes at an instance where there is barely enough time to restart the sams task to finish before its deadline.	sams task is restarted, but didn't finish before its deadline.	It was found that ISR interrupted the timer interrupt after it started execution, but before the exact instruction in which it updates the time value. So the ISR used the old time value thinking there is enough time to restart sams so it finishes.
2	$\theta_1 < \Delta\theta < \theta_2$ and ISR runs at its maximum rate, requesting a margin increase each time.	Wrong value of the safe-zone output.	The ISR interrupted the timer twice in row, making the margin calculations inside that interrupt routine being performed using older, non-updated values of the time.

4 Related Work

While formal design based approaches like Backspace [8] and Magellan [6] use the design itself, the presented approach instead relies on a simulated abstract functional execution trace. This hugely reduces the computational requirements and limits the tracing to specific trace-cycles. While other approaches that rely on higher level abstract functional matching may only start from the initial state (as in [6]) or the final state (as in [8]), the presented approach limits the matching process to short time frames (the trace-cycles) within the given traces. Periodic logging is used to check the on-chip computed signatures in [22], where the usage of parity-checks decrease the number of debugging sessions. However, requiring frequent rerunning, scan-chains and run-stop mechanism keeps such methods from detecting inconsistent faults. For circuits implemented on FPGAs, commercial tools like [1] rely on the continuous tracing of values at the operating frequency, which results in log-size issues. For microprocessors, manufacturers provide propriety solutions for temporal accurate logging [2, 4]. Their closed nature and reliance on compiler-generated meta-data means that conceptually, these approaches cannot be applied e.g. to ASICs. Recently, NuVA [12] verified high speed on-chip transactions, but in turn caused the overall chip performance to drop slightly. In contrast, the methodology proposed here does not affect the chip's performance, uses very simple logic and is applicable to any signal.

5 Conclusion and Future Work

Temporally accurate logging using today's methods is impractical, although it could be the shortest way for capturing and debugging post-silicon timing related bugs. We proposed a novel non-intrusive logging scheme and a reconstruction approach CATRA to provide accurate information about the on-chip execution. This allows for the first time, to capture and analyze timing-related sporadic errors.

We are currently developing methods for efficient times-tamps auto-generation for less solutions of CATRA under-specific conditions and shorter computation time. Also the computational complexity of using the functional-check part of the footprints in traces alignment is under analysis.

References

1. ChipScopePro (2017), `www.xilinx.com/products/design-tools/chipscopepro.html`
2. Embedded Trace Macrocell block specification, www.arm.com (2017)
3. Gaisler Research. http://www.gaisler.com (2017)
4. System Navigator Probe, http://www.mips.com (2017)
5. Abramovici, M., Bradley, P., Dwarakanath, K., Levin, P., Memmi, G., Miller, D.: A Reconfigurable Design-for-Debug Infrastructure for SoCs. In: DAC (2006)
6. Ahlschlager, C., Wilkins, D.: Using Magellan to Diagnose Post-Silicon Bugs. In: Synopsys Verification Avenue Technical Bulletin, vol.4, no.3, pp.15 (2004)
7. Brummayer, R., Biere, A.: Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In: Proc. Int'l Conf. Tools and Algorithms for the Construction and Analysis of Systems (2009)
8. De Paula, F.: Backspace: Formal Analysis for Post-Silicon Debug Traces. Phd. Thesis, University of British Colombia (2012)
9. Fredrikson, M., Christodorescu, M., Jha, S.: Dynamic Behavior Matching: A Complexity Analysis and New Approximation Algorithms. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) Automated Deduction – CADE-23 Proceedings (2011)
10. Hu, B., Huang, K., Chen, G., Knoll, A.: Evaluation of Run-time Monitoring Methods for Real-Time Events Streams. In: ASPDAC (2014)
11. Mitra, S., Seshia, S.A., Nicolici, N.: Post-Silicon Validation Opportunities, Challenges and Recent Advances. In: DAC (2010)
12. Nassar, A., Kurdahi, F.J., Elsharkasy, W.: NUVA: Architectural Support for Runtime Verication of Parametric Specications over Multicores. In: CASES (2015)
13. Nguyen, M.D., Wedler, M., Stoffel, D., Kunz, W.: Formal hardware/software co-verification by interval property checking with abstraction. In: (DAC) (June 2011)
14. Park, S., Mitra, S.: IFRA: Instruction Footprint Recording and Analysis for Post-Silicon Bug Localization in Processors. In: DAC (2008)
15. Reinbacher, T., Függer, M., Brauer, J.: Runtime Verification of Embedded Real-Time Systems. Formal Methods in System Design 44(3), 203–239 (Jun 2014)
16. Schmidt, B., Villarraga, C., Fehmel, T., Bormann, J., Wedler, M., Nguyen, M., Stoffel, D., Kunz, W.: A New Formal Verification Approach for Hardware-dependent Embedded System Software. IPSJ Transactions on System LSI Design Methodology (2013)
17. Schuster, T., Meyer, R., Buchty, R., Fossati, L., Berekovic, M.: SoCRocket-A virtual platform for the European Space Agencys SoC development. In: ReCoSoC, (2014), `availableathttp://github.com/socrocket`
18. Shojaei, H., Davoodi, A.: Trace Signal Selection to Enhance Timing and Logic Visibility in Post-Silicon Validation. In: ICCAD (2010)

19. Souyris, J., Pavec, E.L., Himbert, G., Borios, G., Jégu, V., Heckmann, R.: Computing the Worst Case Execution Time of an Avionics Program by Abstract Interpretation. In: 5th International Workshop on Worst-Case Execution Time Analysis (WCET) (2005)
20. Vermeulen, B., Goossens, K.: Debugging Systems-on-Chip. Springer, New York (2014)
21. Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., Stenström, P.: The Worst-case Execution-time Problem: Overview of Methods and Survey of Tools. ACM Trans. Embed. Comput. Syst. (2008)
22. Yang, J., Touba, N.: Enhancing Silicon Debug via Periodic Monitoring. In: Proc. of Symposium on Defect and Fault Tolerance (2008)