Automatic Detection of Concurrency Bugs through Event Ordering Constraints

Luis Gabriel Murillo, Simon Wawroschek, Jeronimo Castrillon, Rainer Leupers and Gerd Ascheid
Institute for Communication Technologies and Embedded Systems (ICE)
RWTH Aachen University, Germany
Email: {murillo,wawroschek,castrillon,leupers,ascheid}@ice.rwth-aachen.de

***Abstract*— Writing correct parallel software for modern multi-processor systems-on-chip (MPSoCs) is a complicated task. Programmers can rarely anticipate all possible external and internal interactions in complex concurrent systems. Concurrency bugs originating from races and improper synchronization are difficult to understand and reproduce. Furthermore, traditional debug and verification practices for embedded systems lack support to address this issue efficiently. For instance, programmers still need to step through several executions until finding a buggy state or analyze complex traces, which results in productivity losses. This paper proposes a new debug approach for MPSoCs that combines dynamic analysis and the benefits of virtual platforms. All in all, it *(i)* enables automatic exploration of SW behavior, *(ii)* identifies problematic concurrent interactions, *(iii)* provokes possibly erroneous executions and, ultimately, *(iv)* detects concurrency bugs. The approach is demonstrated on an industrial-strength virtual platform with a full Linux operating system and real-world parallel benchmarks.**

I. Introduction

Multi-processor systems-on-chip (MPSoCs) have spread rapidly to all electronics industry sectors due to power and thermal limitations of monolithic cores. Surveys state that already around 50% of new design projects in 2012 used two or more application processors [1], and this number will only increase in coming years. The complement of multi-processor hardware (HW) is efficient concurrent software (SW) so that available computing resources can be fully exploited.

Concurrent software is essential for achieving the desired system efficiency and provides a key product differentiator. However, with SW aspects already accounting for more than 50% of the total design costs [2], new products are subject to greater verification efforts and costs associated to concurrency-related issues. Writing correct parallel SW is difficult. Concurrency can lead to severe bugs when unintended interactions occur among system units. To avoid this, programmers have to take measures to keep all relevant concurrent actors in sync, but this is in general hard to realize. Human beings are not very good at understanding concurrent practical tasks [3] and even the most experienced programmers make mistakes when writing concurrent programs. Finding a concurrency-related bug is just as hard. Most bugs remain unnoticed and surface only after systems have been reliably running for months [4]. When they appear, they are extremely hard to reproduce. Similarly, fixing a concurrency bug is also cumbersome. In practical settings, many bug fixes released by programmers do not remove the target bug but just keep it hidden, whereas other fixes introduce more buggy behavior [5]. All this has led to a huge need for tools that facilitate concurrent SW development and debug for modern MPSoCs. In fact, the design of concurrent SW is considered as a long-term "grand challenge" for the electronics industry [2].

978-3-9815370-2-4/DATE14/©2014 EDAA

Being able to reproduce an exact buggy state in an application substantially improves the debugging experience [4]. Furthermore, the entire debug process can be accelerated by orders of magnitude when adding systematic exploratory techniques that search for and trigger bugs. This can be achieved through extensive monitoring and precise control of the execution of all individual SW tasks and components on the target so that on- or off-line algorithms can manipulate them at will. Since off-the-shelf hardware has restricted debug controllability and observability, Virtual Platforms (VPs) can play an important role as substitutes for debugging concurrency issues. By using them, it is possible to gain reproducibility and have full non-intrusive inspection and control of a system that closely mimics an MPSoC. Nevertheless, VPs provide only an execution vehicle and on top of them it is still necessary to deploy advanced concurrency monitoring and analysis.

Contributions. This paper introduces a new debug approach covering all necessary steps to successfully find concurrency bugs in MPSoCs, namely dynamic *monitoring*, *control*, *analysis* and *replay*. As a major achievement, it enables automatic exploration of applications so as to find a wrong interaction of system events leading to a buggy execution. Firstly, the approach includes a monitoring and control framework connected to a target MPSoC VP. The monitoring abstracts relevant multi-processing SW and HW interactions, and generates a *high-level event trace* with semantic information useful for analysis. Trace and concurrency analysis are also introduced that search for non-deterministic behavior and event permutations (or *interleavings*) potentially leading to a buggy execution. If a potential execution is found that is likely to change the application's behavior, it can be explored by explicitly forcing the program into the believed erroneous state using a controller. This strategy improves the likelihood to generate task interleavings with bugs that can occur during normal execution but usually remain hidden. With the platform monitoring, control and event analysis at hand, repeated program execution becomes much more viable than other approaches for finding bugs. Our debug approach, which is shown in Fig. 1 and whose components are explained throughout this paper, will be demonstrated with a quad-core ARM MPSoC with Linux built in a commercial VP framework.

Relation to Previous Work. Automatic concurrency-related bug detection for high-end applications has been a hot topic in the last decade. Dynamic analysis techniques and testing [4], [6], [7], compared to static code analysis [8] and model checking, usually serve to identify errors with higher accuracy and are more applicable to real systems. More recently, debug through symbolic execution [9] and hybrid static-dynamic

checking [10] have shown to be very effective to classify and find complex bugs (e.g., bugs involving concurrent accesses to multiple variables). However, no analysis framework has targeted specifically MPSoCs, and embedded developers still rely on manual breakpointing and stepping of software. Furthermore, most previous work relies either on instrumentation with compilers (e.g., on source code, binary or bytecode) or on modifications to parts of the target SW stack (e.g., a custom OS scheduler). In contrast, our approach builds upon non-intrusive monitoring of HW/SW events. Moreover, it achieves bug exploration through controlled behavioral replay at the lowest granularity level (i.e., event pairs). This is compatible with MPSoC programming where VP frameworks, such as Synopsys Virtualizer [11], help by providing full-system reproducibility, controllability and visibility. Besides the framework for interrupt-related bugs in [12], systematic or analytical ways to debug concurrency issues in MPSoCs are scarce.

Outline. This paper is organized as follows. Section II presents the problem formulation. Section III discusses concurrency analysis techniques that help to find non-deterministic behavior. This is complemented by our approach to unveil bugs through iterative behavior exploration in Section IV. Section V presents results of our debug approach when used with a commercial multi-core ARM VP running Linux. Results cover examples of applications used for benchmarking of parallel systems. Finally, Section VI concludes this work.

II. Event-based Debugging: Problem Formulation

A programmer may simply forget to insert synchronization artifacts (e.g., locks and semaphores) in critical sections when writing parallel code, such as for a multi-task OS. In consequence, bugs related to data races might occur, e.g., *(i)* when a certain code section must execute atomically with regard to other concurrent actors but this is not respected (i.e., an *atomicity violation*), or *(ii)* when the semantic order between two or more code sections is flipped (i.e., an *order violation*) [5], [10]. Finding the cause of these bugs is very problematic due to both their non-deterministic nature and their random effects. They often appear at some point in time without any reference to the originating instruction. In embedded systems, the variety of processing elements and communication mechanisms significantly increases the sources of concurrency issues as well as debug complexity. Programmers, coding from drivers to high-end applications, have to ensure proper interaction among concurrent SW tasks and a plethora of HW-dependent features (e.g., direct memory access, interrupts and mailbox devices). This demands for a mechanism to handle concurrency-related system events.

A. Debug Abstractions

We adopt the strategy of looking at an application as an event trace $T = \{e_1, ..., e_n\}$ which is a sequence of concurrency-related system events that reflects their *occurrence order.* This notation was first introduced by Lamport [13] to deal with causality in distributed systems. Later algorithms, such as [14], enable computing timestamps to discover exact partial order relations between event pairs. This notation provides an abstraction of the system which allows applying high-level debug algorithms. Extensions and new definitions are presented in this paper in order to adapt the formalism to events that can be non-intrusively obtained and controlled in a VP on the fly.

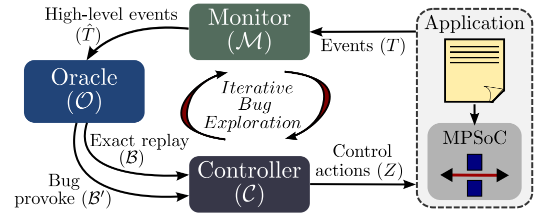

Fig. 1: Iterative bug exploration flow

Definition 1 (Atomic event)**.** An atomic event $e \in T$ is a 5-tuple (t, p, c, a, d), where t is a timestamp, p is the associated physical component, c is a logical execution context (e.g., SW task "X"), a is a concurrency-related action (e.g., task creation or shared memory access), and d represents action-specific data (e.g., a memory address or a lock ID).

For concurrency analysis, it is important to have events that *(i)* contain properties revealing their location and nature and *(ii)* represent atomic actions influencing the overall global program state. Furthermore, having order relationships between events in a trace is of utmost importance.

Definition 2 (Occurred before)**.** An *occurred before* relation, denoted as $e_i \prec e_j; e_i, e_j \in T$, exists *iff* during tracing e_i was observed to occur before e_j. The relation "$\prec$" is transitive.

Definition 3 (Happens before)**.** A *happens before* relation, denoted as $e_i \rightarrow e_j; e_i, e_j \in T$, exists *iff* e_i is considered to *always* occur before e_j in all *feasible interleavings* (i.e., interleavings that could be observed during tracing).

$(T, \prec)$ and $(T, \rightarrow)$ correspond to partially ordered sets (*posets*). The "$\prec$" relation contains observed events in the order that they were actually recalled during execution. The "$\rightarrow$" relation, if found for a trace, contains event orderings which are always enforced by *program order*, *synchronization* and/or *transitivity*. For program order, it is understood that two events e_i and e_j emitted sequentially within the same execution context c will never occur in swapped order. For synchronization, it is understood that either using inter-thread synchronization (e.g., *signal/wait*) or the creation/deletion of a new context c can enforce a specific order (e.g., an event e_j within a task will only happen after the creation, say e_i, of that same task). Finally, if $e_i \rightarrow e_j \wedge e_j \rightarrow e_m$ holds, then $e_i \rightarrow e_m$ follows by transitivity. Based on the "$\rightarrow$" relation, it is also possible to infer when two events are *concurrent* ("$\leftrightarrow$"):

$$e_i \leftrightarrow e_j \Leftrightarrow e_j \leftrightarrow e_i \Leftrightarrow e_i \not\rightarrow e_j \wedge e_j \not\rightarrow e_i$$

Definition 1 states that each event e maps to a precise *spatio-temporal location* through t and p. The *semantic information* (i.e., the triple (c, a, d)) of most concurrency-related actions, however, cannot be derived from a single instantaneous state change extracted at runtime. This is particularly true when the target is to be monitored non-intrusively, as is the case with VPs. The only events that can be non-intrusively traced are the very simple HW-level events which are observable with a traditional debugger (e.g., a machine instruction execution or a signal change). Under these circumstances, programmer-introduced multi-processing artifacts and OS decisions, such as a *semaphore lock* or a *task preemption*, are *high-level events* (HLE) that do not correspond to a single observable event but to a set of them (e.g., several *instruction execution* and *shared memory access* events in a certain order).

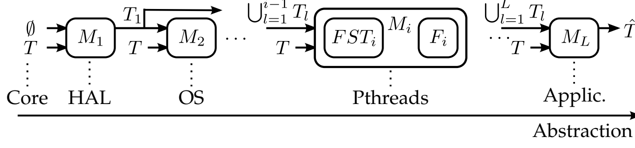

Fig. 2: Event monitoring

Definition 4 (HLE)**.** A *high-level event* is a composition of HW-level events $\hat{e} = (E \subset T, \hat{t}, \hat{p}, \hat{c}, \hat{a}, \hat{d})$ that can be non-intrusively observed in a target.

To give meaningful information, an analytical debugger must operate on traces of HLEs. We formulate this as:

Problem 1. Given a low-level trace $T = \{e_i\}$, obtain a high-level trace $\hat{T} = \{\hat{e}_j\}$ that captures spatio-temporal location and semantic information of T with *programmer-relevant* events.

This problem can be decomposed into, *(i)* **grouping:** How to group low-level events into *programmer-relevant* events, and *(ii)* **propagating:** How to propagate spatio-temporal location and semantic information to these HLEs (i.e., how to compute $\hat{t}, \hat{p}, \hat{c}, \hat{a}$ and $\hat{d}$). Notice that whether or not an event is relevant depends on the programmer abstraction (e.g., firmware or OS). Similarly, event semantic information can be propagated, enriched and generated at different HW/SW layers. For instance, several memory accesses and instructions executed at the HW level can become an OS-level event for creating a new task. We therefore define a layered approach to grouping events according to typical abstraction levels (e.g., HW, OS, middleware and application).

To solve Problem 1, the debugging framework has a layer-aware system monitor $\mathcal{M}$ connected to the target (see Fig. 1). The monitor is composed of modules, one for each HW/SW layer, that processes events as they come from the system, producing HLEs. Each module $M_i \in \mathcal{M}$ receives low-level events in T and HLEs from all previous layers (produced by $M_1, ..., M_{i-1}$), and generates a set T_i of HLEs that are relevant at layer i. Formally, a module can be seen as a pair $M_i = (FST_i, F_i)$, where FST_i is a finite state transducer, in charge of **grouping**, with input alphabet $\Sigma_i \subseteq T \cup (\bigcup_{l=1}^{i-1} T_l)$ and output $\Gamma_i \subset \mathcal{P}(T)$. Since HLEs can be nested (e.g., OS I/O function calling a synchronization primitive), all FSTs are provided with a stack. F_i is a function in charge of **propagating**, so that $\forall E \in \Gamma_i : F_i(E) = (\hat{t}, \hat{p}, \hat{c}, \hat{a}, \hat{d})$. After a sequence of modules for monitoring L layers, the desired high level trace is available for further processing in the bug exploration flow. It is defined by the HLEs produced by the last module (T_L), i.e., $\hat{T} = \{\hat{e} = (E, \hat{t}, \hat{p}, \hat{c}, \hat{a}, \hat{d}); E \in \Gamma_L \wedge (\hat{t}, \hat{p}, \hat{c}, \hat{a}, \hat{d}) = F_L(E)\}$. The logical connection of the monitors is shown in Fig. 2.

The internals of the FSTs and the F functions depend on implementation details of each corresponding HW/SW layer. They require system information, such as which event sequences represent task creation and which memory accesses affect OS data structures. The grouping mechanism (FST) is similar to how debug awareness layers work in source debuggers (e.g. the thread layer in GDB). It requests breakpoints and watchpoints on binary symbols and variables that are related to a certain HLE (e.g., for Linux, a call to function `copy_thread` marks thread creation). The propagating mechanism (F) is more involved. For a given group $E \subset T$, the location information is obtained from a so-called *anchor* event $e_k \in E$, i.e., $(\hat{t} = t_k, \hat{p} = p_k)$. For example, given a set of events representing the creation of a task, the anchor event corresponds to the last event: the return from `copy_thread`. The semantic information $(\hat{c}, \hat{a}, \hat{d})$ is generated based on the layer-specific knowledge and the information contained in the low-level events. For example, a sequence of events with a HW context c, a memory access a and related addresses d, can be transformed into an HLE with thread context $\hat{c}$, mutex acquire action $\hat{a}$ and mutex object ID $\hat{d}$.

A monitor, as described here, requires a retargetable and extensible debugger back-end connected to a VP. For instance, by using the debugger from [15], one can define a hierarchy of debug components to extract information from system layers.

For two HLEs $\hat{e}_i, \hat{e}_j \in \hat{T}$ with different anchor events $e_{k_i}, e_{k_j} \in T$, the "$\prec$" and "$\rightarrow$" relations can be defined on $\hat{T}$ as follows:

$$\hat{e}_i \prec \hat{e}_j \Leftrightarrow e_{k_i} \prec e_{k_j} \qquad \hat{e}_i \rightarrow \hat{e}_j \Leftrightarrow e_{k_i} \rightarrow e_{k_j}$$

In case $\hat{e}_i, \hat{e}_j$ have the same anchor $e_k = e_{k_i} = e_{k_j}$, it does not make sense to order them. Considering their location, they are effectively the same event. Nevertheless, their respective semantic information is still useful for analysis. For the sake of simplicity, and without loss of generality, we refer to all events as $e \in T$ in the rest of this paper.

B. Software Behavior Control

An event trace, if *complete* (i.e., every inter-task communication and non-determinism source is captured), precisely represents the behavior of a given execution. Hence, the system can behave the same if a *controller* $\mathcal{C}$ is added that forces the platform to produce such a trace again (see Fig. 1). A different (and possibly buggy) behavior can also result if the controller forces the production of a slightly different trace.

To affect the overall system state, the controller can inject a set of control operations $Z = \{z_1, ..., z_m\}$ which cause the system to delay or anticipate the occurrence of events. To that end, either *asynchronous* or *synchronous* operations might be needed depending on the event's nature. Synchronous operations are bound to synchronize with a specific execution context (e.g., restoring a register of task "X"), whereas asynchronous are not (e.g., unconditionally writing data to a global register). Control operations may refer to different contexts and different points in time. For instance, deliberately scheduling a task to force an event's occurrence is done through a function call injection, which involves writing to memory, changing register values and restoring registers on function return.

Behavior control is achieved by enforcing a specific order of events from a reference trace T. Let $\mathcal{B} \subset T^2$ be the set of *control constraints*, i.e., a set of event pairs that have to be enforced during execution to trigger a given behavior. Using the set $\mathcal{B}$ as an input, the controller $\mathcal{C}$ produces appropriate actions $Z = \mathcal{C}(\mathcal{B})$ that steer the system behavior accordingly (see Fig. 1). The controlled execution of a system can be then seen as a function $\xi(\mathcal{C}, Z, \mathcal{B}) = \dot{T}$, where $\dot{T} = \{e_i\}$ denotes the resulting trace. The problems of how to deterministically replay an execution and how to provoke bugs reduce to that of determining the right control constraints $(e_i, e_j) \in \mathcal{B}$. For instance, a *strict replay* is achieved if $\mathcal{B} = \prec$, i.e., $\xi(\mathcal{C}, Z, \prec) = T$. In this case, the controller would serialize the whole platform execution by forcing all events in T to

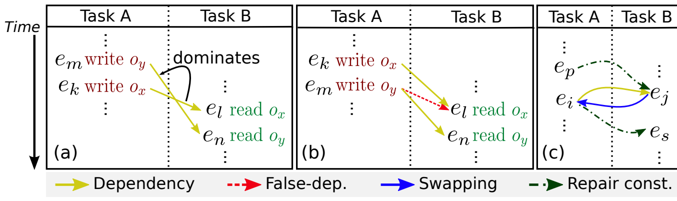

Fig. 3: (a) Domination, (b) false-dependency and (c) swapping

appear in the same order through iteratively suspending and resuming execution contexts as needed. This strategy, albeit possible, significantly reduces the execution speed due to the serialization of contexts and the overhead caused by the excessive number of control interactions. A better strategy is to serialize only inter-task dependencies, as serializing contexts and/or events that do not interfere is unnecessary. In general, the amount of serializations implied by the set $\mathcal{B}$ should be minimized. All in all, an *oracle* is needed that determines a suitable $\mathcal{B}$ which leads first to a deterministic replay or, if manipulated, provokes a bug as shown in Fig. 1.

Problem 2. Given the poset $(T, \prec)$, find an oracle $\mathcal{O}$ that determines $\mathcal{B} \subset T^2$, such that $\xi(\mathcal{C}, Z, \mathcal{B}) = \dot{T}$, $\dot{T} \equiv_b T$ and $|\mathcal{B}|$ is minimized. The relation '$\equiv_b$' denotes that T and $\dot{T}$ represent the *same execution behavior* but do not necessarily contain the same events or order. Behavior is checked by comparing program output (e.g., file I/O) or exit codes.

III. Analysis and Constraint-Based Replay

Dependency extraction. Without inter-task *dependencies*, the execution order would not matter for parallel programs. The search for dependencies in an event trace is a key concept of concurrency analysis [10], [16].

Definition 5 (Dependency)**.** Two events $e_i = \{t_i, p_i, c_i, a_i, d_i\}, e_j = \{t_j, p_j, c_j, a_j, d_j\} \in T$ *are dependent iff* they reference at least one common object o, $c_i \neq c_j$ and if inverting the order of $e_i \prec e_j$ has an effective change on the program behavior.

We model each identified shared object o (e.g., shared memory address or semaphore) as a list of *versions* $o = \{m_1, ..., m_n\}$, where each version is a triple $m = (e_o, d, V)$ consisting of an owning event e_o, some object specific data d and a list of visitor events V. Dependencies for every o are extracted with an online algorithm based on [17]. A shared object is initialized with a null version with $V = \emptyset$. Every access to a shared object o from an event e can be recorded either as a *modification* or a *visit* operation. A *modification* indicates a change which is observable by other contexts and adds a new version to o (i.e., $o \leftarrow o \cup m_{\text{new}} = (e, d, \emptyset)$). *Visiting* means that the event only reads the object's value; the event is added to the visitors list of the newest version (i.e., $V_{\text{last}} \leftarrow V_{\text{last}} \cup e; V_{\text{last}}$ from m_{last}). Given an object o, our algorithm marks an event pair (e_i, e_j) as dependent if *(i)* the events own different versions $m_i, m_j \in o$ (*modify−modify*), *(ii)* if e_j happens to be in the visitor list V_i of an m_i owned by e_i (*modify−visit*) or *(iii)* in the visitor list of any version m_p preceding m_i (*visit−modify*). Finally, let $D \subseteq \prec$ be the set of all dependencies.

Synchronization extraction. Dependencies are harmless if they are *synchronized*. Synchronized dependencies are those $e_i \prec e_j \in D$ for which the execution order cannot be reversed, for they comply with $e_i \rightarrow e_j$ (see Definition 3). Finding the "$\rightarrow$" relation on T is possible since all synchronization primitives (e.g., mutexes, signals, messages) and context maintenance actions (e.g., task creation and deletion) are exposed by a corresponding HLE. For computing "$\rightarrow$", we use *distributed clocks*, which are similar to vector clocks in [14]. An event e_{create} (e.g., a *thread create*), for instance, updates the local timestamp of the calling context (i.e., the context where the function is executed) and then creates a new context which inherits its timestamp. The very first event of the new context, e_{first}, is aware of the progress of e_{create} and thus they are synchronized: $e_{\text{create}} \rightarrow e_{\text{first}}$. Distributed clocks apply to task create and join operations, but also to shared objects that enforce absolute synchronization and can transport a sender's clock, such as messages (or signals). Mutexes, however, do not enforce an absolute order but rather a local one (e.g., a mutex *acquire* will only succeed after the owner *releases* it). We use the aquire-release semantics of mutexes to extend the set "$\rightarrow$": Considering $e_{a_i} \prec e_{r_i}$ are a mutex acquire and its corresponding release happening in context c_i, if context c_j emits an acquire event e_{a_j} such that $e_{a_i} \rightarrow e_{a_j}$, then $e_{r_i} \rightarrow e_{a_j}$ holds. Let $D_u = (D \setminus \rightarrow) \subseteq \prec$ be the set of all unsynchronized dependencies.

Dependency set pruning. After extracting synchronization, the set D_u can be further minimized. In fact, there can be a dependency $e_i \prec e_j$ which is always executed in order given that the controller $\mathcal{C}$ enforces the execution order of another dependency $e_l \prec e_m$. This is possible due to *(i)* the transitivity of "$\prec$" and to *(ii)* synchronization between the events e_i, e_j, e_l, e_m. A similar consequence of transitivity is that a virtual dependency can be introduced to force the execution order of two real dependencies from D_u. For this, *domination* and *false-dependencies*, shown in Fig. 3(a) and (b) respectively, are introduced.

Definition 6 (Domination)**.** Given two pairs $b_i = (e_k, e_l), b_j = (e_m, e_n) \in T^2$, we say $b_i \operatorname{dom} b_j$ *iff* $(e_k \equiv e_m \vee e_m \rightarrow e_k) \wedge (e_l \equiv e_n \vee e_l \rightarrow e_n)$. This indicates that b_j is a redundant element w.r.t. b_i. The relation "dom" is transitive.

Definition 7 (False-dependency)**.** Given $b_i = e_k \prec e_l, b_j = e_m \prec e_n \in D_u$, the set of all false dependencies for the pair (b_i, b_j) is given by $fdep(b_i, b_j) = \{b_{\text{false}} \in (T^2 \setminus D_u) \mid (b_{\text{false}} \operatorname{dom} b_i) \wedge (b_{\text{false}} \operatorname{dom} b_j)\}$. Every *false dependency* b_{false} can replace $b_i, b_j \in D_u$ and keep their respective orders. Let D_{false} be the union of all elements in the image of $fdep$, which contains all false dependencies.

Recalling Problem 2, the oracle $\mathcal{O}$ needs to create the sets D_u and D_{false}, and then check for domination in order to construct $\mathcal{B}$. The solution set $\mathcal{B}$ is expressed as:

Theorem 1. Given the poset $(T, \prec)$, the least set $\mathcal{B} \subset T^2$ which ensures that $\xi(\mathcal{C}, Z, \mathcal{B}) = \dot{T} \wedge \dot{T} \equiv_b T$ is given by:

$$\mathcal{B} = (D_u \cup D_{\text{false}}) \setminus D_{\text{dom}}, \text{ where}$$

$$D_{\text{dom}} = \{b_y \in (D_u \cup D_{\text{false}}) \mid \exists b_x \in (D_u \cup D_{\text{false}}) : b_x \operatorname{dom} b_y\}$$

Proof: By definition, every unsync. dependency $b_i \in D_u$ must be considered or the program behavior could change. This is enforced by b_i or other $b_j \in \mathcal{B}; b_j \operatorname{dom} b_i$. False dependencies and considering "dom" on all $b_i, b_j \in \mathcal{B}$ ensure that there is no other element that can further minimize $|\mathcal{B}|$. By contradiction and without lose of generality, consider two elements $b_y, b_z \in \mathcal{B}$. Consider also a

b_x that can replace b_y, b_z further reducing $|\mathcal{B}|$. Then, $(b_x \operatorname{dom} b_y) \wedge (b_x \operatorname{dom} b_z)$ should hold. If $b_y, b_z \in D_u \Rightarrow b_x \in fdep(b_y, b_z) \Rightarrow b_x \in D_{\text{false}} \Rightarrow b_y, b_z \in D_{\text{dom}}$, thus contradicting $b_y, b_z \in \mathcal{B}$. If $b_y, b_z \in D_{\text{false}}$, then $\exists b_p, b_q \in D_u$: $(b_y \operatorname{dom} b_p) \wedge (b_z \operatorname{dom} b_q) \Rightarrow (b_x \operatorname{dom} b_p) \wedge (b_x \operatorname{dom} b_q) \Rightarrow b_x \in D_{\text{false}} \Rightarrow b_y, b_z \in D_{\text{dom}}$, thus contradicting $b_y, b_z \in \mathcal{B}$. When $b_y \in D_u$ and $b_z \in D_{\text{false}}$, $b_x, b_y \in \mathcal{B}$ is also contradicted, merging the two cases above. ■

So defined, the set of control constraints $\mathcal{B}$ corresponds to unsynchronized dependencies which represent *races* (except for false dependencies). Races in $\mathcal{B}$ can still be either *harmful* or *harmless* [9]. For the latter, order does not have to be enforced to achieve $\dot{T} \equiv_b T$ (i.e., they do not affect program correctness). Therefore, it can be argued that harmless races could be removed from $\mathcal{B}$. This does not contradict the minimality requirement of $|\mathcal{B}|$ however, but rather shows the limits of pure dynamic concurrency analysis. Identifying *harmful* races needs additional semantic information, e.g., extracted by a compiler [9].

Constraint-based replay. Given the set of control constraints $\mathcal{B}$, we implement the controller as a set of parallel routines that interact with the system simulator. Each routine controls the execution of a single context c by using the algorithm shown in Fig. 4 on a per-context filtered trace $T_c \subset T$. In order to steer execution, the controller needs to be able to *(i)* $suspend$ the current context and $resume$ others associated to an event, *(ii)* determine a $match$ between two events and *(iii)* $provide_input$ necessary for the system, if any, such that an event is generated (e.g., for events associated to random input). If a given routine expects an event which is a constraint's *second* event e_{second}, it must suspend execution if its *first* event e_{first} has not occurred yet. Complementarily, if e_{first} happens, the context of the corresponding e_{second} (i.e., $context(e_{\text{second}})$) is resumed and its routine unblocked in case that it is already waiting for e_{first}. This way, two routines cooperate to enforce the execution of a single constraint which is always associated with two different contexts. $Suspend/resume$ both work by managing an internal queue for every context such that repetitive calls are cumulative but the action is done only once (i.e., suspend with the first call, resume when the queue is empty). Due to the properties of $\mathcal{B}$, this algorithm will repeatedly and deterministically yield the same execution behavior captured by the original event trace T.

IV. Behavior Exploration for Bugs

A more interesting usage of the previous concepts, and the main goal of this paper, is to automatically explore the system's behavior for bugs. Every race obtained after considering dependencies, synchronization and domination (but excluding false dependencies) might indicate a conflictive order-dependent inter-task interaction, which might lead to a bug. Let $\mathcal{B}' = \mathcal{B}|_{D_{\text{false}}=\emptyset} = D_u \backslash D_{\text{dom}}$, i.e., the set produced by the oracle without false dependencies. Every $s_i \in \mathcal{B}'$ is a candidate conflict for bug exploration. The controller can apply several strategies to use each s_i in order to trigger a latent bug.

Simply *swapping a constraint* could unveil an atomicity or order violation bug (if the constraint represents a harmful race in a semantically atomic or ordered application section). Violations affecting the semantic correlation of application variables [9] can also be provoked by swapping several races that lead to break multi-variable correlation. To provoke and

```
1:  while T_c ≠ ∅ do
2:      e_expected ← pop_first(T_c)
3:      provide_input(e_expected)
4:      for all s = (e_first, e_second = e_expected) ∈ B do
5:          mark s
6:          suspend(c)
7:      end for
8:      e_current ← (... wait for next event in c from the target)
9:      if ¬matches(e_current, e_expected) then
10:         replay error
11:     end if
12:     for all s = (e_first = e_current, e_second) ∈ B do
13:         if s is marked then
14:             resume(context(e_second))
15:         end if
16:         B ← B \ s
17:     end for
18: end while
```

Fig. 4: Constraint-based replay routine for a single context

find bugs, we implemented a pseudo-random constraint swapping algorithm for iterative bug exploration. The algorithm works by iteratively reversing the order of a random constraint $s_x = (e_i, e_j) \in \mathcal{B}'$ into $s_{\text{swapped}} = (e_j, e_i)$ and letting the controller replay the application. Note that this order reversal does not lead to a cycle in the happens-before relation (which would be an inevitable deadlock). If it did, there would be a redundant path from e_i to e_j, which disagrees with Theorem 1. However, s_x may dominate other dependencies. Since swapping affects the original order, it leaves previously dominated dependencies ignored. To avoid this, additional *repair constraints* are added to recover the order of adjacent dependencies. Assuming P_{e_i} and S_{e_j} are the respective sets of events immediately preceding and succeeding the events e_i and e_j of the selected constraint s_x, then the repair constraints after swapping s_x are (see Fig. 3c):

$$\{e_p \prec e_j | e_p \in P_{e_i}\} \cup \{e_i \prec e_s | e_s \in S_{e_j}\}$$

If the application fails during replay, the latest swapped constraint is most probably involved in the bad behavior. If the application succeeds, the new trace is used for the next iteration. A swapping history must be kept to minimize the chance of replaying the same execution. Despite the randomness of this strategy, we believe its search space is promising. In contrast to blind heuristics such as Chess [4], each swap actually considers a dependency which is likely to change program behavior. On the other hand, since it only explores the outermost visible dependencies, the algorithm has an effect similar to tools that randomly cause slight schedule changes. If an application is completely synchronized by dependencies, but the analysis is unaware, the heuristic will try to swap synchronization points again and again. The exploration approach can be improved by using application semantic information (e.g., to identify harmless races) or clues about a sought bug (e.g., bug patterns).

V. Test Cases and Results

To test our approach, we implemented a debug framework that connects to a Synopsys VP of the ARM Versatile Express CoreTile board. The VP consists of four instruction-set-accurate Cortex-A9 cores, an AXI bus and a set of peripherals (e.g., USART, LCD controller). It supports booting a SMP Linux 3.4.7 kernel. On it, we ran different C applications to test our framework under different code complexity levels. The debug framework is able to non-intrusively monitor and abstract events related to OS thread

management (e.g., via Linux `thread_info`), process/library loading (i.e., via `vma_link`), signals (e.g., `SIGSEGV`) and file input/output (i.e., `read` and `write`). Pthreads calls for mutexes (e.g., `pthread_mutex_lock`) and barriers as well as shared memory accesses are also monitored. We also built the execution controller from Fig. 4 for the target Linux. It uses debug and control VP interfaces so as to externally inject calls (i.e., manipulating/saving/restoring registers) to kernel routines which induce task suspension (i.e., `schedule`, `sched_yield`) and resumption (i.e., `wake_up_process`). Finally, the analysis proposed in Section III was added.

Three different applications were analyzed. Ocean (simulation of ocean movements) and fmm (n-body interaction simulation) were taken from the SPLASH-2 benchmark [18]. Pigz [19] (compression), which is used in mobile devices, was also used. Table I shows amount of lines-of-code (LOC), threads, events (both low-level and HLEs) and all extracted conflictive races (races on mutex objects and shared memory).

For the three applications, only two races were shared "memory races" (in ocean). Such races could relate to atomicity violations (the most common type of bugs [5]) and are deemed to be more problematic. Races on mutexes denote relative orderings of critical sections and are less likely related to bugs. However, bug exploration with our tool did not trigger a buggy state for ocean and fmm. This was expected as SPLASH-2 is a stable benchmark. Later, we manually confirmed (in several man-days) that the two memory races in ocean were harmless. Then, we decided to randomly remove a pair of mutex acquire/release and observe the effect. Fmm_b and ocean_b in Table I correspond to the mutex removal in fmm and ocean from `fmm.C:169-172` and `slave1.C:516-518` respectively. Exploring ocean_b for bugs, now with 3 memory races, immediately led to a buggy state. The constraint swapping algorithm decided to swap the order of the new conflictive race. A bug was identified by comparing the program output to a reference (the resulting "residual norm" output differs). Similarly the first bug exploration in fmm_b, now with 2 memory races, caused the program to fail with an error message and return (-1) as exit code. This is fortunate considering the randomness of our swapping algorithm but an exhaustive exploration of all races would also be possible. Our tool is also aware of the swapped event order (including actions, contexts and extra data) causing the bugs, thus it provides comprehensive debug output for a programmer (e.g., source location, thread IDs and buggy interleaving). We repeated the random removal of mutex acquire/release pairs several times with similar results. For pigz, the situation was different. The identified "mutex races" were explored trying to find an order-violation bug. However, after 200 exploratory iterations no bug was found. Later, by inspecting pigz manually, it was discovered that it is in fact correctly synchronized. Thus, our tool was just swapping synchronization points that are not recognized as such. Not finding bugs in pigz is consistent with the developer's repository log, where concurrency-related issues are not mentioned.

Exact behavior replay, another useful feature of our debug approach, worked successfully for all applications. Table I shows the simulation performance, given as the ratio between simulation and wall-clock time, for the original and the replayed run. A slowdown of ~10x to ~77x was seen during replay w.r.t. the original. Our monitoring and control injection for automatic debug naturally has a cost on simulation speed. Still, it is less than with a strict replay (i.e., serialization of all events), which yields ~1600x slowdown in ocean. Parallel or hybrid simulation [20] can be used to mitigate the slowdown.

	Statistics		Events		Races		Sim./Wall-clock	
Name	LOC	Threads	HW-level	HLE	Total $\lvert\mathcal{B}\rvert$	Sh. Mem	Original	Replay
fmm	3255	3	29994	21697	8	0	0.05	0.005
ocean	4198	2	151580	139759	31	2	0.07	0.0008
pigz	5333	3	19679	18075	10	0	0.019	0.0018
fmm_b	3253	3	23425	18925	9	2	0.05	0.0049
ocean_b	4196	2	149511	138240	31	3	0.07	0.0009

TABLE I: Analysis of selected applications

VI. Conclusions

This paper presented a debug approach which monitors, controls and explores behavior of MPSoCs. The approach is based on dynamic abstraction and analysis of non-intrusively monitored events in VPs. The analysis identifies conflicting concurrent interactions which can either be used to reproduce a behavior or manipulated to provoke bugs. An iterative exploration process based on random constraint swapping, which helps to detect actual bugs, completes the debug approach.

Acknowledgment

This work has been supported by the European Comission through the FP7 EURETILE project (www.euretile.eu).

References

[1] UBM Tech Electronics, "Embedded market study," http://e.ubmelectronics.com/2013EmbeddedStudy/, 2013.

[2] International Technology Roadmap for Semiconductors (ITRS), "Design," http://www.itrs.net/, 2011.

[3] H. Pashler, "Dual-task interference in simple tasks: data and theory." *Psychological bulletin*, vol. 116, no. 2, 1994.

[4] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. Nainar *et al.*, "Finding and reproducing heisenbugs in concurrent programs," in *USENIX Symp. on operating systems design and implementation*, 2008.

[5] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics," in *ASPLOS*, 2008.

[6] J.-D. Choi and A. Zeller, "Isolating failure-inducing thread schedules," *SIGSOFT Soft. Eng. Notes*, vol. 27, no. 4, 2002.

[7] P. Joshi, M. Naik, C.-S. Park, and K. Sen, "CalFuzzer: An extensible active testing framework for concurrent programs," in *CAV*, 2009.

[8] D. Engler and K. Ashcraft, "RacerX: effective, static detection of race conditions and deadlocks," *SIGOPS Oper. Syst. Rev.*, no. 5, 2003.

[9] B. Kasikci, C. Zamfir, and G. Candea, "Data races vs. data race bugs: telling the difference with portend," in *ASPLOS*, 2012.

[10] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang *et al.*, "MUVI: automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, 2007.

[11] "Virtualizer," http://synopsys.com/Systems/VirtualPrototyping.

[12] T. Yu, W. Srisa-an, and G. Rothermel, "SimTester: a controllable and observable testing framework for embedded systems," *SIGPLAN Notices*, vol. 47, no. 7, 2012.

[13] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, 1978.

[14] C. J. Fidge, "Timestamps in message-passing systems that preserve the partial ordering," in *Australian Computer Science Conference*, 1988.

[15] L. G. Murillo, J. Harnath, R. Leupers, and G. Ascheid, "Scalable and retargetable debugger architecture for heterogeneous MPSoCs," in *S4D*, 2012.

[16] R. Agarwal and S. D. Stoller, "Run-time detection of potential deadlocks for programs with locks, semaphores, and condition variables," in *Parallel and distributed systems: testing and debugging*, 2006.

[17] T. J. LeBlanc and J. M. Mellor-Crummey, "Debugging parallel programs with instant replay," *IEEE Trans. Comput.*, vol. 36, no. 4, 1987.

[18] "SPLASH-2," http://www.capsl.udel.edu/splash/.

[19] "pigz," http://zlib.net/pigz, https://github.com/madler/pigz.

[20] L. G. Murillo, J. Eusse, J. Jovic, S. Yakoushkin, R. Leupers *et al.*, "Synchronization for hybrid MPSoC full-system simulation," in *DAC*, 2012.