DEBUGGING CONCURRENT MPSOC SOFTWARE WITH BUG PATTERN DESCRIPTIONS

Luis Gabriel Murillo, Weiqing Zhou[*]*, Juan Eusse, Rainer Leupers, Gerd Ascheid*

Institute for Communication Technologies and Embedded Systems (ICE)
RWTH Aachen University
Aachen, Germany
{murillo,zhou,eusse,leupers,ascheid}@ice.rwth-aachen.de

ABSTRACT

As MPSoCs become key components for the electronics industry, the programmability problem poses an ever increasing burden on the software development process. In addition to the difficulty of writing parallel applications, concurrency bugs are usually hard to find, understand and reproduce. Programmers writing parallel software need more support from the debugging tools in order to understand harmful effects of concurrent interactions. This paper introduces a debugger framework that detects concurrency bugs dynamically, based on user defined bug pattern descriptions. The framework can be configured to address different MPSoCs and different low-level APIs.

***Index Terms*—** MPSoC, embedded software, parallel programming, debug, assertions, bug patterns

1. INTRODUCTION

Multiprocessor System-on-Chip (MPSoCs) are rapidly becoming key components of modern embedded systems. However, writing parallel programs that make use of MPSoC platforms is difficult. Programmers have to face new challenges due to the nature of parallelism. The concurrent activation of system components leads to non-deterministic *processing interleavings* which could cause system failures. Deadlocks, livelocks, atomicity and order violations are common failures a programmer needs to deal with during software development.

In concurrent software, dedicated synchronization functions are used to enforce correctness in a system. Anticipating how synchronization behaves under unknown processing interleavings is difficult in a complex system. In real applications, the effects of concurrency are usually underestimated, thus resulting in concurrency bugs that are neither found nor fixed easily. In some cases bugs are difficult to understand to such a degree that fixing them introduces new erroneous behavior. This clearly shows the programmers' difficulty of reasoning about concurrent execution [1].

A concurrent application can be abstracted as a set of *events*, directly related to communication and synchronization among concurrent entities (e.g. low-level APIs, accesses to shared variables, scheduler activity). By observing the ordering of events in the system it is possible to spot harmful effects of processing interleavings. Based on this premise, new developments in concurrent software verification have succeeded to create models and tools that predict erroneous behaviors in parallel applications. However, the *dependencies* among events have to be known in order to find which ordering of events leads to a wrong execution. Deep knowledge of the behavior of low-level APIs and details about their implementation are needed to find these dependencies. This usually restricts the application of the theoretical models.

All existing event-based concurrent debugging techniques are developed by the general purpose computing community, and mostly target end-user application development. Programmers rely on solid APIs and OSs, and low level programming is scarce. Additionally, these debugging methodologies are normally intrusive (e.g. code instrumentation or function wrappers on APIs). In general, these techniques are not suitable when developing embedded software for MPSoCs. They do not consider heterogeneous architectures and communication infrastructures or irregular software stacks. They are also not applicable at different layers of the software stack.

This work presents a debugger framework that detects concurrency bugs in heterogeneous MPSoCs. The framework allows to specify bugs for a given system by defining patterns of event sequences and dependencies through an assertion-like language. At runtime, the debugger monitors the system events, finds wrong orderings according to the bug patterns and generates debug information that helps to find the cause of a failure.

In contrast to previous work, our debugger targets software development for embedded systems rather than for general purpose computers. By allowing the specification of bug patterns and events for a given system, our framework can be applied to heterogeneous MPSoC architectures and irregular software stacks. Furthermore, the proposed framework is compatible with state-of-the-art Virtual Platforms (VP), thus

[*]Also with Advanced Learning and Research Institute (ALaRI). University of Lugano, Switzerland.

enabling non-intrusive MPSoC debugging for software development at early design stages.

The rest of this paper is organized as follows. Section 2 presents related works in the area of concurrent software debugging. In section 3, the concept behind our MPSoC debugging technique is introduced. A framework based on assertion-like bug pattern debugging is presented in section 4. Next, a case study that demonstrates the usability of our debugging approach is presented in section 5. Finally, we conclude this work in section 6.

2. RELATED WORK

Previous work, particularly in concurrent software verification, aimed at abstracting and analyzing parallel applications, in terms of their synchronization and communication activity. Formal methods, like the ones presented in [2, 3], use detailed mathematical models to generalize and abstract the concepts of concurrent events and their dependencies. Other approaches use a lighter abstraction of concurrency but introduce on-line monitoring agents to observe system dynamics. Examples can be found in the MPI-aware debugger shown in [4] and in the tool for exploring processing interleavings presented in [5]. Few studies have addressed taxonomies of concurrent bugs, being major contributions to the field the works presented in [6, 1]. A preliminary categorization of concurrency bugs and their identifiable patterns were presented in [6]. In [1], the authors presented an extensive analysis of real world bugs and draw interesting conclusions about future directions in concurrent software debugging.

Our debugger framework features a unique combination of synchronization and communication abstractions, assertion-like bug patterns and dynamic monitoring of system properties. Furthermore, it applies these concepts to embedded systems, unlike its predecessors.

3. DEBUGGING SOFTWARE WITH BUG PATTERN DESCRIPTIONS

In our framework, concepts from concurrent software verification, distributed systems and hardware verification and validation are combined to define a new debugging methodology. Abstraction of synchronization and communication activity are common practices in the first two areas. On the other hand, assertional languages are widely used in hardware validation and verification (e.g. PSL [7], SVA [8]) in order to condition the correct behavior of a system. Although their typical usage is at the implementation level, recent studies have applied the same concepts to highly abstracted hardware models, as in SystemC-TLM platforms [9].

This section, apart from covering roughly some background, presents the main concept behind our MPSoC debugging technique.

3.1. Concurrent Events and Bug Patterns

The execution of a concurrent application can be seen as a sequence of events contributing to communication and synchronization [10]. Each event e can be classified according to behavioral, spatial and temporal characteristics, namely *attributes*, that uniquely identify it. Thus, attributes specify details such as the core which produced the event, the task to which it belongs, the action the event performs, the resource the event affects, among others.

Different types of events can be found in a concurrent application. Execution of synchronization primitives, accesses to shared variables, and usage of communication messages (e.g. through queues or NoC routers) result in system events that define an intended order in a concurrent application. Other system events abruptly change the execution order (e.g. scheduler's actions, like preempting a task). Depending on the implementation, events correspond to the execution of a machine instruction, a basic block or a function.

A bug appears when the ordering of dependent events is altered by unexpected processing interleavings. *Dependency* is defined as a relation between two events, expressed in terms of equality or inequality among their attributes. Wrong event orderings and event dependencies can be specified by defining bug patterns (BPs) for a given system. A BP is defined as an ordered sequence of events $(e_1...e_n)$ so that (i) if e_k happens before e_l then $k < l$, and (ii) e_l has a dependency on e_k. Finding a BP during execution means that the application correctness is compromised.

However, analyzing proper event orderings and evaluating event dependencies implies knowledge of low-level implementation details behind each event. For this reason, we define an *events specification* for every different system. System events are specified by using an OS/API description with relevant symbol names and function callbacks. This allows interaction with a particular platform in order to obtain events and their attributes at runtime. Normal debugger functionality, such as setting breakpoints and watchpoints, is used to trigger the capture of events. Commercial VP solutions, like Synopsys Platform Architect (SNPS-PA) [11], provide all the interaction needed to obtain events and their attributes without disturbing the simulated system.

3.2. Assertion-like Bug Pattern Descriptions

In our methodology, failures in a given system are separated and specified as BPs We introduce an assertional language called *BPDLang* that allows specifying BPs. BPDLang has Linear Temporal Logic (LTL) properties and is based on Assertion Based Verification (ABV) such as PSL. Both event ordering and dependencies can be specified with BPDLang.

BPDLang allows to specify erroneous sequences of high-level events in software, in a similar way the language described in [9] specifies sequences of transactions for verification of system level simulators. In BPDLang, a BP is spec-

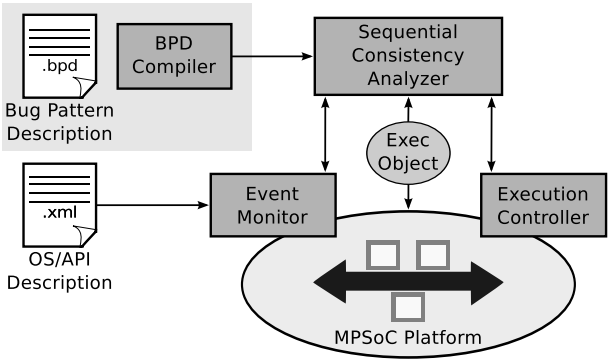

Fig. 1. MPSoC debugger framework architecture

ified using the keyword `pattern` which defines the placeholder for a list of *propositions*. Additionally, each pattern defines a header section that can be used to configure specific actions when the bug is detected (e.g. alert the user, stop the simulation, call a user defined callback function). Each proposition acts on an event e (keyword `on` plus field $trigger$), specifies event dependencies (field $condition$) and defines other system properties to provide temporal control and control flow among propositions (fields $timeout$, $exit$, $skip$, $restart$, $repeat$). The latter enhances the expressiveness of the language and makes it more effective for debugging.

Listing 1 shows an inter-task deadlock pattern described with BPDLang. In the code excerpt, a pattern is created with 4 propositions, w, x, y and z, corresponding to 4 events, namely a, b, c and d. The sequence on which they appear in the pattern (i.e. $wxyz$) defines an ordering of events that will be evaluated at run-time to identify the occurrence of a bug. In each proposition, $trigger$ indicates that w, x, y and z will be evaluated upon the execution in software of a "lock_acquire", a "lock_release", a "lock_acquire", and a "lock_release" respectively. On the other hand, a, b, c and d are only recognized as a sought event if their associated $condition$ is evaluated to be true. The $condition$ field defines the dependency of events a, b, c and d in terms of their attributes "task id" (tid) and $resource$. According to the bug pattern in listing 1, a bug will be found when:

- A "lock_acquire()" function is triggered in any task or processing element. (event a)
- A "lock_acquire()" function is triggered in a task different to the one that triggered event a, and locks the same resource. (event c)
- An abnormally long time is spent in this situation (given here as 200ms). (event d)

Event b and the trigger part in event d are introduced to cancel the entire pattern and declare the bug as non existent. The occurrence of a "lock_release()" function, the fulfillment of the $condition$ and a value of $true$ for the $exit$ qualifier are used in this regard.

Listing 1. BPDLang describing inter-task deadlocks

```
pattern inter_task_deadlock
{
  proposition w on a
  {
    trigger:=lock_acquire;
  }
  proposition x on b
  {
    trigger:= lock_release;
    condition:= b.tid==a.tid &&
               b.resource==a.resource;
    skip := true;
    exit := true;
  }
  proposition y on c
  {
    trigger:= lock_acquire;
    condition:= c.tid!=a.tid  &&
               c.resource==a.resource;
  }
  proposition z on d
  {
    timeout := 200ms;

    trigger:= lock_release;
    condition:= d.tid==a.tid &&
               d.resource==a.resource;
    exit := true;
  }
}
```

4. DEBUGGER FRAMEWORK ARCHITECTURE

The previous methodology is used in the debugger framework shown in Figure 1. Our framework is composed of four components, namely *Sequential Consistency Analyzer*, *Event Monitor*, *Execution Controller* and *Bug Pattern Descriptions Compiler*. The main idea behind this design is to clearly define and differentiate components than must be adapted for other platforms or applications.

When debugging, the user provides a *Bug Pattern Description* (BPD) file written in BPDLang. The BPD specifies a set of possible concurrency bugs for the system under analysis and lists the events that need to be monitored at runtime. An OS/API description file (OS-API.xml) specifies the implementation-dependent interactions on which event monitoring relies. After the framework inputs have been defined, each block in the framework performs the following tasks:

BPD Compiler. Processes the BPD and generates (i) a list of events that should be monitored, and (ii) executable representations of the bug patterns.

Event Monitor. Interacts with the platform using debugger commands, retrieves the events and streams them to the analysis layer (i.e. to the Sequential Consistency Analyzer). Its interaction with the system is defined by the OS/API description file.

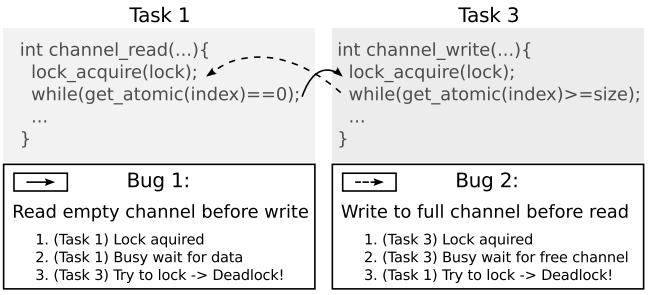

Fig. 2. Wrong implementation of FIFO access functions in low level API.

Sequential Consistency Analyzer. Looks for concurrency bugs by evaluating the application execution and filtering it with patterns inside the executable BPD representation. It also steers the system execution through the Execution Controller

Execution Controller. Provides an interface between the analysis layer and the platform specific debug control commands.

5. CASE STUDY

The bug pattern assertion approach was applied when debugging low level software of an MPSoC composed of 1 Tensilica Diamond DC_B_570T (controller processor) and 3 XRC_D2MR DSP-like extensible cores [12]. The system's software stack consists not only of multi-tasking run time environment (MTRTE) and priority-based scheduler (DSCHED) for each individual processor, but also contains several communication and synchronization APIs. The framework of Figure 1 was connected to a VP implemented using SNPS-PA. An OS/API description to specify how to capture relevant events was created with basis on the system APIs. BPs for deadlocks, data races and atomicity violations were created using BPDLang.

The debugger framework was tested when MTRTE and DSCHED were ported to one of the Tensilica cores. In that case, a "working" parallel MJPEG application, which uses FIFO channels for communication, was used to test the new MTRTE and DSCHED. After some executions, the task scheduling mechanism in DSCHED stopped completely and the MJPEG application executed wrongly. Our framework detected two bugs, shown in Figure 2, caused by a wrong implementation of the FIFO access functions ("channel_read()" and "channel_write()"), thanks to the simple deadlock description shown in Listing 1.

6. CONCLUSIONS

The use of communication and synchronization events and their interactions as debugging information, reduces the difficulty of concurrent programming. In this way, users can understand better the complex effects of non-determinism and processing interleavings. Moreover, MPSoC debuggers need to be flexible enough and allow configurations, specially during early design stages when application and architecture are constantly changing. Programming at different levels, such as in OS porting or middle-ware and driver development, are tasks that should benefit from new MPSoC debugging techniques. The Debugger Framework presented in this paper is a tool that successfully integrates these characteristics. When used together with virtual platforms, the framework becomes non-intrusive and the system allows deterministic executions thus covering cases where other debugging techniques fail.

7. REFERENCES

[1] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics," *ASPLOS 08*, 2007.

[2] Vineet K. and Chao W., "Universal Causality Graphs: A precise happens-before model for detecting bugs in concurrent programs," *Computer Aided Verification*, 2010.

[3] N. Sinha and C. Wang, "On interference abstractions," *SIGPLAN Not.*, 2011.

[4] D. Kranzlmüller, S. Grabner, and J. Volkert, "Debugging with the MAD environment," *Parallel Comput.*, 1997.

[5] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. Nainar, and I. Neamtiu, "Finding and reproducing heisenbugs in concurrent programs," *8th USENIX Symposium on operating systems design and implementation*, 2008.

[6] Eitan Farchi, Yarden Nir, and Shmuel Ur, "Concurrent bug patterns and how to test them," in *Int. Symposium on Parallel and Distributed Processing (IPDPS)*, 2003.

[7] IEEE Std 1850-2007 IEC 62531:2007 (E), "IEC standard for property specification language (PSL)," .

[8] IEEE 1800-2009, "IEEE standard for system verilog-unified hardware design, specification, and verification language," .

[9] K. Tomasena, J.F. Sevillano, J. Perez, A. Cortes, and I. Velez, "A transaction level assertion verification framework in SystemC: An application study," *Advances in Circuits, Electronics and Micro-electronics*, 2009.

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

[11] "Synopsys Platform Architect," `http://www.synopsys.com/systems/architecturedesign`.

[12] "Tensilica Diamond and Xtensa processor families," `http://www.tensilica.com/`.