Debugging Component-Based Embedded Applications

Kevin Pouget[*†]

Vania Marangozova-Martin[†]

Miguel Santana[*]

Jean-François Mehaut[†]

[*]STMicroelectronics
Crolles, France
first.lastname@st.com

[†]University of Grenoble, LIG
Grenoble, France
first.lastname@imag.fr

ABSTRACT

With the advent of parallel architectures in the domain of embedded systems, developing applications fully adapted to their underlying platform becomes more and more complicated. Finding and fixing bugs in such environments is even trickier. Furthermore, current complex applications cannot be developed from scratch, only based on programming language primitives. They have to follow advanced programming models and their execution will be driven by the APIs of the key underlying libraries.

In this paper, we propose a new approach for source-level debuggers. Going beyond their long-established ability to support sequential programming languages, we describe the functionalities a debugger should be able to provide to debug embedded and parallel component-based applications. Then we demonstrate our solution to this problem with a debugger targeting the component framework used on an MPSoC platform. We also explain the development challenges we faced during the implementation of this GDB-based debugger and illustrate its efficiency though a case study of an image processing application.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Code inspections and walk-throughs*; D.4.7 [**Operating Systems**]: Organization and Design—*Real-time systems and embedded systems*; D.3.2 [**Programming Languages**]: Language Classifications—*Componentware*

General Terms

Debugging, Component, Embedded, MPSoC

Keywords

debugging, components, parallel, embedded

SCOPES '12, May 15-16, 2012, St. Goar, Germany

1. INTRODUCTION

In order to answer the ever increasing demand for computational power, Multi-Processor-on-Chip (MPSoC) architectures have been introduced during the last decade in the domain of embedded systems. With high-quality image and video processing or voice encoding and decoding, embedded applications must be fully adapted to their underlying platforms to meet performance expectations. However, developers also have to face time-to-market requirements, meaning that application design, development and refinement phases must be as efficient as possible.

Component-oriented software engineering [7] is a programming model which puts an important focus on the design of independent building blocks and on reusability. This allows applications to be constructed dynamically by binding together such components. In order to exploit MPSoC parallel architectures, components can also be defined as *runnable* and bound to an execution context such as a thread or a process. As the underlying framework is in charge of component mapping, scheduling and communications, it will be able to optimize the deployment of components according to the different aspects of the underlying MPSoC architecture (e.g., shared or distributed memory, processor characteristics) and runtime requirements.

However, when it comes to the debugging phase, it appears that common tools like interactive debuggers only provide programming language-level features. Apart from threads and processes, none of the higher-level concepts used by developers to build applications are materialized in the debuggers. We believe that the next generation of debuggers should take into account the programming model as well as the runtime environment of applications. Indeed, they have a key role in the overall execution, and they reflect the programming guidelines followed during the development.

The main contribution of this article is a novel approach for the design of interactive debuggers, which describes the key functionalities that should be provided to allow an efficient debugging of dynamic component-based applications. We also detail how this approach has been implemented with an industrial component framework for an MPSoC system, on top of GDB, a popular free debugger.

This article is structured as follows: Section 2 introduces the problems encountered during component application debugging. Then, we detail in Section 3 our proposal for an efficient component-aware debugger. In Section 4, we discuss the existing parallel debuggers, as well as the current solutions for debugging components and objects-oriented ap-

plications. In Section 5 we describe our proof-of-concept debugger in the context of an embedded component framework developed by STMicroelectronics. We present how it is implemented as a GDB extension in Section 6. We evaluate its efficiency in Section 7 with the debugging of an image processing application case study. Finally, Section 8 summarizes the contributions of this paper and details the future work.

2. MOTIVATIONS

Interactive debugging is a complex activity where developers try to figure out where the actual code behaviour diverges from expectations. To do that, they must have a total and precise control over the application execution. During development phase, the component framework provides the relevant primitives required to implement a component-oriented application: interface and architecture definition languages, encapsulation mechanisms, deployment and binding primitives and so on. However, the debugging of component-based application is currently quite complicated, because of these additional abstractions. Traditional interactive debuggers are not aware of the component-related information, so they can only provide source-level features. In this section, we describe the main issues currently encountered by developers while debugging component-based applications.

Dynamic Architecture Components are standalone computation entities interconnected through their interfaces. They have the ability to be dynamically instantiated and bound to other components. Hence, the application architecture will change over time, according to the execution requirements. Traditional debuggers are not able to present most of these dynamic aspects. The similarities between threads and components may allow debuggers to list the live components, however no information about the interconnection network will be available.

Component Interactions During their lifespan, components offer services to the rest of the system through their *provided* interfaces. They may need services from other component to perform their task, which are described by the *required* interfaces. The execution of the components will be driven by the events received on each of these interfaces. The notion of interconnection does not exist in existing debuggers, so developers have to manually figure out the current component bindings and play subtly with breakpoints in order to follow a payload through an interface call.

Information Flow A component-based application can be composed of a large number of components, which will successively apply a given transformation to an information. In this case, developers will have to figure out the path followed by the suspicious pieces of information to understand the application state. However, existing debuggers only provide details about the *current* processor and memory state.

Current state-of-the-art debuggers do not account for such runtime information about the programming model. In order to simplify the debugging activity, an interactive debugger should be able to catch and work with the key events of the component framework, and provide the developer with the high-level abstractions used to develop the application.

In the next section, we present the debugging approach we designed to help developers to locate more easily the problems of component-based applications. For each of the main steps of the debugging process, we highlight the features which can be used to overcome the issues presented in this section.

3. COMPONENT-AWARE DEBUGGING

Our contribution consists of a set of functionalities that a debugger should implement, in order to allow efficient component-based application debugging. Our approach relies on the detection and handling of the key events executed by the component framework. These events should be interpreted to follow the evolution of the component-related aspects of the application. They allow the debugger to provide accurate information and control mechanisms to debug component-based applications.

Stop the Application Before the Error The debugger should leverage the additional events introduced by the component framework to provided more adapted flow control mechanisms. Namely, developers should be able to stop the execution upon specific variations in the application architecture, like components instantiation or destruction, or specific interface bindings.

Execute the Application Step-by-Step In addition to the previous mechanisms, the interactions between the components should be exploited to improve step-by-step execution. It should be possible to control the application according to the activity of a component interface and step into an interface call the same way one steps into a function call (it may involve several framework-dependant calls and/or another execution context). Developers should also be able to use the information flow to follow a given payload across the various components. For instance, setting a breakpoint on a message should stop the execution each time this message is transmitted to a new component.

Inspect the Application State Component architecture is an important aspect of the application state. So, the developer should be provided with an overview of the components currently deployed, including the interface bindings. The application information flow should also provided further details to the developer. First, message counters on component interfaces should facilitate the detection of unexpected usages. Second, knowing the route followed by messages should help developers to understand how the application reached its current state.

Two-Level Debugging Finally, as the instructions of component applications are eventually written in a given programming language and executed by the processor like traditional codes, language-based and low-level debugging commands should still be available. Indeed, although some bugs may lay in the programming model-related aspects of the application, there is a chance that the problems are hidden deep down in the language instructions. So, memory and processor inspection, breakpoints and watchpoints (maybe

component-specific) and other step-by-step execution control primitives should be directly available.

Section 5 and 6 present a proof-of-concept debugger implementing this approach, and Section 7 details a study case analysis of its efficiency. The following section discusses related work focusing on the debugging of complex applications and programming models.

4. RELATED WORK

Our approach lays at the confluence of different research areas. The first part of this section explores how parallel debugging is currently done in the context of High Performance Computing (HPC), which involves large numbers of system threads and processes interacting with one another. Shared-memory and data-parallel paradigms are out of the scope of this work, so the focus is put on the debugging of message-passing based application. Then object and component debugging is explored, with an analysis of how these programming models are currently handled in debuggers.

4.1 High Performance Computing Debugging

There exist different solutions to debug parallel applications. However, few of them investigate the question of programming models. The first part of this section considers two debuggers which try to reduce the quantity of information brought to the developer. Then, the second part introduces some approaches which try to take into account the application runtime environment in the debugger.

Large-scale applications, where thousands of tasks are executed simultaneously, involve tremendous quantities of information. A debugger must be able to filter out part of this data, first of all in order to perform efficiently, but also not to flood the developer with unusable information. In [2], Balle *et al.* define a debugger architecture based on a tree-like network of aggregators, with fully functional debuggers at each leaf. The aggregators provide an output-reducing mechanism, which should scale down the quantity of output data generated by the application. They identified three different types of output (identical, identical except a small variation and widely different), which allow the aggregators to merge them accordingly. Hence, the aggregator network limits the quantity of information reaching the top-level debugger and ensures an acceptable response time.
In a similar way, parallel debuggers like DDT [1] try to simplify the debugging activity by joining together duplicated information. For instance, processes can be grouped according to the function they are currently executing or in a tree merging the stacks of each process. Then, only a specific group will be debugged, allowing the developer to understand the divergences within a group more easily and thus the cause of the problem.
These two solutions are orthogonal to our problem, in the sense that we did not focus on the situation where a large number of identical tasks are involved. Indeed, we believe that interactive debugging should essentially target *functional* bugs, and other tools should be used to understand performance issues. So in most situations, it should be possible to narrow the problem down to a limited number of tasks. It is clear that our approach could not scale to debug several thousands of components: the debugger needs to keep information about each entity and track their interactions, so the parallel speedup of the application will certainly be completely lost. This will consequently lead to an unacceptable execution time. However, in order to improve the debugging of limited-scale applications, DDT-like mechanisms could be extended to support some of the new abstractions we have defined. For instance, groups could be formed according to the component definition, link or interface.

These research areas focus on the best way to present *well-known* information to the developer like the program stacks, memory contents, application and debugger outputs. But they do not look for *new* information. Let us now consider the few attempts which try to allow the debugger to provide new kinds of information.

The literature provides only a few examples which try to integrate the notion of message passing in the debugger. In [4], Cownie *et al.* present how they implemented MPI[1] message queue interpretation in TotalView. They introduce the idea of representing *conceptual* information about the message-passing model in the debugger. However they did not really go further than listing (in a standard way, with regard to the debugger and the MPI library implementation) the content of the internal message queues.
In [11], Schaeli *et al.* describe an interesting solution which provides a visual representation of the messages exchanged by MPI processes and allows the developer to explicitly control the ordering of the message-passing events. The authors explain that their main goal is to automatically or manually detect race conditions. Consequently, their approach does not focus on the other aspects of debugging, and their tool does not allow interactive debugging commands like step-by-step execution or memory inspection. The implementation they propose is based on the profiling API of MPI, which allows an external library to execute code upon specific events triggering. For instance, the process may wait for a debugger order before sending a message. Although this approach does not target the same class of problems as our work, their solution could be leveraged to adapt our work to MPI libraries.

4.2 Object and Component Oriented Application Debugging

The debugging challenges of component-based application shares some similarities with object-oriented applications: entities are dynamically created, they communicate with one another, and their inter-connections change over time. The authors of [10] present an object-centric debugger, which aims at shifting the debugger focus from the execution stack towards the objects themselves. We share part of their motivations, however they consider different abstraction levels and constraints. Their solution is based on the ability to dynamically modify the behavior of individual objects, for instance to hook object instantiations, method calls, etc. and inject a debugger notification. So it requires a programming language evolved enough to offer this capability, typically an interpreted language, or predefined hook points. Our work is more oriented towards low level languages, like the C language, which is frequently used in embedded systems. They also target programming language-level debugging, as they work with programming language concepts, whereas our effort is focused on the programming model. This means that they do not address the problem of bringing the debugger

[1]Message Passing Interface [9]

closer to the programming model abstractions used in the applications, and the developer will be left with overwhelming information about low level details. Nevertheless, their approach could be used in conjunction with ours, once the programming model related part of the debugging activity has narrowed down the problem search-space.

General component-based application debugging presents an additional difficulty, which is out of scope of this work, as it is not relevant to embedded systems. Component frameworks usually have the ability to bring together components written in different languages and/or black-boxes provided by third parties. In this scenario, traditional debuggers are of little help, as most of them do not support multi-language debugging, or debugging is simply not possible in the case of black-boxes. The authors of [14] propose a solution to this problem: they squeeze debugging components in between the application components. These components are able to *monitor* the component interface activities without any knowledge about the implementation and/or without exhibiting the multi-language debugging problem. The article describes some of the features which can be achieved with this concept, which include:

- Record and replay of the interactions, which allows a standalone re-execution of a component.
- Data and flow control analyses at component level.
- Setting breakpoints on the interfaces for interactive debugging.

This approach is non-negligibly intrusive as it requires the developer to build and manually connect the interface monitors (although they mention that future versions should be able to do it automatically). Also, as the debugging mechanisms are directly integrated in the application and certainly have a significant cost (especially in their context of scientific computing), the debug and production versions of the application will inevitably be different, and this discrepancy may hide some of the bugs.

5. DEBUGGING FEATURES FOR AN EMBEDDED COMPONENT FRAMEWORK

In this section, we describe our proof-of-concept debugger and the features it provides, which implement the approach described in Section 3. We focus on the embedded system *Platform 2012* (P2012) [12] and its component framework *Native Programming Model* (NPM).

P2012 is a low consumption, parallel and embedded platform research project developed by STMicroelectronics and CEA. It targets high-definition audio and video processing. The platform is shaped as an accelerator architecture, using a multicore general-purpose ARM processor on the host[2] side and clusters of STxP70 configurable processors on the fabric side. As depicted in Figure 1, the processing elements of a cluster share the L1 memory. Inter-cluster communication is done through the L2 memory, whereas host-fabric exchanges are performed by DMA[3] controllers with the L3 memory.

[2]We use the term *host* to refer to the general-purpose processor of the platform, which is different from its remote debugging/development meaning.
[3]Direct Memory Access

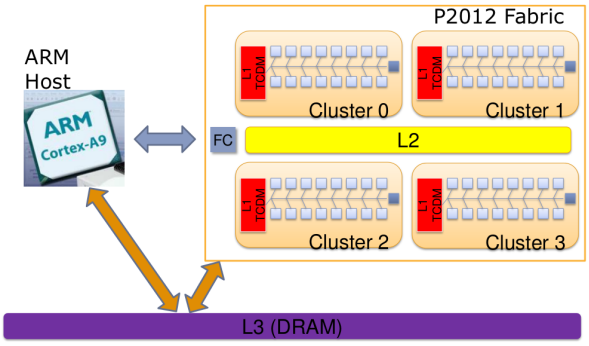

Figure 1: P2012 architecture

NPM is a component-based programming environment developed to exploit P2012 architecture. It offers a highly optimized framework providing guidelines for the implementation of application components, pattern-based communication components and a deployment API for the host side. In order to exploit the processors of the platform efficiently, NPM implements the concept of runnable components. Such components have to implement a specific interface, which will be triggered by the framework in a dedicated processor. The components will then be able to execute parallel code on the available processors of their cluster, based on the fork/join model [8]. We will only focus on the component aspect of NPM.

Although the component programming model is not widely used yet, it is well suited for embedded systems [5]. Indeed, components allow the adaption of the application architecture to the runtime constraints such as the workload, the power consumption or the available processors. The major reason of their low popularity seems to be the strong requirements that embedded systems must satisfy, like timeliness, quality-of-service or predictability, which are not achieved by traditional component frameworks.

In the following sections, we describe NPM's API along with the associated debug features provided by our debugger.

5.1 Host-side Deployment and Management

The deployment of runnable NPM components is done in two stages, from the host side of the application. The component is first instantiated on a given cluster (`CM_instantiate`). At this stage, the component can be configured and its interfaces must be bound to other component interfaces. Then, its `run` method can be triggered once or several times (`NPM_runRTMComponent`). Finally, component instances must be destroyed to release the memory (`CM_destroy`).

```
int CM_instantiate (name, target_cluster, *comp);
void NPM_runRTMComponent (instance);
int CM_destroy (instance);
```

Our debugger is able to catch the events corresponding to these functions and allows the developer to stop the execution at these points. Components are named according to the file in which they were compiled, and the instances are identified by a unique number. They can be in one of three states:

selected One of the execution contexts has been stopped in this component, and the debugger commands will be related to this component.

not selected One of the execution contexts has been stopped in this component and the debugger can switched to this component.

no execution context The component is not currently running.

The indication about the target cluster (`target_cluster`) is also interpreted. The host application is represented as a component to unify the handling of the different entities.

NPM allows components to be bound at runtime from the host with predefined communication mechanisms. A DMA controller transfers data from the host to the fabric (i.e., to the components); whereas a FIFO[4]-buffer based link transfers data between two components, either using shared memory inside a cluster or inter-cluster communication mechanisms:

```
int NPM_instantiateDMAPullBuffer (bufferId,
              consumerComp, consumerInputItf, ...)
...
int NPM_instantiateFIFOBuffer (bufferId,
              producerComp, producerOutputItf,
              consumerComp, consumerInputItf, ...)
```

In this scenario, the debugger recognizes the binding being established and provides it to the developer. A `DMA pull` binding connects the interface `consumerInputItf` of the component `consumerComp` to the pseudo-interface `buffer Id` of the host component. It is also possible to catch further details about the DMA configuration such as the buffer size and location or the access pattern. The same idea applies to `DMA push` and `FIFO` bindings. The target cluster information discovered at component instantiation gives a hint about the algorithm (shared or distributed memory) selected by the framework to implement the `FIFO` bindings.

Thanks to this information, developers have an accurate overview of the current deployment state of the application. As NPM components may have an execution context, the debugger memorizes the mappings between the internal threads and the application components. This gives the ability to developers to see the call stack of all the running component. Switching from one component to another can either be done directly (with their unique identifier) or through the name of the interfaces.

5.2 Communication Interfaces

As mentioned above, the components of NPM applications communicate through `Pull` and `Push` predefined interfaces. The `Pull` interface allows the reception a full data buffer, whereas the `Push` interface provides empty buffers and sending mechanisms.

```
interface npm.buffer.PullBuffer {
    /* Informs the communication component that is
       should start to fetch the buffer that will
       be returned by the subsequent call to the
       pull method.  */
    void fetchNextBuffer();
    /* Returns a buffer from which data can be
       read. This method may be blocking until a
        buffer is available.  */
    void *pull ();

    /* Releases a previously pulled buffer.  */
    void release(void *buffer);
    ...
}
interface npm.buffer.PushBuffer {
    /* Returns a buffer into which data can be
       produced.  */
    void *getBuffer();

    /* Push a previously returned buffer.  */
    void push (void *buffer);

    /* Wait for the termination of every ongoing
       transfers of previously pushed or sent
       buffers.  */
    void waitTransfers();
    ...
}
```

On the debugger side, we consider the `Push` and `Pull` interfaces as communication *endpoints* and model DMA controllers and FIFO-buffer links as inter-component *links*. In both interfaces, we elected the methods responsible for messages departure and arrival, namely `PushBuffer push` and `PullBuffer pull`.

These primitives give the debugger the ability to detect communication events, hence allowing the developer to control the execution based on the messages sent and received by the components. Component interface specific breakpoints can be set, and a finer-grain control can be achieved with condition checking. A message counter is also incremented every time a message is transmitted by the interface.

5.3 Message-Based Flow Control

NPM communication components exhibit clear and well-defined message-handling patterns. This allows our debugger to detect that, when the `PushBuffer push` interface method is triggered, a piece of information is leaving the sender component, and when the corresponding `PullBuffer pull` returns, the target component is able to process this very information. The message entities modeled in our debugger are purely conceptual, and therefore are not bound to any data structure of the application or the component framework.

In some cases like pipeline-shaped application architectures, developers may be able to identify such routing patterns in the components. This enables the message-based flow control capabilities of our approach. The idea of routing table is borrowed from networking, where routers, like components, transmit messages according to their incoming interface and information contained in the payload. Here, developers will define the tables principally based on the component and interface names, but also according to the current application architecture and memory state, as schematized in Figure 2.

Message-based flow control allows developers to set breakpoints on messages, instead of memory locations. To enable this feature, messages have a unique identifier and can be listed, either all at once or per component or link. Given

[4]First In First Out

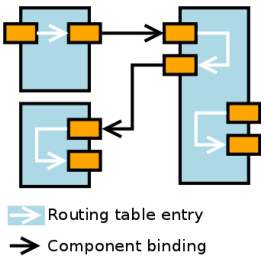

Figure 2: Routing table for message flow debugging.

a message identifier, one can set a permanent or temporary breakpoint which will stop the execution the next time the message is handled. Coupled with a stamping mechanism, either generic or defined by the developer, our debugger can give further information about the message history. The generic stamp will contain the component name and unique identifier, as well as the interface name and direction (message sent or received). The stamp list will inform the developer about the route followed by a message over the components.

The concept of message history can be illustrated as follows: considering a streaming application which applies a set of filters to frame data. Each filter is implemented in a dedicated component (`Component A`, `B` and `C`), similarly to Figure 2.

If the debugger does not have routing information about the application components, new messages will be generated each time a communication occurs:

```
Message 1:
    Component A                 # Message created
    Component A::Interface A.1 # Message sent
    Component B::Interface B.1 # Message received
Message 2:
    Component B                 # Message created
    Component B::Interface B.2 # Message sent
    Component C::Interface C.1 # Message received
```

which means that `Component A` first sent a message to `Component B`. Then, `Component B` sent a message to `Component C`.

Now, if the developer could provide a simple routing table expressing that `Component B` transmits the incoming messages towards `Component C`, then the history information would be simplified:

```
Message 1:
    Component A                 # Message created
    Component A::Interface A.1 # Message sent
    Component B::Interface B.1 # Message received
    Component B::Interface B.2 # Message sent
    Component C::Interface C.1 # Message received
```

which clearly indicates the route followed by the message.

Now that the features provided by our debugger have been detailed, the following section introduces its inner implementation specificities and how GDB was extended to achieve this.

6. EXTENDING GDB TO SUPPORT COMPONENT DEBUGGING

GDB is the debugger of the GNU project [13]. It has a very large user community in both general and embedded computing. We decided to base our work on this tool because of its advanced process inspection and control capabilities and to simplify user handovers. Moreover, recent versions of GDB export PYTHON bindings, which allow extensions to be developed easily and efficiently. Thus, all the classic functionalities of GDB are available in our debugger. Figure 3 presents a schematic representation of a traditional debugger like GDB connected to an MPSoC platform, as well as our component-awareness extension.

In order to limit the intrusiveness of the debugging in the execution, we decided not to alter the component framework. Therefore, the debugger needs to update its internal representations whenever the component architecture is modified.

Our runtime-information capture mechanism relies on internal function breakpoints set at the entry and exit of the programming model-related functions exported by the component framework. Based on the API definition, calling conventions and debug information, we parse the relevant function arguments to extract the information required to update the internal representations.

In the current NPM implementation, the framework API and implementation are rich enough to capture all the information required to implement our support. However, in some other component frameworks, further compile-time information might be required, such as the interface and architecture specifications used by the component builder.

Function breakpoints extend the traditional breakpoint mechanism by attaching to the breakpoint the semantic definition of the operation it monitors. Each time the breakpoint is triggered, a specific action is executed to update the internal representations. Based on NPM API and source code, we elected the locations responsible for key component operations and implemented the corresponding representation modification back to the debugger's function breakpoints. The concept of *finish breakpoint* was also introduced in GDB PYTHON API to programatically catch the return point of a function. This allows the interception of output and updated parameters.

In order to allow the debugger to manage the concepts

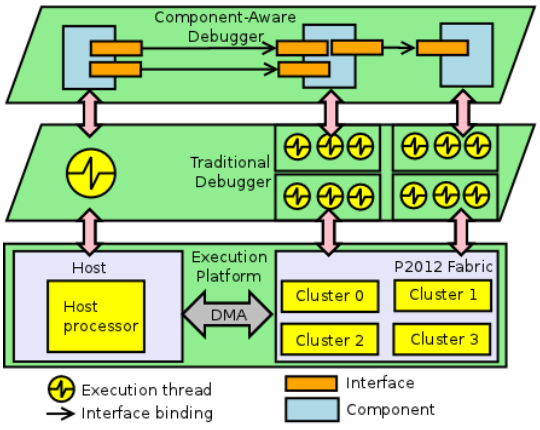

Figure 3: Two-level debugging of an MPSoC platform.

of NPM component-based programming, we defined three kinds of entities:

- The application components.
- The endpoints, corresponding to component interfaces.
- The links, corresponding to the bindings between the interfaces. In NPM, bindings are implemented with FIFO or DMA patterns.

The top part of Figure 3 presents how these entities are internally used to represent a component-based application.

In the current version of the debugger, no static information about the component model is used in addition to the standard DWARF [6] debug information. This allows the debugger to connect the application more easily. However, we could exploit compiler-generated information about the component structures and application architecture to enrich our model and optimize general performance.

Indeed, our frequent use of breakpoints introduces a slowdown in the application. This is mainly due to communication-related function breakpoints which may be triggered very frequently in communication-intensive applications. In order to cope with this problem, one can disable this set of breakpoints until the "critical part" of the execution is reached. This can be done for instance with the life-cycle events or through classic component breakpoints (respectively watchpoints) . Component-specific breakpoints, similarly to thread breakpoints, ensure that the component which stops the execution is actually the component associated with the breakpoint.

7. CASE STUDY: DEBUGGING A PYRAMIDAL FEATURE TRACKER

This section presents a case study which illustrates the efficiency of our approach. We use an application executing the Pyramidal implementation of the Kanade-Lucas feature Tracker (PKLT) based on NPM components and running on the X86/POSIX simulator of P2012.

Feature tracking consists in identifying interesting points (features) in an initial image and following their motion in the subsequent images (tracking). Figure 4 presents a visual representation of several features tracked from one image to another. Bouguet explains the algorithm in [3], which is divided into two parts. First, the images are subsampled, with different ratios, to create a pyramidal representation. The bottom of the pyramid is the largest image, the top is the smallest. Then, the feature tracker is applied iteratively to the different levels of the pyramid.

The remainder of this section starts with an overview of the application implementation. Then, it details how our debugger presents the information of the current architecture to the developer. Finally, it explains how we leveraged our debugger to detect a communication bug in the application implementation.

7.1 Application Implementation

The scenario that we considered in this case study consists of a PKLT feature tracking between two images with, for the sake of simplicity, a two-level pyramid. The application is implemented with two types of components plus the host-side task:

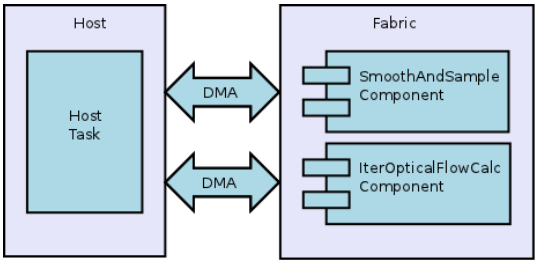

Figure 5: PKLT component architecture

SmoothAndSample is in charge of creating a new pyramid level. It receives the $n-1^{th}$ level image as input and returns the n^{th} level image.

IterOpticalFlowCalc performs a feature tracking between two images. It expects *previous* and *next* images in input, as well as the previous level features. It returns the new feature tracks.

Host side is in charge of the coordination of the components and programs the DMA controller to transfer the images to and between the components.

Both components execute their core algorithm on up to sixteen share-memory processing elements of a P2012 cluster. Figure 5 presents a schematic representation of this architecture.

7.2 Debugger Representation of the Architecture

In order to see the live deployment state of the application, developers need to stop the execution at an interesting location. For instance when the first component execution is triggered, that is, with the `run` event. The excerpt below[5] also contains the framework initialization, (i.e., when the `host` component is instantiated), as well as the instantiation of the `SmoothAndSample` component.

```
(gdb) component catch run
Catching components 'run' events
(gdb) run
...
[New component instantiated #1 Host[31272]]
...
[New component instantiated #2 Component[Smooth..]]
...
[Stopped on 'run' method of #2 Component[Smooth..]]
```

Then developers can list the currently known components, along with their interfaces (`info components +itf`). There are two components in the excerpt below, the `host` and a `SmoothAndSample` component. Developers can also list the inter-bindings, per component (`info connections`) or per link (`info links`). They will learn that the components are connected through two interfaces (the DMA links).

```
(gdb) info components +itf
  #1 Host[31272]
        Name.   Id.
```

[5]We assume that the reader is familiar with GDB command-line syntax. Otherwise, please refer to [13].

Figure 4: Feature tracking between two images

```
        DMAPush/0x8050f7c
        DMAPush/0x8050fbc
...
* #2 Component[SmoothAndSampleProcessor.so]
        Name.             Type.
        srcPullBuffer     (PullBuffer)
        dstPushBuffer     (PushBuffer)
...

(gdb) info connections
Interface.    Link.   Remote Itf.   Remote Comp.
#1 Host[31272]
DMAPush/0x... DMALink srcPullBuffer Component... #2
DMAPull/0x... DMALink dstPushBuffer Component... #2
 ...
#2 Component[SmoothAndSampleProcessor.so]
srcPullBuffer DMALink DMAPush/0x... Host[31272]
dstPullBuffer DMALink DMAPull/0x... Host[31272]
...

(gdb) info links
Link Interface                  Component
#1 DMALink
     DMAPush/0x8050f7c           Host[31272]
     srcPullBuffer (PullBuffer) Component... #2
#2 DMALink
     DMAPull/0x8050fbc           Host[31272]
     dstPushBuffer (PushBuffer) Component... #2
...
```

The following section describes how we used our debugger to discover the location of a bug in this application.

7.3 Data Transfer Error

During the validation process of the application, the test suite reported that the features reaching or leaving the *bottom* of the images coming from *some* cameras were not correctly tracked.

In order to understand how this bug was localized, we need to detail the algorithm of the `SmoothAndSample` component. This component is in charge of creating a new level of the image pyramid. It receives the input image line by line from its `srcPullBuffer` interface, applies a first "horizontal" parallel filter and sends it temporarily to the host through the `dstTmpPushBuffer` interface[6], because of memory constraints. Then the temporary image is retrieved again from the `srcTmpPullBuffer` interface, and a column by column "vertical" parallel filter is applied. The columns of the output image are pushed to the `dstPushBuffer` interface right after they have been processed.

The `srcPullBuffer` and `dstTmpPushBuffer` interfaces in the first part of the algorithm, and the `srcTmpPullBuffer` and `dstPushBuffer` interfaces in the second part are respectively supposed to be invoked in a lock-step fashion. All the data incoming in the `source` interface are processed and sent to the `destination` interface.

In order to verify this assumption, we set a breakpoint on component `destroy` events and executed the application until the first component destruction:

```
(gdb) component catch destroy
Catching components 'destroy' events
(gdb) run
...
[Stopped on 'destroy' event of #2 Component...]
```

Then, we asked the debugger to list the message counters of the component interfaces, and noticed that the figures did not follow the expectations. Each interface was supposed to process 35 messages. The `dstTmpPushBuffer` interface received one unexpected message, whereas `dstPushBuffer` was lacking one.

```
(gdb) info components +counts
~ #2 CommComponent[SmoothAndSampleProcessor.so]
    srcPullBuffer #35 msgs
    dstTmpPushBuffer #36 msgs
    srcTmpPullBuffer #35 msgs
    dstPushBuffer #34 msgs
```

Once this condition was noticed, it was straightforward to locate and fix the default in the code: the image size is divided evenly between the processors and, when the size cannot be entirely divided, the remainder part is processed in sequence. In our scenario, the bug was located in this remainder handling: the last message was sent to the temporary interface (`dstTmpPushBuffer`) instead of the final one (`dstPushBuffer`).
This bug would have been tricky to detect without our debugger. First, the developers would have had to investigate both of the components, as there was no obvious way to guess the faulty component. Then, only a precise code reading, maybe with the step-by-step capabilities of a traditional debugger, would have highlighted the issue.

In order to further validate the usefulness of our approach, it would be interesting to provide application developers

[6]The temporary interfaces we removed from the excerpts in the previous part for readability reasons.

with our debugger at the beginning of the development process. Indeed, when we started PKLT study, it already passed the main validation tests. Key problems in the component architecture usually arise in the early steps of the development, and it would be interesting to know how our tool would help developers to solve such issues, in comparison with classic debuggers.

8. CONCLUSION AND FUTURE WORK

In this article we introduced our new approach to debugging component-based applications interactively. We noticed two drawbacks in the available tools. First, current debuggers do not account for the concepts introduced by programming models (except at system level), such as the components and their interactions. Second, traditional language-level static debuggers are of little help when debugging complex and dynamic component-based applications developed for MPSoC systems.

We presented the main issues faced by developers when they try to debug such applications, which are related to the dynamic architecture, the components interactions and the complex flow of the information over the components. We described our contribution to solve these problems, an approach where debuggers are able to provide developers with high-level abstractions related to the component programming model. Based on the detection and interpretation of key events in the execution of the component framework, our approach allows the debugger to dynamically discover component instantiations, bindings, deployments and interactions. It also allows the computation of a live representation of the component interconnections. Thus, the approach offers to developers a better overview of the current state of dynamic applications. It also allows setting breakpoints on specific components, on their interfaces or on their life-cycle events. We went one step further by describing how debuggers should represent the messages carried throughout the components. This information helps developers to better understand the route followed by the application message. It also allows the control of the execution flow based on the activities of these messages.

Then we described how we applied this approach to an industrial parallel component framework of the MPSoC system *Platform 2012*, developed by STMicroelectronics and CEA. We presented the different parts of the framework API, along with the associated debug capabilities. We also detailed how we developed a proof-of-concept debugger as an extension of GDB, the free debugger of the Gnu project. GDB is frequently used to debug applications running on embedded systems. Based on its Python API, we introduced the concept of *function breakpoints* as extensions of the traditional breakpoints. These breakpoints are tailored to the framework function they monitor and update the debugger internal representations each time their function is executed.

Finally, we presented a case study that illustrates the debugging of an image processing application. We detailed the scenario of a faulty execution, which would have been complicated to understand with traditional tools, and highlighted how our approach facilitated the identification of the problem and allowed us to figure out where the defect was located in the code.

Interactive debugging of complex applications based on their programming model appears to be a promising direction. In future work, we will investigate how the idea of dynamically reconstructing the programming model based on framework events can be applied to different models. We expect our debugger to be able to easily encompass them, thanks to a generic code base. Using GDB as the low-end debugger should also help other developers to adapt our tool to their environment.

9. REFERENCES

[1] Allinea Software. Parallel Debugging is Easy, 2008.

[2] S. M. Balle, B. R. Brett, C.-P. Chen, and D. LaFrance-Linden. Extending a Traditional Debugger to Debug Massively Parallel Applications. *J. Parallel Distrib. Comput.*, 64:617–628, May 2004.

[3] J.-Y. Bouguet. Pyramidal Implementation of the Lucas Kanade Feature Tracker. Description of the Algorithm. `http://robots.stanford.edu/cs223b04/algo_tracking.pdf`, 2000.

[4] J. Cownie and W. Gropp. A Standard Interface for Debugger Access to Message Queue Information in MPI. In *Proceedings of the 6th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 51–58, London, UK, 1999. Springer-Verlag.

[5] I. Crnkovic. Component-Based Approach for Embedded Systems. In *Ninth International Workshop on Component-Oriented Programming*, June 2004.

[6] Free Standards Group. The DWARF debugging standard. `http://dwarf.freestandards.org/`.

[7] H. Jifeng, X. Li, and Z. Liu. Component-Based Software Engineering – The Need to Link Methods and their Theories. In *Proc. of ICTAC05, International Colloquium on Theoretical Aspects of Computing, Lecture Notes in Computer Science 3722*, pages 72–97. Springer, 2005.

[8] D. Lea. A java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*, JAVA '00, pages 36–43, New York, NY, USA, 2000. ACM.

[9] Message Passing Interface Forum. MPI: A message-passing interface standard, 1994.

[10] J. Ressia, A. Bergel, and O. Nierstrasz. Object-centric debugging. In *In Proceeding of the 34rd international conference on Software engineering, ICSE '12*, 2012. To appear.

[11] B. Schaeli, A. Al-Shabibi, and R. D. Hersch. Visual Debugging of MPI Applications. In *Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 239–247, Berlin, Heidelberg, 2008. Springer-Verlag.

[12] STMicroelectronics and CEA. Platform 2012: A Many-core Programmable Accelerator for Ultra-Efficient Embedded Computing in Nanometer Technology. `http://www.cmc.ca/WhatWeOffer/~/media/WhatWeOffer/TechPub/20101105_Whitepaper_Final.pdf`, 2010.

[13] The Gnu Project. GDB. `http://www.gnu.org/software/gdb/`.

[14] T. Wilde and J. A. Kohl. Port Monitor: A Monitoring & Debugging Approach For Component Frameworks. In *Workshop on Component Models and Frameworks*

in High Performance Computing (CompFrame 2005), Atlanta GA, 2005.