Programming-Model Centric Debugging for Multicore Embedded Systems

Kevin Pouget

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel : 7 août 2006

Présentée par

Kevin Pouget

Thèse dirigée par **Jean-François Méhaut**
et codirigée par **Luis-Miguel Santana-Ormeno**

préparée au sein du **Laboratoire d'Informatique de Grenoble**
et de l'**École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

Programming-Model Centric Debugging
for Multicore Embedded Systems

Thèse soutenue publiquement le **3 février 2014**,
devant le jury composé de :

M. Noël DE PALMA
Professeur à l'Université Joseph Fourier, Président
M. Radu PRODAN
Associate Professor à l'Université d'Innsbruck, Rapporteur
M. François BODIN
Professeur à l'IRISA, Rapporteur
M. Rainer LEPEURS
Professeur à l'Université RWTH Aachen, Examinateur
M. Jean-François MÉHAUT
Professeur à l'Université Joseph Fourier, Directeur de thèse
M. Luis-Miguel SANTANA-ORMENO
Directeur du centre IDTEC à STMicroelectronics, Co-Directeur de thèse

ACKNOWLEDGMENTS

Well, there we are, in a few days I'll defend my PhD thesis. I remember quite well when in 2004, 10 years ago already, I chose to apply at Toulouse's IUT to get a two-year degree. I wasn't sure I would enjoy longer studies! And again with the Master degree, I took the "professional" path for the same reason. But after a gap year in England, I quickly realized that the university benches are actually quite attractive, and I applied for a PhD candidate position in Grenoble.

I would like to thank warmly my two advisers, Miguel Santana and Jean-François Méhaut, for that great opportunity they offered me. Miguel thrust me from our very first phone interview in the backyard of London's Paddington Hospital, where I worked at that time. The only doubt he expressed was whether I would be able to stay at the same place for three years :) The region of Grenoble and the PhD work proved to be successfully combination, as I even plan to stay here for the next years! Jean-François also supported me from the beginning of this work, and did a great job helping me to recognize and put forward the scientific aspect of my work. Thank you both once again.

Besides, I would like to thank the rest of my thesis committee: Noël de Palma gave me the honor of chairing the jury; Radu Prodan and François Bodin accepted to review my dissertation and made insightful remarks about it; and Rainer Leupers accepted to assess my thesis defense. I am really grateful to you all for the attention you put on my work.

I also wanted to thank my colleagues, offices-mates and friends, in particular Jan & Patricia, Giannis, Marcio, Serge, Naweiluo, and the teams at ST and the lab, with whom I enjoyed talking about science and computer science, but also mountains, flying machines, hiking and skiing, debugging, bugs and free software ... I must recognize that it sometimes spread over work-time, but would I have been more productive without these chit-chats? I hardly think so!

Finally, I wanted to thank my family, and my step-family, for their support and help all along the last three decades (almost). That's amazing you literally spread over the world during that PhD time: Japan, Brazil and Marquesas Islands while I was here, playing with bits and bugs :) Last but not least, I wanted to thank my love Marine, who made me the honor of becoming my wife almost three years ago. Thanks for helping me during all that time, thank you very much.

Contents

List of Figures

List of Tables

ACRONYMS

ST	STMicroelectronics. 7, 11, 13, 25, 29, 50, 61–63, 74, 94, 109, 125, 133, 137, 139
STHORM	ST Heterogeneous Low Power Many-core. 13, 14, 17, 22–24, 63, 65, 70, 78, 83, 85, 87, 94, 95, 108, 136, 137
MCGDB	Model-Centric GDB. 65–67, 69, 70, 72, 73, 75, 78, 79, 81, 85, 86, 91–101, 103–105, 107–109, 136, 137
MPSoC	Multi-Processor-System-on-a-Chip. 1–3, 5, 6, 8, 11–15, 17, 24, 25, 27, 29, 30, 33, 36, 44, 45, 63, 75, 83, 108, 113–115, 123–125, 131–135, 137–139
AMP	Asymmetric Multi-Procesor. 13
API	Application Programming Interface. 15, 23, 33, 36, 52–55, 61, 63, 70, 75, 79, 81, 82, 114, 116, 120, 121, 126, 130, 140
CBSE	Component-Based Software Engineering. 18
CISC	Complex Instruction Set Computing. 11
CPU	Central Processing Unit. 6, 7, 19, 21, 24, 75, 78, 99, 101, 123
DMA	Direct Memory Access. 14, 67, 69, 87, 90, 91, 95
DSP	Digital Signal Processor. 1, 11, 24
FIFO	First In First Out. 67, 69, 113
GPGPU	General-Purpose Graphical Processing Unit. 3, 14, 15, 18, 21, 24, 75, 115, 117, 118, 124, 138
GPU	Graphical Processing Unit. 14, 24, 43, 78, 81, 82, 99, 101, 106, 108, 117, 137
HPC	High-Performance Computing. 3, 8, 14, 36, 75, 99, 113, 115, 137
IDE	Integrated Development Environment. 125, 140
IP	Intellectual Property. 14, 113
ISA	Instruction Set Architecture. 13
JTAG/TAP	Joint Test Action Group/Test Access Port. 114

MMU	Memory Management Unit. 114
MPI	Message-Passing Interface. 16, 106, 116, 121, 126, 140
NoC	Network-on-Chip. 113
NPM	Native Programming Model. 23, 24, 65–67, 69, 70, 85, 88
OpenCL	Open Computing Language. 16, 21, 23, 24, 29, 55, 65, 75, 78–83, 99–101, 103, 104, 106–108, 120, 121, 130, 136
OS	Operating System. 3, 14–17, 21, 51, 66, 113, 114
PEDF	Predicated Execution Dataflow. 23, 24, 61, 65, 70, 71, 73–75, 83, 94, 96, 109
RISC	Reduced Instruction Set Computing. 11
RTL	Register Transfer Level. 74, 83
SIMD	Single Instruction Multiple Data. 14, 36, 44, 134
SMP	Symmetric Multi-Processor. 13
SoC	System-on-Chip. 113
SPMD	Single Program Multiple Data. 14
TLM	Transaction-Level Modelling. 114
VHDL	VHSIC Hardware Description Language. 113

1

INTRODUCTION

Nowadays, consumer electronics devices become more and more ubiquitous. With the new generation smartphones, tablets, set-top boxes and hand-held audio/video players, multimedia embedded systems are spreading at a fast pace, with a constantly growing demand for computational power.

During the last decade, Multi-Processor-Systems-on-a-Chip (MPSoCs) have been introduced to the market to answer this demand. These systems-on-a-chip typically feature general-purpose multicore processors, but also clusters of domain or application-specific processors (*e.g.,* Digital Signal Processors(DSPs)) or lightweight processing elements. These processors can have different instruction sets (also known as *heterogeneous* computing), which allows manufacturers to optimize their micro-architecture to perform specific tasks. Besides, such chip designs allow platforms to offer high computation performance while maintaining low power consumption.

However, a significant drawback counterbalances the appealing capabilities of MPSoCs. Indeed, although multicore heterogeneous programming can provide a solution to the computational needs, it also increases development, verification and validation complexity. In the embedded system industry, these aspects are key to maintain a good time-to-market. Hence, it is crucial for companies to lower as much as possible their impact.

With respect to the development aspect, programming models provide well-studied guidelines, communication algorithms and architecture designs to draw application specifications. In the context of MPSoC programming, relying on such models can help developers not to reinvent the "*parallel computing wheel*" and improve development time. Furthermore, at implementation time, the code implementing these low-level and platform-specific structures can be bundled into application-independent libraries. Reusing such libraries also contributes to reduce the time-to-market. In the following, we refer to these libraries as *supportive environments*.

However, MPSoC parallelism also exacerbates the challenges of application verification and validation. Concurrency in execution flows introduces bugs which could not exist during a sequential execution, such as deadlocks and race conditions. Besides, supportive environments can twist and bend the execution flows to follow the programming model directives. Hence, the overall application and runtime libraries complexity makes it particularly hard to pinpoint problems and understand their root cause.

We continue this chapter with an introduction to embedded systems and MPSoCs (Section 1.1), then we present the general domain of this thesis: software verification, validation and debugging (Section 1.2). We carry on in Section 1.3 with an overview the debugging challenges specific to MPSoCs. After that, we jump to the core of our topic and present in Section 1.4 the general objectives of this thesis. We finish this chapter with a description of the scientific context in which we carried out this research work (Section 1.5) and an outline of the organization of this manuscript (Section 1.6).

1.1 EMBEDDED SYSTEMS AND MPSOCS

In 1992, the IEEE proposed the following definition of an embedded system:

> "A computer system that is part of a larger system and performs some of the requirements of that system; for example, a computer system used in an aircraft or rapid transit system."
>
> IEEE, 1992

The popularity of such systems has increased at a high pace during the last decade, in particular with consumer devices that are getting "*smarter and smarter*": engine regulators in the automotive industry, smartphones, hand-held game consoles, set-top boxes, ...

In all these devices, a *computer* sub-system is in charge of at least one part of the overall requirements. It may be a narrow aspect, like the engine regulator compared to an entire automotive, or it may be more prominent, such as the new generation smartphones which nowadays appear similar to general-purpose computers.

All these computer systems share an important characteristic: they operate in constrained environments. These constraints can take different aspects: a limited power supply can entail processor frequency limitations and/or a micro-architecture focused on energy efficiency rather than raw speed; physical space and chip costs are also part of the balance, bringing limitations in the memory space size (volatile and non-volatile) and available input and output systems (screen, keyboard, mouse, *etc.*).

Aside from these constraints, embedded systems are not that different from their general-purpose counter-parts: one or multiple processors read data from memory, execute instructions and write back the result, if any. Hence, application development for embedded systems appears similar to classic programming, except that developers must keep into consideration *device*-specific requirements.

Along that, and similarly to general-purpose computers, another urge emerged in embedded systems during the last decade: applications require high performance to be able to provide advanced 3D capabilities or support high-definition video standards. And to answer this need, embedded system manufacturers began to increase the number of processors and cores available to software applications. They first augmented the number of processors on the board, but in recent years, they have started to adopt

a new design: MPSoC. Here, processors are not physically independent from one another, but rather all integrated on the same chip. This design strongly reduces the energy consumption of such processors, as well as the heating factor. Inter-processor communication is also faster, as data remain inside the chip instead of circulating over the board's internal interconnection network.

However, an important distinction separates MPSoC parallelism from mainstream computers: the processors heterogeneity. As embedded systems aim at performing a very specialized service, they do not require a general-purpose computation power, but rather a task-specific one. We can illustrate this idea with a mobile-phone featuring 1/ a digital-signal processor to communicate with the wireless network, 2/ a multimedia processor to play music and videos, and 3/ a more generic one to run the Operating System (OS) and user applications.

Orthogonally, MPSoCs have also attracted a new kind of consumers towards embedded systems: High-Performance Computing (HPC) manufacturers. Indeed, the race for performance of general-purpose computers is facing a strong and tall barrier: the energy wall. Current HPC computers can provide up to 33 petaflop/s[1], at the energy price of $18MW$. Considering the next target of one exaflop per second, a linear extrapolation of the electric consumption sets the bill around $30 \times 18 = 540MW$. This figure is equivalent to the energy produced by a nuclear power plant reactor[2]. Hence, emergent HPC solutions are starting to consider energy-efficient processors such as MPSoCs.

For similar reasons, General-Purpose Graphical Processing Units(GPGPUs), the processors of high-end graphics card, are also more and more present in HPC. For instance, the current TOP500.ORG nº2 supercomputer (and previously nº1), TITAN[3], is powered by more than 260,000 NVIDIA K20X GPGPU accelerator cores.

We will see in the next chapter (Chapter 2, Section 2.1.1) that these two device families share similar characteristics, both at the energy level and regarding their programmability. They mainly differ in the number of processing elements they feature, and their degree of independence.

From a software perspective, programming such parallel and heterogeneous systems require new development tools, well-tailored to their particular hardware characteristics. Hence, academic and industrial researchers study how to adapt existing programming methodologies to these requirements(*e.g.*, component or dataflow programming) or develop new ones (*e.g.*, kernel-based accelerator programming). We further discuss these examples in the following chapter and throughout this document.

1 Tianhe-2 (MilkyWay-2), Top500.org nº1 (June 2013), http://www.top500.org/system/177999
2 U.S. Energy Information Administration, http://www.eia.gov/tools/faqs/faq.cfm?id=104&t=3
3 TITAN, TOP500.ORG nº2 (June 2013) http://www.top500.org/system/177975

In the following section, we introduce the problem complementary to embedded system programming, that is, how to ensure the correctness of such applications.

1.2 EMBEDDED SOFTWARE VERIFICATION AND VALIDATION

Verification and validation is a crucial aspect of software development. Indeed, an application not following its specification is of limited interest. But in some particular environments, consequences may rise rapidly. In the domain of embedded systems, we can find numerous examples of life-depending, costly and/or hardly accessible devices: pace-makers, automotive industry, satellites, ... For such critical cases, software elements must provide a high-level of correctness and reliability guarantees.

In different circumstances, such as consumer electronics, the need for correctness appears more as a business requirement. In the embedded system industry, the time-to-market is an important constraint for products to be successful. If a company releases its device *before* competitors, it has a higher chance of adoption. In parallel to this aspect, software correctness is also essential to ensure a good user experience. Applications showing repetitive crashes, lags, or any kind of unexpected behavior will be heavily criticized by its users.

Verification and validation aims at limiting these artifacts. It is a large research topic, ranging from static analysis to execution profiling, through provable programming models, compiler verification and interactive debugging. It encompasses a wide set of skills and abilities, for both developers fixing their applications and computer scientists developing the tools and methodologies.

We can distinguish three stages of application debugging techniques: pre-execution, live and *post-mortem*, which depend on the moment when the verification is done, as illustrated with Figure 1.1. Let us explain this distinction with a simple example of a C code producing a segmentation fault[4]:

```
if (*i == NULL) { *i = x; }
```

Pre-Execution Analysis consists in analyzing the source code to detect potential problem. In our example, analyzing the possible values of `i` would highlight that the variable can only be `NULL` in the affectation.

Live debugging consists in analyzing the processor and memory state during the execution. Here, the process would terminate (segmentation fault) on the affectation. Printing the value of variable `i` would reveal that its value was `NULL`.

Post-mortem debugging consists in instrumenting the source-code to gather execution information. This instrumentation can be implicit, through hardware module or during compilation, or explicit with tracing statements. In our example, an explicit

4 This example assumes a GCC compiler and Linux kernel; see [WCC+12] for more detailed information about what can happen with C undefined behaviors and null pointer dereference.

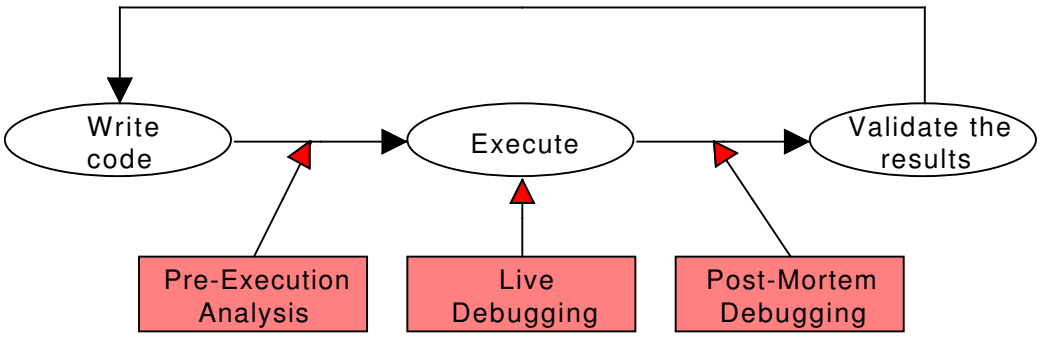


Figure 1.1: Three Stages for Application Debugging

instrumentation would involve logging the value of `i` before the assignment, and parsing the execution trace to spot its invalid value.

Hence, the purpose of verification and validation, and debugging in particular, is to locate and fix "*bugs*" from the source code. But before going further, we need to make explicit the definition of a *bug*, as the term is rather colloquial. In [Zel05], *Zeller* dissected the precise meanings behind it and proposed a more explicit terminology:

1. The programmer creates a *defect* in the source code, by writing incorrect instructions.
2. When this defect is executed, it causes an *infection*, that is, the program state differs from what the programmer intended.
3. The infection *propagates* in the program state. It may also be overwritten, masked or corrected by the application.
4. The infection leads to a *failure*, that is, an externally observable error or a crash.

In this document, we refer to *bugs* when the distinction between the different aspects is not important. We use the notions of *errors/crashes* or *defects* to refer to visible problems and source code problems, respectively.

One component of software verification and validation is still missing before concluding this section: performance debugging. Indeed, timing and timeliness are important aspects of the non-functional part of software specification. In the context of MPSoC development, validating such constraints requires the usage of highly accurate platform simulators, or better real boards. It can involve different post-mortem techniques, such as profiling and trace analysis.

This aspect of application debugging is out of the scope of this document, as we chose to focus on *live* debugging. Indeed, this debugging technique, and in particular interactive debugging, severely disrupts the continuity of the execution and hence alters its time-related behavior. Incidentally, the bug terminology described above is also not adapted to performance debugging. Therefore, the work we present exclusively targets

functional debugging of MPSoC programming. We present this aspect with more details in the following section.

1.3 INTERACTIVE DEBUGGING OF MULTICORE EMBEDDED SYSTEMS

Interactive debugging consists in exploring, analyzing and understanding how an application is *actually* executed by the underlying platform, and confronting it with what the developer *expects* from the execution. The point where the two versions diverge may hide a code defect. Locating this divergence involves and requires a scientific reasoning: once developers notice an application error, they draw hypotheses on the infection point and its propagation path. Interactive debugging tools help them to verify or invalid these hypotheses by allowing program state inspection at different points of the execution. The tool used for that purpose is commonly known as a *debugger*, although this name is not completely meaningful. Indeed, debuggers are not primarily concerned with *finding* bugs, but rather helping developers to understand the details, subtleties and convolutions of the application execution. Debuggers allow developers to stop the execution under different conditions:

Breakpoint when a processor reaches a specific function, source-code line, assembly instruction, or memory address,

Watchpoint when the code tries to read or write a particular memory location,

Catchpoint another possibility is to "*catch*" system events occurring in the application, such as system calls or UNIX signals.

Once the debugger has stopped the execution, it gives the control back to the developer, who tries to understand the exact application state. In order to do this, the debugger provides a set of commands to inspect the different memory locations. With the help of debugging information provided by the compiler (usually embedded in the binary file), the debugger can display the values of the memory bits in the relevant format (that is, as an integer, a character, a structure, *etc.*). These memory bits can come from different locations: a local variable in the call stack, a global variable, a Central Processing Unit (CPU) register, a made-up address or a computed value.

However in applications for multicore embedded systems, a showstopper hinders the road of interactive debugging: supportive environments' runtime libraries drive an important part of the execution. From the debugger users' point of view, this means that they will not be able to control their applications as seamlessly as for standard applications (*i.e.*, applications where developers can control and access the *entire* source code). Indeed, runtime libraries manipulate the execution flow(s), for instance to schedule multiple entities on a single processor core or to transmit and process messages. The semantics of these operations goes *beyond* the traditional assembly-based capabilities of current interactive debuggers.

In the following section, we present the general objectives of this thesis in order to lighten the challenges faced by application developers of multicore embedded systems and help them to locate the problems in their applications more easily.

1.4 OBJECTIVES OF THIS THESIS

We believe that interactive debugging can provide a substantial help for the development and refinement of applications for multicore embedded systems. Indeed, although high-level programming models simplify application development, they cannot systematically guarantee the correctness. If some models allow extensive compile-time verification, such benefits come at the price of strong programming constraints and a reduced expressiveness. On the other side, models supporting a larger set of algorithms, and especially dynamic behaviors, usually cannot provide such guarantees.

For these models, interactive debugging stands as an interesting alternative. It offers the ability to control and monitor application execution with various granularities (source code, machine instructions, CPU registers, main memory, ...), which would be impossible to achieve with other approaches.

However, in its current state, interactive debugging is not adapted yet to debug applications relying on high-level programming models. Source-level debugging has evolved to support multiple flows of execution and inspect their memory contexts, however the semantics of debugging commands has remained the same as for sequential applications: exclusively based on processor control and symbol handling (breakpoints, step-by-step execution, printing memory locations/registers/variables, *etc.*).

Our objective in this thesis is to move the abstraction level of interactive debugging one step higher in the application representation. Thus, it would meet the abstraction level used by developers during application design and development, that is, the representation defined by the programming model.

We expect these improvements to relieve developers from the burden of dealing with uncontrollable (with current interactive debugging approaches) runtime environments. In addition to hindering experienced developers in their duties, unfitted debuggers also discourage junior programmers to use such tools to tackle their problems, because of the steep learning curve.

Finally, we intend to describe generic guidelines to facilitate and encourage the development of similar high-level debuggers for different programming models. Indeed, we believe that having a unified set of high-level debugging tool would help developers to switch from one model to another more easily.

1.5 SCIENTIFIC CONTEXT

This thesis was funded by a CIFRE ANRT partnership between STMicroelectronics (ST) and the LIG (*Laboratoire d'Informatique de Grenoble*) laboratory of the UNIVERSITY OF GRENOBLE, France. The research work was carried out in ST's IDTEC (Integrated

Development Tools Expertise Center) team, whose role is to provide its customers with development and debugging tools tailored to the company's embedded boards. This thesis work, essentially industry-oriented, was directly part of the team's mission.

The scientific and academic part of the thesis has been carried out in the NANOSIM (Nanosimulations and Embedded Applications for Hybrid Multi-core Architectures) team of the LIG. NANOSIM's research targets the integration of embedded systems into HPC environments. In this regard, they are interested in energy-efficient MPSoCs, how to develop applications for such architectures, and eventually how to refine the code to exploit the boards' performance optimally.

1.6 ORGANIZATION OF THE THESIS

The rest of this document is divided into three parts, as follows:

- Part I introduces the tools and methods to develop applications for MPSoC systems and presents the state-of-the-art of software debugging in this context (Chapter 2). Then, we detail the generic principles of our contribution, programming-model-centric debugging, and we explain how it applies to different programming models used for MPSoC development (Chapter 3).
- Part II presents a practical study of model-centric debugging: we first provide building blocks for the development of such a debugger (Chapter 4), then we describe how we transformed our abstract, model-level debugging propositions into actual tools, to debug applications running in different programming environments for an industrial MPSoC (Chapter 5). Lastly, we study the benefits of model-centric debugging in the context of industrial application case-studies (Chapter 6).
- Part III finally reviews the literature related to the debugging of embedded and multicore applications (Chapter 7), draws the conclusion and details the future work of this thesis (Chapter 8).

Part I

Debugging Multicore Embedded Systems with Programming Models

2

PROGRAMMING AND DEBUGGING MULTICORE EMBEDDED SYSTEMS

Setting the Stage:
Context, Background and Motivations.

In comparison with general-purpose computers, multicore embedded system architectures appear to be more diversified. Their task-specific nature blurs away the need for hardware and software compatibility that we are used to see in general-purpose computing. Likewise, their environmental constraints, for instance energy limitations or cost pressure, led to shifts in the design of their internal microarchitecture: instead of the general-purpose Complex Instruction Set Computing (CISC), 64 bits x86 processors, embedded systems tend to favor Reduced Instruction Set Computing (RISC) processors, like ARM processors or ST's STxP series (see next section for details). Furthermore, in order to meet applications' performance expectations with a limited energy consumption, multicore embedded systems started to incorporate specialized hardware processors such as DSPs or even dedicated circuits [Wol04].

In both general-purpose and embedded computing, programming *multicore* processors is well-recognized as a difficult task. In order to gather and reuse the design and algorithmic knowledge gained over the years, the good practice of relying on programming models and supportive environments has emerged in parallel application development. Respectively at design and implementation time, these abstract models and their coding counter-parts provide developers with well-studied implementation building-blocks for the development of their parallel applications. They also offload application developers from programming the low-level and deeply architecture-specific aspects of their programs.

However, debugging such multicore applications is notoriously more difficult than sequential applications, and it gets even worse with MPSoC heterogeneous parallelism. One reason for the increase of difficulty is that concurrent environments bring new forms of problems in the bug taxonomy, which do not exist in sequential codes. We can cite for instance deadlocks or race conditions. The former corresponds to a situation where a set of tasks are blocked, mutually waiting for data from one another and the later stands for a data dependency defect, where the infection and failure are only triggered in specific and non-deterministic task scheduling orders. These conditions, although still a frequent problem for multicore application programmers, undergone heavy literature studies during the last decades [LPSZ08]. We will rather focus on less

specific problems, such as code that do not follow their specifications, and how to help developers to better understand the details of the application execution.

In this chapter, we review the key elements required to understand the background of this thesis. We first detail in Section 2.1 the specifications of the embedded systems we target, as well as the concepts of programming model and supportive environment. Then, we illustrate these notions in Section 2.2 with the description of three programming models and present how our target platform supports them. Our latter analyses and experimentation rely on this programming ecosystem. Finally, in Section 2.2, we study the current state-of-the-art of MPSoC application debugging. We first explain the challenges faced while debugging such applications, then we present the tools currently available and point out their limitations.

2.1 MPSOC PROGRAMMING: HARDWARE AND SOFTWARE TERMINOLOGY

In this section, we detail the hardware and software terminology used in this thesis manuscript. We start at hardware level with a presentation of embedded systems featuring multicore and manycore processors. Then, from a more general point of view, we explain our vision of programming models. Finally, we reach the software level and detail the notion of supportive environments. The concepts of programming models and environments are frequently used in computer science and software development, but their exact signification varies among the different communities.

2.1.1 *Multicore, Manycore and MPSoC Systems*

MULTICORE AND HETEROGENEITY

Nowadays, it is well-accepted that the increase of processor's computing performance has hit a wall. Currently, there exists a threshold around 20-30nm of lithography resolution that prevents micro-transistor frequency increase while maintaining an acceptable heat factor. Table 2.1 presents a comparison of three recent general-purpose and embedded processors, where we can notice the current limits in terms of microprocessor lithography and frequency.

	Lithography	Frequency	Release date	# of cores
INTEL XEON E5-1660 V2[1]	22nm	3.7GHz	Q3 2013	6
AMD FX 9590[2]	32nm	4.7GHz	Q3 2013	8
ARM CORTEX A9[3]	45nm	2GHz	2009	4

Table 2.1: Processor Comparison.

Note 1 http://ark.intel.com/products/75781

Note 2 http://www.amd.com/us/products/desktop/processors/amdfx/pages/amdfx-model-number-comparison.aspx

Note 3 http://www.arm.com/files/pdf/armcortexa-9processors.pdf

To counter the lack of raw performance improvements, processors started to grow "*horizontally*", instead of "*vertically*". That is, by increasing the number of cores on a single processor die. Hence, multicore processors have flourished in personal computers, which now frequently feature bi, quad or even octo-core processors, as shown in Table 2.1.

This trend has also reached embedded devices, but with additional constraints such as reducing costs and energy consumption. Indeed, providing a dedicated processor for voice processing, another for the video camera, a third one for the user interface, *etc.* can improve the overall performance drastically.

However, a sharp distinction separates these two aspects of multi-processing. Personal workstations usually feature Symmetric Multi-Processor (SMP), whereas embedded system have Asymmetric Multi-Procesor (AMP), or *heterogeneous* processors (we use this term in the rest of the document).

In SMP architecture, all the processors have the same micro-architecture, or at least a common Instruction Set Architecture (ISA) and share the memory address space. On the other hand, in AMP or heterogeneous multi-processing, the processors are not uniform and hence cannot share the whole memory space.

Multicore MPSoCs

MPSoC systems can be classified as multicore or manycore processors, depending of their design goals. Boards targeting multimedia markets will favor designs with a limited number of cores, well-optimized to the application they will host; whereas those targeting intensive computation will tend to manycore designs.

ST Heterogeneous Low Power Many-core (STHORM) is an MPSoC industrial research platform which was developed in collaboration by ST and *CEA* [BFFM12, MBF+12]. It was formally known as PLATFORM 2012/P2012 until it recently reached product maturity. (ST first products featuring a STHORM subsystem will be available in early 2014.) We will use its design and software ecosystem as a reference design throughout this document, as the platform has been well-accepted in both academic and industrial embedded computing communities.

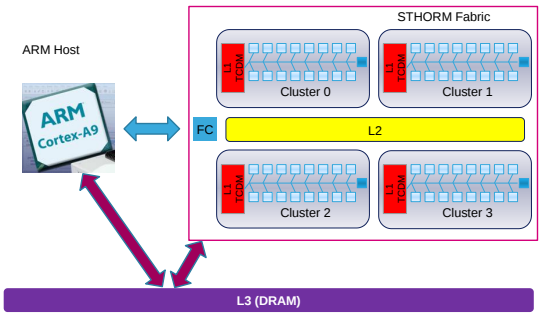


Figure 2.1: Internal Architecture of STHORM MPSoC System

Figure 2.1 presents the internal architecture of STHORM. The platform design originated from computing accelerators, with a general-purpose dual-core ARM CORTEX

A9-MP processor on one side, running a LINUX-based OS, and the *computing fabric*, which consists of clusters of up to 16 processing elements (*i.e.*, processor cores or dedicated hardware Intellectual Properties (IPs)) sharing their memory space on the other side. Figure 2.1 depicts four clusters, however the number can vary according to the design requirements. Additionally, each cluster can embed dedicated hardware IPs, specially crafted for a particular operation. We come back to this aspect later on (Chapter 6, Section 6.2) with an example of a H.264 video decoding application that exploits this capability.

Within a cluster, the 16 cores communicate through the shared L1 memory bank, and their execution is coordinated by a *cluster controller* core. Clusters can communicate with each other through a shared memory-space in the L2 memory bank; and they interact with the host processor through the L3 memory bank, interfaced by a Direct Memory Access (DMA) controller.

As we further detail in Section 2.2 with the description of STHORM's programming models and environments, applications running on the platform can exploit the host processor as well as all or some of the clusters. Inside a cluster, they can operate on the cluster controller and/or the processing elements. The programming environments provided with STHORM offer developers the ability to choose the model which will be the most adapted to their requirements.

STHORM MPSoC can run Single Program Multiple Data (SPMD) codes as its processor cores are autonomous and can execute simultaneously the same program at independent points. This is an interesting capability, in comparison with Graphical Processing Unit (GPU) processors, whose processor cores run in a lockstep Single Instruction Multiple Data (SIMD) fashion.

Manycore GPGPUs

Graphics card processors started to open up towards general purpose computing since the beginning of 2003 [NVi09]. At that time, the HPC community discovered that they could exploit GPGPU computing power for broader purposes than only rendering 3D and high-definition images on a monitor screen. These processors are massively parallel, as for example, NVIDIA's TESLA K20X and its 2688 cores, clocked at 732MHz[1].

However, we must balance this impressive figure with the fact that GPU cores are not completely independent from one another. They rather operate in a SIMD fashion, which means that *all* the cores (or a subgroup of them—we further expand on this aspect in Subsection 2.2.1) must execute the same instruction at the same time. Nevertheless, current GPGPUs are more flexible than antique SIMD computers, as they allow cores to follow different branches of a conditional test:

```
if (thread_id % 2)
  // do A
```

1 Nvidia Tesla K20X board specification. http://www.nvidia.fr/content/PDF/kepler/Tesla-K20X-BD-06397-001-v05.pdf

```
else
  // do B
```

In this situation, all the cores with an odd `thread_id` will execute branch `A`, while those with an even `thread_id` will be blocked. Upon branch `A` completion, the cores with an even `thread_id` will execute branch `B`, and the other group will be blocked. Hence, we can understand with this behavior that GPGPU processors still have a single instruction pointer, but they can disable instruction treatment upon specific conditions. The direct drawback of such code is that half of the processing power is lost during the conditional treatment.

Now that we have presented how MPSoC and GPGPU platforms are organized, we move forward to the software level and discuss how developers can program such complex architectures. In the following subsections, we present the notion of programming model and it embodiment as supportive environments. Programming models aim at providing developers with a high-level development interface, less complex and more generic than the underlying architecture.

2.1.2 *Parallel Programming Models*

Skillicorn and *Talia* provided in [ST98] an interesting definition for what they call a *model of parallel computation* (we refer to it as a *programming model*):

> A model is an abstract machine providing certain operations to the programming level above and requiring implementations for each of these operations on all of the architectures below. It is designed to separate software-development concerns from effective parallel-execution concerns and provides both abstraction and stability.

Instead of developing parallel applications based on raw hardware or OS primitives, developers can base their design and development on programming-model abstract machines. These high-level machines aim at simplifying software development by providing developers with well-studied programming abstractions, detached from the heterogeneity of hardware architecture families.

Programming models define an Application Programming Interface (API) which provides stability and separation-of-concern. Stability arises from the independence of the interface from the underlying implementation and hardware. Indeed, applications can benefit from their respective evolution and improvement without any code modification. Separation-of-concern allows software programmers to exclusively focus on the application implementation, whereas another team, specialized in OS and low-level programming, will be in charge of the abstract machine implementation. In the

context of parallel computing, programming models should also facilitate application decomposition into parallel *entities*[2] that run on distinct processors.

Programming models should also provide a software development methodology, in order to bridge the gap between the *semantic* structures of the problem and the actual structures required to implement and execute it.

In [Vaj11], *Vajda* distinguished two general families of programming models (or paradigms): those based on a *global state* and those that are *de-centralized*. In the former family, the program state can be fully characterized at any point in time. In the later, there is no global state, but rather autonomous and un-synchronized entities, reacting to requests coming from other components of the system.

These broad families can be divided again and again in sub-families. In the following section (Section 2.2.1), we will study two de-centralized models, which belong to the *task-based* family, and one global state, that belongs to the *data-parallel* family.

Under this perspective, we can affirm that programming models are key abstractions for the design of multicore applications. However, as the term "model" implies, at this stage they only provide *conceptual* tools. Supportive environments are in charge of making concrete the programming models' guidelines.

2.1.3 *Supportive Environments*

A *supportive environment* consists of the programming framework instantiating a particular programming model abstract machine, as well as the runtime system which drives the application execution on a particular architecture. It corresponds to the second part of the programming model definition: "*... requiring implementations of each of these operations on all of the architectures below*."

As mentioned earlier, parallel programming models need to facilitate application decomposition, thus their supportive environments are in charge of the mapping and scheduling of these entities onto the available processors. Hence, they also have to implement and optimize the communication operations defined by the model.

Supportive environments appear to the application as a high-level programming interface to the platform and its low-level software. In order to decouple software developers' work from environment implementers', this interface aims at being stable over the time, for instance through standardization (*e.g.*, Message-Passing Interface (MPI) [MPI94] or Open Computing Language (OpenCL) [Khro8]). They are often implemented through reusable libraries and frameworks, and optimized for the target platforms and execution conditions.

It is important to note that there is no strict boundary between a supportive environment and the computer OS, especially in the context of embedded computing. Indeed, as *Vajda* underlined in [Vaj11], an OS can be seen as *the abstraction layer on top of the*

2 The names *tasks* or *threads* also exist in the literature, but we avoid the latter which can be confused with OS threads of execution.

underlying hardware. Hence, it appears similar to a low-level supportive environment. The key distinction is that the role of OS is more oriented towards hardware resources sharing and management.

EXECUTION OF PROGRAMMING-MODEL BASED APPLICATIONS

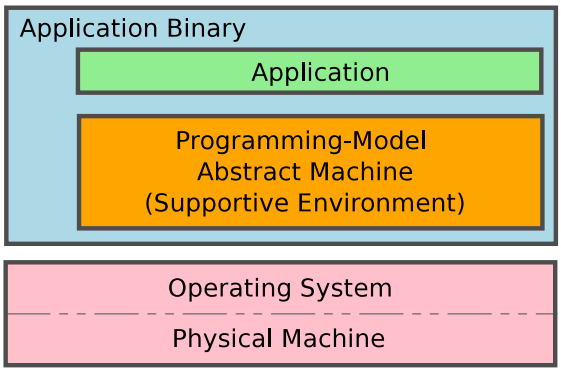


Figure 2.2: Organization of a Programming-Model-Based Application

Figure 2.2 presents an overview of the general organization of an application based on a programming model. We can distinguish three different layers:

1. The application itself, which consists of code written by developers to implement their algorithms in a given programming model and language.
2. The abstract machine defined by the programming model and implemented by the supportive environment.
3. The physical machine, abstracted through the OS.

During the execution, the boundary between the application layer and the supportive environment disappears. The OS is only aware of the surrounding box, the application binary, and the different processors used by the application (*i.e.,* the threads and processes it created). Hence, from a "*system*" point of view, there is no distinction between an application relying on one programming model or another, or even from an application exclusively relying on OS primitives.

In the following section, we introduce concrete examples of programming models and environments. We chose these examples as they are part of SThorm ecosystem, and hence offer three alternatives to program a single MPSoC system.

2.2 PROGRAMMING MODELS AND ENVIRONMENTS FOR MPSOC

Now that we have clarified the notions of programming model and environment, we exemplify these definitions with three case-studies. These models (Subsection 2.2.1) and environments (Subsection 2.2.2) are part of SThorm development toolkit, and provide alternative ways to exploit the platform. In the remaining of this dissertation,

we regularly come back to these models to illustrate how our model-centric debugging proposal applies to this industrial environment.

2.2.1 *Programming Models*

In this subsection, we first present Component-Based Software Engineering (CBSE). Strictly speaking, CBSE is not a programming model, but rather a software development approach. However, as far as we are concerned in this work on debugging, we assume we can blur away this distinction. Components are also based on the task model, however their interconnection network can change over the time.

Then, we continue our examples with dataflow programming, which is another task-based programming model that focuses on the flow of information between the tasks.

Finally, we introduce kernel-based accelerator programming, which radically differs from the two former models. In this data-parallel model, which stemmed from GPGPU computing, a "*host*" processor drives the computation, by preparing work to be executed on accelerator processors.

COMPONENT PROGRAMMING FOR MPSOC

In the component programming model [JLL05], the key task decomposition consists in *components* providing services to one another. The model and its development methodology underline that developers should design components as *independent* building blocks and favor reusability. Hence, the interface between components should be defined in a language-independent description file, and each component should provide an architecture description file, which formally specifies the nature of the services it *offers*, as well as those it *requires*.

Hence (or rather theoretically), developers should be able to develop component-based applications just by interconnecting the right components with one another. In practice, the *right* component may not be readily available, so developers have to build them themselves.

To express parallelism, components can provide a "*special*" service, whose interface definition is provided by the abstract machine: an interface for "*runnable*" components. With regard to the C programming language, we can compare this service to the `int main(char *argv, int argc)` function prototype. Similarly, the abstract machine will request this service execution at the relevant time and on a dedicated execution context.

Developers can also program dynamic reconfigurations of component interconnections, for instance to adapt the architecture to different runtime constraints, if the abstract machine supports it.

The abstract machine defined by this model is in charge of components' life-cycle management: deployment, bindings and, if relevant, mapping and scheduling. It also handles inter-component communications, which may be as simple as function calls, but also involve different processors and memory contexts.

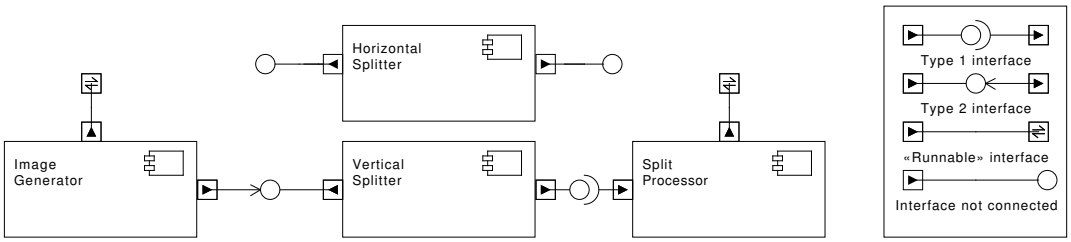


Figure 2.3: Component Programming Example.

Figure 2.3 presents an example of a component-based application. As the names suggest, the purpose of this application mockup is to generate an image, split it horizontally and process each of the chunks. We can notice that two components (`ImageGenerator` and `Split Processor`) are runnable, and the horizontal and vertical split algorithms are implemented in two distinct components. These two components perform a similar task, hence they provide the same interfaces. During the execution, the application have to reconfigure its architecture to switch from one splitting algorithm to the other.

Although the component programming model is not widely used yet for embedded systems, it is well suited to their requirements [Crn04]. Indeed, components allow the adaption of the application architecture to the runtime constraints such as the workload, power consumption or available processors. The main reason of their low popularity appears 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.

DYNAMIC DATAFLOW PROGRAMMING

Researchers have developed and improved the dataflow programming models since the 1970s/1980's, as an alternative to conventional paradigms based on *von Neumann* processors [JHRM04]. These models shift the developer focus away from the stream of instructions executed by the CPU (*i.e.*, executing an instruction and incrementing the program counter) and push it towards the dependencies between data transformations. Put another way, this means that an instruction, or a block of instructions, is not executed when the program counter reaches it as in imperative programming, but rather when its operands are ready. These models were explicitly designed to exploit parallel architectures and try to avoid the main bottlenecks of *von Neumann* hardware: the global program counter and the global updatable memory.

As dataflow models put an important focus on data dependencies, applications designed with such models can form a directed graph, allowing verification of mathe-

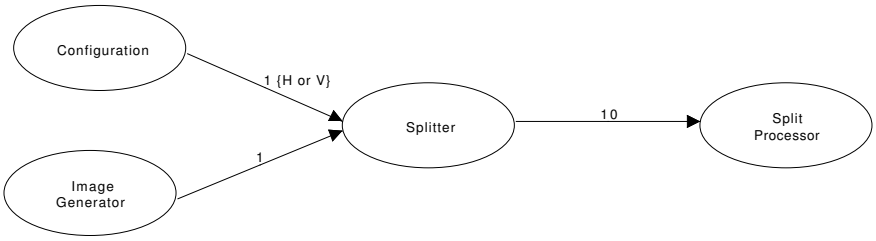


Figure 2.4: Dataflow Graph Example

matical properties. The nodes of the graph, named *actors*, correspond to the different data transformations of the application. The inbound arcs represent the data *arriving* to the actor (*i.e.,* input parameters of imperative languages) and the outbound arcs represent the data they generate (*i.e.,* output parameters). Thus, the arcs materialize the data dependencies. The dataflow models consider the information transmitted over the actors as *immutable*, hence the concept of "*input/output*" parameters of imperative languages does not exist in this context.

Figure 2.4 presents the graph of a dataflow implementation of our application mockup, composed of four actors. A configuration actor indicates whether the split should be horizontal (`H`) or vertical (`V`), and figures on the connections indicate how many tokens each actor expect to send or receive to complete its task.

In [BDT12], *Bhattacharyya et al.* distinguished two classes of dataflow models: *decidable* models and *dynamic* ones, more general. The former class enforces strong constraints to developers so that all the *scheduling decisions* can be made at *compile time*. This entails that the compiler can prove application termination, but also guarantees that the execution will run deadlock-free. It also enables powerful optimization techniques and helps design validation through static analyses. However, the development constraints strongly limit the set of implementable algorithms and essentially restrain it to static problems. Synchronous dataflow [LM87] is a famous example of decidable model. It is frequently involved in embedded and reliable systems, thanks to the extended guarantees it provides.

On the other hand, dynamic models are more permissive. In particular, they allow actors to produce and consume tokens at variable rates, that is, rates not predictable at compilation time. Multimedia video decoding, and more generally multi-standard or adaptive signal processing applications frequently require such dynamic processing capabilities. *Bhattacharyya et al.* illustrate in [BDT12] how MPEG or MP3 decoding involve large fluctuations dependent of the stream content.

The abstract machine of dataflow models defines actors, consuming and producing data from their inbound and outbound interfaces. At runtime, it connects the different interfaces and transports data between the actors, according to the dependency constraints defined in the dataflow graph. This graph may be defined explicitly or implicitly, depending of the supportive environments.

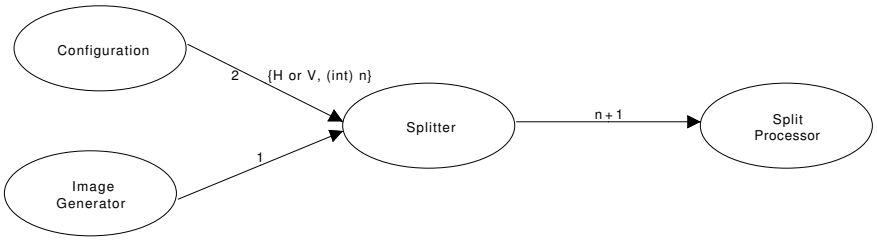


Figure 2.5: Dynamic Dataflow Graph Example

We can recognize that the illustration of Figure 2.4 was implicitly based on a *decidable* model, as the emission and reception rate of the actors is fixed. In the graph of Figure 2.5, we reworked our dataflow application and added a dynamic aspect: now, the `Configuration` actor sends a second token indicating in how many chunks the image should be split (`n`). Consequently, actors `Splitter` and `Split Processor` respectively send and receive a variable number of tokens (`n+1` informally indicates that the first token will contain the number of tokens— `n` –that will follow).

KERNEL-BASED ACCELERATOR PROGRAMMING

In this last subsection, we present the programming model behind the OpenCL [Khro8, TS12] standard (and incidentally close to NVIDIA CUDA's[3]). OpenCL aims at supporting "*heterogeneous computing on cross-vendor and cross-platform hardware [...], from simple embedded microcontrolers to general purpose CPUs [...], up to massively-parallel GPGPU hardware pipelines, all without reworking code.*" As the primary target of this programming model are accelerators and GPGPUs, its design took into account to the absence of outstanding OS managing the accelerator processors. For the same reasons, the model does not assume shared memory between the main CPU (the *host*) and the accelerators.

In this model, the *host*-side of the application prepares the work that the accelerators will execute in parallel. It consists of subroutines (*kernels*), optimized and compiled on-the-fly for the target processor architecture. Likewise, memory units (*buffers*) are dynamically allocated and transferred from/to the accelerator memory space upon request from *host*. *Kernel* execution is also triggered from the *host*, which specifies the number and arrangement of the processors executing the *kernel*. The *host* pushes all these operations into a *command queue* that can be configured to process operations *in-order*, which means that the *host* pushes the requests in the logical order and the accelerator will process them in the same order; or they can be set *out-of-order*, which means that the accelerator will process the requests as soon as possible. In this case, the *host* can indicate operation dependencies through execution markers. Typically, this mode allows computation-communication overlap exploitation.

We can distinguish four classes of entities in kernel-based applications:

3 Cuda Parallel Computing webpage, http://www.nvidia.co.uk/object/cuda-parallel-computing-uk.html

Devices A computer system may feature multiple accelerators, independent from one another. Hence, a device corresponds to an entity able to run parallel kernels.

Command Queues The command interface between the main processor and an accelerator device. It receives operation requests from the application, which are processed by the accelerator device.

Kernels The parametrizable code that is executed on accelerator devices. It is comparable to C functions.

Memory Buffers Memory areas in the accelerator devices' address space. They can be read-only, write-only, or read-write. It is comparable to C pointers to dynamically allocated memory.

The abstract machine defined by the kernel-based accelerator programming model is in charge of providing information about the accelerators available on the platform and executing the different operations pushed into the command queue. In also triggers parallel kernel executions into the relevant accelerator execution context.

Figure 2.6 illustrates the last variation of our mockup application. `ImageGenerator`, `Image Splitter` and `Split Processor` are accelerator kernels, and `Image` and `Splitted Image 1..n` are buffers instantiated in the accelerator memory space. The host side of the application should first request the instantiation of the `Image` buffer, set it as output parameter of kernel `Image Generator` and trigger its execution. And so on and so forth with the execution of the other kernels.

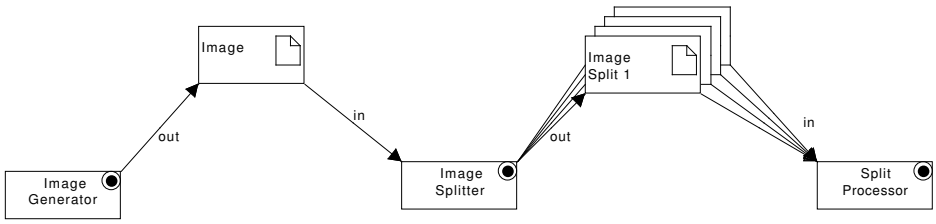


Figure 2.6: Kernel-Based Programming Example.

In the following subsection, we present how these three programming models are provided in STHORM development toolkit.

2.2.2 STHORM *Supportive Environments*

In this subsection, we introduce the supportive environments implemented by STHORM for component, dataflow and kernel-based programming. We will study these environments with more details in Chapter 5 and Chapter 6 where we explain how to debug applications developed from these environments.

NATIVE PROGRAMMING MODEL (NPM)

Native Programming Model (NPM) is a component-based programming environment developed to exploit STHORM architecture at a low level. 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 supports 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 [Lea00]. We will only consider the component aspect of NPM.

PREDICATED EXECUTION DATAFLOW (PEDF)

Predicated Execution Dataflow (PEDF) is a framework for dynamic hybrid dataflow programming, designed to exploit STHORM heterogeneous architecture. It provides a *structure* dataflow model, similar to what was presented in [JHRM04]. PEDF also originates from *dynamic* dataflow modelling [BDT12], so it does not enforce any constraint in actors' sending and receiving rates. Besides, it offers advanced scheduling capabilities, allowing the modification of the dataflow graph behavior during its execution (based on a set of predicates) or run some parts of the graph at different rates. It is based on the C++ language to benefit from the existing tool-chain (the compilation suite, but also the platform simulators). Figure 2.7 illustrates a PEDF graph of a simple module, composed of two filters and a controller.

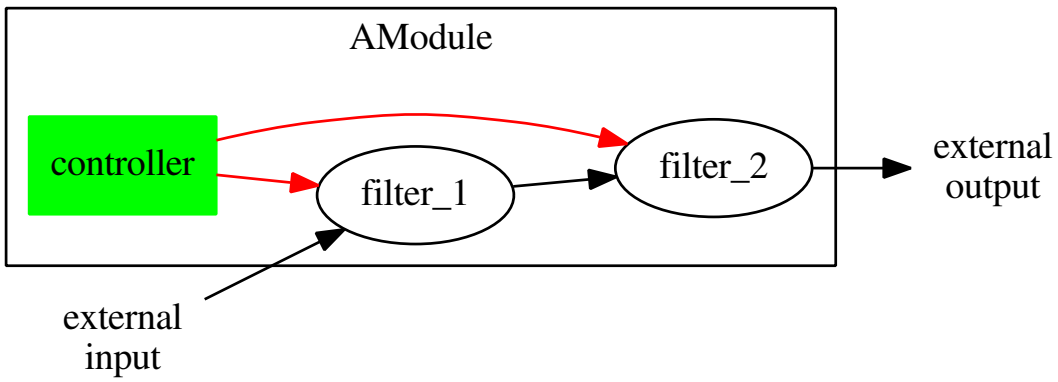


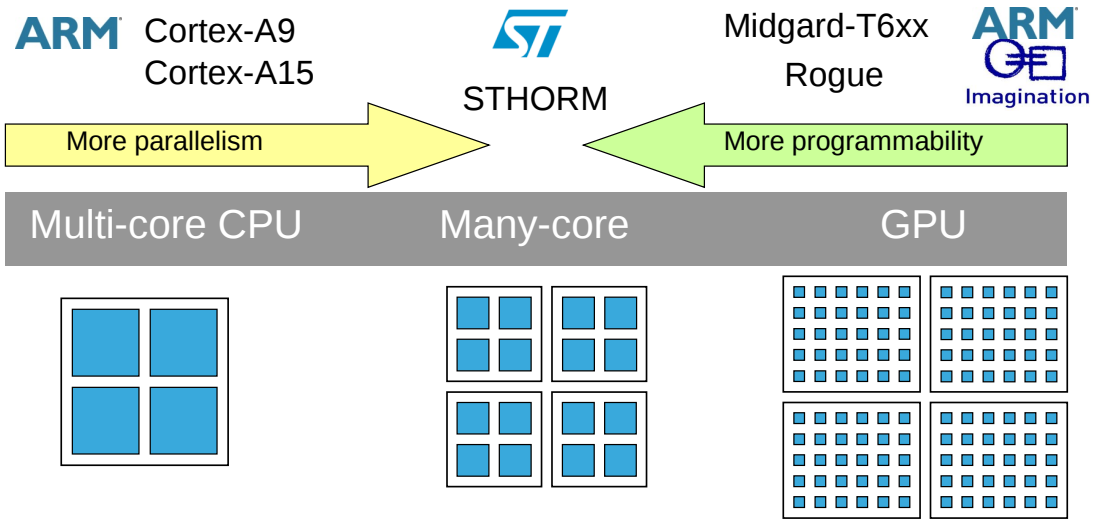
Figure 2.7: PEDF Dataflow Graph Visual Representation of a Simple Module.

OPEN COMPUTING LANGUAGE (OPENCL)

OpenCL is the combination of a standardized API and a programming language, close to the C language [Khr08]. It aims at offering an open, royalty-free, efficient and portable (cross-vendor and cross-platform hardware [TS12], but not in terms of

performance) interface for heterogeneous computing. OpenCL can be used to program GPGPUs, but also DSPs, CPUs or MPSoC processors.

STHORM provides an implementation of OpenCL, which is presented as a *standard*-oriented alternative to the other programming environments (NPM components and PEDF dataflow). *Paulin* also highlighted in [Pau13] that STHORM's OpenCL environment stands at the convergence of two criteria: more parallelism than multicore CPUs and more programmability than GPUs. Figure 2.8 highlights this positioning.



Based on [Pau13]

Figure 2.8: OpenCL as a Standard of Convergence

2.2.3 *Conclusion*

During application design and development, programming models and supportive environments provide developers with efficient tools for exploiting MPSoC systems. However a key step is still missing in this organization: application verification and validation. The implementation of such applications always holds a significant complexity, and hence, ensuring the correctness of the code is a difficult task.

In the following section, we highlight the challenges faced by developers during the debugging of MPSoC applications, the effect of runtime environments and the tools available to detect and locate application defects.

The Disruptive Element

2.3 DEBUGGING MPSOC APPLICATIONS

In the previous sections, we have seen that MPSoCs offer powerful computing environments, yet their complex hardware architectures requisite developers to rely on

programming models and supportive environments for an efficient application design and development. These programming models and environments also tend to make the debugging activity more complex, by adding intermediate layers in the application execution. Indeed, it appears that the development of supportive environment frequently neglects the debugging phase of the application life-cycle.

We start this section with an overview of the tools and techniques available today to developers for application debugging and present the advantages of *interactive* debugging. Then, we highlight the key debugging challenges developers face during MPSoC application development, for each of our three recurring models. Finally, we draw our conclusions regarding the abilities of current solutions.

2.3.1 *Available Tools and Techniques*

In the introduction of this document (Chapter 1, Section 1.2), we distinguished three stages of application debugging. In the following, we present two pre-execution approaches—pen-and-paper code study and static/formal analysis—and a *post-mortem* one—trace analysis. We point out some of their deficiencies, then we underline how live/interactive debugging overcomes these different problems.

Pen-and-Paper Code Study is one of the primary steps of application verification and validation. It requires a very good understanding of the application execution, and with a broader view, of the entire computer. It also allows high-level algorithmic analysis [CLRS09], such as complexity estimations, termination proof, *etc.* However in complex applications, developers cannot always control or even know the entire range of input parameters, so they may not be able to analyze exhaustively the set of execution states.

Static and Formal Analysis [Ölv11] may be exhaustive under certain conditions, such as strongly constrained and formally-defined programming languages, models and applications. In such cases, it provides the best guarantees, thanks to the mathematical models proving the correctness. However, the benefits for codes not matching these criteria are limited, which is the case for most dynamic applications.

Trace Analysis [KWK10] provides information gathered almost transparently from actual application executions. Data are collected from *trace-points*, statically or dynamically inserted in the code. The trace processing is done offline, *post-mortem*, either manually, through data mining [LCBT+12] or graphically with visualization tools (Figure 2.9 illustrates KPTrace[4], ST's trace visualization environment). However, trace analysis allows no interactivity and the information they collect is not exhaustive. Indeed, only a predefined number of trace-points are active during a given execution, and the more there are active tracepoints in an execution, the more intrusive the tracing is. On the other hand, the lack of interactivity can be an advantage for long-running executions.

4 Dynamic system tracing with KPTrace — STLinux, http://www.stlinux.com/devel/traceprofile/kptrace

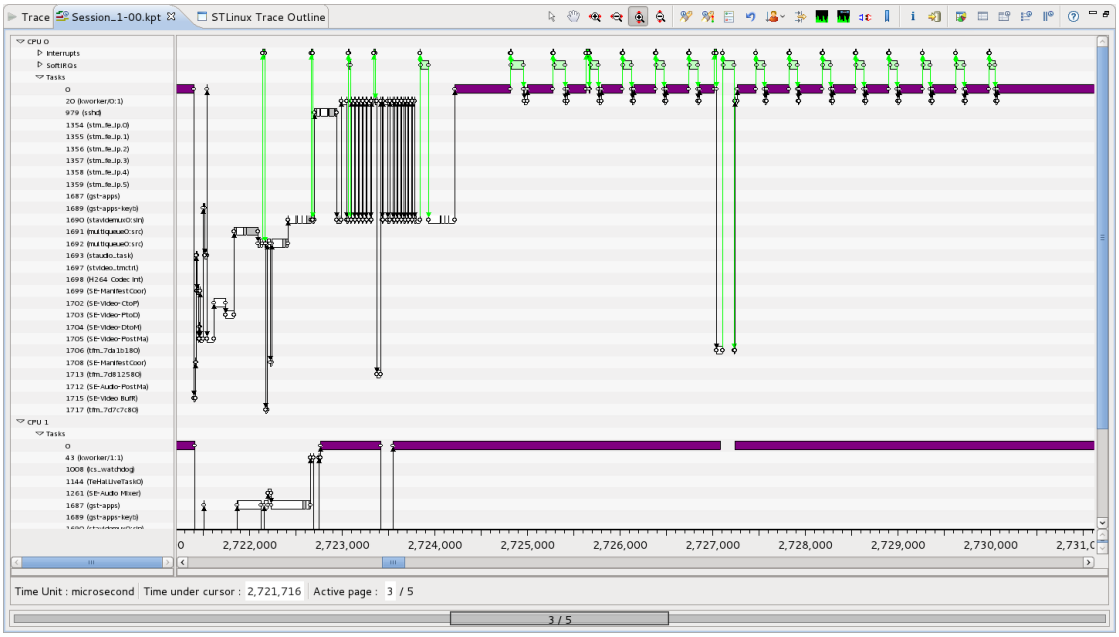


Figure 2.9: KPTrace Trace Analysis Visualization Environment

Interactive debugging allows developers to carry out their debugging experimentation under a different perspective, orthogonal to the approaches presented so far. Namely, it provides an *interactive* interface which allows developers to precisely control each step of the execution flow and to display the content of reachable memory regions (stack, heap, processor registers, *etc.*). Here are its main properties:

Interactive The control offered by the debugger is fully interactive, which means that developers can draw and verify different bug explanation hypotheses without necessarily rebuilding or re-executing the application. They can leverage a large set of features (breakpoints, step-by-step, watchpoints, *etc.*) to stop the execution and investigate the application state.

Transparent for the Execution The debugger control is virtually transparent for the application, so developers can execute their code normally until an erratic situation occurs. After the first application execution stops, the time-related behaviors will be altered (*e.g.*, inputs and outputs, scheduling decisions, timeouts, *etc.*). However, the rest of the execution conditions remains unaffected.

Exhaustive Once the application has been stopped, developers are free to examine the entire[5] execution context, that is, processor registers and local and global variables.

In the context of applications based on programming model, this last item is subject to caution and is further discussed in the following sections. Indeed, the programming-model abstract machine holds a significant part of the application state, and current debuggers do not provide support for accessing it.

5 Under the assumption that the debugger has access to the corresponding debug information and that the application binary was not overly optimized.

2.3.2 *Debugging Challenges of Model-Based Applications*

Interactive debugging is a complex activity where developers try to figure out where the actual code's behavior diverges from expectations. In order to do that, they must have a *complete* and *precise* control over the application execution. In the following, we stress out some important debugging challenges introduced by MPSoC programming models and environments. These aspects are key to offer an optimal application control, yet, as we point out, they are not addressed by current interactive debugging tools.

COMPONENT APPLICATIONS

Component application debugging faces the problem of the importance of inter-component communications, as well as the graph structure of the architecture that can change depending on applications' runtime constraints.

Dynamic Architecture Components are standalone computation entities interconnected through their interfaces. They are dynamically instantiated and bound to other components. Hence, the application architecture can change over time, according to execution requirements.

Source-level interactive debugging is 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 *provided* and *required* interfaces. Component execution is driven by the events received on each of these interfaces.

The notion of *interconnection* does not exist in source-level interactive debugging, so developers have to manually figure out the current component bindings and play subtly with breakpoints in order to follow the execution flow through an interface call.

Information Flow A component-based application can be composed of a various number of components, which apply transformations to the information stream. In this case, developers must figure out the path followed by the suspicious pieces of information to understand the application state.

Source-level debugging only provides details about the *current* processor and memory state.

DYNAMIC DATAFLOW APPLICATIONS

The difficulties of dynamic dataflow application debugging come from the fine-grain actor network and the importance of inter-actor communications.

Graph-Based Application Architecture The architecture of the application depends on its data-dependency graph. However, source-level debugging can only offer a

sequential—or multi-sequential if the runtime environment is parallel—view of the source code instruction stream. All the arcs of the graph are unavailable to developers.

Token-Based Application State The validity of a dataflow algorithm depends on the correct dispatching of data tokens. As in the theoretical models, nodes are stateless, the set of tokens present in the application holds the entire execution state. Thus, it is important for developers to have the ability to query the debugger about these tokens. This concept does not appear at language level and thus source-level debugging cannot provide such information.

Token-Based Execution Firing The execution of a dataflow actor can only start when the required input tokens have been generated. This concept does not exist in the *von Neumann* model, where the execution of a statement is only conditioned by the path followed by the program counter.

Non-Linear Execution Steps When a dataflow assignment instruction is executed, the actors waiting for this data become executable. Semantically, this forks the execution flow, which follows not only its normal (that is, sequential) stream but also the different outgoing arcs of the node.

Information Flow Dataflow applications can be composed of numerous actors, which successively apply a specific transformation to their incoming data. In order to understand the current value of a token, developers have to figure out the exact sequence of transformations undergone by this token. More concretely, this implies that the debugger should record the different token values over their processing steps.

KERNEL-BASED APPLICATIONS

As kernel-based programming is clearly apart from the two former models, the challenges of these applications are also more distinct. Here, the difficulties arise from the importance of the interactions with the abstract machine and its internal state.

Internal State Kernel-based applications are executed both in the general-purpose processor and in the accelerator. As these two sides do not share memory, the abstract machine implemented by the environment holds an important part of the application state: list and state of instantiated buffers and kernels, state of the execution queue, *etc.*

Source-level debugging considers the supportive environment as a black box. Hence, developers cannot access or query any of these information.

Calling Conventions In kernel-based applications, developers must programmatically *configure* the abstract machine to execute kernels. (We can compare that to the C calling conventions, where, at assembly level, function parameters and return address are pushed in the stack before jumping to the address of the function's first instruction.)

In this case, developers must manually figure out (or guess) how the abstract machine was configured to execute a given kernel.

Operation History Kernel-based applications must frequently interact with the underlying abstract machine in order to prepare and trigger kernel execution, instantiate, read and write memory buffers, *etc.* Hence, developers should have an easy access to this history, in order to better understand how the application has reached its current state.

To illustrate the complexity of the interactions between the application and kernel-based programming abstract machine, Figure 2.10 presents a schematic representation of the OpenCL operations required to execute a kernel with a memory buffer parameter. It is inspired from UML sequence diagrams and reads similarly. Column "`:Application`" corresponds to the operations executed by the host, and the other entities are part of the abstract machine. We can count a minimum of ten operations. The equivalent computation written in plain C would not have required much more than three lines of code (the body of the `main` function):

```
void process (int *buffer);
int main () {
   int *buffer = malloc (...);
   *buffer = ...;
   process(buffer).
}
```

2.4 CONCLUSION

In this chapter, we reviewed the background of MPSoC programming and debugging. We first presented MPSoC architectures and introduced different programming models able to exploit their inner characteristics. Indeed, developing applications making the best use of MPSoC's multicore processors and unique designs requires advanced algorithmic concepts, combined with efficient runtime libraries. To meet these requirements, we presented three alternative programming models and environments: component-based programming, dataflow programming and kernel-based accelerator programming.

However, from a verification-and-validation perspective, programming models split into two categories: either they enforce strong constraints through static programming and can offer interesting correctness guarantees; or they are less restrictive and cannot provide any verification help. As this thesis work was carried out in a multimedia-oriented division of ST, we focused on the latter category. Indeed, multimedia programming often requires dynamic programming capabilities. For instance in video decoding or image processing, the execution may depend on the content of the media.

We presented different debugging tools and techniques: pen-and-paper code study, static and formal analysis and trace analysis, and compared them against interactive

debugging. Through this comparison, we highlighted that interactive debugging provides a compelling approach, which allows developers to interact with the application and inspect the different execution paths and program states.

However, we stressed out that source-level interactive debugging is not sufficient for efficiently tackling applications that rely on a programming model. Indeed, the models, and more concretely their supportive environments, introduce high-level abstractions in the applications structure, which are not represented nor taken into account by current tools operating at source-level.

In the following chapter, we present our contribution to answer these challenges: programming-model-centric interactive debugging. We first detail the general and generic principles, then we explain how we applied these principles to our three MPSoC programming models.

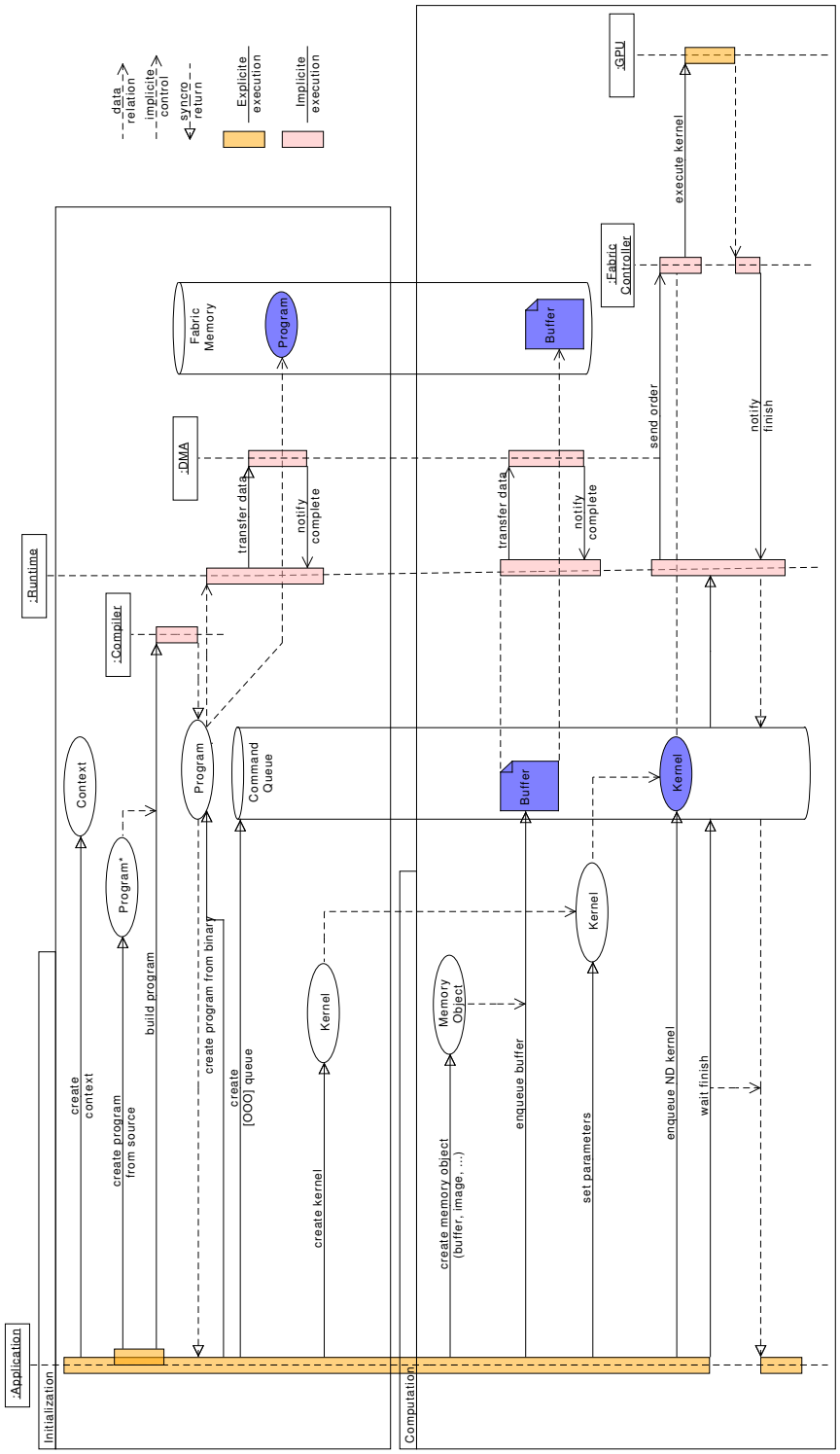


Figure 2.10: Sequence Diagram of a Basic Kernel Execution

3

CONTRIBUTION: PROGRAMMING-MODEL CENTRIC DEBUGGING

The Hero

Although MPSoC systems provide a powerful computing environment, their performance appears difficult to exploit without advanced and high-level programming models. At implementation time, supportive environments such as middleware frameworks or runtime libraries materialize programming-model guidelines. The interactions between the application and the underlying frameworks are defined by the API, based on the programming model's abstract machine interface.

At the same time, the complexity of these environments hinders the verification and validation process. Indeed, they abstract away the simple and generic representations provided by the hardware platform: instead of a set of *von Neumann* processors, executing sequential streams of instructions, applications are now defined with high-level structures.

In this chapter, we present our contribution to lighten the difficulties of interactive debugging of such applications. In the first section (Section 3.1), we explain the principles of programming-model centric debugging. Then, in Section 3.2, we delimit its scope of applicability. Finally, in Section 3.3, we study how these generic principles can be applied to our three MPSoC programming models.

3.1 MODEL-CENTRIC DEBUGGING PRINCIPLES

Our contribution consists of a set of functionalities that a debugger should implement in order to offer a *programming-model centric* vision of multicore application debugging. With this approach, we expect to provide developers with more efficient tools to debug their applications. Instead of working with system abstractions like threads and processes, they will interact with the very entities and communication operations defined by the programming model abstract machine of their application.

Our approach relies on detecting and handling of the key programming-model related operations of the execution. Debuggers should interpret these operations to be able to follow the abstract-machine state evolution. This would enable them to provide accurate and high-level information and control mechanisms to debug applications based on programming model.

3.1.1 *Providing a Structural Representation*

The application architecture is an important aspect of the state, that debuggers should monitor and represent. It can be static or dynamic. In the latter case, the architecture can vary over the time, so it is crucial that debuggers follow its evolution in order to provide an accurate view of the current deployment.

Debuggers should provide developers with catchpoints (*i.e.,* operation breakpoints) on these architecture-modification operations.

They should also capture and represent the relationship between the different entities, in particular when the model explicitly defines such connections. Otherwise, different metrics can be evaluated to estimate the entities' affinities, such as their communication frequency or the underlying processor topology, if entities are pinned to a particular core. The set of entities and inter-connections forms a graph (directed or not) that can help developers to detect unexpected situations, for instance by studying the graph shape or the dynamic of inter-entity exchanges.

3.1.2 *Monitoring the Application's Dynamic Behaviors*

The different entities of a parallel application usually collaborate in order to complete their task. Hence, debuggers should provide information about these interactions. Namely, they should interpret communication and synchronization events and represent them with respect to the graph structure of the application. They should also be aware of the pattern and the semantics of communication operations in order to precisely interpret their behavior. For instance, communication patterns can be one-to-one, one-to-many, global or local barriers, *etc.* The semantics of the link can be First-In-First-Out (FIFO) or implementation specific. This last case might require further cooperation with the supportive environment, as discussed in the following chapter (Chapter 4, Section 4.2).

Debugger should also allow developers to stop the execution based on these interactions.

3.1.3 *Interacting with the Abstract Machine*

Parallel programming models should facilitate application decomposition, hence a programming-model centric debugger should be able to distinguish and identify these different entities. It should also provide indications about their inner state, like their schedulability or outstanding communication events.

A modification of an entity state may indicate a turnaround in the execution, so debuggers should provide watchpoints (*i.e.,* state-modification breakpoints) for such events. These watchpoints would allow developers to reach different time and space locations of the execution more easily.

Two-level Debugging

Finally, as the instructions of any application are eventually written in a standard programming language and executed by the processor like traditional code[1], 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 entity-specific) and other step-by-step execution control primitives should be directly available.

3.1.4 *Open Up to Model and Environment Specific Features*

Different programming models do not provide the same functionalities, not do they require the same debugging capabilities. Therefore, programming-model centric debuggers should adapt their debugging features to the specifics of the programming models and environments they are targeting. At this stage we can only provide hints about such features, but Section 3.3 and later Chapter 6 provide more detailed examples.

- Debuggers can follow messages transmitted from entity to entity, either based on a model-defined routing table for the entity being considered, or through user-provided tables;
- Debuggers can check user-defined constraints on the graph topology, on message payload, paths, *etc.*, and stop the execution in case of violation.

More advanced features can also be designed thanks to the strong programming-model knowledge achieved by the debugger:

- Debuggers can detect deadlock situations with loops in the graph of blocking communications;
- If the debugger supports non-stop debugging [SPA+08], "*smart*" breakpoints can stop the tasks trying to communicate with tasks already stopped by the debugger, in order to limit the intrusiveness.

In the following section, we delimit the scope of applicability of this approach.

3.2 SCOPE OF APPLICABILITY

Model-centric debugging can be applied to various kinds of targets. Its primary objective is task-based programming models for multicore processors. Indeed, such tasks should communicate with each other and form, implicitly or explicitly, a graph. They should also be executable in parallel. Component and dataflow programming perfectly fit in this area.

1 We assume compiled languages here, but the rationals are similar for interpreted languages.

However, the scope of application is broader than that, as we demonstrate later in this chapter with kernel-based programming. Any programming model defining an abstract machine complex enough may benefit from this approach. And, as explained in the previous chapter, the notion of abstract machine is loosely defined, on purpose. Thus, model-based debugging can be applied on top of any API, provided that someone devotes time to its implementation.

We can exemplify this last point with a video decoder, where a model-centric debugger could recognize the different modules (*e.g.*, sound decoder, beginning/end of a frame, the error channel, *etc.*). This would help developers to understand more rapidly the current state of the execution: decoding frame N, previous frame dropped, error channel empty, *etc.*

On the other hand, it is important to note that we only focused on a particular aspect of multicore computing: analyzing the cooperation between entities running in parallel. We do *not* address the problem of debugging a *large* number tasks, neither the *time*-related challenges of concurrent executions.

The main reason for that is that we believe that interactive debuggers are not suitable for this kind of problems. Indeed, for the former aspect, the *quantity* of information developers can understand at each step of the execution limits the possibilities of interactive debugging. If thousands of tasks are running concurrently, developers cannot go through all of them and verify that their state matches their expectations. Designing tools offering such capabilities is another research topic. Instead, they should try to narrow the problem down to a minimum size, both in term of number of parallel executions and processing time. Time-related issues are well-studies, although not yet solved. Limiting the intrusivity of interactive debugging, and furthermore improving it for such problems is yet another independent research topic.

For similar reasons, we do not target SIMD parallel computing. For such applications, a simple alternative would consist in running the code sequentially and use traditional debugging tools (or model-centric, if applicable).

Finally, the industrial context of this thesis set an additional constraint to the scope, which was that the work should focus on applications scaled for the companies' embedded boards. This implied embedded multicore MPSoC platforms, but not large-scale, HPC-like computers.

Now that we have delimited the scope of applicability, we present, in the next section, how the principles of model-centric debugging apply to our three programming models.

3.3 HOW DOES IT APPLY TO DIFFERENT PROGRAMMING MODELS?

The principles of model-centric debugging presented in Section 3.1 are generic and independent of a particular programming model. In this section, we explain how we specialized and applied these principles to our three MPSoC programming models. For each of them, we highlight different benefits of model-centric debugging: dynamic architecture reconfiguration and message exchange of component-based ap-

plications, graph-based architecture and flow of data inside dataflow applications and the interactions between kernel-based applications and the abstract machine.

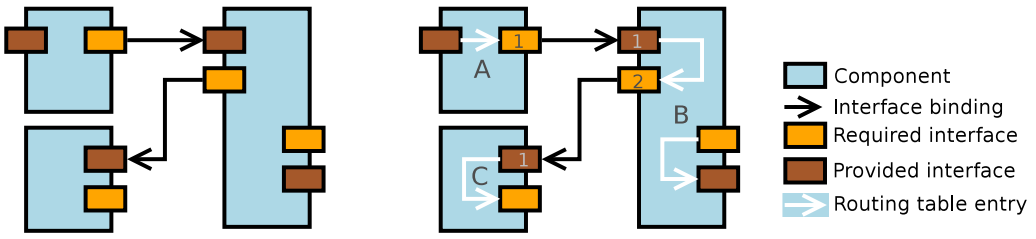
3.3.1 *Component Debugging*

In this subsection, we describe how model-centric debugging can be applied to component-based programming. This work was published in [PSMMM12]. It validated the first steps of this thesis.

PROVIDING A STRUCTURAL REPRESENTATION

In component-based software engineering, the main entity division consists in components, as the name suggests. The description of a component consists in a list of *provided* services that the component implements. The description also specifies the list of *required* services. These service interfaces must be connected to the matching *provided* interfaces of another component before the execution. Component interfaces have a precise type (like C function prototypes), in order to guarantee the consistency of inter-component exchanges. Optimally, these types are architecture independent.

✓ A model-based debugger must be able to exhibit these abstractions: through a representation based on directed graphs, components can be depicted as *nodes*. Each node can have incoming and outgoing arcs, corresponding respectively to required and provided interfaces. Figure 3.1(a) shows a diagram of a simple structural representation. It contains unconnected required interfaces, which may indicate that the application is not in a runnable state. (The component abstract machine may allow *optional* interfaces, but this is out of the scope of this document.) Besides, a component description may provide *names* for the interfaces. This information can further improve user's debugging experience by simplifying the interactions between the developer and the debugger.



(a) (b) with routing table for message-based flow control.

Figure 3.1: Structural Representation of Interconnected Components.

Component frameworks can also provide predefined interfaces, called *system* interfaces (as other interfaces, they can be either provided or required). We will cite only one example which may appear in any framework targeting multicore systems: the *runnable*

service. Components can implement this *provided* interface in order to be executed on a dedicated processor core.

✓ Model-centric debuggers can be aware of such *system* services and handle them appropriately. In the case of the *runnable* interface, this implies that the debugger should detect:

- the operation *triggering* the component execution ,
- the beginning of the component execution and the memory and processor contexts it is bound to,
- any operation blocking or diverting the execution flow away from the component,
- and finally, the end of the service execution.

These events, exemplified in the component case-study analysis (Chapter 6, Section 6.1), improve the accuracy of the application's structural representation.

MONITORING APPLICATION DYNAMIC BEHAVIORS

We presented above how model-centric debugging leverages component structures and interface connections. However, it is important to note that this graph network may vary over the execution timespan. Indeed, the application can chose to deploy new components or re-wire the interconnection network, to adapt its architecture to runtime constraints.

✓ In this case, a model-centric debugger should detect these abstract machine operations, and reflect the modifications on the graph presented to the user. In order to improve the control over the execution flow, the debugger should also provide catchpoints for this operations. For instance, this would allow developers to stop the execution when a connection is created between components A and B, or the first time a particular interface is disconnected. Similarly, component life-cycle events discussed above can offer interesting execution catchpoints.

Another aspect of application dynamic behavior is the information exchanged between the components. This information is two-fold. First, the service request: one component invoking another one, through an interface call. The Second, the interface call *data*—that is, the parameters and return values.

✓ In this regard, model-centric debuggers should handle both of these two aspects. Interface calls are similar to "*native*" function calls, though more complex. Indeed, they may involve multiple framework-implementation-dependent function calls that developers cannot understand (because, for instance, the framework implementation is a black box). Bypassing this roadblock would involve, first, understanding which component is *currently* connected to the other side of the interface, and second, locating where the service is actually implemented. This second step is not complicated, but nevertheless, the overall operation disrupts developers from their bug tracking activity. Furthermore, in case of multicore environments, the two components involved in the

interface call may not be running in the same memory and execution context. Following such remote function calls is even more difficult for developers.

Regarding the interface call data, we can imagine more advanced mechanisms, such as the concept of message-based flow control, where a message corresponds to the information transmitted during an interface call. This mechanism 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, per component or per link. Given 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, the model-centric debugger will be able to give further information about the message history. A generic stamp contains the component name and a 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.

In some cases, such as pipeline-shaped application architectures, developers may be able to identify routing patterns within their components. In this situation, they can provide routing tables to the debugger based, for instance, on component and interface names, but also according to the current application architecture and memory state.

We can illustrate the concept of message-based flow control as follows: consider a streaming application which applies a set of transformation to frame data. Each transformation is implemented in a dedicated component (`Component A` , `B` and `C`), as in to Figure 3.1(b).

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
```

We can read 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 debugger could simplify the history information and better indicate the route followed by the message:

```
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
```

To complete this application study, the following subsection will tackle the question of model-centric debugging in the context of dataflow-based programming.

3.3.2 *Dataflow Debugging*

The principle of dataflow computing strongly emphasized the importance of data dependencies, even more than the data processing itself. Hence, a model-centric debugger for dataflow programming should address both aspects. In the first part of this subsection, we study how the debugger should leverage data dependencies to provide a structural representation of the application; then, in the second part, we focus on actor execution to monitor application dynamic behaviors.

This work was published in [PLCSM13b, PLCSM13a].

PROVIDING A STRUCTURAL REPRESENTATION

A dataflow application consists of a set of actors, applying transformations on flows of incoming data and generating one or multiple outgoing data flows. The constraints governing the data flow rates depend on the dataflow flavor choice, and does not affect the design of the model-centric debugger in a significant way[2]. Besides, similarly to component-based applications, dataflow actors and data dependencies form a graph structure. However, data dependencies usually cannot be dynamically reconnected. Likewise, the concept of component interfaces does not fit in this model, as data dependencies (data types) are simpler than component interfaces (function prototypes)

✓ Hence, we can use similar techniques to provide a structural representation of the application: mapping actors onto graph nodes, and data dependencies onto directed arcs.

An important distinction between dataflow actors and components lies in the level of implementation: a component should provide an "*entire*" service, whereas a dataflow actor should only implement a particular data transformation. Hence, this implies that dataflow graphs will be more complex.

✓ To cope with this aspect, the model-centric debugger should be able to present the dataflow architecture as a proper graph. Figure 3.2—part of the dataflow case-study of Chapter 6, Section 6.2—illustrates a possible representation of a dataflow graph.

2 Such variations are rather considered at implementation time, presented in Chapter 5.

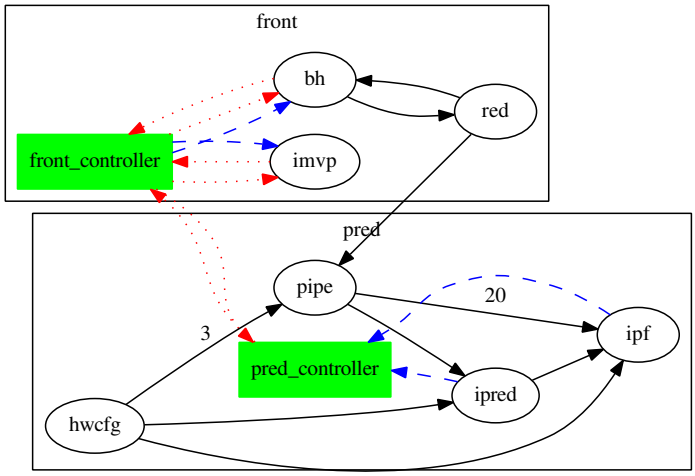


Figure 3.2: Graph of Dataflow Actors and Data Dependency of a Dataflow Application.

MONITORING APPLICATION DYNAMIC BEHAVIORS

In dataflow applications, the principal dynamic behaviors consist in the data tokens transmitted between the actors. Indeed, their availability, or absence, will trigger or block actors execution. Put another way, dataflow tokens control the scheduling of actors' execution. Hence, a model-centric debugger should pay special attention to these entities.

✓ Message-based flow control, presented with component debugging (Section 3.3.1), can be useful for dataflow debugging as well. Dataflow links are more complex than component interface calls, as they can *buffer* tokens, until they are effectively required at destination. As an example, Figure 3.3 schematizes an actor, Actor 1, sending tokens to another actor, Actor 2. Actor 2 does not consume the tokens at the same rate as Actor 1 produces them, and therefore, the data link buffers the tokens.

If a developer wants to stop the execution when the *last* token (`B*`, which is identical to `B`) arrives at Actor 2, he needs to stop the execution each time a token is received, determine if it is the right one (that is, distinguish `B` from `B*`, maybe by remembering that the *previous* token should have been `C` instead of `A`). This kind of task can be entirely automatized in a model-centric debugger.

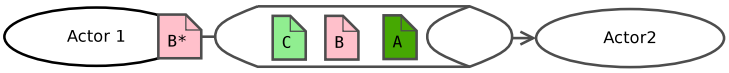


Figure 3.3: Tokens Exchanged and Buffered between Dataflow Actors.

In addition, model-centric debuggers can provide various token counters to its users: Figure 3.2 contains two numbers, between actors `hwcfg`, `pipe` and `ipf`. These numbers indicate how many tokens are currently buffered in the link. If the value is too high, this may indicate a problem in the sender or receiver implementation. Likewise, debuggers can provide counters on actors' data dependencies. Either manually or

through conditional breakpoints, this can help developers to detect irregularities in the actors' sending or receiving rates (for instance, two interfaces must send messages in a lockstep fashion, or twice as fast, *etc.*)

Model-centric debuggers can also use data-tokens to simplify the problem of non-linear execution steps — presented earlier with the dataflow debugging challenges (Chapter 2, Section 2.3.2). Indeed, the execution flow forks stem from the token emitted during the dataflow assignment. Hence, setting a double breakpoint, one right after the assignment and the other one on the token itself will resolve the problem.

Finally, the knowledge gained with the dataflow graph and actors' send and receive operations allows designing more complex features, such as deadlock detection: when there is a loop in the directed dataflow graph, a deadlock may arise. Figure 3.4 presents such situations, with multiple cycles:

1. `hwcfg` → `pipe` → `ipred` → `hwcfg`
2. `hwcfg` → `pipe` → `ipred` → `ipf` → `hwcfg`
3. `hwcfg` → `pipe` → `ipf` → `hwcfg`

Then, if all the actors of the loop wait for a data token from a previous actor, the situation is blocked as none of them can escape the blocking operation. The actors and links highlighted in bold red (item 3 above) are in this situation.

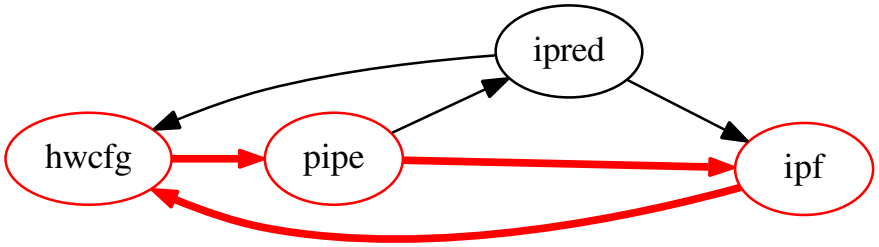


Figure 3.4: Dataflow Actors in a Deadlock Situation.

Now that we have presented how model-centric debugging can be applied to dataflow programming, the following subsection will explore how it fits into kernel-based applications.

3.3.3 *Kernel-Based Accelerator Computing Debugging*

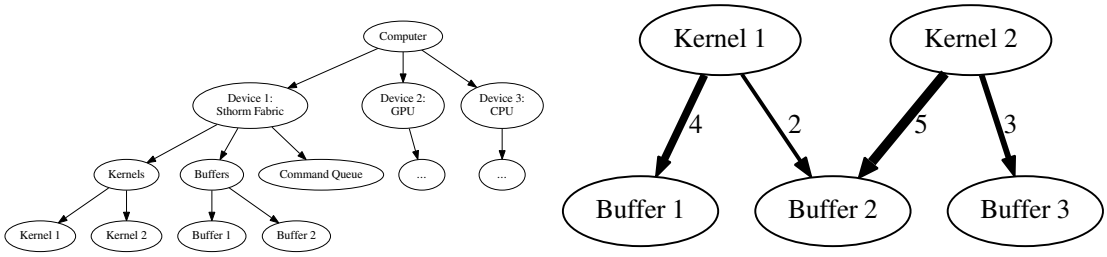
Our last programming-model case-study differs from the previous ones, in the sense that it is not based on the task paradigm. Hence, the structural representation of the application does not hold the same importance. In the following paragraphs, we briefly sketch an example of structural representation for model-centric debugging, then we go on and analyze the question of debugging the interactions between the application and the abstract machine.

PROVIDING A STRUCTURAL REPRESENTATION

The kernel-based programming model specifies various execution entities, such as the devices, command queues, kernels and buffers. It also has the peculiarity of assuming a particular execution architecture, based on accelerator and GPU architectures, with sets of execution platforms (devices) which do not share memory, neither with one another nor with the main processor. Hence, the structural representation of kernel-based applications mainly consists in the following hierarchy, as illustrated with the graph representation in Figure 3.5(a):

$$\text{Computer} \rightarrow \text{Devices} \rightarrow \text{Command Queues} \rightarrow \{\text{Kernels, Buffers}\}$$

However, this organization is mainly involved in the *configuration* phase of the application. During the *computation* phases, kernels and buffers are the only active entities. Thus, the structural representation should focus on them, as presented in Figure 3.5(b). Kernel-based computing does not define any *strong* relationship between kernels and buffers. Hence, Figure 3.5(b) was sketched using *usage* metrics: the more a buffer is used as a kernel parameter, the thicker their connection arrow.



(a) Full-Scale View (b) Focused View

Figure 3.5: Structural Representation of a Kernel-Based Application.

INTERACTIONS WITH THE ABSTRACT MACHINE

Because kernel-based applications are divided in two parts, the model's abstract machine plays the role of the middleman between the main processor context and the accelerator one. Furthermore, the accelerator context is "*headless*", hence all the piloting work is done through the host-side interface of the abstract machine.

✓ In a model-centric debugger, we expect the tools to help developers to better follow and understand the sequence of these operations. Indeed, as the entire state of the accelerator device is held by the abstract machine, developers cannot access or query any of this information in a classic source-level debugger.

A model-centric debugger for kernel-based programming models can dynamically draw a sequence diagram, by interpreting the different interactions between the application and the abstract machine. This diagram will help them to figure out the current state of the execution more easily. Indeed, developers can quickly review which kernels were last executed, or deduce the state of a buffer after its last operations (*e.g.*, part of a kernel execution, transferred to the host memory, released, ...).

Finally, a model-centric debugger can also provide potential error hints to programmers. In particular, in the case of memory buffers, some operation sequences are likely to hide a defect: a buffer read or written twice in a row, a read-only buffer set as kernel parameter before its initialization, or conversely, write-only buffer not read after a kernel execution, *etc.*

3.4 CONCLUSION

In this chapter, we presented our contribution to improve developer experience during the debugging of parallel applications based on MPSoC programming models. We first stated the generic principles of model-centric debugging, which spread in three directions:

1. providing a structural representation of the application,
2. monitoring the application's dynamic behaviors,
3. allowing developers to interact with the abstract and actual machines.

Through these three axes, we defined the requirements for designing a high-level debugger, well-fitted to tackle the debugging of application based on programming models.

We also demarcated the scope of applicability of model-centric debugging: primarily, multicore applications relying on programming models guidelines and supportive environments. We also noted that this is not a hard limit, as model-centric debugging can be applied to any kind of abstract machine, provided that its interface is stable enough.

On the other side of the scope, we explained that model-centric debugging is not suited for large-scale application deployments, as it would overflow developers with more information that they could handle. We also noted that, for similar reasons, model-centric debugging would not provide great help in solving problems of applications based on data-parallelism (SIMD).

Next, we studied how model-centric debugging applies to our three MPSoC programming models. We highlighted that component and dataflow debugging can directly benefit from the structural representation, thanks to their task-based aspect. In the same mindset, model-centric debugging nicely fits into their communication operations, and we describe how messages and tokens can improve the controlability of such applications. Finally, we noted that model-centric principles also apply to more distinct models, such as kernel-based accelerator programming. We noted that the interest of the structural representation is not as valuable as in the other models. However the nature of the interactions between the application and the abstract machine opens doors for designing interesting model-centric debugging features such as sequence diagrams of the different interactions.

Extending debuggers with application-specific debugger knowledge was discussed in [MCS+06], where the authors indicated that debugger scripts "*often want to create*

a redundant model of the execution" and "*need to define problem-specific commands*." We share these convictions, however the approach described in the article only considers "*per-application*" script extensions (*e.g.,* following items pushed in and out of a priority queue). In our context of embedded and multicore programming, we believe that the extended debugger knowledge should better target programming models and environments. Indeed, this programming level offers a high-level interface to the application, well-suited for building debugger extension, and also has the important advantage of being shared among multiple applications.

In the following part, we carry out a practical study of model-centric debugging, from debugger design and implementation to case-studies of industrial application debugging. This debugging framework is an integral constituent of our contribution, as it provides a unified environment for debugging MPSoC applications based on any of the three programming models we studied.

Part II

Practical Study of Model-Centric Debugging

4

BUILDING BLOCKS FOR A MODEL-CENTRIC DEBUGGER

The Adjuvant

In this chapter, we detail the implementation of the key building blocks of a model-centric debugger, based on the directions drawn in the previous chapter, Section 3.1. Figure 4.1 presents the different layers of the architecture of our model-centric debugger implementation. The two bottom layers are the usual debugging entities: the execution platform (the debuggee), which runs the application, and a source-level debugger.

In Chapter 2, we presented the bottom part of the figure, the execution platform, hence now we continue with the upper layers. We start in Section 4.1 with the middle part and present the source-level debugging functionalities required to implement a model-centric debugger. Then we continue with the top part of the figure, which corresponds to the programming-model centric debugger itself. We divided its implementation study into three axes:

1. the debugger needs to capture the information required to follow the evolution of the abstract machine state. In the figure, this corresponds to the *pink arrows* connecting the upper layer, the source-level debugger and the execution platform (Section 4.2).
2. the debugger needs to define internal structures reflecting the abstract machine organization, and update them according to the evolution of the machine state. These structures are depicted in the schema with the *three interconnected entities* (Section 4.3).
3. the debugger must provide a high-level user-interface, allowing developers to efficiently interact with the abstract machine. Most of commands provided through this interface are parameterized by the current state of the internal representation. This interface is represented by the *user* on top of the diagram. (Section 4.4).

4.1 SOURCE-LEVEL DEBUGGER BACK-END

We noted in the previous chapter (Chapter 3, Section 3.1.3) that it was important for a model-centric debugger to be able to operate at two levels, with source-level debugging capabilities. This requirement led us to decide to implement our prototype as an

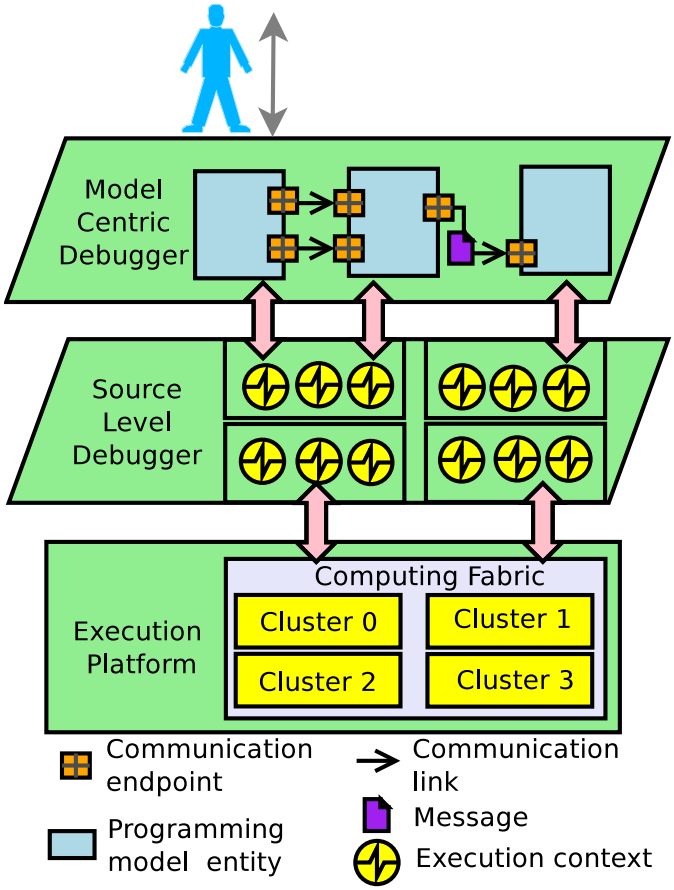


Figure 4.1: Model-Centric Debugging Architecture for an MPSoC Platform

extension of an existing source-level debugger. This way, all the low-level functionalities are natively available to application developers, and we can directly leverage this technology for the implementation of our tool.

Hence, our prototype debugger relies on GDB, the free debugger of the GNU project [Gnu13]. GDB has a wide 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 and product handovers (ST's IDTEC team already provides its customers with GDB-based products [GADP+10]). Moreover, recent versions of GDB export PYTHON bindings, which allow an easy and efficient development of extensions.

In the following, we present the two main GDB capabilities our implementation relies on: internal breakpoints and PYTHON scripting. A third aspect, memory inspection, is presented in Appendix A.

4.1.1 *GDB Breakpoints*

A "*break-point*" is a debugger operation which breaks the execution flow at a given point. More formally, a "*point*" is a memory location, for instance a function address, a source-code line (the debugger looks up the corresponding assembly instruction address in the debug information) or a user-provided memory address. "*Breaking the execution flow*" means that, if one of the processors hits a breakpointed address, the debugger will give the control back the its user interface, allowing developers to interact with the application in the current memory context.

Under the hood, a "*breakpointed address*" is an illegal instruction written at the given memory location (the debugger takes care to backup the original instruction beforehand). The execution of this illegal instruction triggers a processor *fault* whose handling depends of the OS. For instance, LINUX kernel can be configured to send a signal (`SIGTRAP`) to the process' debugger. Upon receiving this signal, the debugger recognizes the breakpointed address and switches to its interactive mode. Continuing the execution involves switching back and forth between the debugger and the application execution contexts, in order to actually execute the breakpointed instruction, but also ensuring that no processor misses the breakpoint stop at the same time. *Sidwell et al.* explained in [SPA+08] these challenges.

Conditional breakpoints improve breakpointing with *conditional* expression testing. It is important to note that these conditions do not change the low-level aspects presented above. It only conditions whether, upon a breakpoint hit, the debugger gives the control back to the user-interface or *silently* continues the execution. For instance in a C application:

```
(gdb) break main if argc == 3
```

At the beginning of the `main` function, this conditional breakpoint will only give the control to the user *if* the process has 3 arguments. But in *all* cases, the execution will be stopped and the debugger will internally check the condition.

Internal Breakpoints are a special class of breakpoints, which are not set by the debugger user, but by the debugger itself, for internal purposes. They allow the debugger to perform specific internal actions. In LINUX-based environments, GDB uses some of them to monitor the dynamic linker (-1), threads (-2 and -3), POSIX `longjump` (-4 and -5) or handle C++ exception (-6):

```
(gdb) maintenance info breakpoints
Num Type            Disp Enb Address      What
-1  shlib events    keep y   0x302e80f7a0 <_dl_debug_state>
-2  thread events   keep y   0x302f406ad0 <__nptl_create_event>
-3  thread events   keep y   0x302f406ae0 <__nptl_death_event>
-4  longjmp master  keep n   0x302f40dc90 <siglongjmp>
-5  longjmp master  keep n   0x302ec35b13 <__longjmp+51>
-6  exception master keep n  0x303040fa90
```

To further explain internal breakpoints, we can look at the two thread-event breakpoints. They are both internal to the GLIBC threading library (the NPTL here, [Molo3]), as shown by the `__nptl_` prefix. The first event (`__nptl_create_event`) is triggered—that is, the function is executed—by the threading library when a new thread is created. Upon the breakpoint hit, GDB queries further information about the thread and registers it to its thread list. Finally, the second event, (`__nptl_death_event`) is triggered by the library when the thread has completed its task. GDB knows that it can remove it from the thread list. During the processing of these events, GDB displays the following messages in the console. They are the only notifications of these breakpoints. Our former work [PPCJ10] further explain this process:

```
[New Thread 0x7fffddff6700 (LWP 18539)]
...
[Thread 0x7fffddff6700 (LWP 18539) exited]
```

4.1.2 *GDB Python Scripting*

PYTHON scripting has been present in GDB since its version 7.0 in 2009. It allows extending GDB behavior through an external API, instead of writing C code. This modularity has multiple benefits:

Stable The API only exports stable functionalities, backward compatible over future releases. This is in opposition with the internal C code, for which no stability guarantees are provided. Hence, an extension developed in C may break after any new release and must be verified each time.

Focused/documented The PYTHON API was designed with the explicit goal of allowing user scripting. Hence, it provides a focused set of functionalities and hides GDB's internal complexity. In addition, all the elements of the API are documented and accessible online[1].

1 Debugging with GDB– PYTHON API, http://sourceware.org/gdb/current/onlinedocs/gdb/Python-API.html

Efficient development PYTHON is a widely used programming language, fully flavored[2] and used for large-scaled development such as the YOUTUBE website. This is invaluable in comparison with GDB's original scripting capabilities which could only automate repetitive console command-lines.

External PYTHON extension can be distributed independently from GDB, without requiring a software recompilation. In comparison, C improvements must be distributed either as source-code patches or through pre-compiled binaries.

On the downside of GDB' PYTHON API, we can remark its youth. Indeed, at the beginning of this thesis work (early 2011), it was only a few years old, and some API bindings were found missing (we discuss in the next paragraph how we solved that issue).

Extending GDB's Python API In the previous paragraph we praised PYTHON development and disregarded the C side. However, it is important to note that only *extensions* can be developed in PYTHON; all the core, low-level and maybe performance-critical work must be done in C. In particular, the bindings between GDB internals and PYTHON environments must also be written in this language.

In order to implement our model-centric debugger in PYTHON, we extended GDB PYTHON interface to support all of our needs. In particular, we contributed the following patches:

- allow PYTHON to detect that multiple breakpoints were hit when the execution stops[3],
- add a "*getter*" to retrieve the process— `inferior` in GDB parlance—currently selected[4],
- add PYTHON notifications when a process exits[5], or when a shared-library is loaded[6],
- and finally, the most important patch of the serie was the introduction of the concept of *finish* breakpoint[7], crucial to our implementation design. We describe its rationals in the following section.

All of these patches were documented, tested against regressions and reviewed by GDB maintainers. (The review process looks at code correctness, coding style, documentation quality, *etc.*) After cycles of updates/reviews, the patches met expectations and hence were accepted for inclusion in the official code-base. Committing these patches in the GNU repository meant that, from the next version onwards, the official GDB releases would include our improvements. Furthermore, it also implies that, thanks to the non-regression tests, the community would ensure that the functionalities keep working as expected along the time.

2 PYTHON Success Stories, http://www.python.org/about/success/
3 https://www.sourceware.org/ml/gdb-cvs/2011-09/msg00085.html
4 https://www.sourceware.org/ml/gdb-cvs/2011-09/msg00086.html
5 https://www.sourceware.org/ml/gdb-cvs/2011-10/msg00019.html
6 https://www.sourceware.org/ml/gdb-cvs/2011-10/msg00034.html
7 https://www.sourceware.org/ml/gdb-cvs/2011-12/msg00230.html

In the following sections, we discuss the core of our model-centric debugger implementation, starting with the mechanisms used to capture the state of abstract machines.

4.2 CAPTURING THE ABSTRACT-MACHINE STATE AND ITS EVOLUTION

In order to monitor the state of the abstract machine running the application, the debugger needs to hook each of the operations affecting its state. In this section, we present the capture mechanism we implemented and analyze its efficiency. Then, we compare it against two alternative mechanisms and finally summarize their respective interests.

API Interception with Breakpoints and Debug Information

The information-capture mechanism we implemented for our debugger prototype relies on debugger-internal breakpoints set on the state-modifying operations of the environment API (*e.g.,* function symbols in C-based languages). Internal breakpoints are transparent for the user, who is not notified of the brief execution stop. When they are triggered, the debugger parses operation arguments with the help of the API definition and architecture calling conventions. Then, it decides to continue the execution or gives the control back to the user interface. Figure 4.2 presents a sequence diagram of these interactions. The concept of *finish* breakpoint was introduced in GDB Python API to programmatically catch the return point of a function. This allows the interception of output and updated parameters. If further information is required, and if application source-code and debug information (*e.g.,* Dwarf structures [Fre10]) are available, the debugger can use this knowledge to intercept implementation-specific knowledge about the abstract machine and better represent its state.

Evaluation The key benefit of this approach is that is does not require any additional debugging support from the application or from the programming-model supportive environment. Furthermore, the debugger can be developed with no cooperation with the supportive environment team. In the context of prototype development, this is a significant advantage

The portability of this mechanism depends on whether the debugger relies or not on source-code hooks, in addition to API interception. If it does not happen, the debugger might be version/vendor independent. Otherwise there is a strong dependency between the debugger and the supportive environment which limits the portability.

However, the main drawback of that approach is that the internal breakpoints may slow down application execution. This is particularly true for data exchange breakpoints, which can be triggered very frequently in a communication-intensive application. We further discuss this point in the concluding remarks in Section 4.5.

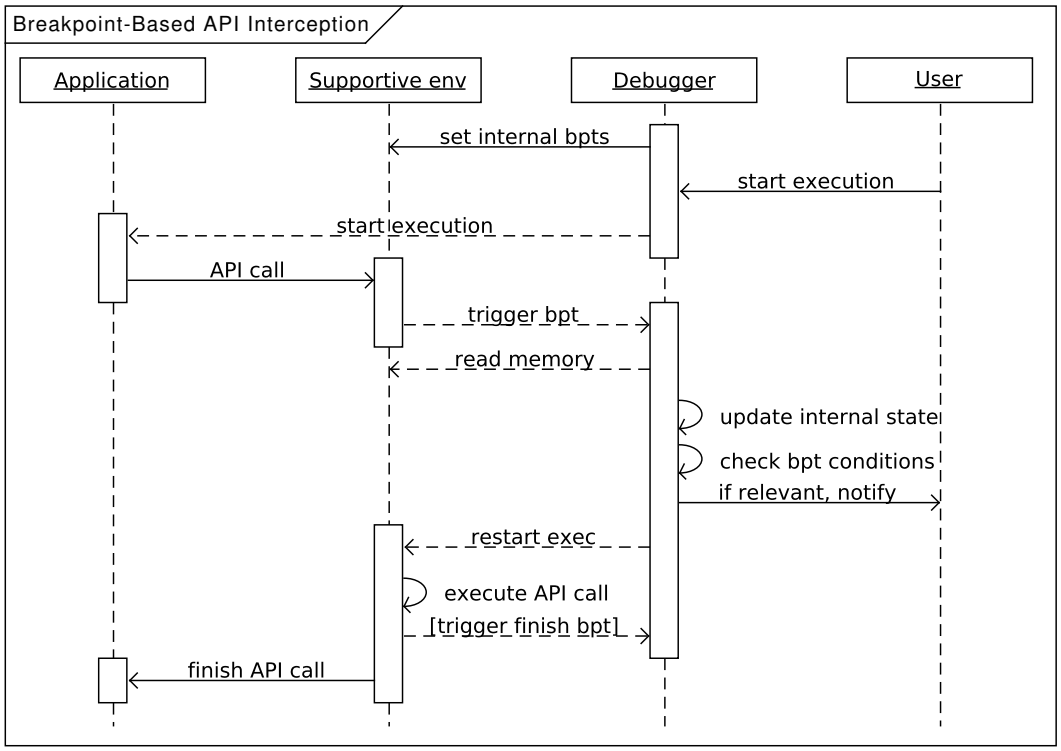


Figure 4.2: Diagram Sequence of Breakpoint-Based API Interception

Alternative Mechanisms

In order to build a more resilient debugger, different approaches can be considered.

API Interception with a Preloaded Library If the key objective of the debugger is to be portable across multiple environment versions and vendors, the debugger can preload a debugging library in the application memory space. The OpenCL debugger Gremedy Gdebugger [Gra10] and the SystemC one [RGDR08], presented respectively in Sections 7.3 and 7.1 rely on a similar capture mechanism.

In this case, the preloaded library implements debugging stubs for all the state-modifying operations of the environment API. These stubs collect the information required to follow the evolution of the abstract machine, and then redirect the execution flow towards the original environment function.

As breakpoints are only triggered when the debugger and the library need to exchange information, the execution speed is virtually unaffected. It also has a very good portability over different supportive environment versions and vendors as the capture mechanism only relies on the API definition. However, the amount of capturable information is limited to the parameters carried over the environment API. Although it is not exhaustive, the OpenCL and SystemC debuggers show that it can be sufficient.

Specialized Debug Module If there is no portability requirement, a specialized debug module inside the supportive environment code, can maximize the amount of capturable information. Thread debugging in GNU/LINUX systems [Molo3] relies on this design, with the debug module generically named `libthread_db` (See Subsection 4.1.1 and [PPCJ10]).

This capture mechanism involves a strong collaboration between the environment and the debugger, and hence between their respective development teams. In this mechanism, the environment debug module prepares and provides the information about the abstract machine to the debugger. As the environment implements the abstract machine, it has access to all the relevant internal states. Thus it is the most suitable location to gather these details.

SUMMARY

Table 4.1 summarizes the benefits and drawbacks of these capture mechanisms against the following criteria:

Amount of Capturable Information How much abstract machine state information can be captured?

Execution Overhead How is the execution slowed down by internal breakpoints?

Cooperation Between Debugger and Environment Does the debugger team need to cooperate with the environment developers to implement the debugging support?

Portability Can we use the debugger with another environment version or vendor?

	Breakpoints and Debug Information	**Preloaded Library**	**Specialized Debug Library**
Capturable Information	High	Limited to API	**Full**
Execution Overhead	Significant	Limited	Limited
Cooperation btw. Debug and Env.	**None**	Low	Strong
Portability	Variable	**Very Good**	Vendor Specific

Table 4.1: Information Capture Mechanisms Trade-Off

Once a capture mechanism has been chosen for the implementation of the debugger, the next step consists in creating a representation for the programming-model abstract machine, and animating it according to execution events.

4.3 MODELLING THE APPLICATION STRUCTURE AND DYNAMIC BEHAVIOR

In this section, we present the data structures and mechanisms we developed to represent the application architecture and follow the evolution of the programming model's abstract machine.

4.3.1 *A Ground Layer for Communicating Tasks*

In order to offer a generic ground layer for the representation of application relying on a communicating-task model, we designed a set of classes, presented below. These classes, schematized in the upper layer of Figure 4.1, can be extended to fit programming models requirements. They consist in `entity` objects holding connection `endpoints` bound together through `links`. `Entities` can exchange `messages` over these `links`:

`Entity` objects represent the different tasks of the programming model. They are usually schedulable and bound to an execution context of the execution platform. They hold the list of `Endpoint` objects associated with the task they reflect.

`Endpoint` and `Link` objects represent the relationship between the tasks of the model. They can transfer `Message` objects from one entity to another, according to the abstract machine specification.

`Message` objects are transmitted between `Entity` objects. They are usually not associated with a particular object of the environment; they only reflect the logic of the abstract machine operations (*e.g.*, message transmitted to a link, message received by its target, ...).

Developers of new model-centric debuggers can derive and specialize this abstraction layer to fit the representation of their communicating tasks models, as we do in the following chapter for component and dataflow programming.

On the other side, models like kernel-based accelerator programming are too different from communicating-tasks models and cannot directly benefit from this work. However, in the case of kernel-based programming, the application structure is less complex (no notion of graph, interface connection or messages) and hence easier to model.

4.3.2 *Following Dynamic Behaviors*

Once we had modelled the application structure in the debugger, the next step consisted in updating the representation according to the evolution of the abstract machine state. We presented in the previous section the low-level aspects of this task.

At the *representation* layer, we attach a PYTHON function to internal breakpoints set at supportive environment functions entry point and, if necessary, exit points. Consider for instance the following dataflow code snippet. Functions `next()` and `send(...)` are provided by the dataflow environment to send and receive data tokens, respectively:

```
flg = ctlr.next()
...
out_1.send(treat(cnt))
```

Different steps are required to follow the state of the abstract machine during their execution:

1. set a breakpoint on the function call,
2. wait for the breakpoint hit. Upon this event,
   a) identify the source actor (here the actor executing the code),
   b) the target (`out_1` , connected to the data dependency of the same name),
   c) and the token value (the result of `treat(cnt)`);
   d) create a message (token) object with the content found at the previous step, and push it into the relevant data link,
   e) continuing the execution*.

* The function is non blocking and does not return any answer, so it is not necessary to wait for the function to complete.

Regarding function `next()` , the steps are similar, except that the message object already exists and the function call is can block until a message is available:

1. set a breakpoint, wait for its hit and identify the source and target,
   a) set the actor/interface state to " `blocked` ".
   b) if relevant, check for deadlocks (see Paragraph 3.3.2 in the previous chapter)
2. set a *finish* breakpoint at the exit point and wait for its hit. Upon that,
   a) pop a token for the corresponding data link and push it into the target actor
   b) (internally, verify that the token content is identical to the return value)
   c) set the actor state to " `working` ".

Model-Centric Catchpoints In addition to these operations, the internal breakpoints can also check for model-centric catchpoints. For instance, after the n^{th} token sent over a given interface, or with a specific payload, *etc.*

GDB PYTHON breakpoint interface looks as follows (its is based on PYTHON class inheritance):

```
class MyBreakpoint(gdb.Breakpoint):
    def __init__(self):
        gdb.Breakpoint.__init__(self, "function", internal=True)

    def stop(self):
        return True or False # according to requirements
```

Hence, stopping or continuing transparently the execution simply consists in returning a boolean flag. Our framework provides a more advanced class, named `FunctionBreakpoint` that model centric debugger have to extend:

```
class FunctionBreakpoint:
    def __init__(self, spec) # automatically internal
    def prepare_before(self) # returns (stop, finish, data)
    def prepare_after(self, data) # return stop
```

These "function breakpoints" are initialized with the specification (name or address) of the function they target. When the execution hits the entry-point breakpoint, the debugger calls the method `prepare_before`. This method should return a triplet, indicating (1) if the execution must be stopped, (2) if the finish breakpoint must be set, and (3) some data to provide to the second callback. If asked with boolean (2) method `prepare_after`, is called when the execution hits the exit-point of the function, with the data parameter (3).

In the next section, we move out of the debugger internals and present how we built the interface between the model-centric debugger and its users.

4.4 INTERACTING WITH THE ABSTRACT MACHINE

Programming model-centric debugging tries to raise interactive debugging to the level of the model's abstract machine. Hence, interacting with the abstract machine should be as easy as it is with the hardware machine. The examples presented below are only for illustration purpose. We borrowed them from the following chapters where they are thoroughly explained.

4.4.1 *Application Structure*

Once the debugger has internally modelled the application structure and can dynamically capture its evolution, it needs to present it to the user. We can distinguish three alternative ways for this presentation, varying in terms of user comfort versus environment complexity:

Pure Command-Line Interface The command-line is GDB's main and almost[8] exclusive native user interface. This is the simplest and most portable interface, which can be used in any debugging environment. This mode only accepts textual information, hence all the structural elements have to be flatted down. As an example, the interconnection of Figure 4.3(b) between filters `pipe`, `ipred` and `ipf`, part of module `pred`, can be listed as follows:

```
(gdb) info filters pred pipe ipred ipf +interfaces
#8 pred (module)
        Owner pipe
        Owner ipred
        Owner ipf
#11 pipe
        Ownee pred
        Pipe2AddLumaMB_out PedfArrayStreamOut<LumaMB_t>  [> #12]
        LumaCBF_out        PedfArrayStreamOut<LumaCBF_t> [> #14]
#12 ipred
        Ownee pred
        Pipe2AddLumaMB_in  PedfBaseDataStream            [< #11]
        LumaResNotNull_out PedfArrayStreamOut<LumaCBF_t> [> #14]
#14 ipf
        Ownee pred
        LumaResNotNull_in PedfBaseDataStream             [< #12]
        LumaCBF_in        PedfBaseDataStream             [< #11]
```

The implementation of these listings follows naturally from the data structures presented in the previous section. This textual representation was sufficient for our study of component programming, however it rapidly shown its limits with the complex graphs of dataflow programming.

Command-Line and Static Image The second alternative involves delegating the graph rendering to a dedicate tool. We used for this purpose GraphViz[9] and its graph description language, DOT. Figure 4.3 highlights how the structure of the previous paragraph is described in DOT (4.3(a)) and the image generated from this description (4.3(b)). Figure 3.2 corresponds to a more complex graph, rendered with the same tool.

Our debugger implementation generates DOT description files, then defers the rendering and displaying of the image to system tools. This means that people in charge of deploying the debugger can easily adapt these last steps the their environment. However, the drawback of this approach is that the graph viewer offers no interactivity, and graph manipulation falls back to the command-line interface. An improvement for that is discussed in the following subsection, with command-line completion.

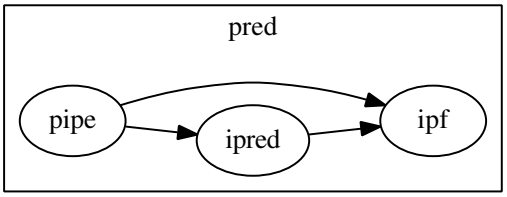
8 We let apart the TUI (Text User Interface), which would have required fuzzy ASCII art drawing for our purposes.

9 Graph Visualization Software, http://www.graphviz.org/

```
digraph G {
  rankdir = LR;
  subgraph  cluster_top_pred {
    label = pred
    pred_pipe [label="pipe"];
    pred_ipred [label="ipred"];
    pred_ipf [label="ipf"];
  }

  pred_pipe->top_pred_ipred;
  pred_pipe->top_pred_ipf;
  pred_ipred->top_pred_ipf;
}
```

(a) source code



(b) graph structure

Figure 4.3: Simple Graph Representation with GraphViz

Integrated Graphical Interface The last alternative would consist in extending one of GDB's graphical interfaces. In particular, Eclipse CDT (C/C++ Development Tools) and its DSF (Debugger Services Framework) appear as an interesting solution. In this case, developers could use the different parts of the graph to query the application state or set catchpoints, similarly to setting a breakpoint by clicking on a source-code line.

Our current implementation did not focus on such a integrated graphical interface, however it is part of future development work. Indeed, ST IDTec team plans to integrated model-centric debugging within its development tool, STWorkbench[10] .

4.4.2 *Model-Centric Command-Line Interface Integration*

In order to build the user-interface of our model-centric debugger, we leveraged GDB Python command-line extension capabilities. GDB's Python API allows extensions to offer context-aware command completion, which reinforces the feeling that the abstract machine environment is well integrated in the debugger.

The listing in Figure 4.4[11] presents how command-line completion benefits from the graph knowledge:

① the completion proposes the names of PEDF H.264 filters. Filter `pipe` will be selected.

② the completion lists the possibles catchpoints. We want to stop on the next outgoing token: `catch send_to`.

10 STWorkbench Integrated Development Tool — http://www.st.com/web/en/catalog/tools/PF250516

11 The keyword `filter` is a synonym of *actor* in PEDF dataflow dialect.

```
(gdb) filter <TAB>                                          ①
bh red imvp pipe hwcfg ipred ipf
(gdb) filter pipe catch <TAB>                               ②
work send_to receive_from
(gdb) filter pipe catch send_to <TAB>                       ③
ipred ipf
```

Figure 4.4: Setting Catchpoints With Command-line Completion

③ the completion takes the context into account (catching token outgoing from filter `pipe`) and suggests the two possible endpoints, filters `ipred` and `ipf`.

4.4.3 *Time-base Sequence Diagram*

The last part of the developer-debugger interactions consists in drawing time-base sequence diagrams to describe how the application interacts with the abstract machine. For our prototype implementation, we decided to emphasize the visual aspect of the diagram and directly aimed at graphical tools. We chose to rely on an ST internal data-viewer named TChartLite. TChartLite is part of STWorkbench, an Eclipse environment customized to ST boards, and already used for data visualization in tools like KPTrace[12]. Figure 4.5 illustrates how we used TChartLite to represent the configuration and execution of an accelerator kernel.

To preserve the command-line nature of our debugger, we decided to decouple as much as possible the debugger implementation from the viewer, running in STWorkbench. Hence, the information required for the visualization are streamed into a unidirectional pipe (for instance in Unix environments, named piped, Unix or network sockets). All the control logic remains in the debugger, and STWorkbench passively manages the visualization work.

As noted earlier, the entire model-centric logic could also be integrated into a graphical debugger environment, but this is out of the scope of this thesis.

4.5 EVALUATION AND CONCLUSION

In this chapter, we presented how we developed our model-centric debugger as an extension of source-level debugging. We explained that we chose GDB for it is a powerful and popular free debugger. However, as we illustrated in Figure 4.1, any source-level debugger can be used for this purpose. We expect only a few capabilities

12 Dynamic system tracing with KPTrace — STLinux, http://www.stlinux.com/devel/traceprofile/kptrace/

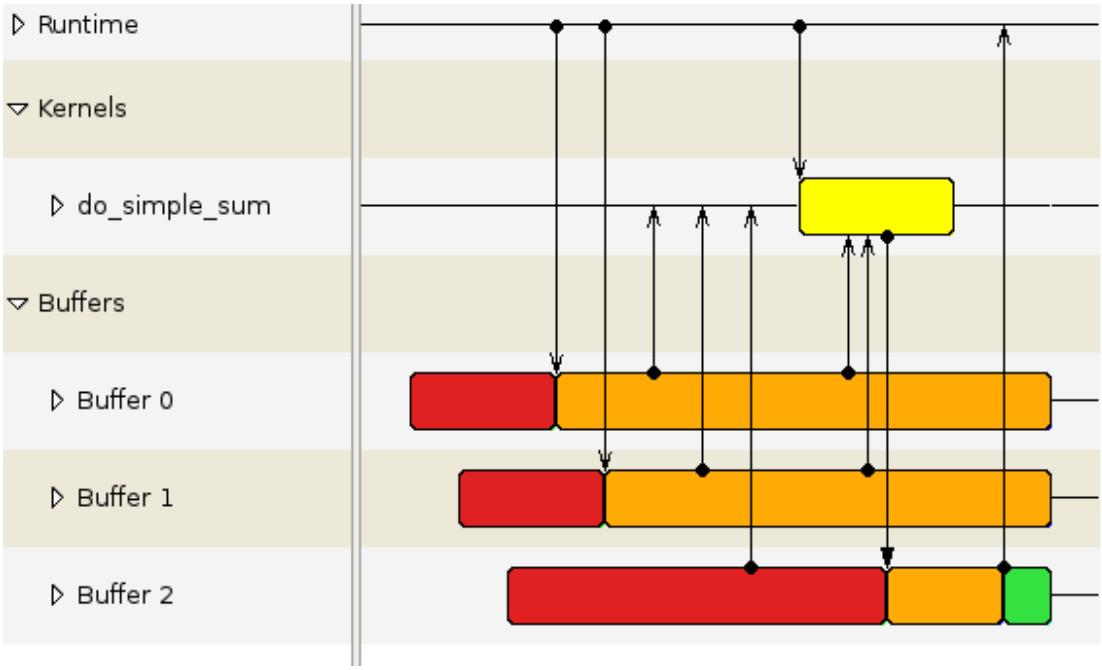


Figure 4.5: Time-base Sequence Diagram of the Configuration and Execution of an Accelerator Kernel.

from this debugger, materialized with the vertical arrows of the figure. First and foremost, it has to be extensible, *e.g.*, through a scripting API or directly in its code base. Secondly, regarding the interactions with the platform, the debugger must allow setting user-transparent breakpoints and examining the different memory spaces (variables, registers, ...).

We explained that the capture mechanism of our tool relies on such internal breakpoints to intercept API function calls. However, this approach can slow down applications' execution. In order to cope with this problem, our debugger allows developers to temporarily disable data-exchange breakpoints, until the execution reaches the "*critical part*" of the code. Architecture modification operations usually do not rely on the same breakpoints so they can still be used. Source code entity-specific breakpoints and watchpoints can also help reaching the suspicious area faster. Alternatively, we could also change the capture mechanism to a more efficient one.

Another limitation of our implementation is the implicit requirement that the underlying source-level debugger must have access to the entire memory-space of the application. The development of our debugger was done in the context of applications running under MPSoC platform simulators, hence this requirement was easy to meet. In different situations, such as debugging a heterogeneous application running on a real-board, it might be more difficult to find a suitable source-level debugger.

In the following chapter, we study how we interpreted and implemented the model-centric principles drawn in Chapter 3 in the context of the programming environment of STHORM, ST's MPSoC platform.

5

MCGDB, A MODEL-CENTRIC DEBUGGER FOR AN INDUSTRIAL MPSOC PROGRAMMING ENVIRONMENT

Resolution Elements

In the previous chapter, we detailed the key software blocks of a model-centric debugger. In this chapter, we continue this practical study with the presentation of Model-Centric GDB (mcGDB), our model-centric debugger prototype. For each of our three programming models, we come back on the programming environment descriptions of Chapter 2, Section 2.2.2, and detail the associated debugging features.

We first discuss in Section 5.1 Sthorm's component framework, with an emphasize on the dynamic aspect of component deployment and management, as well as inter-component communications and following the flow of messages.

Then we dig into Sthorm's dataflow framework in Section 5.2 and highlight how the graph structure is graphically presented to developers. We also discuss PEDF specific scheduling capabilities and how mcGDB can represent them and control the execution flow.

Lastly, we focus on OpenCL kernel programming in Section 5.3 and how we can debug it. In this environment, we insist on how we represent the application dynamic behavior through a graphical sequence-diagram-like representation of the interactions. We also discuss how we extended mcGDB's OpenCL module to support NVidia Cuda, a similar yet commercial competitor programming environment.

We conclude this chapter in Section 5.4 with an overview of the common and diverging aspects of these three implementation and elements of work estimation for adapting mcGDB to other programming environments.

5.1 NPM COMPONENT FRAMEWORK

In this section, we come back on NPM, Sthorm's component programming environment. We continue the NPM introduction of Chapter 2, Section 2.2.2 and further explain its component deployment and management interface, as well as inter-component communication. For each of these aspects, we present the debugging features we developed. This implementation study was part of publication [PSMMM12].

5.1.1 *Component Deployment and Management*

The deployment of runnable NPM components is done in two phases, from the host side of the application. A component is first instantiated on a given cluster ①. At this stage, the application can configure it and connect its different interfaces. Then, the application triggers the component execution (`run` method ②), once or several times, and finally destroys ③ the component instances, to release their memory.

```
int  NPM_instantiate (const char*  name, int target_cluster,
                      void **instance);                          ①
void NPM_runRTMComponent (void * instance);                      ②

int  NPM_destroy (void *instance);                               ③
```

✓ MCGDB catches the events corresponding to these functions and allows developers to stop the execution at these points:

```
(gdb) component [ID|name] catch {instantiate|run|destroy}
```

We name the components according to the file in which they were compiled (parameter `name` in ①), and the instances are identified by a unique number. MCGDB also interprets the indication about the target cluster (parameter `target_cluster`) and displays it in the user interface. We chose to represent the host part of the application as a normal component to unify the handling of the different entities.

During the debugging session, components can be in three different execution states, as illustrated in the state diagram in Figure 5.1:

No execution context The component has been instantiated and not yet destroyed, but it is not currently running.

Running The component is executing its `run` method and currently scheduled into an execution context.

Asleep The component is inside its `run` method but currently unscheduled.

Handling *asleep* components demands an additional attention, as their processor context is stored inside the abstract machine memory, rather than inside the OS memory. Our previous work [PPCJ10] and [VMD04] detail how to locate and switch between SystemC user threads, based on XPG4 `ucontext` functions[1].

If we focus on the implementation of the `NPM_runRTMComponent` hook ②, we can foresee with the function prototype that two steps will required to capture all the information about the event. First, we need to set a breakpoint at the function entry

1 See `/usr/include/ucontext.h` for further details.

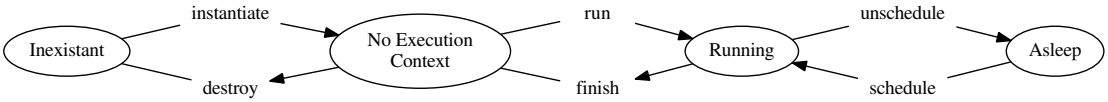


Figure 5.1: State Diagram of NPM Component Debugging.

point, in order to capture the component name, target cluster and the *address* of the component instance handle (`*instance`). After that, we must release the execution flow and wait for the function completion. If the function returns successfully (*i.e.,* return code is 0), then we can record the actual instance handle (`**instance`).

NPM allows developers to bind components at runtime from the host, with predefined communication mechanisms. A "*DMA controller*" link ④ transfers data from the host to the fabric (*i.e.,* to the components); whereas a First In First Out (FIFO)-buffer link ⑤ transfers data between two components, either using shared memory inside a cluster or inter-cluster communication mechanisms:

```
int NPM_instantiateDMAPullBuffer (NPM_DMAPullBuffer_t *bufferId,
            void *consumerComp, char *consumerInputItf, ...);       ④
...
int NPM_instantiateFIFOBuffer  (NPM_fifoBuffer_t *bufferId,
            void *producerComp,  char *producerOutputItf,
            void *consumerComp,  char *consumerInputItf, ...);      ⑤
```

✓ Upon such events, MCGDB detects that the abstract machine is about to connect two components, and updates its internal representation accordingly. In the current version of implementation, this is also the occasion for MCGDB to discover component interfaces.

For instance, a `DMA pull` binding ④ connects the consumer interface (`consumer-InputItf`) of the component `consumerComp` to the pseudo-interface `bufferId` of the host component. Parameter `consumerInputItf` corresponds to the name of the interface. Developers will be able to refer to this name during the debugging session. The same idea applies to `DMA push` (not shown here) and `FIFO` bindings ⑤.

It is also possible to catch further details about the link configuration such as the buffer size and location, or the access pattern. 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.

5.1.2 *Communication Interfaces*

As mentioned above, NPM components 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. The following excerpt presents the definition of `Pull` and `Push` interfaces, written with MIND[2] architecture description language:

```
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 ();                                              ⑥

    /* 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);                                  ⑦

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

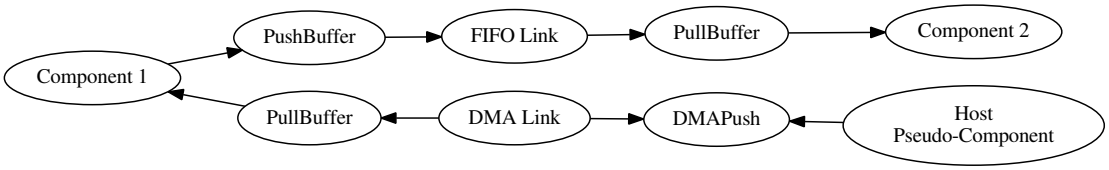


Figure 5.2: NPM Component Communication Interfaces.

Figure 5.2 presents the interconnection of two components and the host-side of the application.

2 MIND - OW2 Consortium http://mind.ow2.org/

✓ In McGDB, we consider the `Pull` and `Push` 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 `PullBuffer pull` ⑥ and `PushBuffer push` ⑦.

These primitives give McGDB the ability to detect communication events, hence allowing developers to control the execution based on the messages sent and received by the components. Hence, developers can set catchpoints on component interfaces ⑧. They can also achieved a finer control grain with condition checking, such as filters on the message payload or target. Lastly, a message counter is incremented every time a message the interface transmits a message.

```
(gdb) component [ID|name] interface name {next|catch_all}          ⑧
```

5.1.3 *Message-Based Flow Control*

In this last subsection on the implementation of McGDB's NPM component module, we detail how we implemented the message-based flow control capabilities presented in Chapter 3, Section 3.3.1.

We built this message-based flow control of McGDB upon NPM `Push` and `Pull` interfaces, in the continuation of what we presented in the previous subsection.

In the debugger code, we implemented the message management logic inside the `endpoint` classes. Hence, when a component calls a `PushBuffer push` method, McGDB invokes the corresponding Python method. In this case, we read information about the message (*i.e.*, the buffer) and push it into the link object connected to this endpoint.

At some point, the component on the other side will call its `PullBuffer pull` method to receive the message. If this occurs *before* the message was sent, the component execution will be blocked. On the debugger side, the entry breakpoint will be hit, but the exit one will be delayed until the message actually arrives. When the exit breakpoint is triggered, McGDB knows that a message arrived.

All the NPM links operate on a FIFO mode. Hence, McGDB's *link* objects just push and pop messages from a queue to represent the message exchanges.

McGDB's interface offers the following commands for message handling:

```
(gdb) message {enable|disable}-tracking                             ⑨
(gdb) info component [ID|name] +messages                            ⑩
(gdb) info links [ID] +messages                                     ❶
(gdb) info message [ID]*                                            ❷
```

```
(gdb) message forget ((ID)*|all)                    ❸
(gdb) message {follow|unfollow} (ID)*               ❹
```

First of all, message tracking must be activated with command ⑨. Then, commands ⑩ and ❶ list the messages contained in a given component or a given link, whereas command ❷ list them all. Command ❸ hides a message from the listings, and command ❹ tells the debugger to stop the execution each time a given message is processed (*i.e.*, received or sent).

However, this last command ❹ is of limited use in the current situation: messages can only move from one component to another (that would trigger two stops). Hence, the following step in the debugger development consisted in allowing components to transmit messages.

To this purpose, we design an *overlay* API, that developers can implement to inform MCGDB about component routing schema:

```
class ComponentOverlay:
  def consume_message(self, endpoint, msg)
  def produce_message(self, endpoint)
  def get_messages(self)
```

During the component application initialization, MCGDB looks up component overlays (currently with name comparison). In case of success, it will replace its generic message methods by the ones from the overlay. They will be called in the following order:

1. `consume_message` is called when the component receives a message from a given endpoint. It can store it internally, mark it and/or delete it,
2. `get_messages` should return the list of messages held in the component,
3. `produce_message` either generates a new message by reading/decoding the memory or re-send one it stored earlier. It can access the graph structure and in particular the message outgoing endpoint.

This aspect is further discussed and exemplified in NPM case study, in Chapter 6, Section 6.1.4.

We continue this implementation study with STHORM dataflow programming environment, PEDF.

5.2 PEDF DYNAMIC DATAFLOW

In this section, we first present how we built the dataflow graph of Figure 5.3 with PEDF and detail the debugging features associated with each of the steps. Then, we come back on PEDF scheduling capabilities and filters' execution. This implementation study was part of publications [PLCSM13a, PLCSM13b].

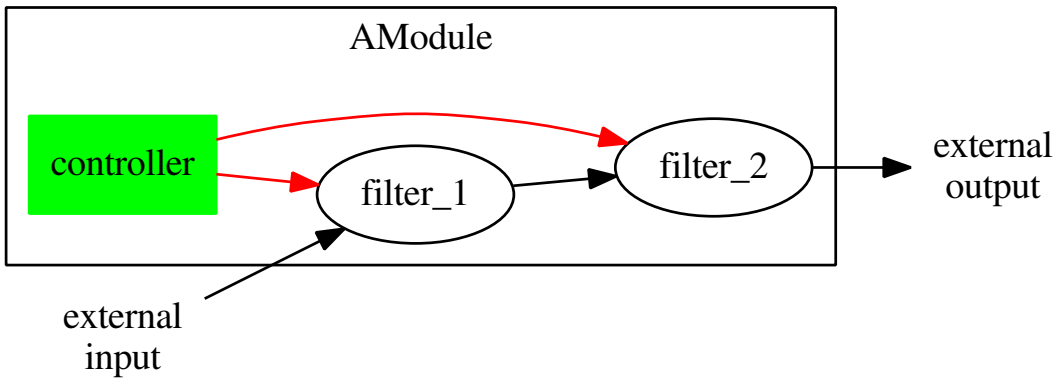


Figure 5.3: PEDF Dataflow Graph Visual Representation of a Simple Module.

5.2.1 *Graph Reconstruction*

PEDF dataflow graph is built with MIND[3] architecture compilation tool-chain, augmented with PEDF annotations. MIND provides a description language to specify filter's architecture and interfaces. Its compiler generates a C++ version of the architecture, based on PEDF and platform-specific templates. PEDF defines three classes of entities:

Filter It is a computing entity, corresponding directly to the *actors* of the dataflow model. Filters have inbound and outbound data links. The code of a filter is written in a subset of the C language which will be eventually synthesized into a hardware accelerator.

Controller There is one controller per module, which is responsible for the scheduling of the module's filters (*i.e.*, registering filters for execution to the runtime system), according to the application algorithm. A controller runs on a cluster controller core of the fabric.

Module It corresponds to a sub-graph of filters and a controller. Like filters, modules have inbound and outbound data links, corresponding to the unconnected arcs of the inner graph. Thus, modules can be hierarchically interconnected.

The following code snippet, from which Figure 5.3 was generated, presents the definition of module `AModule`. It contains a controller and two filters, and it defines two external connections. The filter definition is presented afterwards. In the last lines of the module definition, we can see how the different connections are bound together.

```
@Module
composite AModule {
  contains as controller {
```

3 This is a corner-case usage of MIND, which primarily targets component programming. Here, it is mainly used for is architecture description and deployment capabilities

```
    output U32 as cmd_out_1;
    output U32 as cmd_out_2;

    source ctrl_source.c;
  }

  // External connections
  input  U32 as module_in;
  output U32 as module_out;

  // Sub-components
  contains AFilter as filter_1;
  contains AFilter as filter_2;

  // Connections
  binds controller.cmd_out_1 to filter_1.cmd_in;
  binds controller.cmd_out_2 to filter_2.cmd_in;

  binds this.module_in     to filter_1.an_input;
  binds filter_1.an_output to filter_2.an_input;
  binds filter_2.an_output to this.module_out;
}
```

Filters `filter_1` and `filter_2` are both defined by the primitive type `AFilter` presented below. They have private and attribute data, as well as an input and an output data dependencies. They also have a control input dependency.

```
@Filter
primitive AFilter {
  data      stddefs.h:U32 a_private_data;
  attribute stddefs.h:U32 an_attribute;
  source    the_source.c;

  input  stddefs.h:U32 as an_input;
  input  stddefs.h:U8  as cmd_in;
  output stddefs.h:U32 as an_output;
}
```

✓ In MCGDB, this graph structure is dynamically reconstructed during the initialization phase of the framework. To this purpose, we extended the generic definitions of entities, links and endpoints presented in Chapter 4, Section 4.3.

Entities We extended the `Entity` class to support PEDF's filter, module and controller definitions. A parent class implements the module definition with its different dependency links, and two sub-classes define the filters and controllers, with attributes specifying if the actor is running, currently scheduled, a pointer to its execution context and another to its PEDF C++ object instance.

Endpoints and Links In the implementation of McGDB, we distinguished three kinds of inter-entity links and endpoints:

Data streams transmit data tokens between filters,

Control streams transmits control tokens between a filter and a controller,

Ownership links materialize the owner relationship between modules and filters or controllers. They do not transmit information.

Additionally, to correctly support data streams modelling, we implemented a concept of link bridges, which helps building the module interconnection (*i.e.,* the "*external*" links of Figure 5.3).

In the current version of the implementation, McGDB uses PEDF internal C++ classes to perform the graph reconstruction. We set a breakpoint at the end of the graph initialization, and parse the different objects to discover the application architecture.

During the debugging session, McGDB presents the dataflow graph upon user request. Besides, graph information will be provided to developers through most of the dataflow-related functionalities of the debugger. Auto-completion capabilities make it straightforward for developers to use filter and connection names while they are typing their commands.

5.2.2 *Scheduling Monitoring*

Module controllers are responsible for triggering the execution of the filters of their module. Filter execution model is based on execution *step*s. For each *step*:

1. The controller decides which filter must be executed: `ACTOR_START(name)`.
2. The `WORK` method of filters scheduled for execution is started.
3. The controller can wait for the actual beginning of the execution: `WAIT_FOR_ACTOR_INIT()`.
4. The controller can request filters to stop their execution at the end of this *step*: `ACTOR_SYNC(name)`.
5. The controller can wait for the actual end of the *step*: `WAIT_FOR_ACTOR_SYNC()`.

(NB: `START` and `SYNC` commands can be merged into a single `ACTOR_FIRE` command).

This scheduling capability is not part of the common dataflow models, although it shares some similarities with control tokens, with the exception that the deterministic property is lost.

✓ McGDB captures this information, so that developers can quickly review which filters are ready to be executed, not scheduled, or have already finished the *step*.

Developers can also request an execution stop at the beginning or end of a *step*, or when a controller schedules a filter for execution:

```
(gdb) controller [name|ID] catch_next  {FIRE|WAIT_INIT|WAIT_SYNC|...}
(gdb) filter     [name|ID] catch_next  {SCHEDULE|WORK}
```

We also initiated a joint-work on a *live* visual representation of PEDF execution, in cooperation with ST's PEDF tools development team. Figure 5.4 shows how their visualization environment displays a trace-based (*i.e.*, post-mortem) PEDF execution. Hence, our first goal is to connect this environment with our model-centric debugger, so that it can provide a *live* visualization of the execution. In a second step, we expect to improve this time chart with additional model information, such as the data and control tokens exchanged by the different entities.

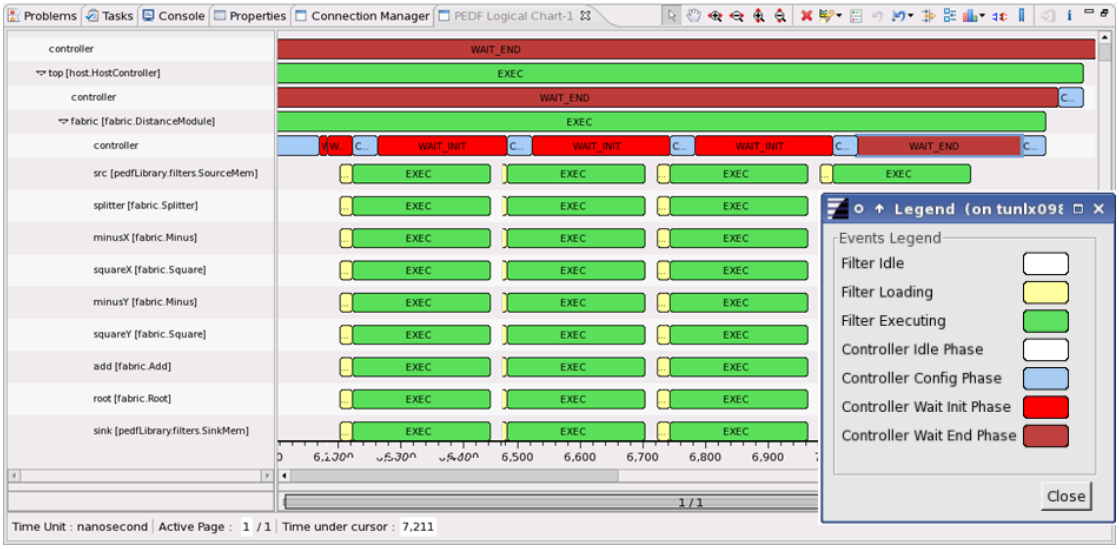


Figure 5.4: PEDF Trace-Based Time Chart of Filter Scheduling.

5.2.3 *Filter Execution Flow Control*

PEDF filters implement the core data processing tasks of the application. They are intended to be synthesized into hardware accelerators, and for this reason, strong constraints have been defined for their implementation. In particular, the use of a restricted subset of the C language, which permits a direct transformation to Register Transfer Level (RTL) circuits. Filters must define a `WORK` method, implementing one *step* of the processing. They can access their private data, attributes and connections with the name specified in the architecture definition, prefixed by `pedf.data.`, `pedf.attribute.` and `pedf.io.`, respectively.

With respect to the dataflow part, the data exchanges are transparent to the developers. In the previous example, a filter can read (respectively write) its data with

`pedf.io.an_input[n] = d;` where `n` is the highest unread (respectively unwritten) index. (This array notation corresponds to the *structure* model of dataflow mentioned earlier.)

✓ In MCGDB development, we focused on the flow-of-token aspect, which is key to the dataflow model. Namely, we enabled the possibility of following a token through a dependency by intercepting the indexes of the token pushed in and out of the link. As the model and the implementation ensure that the data order is preserved, we can stop the execution at the right location in a deterministic way:

```
(gdb) filter  [name|ID] catch_next WORK
(gdb) filter  [name|ID] interface  [name|ID]  {next|break}
(gdb) filter  [name|ID] {info|follow} last_token
```

Internally, PEDF relies on C++ operator overloading to implement its communication mechanisms based on square brackets. Hence, we had to set breakpoints at the entry and exit points of these function to capture token emission and reception:

```
template<class ArrayStream, class T>
class PedfTokenSelector {
  T &operator=(const T &val) { ... }
}

template<class T>
class PedfArrayStreamOut: public PedfBaseDataStream {
  T &operator[](int idx) { ... }
}
```

In the following section, we study our last programming environment, OpenCL, which lies at the boundary between MPSoC embedded computing and HPC.

5.3 OPENCL KERNEL PROGRAMMING

As we noted in our study of model-centric debugging for kernel-based programming (Chapter 3, Section 3.3.3), we do not consider the accelerator side of the application, but only its CPU/host code. We introduce in the Related Work chapter (Chapter 7) complementary tools which explicitly target GPGPU kernel debugging.

Hence, MCGDB's debugging support relies exclusively on OpenCL's C API. This scope limitation allowed us to develop a capture mechanism independent of the vendor library implementing the standard.

```c
void checkStatus(int *ptr, char *msg) {
  if(ptr == 0) exit(-1);
}

void simple_sum (int *a, int *b,  int *c) {
  c = *a + *b;
}

void main() {
  int *a = malloc(sizeof(int));
  checkStatus(a, "Couldn't allocate a");
  *a = 5;

  int *b = malloc(sizeof(int));
  checkStatus(b, "Couldn't allocate b");
  *b = 10;

  int *c = malloc(sizeof(int));
  checkStatus(c, "Couldn't allocate c");

  simple_sum(a, b, c);

  printf("c: %d n", c);
  // free ...
}
```

(a) C version

```
cl_context ctx = /* ... */;
cl_command_queue commandQueue = /* ... */;
cl_kernel k_simple_sum = /* OpenCL version of simple_sum.  */;

int a = 5, b = 10, c;
cl_mem d_a, d_b, d_c;

/* Allocate the buffers of the GPU.   */
d_a = clCreateBuffer(cxt, CL_MEM_READ_ONLY, sizeof(int), NULL, &err);
checkStatus(err, "clCreateBuffer d_data failed");
d_b = clCreateBuffer(ctx, CL_MEM_READ_ONLY, sizeof(int), NULL, &err);
checkStatus(err, "clCreateBuffer d_data failed");
d_c = clCreateBuffer(ctx, CL_MEM_WRITE_ONLY, sizeof(int), NULL, &err);
checkStatus(err, "clCreateBuffer d_data failed");

/* Push the values to the GPU memory.  */
err = clEnqueueWriteBuffer(commandQueue, d_a, CL_TRUE, 0,
                                    sizeof (int),  &a, 0, NULL, NULL);
checkStatus(err, "clEnqueueWriteBuffer d_data failed");
err = clEnqueueWriteBuffer(commandQueue, d_b, CL_TRUE, 0,
                                    sizeof (int),  &b, 0, NULL, NULL);
checkStatus(err, "clEnqueueWriteBuffer d_data failed");

/* Set the kernel parameters.  */
err = clSetKernelArg(k_simple_sum, 0, sizeof(cl_mem), (void *) &d_a);
checkStatus(err, "clSetKernelArg d_a");
err = clSetKernelArg(k_simple_sum, 1, sizeof(cl_mem), (void *) &d_b);
checkStatus(err, "clSetKernelArg d_b");
err = clSetKernelArg(k_simple_sum, 2, sizeof(cl_mem), (void *) &d_c);
checkStatus(err, "clSetKernelArg d_c");

/* Trigger the kernel execution.  */
size_t global_work_size[] = 1;
err = clEnqueueNDRangeKernel(commandQueue, k_simple_sum, 1, NULL,
                                global_work_size, NULL, 0, NULL, NULL);
checkStatus(err, "clEnqueueNDRangeKernel do_simple_sum failed");

/* Get the result back.  */
err = clEnqueueReadBuffer(commandQueue, d_c, CL_TRUE, 0, sizeof(int),
                                                    &c, 0, NULL, NULL);
checkStatus(err, "clEnqueueReadBuffer d_sum failed");
printf ("sum: %d", c); // then free ...
```

(b) OpenCL version

Figure 5.5: C and OpenCL Versions of a Simple Computation

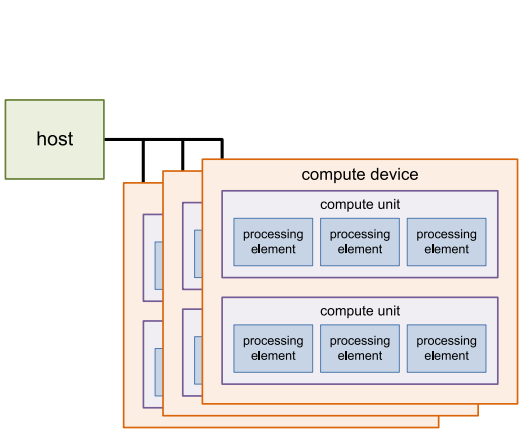
We can see with the code length and density that the OpenCL version of the code would be more difficult to understand and therefore to debug.

In the following, we present how we implemented MCGDB's debugging support for OpenCL applications. We start with an overview of the architecture representation and the control of the execution, then we present with more details our execution visualization environment.

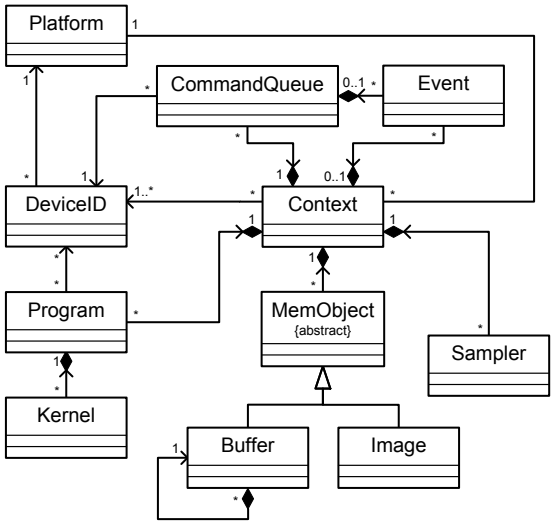
5.3.1 *Architecture Representation and Execution Control*

OpenCL relies on a platform abstraction, as presented in Figure 5.6. In Figure 5.6(a), we can distinguish the host side of the application, running on the CPU, as well as compute devices. This could be a GPU card, or STHORM computing fabric. In the context of STHORM, OpenCL compute units correspond to a fabric cluster, composed of eight processing elements. However in MCGDB, we did not focus on this side of the execution.

Figure 5.6(b) present the UML[4] representation of OpenCL data types and their relationships: a `Platform` contains devices (`DeviceID`); the application can deploy program binaries (`Program`) containing kernels (`Kernel`) on a device; it must also create memory objects (`MemObject`), such as buffers (`Buffer`) or images (`Image`), bound to device contexts (`Context`). OpenCL provides command queue structures (`CommandQueue`) to drive the accelerator execution (memory transfers, kernel executions, ...). Command queue can generate runtime events (`Event`) providing information about the queue processing.



By Björn König, Wikimedia Foundation, Licence CC-by-sa 3.0/de



Taken from OpenCL v1.1 specification [Khr08]

(a) platform architecture (b) UML class diagram

Figure 5.6: OpenCL Platform Abstraction.

4 Unified Modeling Language http://www.uml.org/

✓ In McGDB's current implementation, we mainly focused on detecting OpenCL kernels and buffers, as well as the operation pushed into the command queue. We let apart platform, device and context distinctions, as our development environments have a single accelerator. Taking them into consideration would only involve additional commands in the same spirit.

To that purpose, we hooked the relevant functions of OpenCL API and implemented the correspond information capture mechanisms:

```
cl_program clCreateProgramWithSource (context, ..const char **str);  ①
cl_kernel clCreateKernel (program, const char *kernel_name, ..));    ②
cl_int clSetKernelArg (kernel, arg_index,..., const void *arg_val);  ③
cl_int clEnqueueNDRangeKernel (command_queue, kernel, work_dim);     ④

cl_mem clCreateBuffer (context, ..size_t size, void *host_ptr, ..);  ⑤
cl_int clEnqueueReadBuffer (command_queue,
                            cl_mem buffer, ..., void *ptr, ...); ⑥
```

From the debugger event generated by the execution of these functions, we capture a graph of object instances, the relationship between them their logical state. Hence, McGDB can provide developers with details such as where a kernel was created ⑦, from which program, and the same for programs and buffers ⑧ (from functions ①, ② and ⑤, respectively). When the application enqueues a kernel execution ④, they can list of parameters set to this kernel ⑦ (from function ③), and link them, if relevant, to the corresponding buffer object ⑧.

```
(gdb) info programs
(gdb) info kernels [name|ID] {+where|+params}                        ⑦
(gdb) info buffers [ID] {+where|+params}                             ⑧
```

Regarding the execution control, developers can set catchpoints on kernel operations ⑨ or memory transfers ⑩.

```
(gdb) kernel [name|ID] catch {all|enqueue|set_arg|...}               ⑨
(gdb) buffer [ID] catch {all|transfer|read|write|set_arg|...}        ⑩
```

In the following subsection, we discuss the execution visualization environments we designed for McGDB's OpenCL debugging module.

5.3.2 *Execution Visualization*

We presented at the beginning of this section (Figure 5.5(b)) that an OpenCL code is dense and harder to read than its C counterpart. Hence, understanding and debugging

such codes is more difficult than usual. To limit this effect, we design a *model-aware* execution visualization environment to improve interactive debugging. Our goal here was to represent graphically the interactions between the application and the supportive environment, so that developers can quickly review what is happening during the application.

Figure 5.7 is a screenshot of the environment as it was at the end of the execution of Figure 5.5(b). In the chart, the time goes from left to right, with events successively added to the plot. We can read, chronologically:

- three buffers were created. Color red means that they are not initialized.
- buffers 0 and 1 were populated. Their color changed from red to orange.
- buffers 0, 1 and 2 were set as kernel parameters
- kernel `do_simple_sum` was triggered. The arrows on the yellow box indicate that the kernel may be interacted with the buffers. Buffer 0 and 1 are read-only, hence their content could not change. Buffer 2 is write-only, so the kernel execution should have populated it (hence the color change)
- the content of buffer 2 is read from the host. The green color indicates that the content has been saved.

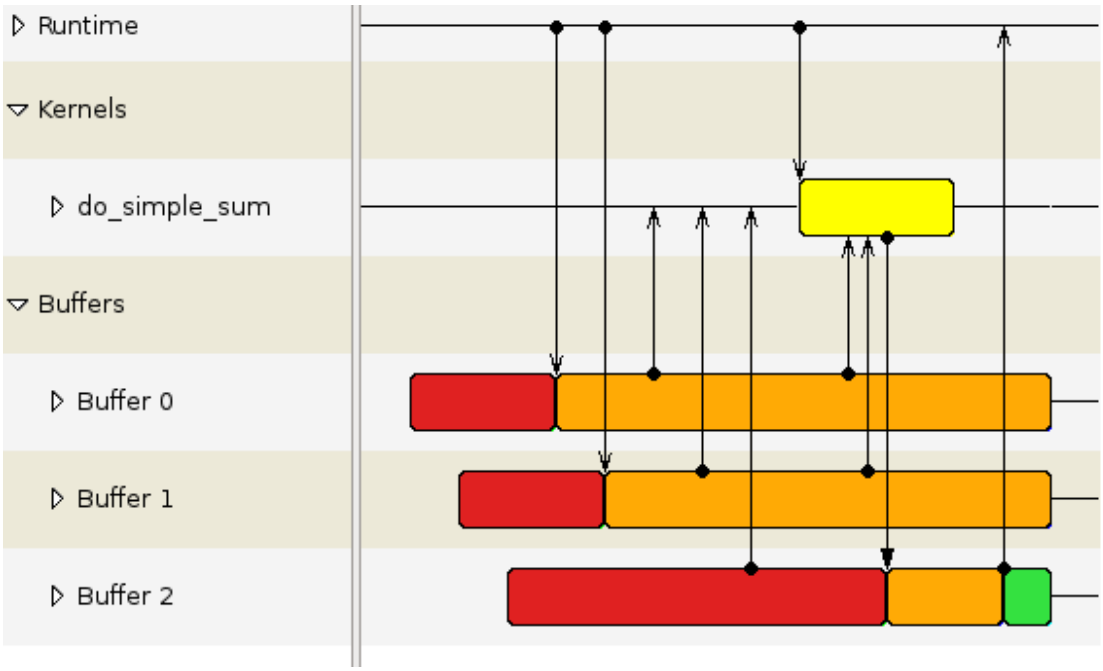


Figure 5.7: Time-base Sequence Diagram an OpenCL Kernel Execution.

We already presented the generic building blocs for the implementation of this visualization tools in the previous chapter, Section 4.4.3. The part specific to OpenCL consisted in gathering event information (*i.e.*, name of the function, buffer and kernels involved, ...) within the model-centric capture mechanisms. On the visualization side, it consisted in coding the graph updates for all the events we considered. For instance,

a memory transfer event creates a link between the runtime and the corresponding buffer object, as well as changing its state.

In the last part of this section about McGDB's OpenCL module implementation, we present how we adapted the OpenCL module code to support another kernel-based programming model, NVIDIA CUDA.

5.3.3 *Portage to* NVIDIA CUDA

NVIDIA CUDA is a kernel-based programming environment for NVIDIA GPU processors. It relies on a programming model very close to OpenCL, hence it allows us to demonstrate that, for a given programming model, only a minimal engineering effort is required to adapt the debugger to support one environment or another[5].

To highlight the similarities and distinctions between both environments, we go through the main functions used to capture OpenCL abstract machine state updates and present how they translate in CUDA environment. One important distinction between these environments is that CUDA relies on its own (pre-)compiler. Hence, some of our debugging hooks were constructed by analyzing intermediate compilation files. CUDA compiler version `5.5.0` supports it with `nvcc`'s `--keep` flag.

KERNEL CREATION AND EXECUTION

OpenCL relies on the following functions to create and execute kernels:

```
cl_kernel clCreateKernel(..., const char *name, ...)
cl_int clSetKernelArg(cl_kernel kernel, cl_uint arg_index, ...)
cl_int clEnqueueNDRangeKernel(cl_kernel kernel, cl_uint work_dim, ...)
```

CUDA kernels are not explicitly instantiated in the code, but rather included as standard functions. However, the compiler appears to generate "device stubs", in charge of triggering the kernel execution on the GPU. These stubs share a common prefix, `__device_stub__`, followed by the C++-mangled name of the kernel. With the help of GDB symbol lists, we can extract the kernel names and use function breakpoints to get execution notification:

```
(gdb) info functions __device_stub__
All functions matching regular expression "__device_stub__":
```

5 In this work, we only considered the *runtime* CUDA API. Its *driver* API is open for future work.

```
File hello.cudafe1.stub.c:
void __device_stub__Z4helloworldPcPi(char*, int*);
```

This function call event also allows us to capture the parameters of the kernel execution. The last key information regarding the kernel execution is the number of GPU cores required to run the kernel. In the original source code, this setting is passed to the kernel in a CUDA-specific syntax:

```
helloworld<<<dimGrid, dimBlock>>>(ad, bd);
```

The analysis of intermediate files reveals that thus particular function call is rewritten into the following C code:

```
(cudaConfigureCall(dimGrid, dimBlock)) ? ((void)0) : helloword(ad, bd);
```

Setting a breakpoint on function `cudaConfigureCall` allows us to capture the information we required, but also detect configuration failures[6].

MEMORY OBJECT MANAGEMENT

OpenCL memory objects are managed with the following families of functions:

```
cl_mem clCreateBuffer(size_t size, void * host_ptr, ...)
cl_int clEnqueueReadBuffer(cl_mem buffer, ...);
cl_int clReleaseMemObject(cl_mem memobj)
```

They translate seamlessly into CUDA API:

```
cudaError_t cudaMalloc (void **devPtr, size_t size)
cudaError_t cudaMemcpy (void *dst, void *src, enum cudaMemcpyKind kind)
cudaError_t cudaFree (void *devPtr)
```

These few adaptions of the OpenCL debugger were sufficient to have a working prototype of a model-centric CUDA debugging, as we present in Chapter 6, Section 6.3.2, with CUDA-base scientific computing case study.

6 In this case, there is no obvious way to link the failure with the kernel that was supposed to run.

In the last section of this chapter, we draw our conclusions on the implementation of model-centric debugging on MPSoC programming environments, and on the workload it requires to port a debugger implementation toward another environment.

5.4 CONCLUSION

In this chapter, we studied how to implement the model-centric debugging principles defined in Chapter 3. We went through three distinct programming models and their MPSoC supportive environments and highlighted the axes we followed to adapt the model-level propositions into actual debugging commands.

To conclude the implementation chapter, we wanted to come back to the question of how generic this implementation is, or conversely, how difficult would it be port the implementation to another programming model and environment.

The answer to that question is that the closer the programming models are, the easier it is. Consider first dataflow and component programming. Both models are based task, connected through a more-or-less variable graph structure. Hence, an important part of the model analysis and debugger implementation will be shared.

Now, consider dataflow and kernel programming. The models share very few properties. Hence, only the completely generic parts of the debugger will be shared, that is, event breakpoints, user-interface constructs, *etc.*

Finally, let us consider a last example, two supportive environments implementing the kernel-based accelerator programming model: OpenCL and NVIDIA CUDA. Although these environments are developed and maintained by independent (and competing) companies, the adaptation of our OpenCL debugger and execution visualizer to support the key functionalities of CUDA was carried out in less that a day.

Another aspect of the portability question concerns the execution platform. In the current implementation, we only considered the functional simulators of STHORM, as neither the more accurate ones nor the actual board where available. However in the future work, and in particular for PEDF, we plan to extend the debugger to support multiple execution platforms. Indeed, as most of the implementation of the debugger is platform agnostic, model-centric debugging could be use monitor and verify the execution of PEDF application with hardware or RTL filters. Developers would not be able to control the internal parts of the filters execution, but the majority of the debugging features would remain unaffected.

In the following chapter, we develop three case-study illustration to highlight the functionalities and usage of our model-centric debugger.

6

CASE STUDIES

The Adventures

In this chapter, we exercise McGDB and demonstrate the various abilities of model centric debugging, in the context of real-life case studies. We start in Section 6.1 with an example of a debugging session of a component-based feature tracker. This feature tracker is part of an augmented reality application running on Sthorm. Then in Section 6.2, we use McGDB to study the execution of a dataflow implementation of the H.264 video decoding standard. The last section of this chapter, Section 6.3, goes through two scientific applications relying on the kernel computing. These applications are part of the European project Mont-Blanc[1]. We first highlight the debugger representation of a material physics density-functional-theory solver for electronic structures calculations (Subsection 6.3.1), then we detail a debugging session of a 3D seismic wave propagation simulator in Subsection 6.3.2.

The time frame of the thesis work did not allow us to provide development teams with our tool from the *beginning* of application development, which would have been the best conditions to highlight the benefits of model-centric debugging. Nevertheless, this chapter presents an alternative yet important use-case, which takes place during the *maintenance* part of the application life-cycle. In this use-case, we assume that developers do not have an extended knowledge about the application (*e.g.*, because of team switch-overs, lack of documentation, *etc.*), and McGDB is used to discover and understand the dynamic aspects of the application execution. In Section 6.1.5, though, we describe how we understood an unsolved bug in the component feature tracker.

6.1 COMPONENT-BASED FEATURE TRACKER

This section presents a case study which illustrates the abilities of McGDB's component module. 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 Sthorm.

1 Mont Blanc Project, European Approach Towards Energy Efficient High Performance http://www.montblanc-project.eu/

Figure 6.1: Feature Tracking Between Two Images.

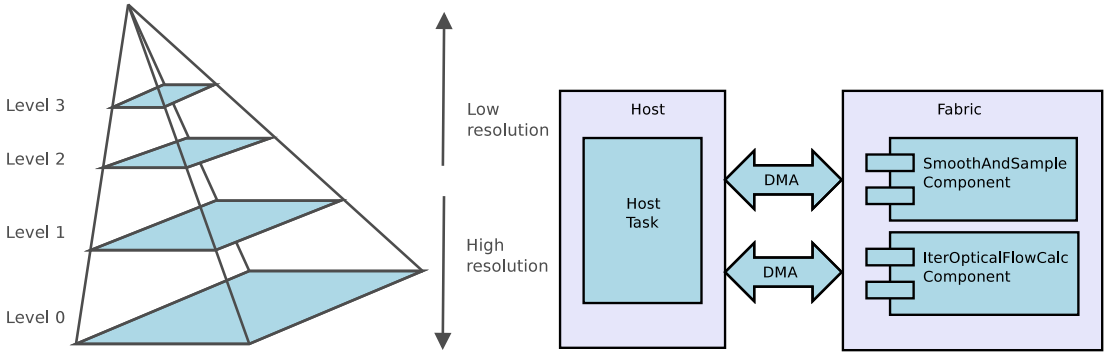
6.1.1 *Pyramidal Kanade-Lucas Feature Tracker*

Feature tracking consists in identifying interesting points (*features*) in an initial image and following their motion in the subsequent images (*tracking*). Figure 6.1 presents a visual representation of several features tracked from the left image to the right one. *Bouguet* explained the algorithm in [Bouoo], which is divided into two parts. First, the images are sub-sampled, with different ratios, to create a pyramidal representation, as shown in Figure 6.2(a). The bottom of the pyramid is the largest image, the top is the smallest. Then, the feature tracking 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 MCGDB presents the information of the current architecture to the developer. Finally, it explains how we leveraged MCGDB to detect a communication bug in the application implementation.

6.1.2 *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 and the host-side task:



(a) image pyramid (b) component architecture

Figure 6.2: PKLT Internal Structures

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 a *previous* and *next* images in input, as well as the features of the previous level. 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 back and forth between the components.

Both components execute their core algorithm on up to sixteen shared-memory cores of a STHORM cluster. Figure 6.2(b) presents a schematic representation of this architecture.

6.1.3 *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 console output extract below also shows 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[SmoothAndSample]]
...
[Stopped on 'run' method of #2 Component[SmoothAndSample]]
```

Then developers can list the currently known components, along with their interfaces (`info components +itf`). There are two components in the console extract 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 notice that the components are connected through two interfaces (the DMA links).

```
(gdb) info components +itf
  #1 Host[31272]
        Name    Id
        DMAPush/0x8050f7c
        DMAPush/0x8050fbc
...
```

```
* #2 Component[SmoothAndSampleProcessor]
        Name              Type
        srcPullBuffer     (PullBuffer)
        dstPushBuffer     (PushBuffer)
...

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

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

In the following, we continue with the debugger representation of the messages exchanged over the different PKLT components.

6.1.4 *Message-Based Flow Control*

In the previous chapter (Section 5.1.3), we described the implementation mechanisms behind message-based flow control. In this subsection, we come back on this aspect and highlight how it can be used for PKLT debugging.

As an example, let us consider a message *sent* (we omit all the information about the *receive* operation) by the `SmoothAndSample` component to the host, as shown in the code snippet of Figure 6.3[2]:

- we set a breakpoint at the beginning of this code `(gdb) component 2 catch run`
- and wait for the debugger to reach it `(gdb) continue`

2 `CALL(itf, method)` is a NPM preprocessor macro, which takes as first parameter the name of an interface of the component, and as second parameter the name of the method to call. It returns the method itself, so the methods parameters should be placed in following.

```
/* Copy the first block to compute and output */
PRIVATE.src = CALL(srcPullBuffer, pull)();
PRIVATE.dst = CALL(dstTmpPushBuffer,getBuffer)();                    ①
CALL(srcPullBuffer, fetchNextBuffer)();

// Processing Loop
for (i = 1; i <= iterations; i++) {
  NPM_dup_job(PRIVATE.nbProcs, METH(HorizontalFilter), NULL);        ②

  /* Transmit computed data while processing */
  CALL(dstTmpPushBuffer, push)(PRIVATE.dst);                         ③

  /* Copy the next block to compute */
  if (i == iterations) { // if last iteration
    ...
  } else {
    PRIVATE.src = CALL(srcPullBuffer, pull)();
    PRIVATE.dst = CALL(dstTmpPushBuffer, getBuffer)();
  }

  CALL(srcPullBuffer, release)(PRIVATE.src);
  CALL(srcPullBuffer, fetchNextBuffer)();
}
```

Figure 6.3: Code Snippet From Component Smooth-And-Sample Source Code.

- we enable message debugging: `(gdb) message enable-tracking`
- ... and continue the execution line-by-line. At line ①, the debugger detects that the component owns a new message:

```
(gdb) next
PRIVATE.dst = CALL(dstTmpPushBuffer, getBuffer)();
[Message #1 created in #2 Component[SmoothAndSampleProcessor]
(gdb) info component 2 +messages
  #1 Message created by #2 Component[SmoothAndSampleProcessor
```

- then we continue the execution until line ③ (buffer `PRIVATE.dst` is populated inside the function call at line ②, according to the content of buffer `PRIVATE.src`)
- at line ③, the component pushed the message through its `dstTmpPushBuffer` interface:

```
(gdb) next
CALL(dstTmpPushBuffer, push)(PRIVATE.dst);
[Message #1 sent by #2 Component[SmoothAnd...] to #1 Host[...]
(gdb) info component 2 +messages
  No message.
(gdb) info links 4 +messages
#4 DMALink
  DMAPull/0x8050ffc            #1 Host[31272]
  PushBuffer/dstTmpPushBuffer #2 Component[SmoothAndSample...]
  #1 Message sent through #2 Component[Smoo...]/dstTmpPushBuffer
```

- now, we tell the debugger to follow the message and continue the execution,

```
(gdb) message 1 follow
(gdb) continue
...
[Message #1 received by #1 Host[31272]]
/* Wait asynchronous events */
NPM_waitRTMComponent(&smoothAndSampleProcessor.rtmComp);
```

- the debugger stopped the execution at the end of the component run, when it is sure that the DMA has finished the data transfer:

```
(gdb) info message 1
  Message 1 created by #2 Component[SmoothAndSampleProcessor]
          sent through dstTmpPushBuffer
```

```
          received by #1 Host[31272]
                   on interface DMAPush/0x8050ffc
```

- finally, we can query additional information about the interface. This shows that the DMA wrote the messages into buffer `p_tmp_out`.

```
(gdb) info component 1 +itf DMAPush/0x8050ffc
  #1 Host[31272]
        Name    Id
        DMAPush/0x8050ffc
        Configured at npm_pklt.c:127:
err = NPM_instantiateDMAPushBuffer(
          &smoothAndSampleProcessor.dstTmpPushBuffer,
          smoothAndSampleProcessor.rtmComp.appComp.comp,
          "dstTmpPushBuffer", (void*) p_tmp_out,
          dstImgWidth*nbProcs*sizeof(unsigned char),  2);
```

To finish this subsection, we illustrate the idea of user-defined routing table.

User-Defined Routing Table

We have seen in Figure 6.3 that component `SmoothAndSample` receives messages, transforms them and sends them back to another interface. However, McGDB cannot detect on its own that the messages reaching and leaving the component are connected. Hence, as the persons in charge of PKLT debugging, we provided the following class to McGDB:

```
class MySmoothAndSampleComponent:
    ...
    def do_consume_message(self, endpoint, msg):                    ④
        if endpoint.name == "srcPullBuffer":
            self.lastSrc = msg
        if endpoint.name == "srcTmpPullBuffer":
            self.lastTmp = msg
        msg.checkpoint("%s <- %s" % (self, endpoint.name))

    def do_produce_message(self, endpoint):                         ⑤
        if endpoint.name == "dstTmpPushBuffer":
            msg = self.lastSrc
        elif endpoint.name == "dstPushBuffer":
            msg = self.lastTmp
        else:
```

```
            return None

        self.src = None
        msg.checkpoint("%s -> %s" % (self, endpoint.name))
        return msg

    def get_messages(self):                                          ⑥
        return (self.lastSrc, self.lastTmp)
```

This class specifies how to list the message it owns ⑥ and what to do when messages arrive ④ in the `SmoothAndSample` component, when it leaves ⑤. We can see that the two latter methods respectively save and return a message, based on the name of the endpoint involved. They also log a comment in the message checkpoint queue.

Eventually, when a message will have transited back and forth between the host and the component, McGDB will be able to display the following information:

```
(gdb) info message 1
  Message #1 created by #1 Host[31272]
         a. sent through interface DMAPush/0x8050f7c
         b. received by #2 Component[SmoothAndSampleProcessor]
                     on interface srcPullBuffer
         c. sent through interface dstTmpPushBuffer
         d. received by #1 Host[31272]
                     on interface DMAPull/0x8050ffc
```

In the next subsection, we conclude this component debugging case study with a description of how we used McGDB to discover the location of a data transfer error in the application.

6.1.5 *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[3], to limit its memory usage. Then the

3 The temporary interfaces we removed from the console output extracts in the previous sections to simplify readability.

temporary image is retrieved again from interface `srcTmpPullBuffer`, and a column by column "vertical" parallel filter is applied. The columns of the output image are pushed to interface `dstPushBuffer` 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 interface `destination`.

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[SmoothAndSample]]
```

Then, we asked McGDB 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, but interface `dstTmpPushBuffer` received one unexpected message, whereas `dstPushBuffer` was lacking one.

```
(gdb) info components +counts
~ #2 CommComponent[SmoothAndSampleProcessor]
    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`):

```
/* Compute last lines if necessary.  */
if(tmp_size > 0){
    ...
    /* Transmit the last lines computer.  */
    CALL(srcTmpPullBuffer, release)(...);
```

```
    CALL(dstTmpPullBuffer, push)(...);
}
```

This bug would have been tricky to detect without a model-centric 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 source-level debugger, would have highlighted the issue.

In the following section, we describe the debugging of a dataflow implementation of a H.264 video decoder.

6.2 DATAFLOW H.264 VIDEO DECODER

This section presents a case study which illustrates our approach in the context of a ST application: a H.264 video decoder [WSBL03] designed with PEDF to exploit STHORM heterogeneous computing fabric. In the following, we introduce the video standard, then we come back on the challenges highlighted in Chapter 2, Section 2.3.2 and explain how each point is addressed by MCGDB.

6.2.1 *H.264 Video Decoding*

H.264/AVC (Advanced Video Coding) [WSBL03] is a 2003 video coding standard approved by the ITU-T[4] and the ISO/IEC MPEG (Moving Picture Experts Group)[5]. It aims at supporting diverse application areas, such as broadcast over cable, satellite; optical and magnetic storage; conversational services over ethernet, wireless and mobile networks, *etc.* Hence, the video standard should offer both flexibility and customizability to address such broad needs. Besides, it also provides approximately a 50% bit rate saving in comparison with prior standards, for an equivalent perceptual quality.

As for the previous ITU-T and ISO/IEC video standards, the scope of H.264 is limited to the central decoder. This means that the standard sets restrictions on the bitstream and its syntax, but software vendors are free to optimize the other parts of the process (pre-processing, encoding and post-processing/error recovery) according to their specific requirements.

In this case-study, we only consider this H.264 central decoder, which was developed to validate PEDF design.

4 International Telecommunication Union, Telecommunication Standardization Sector. http://www.itu.int/en/ITU-T

5 The Moving Picture Experts Group website, http://mpeg.chiariglione.org/

6.2.2 *Graph-Based Application Architecture*

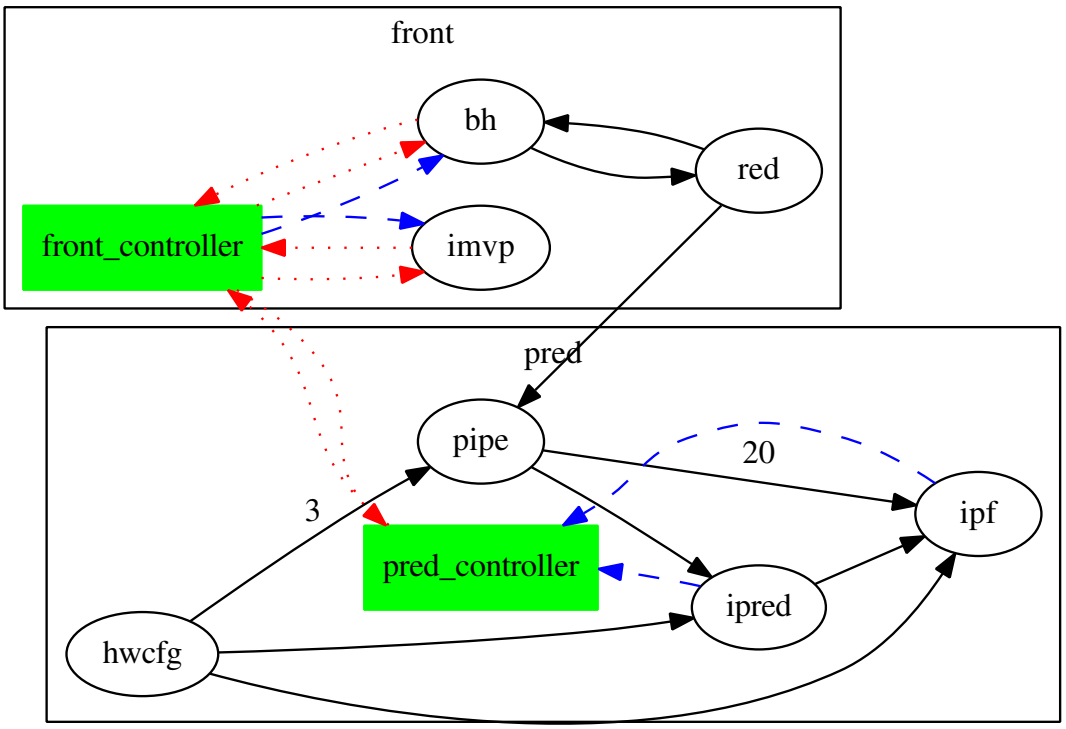


Figure 6.4: Graph of Dataflow Actors and Data Dependency of a H.264 Video Decoder

During the design of the STHORM H.264 decoder application, developers put a special focus on the module/filter decomposition. Indeed, as filters are intended to be synthesized into hardware accelerators, it was important to optimize their architecture and interactions. The graph in Figure 6.4 presents a graphical representation of tha application's dataflow architecture. It is composed of two modules, `front` and `pred`. Each module contains a controller (the green rectangular boxes) and a set of filters (the round boxes). The arrows connecting the different entities materialize data-dependencies. We can distinguish three different types: plain-line arrows are pure data links between hardware filters, whereas dotted and dashed arrows correspond to control links, which may be assisted by DMA controllers (the dashed lines).

As we mentioned earlier, this graph is a key element of the application architecture. Thus, it is available through most of the dataflow-related commands. See for instance the command ① in the next subsection, where filter and interface names where suggested by the auto-completion mechanism.

In the current implementation of MCGDB, the graph is plotted with GRAPHVIZ[6] DOT format and displayed with the system's default image visualizer. However, a debugger with a graphical interface could provide a more interactive view where the graph elements could be directly used to interact with the debugger.

6 http://www.graphviz.org/

6.2.3 *Token-Based Execution Firing*

As we described in the previous chapter (Section 5.2.2), PEDF dataflow model differs from the traditional models, in the sense that the execution of an actor is first of all conditioned by the triggering of a `fire` event by its module controller.

Developers can control it with the following *catchpoint* command. In this case, the execution will be stopped when the `WORK` method of Filter `pipe` is triggered:

```
(gdb) filter pipe catch work
```

Developers can also set a catchpoint stopping the execution when a filter has received a given amount of tokens:

```
(gdb) filter ipred catch  Pipe_in=1,  Hwcfg_in=1                    ①
(gdb) filter ipred catch *in=1                                      ②
```

These two commands stop the execution as soon as Filter `ipred` has received a token in both of its two inbound data links, `Pipe_in` and `Hwcfg_in`. The first command ① shows the explicit way, interface by interface, whereas the second one ② applies the condition to all the inbound interfaces.

6.2.4 *Non-Linear Execution*

During step-by-step execution of filter code, developers must pay a special attention on the dataflow assignments. Indeed, these instructions may enable and trigger the execution of other filters, dependent on this data. To accommodate with this eventuality, MCGDB provides the `step_both` command, which inserts a "double" breakpoint, at both ends of the link:

```
(gdb) list
220 // push add2dBlock to ipf
221 pedf.io.Add2Dblock_ipf_out[...] = ...;
(gdb) step_both
[Temporary breakpoint inserted after input interface
                                      'ipf::Add2Dblock_ipred_in']
[Temporary breakpoint inserted after output interface
                                     'ipred::Add2Dblock_ipf_out']
...
[Stopped after receiving token from 'ipf::Add2Dblock_ipred_in']
(gdb) continue
```

```
...
[Stopped after sending token on ‘ipred::Add2Dblock_MB_out‘]
```

In this console output extract, the execution was stopped right before the execution of a dataflow assignment (line `221`), where a token is sent though filter `ipred`'s `Add2Dblock_ipf_out` interface. The command `step_both` instructs MCGDB to stop both ends of the execution. The execution flow first reaches the `ipf` filter, on the other side of the data dependency. Then the developer asks to continue the execution and finally the second stop occurs, right after the assignment. The order of these two stops is implementation and architecture dependent.

6.2.5 *Token-Based Application State and Information Flow*

The fluency of the token flow in the overall application is an important concern for correctness and performance. If two filters connected by a data-dependency do not produce and consume tokens at the same rate, the application may stall because of link over/underflow. It can also lead to erratic results if the synchronization of multiple interfaces is not respected.

As an example, the graph presented in Figure 3.2 shows that the link `pipe` → `ipf` currently holds 20 tokens, which may indicate a problem in the sending or receiving rate. Link `hwcfg` → `pipe` contains three tokens, and all the other links are empty. If requested, MCGDB can record and display the *content* of the tokens (this feature requires a significant quantity of memory and thus has to be explicitly enabled):

```
(gdb) iface hwcfg::pipe_MbType_out record
...
(gdb) iface hwcfg::pipe_MbType_out print
 #1 (U16) 5
 #2 (U16) 10
 #3 (U16) 15
```

In this example, only three messages were recorded, but a communication-intensive filter may quickly generate a large number of tokens, impossible to record efficiently and useless for developers.

Filters can also exhibit clear patterns in their communication behavior. This characteristic can be exploited by MCGDB to improve the details provided to developers. Indeed, this allows following a token over several components. However, this behavior depends of the filter implementation and hence, MCGDB cannot figure it out automatically. The developer has to configure it manually.

For instance, filter `red` acts as a *splitter*: it receives data from filter `bh`, processes it and sends the data it generated to all of its outbound interfaces. The developer can inform MCGDB about it with the following command:

```
(gdb) filter red configure splitter
```

To better illustrate this feature, let place ourselves in a more concrete situation. Let us consider that there is an *observable* error at some point of the execution. With the help of the mechanisms previously described, the developer stops the execution as close as possible to the error trigger. For instance in filter `pipe`, after receiving a token from interface `Red2PipeCbMB_in`:

```
(gdb) filter pipe catch Red2PipeCbMB_in
...
[Stopped after receiving token from 'pipe::Red2PipeCbMB_in']
```

At this point, the developer ensures that the situation is actually erroneous and tries to understand where the fault came from. The information about the token path becomes useful:

```
(gdb) filter pipe info last_token
#1 red -> pipe (CbCrMB_t){Add=0x145D,...}                    ③
#2 bh  -> red  (U32)     127     },,                         ④
```

We can see that the last token was received from filter `red` (step ③), with a given value (`{Add=0x145D,...}`). If this value is incorrect, it means that the error arrived from filter `red`. Step ④ helps understanding the conditions in which this token was produced: after receiving an integer token (127) from Filter `bh`. To complete this information, further details about the filter state can be recorded, such as attribute values.

Recording token contents may appear excessive when querying the content of a single link (it could be directly read from the framework memory), however it becomes mandatory when we want to follow its path over multiple actors.

6.2.6 *Two-level Debugging*

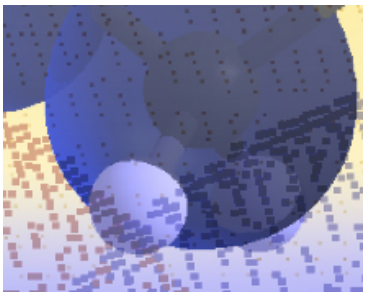
We are aware that model-centric and *dataflow*-centric debugging may not be enough to locate and understand all the possible problems which can occur during the execution of a dataflow application. For this reason, a traditional, full-flavored GDB is always available during the debugging session. McGDB extension only handles the dataflow-specific commands, and the underlying GDB manages the rest of the debugging environment. This means that when the execution is stopped, as in a previous example, the developer can ask McGDB to display the last token received and then use GDB to analyze its C structure and inner content:

```
...
[Stopped after receiving token from 'pipe::Red2PipeCbMB_in']
(gdb) filter print last_token
$1 = (CbCrMB_t){Addr=0x145D, ...}
(gdb) print $1
$2 = { Addr = 0x145D,
       InterNotIntra = 1,
       Izz = 168460492, ... }
```

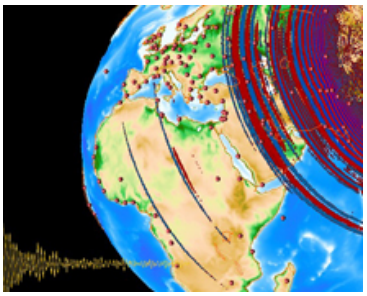
In the following section, we finish this case-study chapter with two scientific applications accelerated through OpenCL and CUDA programming.

6.3 GPU-ACCELERATED SCIENTIFIC COMPUTING

In this last section, we slightly drift apart from embedded computing and get closer to GPU-based HPC scientific computing. We first discuss OpenCL debugging in the context of density functional calculations (Figure 6.5(a)). Then, we present the first steps the portage of MCGDB's OpenCL module towards CUDA programming environment and illustrate the results with a geo-dynamic wave propagation simulator (Figure 6.5(b)).



(a) BigDFT density functional theory solver



(b) Specfem 3D wave propagation simulator

Figure 6.5: GPU-Accelerated Scientific Applications from Mont-Blanc Project.

6.3.1 *OpenCL and BigDFT*

BIGDFT [GOD+09, GVO+11] is a free project implementing density functional theory based on *Daubechies* wavelets. In this domain of physics and chemistry, scientists are interested by electronic structures calculations of systems with a large number of electrons. However, systems with only hundreds of atoms already require huge computational power. Hence, BIGDFT developers first sought for high performance in multi-CPU HPC platforms, but then realized that hybrid GPU-CPU architectures offer a very low price/performance ratio, more attracting for intensive scientific computation. They presented in 2009 [GOD+09] a first version for NVIDIA cards based on CUDA,

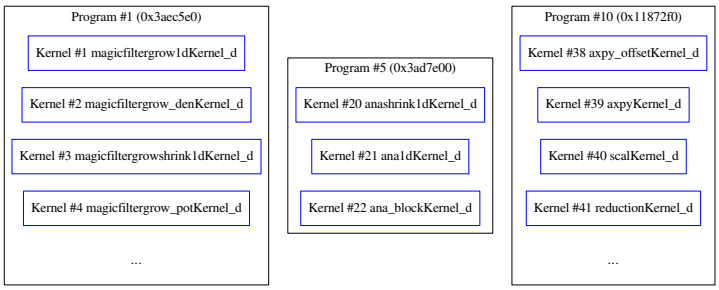


Figure 6.6: Excerpt of BigDFT Program/Kernel Structural Representation.

then a new version in 2011 [GVO+11], more complete, optimized and cross-platform, written with OpenCL.

In the following paragraphs, we study this OpenCL implementation (version 1.7-r24), and more particularly the function in charge of local partial density computation. We first introduce the structural representation of the application, then we present how the execution control mechanisms can be used. We finally illustrate the visualization capabilities of McGDB.

STRUCTURAL REPRESENTATION

The main elements of BigDFT's OpenCL structure are the kernels and programs, as the application only uses one execution context and one command queue. Figure 6.6 shows an excerpt of the debugger representation of three programs, respectively in charge of computing potential energy (`magicfilter*`), wavelet analysis (`ana*`) and linear algebra operations (reduction, scaling and summation). This information is also available in text mode, with the following command:

```
(gdb) info programs +kernels
```

Developers can query McGDB about further kernel and/or program details, such as the location (stack trace) where they were created ①, how many times they were used ②, or their OpenCL handle ③:

```
(gdb) info kernels 41 +where +use_count +handle
Kernel #41  reductionKernel_d
  Creation stack:                                                    ①
    #0 0x0912a13 in create_reduction_kernels (...) at Reduction.c:1365
    #1 0x0900a10 in create_kernels (...) at OpenCL_wrappers.c:95
    #2 0x0902744 in ocl_create_command_queue_id_ (...) at OpenCL_wra...
  Use count: 0                                                       ②
  Handle: 0x3aea590                                                  ③
```

In this example we can see that kernel `reductionKernel_d` was selected through its unique identifier (`#41`). However, kernel can also be selected with a name prefix, or through a handle look up. This last capability can be valuable to developers maintaining applications they do know not very well.

For example in BIGDFT reduction module, we can find codes similar to the following function:

```
void axpy_generic(cl_kernel kernel, ...)  {
   ...
   ciErrNum = clEnqueueNDRangeKernel (command_queue, kernel, ...);
}
```

A source-level debugger would not be able to provide any relevant information:

```
(gdb) print kernel
$1 = (cl_kernel) 0x3ae9d50
(gdb) print *kernel
$2 = <incomplete type>
```

Indeed, the OpenCL standard does not specify the content of the object handlers, and in our case, the vendors chose not to provide debug information for them. However, MCGDB keeps track of these handlers and can directly provide developers with more useful details:

```
(gdb) info kernels +handle=0x3ae9d50 +where
#39  axpyKernel_d
    #0  0x091293d in create_reduction_kernels (...) at Reduction.c:1361
    ...
  Handle: 0x3ae9d50
```

Coming back on kernel `reductionKernel_d` , developers may also need information about the source code that was used to generate this kernel. OpenCL offers the possibility to use precompiled binary code, or compile on-the-fly OpenCL-C code. BIGDFT relies on this latter option, and takes the chance to pre-optimize the code for the execution platform (the device type in this case—CPU or GPU and number of available work groups). Figure 6.7 presents these two version of the code: Figure 6.7(a) is the C version of the kernel code, as it reads in BIGDFT source code. We can remark that it is not straightforward to read, because of the string concatenations, new-line markers and dynamic code constructs. On the other hand, Figure 6.7(b) presents the source code actually provided to OpenCL, and bound by MCGDB to the kernel entity representation.

```
size_t max_wgs = infos->MAX_WORK_GROUP_SIZE;
size_t cutoff = infos->DEVICE_TYPE == CL_DEVICE_TYPE_CPU ? 4 : 64 ;
program << "__kernl void reduction_Kernel_d(uint n,\\n\
        __global double *y, __global const double *x,\\n\
                            __local volatile double *tmp ) {\\n\
size_t i = get_local_id(0);\\n\
size_t g = get_group_id(0)*" << max_wgs * 2 << "+i;\\n\
if (g + " << max_wgs << " < n) {\\n\
  tt = x[g + " << max_wgs << "];\\n\
  tmp[ i + " << max_wgs << "] = tt * tt;\\n\
} else\\n\
  tmp[i + " << max_wgs << "] = 0.0;\\n\
barrier(CLK_LOCAL_MEM_FENCE);  //... \\n\";
do {
  max_wgs /= 2;
  program <<  "if (i < " << max_wgs << ")\\n\
                  tmp[i] = tmp[i] + tmp[i + " << max_wgs << "];\\n\
               barrier(CLK_LOCAL_MEM_FENCE);\\n\";
} while (max_wgs >= cutoff);
```

(a) C generator code

```
(gdb) info kernels 41 +code
#41  reductionKernel_d
----------------8<-------------
__kernel void reduction_Kernel_d (uint n,  ...) {
  size_t i = get_local_id(0);
  size_t g = get_group_id(0) * 2048 + i;
  double tt;

  if (g + 1024 < n) {
    tt = x[g + 1024];
    tmp[i + 1024] = tt * tt;
  } else
    tmp[i + 1024] = 0.0;
  barrier(CLK_LOCAL_MEM_FENCE); //...
  if (i < 512) tmp[i] = tmp[i] + tmp[i + 512];
  barrier(CLK_LOCAL_MEM_FENCE);
  if (i < 256)  tmp[i] = tmp[i] + tmp[i + 256];
  barrier(CLK_LOCAL_MEM_FENCE); //...
}
```

(b) final OpenCL C version.

Figure 6.7: Two Versions of a Kernel Code.

The last OpenCL structural elements consist in the memory objects. Similarly to kernels, McGDB memorizes their creation parameters (read/write, *etc.*) and creation stack trace, however one distinction is that they do not have "official" names. Hence, we choose to parse the name of the variable where the handler is stored. In order to be generic, we allowed the name to be picked up at different places of the stack. For instance, some BigDFT's buffers are created with the following function call:

```
void FC_FUNC_(ocl_create_read_write_buffer,OCL_CREATE_READ_WRITE_...)
         (bigdft_context  *context, cl_uint *size, cl_mem *buff_ptr) {
  cl_int ciErrNum = CL_SUCCESS;
  *buff_ptr = clCreateBuffer((*context)->context, CL_MEM_READ_WRITE,
                             *size, NULL, &ciErrNum);
  ...
}
```

In this case, `buff_ptr` is a meaningless name. The actual buffer name lies one step above in the call stack. Indeed, as the code is mainly written in Fortran, function `ocl_create_read_write_buffer` is only a Fortran to C wrapper. Here, the buffer name can be find as the third parameter in the caller code:

```
call ocl_create_read_write_buffer(GPU%context, wfd%nvctr_c * 8,
                                  GPU%psi_c_i);
```

As this information is application specific, it has to be provided by the debugger user, for instance through a Python configuration file:

```
try_buffer_name(Events.CREATE_BUFFER, "ocl_create_read_write_buffer",
                depth=2, param=3)
```

Hence, with the help of this configuration information, we can provide developers with a more useful buffer list:

```
(gdb) info buffers +handle
# 1 GPU%psi_c (RW)
        Handle: 0x3b531a0
...
# 4 GPU%work3 (RW)
        Handle: 0x4a91910
# 5 GPU%d (RW)
        Handle: 0x511c8c0
...
```

```
# 9 GPU%keyv_c (RO)
        Handle: 0x5e34880
...
```

Now that we have presented how to display the structural aspects of the application, let us detail how to control the application execution.

EXECUTION CONTROL

As we presented in Chapter 5, Section 5.3.1, McGDB's execution control commands are mainly based on kernel and buffer handling. Hence, once kernels have been created, developers can set catchpoints on their command queue submissions:

```
(gdb) kernel reductionKernel_d catch enqueue
Catchpoint set on kernel #41 'reductionKernel_d' enqueue events.
(gdb) continue
...
Caught kernel 'reductionKernel_d' enqueue event.
```

At this point, developers can query the parameters associated with the kernel:

```
(gdb) info kernels +name=reductionKernel_d +key+params
#41  reductionKernel_d
  Arg 0: (cl_uint) 250        Arg 1: Buffer #4 GPU%work3
  Arg 2: Buffer #5 GPU%d      Arg 3: 0
```

We can notice here that OpenCL buffer arguments are converted to their debugger entity. Then, if we are interested in following buffer `#4` usage, we can set a catchpoint on the buffer and wait for the next execution stop:

```
(gdb) buffer 4 catch all
Catchpoint set on Buffer #5 GPU%work3 for all usages.
(gdb) continue
...
Caught buffer #4 usage.
```

To conclude this subsection on OpenCL and BigDFT debugging, we present how McGDB helps visualizing the execution process.

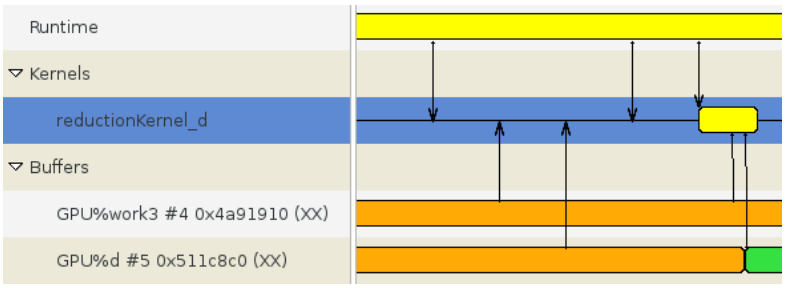
EXECUTION VISUALIZATION

In the previous paragraphs, we presented different elements of BIGDFT execution, and in particular the parameters associated with kernel `uncompress_coarseKernel_d` execution. Figure 6.8(a) shows how MCGDB visualization engine depicts these events. In left-hand side of the figure, we can distinguish four entities: the runtime application, one kernel and two buffers. All the other kernels and buffers were hidden when we selected kernel `uncompress_coarseKernel_d`, to improve the readability.

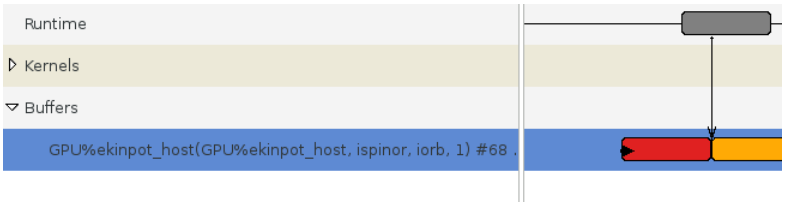
Figure 6.8(b) presents the execution of the following BIGDFT function, which creates and populates a memory buffer:

```
void FC_FUNC_(ocl_pin_write_buffer_async,OCL_PIN_WRITE_BUFFER_ASYNC)
             (bigdft_context *context, bigdft_command_queue *cq,...) {
  cl_int ciErrNum = CL_SUCCESS;
  *buff_ptr = clCreateBuffer((*context)->context, ...);
  oclErrorCheck(ciErrNum, "Failed to pin write buffer!");

  clEnqueueMapBuffer((*cq)->command_queue, ...);
  oclErrorCheck(ciErrNum, "Failed to map pinned write buffer (async)");
}
```



(a) a kernel execution



(b) a buffer allocation and writing

Figure 6.8: Visual Representation of OpenCL Execution Events.

6.3.2 *Cuda and Specfem 3D Cartesian*

In this last subsection, we study a free seismic wave propagation simulator, SPECFEM 3D [KME09], in its CARTESIAN flavor[7]. . SPECFEM 3D simulates seismic wave propagation at the local or regional scale based upon spectral-element method (SEM), with very good accuracy and convergence properties. Its current version (v2.1 of July 2013) supports graphics card GPU acceleration through NVIDIA CUDA. This comes in addition to the MPI support implemented to enable parallel computing. SPECFEM 3D code base is written in FORTRAN2003 (except the CUDA parts) and fully conforms to the standard.

STRUCTURAL REPRESENTATION

In CUDA, we distinguished only two set of entities for the structural representation, kernels and memory buffers. As the kernels symbols are directly parsed from the application binary (and not dynamically instanced as in OpenCL), they available since the very beginning of the execution (this version of SPECFEM 3D defines 36 kernels):

```
(gdb) start
Temporary breakpoint 1 at 0x4fc1b4: file program_specfem3D.f90:30
Starting program: examples/Mount_StHelens/bin/xspecfem3D
[Thread debugging using libthread_db enabled]
[New kernel get_maximum_kernel]
[New kernel get_maximum_vector_kernel]
[New kernel compute_add_sources_acoustic_kernel]
[New kernel add_sources_ac_SIM_TYPE_2_OR_3_kernel]
...
Temporary breakpoint 1, xspecfem3d () at program_specfem3D.f90:30
30          call init()
(gdb) info kernels
#1  get_maximum_kernel
#2  get_maximum_vector_kernel
#3  compute_add_sources_acoustic_kernel
#4  add_sources_ac_SIM_TYPE_2_OR_3_kernel
...
```

SPECFEM 3D GPU buffers are allocated at different parts of the application, so we will only focus on the first set. During the initialization of the code, CUDA buffers are allocated for the mesh structure representation:

```
(gdb) list ../cuda/prepare_mesh_constants_cuda.cu:210
209  // mesh
```

7 Specfem3d Cartesian — CIG http://www.geodynamics.org/cig/software/specfem3d

```
210 print_CUDA_error_if_any(cudaMalloc((void**) &mp->d_xix, ...), ...);
211 print_CUDA_error_if_any(cudaMalloc((void**) &mp->d_xiy, ...), ...);
```

When these functions are executed, McGDB detects the buffer creation and captures the memory pointer returned by the function, as well as the buffer name (`&mp->d_xix` and `&mp->d_xiy` here).

However, the second set of buffers is created though a helper function:

```
// copies integer array from CPU host to GPU device
void
copy_todevice_int(void **d_array_addr_ptr, int *h_array, int size) {
 print_CUDA_error_if_any(cudaMalloc((void**)d_array_addr_ptr,...),...);
 print_CUDA_error_if_any(cudaMemcpy((int*) *d_array_addr_ptr,h_array,
                                    ... ,cudaMemcpyHostToDevice),...);
}
```

In this case, developers can provide a configuration hint to McGDB, similar to what was done with OpenCL:

```
try_buffer_name(Events.CREATE_BUFFER, "copy_todevice_int",
                depth=2, param=1)
```

McGDB will then be able to provide a more intuitive name (`&mp->d_ibool` instead of a pointer address):

```
249 copy_todevice_int((void**) &mp->d_ibool,h_ibool,...);
(gdb) continue
[New buffer &mp->d_ibool]
```

In the following paragraphs, we present how to control the execution of this Cuda application.

EXECUTION CONTROL AND VISUALIZATION

McGDB's Cuda module offers the same control and visualization capabilities as the OpenCL module. Hence, developers can set catchpoints on kernel execution and buffer usage:

```
(gdb) kernel 1 catch execute
Catchpoint set on kernel #1 'get_maximum_kernel' execute events.

(gdb) buffer 1 catch memset
Catchpoint set on buffer #1  memset events.
```

Additionally, developers can also request execution catchpoints with combination of buffer and kernel events (here, the execution will stop only if kernel `#2` is executed with buffer `#34` as parameter):

```
(gdb) kernel 2 catch execute +with_buffer=34
Catchpoint set on kernel #1 'get_maximum_vector_kernel' execute events
with buffer #34 &d_max
(gdb) continue
...
Caught kernel #1 'get_maximum_vector_kernel' execute event.
In ../cuda/check_fields_cuda.cu:163:
959 get_maximum_vector_kernel<<<grid,thds>>>(mp->d_displ,size,d_max);
```

As we can see in Figure 6.9, CUDA also benefits from OpenCL visualization engine.

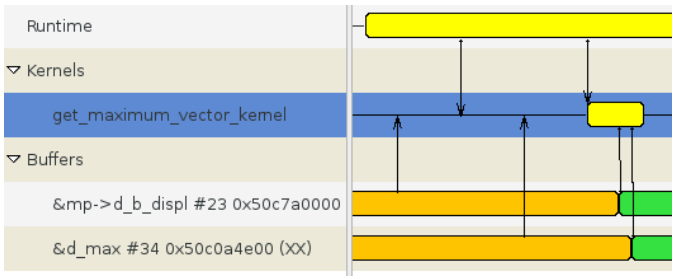


Figure 6.9: Visual Representation of Cuda Execution Events.

6.4 CONCLUSION

In this chapter, we highlighted how our contribution, a model-centric debugging approach, and its GDB-based implementation MCGDB, can be used for interactive debugging of applications based on MPSoC programming models. We presented four use-cases: first, an augmented-reality feature tracker (PKLT) developed with components and a H.264 video decoder module written with an dataflow framework. These applications target STHORM, our reference MPSoC system. Next, we studied two scientific computing applications accelerated with GPU processors. These two applications were written with two different programming environments, based on the same programming model, kernel-based accelerator programming.

As we could only have access to *established* applications (SPECFEM 3D and BIGDFT are for than 10 years old, PKLT and the H.264 video decoder were developed by different division teams at ST), we chose to present an alternative yet important use-case of model-centric debugging, where MCGDB is used by developers unfamiliar with the details of the applications. This use-case corresponds for instance to the maintenance part of the application life-cycle, or when a developer starts working on an under-documented project.

In the following months, further usage studies will be conducted at ST, as part of the product development of our prototype. In particular, the PEDF aspect of the tool is directly related to IDTEC team's mission, and hence will be soon integrated in their development and debugging environment. This will give application developers an easy access to MCGDB and allow us to gather their feedback.

With this illustration of our contribution in mind, we continue in the next chapter with a study of the related work. In this context, we compare how other research work compare, complement or could benefit from model-centric debugging.

Part III

Related Work and Conclusions

7

RELATED WORK

In the previous chapters, we presented and experimented our contribution, an improvement of MPSoC application interactive debugging. This improvement aims at incorporating in debuggers information from the programming models and environments used to develop MPSoC applications. application. But how does this proposal compare to existing work?

Flashbacks

In this chapter, we review the scientific literature related to this thesis contribution. We start in Section 7.1 with the publications related to low-level debugging of embedded systems. Then in Section 7.2 we come back to the context of general-purpose computing and study the literature about HPC application debugging. Finally, we discuss the work more related to programming-model and visualization-assisted debugging in Sections 7.3 and 7.4, respectively.

7.1 LOW-LEVEL EMBEDDED SYSTEM DEBUGGING

Verifying the correctness of embedded systems involves a larger set of abstraction level than on general-purpose computing. Indeed, in addition to the usual application level, manufacturers may have to design dedicated hardware platforms and IP blocks, as well as low-level software. In the following, we review the state-of-the-art techniques that developers can use to tackle problems at such low levels. We start at hardware level with communication analysis, then look at the debugging of platforms running on virtual simulators. We finish this section with an interactive debugger taking into account the OS kernel.

Hardware Level

In [GVN09], *Goossens et al.* presented a debugging environment targeting System-on-Chips(SoCs) with a Network-on-Chip (NoC). Similarly to this thesis work, their goal was to raise the abstraction of the debug process. Their starting point was at a lower level than ours: they started at bit, cycle and IP-bloc transaction levels [GVVSB07], then they explained how their tool presents VHSIC Hardware Description Language (VHDL) modules as a logical NoC topology; hardware FIFO structures as an ordered list of

messages; or retrieves dynamic information about the routers used in communications. They also discussed step-by-step execution over request/responses, transactions and handshakes, instead of the basic clock-cycle level.

This work is in the same spirit as ours, however it lies too close to the hardware to compete or cooperate with our proposal.

Platform-Simulator Level

Rogin et al. proposed in [RGDR08] another debugging environment, for the SystemC platform simulator. SystemC is a set of C++ classes for system-level design and modelling. It supports different levels of accuracy, from cycle-accurate operations to Transaction-Level Modelling (TLM). The authors of the article explained that debugging SystemC systems brings similar challenges to what we faced with programming models: without an additional debugger support, developers (designers in their context) must have an advanced knowledge of SystemC internals.

The debugging environment they introduced exploits SystemC API to provide additional levels of abstractions. They mentioned for instance SystemC signals, ports, events, *etc.* They also emphasized that their environment is based on GDB, to allow an efficient debugging of the functional part of the platform.

The idea behind this tool is in the same mindset as the one which drove this thesis work. Both projects could be integrated to offer a wide-angle view of MPSoC applications through their different levels of abstractions.

Operating-System Level

Georgiev et al. proposed in [GADP+10] to improve source-level debugging with OS-kernel awareness. Instead of focusing on classic applications as we do, they targeted the Linux kernel and its modules and user-space applications. In the context of embedded systems, they accessed the kernel memory space and processors states though a Joint Test Action Group/Test Access Port (JTAG/TAP) port, but it should be possible to obtain similar results with a system-level platform emulator. Their debugger offers the ability to list the different tasks (*e.g.*, user-land processes) and re-program the Memory Management Unit (MMU) to access transparently their virtual memory context. It also allows stopping the execution at key events, such as module management functions or during the interactions with user processes.

Kernel-level debugging is orthogonal to model-centric debugging. Indeed, an MPSoC application such as a video decoder may require a dedicated kernel module to perform its low level tasks (for instance, to stream the decoded sound and image data to output devices). If a program spreads infected memory states between the application and the kernel module, then developers may benefit from both the kernel and the model-centric debuggers. Going one step further, a model-centric debugger could consider the kernel module as an extension of the application and take in into account in an exhaustive

view of the application memory space and architecture.

To continue this literature review, we let aside the embedded system aspect of our work. We only focus on the multicore/parallel (Section 7.2) and the programming-model aspects (Section 7.3) of application debugging, which are more broadly studied.

7.2 HPC APPLICATION DEBUGGING

In this section, we review how the HPC community dealt with our debugging issues. In particular, we present the works tackling large-scale applications, which is similar to many-core MPSoC debugging concerns; then we look at debuggers targeting HPC programming environments as well as GPGPU computing.

Large-Scale Applications

When a computer runs thousands of tasks simultaneously, their execution generates tremendous quantities of information and events. A debugger must be able to filter out part of this data, first of all in order to perform efficiently, but also not to overwhelm developers with unusable information.

In order to support large-scale interactive debugging, *Balle et al.* [BBCLL04] defined a debugger architecture based on a tree-like network of aggregators, with fully functional source-level debuggers at each leaf. Figure 7.1 depicts this organization. The aggregator network provides an output-reducing mechanism, which scales down the quantity of execution output data brought to the user interface. They identified three different types of output (identical, identical except a small variation and widely different), that allow aggregators to merge the output streams. Hence, the network limits the quantity of information reaching the top-level debugger and ensures an acceptable response time. The tree network also carries the debugging commands input by the user and dispatch it to the relevant leaf debuggers.

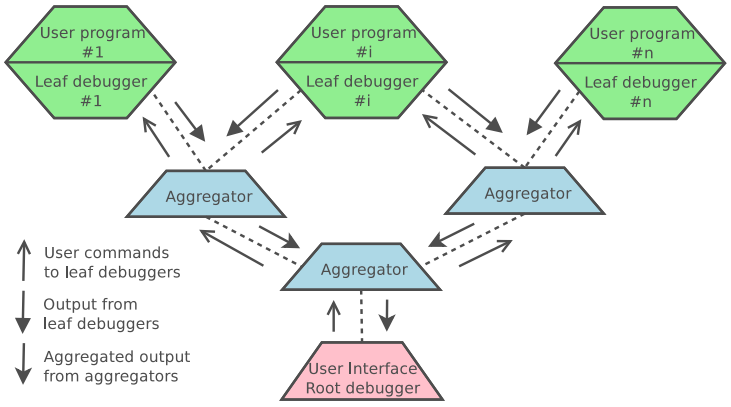


Figure 7.1: *Balle et al.* 's Tree-like Aggregators Network for Large-Scale Debugging.

In a similar way, parallel debugger DDT [Allo8] tries to simplify the debugging activity by merging 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 is debugged, allowing developers to better understand divergences within a group and thus possible bug sources.D

As we noted in the scope of model-centric debugging (Chapter 3, Section 3.2), our proposal does not target, nor addresses, the concerns of large-scale application debugging. Indeed, *increasing* the quantity of information available to developers is certainly not the most suitable path to follow in such situations. Adapting our proposal to large-scale debugging would involve an important refactoring of capture-mechanism as well as output-reduction optimization. Besides, the implementation we proposed assumes a share-memory environment, whereas large-scale computers may only offer distributed memory. Hence, the debugger would have to be adapted to take that configuration into account.

MPI Debugging

The literature provides only few examples which try to integrate the notion of message passing directly in interactive debuggers. In [CG99], *Cownie et al.* presented how they implemented MPI [MPI94] message queue interpretation in TotalView. They introduced the idea of representing *conceptual* information about the message-passing model in the debugger. They focused on the definition of a *standard* and concise API interface between the debugger and the MPI library implementation, in the same mindset as thread debugging, presented earlier. This standardization effort helped its adoption, and as per their website[1], more than eight MPI vendors implemented the interface. However, on the debugger side, only TotalView has the ability to exploit it.

This early work presented an interesting approach, whose design is similar to what exists today in GDB for thread debugging. However, they did not go really further than listing the content of the internal message queues. As MPI relies on a task-based programming model, with entities exchange message with one another, the principles of model-centric debugging could be directly applied to MPI, for instance as an extension of this work.

We discuss another MPI debugging tool in the last section of this chapter, Section 7.4, more oriented towards execution visualization.

Charm++ Debugging

In [JLK04], *Jyothi et al.* described a parallel debugger designed for the Charm++ data-driven parallel programming language. They insisted on the fact that the runtime system is good location to collect debugging and program analysis information. Hence, their debugging architecture is deeply tight into Charm++ runtime, and they implemented the debugging operations directly in Charm++. The debugger interface

1 MPI Debugging Interface, http://www.mcs.anl.gov/research/projects/mpi/mpi-debug/

communicates with the application through a network interface, and a standard, standalone GDB can be connected to the application on demand. They discussed some capabilities of their debugger, such as accessing visible objects as array and queue messages, setting breakpoints on CHARM++-specific entry-points, "*freezing*" and "*unfreezing*" selected processors, *etc.*).

An advantage of their design is that it can operate on distributed memory environments, which is important for CHARM++ debugging. However, the CHARM++-specific debugging commands are limited, and there is no integration between the source-level debugger (GDB) and the higher-level debugging environment.

STARSS DEBUGGING

TEMANEJO [BGNK11] is a debugger for the task-based STARSS programming-model family. These models, similarly to dataflow programming, put an important focus on the data dependencies of the different tasks which form the application. The runtime framework is in charge of executing the different tasks in parallel while preserving the correct ordering. The debugger is able to follow and reconstruct the graph of task executions. This graph is useful to developers as its structure is not known at compile time. Additionally, the debugger also allows developers to control task executions. It can change their priority, block them or measure their duration. TEMANEJO can also launch a debugger (*i.e.,* GDB) upon specific events, such as the beginning of a task execution.

Although the tool offers interesting representations, it currently lacks advanced interactive debugging commands. Indeed, TEMANEJO is totally decoupled from GDB, hence neither of the tools can benefit from the other. GDB remains at source level, and TEMANEJO cannot provide any language-related information.

GPGPU DEBUGGING

In [HZG09], *Hou et al.* discussed GPU-kernel debugging, on the GPGPU side of the application. The first step of their proposal consists in recording all the memory operations carried out by the execution kernels. Then, offline, they allow developers to visualize and analyze the flow of data, both inside a kernel and across several ones. They use compile-time code-instrumentation to gather this knowledge, by logging execution traces. In addition, their code-instrumentation framework also enables an automatic detection of out-of-bound array accesses, uninitialized data accesses and race conditions.

Although *post-mortem* instead of interactive, such debugging capabilities could help developers to understand the problems of the GPGPU-side of kernel-based applications, which is unsolved by our debugger. Some model-centric debugging events could also be recorded during the execution (namely those used for visualization), and interpreted in their data-flow visualizer. This would allow developers to follow a piece of information

transferred between the main memory and the GPGPU memory.

In the following section, we go one step closer and highlight the research work directly related to model-centric debugging.

7.3 PROGRAMMING-MODEL AWARE DEBUGGING

In this section, we confront our contribution with similar publications. Although object debugging may appear surprising in this context, the presented work clearly focuses on programming model behind object-oriented programming, in contrast with the language itself. The name *programming-model-centric debugging* stemmed out of this work on *object-centric debugging* [RBN12].

Object Debugging

The authors of [RBN12] presented an object-centric debugger, which aims at shifting the debugger focus from the execution stack towards the objects themselves. Our proposal shares part of their motivations, although they considered 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 required 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, in particular the C language which is frequently used in embedded systems. They also targeted programming language-level debugging, as they worked with programming language concepts, whereas we focused our efforts on the programming model. This means that they did not address the problem of bringing the debugger closer to the programming model abstractions used in the applications, and the developer is 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.

Component Debugging

General component-based application debugging presents an additional difficulty, which was out of scope of this work, as not directly 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, source-level debuggers are of little help, as most of them do not support multi-language debugging, or debugging is simply not possible in case of black-boxes. The authors of [WK05] proposed a solution to this problem: they squeezed debugging components in between application components. These debugging components are able to monitor component interface activities without any knowledge about the implemen-

tation and/or without exhibiting the multi-language debugging problem. The article described 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 developers to build and manually connect the interface monitors (although they mentioned 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 are inevitably different, and this discrepancy may hide some of the bugs.

Dataflow Debugging

StreamIt is a programming language for high-performance streaming applications [TKA02], which can be related to dataflow programming models and in particular synchronous dataflow [LM87]. The StreamIt Development Tool [Kuo04] provides a graphical environment to assist stream applications coding and debugging. Its debugger is tailored to StreamIt programming language. Similarly to our approach, the debugger takes into account StreamIt specificities and allows developers to interact with a graph representation of the application. It also displays information about the communication channels, such as the tokens they hold or stream statistics.

So their tool appears to tackle the challenges we describe in Chapter 2, Section 2.3.2. However, they did not focus on the problems of *dynamic* dataflow, as StreamIt is a synchronous dataflow language: all data must be received before execution and actors' sending and receiving rate is defined at compile time. So a substantial part of interactive debug challenges is avoided.

Wahl et al. described in 1988 a debugging methodology for dataflow programs [WS88], which shares similarities with our approach. To our knowledge, this is the only research study available related to dataflow debugging. However, at that time, they still had faith in *real* dataflow machines, with non *von Neumann* architectures. Their methodology pointed out that they wanted "*to allow the user to debug a program in a way that is close to his or her conceptual model of the program.*" They also mentioned that "*at the same time, the user must be supplied with a set of debugging commands that includes those with which he and she is familiar with in the context of uni-processor* von Neumann *machines*." We intimately share these convictions, which drove our work for this thesis.

The methodology they proposed is close to our idea of dataflow debugging, however they only skimmed over the interactive debugging aspects and they did not provide details about its actual usage. Our work on dataflow debugging extends and deepens this specific aspect. Furthermore, they explained that their methodology relies on a

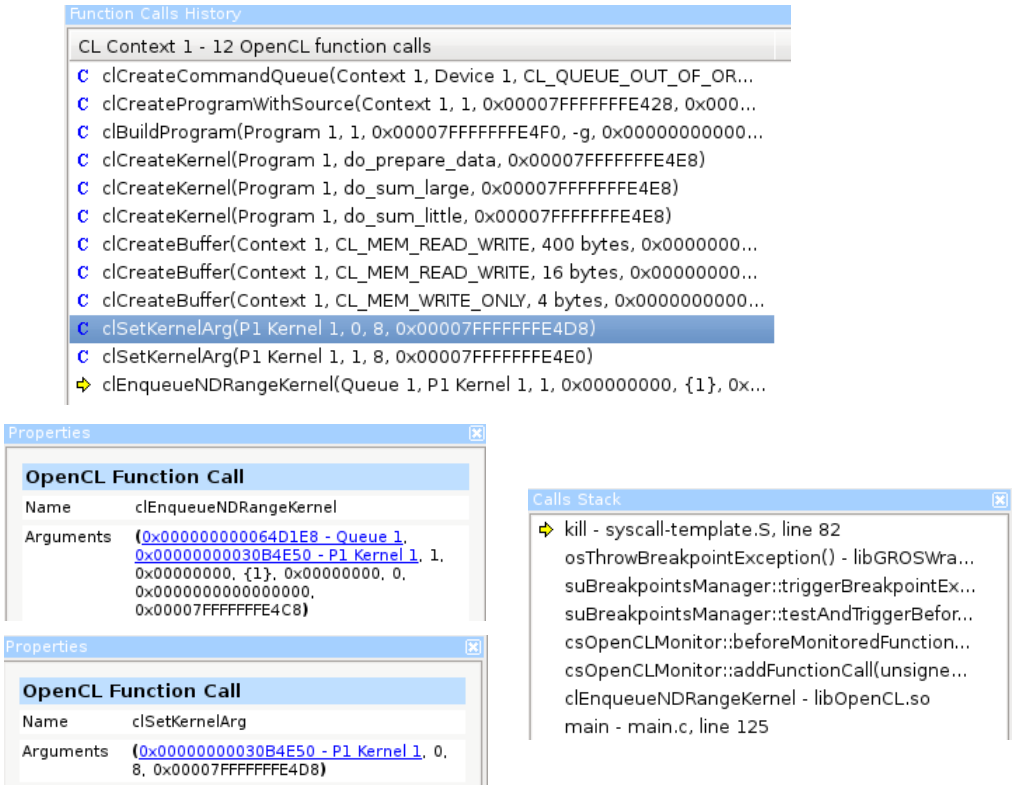


Figure 7.2: GDEBUGGER interface of OpenCL debugging.

dataflow machine simulators, which have to be modified to support debugging. This requirement strongly limits the scope of their work, as their debugging module would have to be implemented at hardware level. Our approach does not face this problem, as the debugger only interacts with the software dataflow environment.

OPENCL DEBUGGING

GREMEDY GDEBUGGER [Gra10] is a commercial debugger for OpenCL applications. Its graphical user-interface allows developers to control the execution based on OpenCL operations. Namely, they can set breakpoints on API functions, OpenCL errors or memory leaks. It is also possible to visualize OpenCL entities relationship and inspect the content of OpenCL buffers. Figure 7.2 presents different parts of the user interface, during an OpenCL debugging session.

These capabilities are interesting, although the *interactive* debugging side of the tool appears to be limited. Indeed, the tool can only analyze OpenCL elements, which limits the benefits for an overall application analysis and make compulsory the use of another debugger to understand the actual code behaviour. The source-level debugger they rely on is entirely hidden to the user, who cannot benefit from its capabilities.

As per our investigations, GDEBUGGER appears to rely on a preloaded shared library to capture OpenCL information. The library communicates with the user-interface through data sockets and a source-level debugger (GDB), similarly to the mechanism we presented in Chapter 4, Section 4.2.

In the following section, we review the publications which investigated how visualization could improve application debugging. We do not come back on GDEBUGGER, as its interface only presents information that could have been shown textually.

7.4 VISUALIZATION-ASSISTED DEBUGGING

To conclude this chapter on the related work, we study different approaches leveraging visualization techniques to improve execution understanding during interactive debugging.

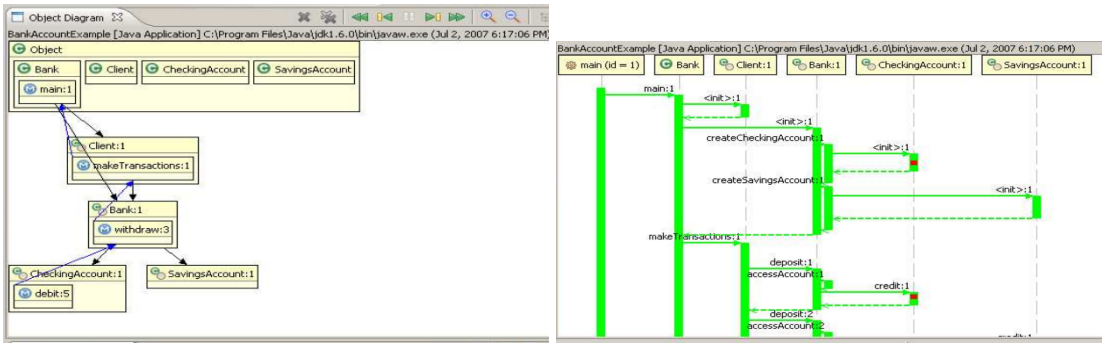
MPI DEBUGGING

In [SASH08], *Schaeli et al.* described an interesting solution which provides a visual representation of the messages exchanged by MPI processes and allows developers to explicitly control the ordering of the message-passing events. The authors explained that their main goal was to automatically or manually detect race conditions. Consequently, the approach did not focus on the other aspects of debugging, and their tool does not allow interactive debugging commands such as step-by-step execution or memory inspection. The implementation they proposed relies 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 contribution does not target the same class of problems as our work, the tool they propose could be extended and coupled with a model-centric debugging for MPI. As they noted, the graphical interface already displays the message-passing graph and provides a high-level view of the communication patterns. They also implemented high-level breakpoint mechanisms, although they only use them to control if a MPI process should run or not.

JAVA DEBUGGING

In [CJ07], *Czyz et al.* presented a *declarative and visual debugging environment* for JAVA applications. The *declarative* aspect is out of the scope of this section, although it influenced the design of their visualization tool. The authors emphasized that current debugger graphical interfaces serve mainly as front-ends for traditional text-based debuggers. They also insisted on the idea that a visual depiction of runtime states can help developers to notice that their mental representation of the application state



(a) graph structure (b) sequence diagram

Figure 7.3: JIVE Visualization of a JAVA execution.

differs from the actual state. Our motivations to start working with visualization tools stemmed from the same observations.

They also explained that a *declarative* debugger should allow developers to search the entire execution history (they named that *query-based* debugging), and hence it should provide runtime support for examining the current state as well as past state. This observation also drove our work, although not to the same extend: we wanted to depict state evolution, but not necessarily provide an exhaustive examination of past states.

To achieve their objectives, they extended an ECLIPSE environment, the JIVE, for JAVA debugging. The environment performs incremental state saving and restoration operations to record and replay/query the execution timeline. Thanks to this knowledge, they can draw object and sequence diagrams of the execution state, as presented in Figures 7.3(a) and 7.3(b), respectively. Our visualization capabilities are similar to what they presented, although our programming-model knowledge allows us to include more information in the diagrams. However, this knowledge has to be implemented for each programming model, whereas theirs works for any JAVA application.

Besides, a drawback of the implementation they proposed is that it operates at language level, and hence, in medium-to-large applications, their diagrams will be overloaded by the number of objects and method activation represented. They mentioned that they were exploring solution to this issue, such as suppressing internal details of some of the objects. Using a model-centric debugger at the lower levels could be another solution.

In the following chapter, we conclude this thesis manuscript and detail the future work.

8

CONCLUSIONS AND PERSPECTIVES

The Final Situation

Nowadays, consumer electronics devices are becoming more and more ubiquitous. With new generation smart-phones, set-top boxes and hand-held audio and video players, multimedia embedded systems are spreading at a fast pace, with a constantly growing demand for computational power. During the last decade, MPSoC systems have been introduced in the market to answer this demand. However, their exotic, multi- and many-core architectures harden application development.

A programming model defines a set of well-studied guidelines that developers can use to design the architecture and algorithms of their applications. At implementation time, supportive environments make concrete the programming models guidelines and provide developers with the relevant coding structures. They also emphasize separation-of-concern, with the low-level development (the implementation of the supportive environment) strictly decoupled from application development.

However, verification and validation of MPSoC systems remains a hard task. In some cases such as heavily constrained programming models, supportive environments may improve the task by allowing compile-time verification of mathematical properties. However, in the case of dynamic multimedia applications, these environments only worsen the situation.

We believe that interactive debugging can provide great help during software development and refinement. Indeed, these tools allow developers to explore and understand how the computer executes the application. With the ability to control CPU execution step-by-step, as well as displaying different memory locations, developers can confront their mental representation of what the code is *supposed* to do, against what it *actually* does.

However, current tools are too low level to offer an optimal control of programming-model-based applications, because they only operate at source and assembly level. The runtime libraries of supportive environment disrupt the linearity of the execution flows, and hide an important part of application state. Hence, the objectives of this thesis

were to raise the abstraction level of interactive debugging, so that it can be closer to the concepts used for application development.

8.1 CONTRIBUTION

In this thesis work, we proposed the principles of programming-model-centric debugging, a new level of application abstraction for interactive debuggers. This approach emphasizes the role of programming-model guidelines in application development, and uses it to offer a new, high-level and accurate vision of the applications. Three articles were published related to this work [PSMMM12, PLCSM13b, PLCSM13a].

We first studied MPSoC application development and debugging. We highlighted the necessity to rely on programming models and supportive environments to reuse well-studied and established coding structures and algorithms. We illustrated these notions with three programming models, used recurrently along the document: component-based programming, dataflow programming and kernel-based GPGPU programming. We also explored the debugging challenges faced by developers while building applications relying on such models. We highlighted that interactive debuggers provide interesting capabilities in comparison with alternative tools, although it appeared that they currently operate at too low level an optimal MPSoC application debugging.

In Chapter 3 we proposed our contribution to lighten these problems: enhancing interactive debugging with programming-model knowledge. We first stated the principles of model-centric debugging, related to providing a structural representation of the application, following its dynamic behaviors, and allowing interactions with the programming model abstract machine. We delimited the scope of application of model-centric debugging, that is, any application relying on an advanced enough abstract machine. We finally studied how our proposition applies to our three programming models. Component and dataflow programming models fit well in the proposal, thanks to the communicating-task paradigm they both extend. These models allow the debugger to distinguish tasks and draw a graph-based architecture diagram. On the other hand, kernel-based programming, highlights a distinct aspect of model-centric debugging, where visualization tools are used to depict the interactions between the application and the abstract machine, over the time.

Then, we conducted a practical study of model-centric debugging. This study aimed at validating its feasibility, with the extension of GDB, the free debugger of the GNU project, as well as industrial supportive environments and real-world application debugging sessions.

In Chapter 4, we detailed the implementation of the key building blocks of a model-centric debugger. Our design heavily relies on an extensible source-level debugger as back-end. Hence, our implementation is based on GDB and its PYTHON interface, though any tool offering source-level debugging services should fit.

After that, we described how we expanded model-centric principles to support the industrial MPSoC environments of our three programming models.

Finally, we demonstrated the capabilities of model-centric debugging in the context of four real-world applications. For each of our programming models, we highlighted important debugging features, and confronted them to their source-level counterpart.

This study underlined how application developers can benefit from our contribution. Indeed, instead of working with system-level entities such as threads and processes, they will be able to control components, dataflow actors or kernels; and query the internal state of programming model's abstract machine. Our tool also dynamically computes and draws a graph representation of the application architecture, instead of the flat set of threads and processes. This representation is a significant improvement, as it provides a more realistic and high-level view of the application state and activity.

Programming-model centric debugging also allows embedded platform vendors to provide not only a set of programming models and environments to program their architectures, but also a unified debugging suite, tailored to these different environments. Developers benefit from this unity, as they will be able to use a homogeneous set of debuggers across different situations. It can also encourage them to tackle other programming models of the platform, knowing that the tools they rely on are built on the same approach.

8.2 PERSPECTIVES

Interactive debugging of multicore embedded applications based on their programming model appears to be a promising direction to lighten the bug tracking hassle. In the following, we present different perspectives of future work:

Strengthen the implementation for production At this time, the tool we developed is a research prototype, but we expect it to rapidly enter into production at ST.

Conduct extensive impact studies This production step will allow us to conduct more concrete studies of the debugger's impact on development and validation time. Indeed, as industrial embedded application development is a long and sensible process, we have not yet been able to provide application developers with our tool during application development process.

Integrate within a graphical debugging environment Our work was mainly conducted in the context of textual environments, with only a weak link with graphical tools (we used them only for visualization purposes). New generation Integrated Development Environment (IDE) offer advanced language analysis capabilities, which could benefit from a model-centric debugging knowledge. Graphical environments would also attract more easily junior developers, who may be reluctant to textual tools.

Integrate within a visualization environment As we mentioned earlier, our visualization mechanisms serve only for visualization purposes. If they could be integrate into

the debugging environment, this would enhance and simplify the interactions between the developer and the abstract machine.

Extends towards other programming models In the Related Work chapter (Chapter 7), we sketched some possible extensions/cooperation of model-centric debugging. We mentioned in particular the idea of coupling our proposal with a hardware platform simulator, which would provide a wide-angle view of the application, at different levels of abstraction: model, language, assembly and hardware. In parallel, MPI debugging could also benefit from model-centric debugging. We presented an interesting visualization environment relying on MPI profiling API, this work could be extended to integrate the interactive debugging of model-centric debugging.

Enrich debugging information generated by compilers As a medium to long-term project, we plan to study how to extend the debugging information generated by compilers. With the help of their intermediate representation data structures, we expect to provide debuggers with high-level and more abstract knowledge about the execution flow.

Appendices

A

GDB MEMORY INSPECTION

Main Memory Representation Thanks to the DWARF [Fre10] debug information encrusted in application binaries by the compiler (if the relevant flag was set, like `-g` in GCC), GDB can display the memory content with the correct representation:

```
(gdb) print *breakpoint_chain
$8 = {
  ops = 0xc4b280 <bkpt_breakpoint_ops>,
  next = 0xf92340,
  type = bp_breakpoint,
  enable_state = bp_enabled,
  number = -1,
  frame_id = {
    stack_addr = 0,
    code_addr = 0,
    ...
  }
  ...
}
```

In this example, we display the content of the global variable `breakpoint_chain` (from GDB source code). We can see that it is a C structure, composed of pointers (`ops` and `next`). Fields `type` and `enable_state` are enums, `number` is a signed integer and `frame_id` is another structure:

```
(gdb) ptype *breakpoint_chain
type = struct breakpoint {
    const struct breakpoint_ops *ops;
    struct breakpoint *next;
    enum bptype type;
    enum enable_state enable_state;

    int number;
    struct bp_location *loc;
```

```
    ...
}
```

Without these DWARF information, GDB would only be able to display the raw content of the memory:

```
(gdb) x/10x 0xea4ad0 # *breakpoint_chain
0xea4ad0: 0x00c4b280 0x00000000 0x00f92340 0x00000000
0xea4ae0: 0x00000001 0x00000001 0x00000003 0xffffffff
0xea4af0: 0x00000000 0x00000000
```

where we can indeed recognize the value of some of the fields :

- `const struct breakpoint_ops *ops` = `0x00c4b280`
- `struct breakpoint *next` = `0x00f92340`
- `int number` = `0xffffffff` (-1)

Function Parameters, Register and Stack Inspection
Function parameter inspection is a key requirement for the implementation of your debugger. GDB interface does not explicitly allow manipulating it, hence we had to build our own support, based on the standardized C calling conventions. In `i386` processors, the parameters are pushed in the execution stack (reachable through the processor register `%SP`). After the function completion, the return value is accessible through register `%eax`. In `x86_64` processors, parameters are stored in registers `%rdi`, `%rsi`, *etc.*, and the return value is stored in `%rax`.

Processor registers can be displayed from GDB command-line, which directly relies on the LINUX's PTRACE API for the implementation (PTRACE is used for the implementation of the majority of the process control and inspection commands).

GDB also allows navigating in the process execution stack and displaying local variables and registers accordingly:

```
#0  create_breakpoint (arg="clCreateKernel", internal=1)
#1  bppy_init (self, args) at gdb/python/py-breakpoint.c:624
...
#102 PyRun_SimpleStringFlags () from /lib64/libpython2.7.so.1.0
...
#115 source_script (file=".gdbinit") at gdb/cli/cli-cmds.c:598
...
#120 main (argc=1, argv=0x7fffffffe0e8) at gdb/gdb.c:34
```

The stack trace (largely trimmed) shows that GDB was sourcing a script file (`.gdbinit`) with PYTHON code. When the execution was stopped, the PYTHON code was creating an internal (`internal=1`) on the (OpenCL) function named `clCreateKernel`.

B

EXTENDED ABSTRACT IN FRENCH

B.1 INTRODUCTION

Aujourd'hui, l'électronique grand public devient de plus en plus présente dans notre environnement. Avec les nouvelles générations de *smart phones*, tablettes, décodeurs de télévision internet ou autres baladeurs numériques portables, les systèmes embarqués dédiés au multimédia se déploient à un rythme effréné, avec un besoin en puissance de calcul toujours plus important.

Durant les dix dernières années, les Multi-Processor-Systems-on-a-Chip (MPSoCs) ont été introduits sur le marché pour répondre à cette demande. Ces systèmes-sur-une-puce (*systems-on-a-chip, SoC*) contiennent généralement un processeur généraliste multi-cœur, mais aussi des grappes (*clusters*) de processeurs dédiés à une application ou un domaine de calcul particulier. Ces processeurs peuvent avoir différents jeux d'instructions (ce type d'architecture est aussi appelé "hétérogène"), ce qui permet aux fabriquants d'optimiser les micro-architectures pour un calcul donné. Cette conception permet ainsi de construire des plates-formes avec une forte puissance de calcul, tout en maintenant une consommation électrique limitée.

Cependant, les attrayantes capacités des MPSoC sont attenuées par un important problèmes. En effet, bien que la programmation multi-cœur hétérogène soit en mesure de fournir une solution aux besoins actuels de puissance de calcul, elle augmente aussi la complexité du développement et de la phase de vérification et validation. Dans l'industrie des systèmes embarqués, ces aspects sont clés pour maintenir une rapide mise sur le marché (*time-to-market*). De ce fait, il sera crucial pour ces entreprises de réduire autant que possible leur impact.

En ce qui concerne l'aspect développement, les modèles de programmation fournissent les lignes directives, algorithmes de communications et modèles d'architectures pour résoudre les problèmes de développement récurrents. Dans le contexte de la programmation MPSoC, ces modèles de programmation vont permettre aux développeurs de pas "réinventer la roue de la programmation parallèle", et réduire le temps de conception de l'application. Pendant la phase d'implémentation, le code correspondant à ces structures bas-niveau et spécifique à chaque plate-forme pourra être séparé de l'application principale, par exemple sous la forme d'une bibliothèque logicielle. La réutilisation de code contribuera aussi à limiter le temps de mise sur le marché de

ces systèmes. Nous appellerons par la suite ces bibliothèques "environnements de programmation et d'exécution" (*supportive environments)*.

Cependant, le parallélisme des MPSoC rend aussi plus difficile la phase de vérification et validation des applications. La concurrence des flots d'exécutions amène de nouveaux types de bogues qui ne pouvaient pas exister dans les exécutions séquentielles, comme les interblocages ou les situations de compétitions. De plus, les environnements de programmation vont fréquemment changer la séquentialité des flots d'exécution pour se conformer aux directives du modèle de programmation. Dans ces conditions, il devient particulièrement difficile pour les développeurs de localiser les bogues logiciels et comprendre leur origine.

Objectifs de la thèse

Nous pensons que le débogage interactif peut fournir une aide substantielle au développement et à la mise-au-point des applications pour les systèmes multi-cœur embarqués. En effet, bien que les modèles de programmation haut-niveau simplifient le développement des applications, ils ne peuvent pas systématiquement garantir leur conformité. Si certains modèles permettent des vérifications statiques avancées au moment de la compilation, ces bénéfices sont possibles seulement au prix de fortes contraintes de programmation réduisant beaucoup l'expressivité du code. De l'autre coté, les modèles supportant un plus grand nombre d'algorithmes, et en particulier ceux avec des comportements dynamiques, ne peuvent en général pas offrir de telles garanties.

Pour ces modèles, le débogage interactif se présente comme une alternative intéressante. Il donne en effet la possibilité de surveiller et contrôler l'exécution des applications à plusieurs niveaux de granularités (code source, instructions machines, registres du processeur, mémoire centrale, ...), ce qui aurait été impossible avec les autres approches.

Cependant, dans son état actuel, le débogage interactif n'est pas encore adapté pour déboguer les applications basées sur des modèles de programmation haut-niveau. Le débogage au niveau des sources (*source-level interactive debugging*) a évolué pour supporter plusieurs flots d'exécutions et inspecter chacun des contextes mémoires, mais la sémantique des commandes de débogage est restée la même que pour les applications séquentielles, c'est-à-dire exclusivement basée sur le contrôle du processeur et la gestion des symboles (points d'arrêts, exécution pas-à-pas, affichage de la valeur d'une adresse mémoire, d'un registre, d'une variable, *etc.*).

Notre objectif dans cette thèse est de rehausser le niveau d'abstraction qu'utilisent les débogueurs interactifs pour représenter les applications. Ainsi, ils pourront permettre de travailler au même niveau d'abstractions pendant les phases de développement et de débogage, c'est-à-dire au niveau défini par le modèle de programmation.

Nous pensons que ces améliorations vont permettre aux développeurs de ne plus avoir à gérer les événements logiciels incontrôlables (avec les approches de débogage interactif actuelles) introduits par les environnements de programmation. D'autre

part, en plus de gêner les développeurs expérimentés dans leurs tâches, les outils mal adaptés découragent aussi les jeunes développeurs dans l'utilisation du débogage interactif.

Enfin, nous avons l'intention de décrire une approche générique pour faciliter et encourager le développement de débogueurs pour d'autres environnements de programmation haut-niveau. En effet, nous pensons qu'avoir un ensemble d'outils unifiés pour le débogage haut-niveau pourra aider les développeurs à passer d'un modèle de programmation à l'autre plus simplement et plus rapidement.

B.2 programmer et débogger les systèmes embarqués multi-cœurs

Dans ce chapitre, nous étudions les éléments nécessaires pour comprendre le contexte de cette thèse. Nous présentons dans un premier temps les architectures MPSoC et nous introduisons différents modèles de programmation capables d'exploiter leurs caractéristiques. En effet, développer des applications tirant le meilleur profit des processeurs multi-cœur MPSoC et de leur conception unique nécessite l'utilisation d'algorithmes avancés, combinés à des bibliothèques logicielles performantes. Pour satisfaire ces besoins, nous présentons trois modèles et environnements de programmation : la programmation par composants, par flots de données et par noyaux de calcul.

Du point de vue de la vérification et validation, les modèles de programmation se divisent en deux catégories : soit ils imposent de fortes contraintes aux développeurs, avec de la programmation statique, et dans ce cas ils peuvent fournir d'intéressantes propriétés pour la validation ; soit ils sont moins restrictifs et ils ne fournissent aucune aide à la vérification. Comme cette thèse s'est déroulée dans une des divisions d'STMicroelectronics (ST) s'occupant de multimédia, nous ne nous sommes intéressés qu'à cette seconde catégorie. En effet, la programmation multimédia nécessite souvent l'utilisation de techniques de programmation dynamique, par exemple pour le décodage vidéo ou le traitement des images, où l'exécution va dépendre du contenu du média.

Nous présentons ensuite différents outils et techniques de débogage : l'analyse papier/crayon, les analyses formelles et statiques, et l'analyse de traces, et nous les comparons avec le débogage interactif. Avec cette étude, nous mettons en évidence que le débogage interactif fournit une approche convaincante, qui permet aux développeurs d'interagir avec l'application durant son exécution et d'inspecter les différents chemins d'exécution et états internes du programme.

Cependant, nous montrons que le débogage au niveau des sources de l'application n'est pas suffisant pour s'attaquer efficacement à des applications basées sur un modèle de programmation. En effet, les modèles, et plus concrètement leurs environnements de programmation, introduisent de haut niveaux d'abstractions dans les structures de

l'application, et ces niveaux d'abstractions ne sont pas pris en compte par les outils actuels.

B.3 CONTRIBUTION : MISE AU POINT CENTRÉE SUR LE MODÈLE DE PROGRAMMATION

Dans ce chapitre, nous présentons notre contribution pour améliorer l'efficacité des développeurs pendant le débogage interactif des applications multicœurs basées sur les modèles de programmation pour les MPSoC. Nous proposons tout d'abord les principes génériques du débogage centré sur le modèle (*model-centric debugging*), qui vont dans trois directions :

1. fournir une représentation structurelle de l'architecture de l'application,
2. observer les comportements dynamiques de l'exécution,
3. permettre aux développeurs d'interagir avec les machines abstraites et physiques exécutant l'application.

À travers ces trois axes, nous définissons les aspects nécessaires à la conception d'un débogueur haut niveau et bien adapté au débogage des applications basées sur un modèle de programmation.

Nous démarquons aussi le champs d'action du débogage centré sur le modèle : les applications parallèles basées sur un modèle de programmation MPSoC. Nous notons cependant que ce n'est pas une limitation forte, car le débogage centré sur le modèle peut s'appliquer à n'importe quelle machine abstraite, à partir du moment où son interface de programmation est suffisamment stable.

Vis-à-vis des limites du champs d'action, nous expliquons que le débogage centré sur le modèle n'est pas forcément adapté aux applications déployées à large échelle, car les développeurs se retrouveraient vite surchargés d'informations qu'ils ne pourraient pas gérer. Nous notons également que, pour des raisons similaires, le débogage centré sur le modèle ne pourra pas fournir beaucoup d'aide pour résoudre les problèmes des applications basées sur le parallélisme de données (Single Instruction Multiple Data (SIMD)).

Ensuite, nous étudions comment le débogage centré sur le modèle s'applique à nos trois modèles de programmation MPSoC. Nous mettons en évidence que les composants et la programmation par flots de données bénéficient directement de la représentation des structures, grâce à la décomposition en tâches. Pour la même raison, le débogage centré sur le modèle de programmation s'intègre aisément dans leurs opérations de communications, et nous avons pu décrire comment les messages et jetons de programmation par flots de données permettent d'améliorer le contrôle de ces applications. Enfin, nous montrons que les principes du débogage centré sur le modèle s'appliquent aussi à des modèles plus différents, comme la programmation d'accélérateurs par noyaux de calcul. Nous notons que l'intérêt des représentations structurelles n'est pas aussi important que pour les modèles basés sur les tâches ; par contre la nature des interactions entre l'application et la machine abstraite permet de

concevoir d'autres fonctionnalités de débogage haut niveau, comme par exemple des diagrammes de séquence des différentes interactions.

B.4 BLOCS DE CONSTRUCTION D'UN DÉBOGUEUR CENTRÉE SUR LE MODÈLE DE PROGRAMMATION

Dans ce chapitre, nous détaillons l'implémentation des principaux blocs de constructions d'un débogueur basé sur les principes énoncés au chapitre précédent. La figure B.1 présente les différentes couches de l'architecture de notre débogueur. Les deux couches inférieures correspondent aux acteurs traditionnels du débogage : la plate-forme d'exécution, qui exécute l'application, et un débogueur niveau source.

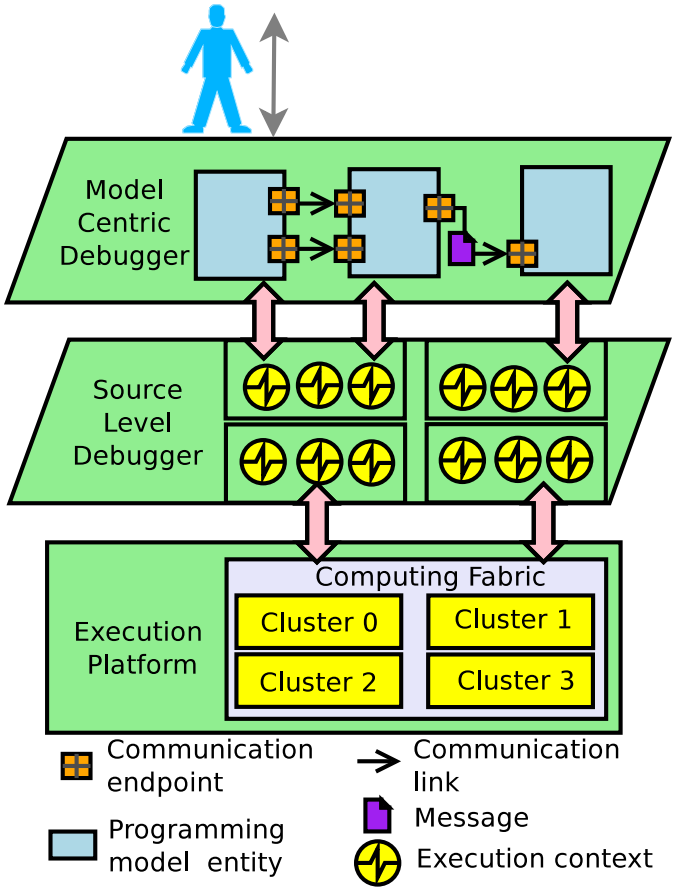


Figure B.1: Architecture d'un débogueur centré sur le modèle pour une plate-forme MPSoC

Dans le chapitre 2, nous avons présenté les plates-formes d'exécution MPSoC (en partie basse de la figure), nous continuons donc ici avec les niveaux supérieurs. Nous commençons avec la partie centrale et présentons les fonctionnalités de débogage niveau source nécessaires pour l'implémentation d'un débogueur niveau modèle. Ensuite, nous continuons avec la partie supérieure, qui correspond au débogueur centré sur le

modèle à proprement parler. Nous présentons l'étude de l'implémentation en suivant trois axes directeurs :

1. le débogueur doit être capable de capturer les informations nécessaires au suivi de l'état interne de la machine abstraite. Dans la figure, cela correspond aux flèches roses connectant la partie haute, le débogueur niveau source et la plate-forme d'exécution.

2. le débogueur doit définir des structures internes capables de refléter l'organisation de la machine abstraite, et les mettre à jour au fur et à mesure. Ces structures sont représentées sur la figure au travers des trois entités inter-connectées.

3. le débogueur doit fournir une interface haut niveau permettant à ses utilisateurs d'interagir efficacement avec la machine abstraite. La plupart des commandes fournies au travers de cette interface devront être paramétrables, en fonction de l'état courant de la machine virtuelle. Cette interface est représentée par l'utilisateur en au sommet du diagramme.

B.5 MCGDB, UN DÉBOGUEUR CENTRÉ SUR LE MODÈLE POUR L'ENVIRONNEMENT DE PROGRAMMATION D'UN MPSOC INDUSTRIEL

Dans ce chapitre, nous continuons l'étude pratique du débogage centré sur le modèle avec Model-Centric GDB (MCGDB), notre prototype de débogueur niveau modèle. Pour chacun de nos trois modèles de programmation, nous revenons sur la description des environnements de programmation de la plate-forme ST Heterogeneous Low Power Many-core (STHORM) (Chapitre 2, Section 2.2.2) et présentons les fonctionnalités de débogage que nous avons proposées.

Nous présentons dans un premier temps le *framework* de programmation par composants de STHORM, en mettant l'accent sur l'aspect dynamique du déploiement et de la gestion des composants, et sur le suivi des communications par message.

Ensuite, nous étudions le *framework* de programmation par flots de données, et nous montrons comment la représentation structurelle de l'application est présentée sous la forme d'un graphe. Nous expliquons aussi comment l'environnement de programmation permet à l'application de gérer l'ordonnancement de ses acteurs, et comment MCGDB prend cela en compte.

Enfin, nous détaillons le travail fait sur Open Computing Language (OpenCL). Dans cet environnement, nous insistons sur la représentation des comportements dynamiques de l'application à l'aide de diagrammes de séquence. Nous expliquons aussi comment nous avons étendu le module OpenCL d'MCGDB pour supporter NVIDIA CUDA, un environnement de programmation similaire à OpenCL, mais aussi compétiteur commercial.

Nous concluons ce chapitre avec un survol des points communs et divergences entre ces trois implémentations, ainsi qu'une estimation du temps nécessaire pour porter McGDB vers d'autres environnements de programmation.

B.6 ETUDES DE CAS

Dans ce chapitre, nous expliquons comment notre contribution, une approche de débogage centré sur le modèle de programmation, et son implémentation McGDB, peuvent être utilisées pour le débogage interactif des applications basées sur les modèles de programmation MPSoC. Nous présentons quatre études de cas : un code de suivi d'objets dans les vidéos (*feature tracking*) développé avec des composants, et un module de décodage de vidéos au standard H.264, écrit avec la bibliothèque de programmation par flots de données. Ces deux applications s'exécutent sur Sthorm, notre système MPSoC de référence. Ensuite, nous étudions deux applications de calcul scientifique, accélérées par des processeurs Graphical Processing Unit (GPU). Ces deux applications ont été écrites dans deux environnements de programmation distincts, mais basés sur le même modèle, la programmation par noyaux de calcul.

Comme nous n'avons eu accès qu'à des applications déjà bien établies (les codes scientifiques ont plus de 10 ans, et les applications pour l'embarqué ont été développées dans d'autres divisions d'ST), nous avons choisi de présenter une utilisation alternative du débogage centré sur le modèle, où McGDB est utilisé par des développeurs non familiers avec les détails des applications. Ce cas d'utilisation correspond par exemple à la phase de maintenance de l'application, ou quand un développeur commence à travailler dans un projet sous documenté.

B.7 TRAVAUX CONNEXES

Dans ce chapitre, nous présentons une étude de l'état de l'art en relation avec notre contribution. Nous commençons avec les publications liées au débogage bas-niveau des systèmes embarqués. Ensuite, nous revenons vers les systèmes plus généralistes et étudions la littérature liée au débogage des applications High-Performance Computing (HPC). Enfin, nous détaillons les travaux traitant du débogage lié au modèle de programmation ainsi que les outils de débogage utilisant des techniques de visualisation.

B.8 CONCLUSIONS ET PERSPECTIVES

Contributions

Durant cette thèse, nous avons défini les principes du débogage centré sur le modèle de programmation. Ces principes permettent aux débogueurs interactifs d'offrir un nouveau niveau de représentation des applications. Notre approche met l'accent sur le rôle du modèle de programmation pendant le développement des applications, et

reprend les lignes directives du modèle pour fournir une vision plus haut niveau et plus précise des applications. Trois articles ont été publiés en relation avec ce travail [PSMMM12, PLCSM13b, PLCSM13a].

Nous avons tout d'abord étudié le développement et le débogage des applications MPSoC. Nous avons mis en évidence la nécessité de s'appuyer sur des modèles et environnements de programmation pour réutiliser les codes et algorithmes déjà mise en œuvre pour résoudre les problèmes classiques. Nous avons illustré ces notions à travers trois modèles de programmation utilisés pour la programmation MPSoC : les composants, les flots de données et les noyaux de calcul General-Purpose Graphical Processing Unit (GPGPU). Nous avons aussi exploré les difficultés rencontrées par les développeurs lorsqu'ils construisent des applications basées sur ces modèles. Nous avons mis en évidence que le débogage interactif fournit des fonctionnalités intéressantes par rapport aux autres outils, cependant il apparaît clair que les outils travaillent à un niveau inférieur à ce qui serait optimal pour le débogage d'applications MPSoC.

Dans le Chapitre 3, nous avons proposé une approche pour alléger ces problèmes, en ajouter la connaissance du modèle de programmation dans le débogage interactif. Nous avons tout d'abord décrit les principes du débogage centré sur le modèle, qui proposent 1/ de fournir une représentation structurelle de l'architecture de l'application, 2/ de suivre les comportements dynamique de l'exécution et 3/ de permettre aux utilisateurs d'interagir avec la machine abstraite définie par le modèle de programmation. Nous avons délimité la portée du débogage centré sur le modèle, qui pourra être mise en œuvre dans l'ensemble des environnements de programmation utilisant une machine abstraite suffisamment haut niveau. Nous avons enfin étudié comment notre proposition pouvait s'appliquer à nos trois modèles de programmation MPSoC. Les composants et la programmation par flots de donnée s'y intègrent très bien, car ces modèles sont tous les deux basés sur la programmation par tâches communiquantes. Le débogueur sera donc capable de distinguer les différentes tâches de l'application et d'afficher des diagrammes représentant l'architecture sous forme de graphe. Notre troisième modèle de programmation, la programmation par noyaux de calcul, met en avant un autre aspect du débogage centré sur le modèle, où des outils de visualisation sont utilisés pour représenter les interactions au cours du temps entre l'application et la machine abstraites.

Nous avons ensuite mené une étude pratique sur le débogage centré sur le modèle. Cette étude visait à valider sa faisabilité, avec l'extension de GDB, le débogueur libre du projet GNU et le support des plusieurs environnements de programmation, et la mise en œuvre de sessions de débogage d'applications du monde réel.

Dans le Chapitre 4, nous avons détaillé l'implémentation des principaux blocs d'un débogueur centré sur le modèle. Notre implémentation repose principalement sur l'extension d'un débogueur travaillant au niveau du code source, GDB et son interface Python. Cependant n'importe quel outil capable de faire du débogage interactif au niveau des sources pourrait être utilisé.

Ensuite, nous avons décrit comment nous avons étendu les principes génériques du débogage centré sur le modèle pour supporter les environnements de programmation MPSoC de nos trois modèles.

Enfin, nous avons démontré les capacités du débogage centré sur le modèle de programmation dans le cadre de quatre applications du monde réel. Pour chacun des nos modèles de programmation, nous avons mis en évidence les principales fonctionnalités de débogage, et nous les avons confrontés à ce qu'il était possible avec un simple débogueur niveau source.

Cette étude montre comment notre contribution peut aider les développeurs dans leur travail. En effet, au lieu de travailler avec des entités de niveau système pour les threads et processus, ils vont maintenant pouvoir contrôler des composants, des acteurs de flot de données ou des noyaux de calcul, et interroger l'état interne de la machine abstraite du modèle de programmation. De plus, notre outil construit et maintient dynamiquement un graphe représentant l'architecture de l'application, au lieu d'un simple ensemble non structuré de threads et processus. Cette représentation constitue une amélioration importante car elle fournit une représentation plus réaliste et plus haut niveau de l'état et l'activité de l'application.

Le débogage centré sur le modèle de programmation permet aussi aux constructeurs de plates-formes embarquées de fournir, en plus des modèles et environnements de programmation, un ensemble unifié d'outils de débogage, adapté à ses différents environnements. Les développeurs vont bénéficier de cette unité, car ils vont pouvoir utiliser un ensemble d'outils homogènes dans différentes situations. Cela peut aussi les encourager à se confronter aux autres modèles de programmation de la plate-forme, en sachant que les outils qu'ils vont utiliser seront basés sur la même approche.

Perspectives

Le débogage interactif des systèmes embarqués multi-cœur centré sur le modèle de programmation semble une perspective de travail intéressante pour réduite la difficulté de la recherche de bogues. Nous envisageons donc de poursuivre ce travail dans les directions suivantes :

Renforcer l'implémentation pour une mise en production L'outil que nous avons développé est aujourd'hui seulement au stade de prototype de recherche, mais nous pensons qu'il pourra bientôt être mis en production au sein d'ST.

Conduire des études d'impacts Cette mise en production devrait nous permettre de conduire une étude sur l'impact de notre outil sur le temps de développement et de validation des applications. En effet, comme le temps de développement des applications pour l'industrie de l'embarqué est compté, nous n'avons pas eu la possibilité de fournir notre outil aux développeurs pendant les phases de développement.

Intégration dans un environnement graphique de débogage Notre travail a été principalement implémenté au sein d'environnements en ligne de commande, avec un

faible lien avec les outils graphiques (nous les avons seulement utilisé pour visualiser l'exécution). Les nouvelles générations d'environnements de développement intégrés (Integrated Development Environment (IDE)) offrent des capacités avancées d'analyse des langages, qui pourraient bénéficier des connaissances du débogage centré sur le modèle de programmation. Les environnements graphiques attirent aussi plus facilement les jeunes développeurs, qui pourraient être effrayer par les outils en ligne de commande.

Intégration dans un environnement de visualisation Comme mentionné dans le point précédent, nos mécanismes de visualisation sont utilisés uniquement pour la visualisation. Si elles pouvaient être intégrées dans un environnement de débogage graphique, cela permettrait d'améliorer et de simplifier les interactions entre les développeurs et l'application.

Etendre vers d'autres modèles de programmation Dans le chapitre des travaux connexes (Chapitre 7), nous avons proposé des possibilités d'extensions/coopération du débogage centré sur le modèle. Nous avons mentionné en particulier l'idée de coupler notre proposition avec des simulateurs de plates-formes, ce qui permettrait d'offrir une vision multi-niveau de l'application : modèle, langage, assembleur et matériel. Dans un autre domaine, les applications basées sur Message-Passing Interface (MPI) pourraient aussi bénéficier de débogage centré sur le modèle. En particulier, nous avons présenté un environnement de visualisation utilisant l'Application Programming Interface (API) de profiling d'MPI. Ce travail pourrait être étendu pour intégrer des commandes pour le débogage interactif de niveau modèle.

Enrichir les informations de débogage générées par le compilateur À moyen ou long terme, nous voulons étudier comment améliorer les informations de débogage générées par le compilateur. Avec les informations sur les structures de contrôles calculées dans la représentation intermédiaire de la compilation, nous pensons qu'il serait possible de fournir aux débogueurs des informations plus haut niveau et plus abstraites sur le flot d'exécution.

BIBLIOGRAPHY

[Allo8] Allinea Software. Parallel Debugging is Easy, 2008. {**116**}

[BBCLL04] Susanne M. Balle, Bevin R. Brett, Chih-Ping Chen, and David LaFrance-Linden. Extending a Traditional Debugger to Debug Massively Parallel Applications. *J. Parallel Distrib. Comput.*, 64, May 2004. {**115**}

[BDT12] S.S. Bhattacharyya, E.F. Deprettere, and B.D. Theelen. Dynamic dataflow graphs. In S. S. Bhattacharyya, E. F. Deprettere, R. Leupers, and J. Takala, editors, *Handbook of Signal Processing Systems (2nd edition)*. Springer, 2012. {**20, 23**}

[BFFM12] Lucas Benini, Eric Flamand, Didier Fuin, and Diego Melpignano. P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, pages 983–987, 2012. {**13**}

[BGNK11] Steffen Brinkmann, José Gracia, Christoph Niethammer, and Rainer Keller. TEMANEJO - a debugger for task based parallel programming models. In *International Conference on Parallel Computing*, 2011. {**117**}

[Bou00] Jean-Yves Bouguet. Pyramidal Implementation of the Lucas Kanade Feature Tracker. Description of the Algorithm. `http://robots.stanford.edu/cs223b04/algo_tracking.pdf`, 2000. {**86**}

[CG99] James Cownie and William 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. {**116**}

[CJ07] Jeffrey K. Czyz and Bharat Jayaraman. Declarative and visual debugging in eclipse. In *Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange*, eclipse '07, pages 31–35, New York, NY, USA, 2007. ACM. {**121**}

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009. {**25**}

[Crn04] Ivica Crnkovic. Component-Based Approach for Embedded Systems. In *Ninth International WCOP Workshop*, June 2004. {**19**}

[Fre10] Free Standards Group. The DWARF debugging standard. `dwarfstd.org/doc/Dwarf3.pdf`, 2010. {**54, 129**}

[GADP+10] Kiril Georgiev, Mathieu Auvray, Serge De-Paoli, Miguel Santana, and Chris Smith. Debugging Embedded Linux Kernel Through JTAG Port. In

Proceedings of the S4D (System, Software, Soc and Silicon Debug), 2010. {**50, 114**}

[Gnu13] Gnu Project. GDB, The GNU Debugger. http://www.gnu.org/software/gdb/, 1986-2013. {**50**}

[GOD+09] Luigi Genovese, Matthieu Ospici, Thierry Deutsch, Jean-François Méhaut, Alexey Neelov, and Stefan Goedecker. Density functional theory calculation on many-cores hybrid central processing unit-graphic processing unit architectures. *The Journal of chemical physics*, 131:034103, 2009. {**99**}

[Gra10] Graphic Remedy. gDEBugger, 2010. {**55, 120**}

[GVN09] Kees Goossens, Bart Vermeulen, and Ashkan B. Nejad. A high-level debug environment for communication-centric debug. In *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, pages 202–207, 2009. {**113**}

[GVO+11] Luigi Genovese, Brice Videau, Matthieu Ospici, Thierry Deutsch, Stefan Godecker, and Jean-François Mehaut. Daubechies Wavelets for High Performance Electronice Structure Calculations: the BigDFT Project. *Comptes Rendus de l'Académie des Sciences*, 339:149–164, 2011. {**99, 100**}

[GVVSB07] Kees Goossens, Bart Vermeulen, Remco Van Steeden, and Martijn Bennebroek. Transaction-based communication-centric debug. In *Networks-on-Chip, 2007. NOCS 2007. First International Symposium on*, pages 95–106. IEEE, 2007. {**113**}

[HZG09] Qiming Hou, Kun Zhou, and Baining Guo. Debugging gpu stream programs through automatic dataflow recording and visualization. *ACM Trans. Graph.*, 28(5):153:1–153:11, December 2009. {**117**}

[JHRM04] Wesley M. Johnston, J. R. Paul Hanna, Richard, and J. Millar. Advances in dataflow programming languages. *ACM Comput. Surv*, 36, 2004. {**19, 23**}

[JLK04] Rashmi Jyothi, Orion Sky Lawlor, and Laxmikant Kale. Debugging support for Charm++. In *PADTAD Workshop for IPDPS 2004*, page 294. IEEE Press, 2004. {**116**}

[JLL05] He Jifeng, Xiaoshan Li, and Zhiming Liu. Component-Based Software Engineering – The Need to Link Methods and their Theories. In *Proc. of ICTAC05, Lecture Notes in Computer Science 3722*. Springer, 2005. {**18**}

[Khr08] Khronos OpenCL Working Group. The opencl specification, version 1.0. http://khronos.org/registry/cl/specs/opencl-1.0.pdf, December 2008. {**16, 21, 23, 78**}

[KME09] Dimitri Komatitsch, David Michéa, and Gordon Erlebacher. Porting a high-order finite-element earthquake modeling application to NVIDIA graphics cards using CUDA. *Journal of Parallel and Distributed Computing*, 69(5):451–460, 2009. {**106**}

[Kuo04] Kimberly Kuo. The streamit development tool: A programming environment for streamit. M.eng. thesis, Massachusetts Institute of Technology, Cambridge, MA, Jun 2004. {**119**}

[KWK10] Johan Kraft, Anders Wall, and Holger Kienle. Trace recording for embedded systems: Lessons learned from five industrial projects. In *Runtime Verification, 2010 International Conference on*. Springer-Verlag (Lecture Notes in Computer Science), November 2010. {**25**}

[LCBT+12] Patricia López Cueva, Aurélie Bertaux, Alexandre Termier, Jean-François Méhaut, and Miguel Santana. Debugging embedded multimedia application traces through periodic pattern mining. In *Proceedings of the tenth ACM international conference on Embedded software*, EMSOFT '12, pages 13–22, New York, NY, USA, 2012. ACM. {**25**}

[Lea00] Doug 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. {**23**}

[LM87] Edward A. Lee and David G. Messerschmitt. Synchronous Data Flow. In *Proceedings of the IEEE*, volume 75, September 1987. {**20, 119**}

[LPSZ08] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. *ACM SIGARCH Computer Architecture News*, 36(1):329–339, 2008. {**11**}

[MBF+12] Diego Melpignano, Luca Benini, Eric Flamand, Bruno Jego, Thierry Lepley, Germain Haugou, Fabien Clermidy, and Denis Dutoit. Platform 2012, a many-core computing accelerator for embedded socs: performance evaluation of visual analytics applications. In *DAC*, pages 1137–1142, 2012. {**13**}

[MCS+06] Guillaume Marceau, Gregory H. Cooper, Jonathan P. Spiro, Shriram Krishnamurthi, and Steven P. Reiss. The design and implementation of a dataflow language for scriptable debugging. *Automated Software Engineering Journal*, 2006. {**44**}

[Mol03] Ingo Molnar. The Native POSIX Thread Library for Linux. Technical report, Tech. Rep., RedHat, Inc, 2003. {**52, 56**}

[MPI94] MPI Forum. MPI: A message-passing interface standard, 1994. {**16, 116**}

[NVi09] NVidia. Whitepaper on NVIDIA's Next Generation CUDA Compute Architecture: TM Fermi. http://www.nvidia.fr/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf, 2009. {**14**}

[Ölv11] Peter Csaba Ölveczky. Formal model engineering for embedded systems using real-time maude. In *AMMSE*, 2011. {**25**}

[Pau13] Paulin, Pierre G. OpenCL Programming Tools for the STHORM Multi-Processor Platform: Application to Computer Vision, 2013. 13th Inter-

national Forum on Embedded MPSoC and Multicore, July 15-19, 2013, Otsu, Japan. {**24**}

[PLCSM13a] Kevin Pouget, Patricia López Cueva, Miguel Santana, and Jean-François Méhaut. Interactive Debugging of Dynamic Dataflow Embedded Applications. In *Proceedings of the 18th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS)*, Boston, Massachusetts, USA, may 2013. Held in conjunction of IPDPS. {**40, 70, 124, 138**}

[PLCSM13b] Kevin Pouget, Patricia López Cueva, Miguel Santana, and Jean-François Méhaut. A novel approach for interactive debugging of dynamic dataflow embedded applications. In *Proceedings of the 28th Symposium On Applied Computing (SAC)*, pages 1547–1549, Coimbra, Portugal, apr 2013. {**40, 70, 124, 138**}

[PPCJ10] Kevin Pouget, Marc Pérache, Patrick Carribault, and Hervé Jourdren. User level DB: a debugging API for user-level thread libraries. In *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–7, 2010. {**52, 56, 66**}

[PSMMM12] Kevin Pouget, Miguel Santana, Vania Marangozova-Martin, and Jean-François Mehaut. Debugging Component-Based Embedded Applications. In *Joint Workshop Map2MPSoC (Mapping of Applications to MPSoCs) and SCOPES (Software and Compilers for Embedded Systems)*, St Goar, Germany, may 2012. Published in the ACM library. {**37, 65, 124, 138**}

[RBN12] Jorge Ressia, Alexandre Bergel, and Oscar Nierstrasz. Object-centric debugging. In *In Proceeding of the 34rd international conference on Software engineering*, 2012. {**118**}

[RGDR08] Frank Rogin, Christian Genz, Rolf Drechsler, and Steffen Rülke. An integrated systemc debugging environment. In Eugenio Villar, editor, *Embedded Systems Specification and Design Languages*, volume 10 of *Lecture Notes in Electrical Engineering*, pages 59–71. Springer Netherlands, 2008. {**55, 114**}

[SASH08] Basile Schaeli, Ali Al-Shabibi, and Roger 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. {**121**}

[SPA+08] Nathan Sidwell, Vladimir Prus, Pedro Alves, Sandra Loosemore, and Jim Blandy. Non-stop multi-threaded debugging in gdb. In *GCC Developers' Summit*, page 117, 2008. {**35, 51**}

[ST98] David Skillicorn and Domenico Talia. Models and languages for parallel computation. *ACM Comput. Surv.*, 30(2), June 1998. {**15**}

[TKA02] William Thies, Michal Karczmarek, and Saman Amarasinghe. Streamit: A language for streaming applications. In *International Conference on*

Compiler Construction, Grenoble, France, Apr 2002. {**119**}

[TS12] J. Tompson and K. Schlachter. *An Introduction to the OpenCL Programming Model*, 2012. {**21, 23**}

[Vaj11] András Vajda. *Programming many-core chips*. Springer, 2011. {**16**}

[VMD04] Joël Vennin, Samy Meftali, and Jean-Luc Dekeyser. Understanding and extending systemc user thread package to ia-64 platform. In *Proceedings of International Workshop on IP Based SoC Design*, December 2004. {**66**}

[WCC+12] Xi Wang, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. Undefined behavior: what happened to my code? In *Proceedings of the Asia-Pacific Workshop on Systems*, APSYS '12, pages 9:1–9:7, New York, NY, USA, 2012. ACM. {**4**}

[WK05] Torsten Wilde and James A. Kohl. Port Monitor: A Monitoring & Debugging Approach For Component Frameworks. In *CompFrame 2005, Atlanta GA*, June 2005. {**118**}

[Wol04] Wayne Wolf. The future of multiprocessor systems-on-chips. In *Proceedings of the 41st annual Design Automation Conference*, DAC '04, pages 681–685, New York, NY, USA, 2004. ACM. {**11**}

[WS88] N.J. Wahl and Stephen R. Schach. A methodology and distributed tool for debugging dataflow programs. In *Software Testing, Verification, and Analysis, Proceedings of the Second Workshop on*, jul 1988. {**119**}

[WSBL03] T. Wiegand, G. J. Sullivan, G. Bjontegaard, and A. Luthra. Overview of the H.264/AVC video coding standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7), July 2003. {**94**}

[Zel05] Andreas Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005. {**5**}

ABSTRACT

In this thesis, we propose to study interactive debugging of applications running on embedded systems Multi-Processor System on Chip (MPSoC). A literature study showed that nowadays, the design and development of these applications rely more and more on programming models and development frameworks. These environments gather established algorithmic and programming good-practices, and hence speed up the development process of applications running on MPSoC processors. However, sound programming models are not always sufficient to reach or approach error-free codes, especially in the case of dynamic programming, where they offer little to no help.

Our contribution to lighten these challenges consists in a novel approach for interactive debugging, named Programming Model-Centric Debugging, as well as a prototype debugger implementation. Model-centric debugging raises interactive debugging to the level of programming models, by capturing and interpreting events generated during the application execution (e.g. through breakpointed API function calls). We illustrate how we applied this approach to three different programming models, software components, dataflow and kernel-based programming. Then, we detail how we developed a debugger prototype based on GDB, for STMicroelectronics's STHORM programming environment. STHORM development toolkit provides supportive environments for component, dataflow and kernel-based programming. We also demonstrate how to tackle software debugging with our debugger prototype through four case studies: an augmented reality feature tacker built with components, a dataflow implementation of the H.264 video decoding standard and two scientific HPC computing applications.

RÉSUMÉ

Dans cette thèse, nous proposons d'étudier le débogage interactif d'applications pour les systèmes embarqués MPSoC (Multi-Processor System on Chip). Une étude de l'art a montrée que la conception et le développement de ces applications reposent de plus en plus souvent sur des modèles de programmation et des frameworks de développement. Ces environnements définissent les bonnes pratiques, tant au niveau algorithmique qu'au niveau des techniques de programmation. Ils améliorent ainsi le cycle de développement des applications destinées aux processeurs MPSoC. L'utilisation de modèles de programmation ne garantit cependant pas que les codes pourront être exécutés sans erreur, en particulier dans le cas de la programmation dynamique, où ils offrent très peu d'aide à la vérification.

Notre contribution pour résoudre ces challenges consiste en une nouvelle approche pour le débogage interactif, appelée Programming Model-Centric Debugging, ainsi qu'une implémentation d'un prototype de débogueur. Le débogage centré sur les modèles rapproche le débogage interactif du niveau d'abstraction fourni par les modèles de programmation, en capturant et interprétant les événements générés pendant l'exécution de l'application. Nous avons appliqué cette approche sur trois modèles de programmation, basés sur les composants logiciels, le dataflow et la programmation d'accélérateur par kernels. Ensuite, nous détaillons comment nous avons développé notre prototype de débogueur, basé sur GDB, pour la programmation de la plate-forme STHORM de STMicroelectronics. Nous montrons aussi comment aborder le débogage basé sur les modèles avec quatre études de cas : un code de réalité augmentée construit à l'aide de composants, une implémentation dataflow d'un décodeur vidéo H.264 and deux applications de calcul scientifique.