BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

DESIGN OF ENVIRONMENT FOR MANY-CORE SYSTEMS DEBUGGING

NÁVRH PROSTŘEDÍ PRO LADĚNÍ VÍCEJÁDROVÝCH SYSTÉMŮ

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR
AUTOR PRÁCE
MICHAL KLČO

SUPERVISOR
VEDOUCÍ PRÁCE
Ing. JIŘÍ HYNEK

BRNO 2016

Zadání bakalářské práce/18408/2015/xklcom00

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2015/2016

Zadání bakalářské práce

Řešitel:	**Klčo Michal**
Obor:	Informační technologie
Téma:	**Návrh prostředí pro ladění vícejádrových systémů**
	Design of Environment for Many-Core Systems Debugging
Kategorie:	Uživatelská rozhraní

Pokyny:

1. Seznamte se s technologií vícejádrových systémů (se zaměřením na systémy typu *many-core*) a analyzujte požadavky spojené s laděním těchto systémů.
2. Seznamte se s prostředím Eclipse, projektem Eclipse CDT - DSF a protokolem GDB MI.
3. Navrhněte rozšíření projektu Eclipse CDT - DSF o možnost ladění vícejádrových systémů. Jednotlivá rozšíření konzultujte s vedoucím.
4. Navržené rozšíření implementujte a řádně otestujte.
5. Zhodnoťte dosažené výsledky.

Literatura:

- Clayberg, E.; Rubel, D.: *Eclipse : Building Commercial-Quality Plugins.* Boston: Addison-Wesley Professional, 2008, ISBN 987-0-321-42672-7.
- Kornaros, G.: *Multi-Core Embedded Systems.* Boca Raton: CRC Press, 2010, ISBN 978-1-4398-1161-0.
- Introduction to Programming with DSF. *CDT Plug-in Developer Guide* [online]. [cit. 2015-10-22]. Dostupné z: http://help.eclipse.org/mars/index.jsp.

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 až 3.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí:	**Hynek Jiří, Ing.**, UIFS FIT VUT
Datum zadání:	1. listopadu 2015
Datum odevzdání:	18. května 2016

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
612 66 Brno, Božetěchova 2

L.S.

[illegible]

doc. Dr. Ing. Dušan Kolář
vedoucí ústavu

Abstract

This thesis describe problem of debugging many-core systems using the integrated development environments. It presents some of the integrated environments, debuggers, their features and analyse them. This thesis also describe designs and implementation of modifications of these tools that helps user to debug many-core system more efficiently and comfortable.

Abstrakt

Táto práca popisuje problém ladenia man-core systémov s využitím intergrovaného vývojového prostredia. Predstavuje niektoré z integrovaných prostredí, debuggerov, ich funkcie a analyzuje ich. Táto práca tiež opisuje návrh a implementáciu modifikácií týchto nástrojov, ktoré pomáhajú uživateľovi ladiť many-core systémy efektívnejšie a pohodlnejšie.

Keywords

debugging, multicore systems, integrated development environment, Eclipse

Klíčová slova

ladenie, viacjadrové systémy, integrované vývojové prostredia, Eclipse

Reference

KLČO, Michal. *Design of Environment for Many-Core Systems Debugging.* Brno, 2016. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Hynek Jiří.

Design of Environment for Many-Core Systems Debugging

Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked out by my own under the supervision of Mr. Ing. Jiří Hynek. Also, Mr. Ing. Ondřej Ilčík provides me information. All sources, references and literature used during elaboration of this work are properly cited and listed in complete reference to the due source.

. . . .
Michal Klčo
May 17, 2016

Acknowledgements

I would like to thank to my supervisor, Mr. Ing. Jiří Hynek for his time, help and willingness. Also, I would like to thank to Mr. Ing. Ondřej Ilčík for the topic of the thesis and professional advices, to Messrs. from Cardwork Laboratory for lending graphic card and to my family for their support and respect.

Contents

1 Introduction **3**

2 Multi-core systems and parallelism **5**
- 2.1 Multi-core processor 5
  - 2.1.1 Many-core system 5
- 2.2 Advantages of multi-core systems 6
  - 2.2.1 Power Dissipation 6
  - 2.2.2 Hardware implementation issues 6
  - 2.2.3 Systemic considerations 6

3 Integrated development environments **7**
- 3.1 Eclipse 7
  - 3.1.1 The Eclipse Workbench 8
- 3.2 C/C++ Development Tooling (CDT) 9
  - 3.2.1 Multi-core debugging features 9
  - 3.2.2 Debugger Services Framework (DSF) 10
- 3.3 Codasip IDE 11
- 3.4 Nvidia Nsight Eclipse 12

4 Debugger **13**
- 4.1 GNU Project Debugger (GDB) 13
  - 4.1.1 GDB/MI 14
  - 4.1.2 CUDA-GDB 15
  - 4.1.3 Process/Thread/Core sets 16
- 4.2 Multi-core Debug Solution (MCDS) 16

5 Design of debugging environment **19**
- 5.1 Data exchange 19
- 5.2 Data visualizing 20
- 5.3 Designs of modification and features 20
  - 5.3.1 Modification of GDB/MI 20
  - 5.3.2 Modification of Visualizer View 22
  - 5.3.3 Modification of Debug View 22
  - 5.3.4 Modifications of resource views 23

6 Implementation **26**
- 6.1 Visualizer View modification 26
- 6.2 Debug View modification 28

6.3 Implementation of hardware service . . . 30
6.4 Testing . . . 30
6.4.1 Time measurement of content drawing . . . 30
6.4.2 Usability test . . . 31
6.4.3 Future improvements . . . 32

7 Conclusion **33**

Appendices **35**
List of Appendices . . . 36

A CD content **37**

B Screenshots of the modified Visualizer View **38**

Chapter 1

Introduction

Even in the past it was known that the operating speed of processor cannot be increased again and again. It was when the first considerations of parallelism appeared. But the processor frequency was not as close to its limits as it is today. During the last years multi-core processors became a common part of the desktop personal computers. Nowadays we can find multi-core processors even in a hand-held and mobile devices. This expansion of multi-core systems has influenced the program languages that are used to develop the software for such systems.

The code debugging is the important phase of the software development and its difficulty depends on the used tool, too. This is the place where the tools called debuggers come in play. The first debuggers used *command line interface.* Later they were integrated together with editors and compilers into *integrated development environments.* It allowed creation of the graphical interface to the debug tools and made the work with debugger more effective and easier. Although this has some limits, too. When a developer needs to examine the state of the single-core processor most of the IDEs are suitable. They provide the views that show specific information like a list of threads and their state, values of variables and registers, disassembly code and notifications from the debugger. IDEs also provides services which can control an execution of a program with just clicking on the few buttons. The problems appear when the program is debugged on a multi-core processor. In that case a developer may want to look over information of more threads simultaneously. A developer may even want to control the execution of the program only on the certain threads and keep running the execution on others. This is the option that most of the debuggers do not support. It could be problem to show just the basic information about threads in some IDEs – in many-core systems there can be thousands of them and the typical view might not show its content properly and make an IDE to react with longer delays. Another request might be displayed with the same type of information of multiple threads, e.g. variables comparison.

These are the tasks for which a lot of IDEs are not built properly. My goal is to design and implement an extension for the Eclipse platform to support debugging of many-core systems. There exist some projects that solve these problems at least partially and some inspiration has been taken for this project from them.

In the following chapters the terms and connections are explained. The second chapter describes multi-core systems and advantages of the multi-core approach. In the third chapter the integrated development environment is introduced especially the Eclipse platform and some of the extending plug-ins. The fourth chapter describes debuggers, the GNU Debugger Project and the known features of debuggers that provide efficient communication with a front-end and debugging of multi-core systems. The other chapters are dedicated to the design and modifications of the debugging environment, implementation and conclusion.

Chapter 2

Multi-core systems and parallelism

The multi-core solutions have become more attractive solutions than uniprocessors in the last years for various reasons. It is known that frequency of a processor can not be risen more and more and the manufacturers of the processors are still closer to the technology limits. Historically the idea of the multiprocessors appeared as in early 1970s. In a landmark 1991 paper by Stone and Cocke [10]the authors argued that operating frequency 250MHz cannot be achieved and therefore there cannot be provided the kind of performance required by the future applications with a single processor. This prediction was proven false and the progress in the speed performance of uniprocessor made parallel processing less attractive. However the multiprocessors made a comeback and took advantage of various systems [6].

2.1 Multi-core processor

A multi-core processor is a single computing unit with two or more processing units (cores) that are able to read and execute program instructions. Simultaneous execution of an instruction increases the overall speed of the processor for programs that are available to parallel computing. Cores are typically integrated onto a single circuit die or onto multiple dies in a single package.

The communication between cores can be designed differently. There might be used shared caches or shared memory or message passing refer to the implementation. There are used also the network topologies like bus or ring to interconnect cores. Multi-core systems can be homogeneous with cores of the same type or heterogeneous with the cores of different types.

2.1.1 Many-core system

There is no specific definition of the many-core system. Generally, it is a high count of cores put onto a single device which is highly parallel. The software also has to be implemented highly parallel to take use of such system as much as it can. It is appropriate for the task that can be decomposed to same small tasks that can be performed simultaneously.

2.2 Advantages of multi-core systems

2.2.1 Power Dissipation

In the past the performance and cost were the primary considerations in the system design. With the arrival of battery-operated mobile devices the energy dissipation became more relevant.

Using a multi-core design complex systems can be divided into power domains. The power switches are then used to cut off a power supply to a sub-system which is not required to be active. It helps cut down the static and dynamic power that would be wasted. It is also common to use dynamic voltage and frequency scaling (DVFS). Sub-systems that need to provide the higher performance can operate at higher frequencies and voltage.

Multi-core system design offers efficient relation between performance and power consumption. In comparison to uniprocessor operating at high frequency with high power dissipation a group of processors with lower frequency and significantly reduced power consumption can provide a comparable performance.

2.2.2 Hardware implementation issues

The *timing closure*[1] problem in an automated design flow causes difficulties. Therefore it is cheaper to design a multiprocessor system-on-chip where the processors work at moderate speed than to design a single processor that work at much higher frequency with similar system throughput. The delays caused by the parasitic inductance and capacitance of interconnects make it difficult to predict critical path delays accurately during a design.

It also brings advantages during the testing. When there is a number of identical cores it is possible to reuse the test patterns and reduce the effort in test generation. Application of random pattern on the identical processor and comparing their responses is another self-test approach. If there is a difference in the responses, it indicates an error.

2.2.3 Systemic considerations

In the case of single processor alternative the compilers written for such processors have a limited scope of extracting the parallelism in applications. There are techniques like *out of order execution*[2] or *speculative execution*[3] to increase the compute power. However extracting parallelism from a single thread is prohibitive. With many applications resorting to multithreading it is more appropriate to have a multi-core system with a compiler that has more visibility of MIMD-type parallelism (Multiple Instruction, Multiple Data).

[1]problems occur when the timing estimates computed during logic synthesis do not match with the timing estimates computed from the layout of the circuit.

[2]http://searchdatacenter.techtarget.com/definition/out-of-order-execution

[3]http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-823-computer-system-architecture-fall-2005/lecture-notes/l13_brnchpred.pdf

Chapter 3

Integrated development environments

Modern applications consist of components implemented in many different technologies. A lot of the tools such as editor, compiler, interpreter or debugger supporting various programming languages are used. Integrated development environment (IDE) provide integration of such tools and tries to support development task flows. The main aim is to improve the productivity of developers. There were found the new ways how to achieve it. Simple editors were replaced by complex ones with a presentation of the program in meaningful ways (showing subclass hierarchy), automatic code completion and visual editors for creating graphical user interfaces (GUIs). Some popular IDEs are Eclipse, NetBeans or Microsoft Visual Studio.

3.1 Eclipse

The Eclipse Platform is an open-ended, language-neutral IDE that was released in late 2001. Most of its functionality is very generic. It takes additional tools to extend the platform to work with new contents or to focus functionality to something specific. These extending modules are called *plug-ins.* Tool provider writes a tool as a separate plug-in which takes care of showing its tool-specific parts of UI in the Eclipse's workbench. The quality of such a tool depends on how well it is integrated with the platform and how well it works with other tools. Tools can be written as a single plug-in or can be split across several plug-ins according to its complexness [8].

Eclipse is not the monolithic SW. It is a small kernel containing a plug-in loader. This kernel is an implementation of the *OSGi* R4 specification and provides the environment on which the plug-ins are executed. OSGi is a runtime model designed and implemented specifically for the Eclipse. This modular design provides the functionality that can be more easily reused to build an application by no-Eclipse developers. The Eclipse Rich Client Platform is the minimal set of plug-ins necessary to create a client application.

Plug-ins are coded in the Java language and generally consist of Java code, libraries, read-only files and some resources. Some of them do not contain code at all. On-line help is the example of a plug-in that consists of HTML pages and no Java code. The important parts of each plug-in are also *MANIFEST.MF* and *plugin.xml* files where declarations of interconnections to other plug-in are defined. These interconnections are created by declaring any number of *extensions* to the *extension points* of another plug-in. The extension

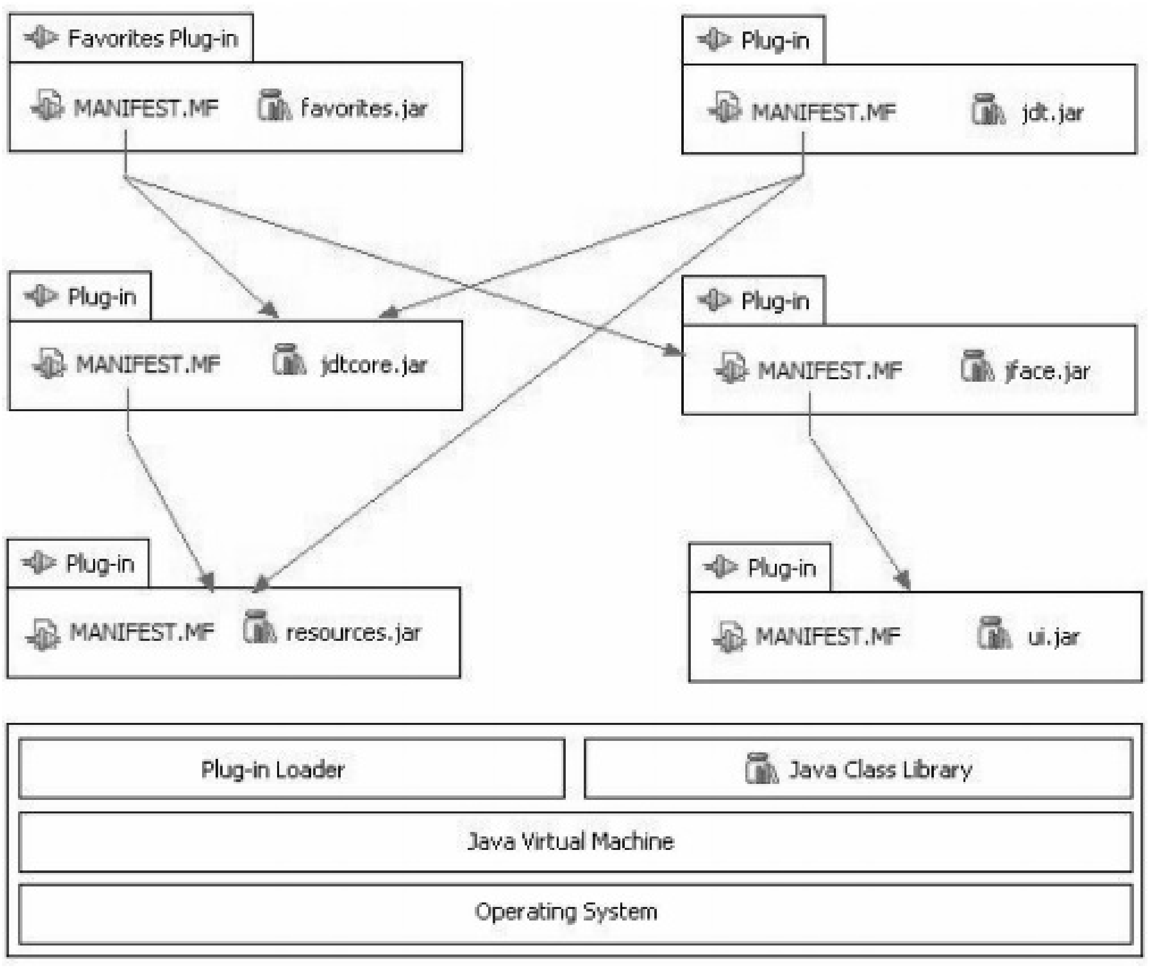


Figure 3.1: Eclipse plug-in structure. [1]

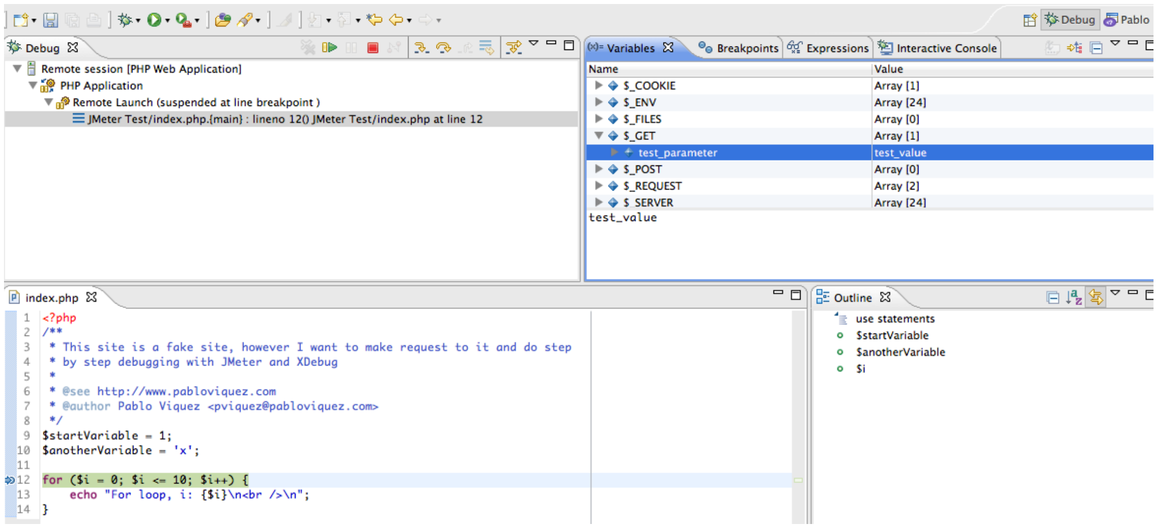
An example of how plug-ins depend on each other.

points are used as a mechanism for coupling chunks of functionality. They can be declared in the plug-in manifest, exposing a minimal set of the interfaces and related classes for other to use. Other plug-ins declare extensions to these extension points, implementing exposed interfaces or referencing on the provided classes.

On the start-up, a set of available plug-ins is discovered, their manifest files are read and a plug-in registry is built according to it. However during the start-up, not every plug-in is activated. They are activated when they are needed by calling a method from a small number of methods used explicitly for activating the plug-ins. This helps the application to handle the big amount of plug-ins and avoids the lengthy start-up. Once activated it remains active until the platform shuts down.

3.1.1 The Eclipse Workbench

The term *Workbench* refers to the desktop development environment and in the case of the Eclipse each Workbench window contains one or more *perspectives*. Perspective is a combination of the various *views* and *editors*. Multiple perspectives can be active, but only one can be shown at once. Views are used to navigate resources and modify the properties



```
Remote session [PHP Web Application]
  PHP Application
    Remote Launch (suspended at line breakpoint )
      JMeter Test/index.php.{main} : lineno 12() JMeter Test/index.php at line 12
```

Name	Value
$_COOKIE	Array [1]
$_ENV	Array [24]
$_FILES	Array [0]
$_GET	Array [1]
test_parameter	test_value
$_POST	Array [0]
$_REQUEST	Array [2]
$_SERVER	Array [24]

```
test_value
```

```php
<?php
/**
 * This site is a fake site, however I want to make request to it and do step
 * by step debugging with JMeter and XDebug
 *
 * @see http://www.pabloviquez.com
 * @author Pablo Viquez <pviquez@pabloviquez.com>
 */
$startVariable = 1;
$anotherVariable = 'x';

for ($i = 0; $i <= 10; $i++) {
    echo "For loop, i: {$i}\n<br />\n";
}
```

```
use statements
$startVariable
$anotherVariable
$i
```

Figure 3.2: Screenshot of the debug perspective in Eclipse IDE.

of resources. They also serve to visualize the data provided by various services. Editors allow to modify specific resource too, but they follow the open-save-close model whereas changes made within a view are saved immediately [1].

Each perspective has its own set of views that can be modified by the user to make the environment more comfortable. On the other hand open editors are shared by all open perspectives. Each view and editor can be resized by dragging the sizing border or moved within the workbench.

3.2 C/C++ Development Tooling (CDT)

C/C++ development tooling is set of plug-ins that extend the Eclipse Platform with specific-language (C/C++) functionality. It includes a lot of features like a compilation of code using *make* tool, source navigation, syntax highlighting, source code refactoring and code generation, visual debugging tools, and knowledge tools like type hierarchy, call graph, include browser and more. [3]

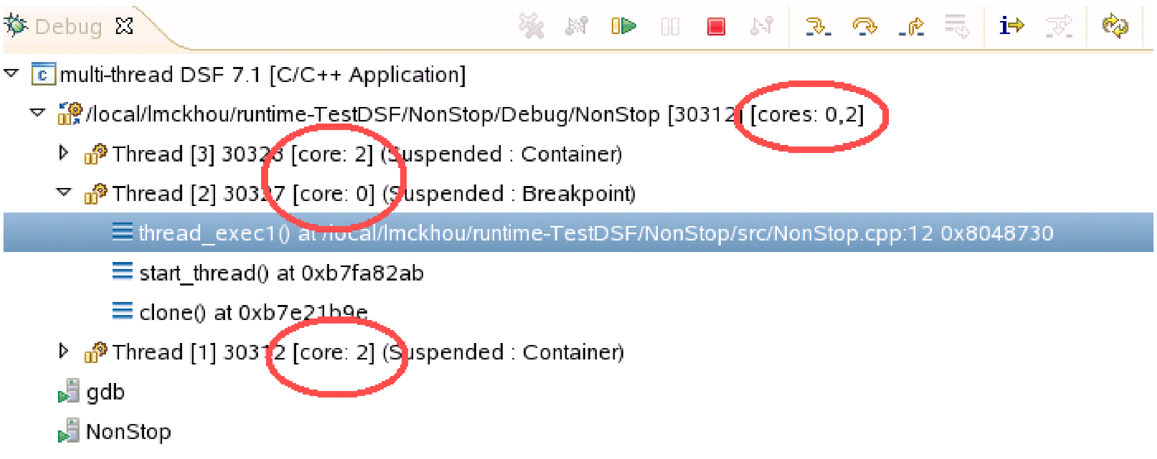
3.2.1 Multi-core debugging features

Showing cores in Debug View labels

The Debug View allows a user to manage the debugging or running of a program. It shows the the stack frame for the suspended threads for each target that is debugged. Each thread in a debugged program appears as a node in the tree. This feature extends thread nodes to show the core they are running on. [2]

Pin & Clone

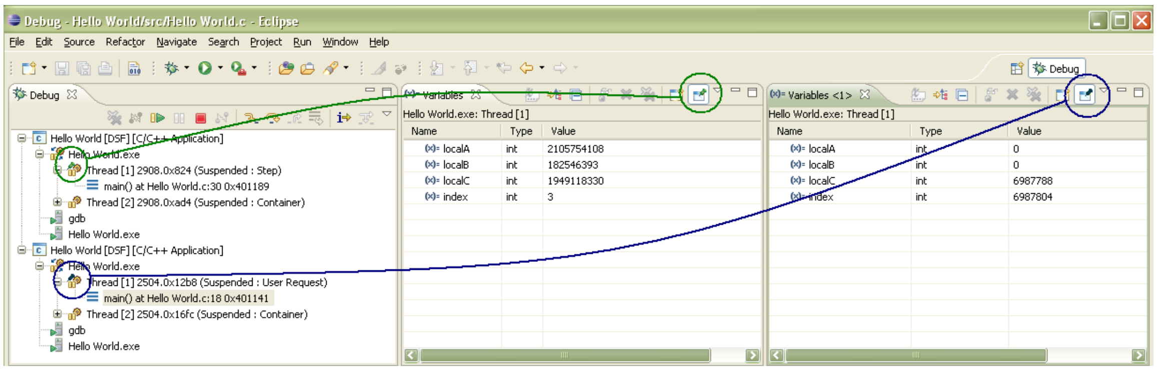
Pin & Clone is the feature that make possible to show same type of information in multiple views but for different threads in each view.



```
Debug

multi-thread DSF 7.1 [C/C++ Application]
  /local/lmckhou/runtime-TestDSF/NonStop/Debug/NonStop [30312] [cores: 0,2]
    Thread [3] 30328 [core: 2] (Suspended : Container)
    Thread [2] 30317 [core: 0] (Suspended : Breakpoint)
      thread_exec1() at /local/lmckhou/runtime-TestDSF/NonStop/src/NonStop.cpp:12 0x8048730
      start_thread() at 0xb7fa82ab
      clone() at 0xb7e21b9e
    Thread [1] 30312 [core: 2] (Suspended : Container)
  gdb
  NonStop
```

Figure 3.3: Screenshot of Debug view. [2]



```
Debug - Hello World/src/Hello World.c - Eclipse
File  Edit  Source  Refactor  Navigate  Search  Project  Run  Window  Help

Debug
Hello World [DSF] [C/C++ Application]
  Hello World.exe
    Thread [1] 2908.0x824 (Suspended : Step)
      main() at Hello World.c:30 0x401189
    Thread [2] 2908.0xad4 (Suspended : Container)
  gdb
  Hello World.exe
Hello World [DSF] [C/C++ Application]
  Hello World.exe
    Thread [1] 2504.0x12b8 (Suspended : User Request)
      main() at Hello World.c:18 0x401141
    Thread [2] 2504.0x16fc (Suspended : Container)
  gdb
  Hello World.exe
```

Variables — Hello World.exe: Thread [1]

Name	Type	Value
localA	int	2105754108
localB	int	182546393
localC	int	1949118330
index	int	3

Variables <1> — Hello World.exe: Thread [1]

Name	Type	Value
localA	int	0
localB	int	0
localC	int	6987788
index	int	6987804

Figure 3.4: Screenshot of Pin & Clone feature. [3]

Multi-core Visualizer

The Visualizer View displays the graphical representation of the current state of a debugged system. It shows cores, threads and processes in the visual grid. The Visualizer View is connected with the Debug View, it allows to select processes/threads with click and drag and apply debugging commands to them.

3.2.2 Debugger Services Framework (DSF)

Debugger Services Framework is API that integrates CDT debugger into the Eclipse. It provides extensive control over the content of debugger views and helps to achieve better performance when debugging slow targets. DSF uses the GNU project debugger as underlying debug engine. It sends a sequence of GDB commands to the GDB according to the user actions and processes the output from GDB to display the state of the debugged system. [2]

One of the interesting features of the DSF is a high utilization of asynchronous methods. These methods use a callback object to indicate their completion and return results. The standard callback object is `RequestMonitor` which contains callback methods. These methods can be overridden to add additional processing after the asynchronous method has

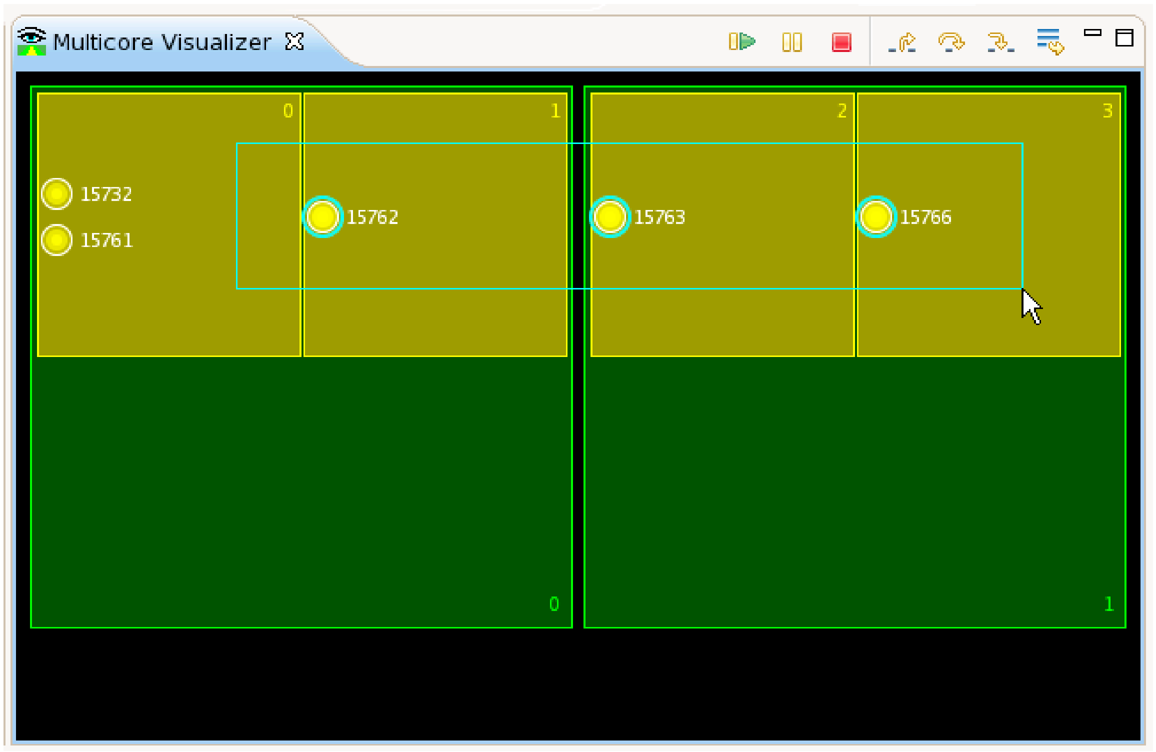


Figure 3.5: CDT Visualizer view. [3]

An example of selecting multiple processes running on multiple cores.

been completed. Another object needed by `RequestMonitor` is `Executor`. It extends the Java class `Thread` that solves the problem of race condition – simultaneous access of multiple processes to shared resources. It handles the synchronization of these processes. Thus if all asynchronous methods are called in one thread (executor) there is no simultaneous access to the shared resources.

3.3 Codasip IDE

Codasip IDE is the IDE based on Eclipse that integrates tools for designing *Application Specific Instruction Set Processors* [5] (ASIPs). The intention of this work is to design a debugging environment for this IDE. The Codasip extends CDT plug-ins to provide a connection with Codasip simulator that is based on GDB version 7.4. Despite that these extensions provide projects (CodAL project), CodAL language support, specific views and perspectives and integration of other required tools.

The debug perspective of Codasip IDE provides a complex environment for debugging designed systems. It consists of detail views that show variables and resources like ASIP's registers, ports, signals and their values and amount of writes and reads. As well as the Eclipse's basic debug perspective it has the Debug View, views for console and debugger output, the Disassembly View and others.

3.4 Nvidia Nsight Eclipse

Nsight Eclipse is modified IDE powered by the Eclipse platform. It integrates the tools for edit, build, debug and profile *CUDA-C*[1] applications.

As CUDA applications run on *graphical processor units*[2] which can be considered as the many-core system, Nsight Eclipse provide debug environment for debugging many-core systems. It involves features like ability to control and observe only a part of the debugged system. Some of the designed features for Codasip IDE have been inspired by Nsight Eclipse.

[1] `https://developer.nvidia.com/about-cuda`

[2] `http://www.nvidia.com/object/gpu.html`

Chapter 4

Debugger

A debugger is a tool used to debug the target program and get useful information about the run of the program like values of variables, disassembly code. It also examines the behaviour of the program when a code is changed. There is an option to run code with the instruction set simulator. This technique allows stop the execution of the program upon meeting some specific condition.

When the debugged program cannot normally continue a *trap* occurs. Then debugger typically shows the position in the code where en error is located. It can show the error in source code or in the disassembly, depends on whether it is a *source-level debugger* (shows the location in source code) or a *low-level debugger* (shows location in machine code).

Debuggers offer features like *stopping (breaking)*. It allows to pause debugged program at some event or *breakpoint* to observe current values of variables or processor's register. Some of the debuggers have abilities like modifying the code during the run or bypassing an error in the program and continuing at a different location.

The most used debuggers like the GDB have only the *Command Line Interface* (CLI) that makes them portable and light-weighted. The most of the developers prefers debugging via *graphical user interface* (GUI) because it is more user-friendly. Debugger front-ends can be compatible with a more CLI-based debugger.

Debuggers contain also the interesting function called reverse debugging that is not commonly used. Reverse debuggers exist for C/C++, Java, Python and some other languages. It makes possible stepping backward in time or causes the slowdown of the target.

4.1 GNU Project Debugger (GDB)

GDB is the one of the most popular debuggers. It supports the debugging of programs written in C/C++, Pascal, Ada and other languages. GDB has four main functions:

- start the program
- stop the program on specified condition
- examine the state of program when it is stopped
- change the things in program

A program can be debugged with the GDB executing simple command **gdb** and the name of the binary file of compiled program as an argument. With the command **run** the program is executed under debugger control. When the program stops execution can be advanced on the next line with command **next** or just **n**. If command **step** or **s** is used instead execution is advanced in any subroutine. Another useful command for examining the content of the stack is **backtrace** or just **bt**. It prints out the stack frame for each active subroutine. For showing another information like values of variables command **print** or **p** with the name of the variable as argument can be used. Commands **continue** or **c** resume the running of the program and **quit** end the debugger. This is the basic list of commands for a debugging program. [9]

4.1.1 GDB/MI

GDB/MI is a line based machine oriented text interface to gdb and is activated with the `-interpreter` command line option. The communication of GDB/MI front-end with gdb consists of three parts:

- commands sent to gdb
- responses to commands
- notifications

Each command evokes exactly one response which indicates completion of the command or an error. In these responses the requested information is sent. In the case of the `resume` command the response only indicates if the target was successfully resumed. Notifications provide the information about the change of target but cannot be associated with a response to commands. Important examples of notifications are the following:

- Exec notifications. They report the changes in target state – when a target is stopped or resumed. This information cannot be included in responses to the resume commands because one resume command can cause multiple changes in different threads.
- Console output and status notifications. They report information about the execution of commands and progress of long-running operations that has to be printed before the command is complete.
- General notifications. These notifications can inform front-end about side-effects of commands or some other changes to a target.

When we want to retrieve the information during the debug session we have to ask in context of a specific thread and frame. The stack frame is part of the stack associated with the calling of function and consists of the given arguments, local variables and the address which the function is executing. This context is maintained and remembered by gdb and it supplies on each command. It means that not each command need the context to be specified. In the case of MI and communication between front-end and gdb, this preserving of context is not so useful. In the front-end, there can be more views associated to different threads or the need to acquire data in the context of thread that is not focused. MI shares the selected thread with CLI and because of that, the simplest way how to switch thread is using the `-thread-select` command. However this could double the count of messages exchanged with front-end and causes longer delays during showing of the actual data. The MI commands allow to specify the context for each command separately by passing the `-thread` and `-frame` options with the global identifier for thread and frame as values.

The *asynchronous command execution* is the capability of gdb to process MI commands even if the target is running. The target has to support it and the front-end has to specify `-gdb-set mi-async on` (`target-async` option instead in version <=7.7) before running the executable or attaching to the target. If asynchronous command execution is off the gdb has to wait for the program to stop before processing further commands.

Some commands that access the running target might work even if gdb accepts them. Therefore the combination of asynchronous command execution with *non-stop mode* is more useful. This mode allows stopping the target thread and examining it while other threads are still running. This helps to debug a program where threads have real-time constraints or need to react to external events. The `set non-stop on` command has to be used before running or attaching program. All execution commands are applied to the one thread, e.g. `continue` command is applied on the thread that is stopped. But stopping just one thread does not change the current thread. Again it is recommended to use `-thread` and `-frame` options.

GDB as the debugger can support debugging of multiple processes or several hardware systems with several cores running many processes on each core. When the target thread is specified with the `-thread` option, it is not known which process that thread belongs to. To discover grouping of the threads in processes and to support of hierarchy machines/-cores/processes, MI introduces the concept of *thread group*. A thread group is a collection of threads. With the `-list-thread-groups` command, top-level thread groups that correspond to the processes of the debugged target can be listed. Each thread group has string identifier that can be passed as an argument to the `-list-thread-group` command to obtain members of the specific thread group. To obtain a list of threads that are not debugged but can be attached, the concept of *available thread group* is introduced.

4.1.2 CUDA-GDB

With the interesting idea to use the high-performance graphics processing unit (GPU) for parallel computing, Nvidia invented programming model CUDA. GPU typically consists of hundreds of small processing units that can be used for the efficient computing of highly parallelizable tasks. For developers it is required to provide a debugger for such device. This is familiar to the problem of debugging programs on the many-core systems.

CUDA-GDB is Nvidia tool for debugging CUDA applications. It is an extension to the port of GDB and is capable of debugging CUDA applications on actual hardware. The already implemented commands of GDB are unchanged. CUDA commands are just prefixed with **`cuda`** keyword. For example, `info thread` prints out list of threads of the target and the `info cuda threads` command prints out a list of a current group of 32 threads called *warp*. [7]

To simplify visualization of the information about the state of an application, commands are applied to the entity of focus. Since the CUDA-GDB also supports debugging on the host (CPU), the entity of focus can be a host thread or a device thread. The entity of focus is always the lowest granularity level device thread. To set the focus on the specific entity, the two kinds of coordinates are used - hardware and software coordinates. The software coordinates consist of threads that belong to a block that in turn belongs to a kernel. Lanes belong to a warp that belongs to *streaming multiprocessor* [11] (SM) that is contained in a device. These are the hardware coordinates. The both of the coordinates can be used interchangeably. One thread is executed on one lane. The interesting fact is that the `step` command can be applied only on a warp (not on a single device thread).

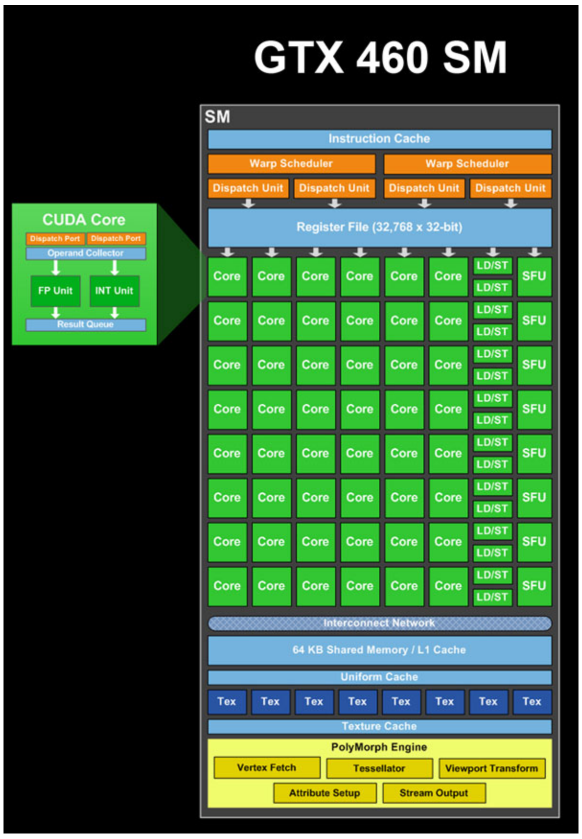


Figure 4.1: Architecture of a stream multiprocessor of GTX460 graphic adapter.

4.1.3 Process/Thread/Core sets

In a program with hundreds or even thousands of threads the two options that GDB currently provides can be limiting. Either work with one thread or all of them at once. PTC sets extend commands of GDB with options to use a command with multiple threads, processes or cores. There are few examples:

- `step .34-59` - step threads numbered between 34 and 59
- `step @2` - step all threads running on core 2
- `interrupt *,future@5-7` - stop everything running on cores 5 to 7, preventing new threads from being started

Although, no record about real deployment has been found.

4.2 Multi-core Debug Solution (MCDS)

This debug solutions handle debugging of embedded multiprocessor *system-on-chip* (SoC). It allows executing real-time accurate tracing of selected processors, buses, signals within the chip non-intrusively and without adding pins to the chip. It offers the benefits like the full visibility of cycle-accurate trace, complex triggering modes, support for code profiling and more [4].

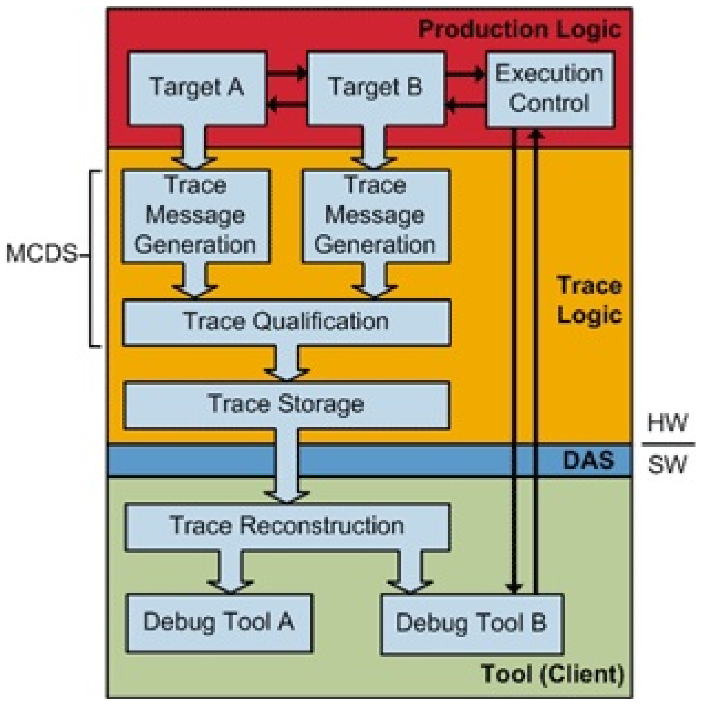


Figure 4.2: Concept of data flow in MCDS. [4]

MCDS works on a principle of collection and store of the trace data by adding logic to a chip. This data are transferred to the debug system for the further analysis.

Included trace logic can be implemented either in production chip or in a special debug die. MCDS supports any number of debug targets. A target can be a signal, a bus or a processor. The trace logic collects the information about the target like instruction pointers, addresses, data, signal values and forwards them to the trace storage when specified condition is met. Message filtering can be used to not collect all of the available data all of the time.

The trace storage holds the trace messages in the memory that can be accessed by a Device Access Server (DAS) over a hardware interface such as JTAG. The trace messages are compressed to minimized required size of memory. The DAS is the abstraction of the physical connection which provides a generic interface to the debug tools.

MCDS consists of software and hardware part of the debug system. Hardware side is set of certain blocks assembled as it needed. *Processor Adaptation Logic* (PAL) or *Bus Adaptation Logic* connects processor/bus to its corresponding *Processor Observation Block*/*Bus Observation Block* (POB/BOB). The observation blocks collect trace data, assemble and buffer trace messages and extract the triggers for trace qualification. *The Multi-core Cross-connect* (MCX) enables simultaneous debugging of multiple targets. And the *Debug Memory Controller* (DMC) sorts messages from observation blocks and forwards them to the user.

This debug solution, however, is not appropriate for debugging many-core systems. Such amount of observed cores is overwhelming for implementation and maybe impossible. Nevertheless, it is used solution for embedded multi-core systems which is worth to analyze.

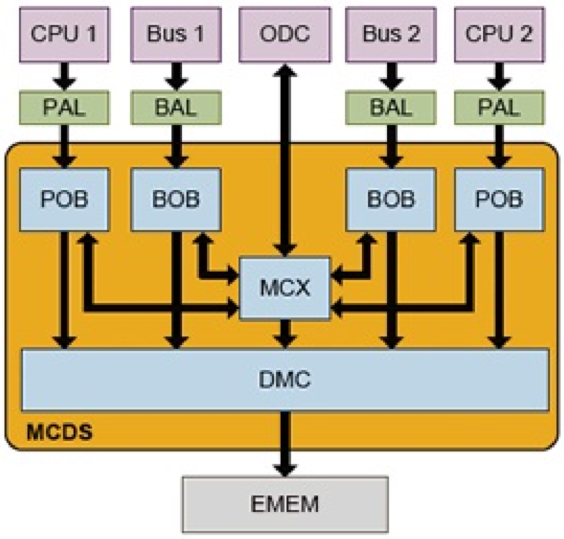


Figure 4.3: Example of hardware implementation of MCDS. [4]

Chapter 5

Design of debugging environment

Debugger integrated into nowadays IDEs is the common thing. Though, the most of them cannot deal with the many-core systems properly. The high amount of cores (hundreds or thousands) causes the slow retrieving of data from debugger and presentation of such amount of information is not compendious.

The main problem can be decomposed into the two partial tasks:

- The first task is discovery of appropriate communication protocol that allows targeting and controlling certain parts of a multi-core system. It is required to ensure that there will be a reasonable amount of exchanged data. If there is a need to get the detailed data about a part of the system, only data about that observed sub-system should be retrieved.
- The second task is finding a way how to show all of the important information from the debugger well-arranged and allow the user to choose which kind of information of selected debugged system part would be displayed.

5.1 Data exchange

This part of the problem needs the DSF to be changed to properly handle communication with the debugger. On the debugger side there are two approaches that can be used – the use of the PTC sets or modification of the GDB/MI (described in 4.1.1).

The PTC sets as mentioned before extend the GDB commands with an option to target the multiple threads. With the combination of DSF and CDT's Visualizer view it could be the most efficient way how to retrieve data from the debugger and control the execution of a debugged program. The single GDB command would be applied on all the threads selected in the Visualizer view. Although, there could be a problem with caching of responses from a debugger.

Another approach is to improve communication trough GDB/MI interface. The options `-thread` and `-frame` allow to specify the context for each command. For each selected thread a command will be sent to the debugger. However, DSF offers the command caching that could make the front-end to not ask for the same information repeatedly.

I chose combination of these two approaches. In this chapter the modification of GDB/MI is described. The idea of using context management that is provided by PTC sets is taken and used to design improvements for the GDB/MI. Although, there have to be ensured sending commands to the debugger according to actions of the Visualizer view.

5.2 Data visualizing

Since the goal is to create an extension for CDT and DSF plug-ins, already created features will be used and improved. First of all the thread hierarchy in the Debug View (described in 3.2.1) could help in the presentation of a big amount of threads. The model of Debug View can be changed to show the machine which the program is debugged on as a node in a hierarchical tree. This node could be expanded to the list of cores and each core would expand into a list of threads. This would decrease the amount of content in the Debug View and amount of data exchanged with the debugger.

Another useful feature is the Pin & Clone (presented in 3.2.1). In the Nvidia Nsight Eclipse (described in 3.4) there can be multiple threads selected (pinned) simultaneously. For all the selected threads the information in other views is shown. This may not be efficient if many threads would be selected but when just a few values of variables on different threads need to be compared, it could be helpful and effective.

The most valuable feature is Visualizer View (presented in 3.2.1). The DSF provides a framework to design own Visualizer view. The key information, that should be visualized are cores, threads, and processes – their activity, state, PIDs. There are different designs of the Visualizer View which are already used in IDEs. Some of them even display the network topology of a processor and its capacity utilization. In this work the Visualizer view will be used and modified to meet the requirements for the Codasip IDE many-core debugging environment.

5.3 Designs of modification and features

In this section the modification of existing features is described. The improvements of the specific features like GDB/MI protocol or Visualizer view are designed to handle debugging of many-core systems. They also need to be compatible with the Codasip IDE and its ASIP design.

5.3.1 Modification of GDB/MI

It is possible to debug multiple threads running on multiple cores through the GDB/MI interface. But as was mentioned in 5.1, there are problems with debugging many-core systems. Specifically the problem is a number of messages and notifications exchanged with front-end and the context management.

The one of the most produced messages in communication between front-end and GDB/MI are messages called *async records* that are associated with the thread activity. In the most cases they are a consequence of GDB/MI commands or target activity and provide additional information about changes. Every time a thread is resumed or stopped, one async record is generated and sent to the front-end. If the non-stop mode (presented in 3.2.1) is not used and one hundred threads are debugged, two hundred async records are generated after each step that is made in the debugging process. Similarly the async records are created when a thread or a thread group is created or exited. Coalescing of these async records is the option which can decrease the number of produced records. To achieve it the modification of communication protocol is needed. Also the timers that delay the sending of records are required.

The following list of messages shows the selected async records and suggested modifications of these records:

- `*running,thread-id="thread"`

  Description:

  - It notifies front-end that thread is running. The `thread-id` field can be the global thread ID that identifies thread or `all` if all threads are running.

  Modification:

  - Additional result `thread-ids="thread-list"` is added instead of the `thread-id` variable. The `thread-list` field can be either the list or the range of global thread identifiers.

- `*stopped,reason="reason",thread-id="id",stopped-threads="stopped",`
  `core="core"`

  Description:

  - It notifies that the thread has stopped. The `id` field identifies the thread that caused the suspension by the global thread ID. The `reason` field determines the reason why thread stopped and the `core` field identifies the core on which the event happened. The value of `stopped` item can be `all` when all threads are stopped. Otherwise it contains identifiers of particular stopped threads. Presently the list always includes a single thread.

  Modification:

  - There can be used the `stopped` field as the list of the global thread IDs and the global thread IDs ranges.

- `=thread-created,id="id",group-id="gid"`
  `=thread-exited,id="id",group-id="gid"`

  Description:

  - It notifies that the thread has been created or exited. The `id` field contains global identifier of thread and the `gid` field identifies the thread group this thread belongs to.

  Modification:

  - There can be added the `ids="thread-list"` result instead of the `id` variable and the `group-ids="gid-list"` result instead of the `group-id` variable. As well as in the case of other async records, the `thread-list` and `gid-list` fields are lists that contain identifiers of threads (or thread IDs range) and identifiers of thread group identifiers.

The reason for the next modification was to reduce an amount of the GDB/MI commands that are sent to the GDB by front-end. One way how to do it is to select a set of commands and extend them with PTC (presented in 4.1.3) sets. There is a demand to control a specific part of the system (select cores, processes, threads) and get information about that part of the system. The program execution commands and thread commands are the most suitable because they allow to stop or resume threads and get information about threads. The commands that switch context will not be extended. The most of front-ends do not rely on currently selected context but they use `-thread-group` and `-thread` options. The following list shows the specific commands that will be extended with PTC sets:

- `-thread-info [ thread-id ]`
- `-exec-continue [--reverse] [--all|--thread-group N]`
- `-exec-interrupt [--all|--thread-group N]`
- `-exec-next [--reverse]`
- `-exec-step [--reverse]`

The modification of this commands consists of adding the `-set` *ptc-value* option. Same as the `-thread` option it allows to choose the target part of the system. The *ptc-value* field can be valued with the following notation:

```
process-id.thread-id@core-id
```

All the `id` fields in this notation can be enumeration of identifiers where identifiers are separated by comma or wildcard (*) to identify all processes, threads or cores. Range of ids can be used, too.

5.3.2 Modification of Visualizer View

The Visualizer View is available as the standalone plug-in that offers the basic graphical representation of the debugged system. A lot of other development environments include modified Visualizer view that is extended to show additional information. Similarly this modification is based on Codasip IDE requirements.

To properly design functional system in Codasip IDE a user has to create the CodAL project. A user has to specify the processing unit (ASIP) of the whole system. He also has to specify the surroundings of ASIPs like memories and interfaces that are used to be connected with other platforms and ASIPs. The levels within the project can be used to divide components hierarchicaly. One level can contain multiple ASIPs and levels. Therefore there will be many options how to build a system with a complex hierarchy of the multi-core or many-core system.

The goal of this modification is to show the hierarchy of debugged system in the Visualizer View. Such graphical representation helps to distinguish different levels of debugged system and helps to identify ASIPs. It is also easier to find and select specific threads that a user wants to control, pin or get information about. Figure 5.1 shows mock-up of the modified Visualizer View that represents the hierarchy of platforms and ASIPs located on different levels.

5.3.3 Modification of Debug View

The Debug View is the key component of the debug perspective. It shows which processes and threads are running on the debugged system. It is also the main interface to control and observe parts of the system. Although there are no improvements to effectively show a big amount of threads and processes.

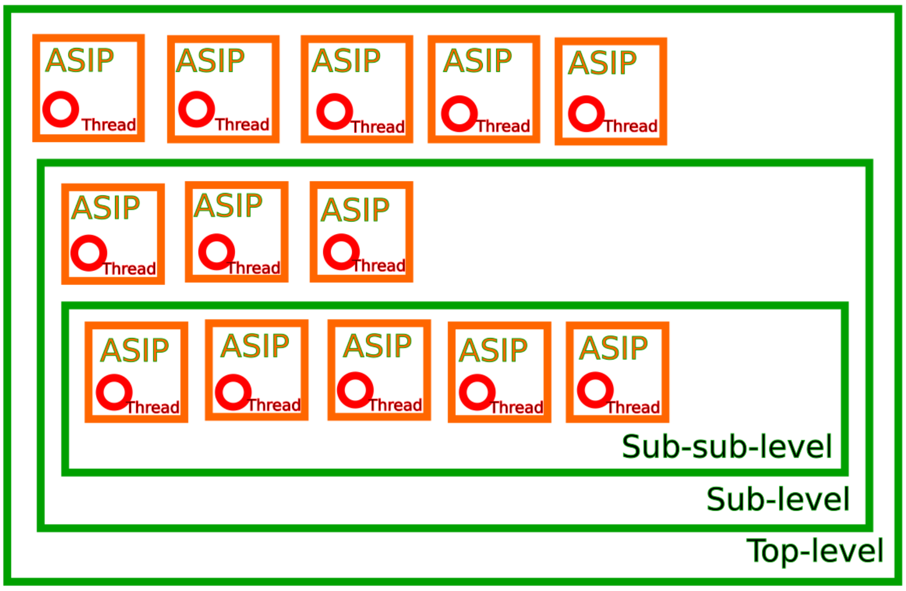


Figure 5.1: Mock-up of the modified Visualizer View.

Example of system with multiple platforms and ASIPs.

Modification of this view consists of two parts – add support for visualization of the Codasip level hierarchy and grouping action for the thread groups and threads. The first part is similar to the modification of the Visualizer View. It divides threads according to the relation with ASIP or level they are running on. Analogous to the Visualizer View it helps to find individual ASIP in a possible complex hierarchy. The second part helps to reduce the amount of shown nodes in the Debug view. When a lot of thread nodes or thread group nodes int the Debug View would be shown on a single level, additional nodes will be created and aggregate the thread nodes or the thread group nodes under these nodes. Also if there is no need to retrieve the detail information about grouped nodes, this decrease of shown nodes reduces the amount of exchanged messages between the front-end and debugger.

5.3.4 Modifications of resource views

Values of the system resources and variables are the valuable information that helps to debug an application. With the parallel application it is useful to be able to compare these values on different thread or core. Pin & Clone is a feature that allows it. It is possible to open one of the detail views (e.g. Register View), pin to the context of the specific thread, open another instance of the same view and pin it to the context of another thread. When two different contexts are compared the views do not occupy a big part of the workbench. However if the user wants to compare resources of many different cores, multiple views can hide other useful information or not show all the information in an efficient way.

The Nsight Eclipse comes with the interesting modification of the Variables View. When the contexts of the threads are selected and pinned the new column is created in the Variables view for each pinned thread. Each value is shown for each variable and each pinned thread in the associated column. This idea is used to design a new view for comparing system resources.

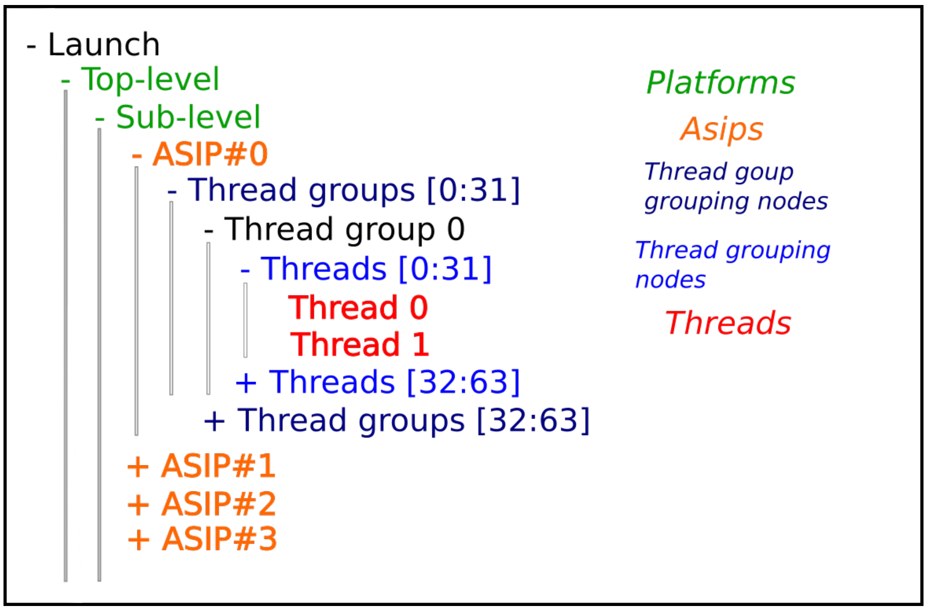
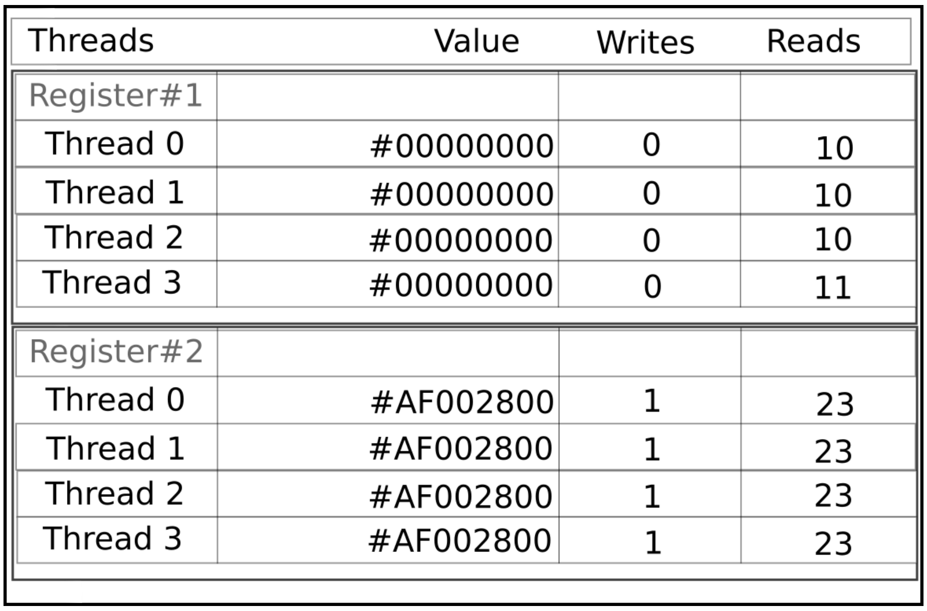


Figure 5.2: Mock-up of the modified Debug View.

Example of visualizing the Codasip level hierarchy and thread group/thread grouping.

The basic idea is to create a view that is able to show multiple values of resources for various contexts. The modification of the Variables View used in the Nsight Eclipse is usable only for that view. It is because there is provided more information about resources in Codasip IDE. Despite the value of the resource there are counts of writes and reads of the resource. In this case it is required to show 3 dimensions – a list of resources, a list of pinned contexts, a list of information of resource. A 2-dimensional view can be divided into blocks that show one item from the third dimension (e.g. one resource from a list of resources). The following image shows one of the possible layouts of the view. Each column provides values of different information, each line provides values of different context and each block is relative to a specific resource.

Even this layout is not proper for every debugged system. In Codasip IDE some of the debugged systems can be heterogeneous and have various types of ASIP. Each ASIP can have different resources and therefore they are not comparable. It requires some protection to avoid the comparison of two or more different ASIPS/core types. However in case of a many-core system a system generally consists of cores of the same type.



Threads	Value	Writes	Reads
Register#1			
Thread 0	#00000000	0	10
Thread 1	#00000000	0	10
Thread 2	#00000000	0	10
Thread 3	#00000000	0	11
Register#2			
Thread 0	#AF002800	1	23
Thread 1	#AF002800	1	23
Thread 2	#AF002800	1	23
Thread 3	#AF002800	1	23

Figure 5.3: Mock-up of the modified Register View.

Example of displaying values of registers for multiple threads simultaneously.

Chapter 6

Implementation

The two of the modifications designed in the chapter 5 have been implemented:

- Visualizer View modification
- Debug View modification

The existing plug-ins for CDT and Codasip IDE were modified or extended. Also the new plug-in `com.codasip.debug.visualizer.ui` was created. All these plug-ins cover the functionality of modifications designed for debugging environment.

6.1 Visualizer View modification

The following list describes the most important classes of the Codasip Visualizer View implementation:

- `CodasipMulticoreVisualizer`

  It provides the core functionality of the view. It instantiates the important classes, creates and populates the *view model* with *data model* contexts [2].

- `CodasipMulticoreVisualizerCanvas`

  It instantiates classes that represent the system components (levels, ASIPs) and draws them.

- `CodasipDSFDebugModel`

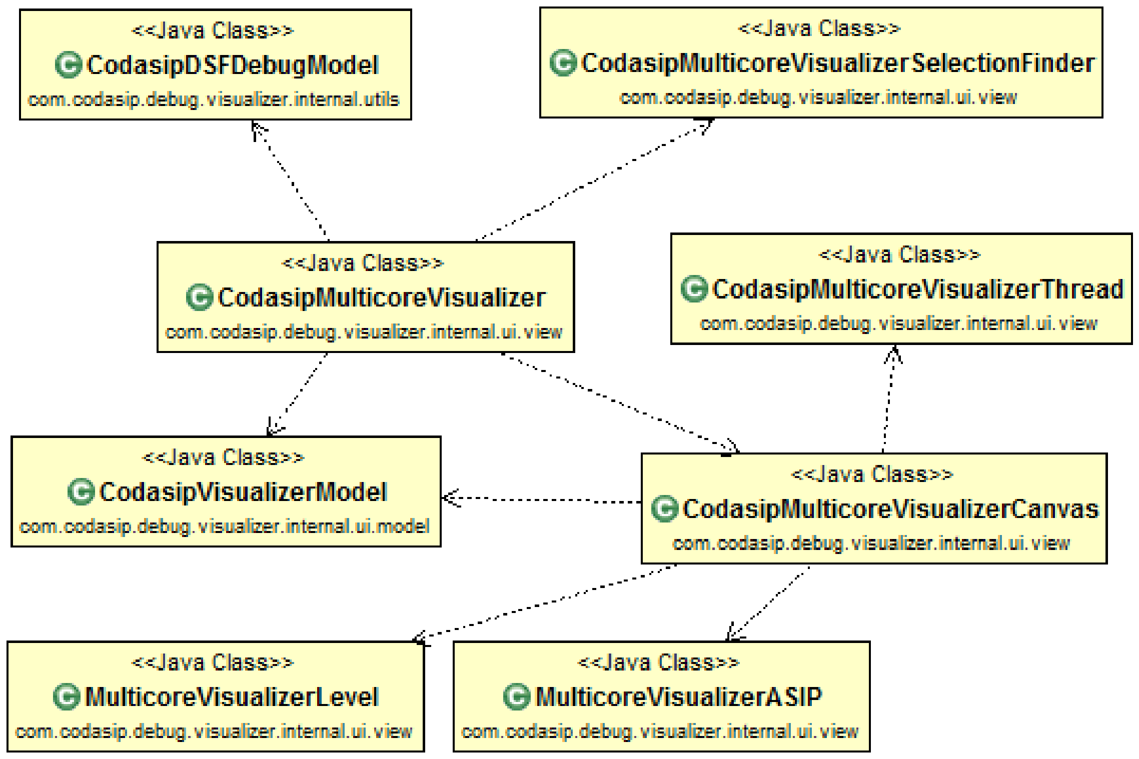
  It uses the DSF services to get the data about the hardware hierarchy and provides this data to other classes of Codasip Visualizer View.

- `CodasipMulticoreVisualizerModel`

  This class represents the view model and holds the data about the components that are drawn in the view.

- `CodasipMulticoreVisualizerLevel`

  It represents the level in the Codasip hardware hierarchy and defines how the level is drawn.



<<Java Class>>
CodasipDSFDebugModel
com.codasip.debug.visualizer.internal.utils
<<Java Class>>
CodasipMulticoreVisualizerSelectionFinder
com.codasip.debug.visualizer.internal.ui.view
<<Java Class>>
CodasipMulticoreVisualizer
com.codasip.debug.visualizer.internal.ui.view
<<Java Class>>
CodasipMulticoreVisualizerThread
com.codasip.debug.visualizer.internal.ui.view
<<Java Class>>
CodasipVisualizerModel
com.codasip.debug.visualizer.internal.ui.model
<<Java Class>>
CodasipMulticoreVisualizerCanvas
com.codasip.debug.visualizer.internal.ui.view
<<Java Class>>
MulticoreVisualizerLevel
com.codasip.debug.visualizer.internal.ui.view
<<Java Class>>
MulticoreVisualizerASIP
com.codasip.debug.visualizer.internal.ui.view

Figure 6.1: Class diagram of the most important classes of Codasip Visualizer View.

- `CodasipMulticoreVisualizerASIP`

  It represents the ASIP in the Codasip hardware hierarchy and defines how the ASIP is drawn.

- `CodasipMulticoreVisualizerThread`

  It represents the thread in the Codasip hardware hierarchy and defines how the thread is drawn.

The most problematic task of the implementation is the visualization of the hardware hierarchy. The basic implementation of Visualizer View can draw just a flat model of processors with multiple cores. However the levels displayed in the view are drawn on top of their parents. It has to be clearly visible what relations are between them and what content do they contain. Also the size and the location of each of the components has to be properly calculated to avoid overlapping.

After the data are retrieved from a backend, the hierarchical tree of levels is created. Hash maps are used to map the relations between levels. The class that represents the level also holds the information about ASIPs that are located in it. At first the size of ASIPs is calculated according to the count of ASIPs, the count of levels and their margins. The drawn ASIPs have to fit to the size of a canvas including all the spaces between components. Then the size and the location of each level is computed based on the number of the ASIPs located in it, number of the child levels and their ASIPs. The hierarchical tree has to be traversed multiple times to make all these calculations.

There are occasions when the displayed content is too excessive. Two features have been implemented to help with it. The basic implementation of the Visualizer View provides option to filter the displayed content. It allows to display only chosen processors and cores. In our case it has to display selected leves and ASIPs. Therefore the parts of the hierarchical tree that contain selected components have to be found and drawn. Some of the levels can be bypassed. Also if there are levels that contain no ASIPs, another levels with no ASIPs or no levels, they are considered empty and they are not visualized. This is the second feature. The levels with no content unnecessarily consume the space. If they are not displayed, other content can be visualized bigger.

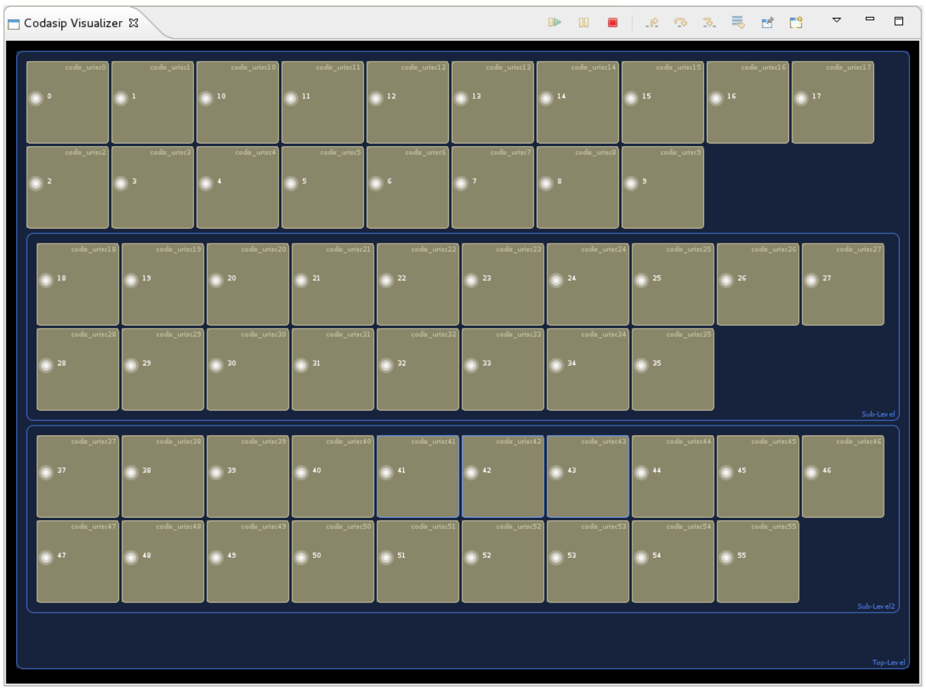


Figure 6.2: Screenshot of the Codasip Visualizer View.

Example of displaying the three levels containing ASIPs.

6.2 Debug View modification

The following list of classes describes the most important classes of the modification:

- `LevelVMNode`

  The class represents the level node in Debug View.

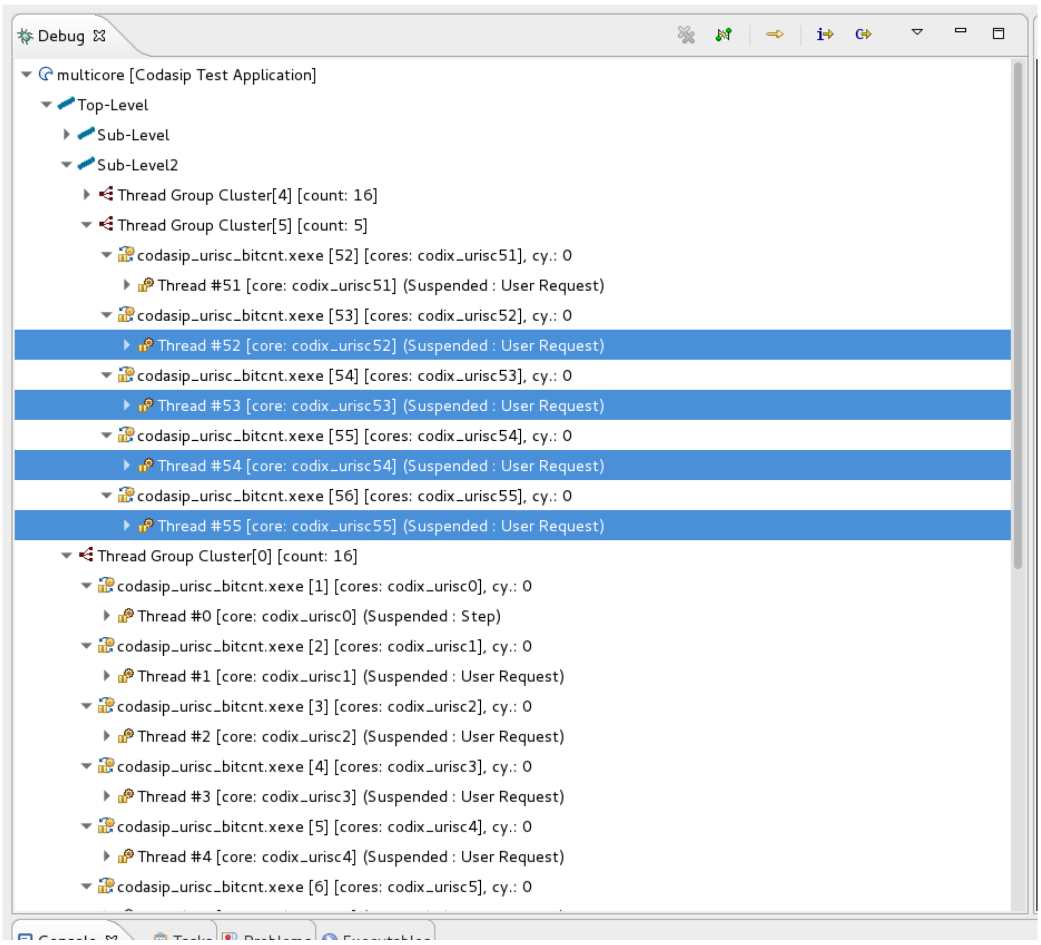
- `ThreadGroupClusterVMNode`

  It represents the thread group cluster node in Debug View which groups the certain amount of thread group nodes.

The implementation of Debug View is located in the Eclipse plug-ins. However the classes mentioned above are located in the Codasip plug-ins that extend Debug View.

Both of the view model nodes use the DSF services to get the information about data model contexts and their properties. They create the view model contexts according to them and populate them with information. The view model contexts of thread group cluster node are created only when there are enough of thread group nodes displayed in the same level. Both of nodes also have to implement the behaviour that defines how they react on certain events. This behaviour specifies when the state or content has to be updated.

When a debug session starts all the nodes in Debug View are expanded. It is the default behaviour given by the implementation of the nodes in the Eclipse plug-ins. This does not happen after the recursive node is added (can have itself as a child). Despite the new nodes implement this behaviour, not all of the nodes are expanded. This is caused by the bug that is documented on Eclipse website[1].


Debug ⊠

```
multicore [Codasip Test Application]
  Top-Level
    Sub-Level
    Sub-Level2
      Thread Group Cluster[4] [count: 16]
      Thread Group Cluster[5] [count: 5]
        codasip_urisc_bitcnt.xexe [52] [cores: codix_urisc51], cy.: 0
          Thread #51 [core: codix_urisc51] (Suspended : User Request)
        codasip_urisc_bitcnt.xexe [53] [cores: codix_urisc52], cy.: 0
          Thread #52 [core: codix_urisc52] (Suspended : User Request)
        codasip_urisc_bitcnt.xexe [54] [cores: codix_urisc53], cy.: 0
          Thread #53 [core: codix_urisc53] (Suspended : User Request)
        codasip_urisc_bitcnt.xexe [55] [cores: codix_urisc54], cy.: 0
          Thread #54 [core: codix_urisc54] (Suspended : User Request)
        codasip_urisc_bitcnt.xexe [56] [cores: codix_urisc55], cy.: 0
          Thread #55 [core: codix_urisc55] (Suspended : User Request)
    Thread Group Cluster[0] [count: 16]
      codasip_urisc_bitcnt.xexe [1] [cores: codix_urisc0], cy.: 0
        Thread #0 [core: codix_urisc0] (Suspended : Step)
      codasip_urisc_bitcnt.xexe [2] [cores: codix_urisc1], cy.: 0
        Thread #1 [core: codix_urisc1] (Suspended : User Request)
      codasip_urisc_bitcnt.xexe [3] [cores: codix_urisc2], cy.: 0
        Thread #2 [core: codix_urisc2] (Suspended : User Request)
      codasip_urisc_bitcnt.xexe [4] [cores: codix_urisc3], cy.: 0
        Thread #3 [core: codix_urisc3] (Suspended : User Request)
      codasip_urisc_bitcnt.xexe [5] [cores: codix_urisc4], cy.: 0
        Thread #4 [core: codix_urisc4] (Suspended : User Request)
      codasip_urisc_bitcnt.xexe [6] [cores: codix_urisc5], cy.: 0
```

Figure 6.3: Screenshot of Codasip Visualizer View.

Example of displaying three levels containing ASIPs and one empty level.

[1]https://bugs.eclipse.org/bugs/show_bug.cgi?id=306868

6.3 Implementation of hardware service

DSF services are used to obtain information about the data model and propagate this information to view model. The special service to obtain data about Codasip hardware hierarchy was created.

The service represented by the `CodasipHardware` class creates the GDB/MI command to retrieve data about hardware hierarchy of debugged system. Retrieved data are parsed from the string form and the hierarchical model is created. The reference on the top level node of this hierarchy is returned to the view model.

6.4 Testing

Complete validation of the correct behaviour of implemented features is not possible. The features are designed and implemented for Codasip IDE 6 that has not been released at the time which this thesis is written. Therefore some functionality is missing or is simulated.

There are 2 tests that were used to test the implemented features:

- time measurement of content drawing
- usability testing

6.4.1 Time measurement of content drawing

This test measures the time which is needed to draw all components in Codasip Visualizer View. The measured time is the time needed to execute method that is responsible for drawing a content of the view. The results expose how big is the impact of count of displayed objects on delays in the user interface.

	ASIP count							
Level count	100	300	500	750	1000	1500	2000	3000
1	0,073	0,114	0,142	0,209	0,412	0,318	0,468	0,817
3	0,080	0,112	0,156	0,210	0,447	0,310	0,498	0,836
9	0,105	0,148	0,186	0,209	0,257	0,385	0,481	0,874

Table 6.1: The time in milliseconds needed to draw content of Codasip Visualizer View.

Table 6.1 shows the results of the test. Various amounts of ASIPs and levels were used as the input data as it is noted in the table. While count of ASIPs can vary from few dozens to several thousands in many-core systems, it is not expected that the user will use more than few dozens of levels. The time of visualizing was calculated as average value of 20 values of the time needed to recalculate the size of graphical objects and redraw them. It was measured as a difference of times at the start and in the end of the method responsible for recalculation and redrawing. The results showed that redrawing of the view's content does not create long delays (all values are under 1 millisecond) in comparison with the delays caused by retrieving the data from debugger. These delays could not be measured due to missing functionality of other tools.

6.4.2 Usability test

The five people from the Codasip employees were chosen to test the usability of the new debug environment. The environment and the basic commands were introduced to them. After that, several debug sessions were executed with prepared input data. They were requested to do some actions in each session. Finally each participating employee has been asked a few questions. The goal of the test was to find out the easy use and the lucidity of the environment.

Modified Debug View and Codasip Visualizer View were the main objects of the test. The following actions were introduced and performed in each debug session:

- start and stop debug session
- do few steps of debug session
- select multiple targets (threads, ASIPs)
- filter multiple targets (only Codasip Visualizer View)
- expand and contract nodes (only Debug View)

The systems that were debugged were not the same as the environment presented. They did not consist of levels and, in some cases, they did not contain the amount of ASIP that were displayed. The systems with up to 56 ASIPs can be debugged using the debugger. Others with higher number of ASIPs were artificially created in GUI code. ASIPs are randomly distributed to all levels of the system. The following systems were debugged in the test:

- 3 levels and 10 ASIPs
- 3 levels and 56 ASIPs
- 9 levels and 500 ASIPs (not supported by debugger)
- 9 levels and 1000 ASIPs (not supported by debugger)
- 9 levels and 2000 ASIPs (not supported by debugger)

In the end of the testing the participating employees answered the following questions:

- How easy it was to perform all asked actions?
- Was new debug environment comfort and fast?
- What were the pros and cons of the environment?
- What are your suggestions to improve the environment?

Some of the shortcomings have been found out from the answers of the testing users. The most of complaints have been aimed to the long launch of debug sessions and unreadable ASIPs IDs when the count of ASIPs was higher than 500. All participants complained about the low value of frames per second when the system with 2000 ASIPs was debugged. 2 of 5 participants did not like the colours used in Codasip Visualizer View. The readability of a hardware hierarchy and the filter feature of Codasip Visualizer View were rated positively.

6.4.3 Future improvements

The tests exposed the limitations of the implemented features. One of the biggest limitations is visualizing more ASIPs than 2 or 3 thousands. In this situation ASIPs have too small size even if the view is resized to the maximum. There are 2 improvements that can help to solve this problem. The first is the ability to expand or contract levels in Codasip Visualizer View. Levels with observed ASIPs could be expanded and other could be contracted to save some space for other objects. Another approach is Codasip Visualizer View with the zoom ability and with scrollbars. The user could zoom in and display the part of the content he would want.

Some of users could desire different colour and shapes of objects displayed in Codasip Visualizer View. Colour palette and list of shapes for each objects would make the visualization more flexible. User could adapt the view to his requirements.

Chapter 7

Conclusion

The goal of this thesis was to design and implement the debug environment for many-core systems. The thesis described the problems with debugging the many-core systems using the integrated development environments. The modifications of IDE and debugger were designed and some of them implemented and tested.

The tests showed that implemented features solve some of the problems with debugging the many-core systems. There are the cases when these features do not solve the problems or only partially. The modifications of Debug View and Visualizer View offer different overview of the hardware hierarchy. They help to locate and identify ASIP in the debugged system. They also offer more convenient observation and control of the specified part of the debugged system. But there are limitations of this environment too. Retrieving data and internal reconstruction of hardware hierarchy make launching of a debug session longer. When there are displayed more than 1 thousand of ASIPs, the visualization loose clarity.

It is possible to improve this environment by implementing other modifications in the future. Implementation of the modification of GDB/MI protocol designed in the section 5.3.1 could solve the long debug session launch. The content of the modfified Visualizer View can be more lucid by adding an option to zoom and scroll the content or expand and contract the part of the content.

The contribution of this thesis consists of design and implementation improvements of debugging environment that will be used in Codasip IDE. The features that were implemented are just prototypes and will be modified in the future according to requirements of R&D and customers of Codasip.

Bibliography

[1] Eric Clayberg and Dan Rubel. *eclipse Plug-ins*. ISBN 978-0-321-55346-1. Pearson Education, 2008.

[2] IBM Corporation. *Eclipse documentation*. 2006. URL: {http://help.eclipse.org/mars/index.jsp} (visited on 04/17/2016).

[3] *Eclipsepedia*. URL: {https://wiki.eclipse.org} (visited on 04/17/2016).

[4] IPextreme. *Infineon Multi-Core Debug Solution*. 2008. URL: {https://www.ip-extreme.com/downloads/MCDS_brochure_080128.pdf} (visited on 03/21/2016).

[5] Kurt Keutzer, Sharad Malik, and A Richard Newton. "From ASIC to ASIP: the next design discontinuity". In: *Computer Design: VLSI in Computers and Processors, 2002. Proceedings. 2002 IEEE International Conference on*. IEEE. 2002, pp. 84–90.

[6] Georgios Kornaros. *Multi-core embedded systems*. ISBN 978-1-4398-1161-0. CRC Press, 2010.

[7] Nvidia. *CUDA-GDB*. 2015. URL: {http://docs.nvidia.com/cuda/pdf/cuda-gdb.pdf} (visited on 04/17/2016).

[8] Jim des Rivières and John Wiegand. "Eclipse: A platform for integrating development tools". In: *IBM Systems Journal* 43.2 (2004), p. 371.

[9] Richard Stallman, Roland Pesch, Stan Shebs, et al. "Debugging with GDB". In: *Free Software Foundation* 51 (2002), pp. 02110–1301.

[10] Harold S Stone and John Cocke. "Computer architecture in the 1990s". In: *Computer* 24.9 (1991), pp. 30–38.

[11] Michael Wolfe. *Understanding the CUDA Data Parallel Threading Model A Primer*. 2010. URL: {https://www.pgroup.com/lit/articles/insider/v2n1a5.htm} (visited on 03/21/2016).

Appendices

List of Appendices

A CD content **37**

B Screenshots of the modified Visualizer View **38**

Appendix A

CD content

The content of CD consists of the following files:

- *src* – a directory that contains plug-ins for Eclipse and source code files
- *ide* – a directory with executable Codasip IDE
- *readme.txt* – a basic description of the application and execution instructions
- *tex_src* – a directory with source codes of technical report
- *report.pdf* – the pdf file of technical report
- *user_guide.pdf* – instructions how to use the implemented features

Appendix B

Screenshots of the modified Visualizer View

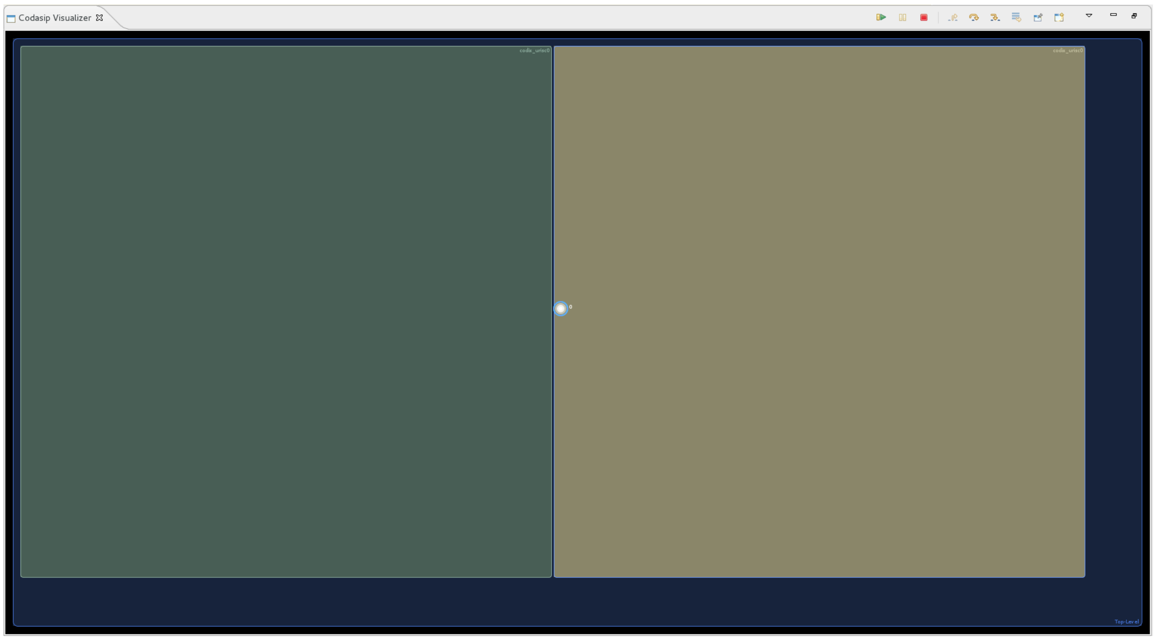


Figure B.1: 1 level 2 ASIPs

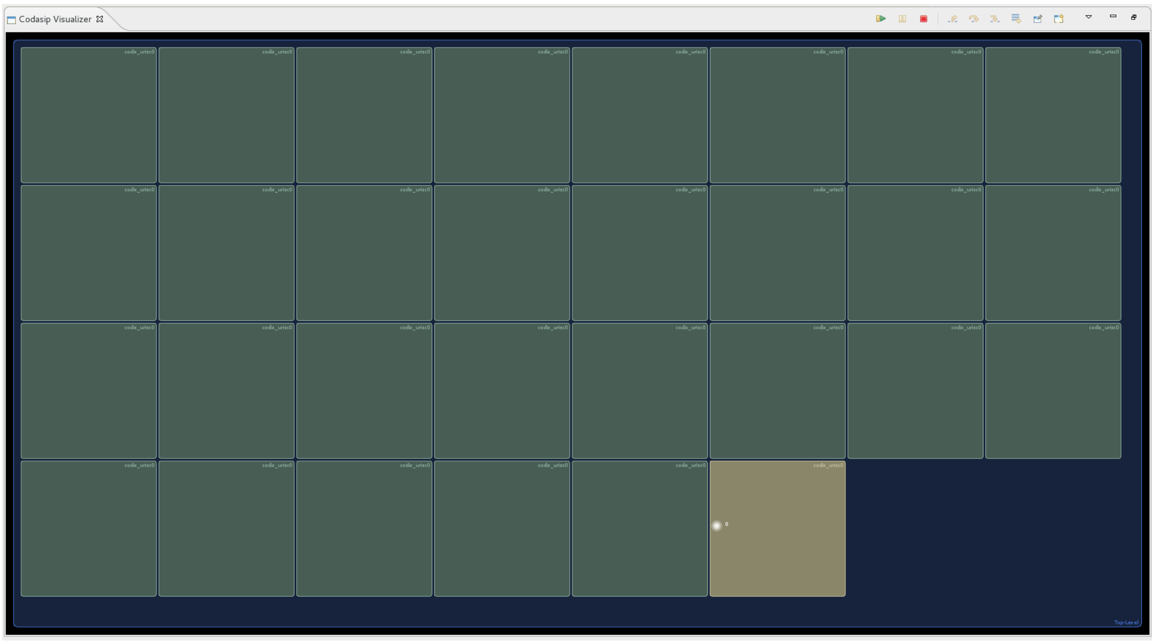


Figure B.2: 1 level 30 ASIPs

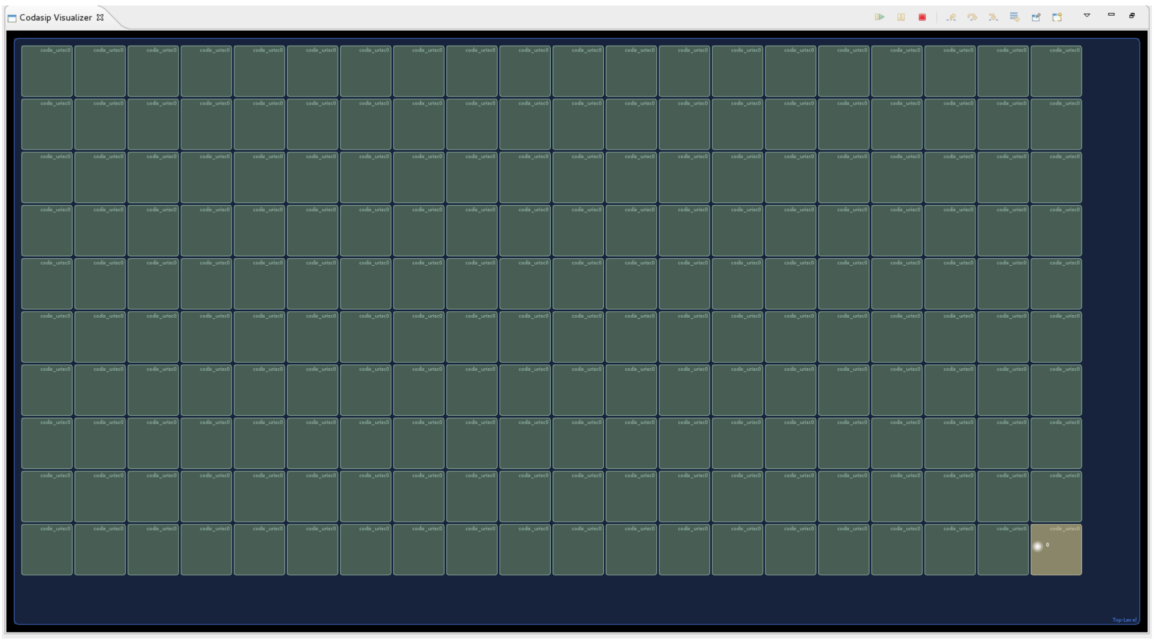


Figure B.3: 1 level 200 ASIPs

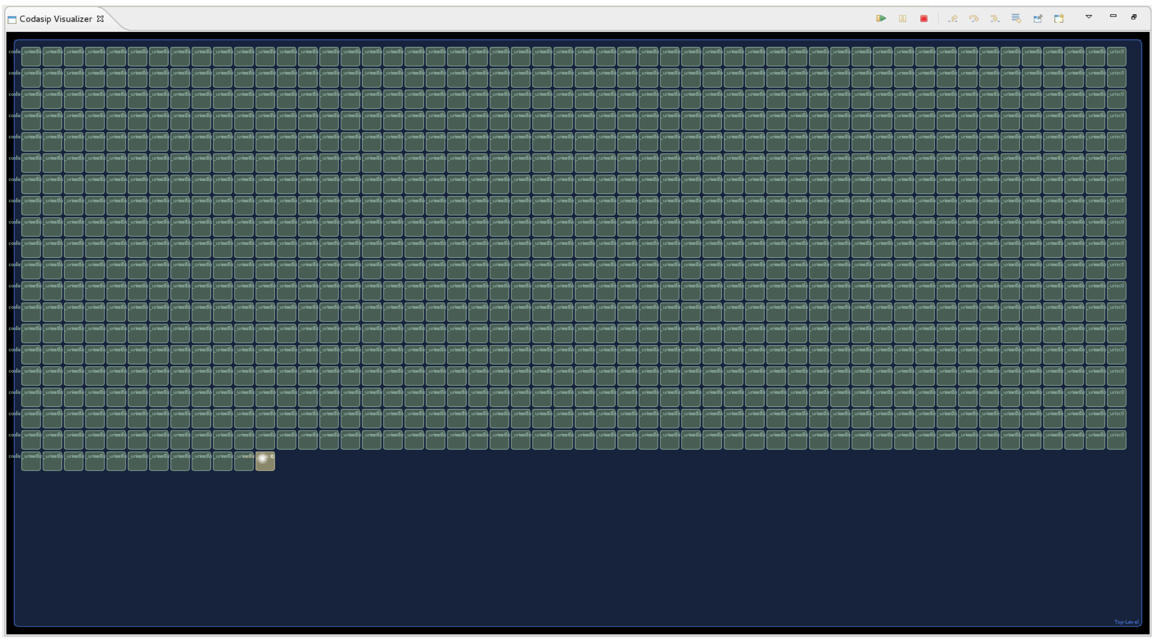


Figure B.4: 1 level 1000 ASIPs

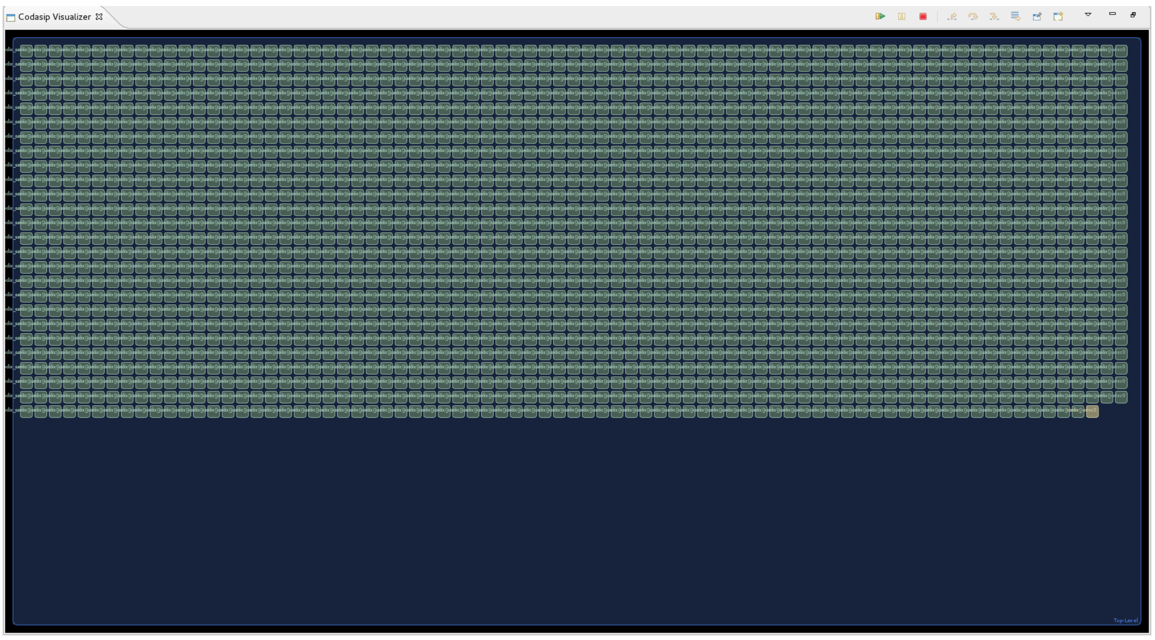


Figure B.5: 1 level 2000 ASIPs

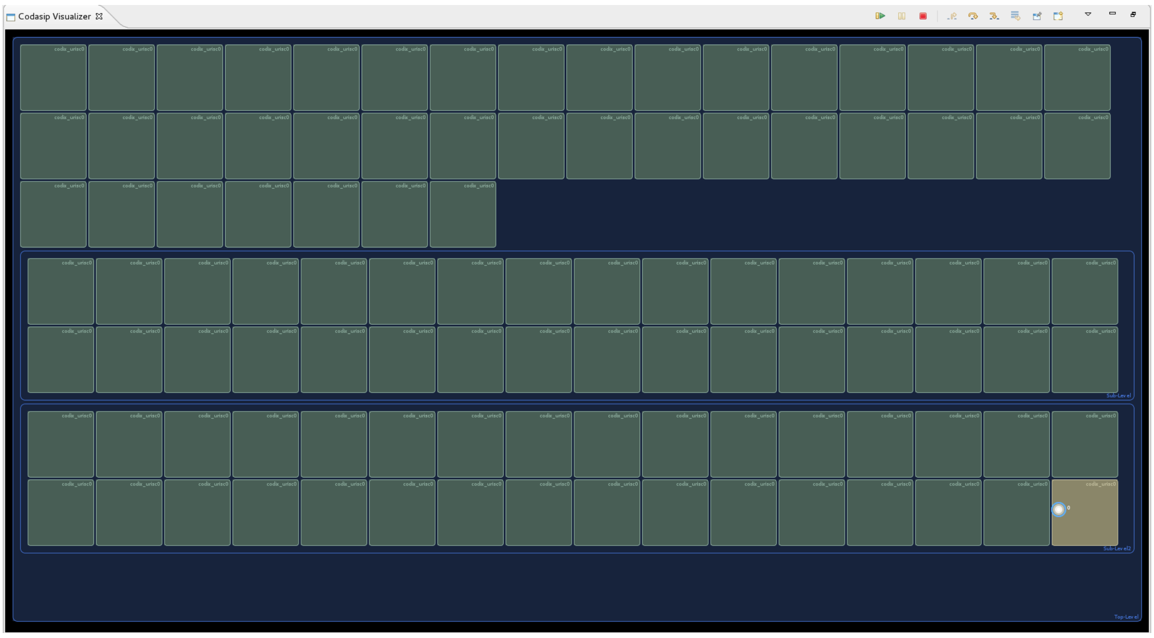


Figure B.6: 3 levels 100 ASIPs

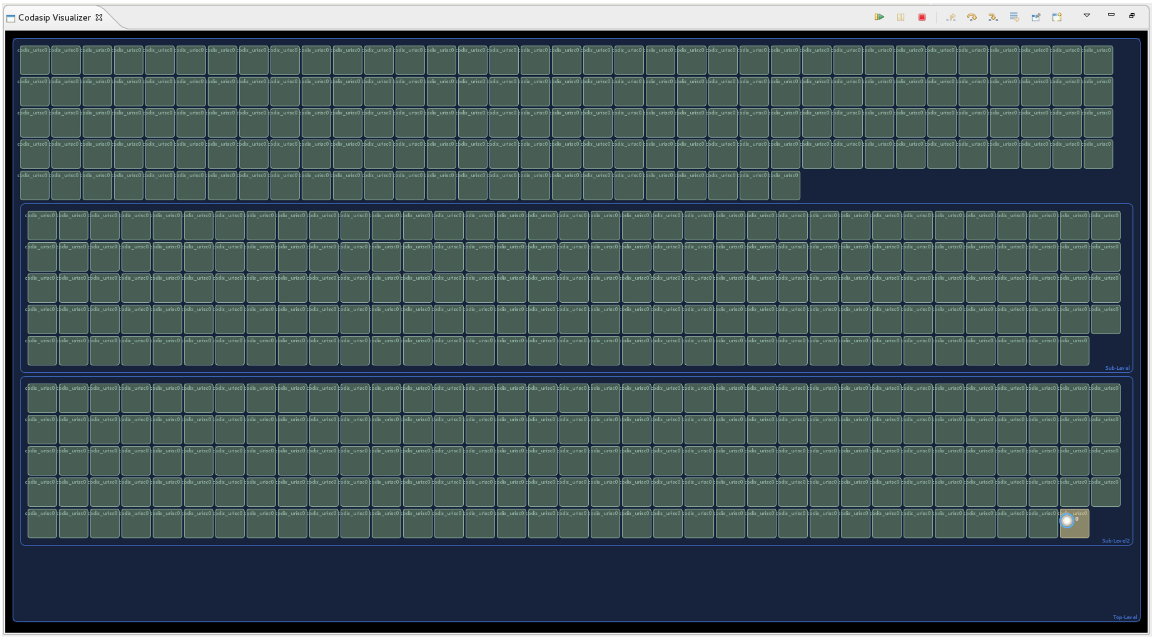


Figure B.7: 3 levels 500 ASIPs

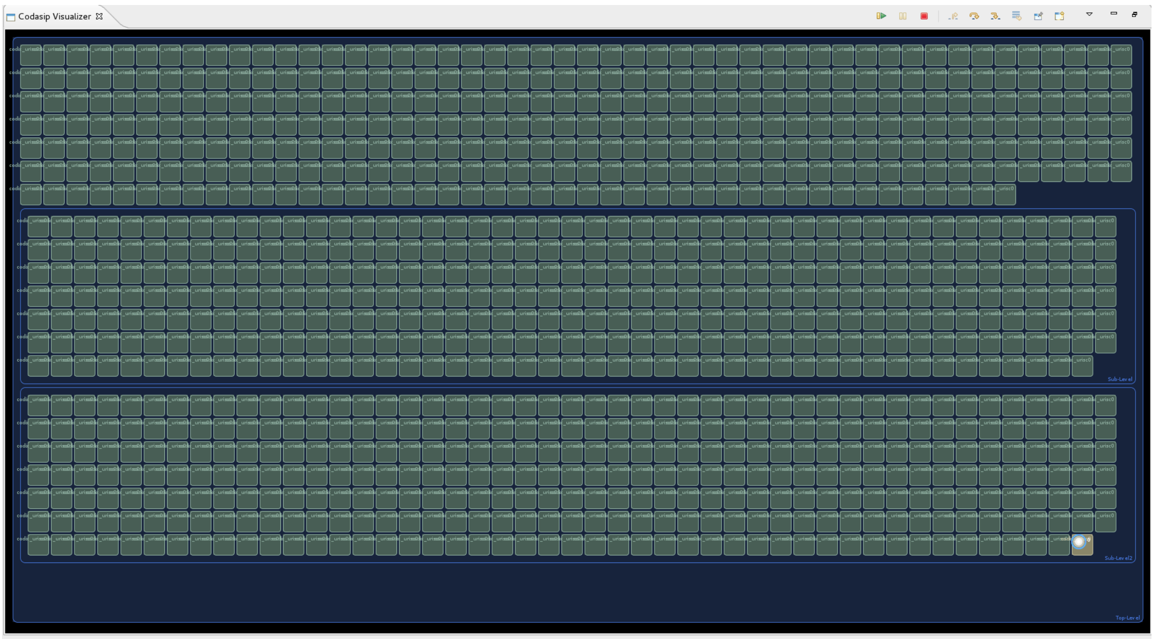


Figure B.8: 3 levels 1000 ASIPs

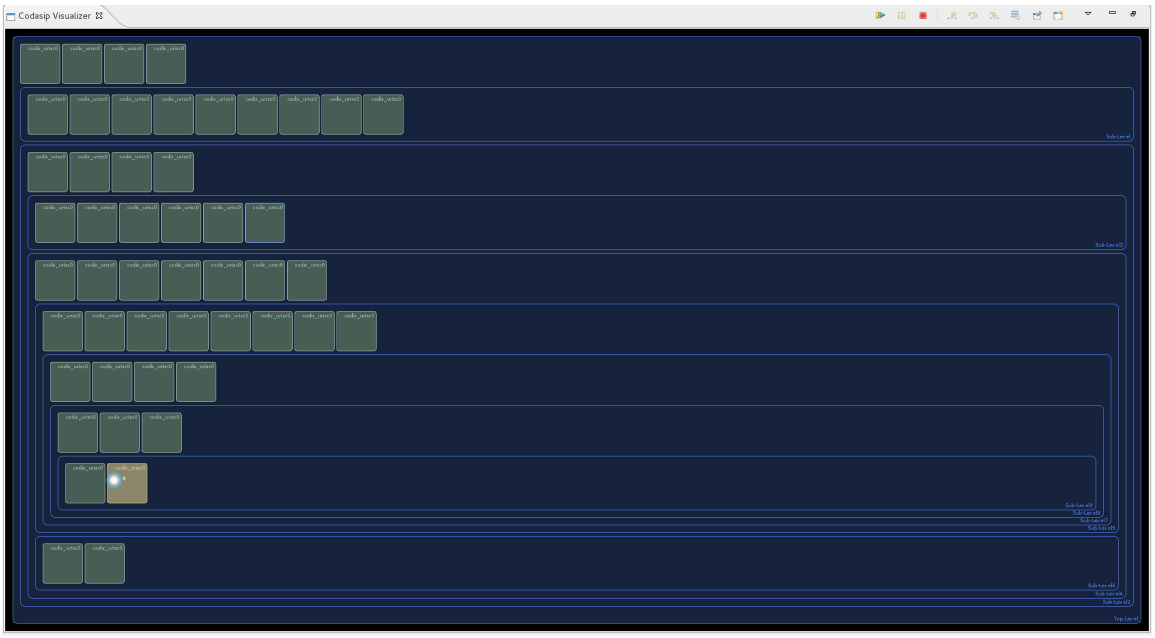


Figure B.9: 10 levels 50 ASIPs

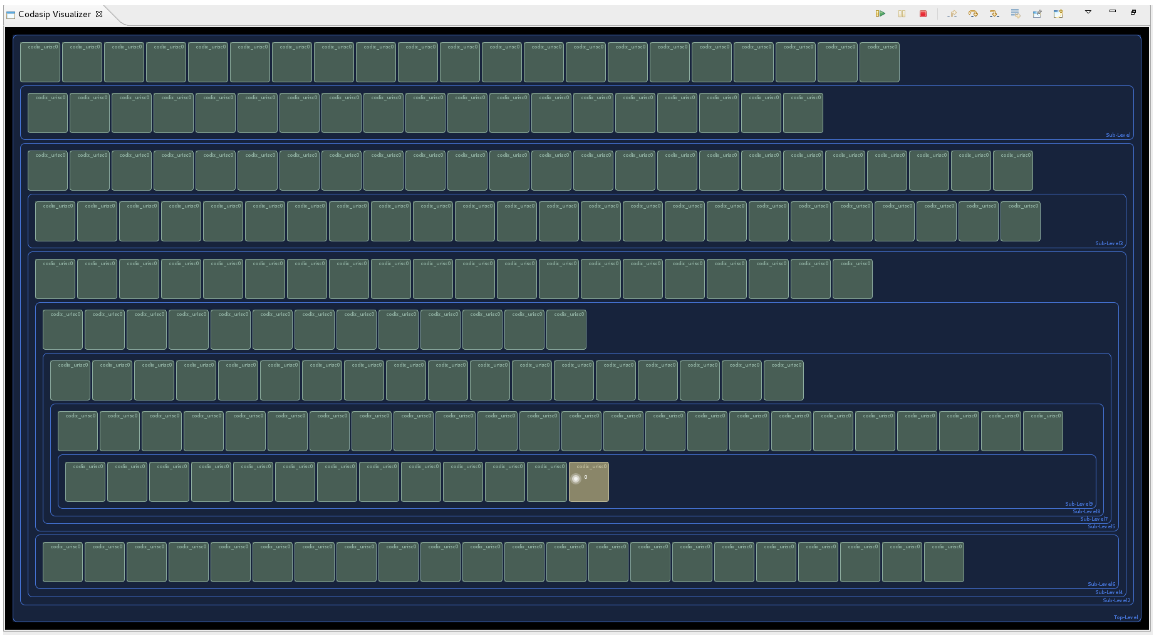


Figure B.10: 10 levels 200 ASIPs

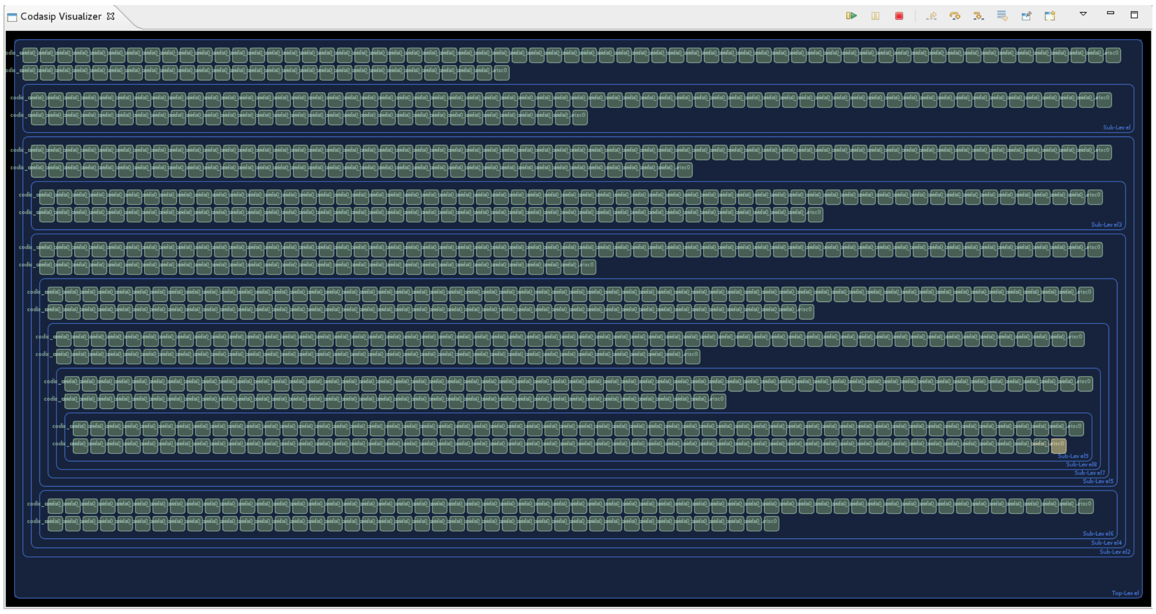


Figure B.11: 10 levels 1000 ASIPs