Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS

Tomáš Martinec

Interactive Debugger for MSIM

Department of Distributed and Dependable Systems

2011

Contents

Název práce: Interaktivní Debugger pro MSIM
Autor: Tomáš Martinec
Katedra: Katedra distribuovaných a spolehlivých systémů
Vedoucí bakalářské práce: Mgr. Martin Děcký
E-mail vedoucího práce: martin.decky@d3s.mff.cuni.cz

Abstrakt: Cílem této práce je rozšíření ladících schopností MSIMu pomocí napojení na vývojové prostředí Eclipse.

MSIM je simulátor počítače založený na procesoru MIPS a jednoduchém hardwarovém modelu. Je zaměřen převážně na výuku principů operačních systémů. V současné době MSIM umožňuje ladit emulovaný kód díky implementaci síťového rozhraní GNU Debuggeru (GDB) a pomocí několika interních ladících příkazů.

Implementace prezentovaná v této práci spojuje MSIM a ladící prostředí Eclipse IDE pomocí GDB rozhraní. Umožňuje tak uživatelsky přívětivé ladění emulovaného kódu na úrovni zdrojových souborů i assembleru. Dále jsou v práci diskutovány pokročilé ladící techniky jako sledování userspace vláken, konstrukce call tree, zpětné krokování a další.

Klíčová slova: debugger, GNU debugger, GDB, Eclipse debugger, simulátor počítače, MSIM, MIPS processor, operační systém

Title: Interactive Debugger for MSIM
Author: Tomáš Martinec
Department: Department of Distributed and Dependable Systems
Supervisor: Mgr. Martin Děcký
Supervisor's e-mail address: martin.decky@d3s.mff.cuni.cz

Abstract: The goal of this thesis is to extend the debugging possibilities of MSIM by connecting it to the Eclipse IDE.

MSIM (*M*IPS *Sim*ulator) is a machine simulator based on a MIPS processor and a simple hardware model. The simulator is primarily used for education of operating systems. At the beginning of this work MSIM provided basic features for debugging the code that runs inside the virtual machine. The basic features are accomplished by implementing GNU Debugger (GDB) remote connection interface and also by several integrated debugging commands.

The work presented in this thesis connects the Eclipse IDE debugging front-end to MSIM via the GDB remote connection interface. Necessary means to provide comfortable experience of both source-level and assembly-level debugging are implemented. Additionally, a discussion about advanced debugging features such as: user space code debugging with a thread scope; call tree construction; reverse execution; and other techniques is presented in the thesis.

Keywords: debugger, GNU debugger, GDB, Eclipse debugger, computer simulator, MSIM, MIPS processor, operating system

Introduction

Implementing an operating system is very difficult and learning it is often even more difficult. For a system programmer one of the hardest kind of problems is solving hardware related errors. Reproducing these error is often tedious and some knowledge and skills from electrical engineering might be also required to diagnose them. Development of an operating system that runs on real hardware would be an unmanageable task for many students, so MSIM simulator was created for education purposes. MSIM simulates a computer machine, which can be composed of MIPS processors, the main memory and several devices. One of the most important advantage of using the simulator is that errors of a simulated program can be almost always reproduced by running the simulation from the beginning. The need for special hardware and more complicated booting is also eliminated by the simulator usage.

MSIM also provides some debugging functionality such as: instruction stepping; instruction breakpoints; memory breakpoints; a trace log of an execution; and a memory dump. Furthermore, the utility *objdump* can be used for obtaining the layout of the program code in the main memory. However, finding errors in the MSIM environment is still much more difficult than errors that an application programmer solves typically; and students of the operating systems course at *MFF UK* spend significant amount of time by debugging. The high difficulty of debugging an operating system for MSIM is probably because of: the usual coding in low level C language; the missing operating system (which usually detects some erroneous conditions in user space); the usage of non-transparent mechanisms and algorithms; and unavailable debugger. Sometimes the system programmer also needs to write in assembler language and an error in such a code can be extremely hard to be found.

This thesis is concerned with an implementation of a GUI based debugger that provides comfortable debugging of MSIM applications. Some mechanisms, which are used in debuggers, are also described. In addition some advanced debugging techniques that are not commonly implemented in debuggers are proposed.

The initial chapters of this text cover terms and introduce debuggers from the point of view of this work. Then the usefulness of the debugger for MSIM is depicted in the chapter *Debugging without a debugger*. The chapters *Design of the MSIM debugger* and *Implementation and know-how* are concerned about realizing the debugger. The last section discusses some unusual or advanced debugging methods. The attachments contain: instructions how to setup development environment from scratch; the user manual; and a discussion about picked difficulties during development of large projects.

1. Terminology

Several specific or not well-known terms are used in this thesis. They are explained in this chapter. Terminology that is closely related to debuggers is introduced in the chapter 2.

1.1 Generally used terms

Generally used, but perhaps, not well-known terms are:

- **Debugger**. Programmers often call logical errors in programs as bugs. The derived word debugger denotes a special piece of software that helps to search those bugs. Debuggers enable the programmer at least to stop the program and inspect internal state of the program.

- **Debuggee**. Not a very common word even in programming. It denotes a program that is being analyzed by a debugger. In the scope of this thesis, a debuggee is typically an operating system that is executed in MSIM.

- **Intrusive debugging**. The intrusiveness is quite a rare word that is often used by developers of debuggers. Intrusive debugging means a debugging that affects the debuggee a lot. Intrusive debugging is an undesired property of debuggers, because it changes behaviour of debuggees during debugging in an essential way.

- **Race condition**. An error in synchronization of parallelly executed code.

- **Translation lookaside buffer (TLB)**. A TLB serves as a cache for virtual-to-physical addresses translation. It caches information from page tables and it is integrated in a processor for fast access. During an address translation the processor looks into its TLB at first. If the needed information is not in the TLB, the processor searches page tables or asks the operating system to fill the TLB with the needed information. Details of the TLB mechanism depends on the type of the processor.

- **Virtual machine**, **Computer simulator**. The term virtual machine designates two things, which are further called more specifically as a **process virtual machine** and a **system virtual machine**.

  A process virtual machine is a program that interprets code of another program and provides necessary API for the interpreted program. An examples of a process virtual machine is the *java* program. This kind of virtual machines is not referred in this work.

  A system virtual machines is a program that emulates work of a computer. Usually it interprets code of an operating system. Examples of system virtual machines are the programs: *QEMU, WMware Workstation*, or *MSIM*.

  In this work, the term **computer simulator** is a synonym for the term system virtual machine.

1.2   Specific terms for this work

Terms that are specific only for this work are:

- **MSIM application**. A program that is executed in MSIM.

- **MSIM debugger**. Software that enables the programmer to debug an MSIM application in the Eclipse IDE. The main goal of this work is to implement it.

2. Overview of debuggers

The purpose of the debuggers is to help the programmer to diagnose the behaviour of a program that is being analyzed. Such a program is called a debuggee. Debuggers achieve its purpose by controlling execution of the debuggee (stopping and resuming the execution, executing the next line of code, stopping after the execution reaches a specific location, ...) and showing the state of the debuggee (values of variables, call stack, ...).

A more general text about debuggers can be found in the initial chapters of Rosenberg's book [1]. This book also presents principles of debugging and one of them - the Heiseinberg principle adapted for debugging - is referred in this thesis. The Heiseinberg principle adapted for debugging is called just the Heiseinberg principle in the scope of this work. The principle states that the observations from the debugger might be misleading if the state of the debuggee is being affected by the debugger. Usually it is impossible not to affect the debuggee at all. So the debugger should be designed to minimalize its intrusiveness.

Let us additionally note two mechanisms in debuggers that are a good illustration how debuggers work - managing breakpoints and stack unwinding. Breakpoint mechanisms are described in [1, Ch. 6] and stack unwinding is described in [1, Ch. 7].

Debuggers are categorized in [1] by several points of view. Two points of view are considered here:

- debuggers according to type of debugging session,

- instruction-level versus source-level debuggers.

2.1 Debuggers according to type of debugging session

1. Userspace debuggers

   Userspace debuggers are the most commonly-used debuggers. A debuggee for this kind of debugger is typically executed in the environment of a general-purpose operating system. These kinds of operating systems use the concept of virtual memory, which among the other things does not allow a process to read the memory of another process. Therefore, userspace debuggers must use a special way how to access the memory of the debuggee. Commonly-used mechanism is based on special functions for debugging that are provided by the API of the operating system. The debugger is executed as a standalone process and controls the debuggee via the API functions.

   An overview of the debugging API can be found in [1, Ch. 4] for several operating systems.

2. Kernel debuggers

There are differences between debugging a userspace application and debugging a kernel of an operating system. It might not be very suitable for the debugger to be a process of the debugged operating system, because the debugger would influence events occurring in the operating system. For example, stopping the operating system might cause troubles to the correct function of the debugger (such as stopped interaction with the user, disabled access to the filesystem, etc.). One approach to overcome this issue is that the debugger hooks into the kernel and implements all the needed functionalities in its own way. Therefore, such a debugger is independent on the services of the operating system. Examples of these debuggers are *SyserDebugger*, *SoftICE* or *BugChecker*. Another approach is running the debugger on a different computer that is connected to the computer where the debugged kernel is running. *KGDB* debugger can be referred as an example of these debuggers. Common ways of the connection between the computers are via the serial port or via the ethernet where UDP or TCP protocol are used. There must be support for such a session in the debugged kernel. With respect to the Heisenberg principle it is beneficial that the kernel module for the communication affects the other parts of kernel as little as possible. For example, avoiding virtual memory usage might be an appreciated feature of the module.

3. Debuggers for programmable embedded devices

Embedded devices often have a realtime operating system or no operating system at all. Writing a program for such a device is like writing a module for an operating system. The debuggers in this area usually require special hardware for connection with the examined device. These debuggers should not be automatically supposed to do all the things that the debuggers for userspace applications do. Also the amount of data transfered between the device and the debugger may be so large that transferring them via the serial port would make the debugging very slow.

Two examples of the such shortcomings are mentioned. At first debugging an *atmega64* device with *AVR Dragon debugger* has been experienced not showing values of local variables. Secondly debugging Siemens *TC65* device via the serial port is more efficient to be done by printing debug messages than with the help of a slow debugger.

4. Debuggers for a system virtual machine

This kind of debuggers can be used for diagnosing an operating system or firmware of an embedded device. However, simulations are usually used in special situations (e.g. in education), so debuggers for system virtual machines are not commonly encountered. The debugger can be implemented directly in the simulator or can be a standalone process, which communicates with the simulator. Debugging simulated programs has significant advantages. The debugging is very unintrusive and the simulator can provide debugging information that are not usually available.

A complex debugger for a simulator is also able to process the structures of the debugged operating system. For example that allows the debugger to recognize userspace threads and processes and therefore work as a userspace

debugger. Such a debugger would be a strong tool for analyzing events across the whole operating system.

2.2 Instruction-level versus source-level debuggers

The adjectives source-level and instruction-level are used in this thesis. This section explains them.

A program code written in a compiled programming language is transformed into a binary code during the compilation. The binary code can be then executed by the machine. Usually the programmer wants to see the code in the form that he has written it in. However, there are domains where the form of the binary code is essential for the programmer (for example, debugging an interrupt system).

The debuggers that shows the original form of the code are called source-level debuggers. On the other hand the debuggers that shows the binary form are called instruction-level debuggers or sometimes assembly-level debuggers. The debuggers can of course show both the forms.

Typically the userspace debuggers shows the code in the original high level language and optionally in the instruction-level form. That is because in very most of the situations there is no need to implement a userspace application in an assembly-level language.

Other debuggers than the userspace ones should definitely show the instructions of the binary form, because an assemly code may be used in the domain of operating systems and embedded devices.

Breakpoints allow the programmer to stop the execution of the debuggee at the desired location. The programmer uses a breakpoint by putting it on the code where the debuggee should be stopped. A source-level breakpoint is put on a line of high-level language code and an instruction-level breakpoint is put on an instruction.

Debuggers allow the programmer also to step the stopped debuggee. Stepping will resume the execution just for one statement of the code. For high-level languages the statement is often one line of the code and such stepping is called source-level stepping. For assembly-level languages the statement corresponds to one instruction and such stepping is therefore called instruction-level stepping.

3. Debugging without a debugger

Analyzing erroneous behaviour of an MSIM application can be done in several ways without the debugger. This section discusses the benefits that the debugger would bring. The following debugging techniques are available without the debugger:

1. Debugging messages

   Debugging messages are commonly used to provide some general information such as size of the physical memory, build time of current program or an overview of program initialization. It is also useful to have the messages in a piece of code where a problematic behaviour is expected. Another efficient usage of the messages is during debugging of recursive functions.

   Printing the messages also works as a limited replacement for the missing debugger. By printing a message the programmer can find out the values of variables or check whether the analyzed code has been executed. However, using the debugger for this purpose is much faster and comfortable. Because in order to print the message the programmer would have to compile and run the program again. Also printing too many debugging messages may not be very readable for the programmer.

2. Reading code

   In the opinion of many, analyzing code has proven to be one of the more efficient ways of debugging an MSIM application. For more complicated problems this technique is often even faster than the use of the debugger. On the other hand the technique has one important disadvantage - the programmer has to think a lot. Analyzing the code can be very exhausting for the programmer if it is done often. [1]

3. Consulting the problem with colleagues

   Sometimes the programmer runs out of all ideas of how to localize the source of bad behaviour. Describing and discussing the problem can then bring new approaches how to find the source of error.

4. Using MSIM built-in support for debugging

   MSIM provides support for instruction-level debugging. For use of the instruction-level debugging the programmer is often required to know the address of the analyzed code or variables. The *objdump* utility can help the programmer to find out these addresses. Also the programmer can dump content of registers and the main memory (instructions or raw data) during the simulation.

---

[1] Also note that this method of debugging has been observed to be very inefficient for beginning programmers. The reason is perhaps that they are not used to how environment and constructs of the programming languages work. Using the debugger would probably help a lot to overcome this issue. Unfortunately most students who don't like programming have some mental problems with using the debugger.

The simulator supports instruction-level breakpoints and instruction-level stepping. Using the instruction breakpoint is not very helpful for common debugging of a C code. The programmer would have to know the memory address for putting the breakpoint. For inspection of variables he would have to search for the meaning of the registers in analyzed location.

The binary of an MSIM application can contain special instructions that are recognized by the simulator. Executing of such an instruction will, for example, switch the simulator to the interactive mode. In the interactive mode the simulation is stopped and the programmer can inspect the state of machine, perform stepping or resuming the simulation, or dump the registers. These special instructions can be valuable during debugging of low level code such as exception handling.

Memory breakpoints are also supported in MSIM. They are typically used when the programmer does not know where his variable is being changed. After the breakpoint is set, the simulator will stop on the memory access on the specified address.

The execution of instructions can be logged. MSIM prints the number of the executing processor, address of the executed instruction, name of the instruction, operands and how the operands changed. The following snippet illustrates that:[2]

```
1 BFC00000 lui a0, 0x8000      # a0: 0x0->0x80000000
0 BFC00000 lui a0, 0x8000      # a0: 0x0->0x80000000
1 BFC00004 ori a0, a0, 0x1000 # a0: 0x80000000->0x80001000
0 BFC00004 ori a0, a0, 0x1000 # a0: 0x80000000->0x80001000
1 BFC00008 sw  0, (a0)
0 BFC00008 sw  0, (a0)
 ...
```

On a machine with one processor it is possible for the programmer to use this execution trace for searching the error. Cooperation with the *objdump* during the trace analysis is a need for efficiency. Knowledge of the used ABI also makes this method more efficient. The size of the trace can easily reach hundreds of megabytes, so orienting in the trace can be overwhelming for the programmer. The trace from the multiprocessor machine is too difficult to be analyzed by a human being. Such a trace would have to be further processed to be more valuable. For example, grouping the executed instructions to the blocks that are related to threads would be useful for debugging race conditions.

---

[2] The snippet is taken from [14] and is shortened to fit the width page better.

4. Design of the MSIM debugger

Implementing a debugger can easily be a complex and work-intensive task. This chapter describes considerations that has led to a quality solution with manageable implementation. The MSIM debugger has some specific features, so this chapter additionally discusses usefulness of these features and approach of their implementation.

4.1   The big decisions

The debugger could be implemented as a part of MSIM or as a standalone process. Integrating the debugger into MSIM would allow the debugger to access the debuggee easily. On the other hand it would be much harder to maintain MSIM stable. The maintainability is the reason why the integration with MSIM is not considered. For example, attempts for creating a GUI for MSIM were made and they were proven unfortunate because of portability and maintainability.

So the preferred way is keeping the debugger as a standalone process. The debugger is supposed be connected with MSIM via a TCP connection on the same computer. The implementation can be done completely from scratch or some existing opensource debugger can be adapted. Writing the whole own debugger would be very labor-intensive so using the opensource debugger is much more efficient way. GDB (GNU debugger) is a good candidate. It is often used debugger in the UNIX environment and it supports remote debugging via a TCP connection.

However, GDB is a console based program and the goal of this thesis is creation of a GUI based debugger. Fortunately GDB provides support for creating a GUI front-end. Again it is possible to create the whole own front-end or use an existing one. The suitable opensource front-end was searched, because the first option is much more work-intensive. Eclipse IDE was very promising and it was chosen. Among the debugger it integrates many tools such as: efficient code editor; code analyzer; or very flexible configuration for building and launching programs. Moreover Eclipse with the CDT plugin already allows to develop and debug C/C++ programs. During the debugging session CDT works as a front-end of GDB.

Choosing GDB as a back-end and the Eclipse IDE as a front-end leads to highly comfortable and efficient debugging. All of the functionality for common debugging is already implemented, but the debugging of a MSIM application is specific in a couple of things. Therefore further changes in the behaviour of CDT and GDB are required. Changes in MSIM are also needed to be done.

Launching a debugging session for an MSIM application should be done in a few steps:

1. Eclipse should run MSIM and GDB.

2. A TCP connection between MSIM and GDB should be established.

3. Eclipse should take control of MSIM by instructing GDB.

Unfortunately running MSIM at the start of the debugging session is not configurable in Eclipse and, therefore, it needs to be added. Both Eclipse and GDB does not support viewing the contents of the TLB or the physical memory. MSIM, for example, is not very ready for debugging of a multiprocessor machine.

For testing of the debugger it is suitable to use an MSIM application that is not too trivial and not too complex. A good candidate for testing is Kalisto. Kalisto is a base for operating systems that are developed by the students of an operating systems course. Kalisto will serve for testing purposes and as the reference MSIM application for this thesis.

Development of MSIM debugger will, therefore, require work with four projects: the CDT plugin of Eclipse, GDB, MSIM and Kalisto.

4.2   Debugging multiprocessor machine

Developing an MSIM application for multiprocessor machine can easily be a nightmare for a programmer. For example, ways how to diagnose an assembly-level race condition are very limited for a multiprocessor machine. Therefore, it is important to support debugging of virtual machines with more than one processor.

A straightforward way of making this debugging possible is representing the processors as threads in the debugger. Such an approach would require mostly changes only on the MSIM side, because the concept of thread monitoring is already implemented on the GDB side. This design would have to be reconsidered if the threads inside the MSIM application should also be monitored.

4.3   Memory access

Accessing memory contents of the debuggee is a basic functionality of debuggers. A few troubles were encountered during implementation of the memory access. These troubles required some designing considerations. The whole issue is described in [7] and [10].

4.3.1   Different memory space for each thread

All the threads in the same process usually have the same address space. However, in MSIM debugging sessions a thread corresponds to a processor, and each processor can be in different addressing mode (or can have different contents of its TLB). Therefore, threads can have different address space for the MSIM debugger.

Fortunately, GDB remote protocol handles this uncommonness correctly, because it specifies the desired thread/processor along with each memory access command. However, problems appeared in the CDT plugin. The plugin does not specify the desired thread along with a memory access command, although the protocol of communication between GDB and Eclipse allows it.

Two approaches for solving this trouble were considered:

1. GDB allows debugging of multiple processes at the same time. The MSIM plugin could use this feature and represent a processor in MSIM as a process. However, this approach would require large changes in the CDT plugin. For example, launching of debugging sessions would require starting of a group of processes, or a few GUI views would have to be updated. Additionally the internal data model would have to be modified very probably. This approach was rejected because of the large expected changes.

2. CDT would be changed to specify the desired thread when sending every memory access command. This approach required much smaller changes and, therefore, it has been implemented.

4.3.2   Address spaces with holes

Generally, some blocks of memory addresses can be invalid. For usual userspace programs it would mean that the blocks of memory are not mapped. And for programs simulated in MSIM it would mean the same, or additionally, it would mean that there is no physical memory on the related addresses.

GDB accesses the memory by commands that specify the starting address and the length of the accessed memory. Such an access operation fails if the starting address is invalid, and it does not matter whether there is any valid memory in the specified block. This causes troubles especially in the CDT plugin, because in the described failure situation it marks the whole memory block as invalid. Therefore, the user can see that some valid memory is marked as invalid. And additionally, sometimes the user is not allowed to refresh the invalidated block.

With the current set of memory access commands, GDB would have to create a command for each accessed byte to prevent the described problem. That would possibly have impacts on performance, so another approach has been proposed. GDB would ask the target (e.g. an operating system or MSIM) about the memory map of the debuggee. Then the Eclipse front-end would use this information for exact determination of invalid blocks of memory.

Obtaining the memory map in GDB is already implemented for a few platforms. However, it is not supported for remote targets; and the memory map is not printed in a comfortable way for parsing in front-ends. And finally, appropriate changes in the CDT plugin would have to be done.

This feature is generally beneficial, so it should be added to the GDB and CDT mainlines. That would require discussions and acceptance of both the communities. The discussions were initiated ([7] and [9]), but the proposed changes

has not been accepted so far. Therefore, this feature has not been implemented yet.

4.3.3   Memory mapped devices

The user of the MSIM debugger could be also enabled to access memory mapped devices via memory views. However, this feature would be related mostly with hardware debugging, so it is unimportant. But at least reading of read-only and read-write devices were allowed, because the changes were small and straightforward.

4.4   Showing TLB content

Please note that the possibility of accessing TLB from an operating system depends on the family of the used processor. On the *Intel IA32* family the operating system can only invalidate TLB contents ([12]). For example, the invalidation is used during a memory address space switching. The rest of the TLB management is handled by the processor. On the other hand, the *MIPS R4000* platform leaves all the TLB management to the operating system ([13]).

4.4.1   How is the knowledge of TLB contents useful

The possibility to show the TLB contents may be handy during implementation of the TLB filling mechanism - which is the main reason why the TLB contents can be shown in the MSIM plugin. The implementation of the TLB mechanism is typically done in early stages of an operating system development and in this phase the programmer may appreciate seeing the TLB contents. However, later the TLB just reflects a part of page tables, so an eventual problem with an address translation would not be caused by TLB handlers very likely. Therefore, showing TLB contents is not expected to be very important for a debugging programmer.

Information in TLB can be also used by a kernel debugger for translation of virtual addresses. However, translating with information in TLB would have a disadvantage for the debugger. The debugger could not conclude anything about the translation when information of translated virtual address is not available in TLB. Therefore, for the debugger it would be better to use page tables for address translation purposes. So, the TLB contents are not very important for internal processing in kernel debuggers too.

4.4.2   TLB diagnostic with common debuggers

Showing of TLB contents is not implemented in common debuggers, because it is more likely related with hardware debugging and it is typically impossible. Additionally, even if obtaining TLB contents would be possible for the debugged platform, more complications would have to be dealt. For example, running the

kernel debugger as a userspace process can change the TLB contents and that violates the Heisenberg principle. Therefore, it might be possible to obtain the TLB contents by using a debugger that connects remotely to the debugged kernel. However, handling of such a connection on the kernel side would have to leave the TLB unchanged. All that summarized, obtaining the TLB contents on a real machine would be a complicated matter or it would not be possible at all.

On the other hand, obtaining the TLB contents from a simulator is very easy. That is also the case of MSIM and, therefore, the contents are shown in the MSIM plugin.

4.5 Passing TLB contents and physical memory to the MSIM plugin

Information about TLB and physical memory is easily obtained in MSIM. The question for this section is how to pass that information to the MSIM plugin.

The chosen way is extending the GDB remote protocol and sending the information through GDB. Thus GDB is able to pre-process the information, but currently, that is not needed.

The other considered way was sending the information through a separate TCP connection from MSIM directly to the MSIM plugin. The advantage of this approach is that it does not require changes into GDB. Therefore, the MSIM debugger would be more maintainable with future versions of GDB.

4.6 Design of implementation

Implementing a debugger is a complex matter and numerous mechanisms are needed to be designed. The designer has to decide how to: make GUI responsive; store user's settings; communicate with the debuggee and control it; allow debugging of more than one debuggee at the same time; design the object model and represent data; and so on...

Most of these decisions has already been made by authors of the chosen projects. Therefore, not many things were left to design in this work. The most important mechanisms are outlined in the chapter 5.

However, there are two notable design issues related to MSIM:

- Changing the MSIM main loop

- Extending breakpoints framework of MSIM

4.6.1 The main loop modification

MSIM performs the simulation in its main loop. In each iteration of the loop it stepped all the processors and then checked whether there is a need for commu-

13

nication with GDB. An example for such a need is a breakpoint hit.

The problem with this mechanism was that more than one processor could hit a breakpoint in one iteration. In such a situation GDB responds only to one of the breakpoint hits.

The solution for this trouble was stepping only one device in each iteration of the main loop.

4.6.2 Breakpoints related to all the processors

Before this work MSIM allowed breakpoints that are only related to one processor. In other words a breakpoint was hit if and only if the related processor executed the instruction where the breakpoint is placed.

The problem was that during a debugging session CDT and GDB relates breakpoints only with the currently debugged thread/processor. That mechanism was unfortunate for the MSIM debugger users, because they expect all the threads/processors to be stopped at (usual) breakpoints.

The solution for this problem was implementing breakpoints that are related to all the processors in MSIM.

5. Implementation and know-how

This chapter describes how debugging of Kalisto-like programs in the Eclipse IDE has been made possible. It is aimed for programmers who possibly want to modify the source files of this project and need an initial introduction. The source codes of this project are well commented, so we just describe what and where is implemented instead of going into details. Additionally, well-tried practices are mentioned.

5.1   The big picture
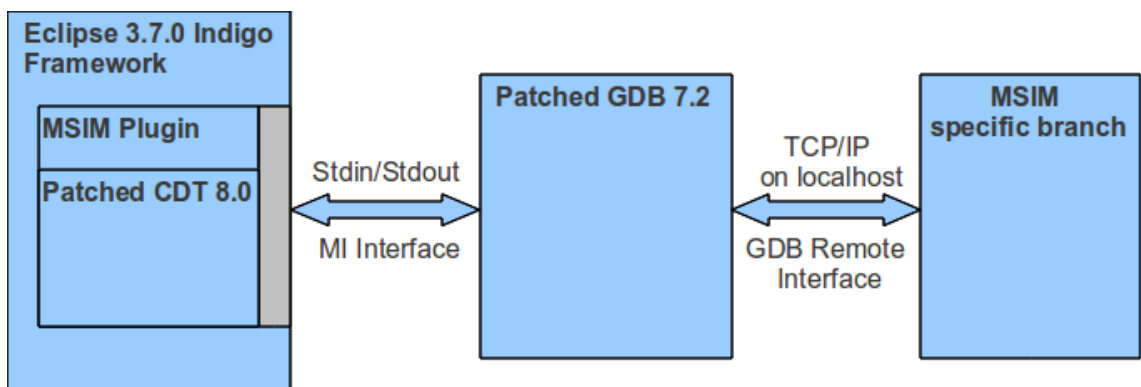
Figure 5.1: **Conception of the MSIM debugger**.

For debugging an MSIM application in the Eclipse IDE four pieces of software are needed: a special branch of MSIM; patched GDB 7.2; the patched CDT 8.0 plugin; and the MSIM plugin for Eclipse.

The debugging session is controlled by the Eclipse framework. During the launch of the session Eclipse starts an MSIM process. MSIM is ordered via the *remote-gdb* command-line option to wait for the GDB connection on the specified TCP/IP port.

After the MSIM process is started, the Eclipse starts a GDB process. The GDB process communicates with Eclipse via the standard input and output and the format of transfered data is specified by the GDB machine interface (MI). The CDT plugin allows to display a *gdb-trace* console window, which logs all the MI communication. This window is very useful for troubleshooting and is also suitable to get an idea of how the MI data looks like. In the next step, Eclipse commands GDB to connect remotely to MSIM. Then the simulation in MSIM is controlled by the Eclipse through GDB.

It might be useful to mention that Eclipse tries to do at least evaluations of debugging information (such as stack unwinding or symbol translation) as possible and leaves this stuff for GDB. On the other hand Eclipse minimize the traffic between itself and GDB by implementing various caching mechanisms.

Communication between MSIM and GDB is client-server based. MSIM is the server and GDB is the client. The format of the communication is defined by the GDB Remote Protocol. Various more or less detailed documentations can be found on the web - this one was used: [5]. MSIM allows printing the data that are transfered between MSIM and GDB. For enabling of the printings the macro *GDB_DEBUG* must be defined during compilation of MSIM. In Eclipse these printings can be seen in the *msim* console window.

Using the *gdb-trace* window and the MSIM printings of the *GDB_DEBUG* macro is the basic approach for analyzing bugs or diagnosing unexpected behaviour of the debugger. It helps greatly to localize the place of problem.

Both MSIM and GDB patch were developed as Eclipse Makefile projects. That provides a good code browsing support and intellisence. Additionally, if you use '-g -O0' compilation options, you can debug these projects in the Eclipse IDE. Running a separate instance of Eclipse for each project (that implies having two workspaces) was preferred, because it allows more comfortable debugging of communication between the two processes.

5.2 Relevant parts of MSIM

If you want to change the GDB interface of MSIM, understanding the following list of things will be probably useful for you:

- Main loop of the simulation

- Device model

- Breakpoints

- GDB interface

5.2.1 The main loop

MSIM is a single-threaded C program, which controls the whole simulation from the loop in function *go_machine* of file *machine.c*. To understand the main loop and how the data transfers to GDB are initiated see the functions: *go_machine*, *handle_gdb*, *gdb_startup*, *should_listen_gdb*, *next_device_step*, and *next_machine_cycle* in file *machine.c*.

5.2.2 Device model

Each hardware unit (CPU, operating memory, keyboard, ...) is represented as a device in MSIM. See the *device_s* structure in *device.h* for a better idea how the devices are represented. The data specific to a device are carried in the *data* field of the *device_s* structure. Each device has its type that defines operations of the device such as reading or stepping. The type is defined by

the *device_ type_ s* structure in the same file. If a device does not support an operation, the appropriate function pointer in *device_ type_ s* has the NULL value.

The general functions for devices are implemented in the *device.c* file. Specialized implementation of devices is located in appropriate files such as *dcpu.c* or *ddisk.c*. If you want to work just with the GDB interface, you will likely want to know how to iterate over devices of the specified type. The function *dev_ next* does it. Here is an example of its usage:

```
/* Iterate over all the processors */
device_s *dev = NULL;
while (dev_next(&dev, DEVICE_FILTER_PROCESSOR)) {
  cpu_t *cpu = (cpu_t*) dev->data;

  // ... do whatever is needed with the processor
}
```

You also might be interested how to access the memory of the simulated program. The functions *mem_ read* and *mem_ write* in file *machine.c* serves that purpose.

5.2.3   Breakpoints

MSIM supports instruction-level breakpoints and memory breakpoints. Both these kinds of breakpoints can be set from the MSIM command-line or from GDB. However, handling of a breakpoint hit differs for the command-line breakpoints and for the GDB breakpoints. So the breakpoints are distinguished by the enum *breakpoint_ kind_ t* in file *breakpoint.h*.

The code in file *breakpoint.c* handles allocation, registration, search and hit of breakpoints.

5.2.4   GDB interface

The GDB interface is implemented in file *gdb.c*. The central function is the *gdb_ session*, which is called from the *handle_ gdb* in the file *machine.c*. During an execution of the *gdb_ session* the simulation remains suspended and the *gdb_ session* handles all the GDB communication until the simulation is resumed by the debugger. The user of the debugger should be able to interrupt the running simulation, so MSIM also reads the input from GDB in a non-blocking way in the function *gdb_ is_ interrupt*.

Hit of a breakpoint that is set from GDB is handled by sending an appropriate event packet to GDB and then waiting for a reply inside the *gdb_ session*.

Additionally, it might be useful to know that the *Hc*, *c* and *s* packets are implemented, but currently they should not be used. The *vCont* packet is used instead of them. For more details see the issue [6].

5.3 GDB patch and troubleshooting

The GDB patch for this thesis allows physical memory and the TLB to be shown in the Eclipse IDE. Additionally, the patch fixes the issues [6] and [8]. These errors are supposed to be corrected in the GDB mainline, so the fixing part of the patch should not be needed in the future. However, the rest of the patch is specific just for MSIM, so the GDB community will not accept it very likely.

5.3.1 Accessing physical memory

Packet for reading the memory is named *vPmem* and for writing the memory *vPMem*. In comparison to the commands *m* and *M* they differ just by usage of physical addresses. So the syntax and meaning of the parameters remains the same as for the *m* and *M* packets.

Functions *remote_read_bytes* and *remote_write_bytes* are used for work with the memory on a remote target. The patch extends them by an additional parameter that specifies whether the memory operation is with physical memory or with virtual memory. The parameter is passed through the global variable *transfer_mode* in file *msim.c*. Using the global variable keeps the changes of the patch minimized, because otherwise the parameter would have to be passed through a long chain of functions.

5.3.2 Reading TLB contents

A special packet named *vtlb* is designed for this purpose. The response for the packet has the following syntax:

ENTRIES_COUNT,ITEMS_COUNT;ENTRY0;ENTRY1; ... ;ENTRY_LAST;

where the *ENTRIES_COUNT* specifies the count of TLB entries and the *ITEMS_COUNT* specifies the count of fields in each entry. The syntax of an entry is:

NAME0=VALUE0,NAME1=VALUE1, ... ,NAME_LAST=VALUE_LAST

Both the names and values of items are text strings without control characters that are used in the remote protocol and syntax of this packet.

GDB does not interpret the obtained TLB contents and just prints them to the output in the MI format. Implementation of TLB reading is placed in the function *remote_read_tlb* in file *remote.c* and in functions of the file *msim.c*.

5.3.3 Extending MI commands

The commands are declared or defined in files *mi-cmds.h*, *mi-main.c* and *msim.c*. The commands *-data-read-memory* and *-data-write-memory* are extended for work with the physical memory. The patch adds a new option *-physical* that tells to access the physical memory.

The MI command for obtaining TLB contents is named *-data-read-tlb*. It can use the *–thread* parameter for specifying the CPU of the obtained TLB. The answer is defined by the following synopsis:

```
ANSWER <- tlb=MI_LIST_OF_ENTRIES
ENTRY <- entry=MI_LIST_OF_ITEMS
ITEM <- NAME=VALUE
```

The *NAME* and *VALUE* are strings obtained from the *vtlb* packet.

You can see an example of the command usage in the *gdb-trace* console window, when you display the *TLB Contents* window.

5.3.4 Troubleshooting GDB

The important files for controlling the debugging session in GDB are *remote.c*, *target.c* and *infrun.c*. They are densely commented, but their functions are sometimes very long, which makes them tough for understanding. It is useful to enable appropriate debugging messages for deep analysis of GDB behaviour. The messages can be enabled by the commands *set debug remote 1* and *set debug infrun 1*. Problems with GDB can be also discussed in the GDB mailing list.

5.4 The MSIM plugin

Before we describe the implementation of the MSIM plugin itself, it would be suitable to mention the following aspects of the Eclipse framework and the CDT plugin:

- plugin extensions and extension-points mechanism,

- the Debugger Services Framework (DSF).

Basically, the MSIM plugin does these things:

- handles new way of launching,

- extends some DSF services to customize Eclipse behaviour,

- adds the physical memory view and the TLB view.

Also note, that the *WindowBuilder* plugin was the preferred tool for creating GUI.

5.4.1 Eclipse plugin extension mechanism

In the Eclipse framework a plugin is a special kind of a java project. A plugin can declare an extension point that can be extended by functionality of other plugins. The plugin can also declare an extension for an extension point. The extensions and extension points of a plugin can be configured in the file *plugin.xml* in the root directory of the project. You can use a comfortable IDE view for editing this file. The options and parameters of extension points and extensions are documented in Eclipse or CDT reference documentation. As an example of a usage of this mechanism see how the *TLB Contents* view was configured and implemented in the MSIM plugin.

5.4.2 DSF services

CDT uses the Debugger Services Framework (DSF) as a comfortable way of implementing GUI data providers. There is a nice tutorial for DSF [11] on the Eclipse sites. Basically, the framework allows an easy use of asynchronously called methods. With the usage of this framework, it is easy to prevent blocking of the thread that handles GUI. Thus the GUI is responsive.

A DSF service is a class that provides specific data to the rest of world, typically in an asynchronous way. Examples of these services are: the *MIMemory* for access to the memory of the debuggee; the *MIBreakpointsManager* for work with breakpoints; or the *MsimBackend* for starting and eventually killing the MSIM process.

The data from services are obtained by giving an appropriate context to the service. The context is an object of a class that implements the *IDMContext* interface. Some contexts have a hierarchical structure and it is useful to understand it. Especially, how variables of type *IDMContext* can be transformed into variables of a more specific type. Methods *getAdapter* and *getAncestorOfType* serves that purpose.

5.4.3 Launching debugging sessions

Launching a debugging session for MSIM is the most similar to the launch of a remote GDB session. The main difference is that the launch of MSIM session additionally requires starting an MSIM process. The most important method is the *MsimLaunchDelegate.launchDebugSession*. It creates all the DSF services that are needed for the session and then it gives initial commands to GDB.

Some DSF services of GDB are left unmodified, some are modified (*MIMemory*), and some are added (*MsimBackend* or *TLB*). The *MsimBackend* service cares about running MSIM. Creation of the services is implemented in the class *MsimServicesLaunchSequence*. Initialization of GDB is done in the class *MsimFinalLaunchSequence*.

Launching MSIM requires some specific settings. These settings are held in a class that implements the *ILaunchConfiguration* interface. The interface

works as a map of attributes and their values. Names of the attributes are taken from classes *ICDTLaunchConfigurationConstants*, *IGDBLaunchConfigurationConstants* and *IMsimLaunchConfigurationConstants*. You can search for usage of these attributes by listing all the references of your desired attribute. The GUI for MSIM debug configuration settings is implemented in classes *MsimLaunchConfigurationTab*, *MsimLaunchTabComposite* and *MsimTabGroup*.

Handling key shortcuts for launching debugging sessions is located in the class *MsimApplicationShortcut*.

5.4.4 Customizing the memory service

The most important change of GDB services is in the way of work with the operating memory. The whole issue is reported in [10]. The CDT plugin considered that all the threads of a process have the same address space. That is not right for a MSIM debugging session, because a thread represents a processor and processors can be in different addressing modes. Therefore, appropriate changes had to be done. To understand the changes well, we should see how the work with memory was done in original CDT. Please be aware that it is a complicated mechanism.

GUI works with instances of *DsfMemoryBlock* class. The memory that is contained in the block is characterized by its start address, its length and by a memory context. Usually memory contexts differentiate memory blocks that are related to different debuggees. The memory block objects listen to memory related events of the memory service and eventually updates their contents. Objects of *DsfMemoryBlock* gets the actual memory contents from the memory service in the method *fetchMemoryBlock*. Instances of *DsfMemoryBlock* are created by the class *DsfMemoryBlockRetrieval*, which specifies the memory context of the block.

The memory service (*MIMemory*) caches the memory obtained from GDB. The cache is invalidated when a resume or a suspend event occurs. *MIMemory* uses two separate caching mechanisms to implement the caching behaviour. The first mechanism stores the continuous pieces of memory in a list for each memory context. These lists are stored in a map that uses memory contexts as keys. In other words *MIMemory* stores a separate list of loaded memory blocks for each debugged process. This mechanism is implemented in the *MIMemory.MIMemoryCache* class.

The second mechanism is implemented in the *CommandCache* class and it caches the results of MI commands. The commands themselves are used as keys for searching the cached result.

Work with the contexts that are given to the memory service is quite a problematic matter. Possible contexts that are used for memory access are a thread group context and a thread context. The structure of these contexts is shown in figures 5.2 and 5.3.

The only memory context of those in figures 5.2 and 5.3 was the thread group context (*GDBProcesses_7_0.GDBContainerDMC*). Therefore, the memory service receives as a parameter even contexts that are not memory con-

```
GDBProcesses_7_0.GDBContainerDMC          // a thread group context
  GDBProcesses_7_0.MIProcessDMC           // a process context
    GDBControlDMContext
```

Figure 5.2: **Parents of a thread group context**.

```
GDBProcesses_7_0.MIExecutionDMC           // a thread context
  GDBProcesses_7_0.GDBContainerDMC        // a thread group context
    GDBProcesses_7_0.MIProcessDMC         // a process context
      GDBControlDMContext
  GDBProcesses_7_0.MIThreadDMC            // an OS thread context
    GDBProcesses_7_0.MIProcessDMC         // a process context
      GDBControlDMContext
```

Figure 5.3: **Parents of a thread context**.

texts. So the service tries to drill the memory context from the passed context. It chooses the nearest parent that represents a memory context, which is always the thread group in the original CDT. Note, that the method *DMContexts.getAncestorOfType* is used for getting the memory context from the given context.

Additionally, context given to the memory service affects also options of the MI memory access command. Using the thread group context will add a *–thread-group group-name* option, and then GDB will read memory of a random thread from the group. Using the thread context will add a *–thread thread-number* option, and then GDB will read memory of the given thread. GDB implements memory access of a thread by:

1. Selecting the thread in the GDB remote communication,

2. Accessing the memory,

3. Selecting the previously selected thread back.

Now let us see what is wrong with the described memory mechanism for an MSIM debugging session and how the mechanism was customized. Firstly, we want to use only *–thread* option for memory commands. Secondly, even if we give a thread context to the memory service, the service will use the thread group context as the memory context. So all the threads of the debuggee will share the same list of cached memory blocks.

The solution for this problem was achieved by creating a new class *MsimThreadDMContext* for both the memory context and the thread context. In order not to violate the function of the previous implementation, the classes that derives *MIMemory* are allowed to provide their own caching mechanism for memory blocks. A new caching mechanism was created in the class *MsimMemory*. The main difference is that the key for the list of cached blocks is a pair of values: memory context - thread context. Additionally, the class *MsimMemoryBlockRetrieval* ensures that a thread group context will be never given to the memory service.

5.4.5 Accessing physical memory

Modifying the memory access mechanism for reading physical memory is quite straight-forward if we understand how the memory mechanism works.

A new memory context *MsimPhysicalMemoryDMContext* for physical memory was created. We also need the *MsimPhysicalMemoryBlockRetrieval* memory retrieval that will return DSF memory blocks with the new memory context. The *Physical Memory* view will use this block retrieval.

Additionally, we need the memory service to reflect this new memory context. Therefore, the service will now use new MI commands *MIDataReadMemoryMsimExt* and *MIDataWriteMemoryMsimExt* for accessing the memory. For previously used memory contexts these commands behave exactly like the previously used ones, but they add the *-physical* option for the *MsimPhysicalMemoryDMContext* context.

The GUI for the physical memory is implemented in the class *PhysicalMemoryBrowser*.

5.4.6 Viewing TLB Contents

Understanding implementation of the *TLB Contents* view is nothing really hard. The implementation is quite straight-forward, because the contents of TLB can be only read and displayed. Additionally, the appropriate DSF service does not use any caching mechanism - the performance impact has not been observed.

The GUI is in classes *TLBBrowser*, *TLBPane* and *TLBTableViewer*. The class *TLB* implements the DSF service.

6. Advanced debugging features

For debugging of interpreted code it is possible to use debugging techniques that are impracticable for debugging on a real machine. Some of these techniques are proposed in the following text.

6.1 Debuggee-specific debugging information

Some advanced debugging features requires the debugger to manipulate with internal data structures of the debuggee. For example, the debugger may need to know: the kernel structure of a thread; the list of active synchronization primitives; the list of opened file descriptors; or page tables of a process.

For MSIM, it would be very hard to obtain these pieces of information just by reading registers and the memory. One way of overcoming this trouble would be that the user would specify additional information to the debugger. For example, he would specify an address of the *current_threads* variable that holds information about currently running threads. Then, the debugger would be able to obtain data about running threads by reading the specified memory.

The described mechanism would not work well if the debugging information was not in a fixed memory place. Therefore, a new way of passing advanced debugging information is needed to be designed. Two approaches are proposed in the following text:

1. using special debugging devices in MSIM,

2. using a special connection to the debuggee.

6.1.1 Communication between the debuggee and the debugger

The first one is defining a special interface between MSIM and the debuggee and then extending the GDB remote interface. For example, the interface between MSIM and the debuggee could be realized by creating new types of instruction. However, perhaps the best would be adding special debugging devices to MSIM. For a better idea of how these devices would work, let us consider such a device for monitoring of currently running threads on a multi-processor machine. Typical kernels hold a special variable for each processor that determines currently running thread on the related processor. During a context switch the debuggee would write to the thread monitoring device along with changing the variable for currently running thread. GDB would ask MSIM for contents of the device to get the list of currently running threads in the debuggee. This mechanism would allow easy additions of debugging devices.

The second way is opening another debugging connection directly to the debuggee. It would work in the same way as debugging usual kernels remotely.

Thus MSIM would not be aware of passing the debugging information at all. The only thing MSIM would have to provide is a device for a network interface card or a serial port. On the other hand, the debuggee would have to implement serving of the device and handlers for the GDB remote communication. Controlling the debuggee via two debugging sessions would require changes in the internal logic of the debugger. For minimal changes the connection to the debuggee would be used just for obtaining information that are difficult for MSIM to obtain. And the connection to MSIM would be used for both controlling and obtaining basic debugging information.

6.2   Extended execution control

Debuggers typically do not allow to resume just one specified thread while the others are stopped. This can be unpleasant for the developer. For example, he may want to debug just one thread and does not allow the others to change the state of the program. Such a debugging option usually requires a cooperation with the scheduler of the operating system. Unfortunately most operating systems do not support such a feature. GDB has interface only for locking the scheduler of the operating system. Using the locking (if it is supported) prevents the current thread to be preempted. Additional details about this matter can be found in [3].

For MSIM it is not hard to implement controlling execution only of the specified processors. Unfortunately GDB and Eclipse are not prepared for such a debugging possibility and therefore changes in them would have to be done. For GDB at least the remote connection interface and the command line interface would have to be extended. For support in Eclipse a checkbox with function *Execute only the current thread* is proposed. This button might be placed next to the buttons *Resume, Stop, Step over, ...*

The described feature could be generalized to control execution of not just a single thread but more selected threads. For example, the programmer debugging a multi-threaded program could use the generalized execution control to test whether a critical section is handled properly. He would stop one thread in the critical section and then he would allow the other threads to run. The code in the critical section would not be secured properly if any other thread would enter it.

6.3   Detection of violated critical sections

Logical errors in thread synchronization are very hard for diagnosis. The following mechanism could help significantly to localize these errors.

In a typical mutual exclusion problem the programmer uses mutex to allow at most one of the threads to execute the synchronized code at the same time. The synchronized code is located in so-called critical sections.

In the proposed mechanism the programmer would annotate the critical sections and specify the related mutual exclusion problem for each section. The

debugger would then put internal breakpoints to the beginnings and ends of the annotated critical sections. Thus the debugger could check whether the mutual exclusion conditions holds every time when a thread enters a critical section. If the condition would not hold, the debugger would stop execution and report it to the programmer. Therefore, the programmer would know that there is a race condition in the shown critical section and he would be in much better position for diagnosing it.

The proposed mechanism can be implemented even in the environment of usual operating systems that runs on real hardware. The information provided by the annotations would have to be reflected by the compiler and stored in the executable file. The operating system would put breakpoints and handle them in the same way as the debugger would do. The simplest way of stopping an execution when the mutual exclusion condition does not hold is killing the process. But for better debugging support the operating system could allow an external debugger to attach to the broken program, or at least generate a core dump of the program.

6.4   Debugging of debuggee's internal threads

The current implementation of the MSIM debugger represents each processor in MSIM as a thread in the debugger. However, a typical debuggee implements its own thread system and the programmer may want to see actions of debuggee's threads. A debuggee's thread can be executed on different processors, so currently the programmer would have to check which debuggee's thread is being executed on the stopped processor. That would not be very comfortable.

Additionally, stepping of debuggee's thread would not work well with the current implementation. For example, let us consider that the programmer steps over a *thread_sleep* call. Note that the debugger performs stepping over by the following actions:

1. Put a breakpoint on the next statement.

2. Resume the debuggee.

3. When the breakpoint is hit, check whether the thread that hit the breakpoint is the same as the stepped thread. If so, stop the execution and inform the user. Otherwise resume the execution and wait for the next breakpoint hit.

The programmer would expect that the execution will be stopped when the stepped debuggee's thread wakes up. However, currently the execution will be stopped when the stepped processor reaches the statement after the *thread_sleep* call. And in that time the processor can execute the code of a different thread. So the execution would be stopped in another debuggee's thread. Moreover, the debugger will not stop the execution if a different processor reaches the statement.

26

And the debuggee's thread can be executed on the different processor when it wakes up. Therefore, the debugger could even never stop after the stepping.

Support for viewing and debugging of debuggee's threads would require the debugger to distinguish two types of threads - threads for representing processors and threads for representing debuggee's internal threads. The user interface of the debugger would have to reflect these two types. And user's actions for execution control (e.g. stepping) would have to be related with either processors or debuggee's threads.

Additionally, the debugger would have to keep track of currently running debuggee's threads and obtain list of all the debuggee's threads. Section 6.1 discusses how to obtain that information.

6.5   Debugging code in virtual memory

So far the code of the debuggee has been always placed in physical memory in addresses that are known after compilation. But the code can be also placed in virtual memory and that brings complications.

Firstly a virtual address space is typically related with a debuggee's internal process. Therefore, debugging of code in virtual memory requires the debugger to access virtual memory of the debugged process. For doing so, the debugger would need at least to obtain page tables of the desired process.

Secondly the code in the virtual memory can be relocated during loading to the memory. For more information about relocation see [2, Ch. 4.1.3, p. 377]. Therefore, the location of the code in the virtual memory can be determined only by the debuggee. And the debugger needs to know the location, because it is used for translation of debugging symbols.

Section 6.1 discusses possible ways how the debugger can obtain page tables or the information about relocation.

6.6   Debugging either userspace or kernelspace code

The programmer may want to debug just events in the kernel and ignore what the examined debuggee's thread does in userspace. Or on the other hand, he may want to debug actions in userspace and ignore what is happening in the kernel.

Whether a debuggee's thread is in userspace or in kernelspace is determined by control bits in the *CP0* coprocessor of the executing processor. MSIM can easily access these bits and monitor their changes. The debugger could use this information to stop when the kernelspace/userspace status of the debugged thread changes, or to filter breakpoints where the user does not want to stop.

6.7   Recording execution

MSIM can provide execution trace log, which allows the programmer to see what the machine has been doing. Unfortunately, the log is usually overwhelming for the programmer, so he uses it only when most other debugging methods failed. However, the debugger could greatly improve readability of the log.

The debugger could easily replace addresses of symbols by the symbols themselves. The trace log of MSIM could be transfered to the debugger during debugging session, or MSIM would store the log into a file and the debugger would post-process the file. Showing the log in an IDE view would be useful, because the debugger could show the related lines of C source code to the selected line of the trace log. Additionally, the debugger could be able to reconstruct the call stack from the log.

The trace log for a multiprocessor machine in its raw form is not very human-readable. That is because the programmer sees instructions of all the processors melted together. It is very hard just to seek what one single processor is doing. The picture 6.1 illustrates how such a log looks like.

Figure 6.1: **Melted trace log for a multiprocessor machine**. The picture shows how MSIM generates log for a machine with three processors. Four instructions are shown for each processor.

The debugger in an IDE could show the log for a multiprocessor machine in much more readable way. It would create a standalone GUI element for each processor to display instructions of the processor. These elements would be grouped one next to the other as the picture 6.2 illustrates.

6.7.1   Call trees

For a single-threaded program the execution can be also described by displaying a call tree. A node of such a tree represents a routine and a transition represents a

28

call of a subroutine. A call tree contains less information than an execution trace, because it contains only names of routines. That may be actually an advantage, because the programmer usually starts to analyze the execution by finding out where he is and how he has got there. And he can see it directly from the call three.

However, the debugger can construct the call stack from an execution trace too. Therefore, the actual advantage of call trees with comparison to execution traces is that call trees shows how routines are called in a larger way. For example, the programmer can see directly from the call three what routine is called right after the current routine returns.

Unfortunately, displaying a call tree to show how the program has been executed has numerous problems:

1. Displaying conditional statements in the tree

   The information in the usual call tree may not be sufficient for the programmer. He may want to know how the execution has been branched inside a routine. Therefore the branching statements such as *if* or *switch* should be shown in the tree as nodes. Additionally, such a node should contain information how the branching condition has been evaluated.

2. Displaying loops in the tree

   Usual presentation of call trees can be easily unreadable. For example, calling a simple routine in a loop with one million iterations would result in displaying one million of nodes. The programmer would get immediately lost in traversing such a tree. Therefore, the loop should be also represented as a node in the tree and the programmer should be allowed to display only subtree of a specific iteration.

3. Multi-threading

   A call tree is related to a single thread. For a multi-threaded program, there is a simple idea of having a standalone call tree for each thread. However, the programmer may want to compare what the threads are doing at specific

```
trace_log_CPU0.txt ✖
1 8000438C     jr    ra
2 80004390     nop
3 800043C4     lw    v0, 0x18(fp)
4 800043C8     addiu v0, v0, 0x1
```

```
trace_log_CPU1.txt ✖
1 80002420     addu  fp, sp, 0
2 80002424     sw    a0, 0x8(fp)
3 80002428     lw    v0, 0x8(fp)
4 8000242C     lw    v0, (v0)
```

```
trace_log_CPU2.txt ✖
1 80002420     addu  fp, sp, 0
2 80002424     sw    a0, 0x8(fp)
3 80002428     lw    v0, 0x8(fp)
4 8000242C     lw    v0, (v0)
```
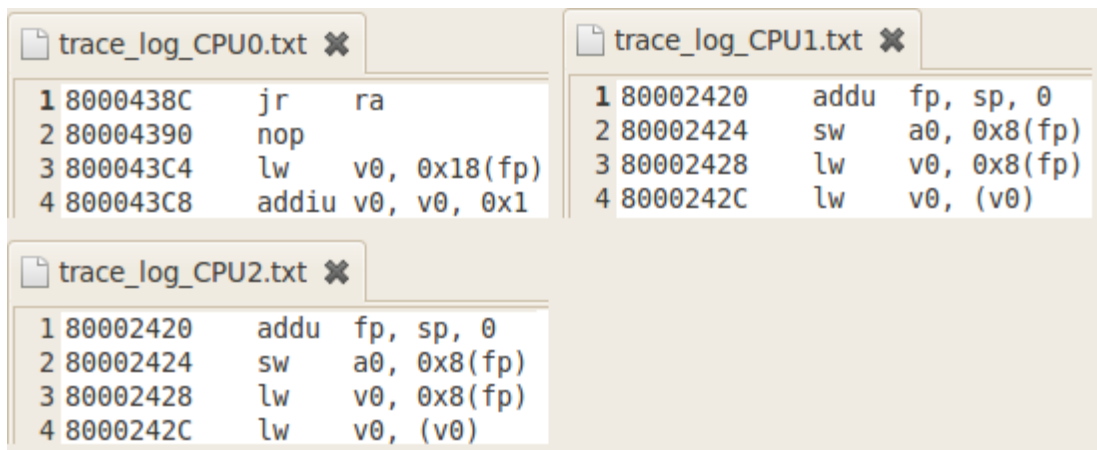
Figure 6.2: **Separated trace log for a multiprocessor machine**.

time. For example, that could be done by selecting a node in one call tree and then asking the debugger to show where the others were when the selected node was being executed. Nodes does not specify the location on instruction-level precision, so the shown result would be a range of possible nodes likely.

4. Handlers for asynchronous events

   Call trees do not reflect asynchronous events such as interrupt handling in a kernel or signal handling in a userspace program. It may be acceptable to consider an execution of an event handler as an execution of a standalone thread. Thus each handler would be related to its own call tree.

5. Call trees for threads that represents processors

   Unfortunately, the call three concept does not make a good sense for recording execution of a thread that represents a processor in MSIM. For example, one assumption of a call tree is that it reflects all the routines of just one thread. However, a routine can be executed on more than one processor and that violates the assumption.

For a summary, call trees are a way of representing an execution of a program. They bring a series of design questions to be practically useful. With the comparison to the execution traces they do not have remarkable advantages. Additionally, the traces are much more easier for implementation in the MSIM environment. Therefore, in author's opinion a theoretical usage of the call tree concept can be expected more likely in debugging.

6.8   Reverse execution

Programmers would appreciate the ability to return the state of the debuggee back to some point. They would possibly like to evaluate what values the variables has had. Also, it is quite common that the debugging programmer steps the program to many times and misses an important moment. Normally he would have to reproduce the whole stepping procedure, but with reverse execution he would just do one step back.

Reverse execution is typically implemented by recording changes made by each instruction. For example, an instruction that would change the memory would correspond to the record: memory changed at address X, previous value Y, new value Z.

Another approach of implementing reverse execution is suggested. MSIM would periodically create a snapshot of the whole machine. In order to return the program to some previous point the MSIM would load the nearest snapshot before that point and execute forwardly to the point.

Reverse execution is obviously very demanding feature for the size of memory. Therefore, memory usage is an important property of implementation. Two situations are considered to compare the two described approaches of realizing reverse execution. At first let us consider that reverse execution could be performed

only for a fixed count of previous instructions. This limitation would not matter in situations such as the example with the programmer who has done too many steps and missed an important moment. For the low enough count of reversible instructions the typical approach of recording changes would be more suitable. That is because remembering changes of so few instructions is not demanding for the size of memory. Additionally, reversing those instructions would be probably much faster than loading the snapshot of the machine.

On the other hand, the programmer may want to reverse the program before the loop with too many iterations, so the limited reverse execution would be useless for this purpose. For unlimited reverse execution it is harder to guess which approach is better and a study of the memory usage is suggested.

There is another notable thing about reverse execution. The state of a program can also be determined by non-deterministic events such as a key press. These events are very hard or impossible to be reversely executed in real machines. For the MSIM environment it is possible to remember changes caused by these events. Therefore, in the MSIM environment the reverse execution may be applied even to non-deterministic programs.

7. Conclusion

In this work we implemented the MSIM plugin for Eclipse, a patch for the GNU debugger, and we modified MSIM. These things together form the MSIM debugger. The debugger allows comfortable debugging of programs that are simulated in MSIM. Thus, the goal of this thesis has been achieved. The table 7.1 reviews debugging possibilities of the MSIM debugger. Hopefully, this work will save a lot of debugging time to MSIM users.

Additionally, many debugging features that can not be implemented in common debuggers, can be implemented in the MSIM debugger. Some of these features are discussed in this work. The usefulness of the features may not be guessed well, but we could have the possibility to observe how students debug an operating system. Thus we could be able to collect typical use-cases of the debugging features and evaluate their benefits. These evaluations would provide a good background for potential research in debugging.

General features	MSIM built-in support	Normal GDB	MSIM plugin for Eclipse	Userspace C program in Eclipse
Debugging in IDE	No	Is not supposed	Yes	Yes
Source-level debugging	No	Yes	Yes	Yes
Assembly-level debugging	Limited	Yes	Yes	Yes, but typically not used
Specific features				
Execution control, stop, resume	Yes	Yes	Yes	Yes
Execution trace	Yes, but with no symbols	No	No	No
Instruction stepping next, in and out	Only next instructions	Yes	Yes	Yes, but typically not used
Instruction-level breakpoints	On physical addresses	Yes	Yes	Yes, but typically not used
Source stepping next, in and out	No	Yes	Yes	Yes
Source-level breakpoints	No	Yes	Yes	Yes
Physical memory access	Yes	No	Yes	No, but not useful
Virtual memory access	No	Yes	Yes	Yes
Watchpoints	On physical addresses	Yes	Yes	Yes
Registers access	Only read	Yes	Yes	Yes
TLB reading	Yes	No	Yes	No, but not useful
Call stack	No	Yes	Yes	Yes

Table 7.1: **Comparison of different debugging scenarios**. The first three columns are related to the debugging of programs executed in MSIM. The last column shows possibilities of an IDE-based debugger for common userspace C/C++ programs to point out the contrast. The pictures are derived from `http://eci2.xstamper.com/ProductDetail.aspx?productid=11420`.

References

[1] ROSENBERG, Jonathan B. *How Debuggers Work : Algorithms, Data Structures, and Architecture*. New York : Wiley, 1996. 256 p. ISBN 0-471-14966-7.

[2] TANENBAUM, Andrew S.; WOODHULL, Albert S. *Operating systems : Design and implementation*. 3rd ed. Upper Saddle River (New Jersey) : Prentice Hall, 2006. 1080 p. ISBN 0-13-132938-8.

[3] *Debugging with GDB* [online]. 2003, updated Jun 2003 [cit. 2011-07-18]. Stopping and starting multi-thread programs. Available from WWW: <http://www.delorie.com/gnu/docs/gdb/gdb_40.html>.

[4] *Debugging with GDB* [online]. 2011, last updated: Mon Jul 18 01:58:45 UTC 2011 [cit. 2011-07-18]. The GDB/MI Interface. Available from WWW: <http://sourceware.org/gdb/current/onlinedocs/gdb/GDB_002fMI.html#GDB_002fMI>.

[5] *Debugging with GDB* [online]. 2010, generated on January, 20 2010 [cit. 2011-07-18]. GDB Remote Serial protocol. Available from WWW: <https://idlebox.net/2010/apidocs/gdb-7.0.zip/gdb_37.html#SEC673>.

[6] *GDB mailing list* [online]. [cit. 2011-07-18]. GDB Remote Serial protocol. Available from WWW: <http://sourceware.org/ml/gdb/2011-05/msg00034.html>.

[7] *GDB mailing list* [online]. [cit. 2011-07-18]. Addition of a special memory reading command. Available from WWW: <http://sourceware.org/ml/gdb/2011-05/msg00137.html>.

[8] *GDB bugzilla* [online]. [cit. 2011-07-18]. Reading memory from target omits the first byte. Available from WWW: <http://sourceware.org/bugzilla/show_bug.cgi?id=12733>.

[9] *CDT mailing list* [online]. [cit. 2011-07-18]. Different address space for each debugged thread. Available from WWW: <http://dev.eclipse.org/mhonarc/lists/cdt-dev/msg22070.html>.

[10] *Eclipse bugzilla* [online]. [cit. 2011-07-18]. Support for gdb memory cache customization. Available from WWW: <https://bugs.eclipse.org/bugs/show_bug.cgi?id=349160>.

[11] *Eclipse documentation - Previous Release (Eclipse Helios)* [online]. 2010, [cit. 2011-07-18]. Introduction to Programming with DSF. Available from WWW: <http://help.eclipse.org/helios/index.jsp?topic=/org.eclipse.cdt.doc.isv/guide/dsf/intro/dsf_programming_intro.html>.

[12] *Intel®64 and IA-32 Architectures Software Developer's Manual : System Programming Guide* [online]. Volume 3 (3A & 3B). [s.l.] : Intel, May 2011 [cit. 2011-07-18]. Invalidating Caches and TLBs, Vol. 3A 2-31. Available from WWW: <`http://www.intel.com/Assets/PDF/manual/325384.pdf`>.

[13] *MIPS R4000 Microprocessor User's Manual* [online]. 2nd ed. Mountain View, California : MIPS Technologies, 1994 [cit. 2011-07-18]. Available from WWW: <`http://d3s.mff.cuni.cz/~ceres/sch/osy/download/R4000_Users_Manual_2Ed.pdf`>.

[14] DĚCKÝ, Martin; HOLUB, Viliam. *MSIM Version 1.3.8 Reference Manual* [online]. Version 1.3.8. 2007, last change 2010-10-05 [cit. 2011-07-19]. MSIM Version 1.3.8 Reference Manual. Available from WWW: <`http://d3s.mff.cuni.cz/~holub/sw/msim/reference.html`>.

List of Figures

List of Tables

A. Setting up the development environment

Creating the development environment is explained for *Windows 7* with *Cygwin*. This platform is probably one of the more complicated platforms for the setup. For *Linux* you may skip steps that make sense only for the *Cygwin* platform. This procedure is not the only way of the setup and you can customize it if you do not like it. Unfortunately, you will probably meet minor issues during the setup that can vary for each platform or version of the distribution. We describe how to handle these issues in *Cygwin*, but something relevant can be changed there in a few months. Therefore, detailed instructions might be misleading in the future. Please be aware that making the whole procedure may take several hours of work.

For further development you may want to work with: both the MSIM plugin and the CDT plugin; GDB; MSIM; and Kalisto. These directions lead to a well-tried development environment that has one Eclipse IDE instance for each of the projects. You may not need diagnosing GDB behaviour, for example, and in such a case you can skip some steps in the GDB part of the procedure.

Firstly, you will need to install binutils for the MIPS platform. Use the script `toolchain.mips.sh`, which can be downloaded on the sites `http://d3s.mff.cuni.cz/~ceres/sch/osy/main.php`. Make sure that you have all the prerequisites installed - it will save you a lot of time.

We will try to make the following directory structure in your working directory:

```
working_dir
  runtime−EclipseApplication // workspace for Kalisto
  workspace_gdb
  workspace_msim
  workspace_msim_plugin
```

So let us start by creating the workspace directories. In the following sections we will fill each of them. The *runtime-EclipseApplication* workspace will be created automatically.

A.1   Workspace for MSIM plugin

At first we need to start with the CDT development. The procedure for CDT is also described in `http://wiki.eclipse.org/CDT/git`. Please see it, because it may be more updated.

1. Download *Eclipse Classic 3.7* from the sites of the Eclipse project. Extract it wherever you want and run it. This will be our Eclipse for developing the MSIM plugin, so select the `workspace_msim_plugin` as the workspace directory. We need to install additional plugins - *EGit* and *Remote System*

Explorer End-User Runtime. The best way of installation is probably going through *Help->Install New Software* menu in Eclipse.

2. Download *CDT8.0* package from CDT sites. We will need only the package (zip archive), so do not install it. Extract it wherever you want and open *Preferences->Plug-in Development->API Baselines* in Eclipse. Click on the *Add Baseline. . .* button and select your extracted CDT folder.

3. Now we need to checkout two additional projects from CVS. Use the CVS client in Eclipse. In *CVS Repositories* view click on the *Add CVS Repository*, paste `:pserver:anonymous@dev.eclipse.org:/cvsroot/tools` in the *Host* edit box and finish the addition. Right click on the *HEAD -> org.eclipse.orbit -> net.sourceforge.lpg.lpgjavaruntime* and select *Checkout As. . . .* Click twice on the *Next* button, then refresh the tags and select the version 1.1 in branches.

4. The second repository is `:pserver:anonymous@dev.eclipse.org:/cvsroot/eclipse` and the project name is *org.eclipse.test.performance.* You do not need to specify branch this time.

5. Now we will clone the CDT repository. The CDT community has recently switched their version control system from CVS to Git, so we will use the installed *EGit* plugin. Clone a Git repository in the *Git Repositories* window. The URL is `git://git.eclipse.org/gitroot/cdt/org.eclipse.cdt.git`. Select only the *cdt_8_0* branch. The directory of the clone does not matter, but I preferred the `workspace_msim_plugin`. When the cloning is done, right click on the *Working directory* in the *Git Repositories* view and import all the projects.

6. Build the workspace. Unfortunately some problems can easily occur in this step. The package *org.eclipse.cdt.internal.core.macosx* could not have a reference to package *org.eclipse.cdt.core*, which can be fixed by adding the reference in the quick fix menu on the appropriate *import* line. You can also meet an error that a major version should be incremented, which can be corrected by the quick fix too. Sometimes it is enough just to clean the workspace and build it again.

7. Make sure that CDT is working by running any CDT project as an *Eclipse Application.* This step should create the *runtime-EclipseApplication* directory as a workspace for the currently launched Eclipse.

8. Checkout the MSIM plugin from `https://svn.fenix-hosting.cz/repos/msim_debugger` to `workspace_msim_plugin`. The login is *public* and the password is *public* too. You can use *Sublipse* plugin or any other SVN client to do it.

9. Apply the patch `msim_plugin/patches/MSIM-CDT8.0.patch` to any CDT project in Eclipse. You can do it through *Team->Apply Patch. . .* menu in the *Project Explorer.*

10. Build the workspace and run the MSIM plugin as an *Eclipse Application.* The MSIM plugin should be now ready for work.

A.2   Workspace for GDB

In this section we will patch GDB and prepare Eclipse for developing GDB. The users might want to patch GDB too, so they can also use these instructions for patching. Note that you does not have to install the Eclipse if you prefer another way of debugging.

1. Download *gdb-7.2* source files and copy it the `workspace_gdb` directory.

2. Copy the `msim_plugin/patches/MSIM-GDB7.2.patch` to the `workspace_gdb` and apply the patch by the command:

   ```
   patch -i MSIM-GDB7.2.patch -p1 -u
   ```

3. Configure GDB for MIPS and build it. You might need the *termcap* library, which might be located in the *libncurses5-dev* package.

   ```
   cd gdb-7.2
   ./configure --target=mips
   make
   ```

4. Make sure that GDB is built properly by running `gdb-7.2/gdb/gdb`.

5. Now we will prepare Eclipse. Download another Eclipse. This time we will need Eclipse for C/C++ development and the version does not matter. Run it and create a new project from the existing code. Code is located in `gdb-7.2/gdb` and choose the appropriate toolchain (e. g. *Cygwin GCC* for the *Cygwin* platform).

6. Change the compilation option *-O2* to *-O0 -g* in `gdb-7.2/gdb/Makefile`. Rebuild the `gdb-7.2/gdb` sources.

7. Start debugging in Eclipse, choose *C/C++ Local Application* and then *gdb/mi debugger*. For *Cygwin* you might need to set the location of source files. The directory where your *Cygwin* is installed should work. Now you should be able to debug the patched GDB.

A.3   Workspace for MSIM

Now let us make MSIM working. We need a specific branch of MSIM, which is located on a *bazaar* based repository. Usage of different versioning system for each project is quite unfortunate. Hopefully, MSIM and MSIM plugin will use the same versioning system in the future. So far you will need a *bazaar* client for development of MSIM. Additionally, you might be required to have a *Launchpad* account for checkout of MSIM.

1. Get the specific branch of MSIM into the `workspace_msim`. The branch is located on the URL `https://code.launchpad.net/~fyzmat/msim-private-tm/trunk`. You might need to upload your public SSH key to *Launchpad*. The following command can be used for obtaining the branch:

```
bzr branch lp:~fyzmat/msim-private-tm/trunk
```

2. Configure the MSIM branch. Package *makedepend*, which can be located in the *imake* package, might be required. Use the command:

```
./configure
```

3. A problem with linking the *readline* library was encountered in *Cygwin*. You might need an installation of *ncurses* library and to add *-lncurses* to the *LIBS* variable in `src/Makefile`. Also, you might have to change the following lines in that makefile:

```
#original lines:
$(TARGET): $(OBJECTS) $(DEPEND)
    $(CC) $(CFLAGS) $(LIBS) -o $@ $(OBJECTS)

#changed lines:
$(TARGET): $(OBJECTS) $(DEPEND)
    $(CC) $(CFLAGS) -o $@ $(OBJECTS) $(LIBS)
```

4. Open Eclipse for C/C++ development and choose the `workspace_msim` as the workspace directory. Create a C project in the same way as for GDB. Uncomment the *DEBUG* line in `src/Makefile.local.template` and save it as `Makefile.local`.

5. When the Kalisto workspace is prepared, you might want to simulate your version of Kalisto during debugging of MSIM. In such a case add symbolic links to the `msim.conf` and the `kernel` directory into the MSIM `bin` directory.

6. After the previous step both MSIM and GDB can be tried. Run MSIM with the parameter *–remote-gdb=10001*, then run GDB and then write a GDB command *target remote :10001*. You should be able to read values of registers in GDB (the *info registers* command).

A.4   Workspace for Kalisto

1. Run the MSIM plugin and create the C project by going through *C Project -> Makefile Project -> Empty Project*.

2. Download Kalisto from `http://d3s.mff.cuni.cz/~ceres/sch/osy/main.php` and copy it to the directory of the created project. Refresh the *Project Explorer*.

3. For compiling on the *Cygwin* platform go to the *Preferences -> Build -> C/C++ Environment* and add the *PATH=;"C:\cygwin\bin"* environment variable. The path reflect the location of your cygwin installation.

4. For launching the debugging session by the key shortcut go to the *Preferences -> Run/Debug -> Launching* and check the *Always launch previously launched application* option.

5. Change the *CCFLAGS* from *-O2* to *-g -O0* in the `kernel/Makefile`.

6. For executing Kalisto in the current development version of MSIM, you can make a symbolic link in the root directory of Kalisto that points to a binary of MSIM in `workspace_msim`.

7. Finally you will be required to create a debug configuration. Open the *Debug configurations* dialog, create a new *C/C++ MSIM Application* configuration, fill in the debugged binary (usually `kernel.raw`), the path for the patched GDB and for MSIM. Now click on the *Debug* button and, hopefully, enjoy.

B. User manual

Users are supposed to understand debugging concepts in general. This manual helps with the initial setup and describes more important GUI elements of the Eclipse IDE.

B.1   Installation

The user needs to install three programs: the branch of MSIM that is developed in this work, patched GDB-7.2, and Eclipse with appropriate plugins. Distribution packages for MSIM and patched GDB are not available, because the MSIM branch is intended to be merged into the MSIM mainline before distribution. Instruction how to build these two programs are in sections A.2 and A.3.

Several problems has been encountered during creation of a standard distribution of Eclipse plugins (patched CDT and the MSIM plugin). Unfortunately, these problems has not been overcome yet. Therefore, the user is currently supposed to build patched CDT and the MSIM plugin and run it inside Eclipse as an *Eclipse Application*. One way to do so is using the instructions in A.1, but it is quite long and uncomfortable.

A much faster way of running the MSIM plugin is prepared. The user has a prepared workspace in the attached CD (`workspace_msim_plugin`). He is supposed to do the following steps:

1. Install *Java Development Kit*.

2. Install *Eclipse Classic 3.7*. The version is important and this instance of Eclipse is also located in the CD.

3. Choose the `workspace_msim_plugin` workspace as the active workspace after starting the Eclipse.

4. Establish an API baseline. The instructions are in the step 2 of the section A.1.

5. Clean the workspace and rebuild it. Versioning errors such as *@since* can be ignored.

6. Open the *Project Explorer* view, right click on the project *org.eclipse.cdt* and run it as an *Eclipse Application*.

B.2   Setting up a new project

Kalisto is intended to be the typical project. Instructions how to setup Kalisto are in the section A.4 of the chapter *Setting up the development environment*.

B.3   Debugging views

This section contains screenshots of important GUI views. The relevant GUI elements are marked and described.
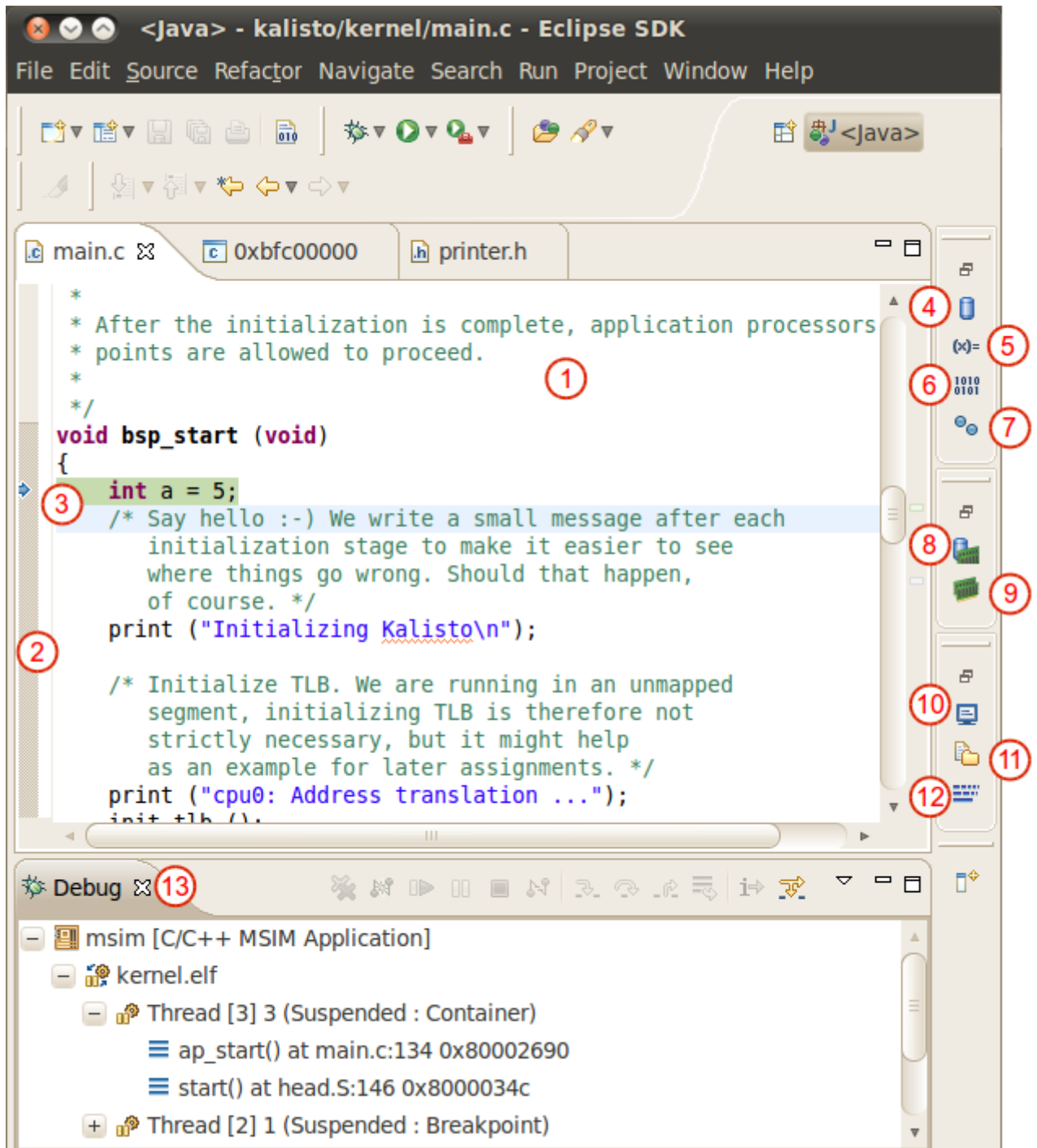
Views can be activated in the *Window -> Show view* menu.

Figure B.1: **Source view with other views in the side toolbar**.

	Name	Notes
1	Source view	
2	Place for breakpoints	Right click to open menu and toggle or enable/disable source-level breakpoints
3	Line where the debugged thread is stopped	
4	Memory Browser view	
5	Variables view	
6	Registers view	
7	Breakpoints view	
8	TLB Contents view	
9	Physical Memory view	
10	Console view	
11	Project Explorer view	
12	Disassembly view	
13	Debug view	

Table B.1: **Marked GUI elements of the view B.1**.
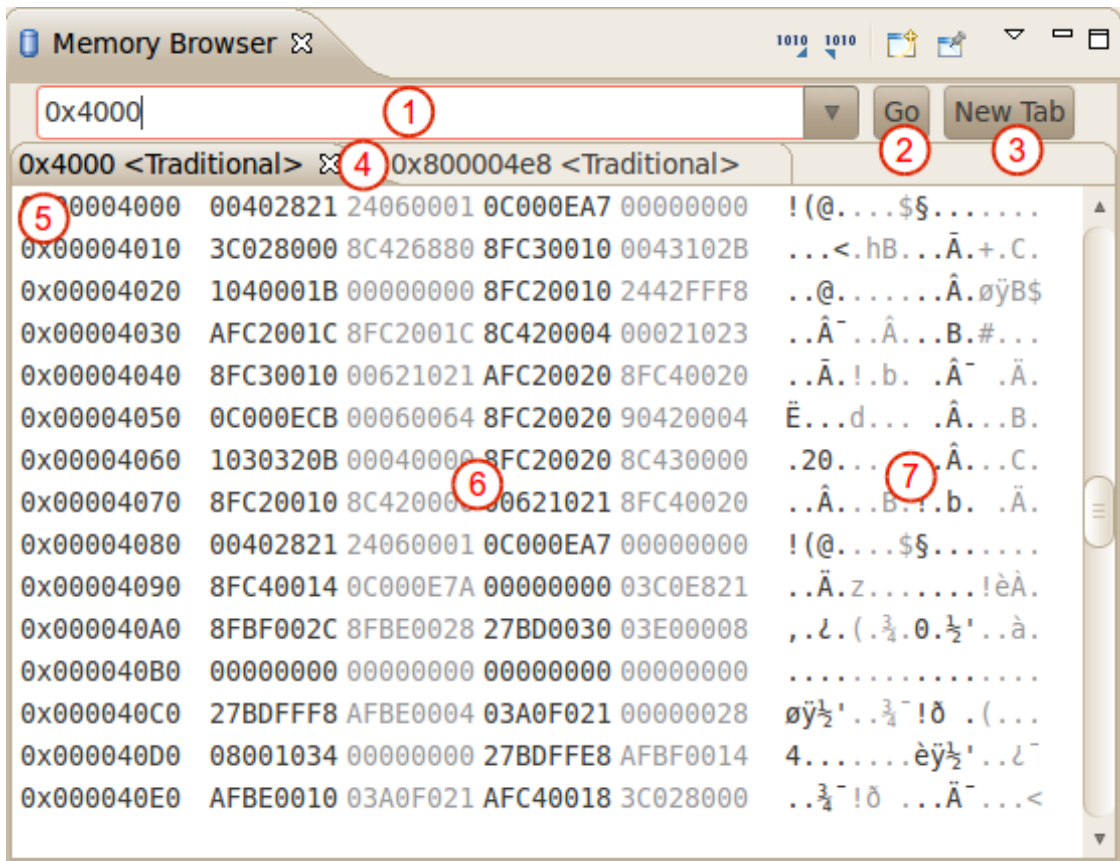
Figure B.2: **Memory browser view**.

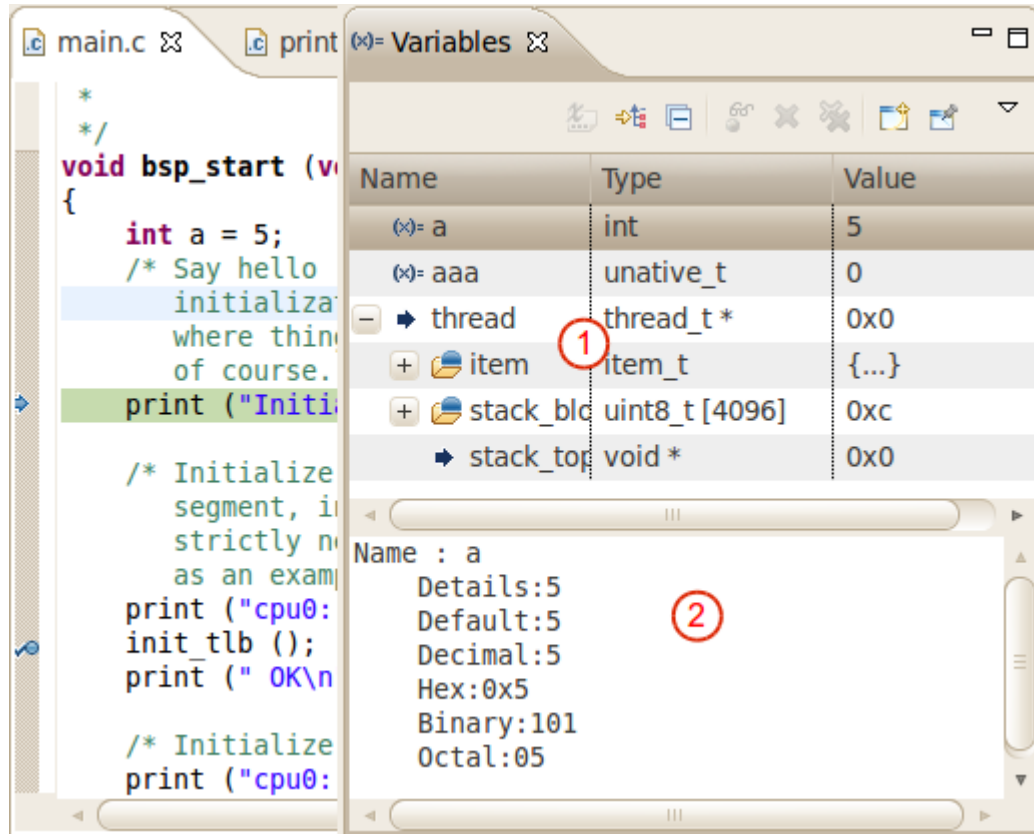|	Name	Notes
1	Expression input	You can type here any expression that specifies an address
2	Go to the specified address	
3	Create a new tab	
4	Tabs for browsing memory	
5	Address column of the tab	
6	Column with hex-dumped memory	It is possible to change the memory by writing desired hexadecimal values in this column.
7	Column with ASCII-dumped memory	It is possible to change the memory by writing desired ASCII chars in this column.

Table B.2: **Marked GUI elements of the view B.2**.

Figure B.3: **Variables view**.

	Name	Notes
1	Table with local variables	The rows contain an identifier, a type, and a value of the related local variable. The user can change values of variables in the last column.
2	Details for values	

Table B.3: **Marked GUI elements of the view B.3**.

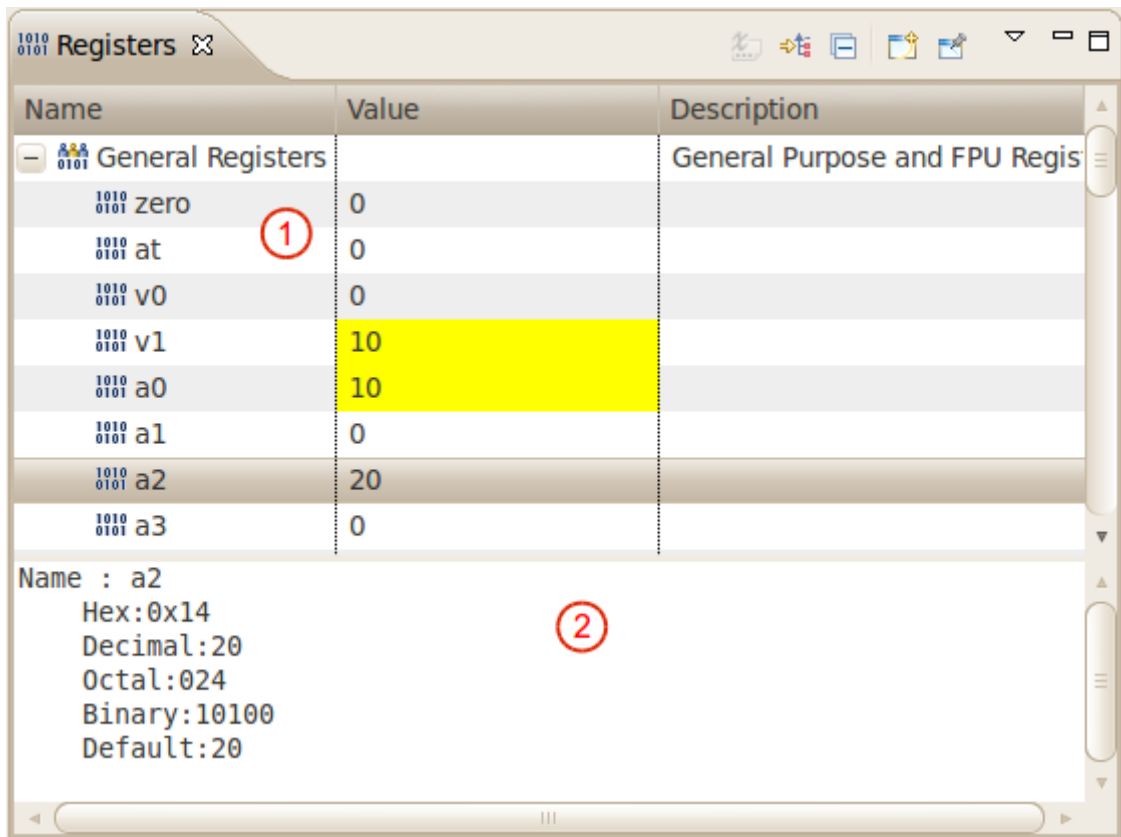|	Name	Notes
1	Table with registers	The rows contain a name of the register and its value. The user can change values of registers in the column with values. Changed registers are colored.
2	Details for values	

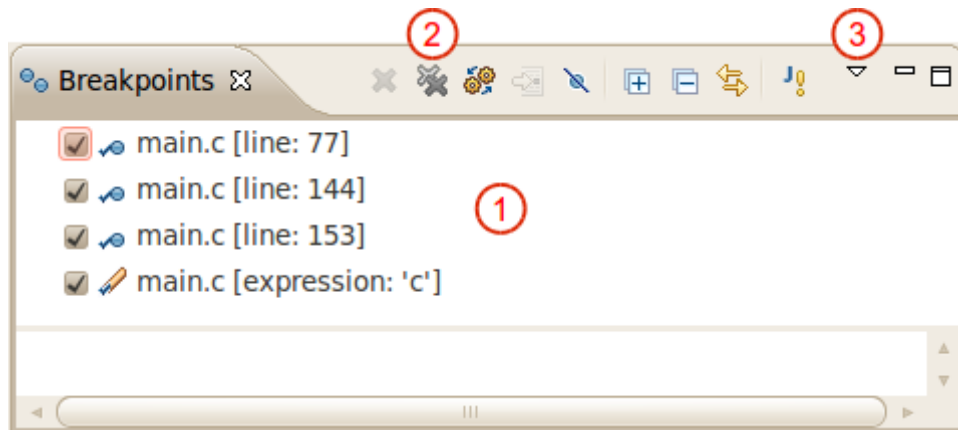Table B.4: **Marked GUI elements of the view B.4**.

Figure B.4: **Registers view**.

Figure B.5: **Breakpoints view**.

	Name	Notes
1	List of breakpoints	Both normal and memory breakpoints are listed.
2	Remove all breakpoints	
3	Additional options	The user can set a memory breakpoint in this menu. Another way of setting a memory breakpoint is via the menu *Run -> Toggle Watchpoint*.

Table B.5: **Marked GUI elements of the view B.5**.

Figure B.6: **TLB Contents view**.

	Name	Notes
1	Index column	Each row reflects translation of one page to one frame. Note that a TLB entry for the R4000 processor maps a pair of the following pages to two frames. The used way of displaying a TLB entry is separating it into two rows.
2	Page index	Address of a virtual page divided by size of one page.
3	Mask	TLB hit occurs when *virtual_address & mask == page_index*. This value is derived from the *PageMask* register. Bits 0-12 are always zeroes, bits 13-24 are set by the debuggee, and bits 25-31 are always ones. See [13] [p. 81] for more details.

4	Global bit	ASID is ignored if this bit is set. Note that this bit is never directly accessed by the debuggee. It is computed during a TLB write as logical AND of *EntryLo0* and *EntryLo1* global bits.
5	ASID	Address space identifier of the entry. Note that the current ASID is stored in the *EntryHi* register.
6	Valid bit	This bit is set if the page-to-frame translation of this row is enabled.
7	Dirty bit	This bit is set if the page is writeable.
8	Frame	Address of the translated physical frame.
9	Coherency bits	Three coherency bits. Not used in MSIM.

Table B.6: **Marked GUI elements of the view B.6**.
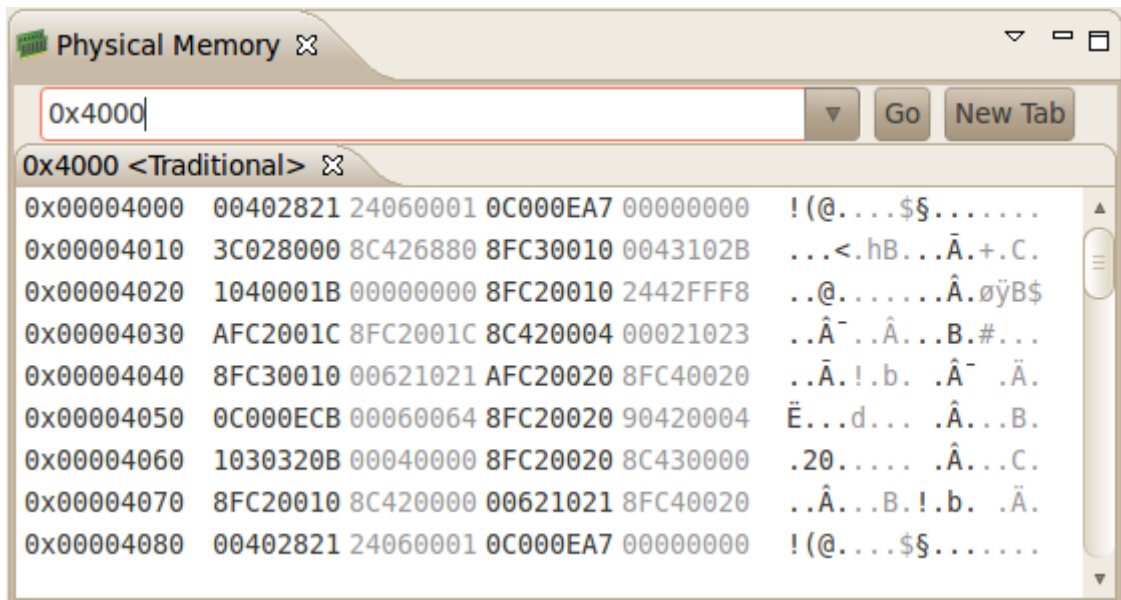
Figure B.7: **Physical memory view**. The usage is the same as for the *Memory browser view*
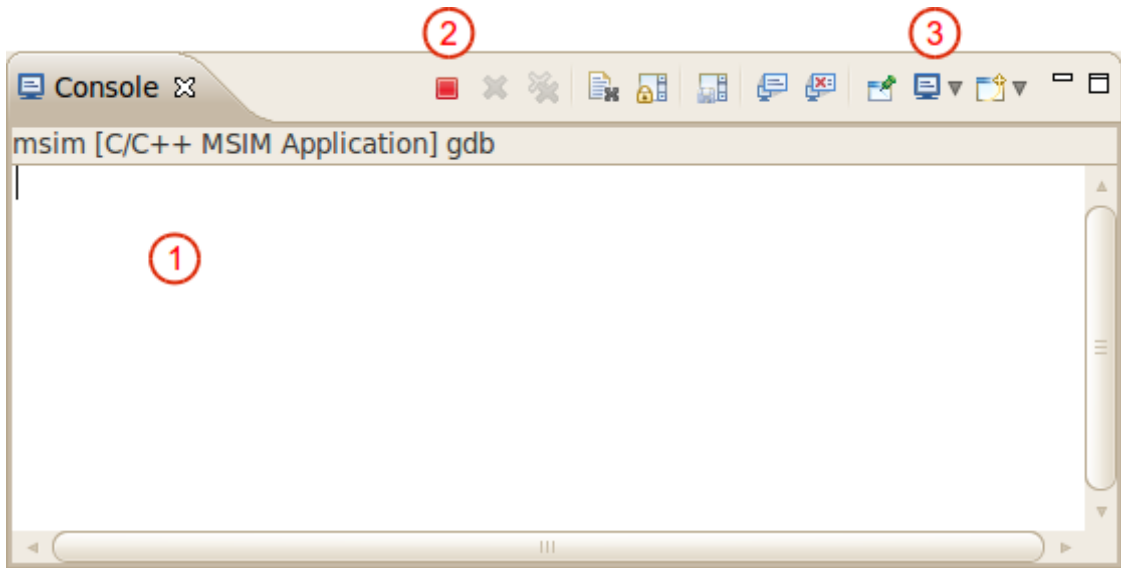
Figure B.8: **Console view**.

	Name	Notes
1	Console output	Useful for seeing MSIM output or GDB/MI communication.
2	Terminate debugging session	
3	Select another console	

Table B.7: **Marked GUI elements of the view B.8**.

Figure B.9: **Project explorer view**. The Kalisto project is currently loaded.
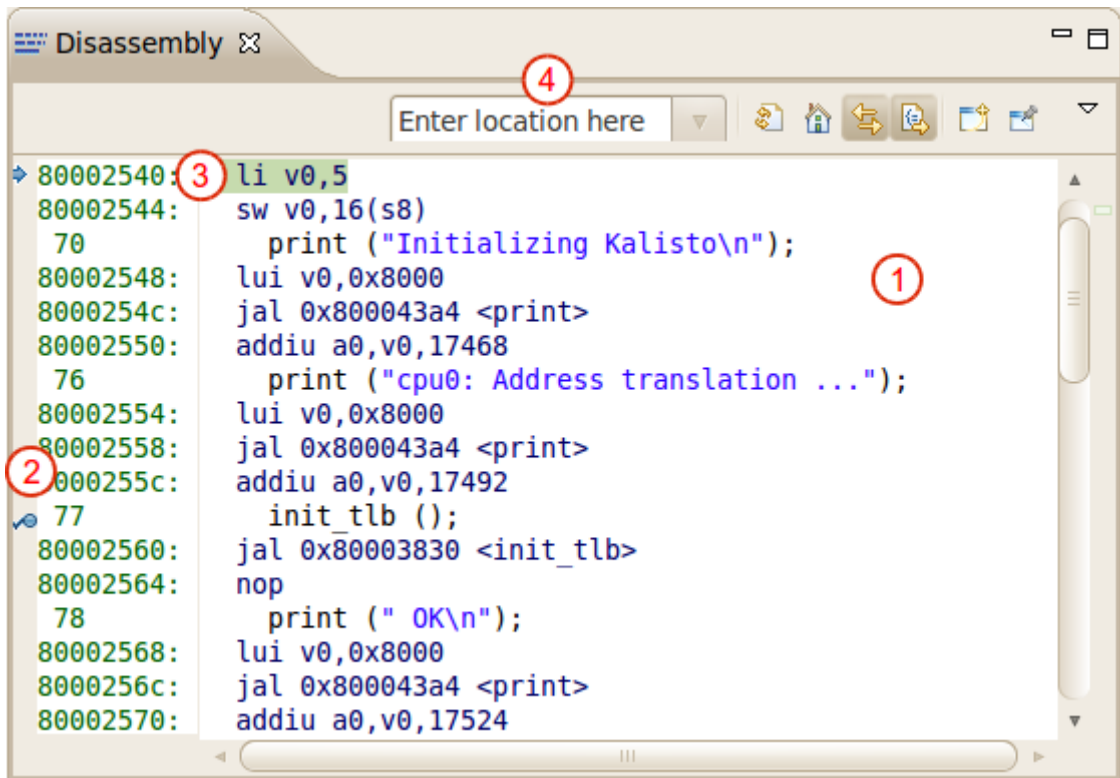
Figure B.10: **Disassembly view**.

	Name	Notes
1	Disassembly view	C source is merged into the instructions. The first column holds addresses of instructions or lines of the C code. The second column contains instructions and the C code. Symbols are added to the known addresses.
2	Place for breakpoints	Right click to open menu and toggle or enable/disable instruction-level breakpoints.
3	Line where the debugged thread is stopped	
4	Search input	Any expression that specifies an address can be typed here.

Table B.8: **Marked GUI elements of the view B.10**.
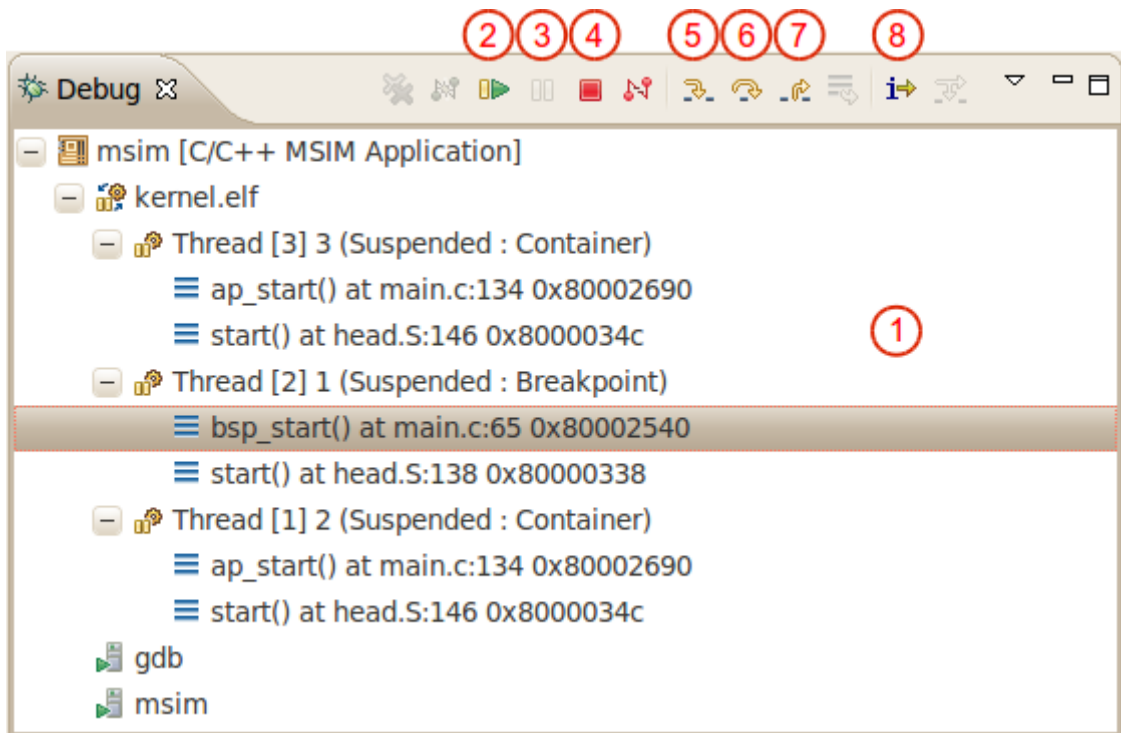
Figure B.11: **Debug view**.

	Name	Notes
1	Debug view	All the processes, their threads and call stacks are listed. The user selects the current thread by choosing it in this view.
2	Resume	
3	Interrupt	
4	Terminate debugging session	
5	Step into	
6	Step over	
7	Step out	
7	Toggle instruction-level debugging	This enables the user to do instruction-level stepping.

Table B.9: **Marked GUI elements of the view B.11**.

C. Summary of files for this work

Files that are needed for further development or deployment of this work are located in the attached CD. Additionally, most of them is available online. The tables C.1 and C.2 summarize locations of the files.

Description	Online URL
Eclipse Classic 3.7	Eclipse download sites
CDT-8.0	CDT download sites
GDB-7.2	GDB download sites
Kalisto	`http://d3s.mff.cuni.cz/~ceres/sch/osy/main.php`
Script for installing MIPS binutils	`http://d3s.mff.cuni.cz/~ceres/sch/osy/main.php`
Source codes of the MSIM branch	`lp:~fyzmat/msim-private-tm/trunk`
Source codes of the MSIM plugin	`https://svn.fenix-hosting.cz/repos/msim_debugger` login *public*, password *public*

Table C.1: **URLs of files for this work**.

Description	Path in CD
Eclipse Classic 3.7	`eclipse3.7.0-linux32` or `eclipse3.7.0-win32`
Workspace for the MSIM plugin	`workspace_msim_plugin.zip`
Workspace for MSIM	`workspace_msim`
Workspace for GDB	`workspace_gdb`
Runtime workspace for Kalisto	`runtime-EclipseApplication`
CDT-8.0	`cdt-master-8.0.0.zip`
GDB-7.2	`gdb-7.2.tar.gz`
Kalisto	`kalisto-0.8.8.tar.bz2`
Script for installing MIPS binutils	`toolchain.mips`
Source codes of the MSIM branch	`repos/msim-specific-branch`
Source codes of the MSIM plugin	`repos/msim-plugin`
Patch for GDB-7.2	`repos/msim-plugin/patches/MSIM-GDB7.2.patch`
Patch for CDT-8.0	`repos/msim-plugin/patches/MSIM-CDT8.0.patch`

Table C.2: **Contents of the attached CD**.

D. Getting familiar with the Eclipse platform

This appendix describes what makes development in the Eclipse platform difficult, but it can be also considered as a general observation of difficulties with orientation in source codes and complications during debugging. Many of the described issues are met during development of a larger project.

Writing plugins for the Eclipse platform is hard for a programmer who is not familiar enough with the platform. Amount of Eclipse source codes is huge and for a single person it demanding to remember deeper level of knowledge about all parts of Eclipse. For example, the implementation of the CDT plugin consists of 7000 java files. Studying documentation does not seem to be very overview giving. Most of the source codes is at least briefly commented. Especially valuable are comments at the beginning of a file that describes: what the class does; how it interacts with its surrounding; and briefly how it is implemented. Unfortunately these comments are often missing. With a bit of luck it is possible to find some information about concepts on web in form of wiki pages, presentations, ...

An unfamiliar developer can become more aware of how most of the mechanisms are designed by programming longer in the Eclipse environment or by cooperation with a familiar colleague. For a standalone programmer a suitable practice for getting familiar with writing Eclipse plugins is searching source codes where a similar thing to the desired one is implemented. Sometimes an example is available for this purpose. The programmer can also use a debugger for trying how the code behaves.

The IDE support is essential for efficient orientation in source codes. The IDE functions *go to declaration*, *find usages*, *show call hierarchy* and *show type hierarchy* are especially useful. However, the following list describes troublesome situations or factors that made the orientation difficult:

1. Search for the place of implementation

   In smaller projects it is commonly possible to guess the location of the searched implementation according to the name of classes. In case of the large CDT plugin this search is often tedious and can take hours to an unacquainted developer. An example is searching for a method that handles clicking the *Debug* button on the *Debug configurations* dialog. The fastest practice to find it seems to be searching the codes and configuration files for a text that is near the button on the screen.

2. Connecting plugins via the configuration files

   In the Eclipse platform the plugins declare theirs extension points and are linked with the extension points of other plugins. This mechanism realizes easy extensibility by new plugins. However, the mentioned IDE functions does not take this into consideration.

3. Invoking methods in another thread

There is a concept in parts of CDT determining that some code should be called asynchronously in a different thread. Sometimes this is unnatural, because there is no clear need for doing it. Such a code breaks logical structure of the call stack - it is more difficult to discover which code invoked an asynchronously executed method.

Fortunately most of the invocations follow the pattern D.1.

```
fSession.getExecutor().submit(new DsfRunnable() {
  public void run() {
    // ... asynchronously invoked code
    }
});
```

Figure D.1: **Pattern for calling asynchronous methods**.

Let us suppose that we would like to find out what methods called the asynchronous action when the debuggee hits a breakpoint in the action. Usually we would see that directly from the call stack, but in this case the last useful record in the call stack will be the *run* method of the submitted *DsfRunnable*. The workaround about this is to put the breakpoint on the *submit* call and debug the action again. Then we will see the upper records in the call stack.

4. Coding style that prevents fast use of the debugger

Preferred coding style varies from programmer to programmer and here is pointed out what has bad impact on debugging. From the debugging point of view it is unfortunate to use more than two or three dotted dereferences on the same line of code.

For example, finding out what leads to a raised *NullPointerException* on the snipped line D.2 may be work-intensive. Just splitting the line as illustrated in D.3 would help a lot.

```
String s = getManager().getConfig(type).getOptions().filter(
    prefs).get(key);
```

Figure D.2: **Too many dereferences in one line of code**.

```
Options opts = getManager().getConfig(type).getOptions();
String s = opts.filter(prefs).get(key);
```

Figure D.3: **Split line D.2**.

The second pointed style, which is not very debuggable, is a usage of complicated expressions in conditions like in the code D.4.

The snippet D.5 allows the programmer to put a breakpoint to the last line and immediately see which condition is evaluated in an unexpected way.

```java
private boolean isInProperPath(IOutput entry, String path) {
    if (entry.path().isPrefixOf(path) && !isExcluded(path, entry.
        getExclusionPattern())) {

        return true;
    }

    return false;
}
```

Figure D.4: **A complicated expression**.

```java
private boolean isInProperPath(IOutput entry, String path) {
    String outputEntryPath = entry.path();
    boolean isOnOutputEntry = outputEntryPath.isPrefixOf(path);

    String exclusionPattern = entry.getExclusionPattern();
    boolean isExcluded = isExcluded(path, exclusionPattern);

    return isOnOutputEntry && !isExcluded;
}
```

Figure D.5: **The improved expression D.4**.