Noname manuscript No.
(will be inserted by the editor)

Efficient Large-Scale Heterogeneous Debugging using Dynamic Tracing

Didier Nadeau · Naser Ezzati-Jivan · Michel R. Dagenais

Abstract Heterogeneous multi-core and many-core processors are increasingly common in personal computers and industrial systems. Efficient software development on these platforms needs suitable debugging tools, beyond traditional interactive debuggers. An alternative, to interactively follow the execution flow of a program, is tracing within the debugging environment, as long as the tracer has a minimal overhead. In this paper, the dynamic tracing infrastructure of GNU debugger (GDB) was investigated to understand its performance limitations. Thereafter, we propose an improved architecture for dynamic tracing on many-core processors within GDB, and demonstrate its scalability on highly parallel platforms. In addition, the scalability of the thread data collection and presentation component was studied and new views were proposed within the Eclipse Debugging Service Framework and the Trace Compass visualization tool. With these scalability enhancements, debuggers such as GDB can more efficiently help debugging multi-threaded programs on heterogeneous many-core processors composed of multi-core CPUs, and GPUs containing thousands of cores.

Keywords debugging · parallel · heterogeneous · multi-core · scalability · tracing · dynamic instrumentation · trace visualization · Trace Compass · GDB · LTTng

Didier Nadeau
Polytechnique Montreal
E-mail: didier.nadeau@polymtl.ca

Naser Ezzati-Jivan
Polytechnique Montreal
E-mail: n.ezzati@polymtl.ca

Michel R. Dagenais
Polytechnique Montreal
E-mail: michel.dagenais@polymtl.ca

1 Introduction

Heterogeneous computers systems with multi-core processors are becoming more popular in many fields. It allows a computer system to contain different processor architectures to benefit from the specific strength of each architecture. Furthermore, multi-core processors allow programs to use parallel computing and improve their performance. However, debugging programs on heterogeneous multi-core systems is an important challenge. The developer can be confused by the large number of threads in the program and have problems keeping track of its execution [16]. As the number of threads increases, it becomes harder for the developer to visualize and control the execution flow. One of the most important tools is the debugger, a tool to control a program execution and inspect its state. Unfortunately, most debugging tools have been in use for a long time, and were designed to debug single-threaded applications.

Many problems can be created by a mistake in thread synchronization. One example is a data race, which occurs when multiple threads try to access a shared variable without concurrency control. In this case, its final value depends on the order in which each thread accesses it, and the result may change between different executions as the thread scheduling varies. Inspecting this type of problem with a debugger can be problematic, as a debugger can be very intrusive and disturb the execution of the program. Therefore, a multi-threading issue may be masked or modified during debugging, greatly diminishing the usefulness of the debugger [13]. Thus, adapted debugging techniques are needed for parallel software.

Another challenge is the large amount of data to be handled by the user. Indeed, as parallel software may contain a great number of threads, there is a lot of information to display, and a large number of log events can be generated. This can confuse the developer by overloading him with information. Therefore, appropriate techniques to visualize and control multi-threaded software are needed.

There are debuggers that offer specialized features and views for debugging programs with multiple threads. TotalView and Allinea DDT [2] propose several views to group a large number of threads for efficient visualization and control. However, these programs are closed-source and their internal algorithms unpublished. One debugging environment, Eclipse CDT (C/C++ Development Tooling), is open-source and has some views adapted for multi-thread debugging. However, the support is still limited [16]. Tracing support is also limited in debuggers, and the tracing algorithms used may involve an important overhead. To our knowledge, the only open-source debugger that supports tracing is GNU Debugger (GDB).

In this paper, the above mentioned challenges have been addressed. The GDB debugger has been enhanced to allow dynamic scalable tracing in order to trace parallel programs with minimal overhead. This enhancement has been achieved by modifying GDB to address limitations for parallel tracing, and using LTTng for trace collection. LTTng has also been enhanced to allow dynamic event registration. Finally, the impact of conditional breakpoints has been measured, and a technique to minimize this impact has been proposed.

Furthermore, Eclipse CDT has been used to propose techniques for the user to interface with the debuggers. A view that automatically groups threads using their call stack has been created, and provides an efficient way for the user to visualize a large number of threads. Another view shows the threads currently executing on the graphic cards, and groups them using their position in the data grid sent to the GPU. Finally, a technique to filter threads executing on the CPU and the GPU has been proposed. This allows the developer to remove unneeded information from the debugging context and concentrate on relevant data.

In summary, the main contributions of this papers is twofold. First we propose a new architecture for low latency dynamic tracing using the GNU debugger (GDB). The second contribution of this paper is proposing a new interface to efficiently view, analyse and filter large number of threads in the open-source IDE Eclipse as an example of a large scale debugging tool.

The expected benefits are an improvement in the necessary time to identify and locate bugs in multi-threaded applications. It can be used to replace the use of logging message, as that requires the program to be recompiled each time. Thus, it is aimed at improving the performance of debugging tools and therefore at reducing development costs, specially for the applications that cannot easily be stopped for logging.

Our proposed solution has applications in complex multi-thread software debugging, including the telecom domain where each peer must be executed in a certain amount of time, thus cannot be stopped. It could also be debugging an application that communicates through network and cannot be stopped, otherwise it would stop sending heartbeats and the connection would fail. The provided user interface is aimed at programs that can be stopped, and to quickly identify which thread is working where.

The main objectives that we try to reach through this study are as to:

1. Identify a relevant parallel heterogeneous system to use for testing.
2. Study the performance of a parallel program controlled by a debugger, and identify the factors limiting it.
3. Evaluate the capabilities of GUIs to allow the user to debug easily on parallel heterogeneous systems.
4. Propose solutions to solve problems and limitations identified with debuggers and graphic interfaces.
5. Integrate these solutions with the free software GDB and Eclipse CDT and test their performance.
6. Present the advances to industry partners working in the field.

In this paper, in the related work section, several available techniques, to debug programs on large scale parallel systems, are discussed. An overview of available solutions for interactive debugging and for using tracing, as well as the limits of each method, are also presented. Then, the proposed contributions to improve interactive debugging in an open-source integrated development environment, and the proposed technique for dynamic tracing used in GDB, are described. Finally, the performance of the various contributions proposed, to enable large scale heterogeneous debugging using open-source tools, is evaluated.

2 Background and Related Work

The aim of this research paper is to improve open source debugging tool for multi-core architectures. In order to establish a baseline and find limitations of current tools, we provide a survey of relevant tools and research papers. Some of this survey is to present the most common tools used for debugging and tracing. We put an emphasis on open-source tools as these are easier to use, compare and improve. The debugging software presented was found through search in academical database, as well as through the recommendations of our industrial partners.

The industrial partners develop debugging tools to run on custom processors that have a large number of cores. These tools are aimed at real-time application, therefore in some case the developers might not be able to stop them to investigate an issue. This means the debuggers must have a minimal overhead on multi-core processors in order not to affect the real-time constraints. Furthermore, the industrial partners need to modify and recompile the tools so they can work on their custom processors. Therefore, they have a need for open-source debugging software the help the user debug programs on large multi-core systems.

The industrial partners provided some references they had already found. Furthermore, they recommended looking at the industry-focused conferences such as EclipseCon and the GCC Developer's Summit. In order to add to these initial articles, we searched in databases such as Web of Science and Ei Compendex for articles related to debugging, tracing, multi-core and open-source software.

2.1 Debugging

A program that runs a large scale parallel system is, by design, more complex that a single threaded one, as there must be multiple threads that interact together to benefit from the architecture [19]. This brings problems that are not present in single threaded program, such as a race conditions. Furthermore, it complicates debugging as certain threads might need other threads to work to be able to continue their execution. Stopping the program during interactive debugging may also mask a bug and make it impossible to find it this way. Thus, simply stopping a program and stepping a thread does not always work anymore [17].

There are several debugging tools available for complex platforms. TotalView and Allinea DDT are the two graphical debuggers most commonly used for high-performance computer debugging [2]. They have specialized support for multi-threaded and distributed debugging. These features are relevant in HPC computing such as Cuda, OpenMP and MPI debugging. WinDbg and the Visual Studio Debugger are closed-source debuggers developed by Microsoft and available only for Microsoft Windows. LLDB [14] is an open-source debugger developed as part of the LLVM project. It does not currently have as many features as GDB.

One common enhancement to facilitate large scale debugging is developing new views to conveniently show the user relevant information. These interfaces aim to show more essential information to the user without overwhelming him. This is extremely important on multi-core heterogeneous systems, since programs usually have very numerous threads to efficiently use their highly parallel architecture. Without specialized support, debugging can become inefficient, as the user will spend too much time understanding the state of the program. The proprietary debuggers, TotalView and Allinea DDT, both offer views that group threads based on various information such as location in the source code, state, etc [2]. Eclipse CDT has a multi-core visualizer that shows threads and their state based on their location on a specific processor core [16]. However, support for multi-core debugging in Eclipse is still limited, as there are few views designed for multi-threaded programs.

Non-stop debugging is a feature of GDB that was developed for multi-thread debugging. Traditionally, all the threads of a program are either running of stopped. When non-stop debugging is enabled, one or more threads can be stopped while other threads are still running. This allows the user to debug specialized systems where certain threads must continue to work, such as when time constraints must be satisfied or a watchdog timer must be handled [18]. Another important feature for debugging is conditional breakpoints. The thread stops when hitting the breakpoint only if a specific condition is met (e.g., a serious erroneous condition was met and the program can be stopped for diagnosing the issue). Otherwise, the debugger simply continues the thread execution and does not notify the user. This reduces the number of notifications received needlessly by the user and improves the debugging efficiency. It becomes even more useful for multi-thread debugging, when the user is more at risk of being overwhelmed by notifications due to the complexity of the program. However, the debugger must still handle the condition evaluation, when a breakpoint is hit, and context switches are involved. This can slow down the program if it is frequently hit.

2.2 Tracing

Tracing is a technique that aims to record a sequence of events that have happened in a program. This technique records and saves each event that occurred, instead of aggregating them and giving a summary, as a code profiler would. Having this full sequence of events can help the developer to investigate in depth an issue, and discover not only the presence of a problem but also its causes [21,23]. Tracing works by inserting tracepoints in the software code. Tracepoints call the tracer to record data when they are hit during execution. In order to give a realistic view of the program, it must minimize its overhead to limit perturbations [6]. Thus, the goal of a tracer is to have a low impact and achieve good performance for data collection.

There are two main instrumentation strategies for user-space software tracing. The first one is called static tracing, because it involves inserting code instrumentation into the program source code before compilation. This means that the tracing instrumentation is always present in the source code [10]. However, the instrumentation code is not necessarily active all the time and may simply be disabled during execution. This type of instrumentation is very efficient as the compiler can optimize the calls to the tracing functions. The main limitation of this method for debugging is the recompilation time required each time a tracepoint must be inserted. When the developers need to insert a new tracepoint while debugging, they have to modify the code and recompile it before continuing, restarting the debugging process.

There are various tools available to statically insert tracepoints into a program. LTTng-UST is the user-space tracer of the *Linux Trace Toolkit : next generation* tool suite. It uses a lockless Read-Copy-Update synchronization to ensure scalability on multi-core systems and optimize data collection [6]. UFtrace is a user-space tracer inspired from the main Linux kernel tracer. It works by compiling a program with option *finstrument-functions* to generate profiling code, and uftrace hooks to this instrumentation to measure metrics such as function duration [12]. Feather-Trace is a tracer with wait-free FIFO for multiple writers [4]. There are also programming libraries available to insert logging code for debugging, with various integrated development environments such as Visual Studio. These logging API are not necessarily tracers but their usage is similar for debugging.

On the other hand, dynamic tracing aims to modify a program dynamically. It is called dynamic because instrumentation is added to the program loaded in memory, after compilation, and does not involve source code modification [10]. Bypassing source code modification allows this kind of tracing to quickly instrument a program. However, its performance is not always as good as static tracing. This category of tracing can be subdivided into three types of tracing.

The first type of dynamic tracing works by replacing an instruction by a trap that is caught by the kernel. The kernel then redirects the program flow to the appropriate instrumentation code for tracing. The main drawback of this technique is the cost associated with the interruption handling, which significantly slows down the program being traced [10]. There are multiple available tracers that use this technique. GDB uses this type of tracepoints for its regular tracepoints [20]. UProbe is a feature of the Linux kernel that provides this kind of tooling and can be used directly, or as a back-end for other tracing tools such as Systemtap [5]. The cost associated with the context switches, from user-space to kernel, limits its use for low impact tracing.

It is also possible to use a virtual machine to interpret the program code at runtime and add instrumentation [10]. This method has an important overhead since it adds another step for instruction decoding. However, it can be very flexible and is used in Valgrind, a very popular tracer and profiler [24]. Valgrind has multiple functionalities such as call graph generation or memory leak detection. This technique is not used for the proposed architecture as its impact is too high.

The third dynamic tracing method replaces an instruction by a jump instruction to the instrumentation code, where the context is saved and the instrumentation is executed. Finally, the original instruction is executed and the program flow returns to the origin of the jump. This instrumentation technique has a very low overhead, similar to static instrumentation, as the cost of a jump is minimal [10]. However, it is limited since not every instruction is large enough to be replaced by a jump. For instance, on the Intel x86 architecture, the instruction must be at least 5 bytes long, or even longer if the jump target is beyond a certain distance. The fast tracing architecture in GDB uses this technique to insert tracepoints. This paper will use this technique as the proposed infrastructure is based on the fast tracepoint architecture in GDB. Furthermore, its cost is low enough to enable efficient tracing in multi-core systems.

Another important factor in tracing is the event collection infrastructure. The available tracers use various techniques to collect the data created by the tracepoints. It can simply involve a lock to access a buffer shared by multiple threads. More sophisticated synchronization methods, such as atomic instructions used by Ftrace [3], or lock-less FIFOs by Feather-Trace [4], for collection can offer a significant improvement in performance. However, these techniques are limited to fixed size events. The lock-less Read-Copy-Update synchronization mechanism used by LTTng provides both scalability on multi-core systems and flexible event sizes. Another aspect that impacts performance is trace data transfer. Many tracers save data in a buffer located inside the program memory and must empty them to allow further trace frames to be collected. A dedicated thread or program can be used to do this transfer in the background, or it can be done by a program also doing other tasks. These factors have an important impact on the performance and impact of the tracer.

2.3 User interface

To efficiently use tracing, the developer must have a way to understand the data. The simplest way to do so is simply to read the trace data using a text interpreter. This method is used by GDB to present the collected trace frames [20]. Babeltrace is a tool to convert traces generated by LTTng, or other compatible tracers, into text format. However, when the data collected reaches a certain size, it is no longer feasible for a human to read it all. Thus, various tools exist to provide views to display trace data [8]. Vampir [15] is a visualization tool focused on massively parallel computer systems. It provides multiple views to easily understand data such as a global timeline view, and a view that shows communication statistics. KCachegrind reads traces generated by the callgrind tool in Valgrind and give a graphical view to facilitate its understanding [24]. Chrome provides a trace viewer for trace data in its own format or for the ftrace data. Trace Compass is an Eclipse based visualization tool that provides various views such as a call graph or a timeline based view, for trace data generated by LTTng or other compatible tracers [22].

Conventional general-purpose processors found in modern computers frequently have multiple cores that each have their own instruction pointer and data set. This means that each core can execute a thread independently of the others. Processors dedicated for graphics use a different kind of architecture and multiple cores share the same instruction pointer. The execution unit in a GPU is a wave, and consists of a group of data items processed simultaneously on a group of cores with the same instruction pointer [11]. Thus, every core running the wave executes the same instruction concurrently on multiple data items. Due to this highly parallel architecture, graphics processors can achieve much higher calculation performance than general-purpose processors but are greatly impacted by conditional branches or concurrency control.

2.4 Heterogeneous System Architecture (HSA)

The HSA foundation is an organization involving academic and industry members that work on heterogeneous systems development. They promote a standard to provide a generic programming model for heterogeneous systems consisting of one or more generic host processors that dispatch work to agents [9]. These agents receive a function called a kernel and a three dimensions grid of data items. Each of these data items has an identifier in its work-group, called the local ID, and an absolute identifier, called the absolute ID. These identifiers are computed from the position of the data item in the grid. A work-group contains adjacent grid items dispatched together to a compute unit. Multiple compute units can be contained in a single agent. The compute unit executes the kernel function on multiple data items at the same time. The group of data items processed simultaneously by a compute unit is called a wave, and multiple waves can be part of the same work-group. A wave is similar to a thread using SIMD instructions, as an instruction is applied at the same time on every data item in the wave. This programming model is applicable for graphics processors, which can be used as agents in a system implementing the standard.

An open-source implementation of the HSA standard is currently being developed by AMD. This framework, called Radeon Open Compute, provides multiple components and allows shared memory space between the host processor and the graphics card. A version of GDB adapted for this software stack is also in development and currently able to interactively debug assembly code running on the graphics card [1]. This debugger still uses a small closed source legacy library for low-level work, that will be replaced to provide a completely open debugger.

In this section, a survey of available debugging and tracing tools for heterogeneous systems has been presented with a focus on open-source tools. Various methods and features to aggregate information and help the user to efficiently debug multi-threaded programs have been detailed. Using tracing in a debugger, where the impact on the program must be kept minimal, and different techniques to do so, have been discussed. The general programming model used for graphics processors in the HSA Foundation standard have been outlined, along with an open-source implementation of the standard. In the following sections, the current features and performance of GDB and Eclipse are analyzed. Then, the contributions to debugging tools are explained and their impact is analyzed.

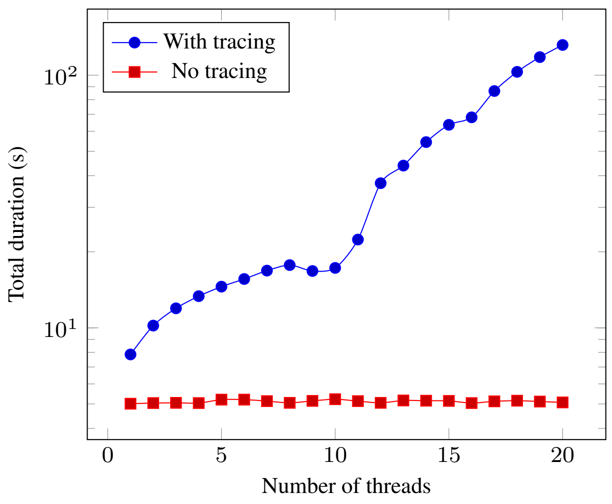


Fig. 1: Total execution time for a test program as a function of the number of threads when a fast tracepoint is inserted

2.5 Limitations of Current Tools

In this section, the Eclipse CDT support for multi-threaded debugging, and the various views available, have been studied. The current support for fast tracing in GDB has been tested and its performance has been evaluated.

The performance of GDB fast tracing has been measured using benchmarks, and user-space and kernel tracing. GDB fast tracing was used on a multi-threaded test program where the tracepoint is very frequently hit, while changing the number of threads in the program. Three experiments were conducted, to measure the execution time, to capture kernel events, and to record access to the GDB tracing function.

The benchmark executes the test program repeatedly and the tracepoint is frequently hit in the main loop of the program by every thread. The script measures the total execution time as the number of threads is varied. Figure 1 shows the results of this test. The total execution time goes from 7.85 seconds for 1 thread to 131.68 seconds for 20 threads. The tracing impact is clear, as the program execution is approximately 17 times faster for 20 threads when there is no tracepoint. Furthermore, the program is significantly faster even for only one thread, going from 5.01 seconds without tracepoint to 7.85 seconds, for a 36 % overhead.

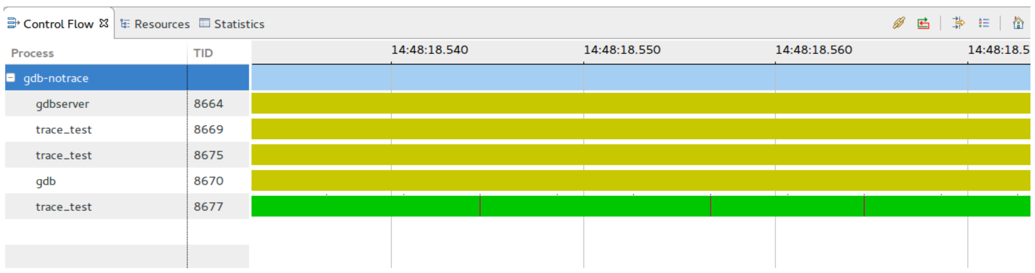


Fig. 2: Representation of program state during normal execution

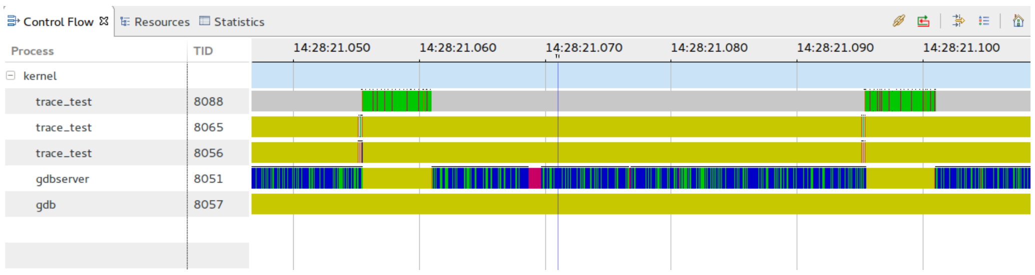


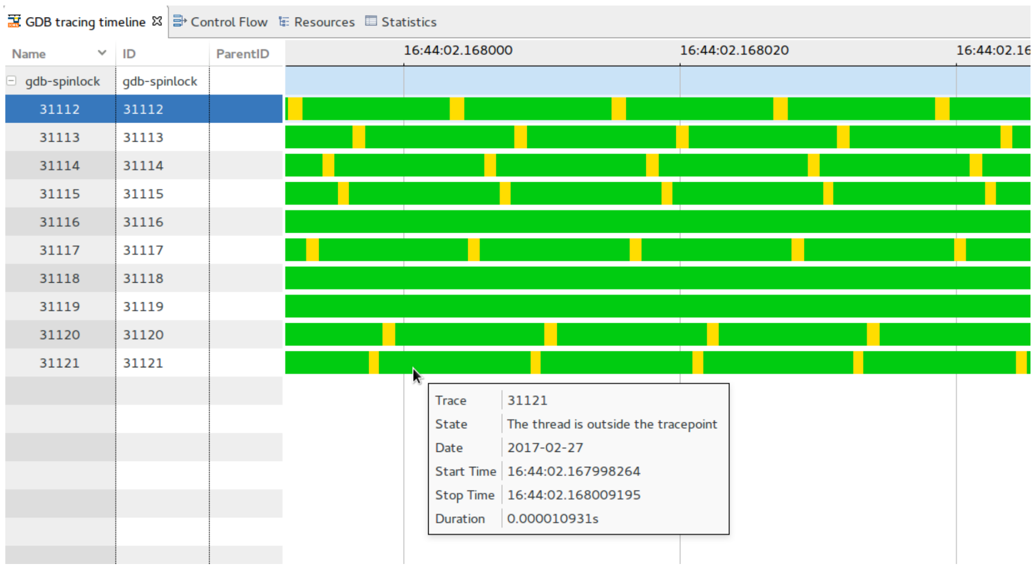
Fig. 3: Representation of program state while tracing it with the default GDB fast tracing architecture

Kernel tracing records kernel events and system calls. It gives the state of each thread during execution and provides a good insight into issues such as resource starvation. Trace Compass, an open-source Eclipse plugin program, is used to visualize the kernel trace collected when GDB is tracing a program. Figure 2 represents the test program executing in GDB without tracing. Each line is a different kernel thread, and the one with TID 8677 is the test program executed with only one thread. This thread stays in the same state, executing, for most of its duration, as it simply does floating-point calculations in a loop. Figure 3 represents the test program when fast tracing is enabled in GDB. In Figures 2 and 3 yellow and red show the blocking states, green depicts the running in the user-space mode while blue represents the system-call execution mode. As shown in Figure 3, the thread doing the calculation is the one with TID 8088 and its state changes multiple times. It stays most of the time blocked, as shown by its grey color, while thread 8051 of GDBServer is transferring the data. These two figures show the high penalty incurred when the program is stopped to flush the trace buffer, which explains why the execution time is higher, even for only one thread, when GDB tracing is enabled.

The last test to analyze the behavior of the tracer is recording the access to the tracing code in GDB. The collection function code, in the In-Process Agent library, is instrumented using LTTng-UST. This collection function is called whenever a standard GDB fast tracepoint is hit. One event for function entry and another for function exit are defined. The test program is started with 10 threads using GDB, and a fast tracepoint is inserted into its calculation loop. Then, access to the trace collection function in GDB is recorded for each thread using LTTng-UST.

Figure 4 shows the trace data in a Trace Compass view. A custom view was defined using the XML plugin. This view shows one line per thread. Each line is either in the *normal* state or in the *tracing* state. The mouse pointer in Figure 4 points to a thread in the *normal* state. Underneath the mouse pointer, we can see the notification window shown when the mouse points to a thread in the *normal* state, i.e., outside the tracepoint. The other state, in yellow, indicates that the thread is in the collection function. Figure 4 shows that there is only one thread in the *tracing* state at a time. This implies that there is a critical section in the tracepoint and mutual exclusion applies. A review of the jump pad code, that calls the collection function, reveals that there is a spin lock inside the jump pad. Therefore, only one thread can have this lock at a time, which explains why there is never more than one thread in the *tracing* state. Furthermore, we can see that there is almost always one thread that is in the tracing state. This implies that there is at least one thread waiting for the lock most of the time, and that it goes into the collection function as soon as possible. This lock is necessary, as there is a single buffer where every thread stores its trace data. However, it greatly limits scalability and is the most important contributor to the performance hit shown in Figure 1 when the number of threads is increased.

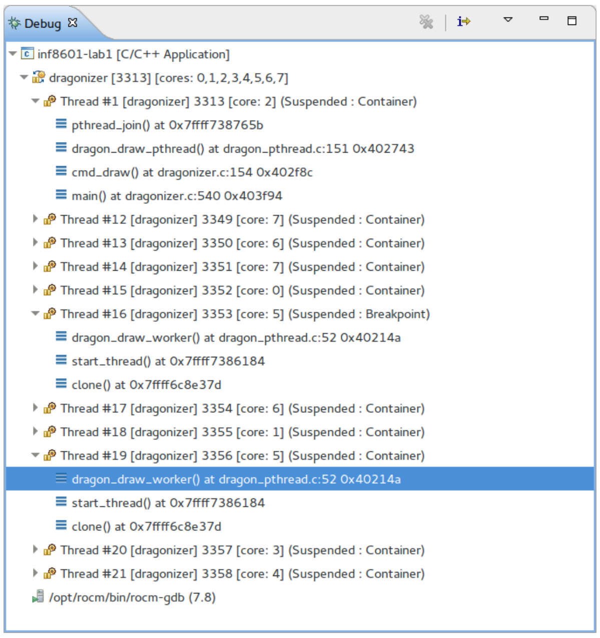
The support to debug code running on general purpose graphics processors in GDB is limited. Nvidia offers a debugger based on GDB to debug CUDA code on its graphics processors. This debugger is based on CUDA, which is closed-source and limited to Nvidia graphics processors.



Name	ID	ParentID
gdb-spinlock	gdb-spinlock	
31112	31112	
31113	31113	
31114	31114	
31115	31115	
31116	31116	
31117	31117	
31118	31118	
31119	31119	
31120	31120	
31121	31121	

Trace	31121
State	The thread is outside the tracepoint
Date	2017-02-27
Start Time	16:44:02.167998264
Stop Time	16:44:02.168009195
Duration	0.000010931s

Fig. 4: Timeline of threads state when a tracepoint has been inserted. Green is tracing and yellow is the normal execution.



```
inf8601-lab1 [C/C++ Application]
  dragonizer [3313] [cores: 0,1,2,3,4,5,6,7]
    Thread #1 [dragonizer] 3313 [core: 2] (Suspended : Container)
      pthread_join() at 0x7ffff738765b
      dragon_draw_pthread() at dragon_pthread.c:151 0x402743
      cmd_draw() at dragonizer.c:154 0x402f8c
      main() at dragonizer.c:540 0x403f94
    Thread #12 [dragonizer] 3349 [core: 7] (Suspended : Container)
    Thread #13 [dragonizer] 3350 [core: 6] (Suspended : Container)
    Thread #14 [dragonizer] 3351 [core: 7] (Suspended : Container)
    Thread #15 [dragonizer] 3352 [core: 0] (Suspended : Container)
    Thread #16 [dragonizer] 3353 [core: 5] (Suspended : Breakpoint)
      dragon_draw_worker() at dragon_pthread.c:52 0x40214a
      start_thread() at 0x7ffff7386184
      clone() at 0x7ffff6c8e37d
    Thread #17 [dragonizer] 3354 [core: 6] (Suspended : Container)
    Thread #18 [dragonizer] 3355 [core: 1] (Suspended : Container)
    Thread #19 [dragonizer] 3356 [core: 5] (Suspended : Container)
      dragon_draw_worker() at dragon_pthread.c:52 0x40214a
      start_thread() at 0x7ffff7386184
      clone() at 0x7ffff6c8e37d
    Thread #20 [dragonizer] 3357 [core: 3] (Suspended : Container)
    Thread #21 [dragonizer] 3358 [core: 4] (Suspended : Container)
  /opt/rocm/bin/rocm-gdb (7.8)
```

Fig. 5: Main Eclipse CDT view to shows threads in the Debug perspective

AMD is working on a version of GDB to support code running on its graphics processors using standards of the Heterogeneous Systems Architecture (HSA) foundation. This version of GDB is based on the Radeon Open Compute environment and is called ROCm-GDB. The current version of ROCm-GDB still uses an old closed-source library to support GPU debugging but it will be replaced with an open-source version. Furthermore, the programming model used is an open standard and could be applied to devices from other vendors.

Support for debugging code running on heterogeneous and multi-core platforms is limited in open-source development environments. Nvidia offers a version of Eclipse, Nvidia Nsight, to develop and debug code running on its graphics processors using CUDA. Nsight is open-source as it is based on Eclipse but it uses CUDA and has the same limits as CUDA-GDB. Both TotalView and Allinea DDT offer various methods and views to deal with a large number of threads, but both of these programs are closed-source.

Eclipse CDT offers support for multi-threaded debugging. The main debug view shows each thread of the program being debugged in an expandable tree, with the parent processes and the stack of each thread. Selecting a thread in this view allows the user to get more information on its context, such as registers and local variables values. Eclipse CDT also offers support for non-stop debugging, or reverse debugging, when used in combination with a version of GDB that allows it. The multi-core visualizer is a debugging view that shows CPU cores and the threads running on them. It gives an overview of the state of the program to the user, and can display more threads that the main debug view.

The scalability of the main debug view is limited. Indeed, as it represents a list of threads that can be expanded to see their stacks, it can be difficult for the debuggers to control a large number of threads. The stacks must be expanded to know where a thread is stopped, but this mean that there are multiple elements shown for each thread. When there are many threads, the view quickly becomes full. For example, as shown in Figure 5, the number of elements makes it hard to see where each thread is. Furthermore, the screen is almost full with 11 threads shown, and we see the stack trace of only 3 of them. This forces the user to scroll or collapse elements and makes it harder to keep track of the program state.

Code executed on the graphics processors using the HSA specifications consists of a function applied to each element in a three dimensions grid. The GPU executes the function in waves, and each wave contains multiple units, that are at the same step in the function but are applied to different elements of the grid. A typical graphics processor can contain significantly more than a hundred waves executing simultaneously. Therefore, these waves can not realistically be displayed in the same way as CPU threads, because it would be inefficient for the user to deal with a list of hundreds of elements.

In conclusion, Eclipse CDT has support to allow the developer to debug multi-threaded programs on a processor. However, it needs significant enhancements and new features to be able to do so efficiently on large multi-core platforms. Support for graphics processors debugging is not present at the moment. Furthermore, the fast tracing architecture used in GDB has issues that strongly limit its performance on multi-threaded programs. In the next section, contributions to both Eclipse CDT and GDB will be proposed in order to optimize debugging on heterogeneous multi-core systems.

3 Proposed Solution

As heterogeneous multi-core systems become increasingly used, debugging tools must be enhanced for these systems. In order to maximize productivity, tools must follow this trend. Many aspects of debugging need improvements to facilitate this process. In the following sections, we detail our work on dynamic tracing using GDB and enhancing interactive debugging using Eclipse.

3.1 Dynamic Tracing

Many tracers and loggers work by providing functions called in the source code and require recompilation to be included in the program. On the other hand, dynamic tracing works by modifying the program in memory, after compilation, in order to instrument it. As explained in the related work section, there are various techniques used to achieve this goal. A first technique inserts a breakpoint and the handler redirects the execution flow to the instrumentation code. It is also possible to interpret the code instead of executing it, thus allowing the interpreter to insert instrumentation on the fly. However, these techniques have an important overhead as the first method requires interruption handling by the operating system, while the second one needs to regenerate instructions before executing them [10]. Another option, called fast tracepoints in GDB, involves modyfing the program in memory to jump to the instrumentation code. This method has a significantly lower overhead than the previous techniques [17].

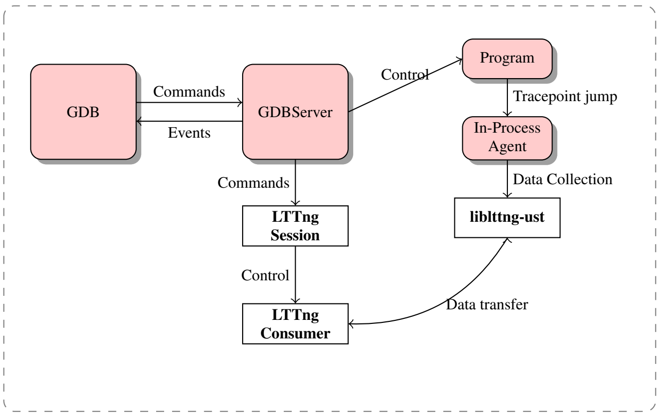


Fig. 6: Proposed fast tracing architecture using LTTng

In this paper, large-scale is used to describe systems that are strongly parallel and have a large number of cores. There is no exact limitation on what constitutes a large number of cores, but it implies that there are enough cores to bring concurrency challenges. Large multi-core platforms include multi-processor computers or specialized many-core processors such as the Xeon Phi, which can have more than 60 cores. For this paper, the computer used for testing has 4 processors with 16 cores each, for a total of 64 cores. The graphics processor used for GPU debugging is the AMD R9 Nano with 4096 processing units, using the Radeon Open Compute stack 1.1.0.

Previously, tracing only involved GDB, GDBServer and the In-Process Agent library. The user interacted with GDB that sent commands to GDBServer. The In-Process Agent library was preloaded into the debugged program and contained a buffer accessed by the threads, in mutual exclusion enforced by a spinlock. Once the buffer was full, the program was stopped and GDBServer transferred the data into its own buffer before restarting the experiment.

The proposed architecture offers performance improvements by using the LTTng tracer developed by our research group (Figure 6). It has a more sophisticated tracing infrastructure that avoids the penalty caused by locking and stopping. The tracer creates one ring buffer per processor core in a memory space shared between the liblttng-ust library and the consumer daemon. Each buffer contains multiple sub-buffers, and the size and number of these sub-buffers can be modified. Atomic instructions are used by the producer and the reader to keep track of the sub-buffer being currently read and the one being written to [7]. Furthermore, the producer uses local atomic instructions that only affect its core and not the whole processor to minimize overhead. The consumer uses a standard compare-and-swap instruc-

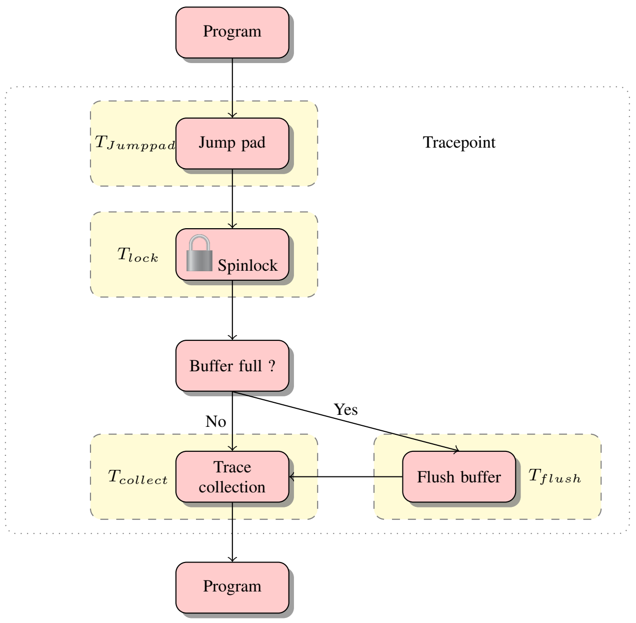


Fig. 7: Standard fast tracepoint workflow

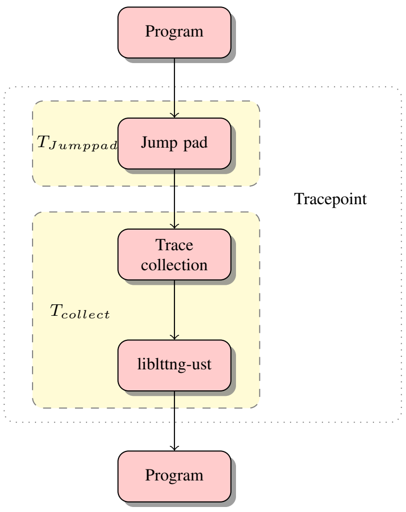


Fig. 8: Proposed LTTng tracepoint workflow

tion, although its overhead is limited because it only uses it once a whole sub-buffer has been read. This allows multiple threads to write trace data simultaneously, and does not require to stop the program to empty the buffer.

Furthermore, we have extended the LTTng tracer to enable dynamic tracing. One limitation of the standard LTTng-UST tracer is that events must be defined before compilation. We propose a method to circumvent this limitation, as we need to dynamically register tracepoints. We leverage the CTF trace format used by LTTng that store the trace definition in a single header file. We define multiple tracepoint at runtime that take char array of various size at runtime. When the user inserts a fast tracepoint in GDB, we calculate the total size needed for the desired tracepoint. Then, GDB selects a tracepoint that has a char array of sufficient size and links it by dynamically modifying the program. When the experiment is completed, we modify the trace definition in the header, replacing the char array argument by multiple arguments of the correct type. This method allows us to use the standard LTTng tracer packaged by many Linux distribution. Thus, usage is simplified as a developer can install the standard package from an available repository.

Tracepoint insertion is handled by GDBServer. It first checks if there is a large enough instruction, at the line where the user wants the tracepoint. On x86 processors, the instruction must be at least 5 bytes long. GDBServer replaces this instruction by a jump to a pad where the context is saved and the tracing function of the In-Process agent is called. The original instruction is executed at the end of the jump pad when the tracer returns.

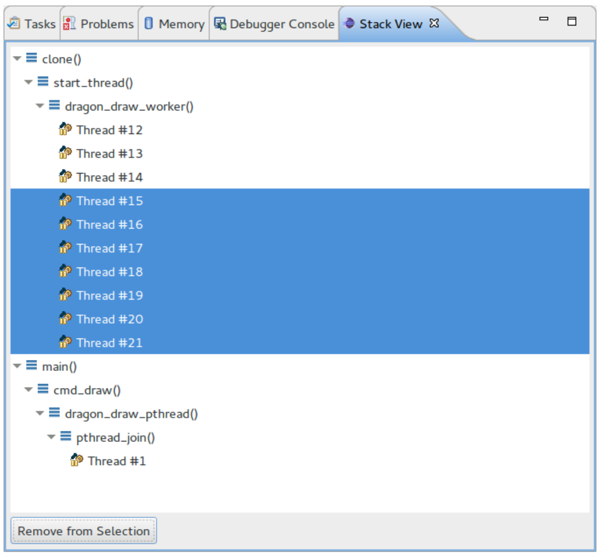
Figure 7 shows the workflow for the original tracing technique. Once the thread is in the jump pad, it waits until it can acquire the spinlock. Then, it goes into the tracing function and reserves space in the buffer. If there is not enough space, it triggers a breakpoint to tell GDBServer to empty the buffer. After GDBServer has restarted the program, the thread saves the data, releases the lock and executes the original instruction before returning to the instruction following the inserted jump.

$$T_{tracepoint} = T_{jumppad} + T_{lock} + T_{collection} + T_{flushing} \quad (1)$$

The time penalty imposed by the default fast tracepoint implementation is given in equation 1. The values of $T_{jumppad}$ and $T_{collection}$ are constants with respect to the number of threads. The value of $T_{jumppad}$ comes from the time it takes to jump to the tracepoint instructions and save the context. The time taken by $T_{collection}$ comes from saving values into the buffer. The component $T_{flushing}$ is affected by the total number of trace frames collected, as the data transfer from the In-Process Agent buffer to the GDBServer buffer occurs only when enough trace frames have been collected for the In-Process Agent buffer to be full. T_{lock} is the component that strongly limits scalability. Indeed, it represents the time spent trying to acquire the spinlock. As the number of threads increases, the tracepoint is hit more frequently and an increasing number of threads try to acquire the lock at the same time. This causes a large penalty, as multiple threads are stuck trying to acquire the spinlock.

Figure 8 represents the workflow when a tracepoint is hit in the proposed architecture. First, the context is saved by the jump pad, as for standard fast tracepoints. Then, the data is collected and sent to an appropriate tracing function in the *liblttng-ust* library. Finally, the thread returns to the program.

$$T_{tracepoint} = T_{jumpad} + T_{collection} \quad (2)$$



```
clone()
  start_thread()
    dragon_draw_worker()
      Thread #12
      Thread #13
      Thread #14
      Thread #15
      Thread #16
      Thread #17
      Thread #18
      Thread #19
      Thread #20
      Thread #21
main()
  cmd_draw()
    dragon_draw_pthread()
      pthread_join()
        Thread #1
```

Fig. 9: Stack Aggregation view implemented within Eclipse CDT to display active threads

Equation 2 gives the time penalty of a fast tracepoint using LTTng. The first component, T_{jumpad}, is the same as in the default implementation. The second component, $T_{collection}$, is extremely similar to the first, as it only uses a different function call to transfer data. It still is constant, unaffected by the number of threads. Both T_{lock} and $T_{flushing}$ are removed for LTTng tracepoints, as there is no lock needed for data flushing.

The proposed implementation brings performance enhancements by using a combination of the GDB dynamic tracing architecture and the LTTng tracer developed by our research group. GDB dynamic tracing enables the developer to insert tracepoints using a simple jump, a few instructions and a call to a function. The cost of this solution is lower than for dynamic tracing using a breakpoint that triggers interruptions. However, the standard GDB fast tracing uses a single lock-protected buffer in the library to save the data. Furthermore, when the buffer is full, the program is stopped by GDBServer to transfer the data and empty the buffer. Both of these factors contribute to slow down a program traced on a large-scale parallel system. The proposed implementation enhances the existing fast tracing architecture by removing these two limitations. By using circular buffers in shared memory to store data from tracepoints, no lock is needed to access the buffers and the program is not stopped while the consumer daemon transfers the data. This allows tracing to scale well when the number of threads is increased.

```
function BUILDTREE($CallStacks$)
    $TreeNode$
    $i \leftarrow 0$
    while i < CallStacks.size do
        APPENDTREE($TreeNode, CallStacks[i]$)
        $i \leftarrow i + 1$
    end while
end function

function APPENDTREE($TreeNode, head$)
    if head == null then return
    end if
    $child = TreeNode.find(head)$
    if child ≠ null then
        APPENDTREE($head.next$)
    else
        $TreeNode.insert(head)$
        APPENDTREE($head.next$)
    end if
end function
```

Fig. 10: Algorithm to build the tree structure for the call stack view.

3.2 Visual Debugging

The first view, implemented to help the user debug a program with a large number of threads, is a call stack aggregation view and is shown in Figure 9. This view automatically groups the threads by their call stacks to display important information in a limited space to the user. The call stacks of each thread are retrieved and merged together to create a tree. Therefore, two threads that have the same call stack will be shown in the same leaf of the three. If their call stacks have the same first functions but diverge at some point, this will be shown in the view. This user interface shows groups of thread that are likely to do related work, as they have a similar call stack. It helps the developers to focus on specific parts of the program, as they can collapse parts of the tree, if the threads in this section are doing work they are not interested in. Furthermore, the threads are logically grouped together, which should help the users to better understand the program work flow, as compared to the standard debug view that simply provides a very long, linear, list of threads.

The intended use of this call stack view is multi and many-core systems running programs with a large number of threads. Therefore, we need a scalable algorithm to build a tree structure for the viewer from the call stacks. TotalView provides a similar interface, however the algorithms to build the data structure are unpublished. Figure 10 shows the algorithm to build the tree. It takes an array of pointers to the head of each thread call stack. The *AppendTree* function is called only once for each stack frame. Furthermore, we have chosen to use an hash table to contain a node children. It ensures *O(1)* complexity to insert or find a children. This gives the algorithm to build the tree a total running time of *O(kn)*, where k is the number of thread and n the average call stack depth. This ensures scalability on multi-threaded programs, as the running time is limited by the total number of stack frames read by the algorithm.

To further improve efficiency of multi-threaded debugging, we propose a filter to be applied on threads. Even with specialized views, overwhelming the developer is still a possibility. The large number of simultaneously active threads increases the frequency of events such as breakpoint hits. Multi-threaded programs running on large systems can contain multiple groups of threads that are doing separate work. Nonetheless, the user could be interested in debugging only one of of these groups. However, some functions can be shared between the groups to reuse code, and inserting a breakpoint can lead to unneeded notifications, complicating the debugging process. While it is possible to use conditional breakpoints in these cases, the user has to manually specify the list of threads concerned for each breakpoint. Using a filter, the users can select the threads they needs to debug, and the filter removes the other threads from perspective. The developer is not notified of breakpoint hits for threads not included in the perspective, the threads are simply resumed. Furthermore, the filter removes unselected threads from the different debugging views. This enables the developer to focus completely on the problematic parts of the program, without seeing the other threads or receiving unneeded notifications.

Even with specialized views, overwhelming the developer is still a possibility. The large number of simultaneously active threads increases the frequency of events such as breakpoint hits. Multi-threaded programs running on large systems can contain multiple groups of threads that are doing separate work. Nonetheless, the user could be interested in debugging only one of these groups. However, some functions can be shared between the groups, to reuse code and inserting a breakpoint can lead to unneeded notifications, complicating the debugging process. To solve this issue, we propose a filter concept inspired from conditional breakpoints. These breakpoints are only triggered if a specific condition is met, thus using a filter to reduce the number of notifications. We generalize this concept by filtering the threads themselves. The filtering not only applies to breakpoints but also to views, and threads that are not in focus would not be shown. This enables the developer to focus completely on the problematic parts of the program, without seeing the other threads or receiving notifications.

Displaying GPU waves is a difficult challenge. As previously outlined, a graphics processor can run more than a hundred waves simultaneously, and each of these waves executes a kernel function on multiple data items. The large number of elements to be displayed simply cannot be shown in a list, as it would be almost unusable. Therefore, a method to automatically group the waves and reduce the displayed information must be used. A call stack aggregation view, as proposed for CPU threads, is not possible. Indeed, functions calls are inlined on the GPU and there is no call stack. However, it is possible to take advantage of the programming model used. Each wave process data items contained in a range of the data grid. Furthermore, waves are contained in work-group that have an identifier along the x,y and z axis. We have used these identifiers to show ranges of work-groups in the debug view. The view uses a four level structure, with the first three levels respectively specifying ranges along the x,y, and z axis. The last level contains a list of waves.

For GPU waves, filtering could not be done in the same way, as every wave in a dispatch is executing the same function. Thus, the waves are not in different sections of the code and cannot be filtered as CPU threads. Due to the specific programming model, another type of filtering is possible. Indeed, as a grid of items is sent to the GPU, it is possible to filter the waves according to their position in the grid, and reduce the number of waves and events that the developer has to handle. The first goal of this filtering is simply to reduce information overloading, in case there is a generic problem in a function that occurs everywhere in the grid. If the problem occurs everywhere, the filter could show only one wave to remove the parallel component from the perspective and facilitate debugging. On the other hand, some problems could occur only in specific sections of the grid, if the function gives an unexpected result for a certain input. In this case, filtering can be used to remove the sections of the grid where there is no problem. Thus, the developer would be able to insert breakpoints anywhere in the function but avoid useless notifications.

4 Evaluation

In this section, we evaluate and validate the proposed contributions. In the first part, the proposed tracing architecture has been tested and its performance measured. Then, the capabilities of the proposed Eclipse views and features are presented and discussed.

The tracing tests were performed on a Linux Fedora 24 computer using Linux kernel version 4.7.9-200.fc24.x86_64. The computer has four Intel Xeon E7-8867 processors at a frequency of 2.50 GHz. Each processor has 16 physical cores with hyperthreading, for a total of 64 physical cores and 128 logical ones. Each processor has a total of 45 MB of cache, and the computer has a total of 256 GB of physical random access memory installed. LTTng version 2.8.0-pre is used for tracing. The work on GDB tracing is based on the GDB 7.11 branch. The work on GPU debugging is based on the ROCm-GDB 1.0 version, on an Intel Core i7-4790 processor, with an AMD R9 Nano card using Eclipse Neon 4.6. Table 1 and 2 show the computers used for the evaluation of this research.

Element	Quantity	Description
Operating System	ND	Ubuntu 14.04.5
Processor	1	Intel Core i7-4790, 3.90 GHz
Graphic Card	1	AMD Radeon R9 Nano
RAM	4	Kingston DIMM DDR3 1600 MHz de 8 GB

Table 1: Computer used for calculations.

Element	Quantity	Description
Operating System	ND	Fedora 24
Processor	4	E7-8867 v3, 2.50 GHz
Graphic Card	0	ND
RAM	4	TBD

Table 2: Server with 64 physical cores used for tracing.

The proposed tracing architecture has been tested with an open-source parallel file compression program, pbzip2. This program is a parallel implementation of the serial file algorithm bzip2 that uses the Burrows-Wheeler algorithm to compress a single file. The benchmarks are done using version 1.1.13 of pbzip2. The program is compiled with debug symbols enabled, 01 optimisation level, and function inlining disabled to facilitate debugging. It is statically linked with the bzip2 library, version 1.0.6 compiled from source. A combination of bash and python scripting is used. There is a total of 15 426 444 tracepoint hits distributed among the different number of threads in each experiment. The same 100 MB randomly generated file is used in each experiment. Figure 11 shows the total execution time for the proposed GDB implementation using LTTng tracepoints, the default implementation using fast tracepoints and the baseline using GDB without tracepoints.

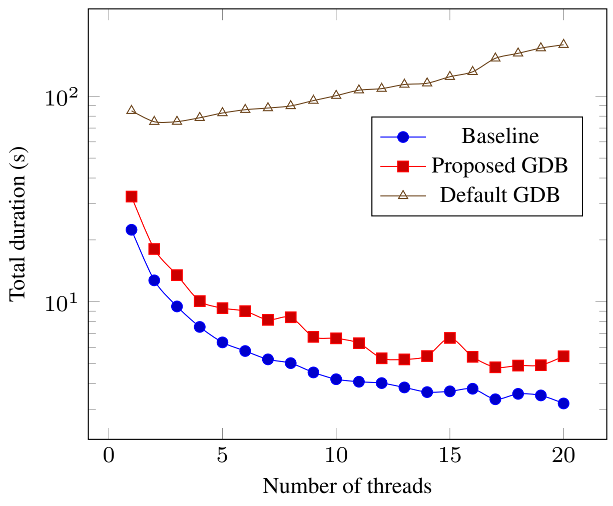


Fig. 11: Total execution time to compress a 100 MB file using pbzip2

As expected, we can see in Figure 11 that the proposed implementation using LTTng tracepoints scales well as the tracer architecture is designed for parallel tracing. The default GDB implementation using fast tracepoints has a significantly higher cost than tracing using LTTng. In the case of a single thread, the total time using GDB with LTTng is 32.5 seconds, less than half the time it takes for the default fast tracepoints in GDB at 85.0 seconds. In this case, no time is lost while waiting to acquire the lock, as there is only a single thread. However, there is a penalty associated with the stops for data transfers while the proposed architecture does not stop the program to transfer data.

When the number of threads is increased, the performance gain of the proposed architecture over the standard fast tracepoints becomes even more noticeable. The LTTng GDB tracing follows the baseline with a small time penalty and it achieves a very similar speedup to the baseline, as seen in Figure 12. On the other hand, the performance of the default tracer is slightly better for two threads than one, and starts to decreases for a larger number of threads. For instance, its execution time with 5 threads is longer than for a single thread. Contention for the spinlock, as multiple threads try to acquire it at a high frequency, is the reason for this performance issue on multi-threaded software. The performance improvement of the proposed architecture on large multi-core systems becomes clear, with the results in Figure 11.

The default GDB tracing implementation uses the tool Trace Visualizer in the console. This visualizer allows the developer to navigate between captured trace frames and look into the data contained by these frames. A similar interface has been proposed. This interface is implemented within GDB in Python, and uses Babeltrace to read trace data. The commands are very similar to what already exists in GDB. Figure 13 shows how this appears on the console. One advantage of using Python to create the commands for trace visualization is the flexibility. Indeed, it is easy for the end-user to modify how the trace data will appear on the terminal and does not necessitate recompilation to work.

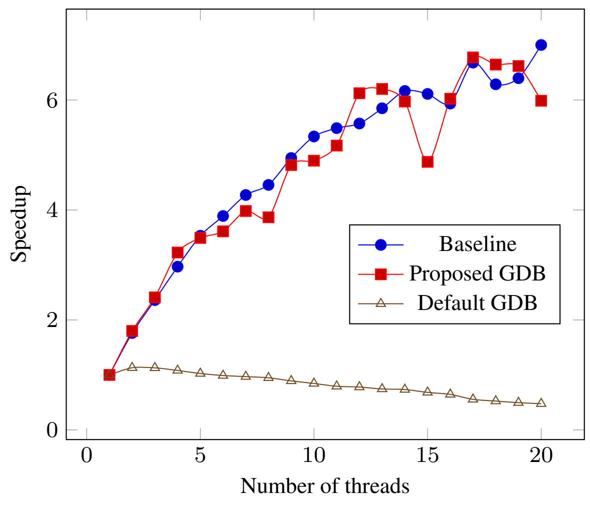
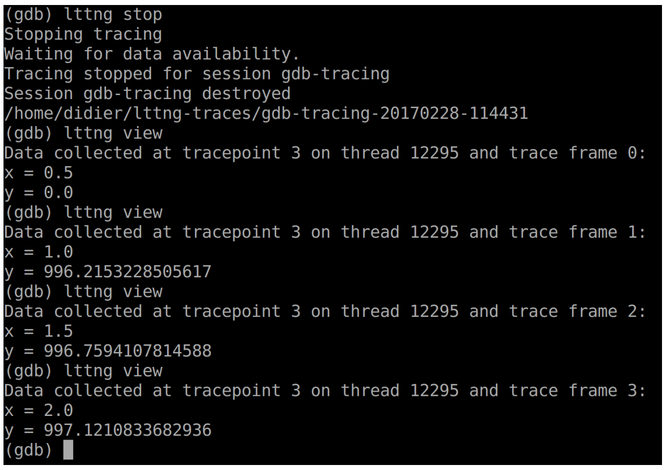


Fig. 12: Speed up of pbzip2 debugging using GDB standard and proposed tracing



```
(gdb) lttng stop
Stopping tracing
Waiting for data availability.
Tracing stopped for session gdb-tracing
Session gdb-tracing destroyed
/home/didier/lttng-traces/gdb-tracing-20170228-114431
(gdb) lttng view
Data collected at tracepoint 3 on thread 12295 and trace frame 0:
x = 0.5
y = 0.0
(gdb) lttng view
Data collected at tracepoint 3 on thread 12295 and trace frame 1:
x = 1.0
y = 996.2153228505617
(gdb) lttng view
Data collected at tracepoint 3 on thread 12295 and trace frame 2:
x = 1.5
y = 996.7594107814588
(gdb) lttng view
Data collected at tracepoint 3 on thread 12295 and trace frame 3:
x = 2.0
y = 997.1210833682936
(gdb) 
```

Fig. 13: Trace data visualization with the proposed GDB fast tracing architecture

Another advantage of the proposed tracer architecture is the ability to enable kernel tracing at the same time as user-space tracing while debugging. Kernel traces can be easily visualized in Trace Compass, as seen in Figure 3. This can help to find issues such as resource starvation or mutex contention. Furthermore, interpreting large amounts of trace data can be hard for a developer. The default fast tracer only provides a terminal based interface to read trace data. The proposed implementation also save the trace data to a file where it can be read by Trace Compass. This visualisation tool allows the developer to write XML analysis and easily create views that display the trace data. An example of an XML view in Trace Compass is shown in Figure 4.

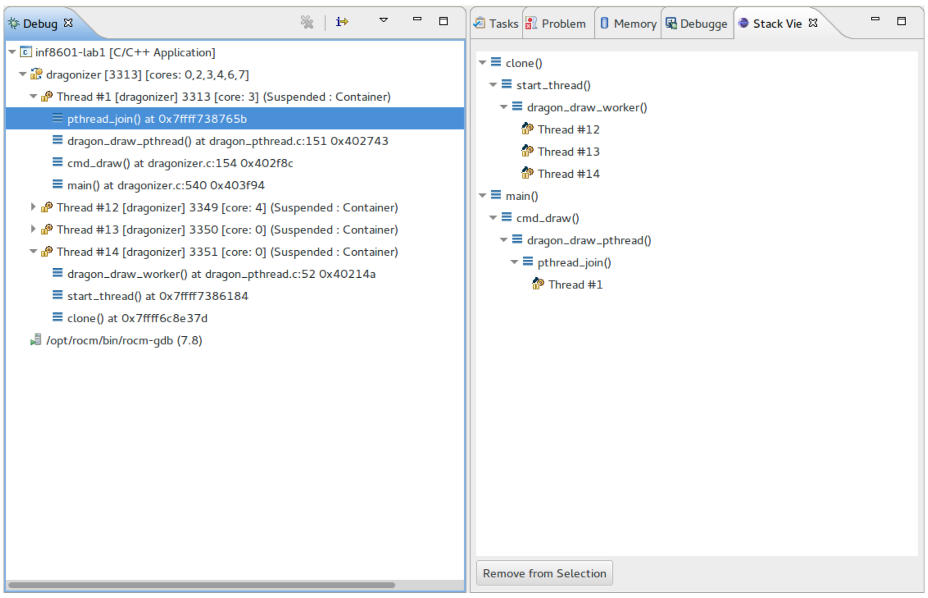
Using Figure 9, we can easily realize that there are two groups of threads. The first contains only one thread waiting in *pthread_join*, while the other contains nine threads doing work in *dragon_draw_worker*. A quick overview also shows that three threads of the second group are waiting on a barrier. When a program contains more threads, the developers can collapse a section of the tree, if threads are working on some parts of the code they are not interested in.

The proposed thread filter has been implemented within GDB. We have decided to use the stack aggregation view to set the filter for the threads, as it already shows logical groups of threads. To enable filtering, the users simply selects the threads they want to remove, as shown in Figure 9 and clicks the *Remove from Selection* button. Eclipse CDT sends the information to GDB, and the filter is created. The selected threads are not shown the next time the program stops. As we can see in Figure 14, the threads selected in Figure 9 are not present because the filter has been applied. Furthermore, if one of the selected thread hits a breakpoint, GDB simply continues and does not notify the user. Filtering has been implemented within GDB to maximize performance. It avoids communication between Eclipse CDT and GDB to decide if a thread is in the selection, and accelerates breakpoint handling. Furthermore, it reduces the size and number of messages sent, as it does not need to send information related to threads excluded by the filter.

The GPU waves have been included in the main debug view using work-groups, as shown in Figure 15. There are four levels in the tree, one for each dimension of the grid, and the final one to display the waves in a work-group. With this view, the developer is able to have a better overview of the program than a long linear list of all the waves. It gives the necessary information to the developer, and allows him to select a specific rang in the data grid to inspect.

The GPU waves filter has also been implemented within GDB. A simple Eclipse view allows the developers to specify the range of position they want to set the focus on. The user can select a combination of ranges along one or more axis. An example is shown in Figure 16. In this case, the ranges selected are 0 to 63 for the X axis, 0 to 47 for the Y axis and 0 to 1 for the Z axis. Then, the debug view, that previously displayed every GPU waves as shown in Figure 15, only display waves that are in the focus region. Figure 17 shows that the only waves displayed are the one that are inside the focus set in Figure 16. The reduced quantity of information helps the users focus on the problems. The users can also reduce the range of the filter as they narrow down on the origin of the problem. This helps the users stay focused on the problem. Furthermore, it allows them to abstract the parallel aspect, by choosing a specific wave of focus, on a problematic region of the grid.

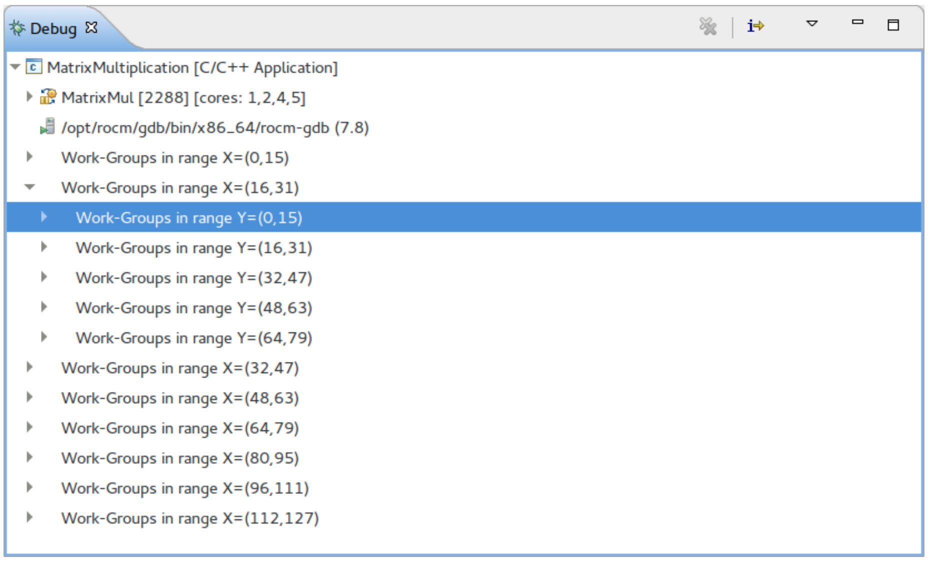
In summary, we have demonstrated the advantages of the proposed architecture over the default tracer. It allows significant performance gains for single-threaded software and even more important gains on multi-core systems. The terminal reader for the traces is easily customizable by the user, without recompilation. Furthermore, it allows the de-



```
inf8601-lab1 [C/C++ Application]
  dragonizer [3313] [cores: 0,2,3,4,6,7]
    Thread #1 [dragonizer] 3313 [core: 3] (Suspended : Container)
      pthread_join() at 0x7ffff738765b
      dragon_draw_pthread() at dragon_pthread.c:151 0x402743
      cmd_draw() at dragonizer.c:154 0x402f8c
      main() at dragonizer.c:540 0x403f94
    Thread #12 [dragonizer] 3349 [core: 4] (Suspended : Container)
    Thread #13 [dragonizer] 3350 [core: 0] (Suspended : Container)
    Thread #14 [dragonizer] 3351 [core: 0] (Suspended : Container)
      dragon_draw_worker() at dragon_pthread.c:52 0x40214a
      start_thread() at 0x7ffff7386184
      clone() at 0x7ffff6c8e37d
  /opt/rocm/bin/rocm-gdb (7.8)
```

```
clone()
  start_thread()
    dragon_draw_worker()
      Thread #12
      Thread #13
      Thread #14
main()
  cmd_draw()
    dragon_draw_pthread()
      pthread_join()
        Thread #1
```

Fig. 14: Debug view and Stack Aggregation view when the filter has been applied.



```
MatrixMultiplication [C/C++ Application]
  MatrixMul [2288] [cores: 1,2,4,5]
  /opt/rocm/gdb/bin/x86_64/rocm-gdb (7.8)
  Work-Groups in range X=(0,15)
  Work-Groups in range X=(16,31)
    Work-Groups in range Y=(0,15)
    Work-Groups in range Y=(16,31)
    Work-Groups in range Y=(32,47)
    Work-Groups in range Y=(48,63)
    Work-Groups in range Y=(64,79)
  Work-Groups in range X=(32,47)
  Work-Groups in range X=(48,63)
  Work-Groups in range X=(64,79)
  Work-Groups in range X=(80,95)
  Work-Groups in range X=(96,111)
  Work-Groups in range X=(112,127)
```

Fig. 15: Eclipse CDT main debug view with GPU waves shown

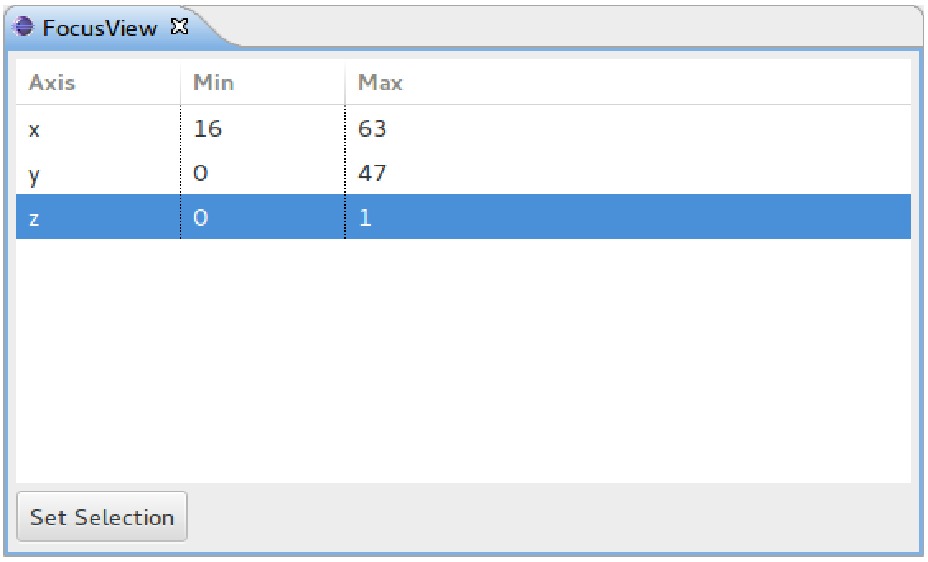
veloper to combine kernel and user-space tracing easily, and Trace Compass can be used to obtain helpful data visualization.

5 Threats to Validity

There are some threats to validity related to our proposed solution that need to be considered.

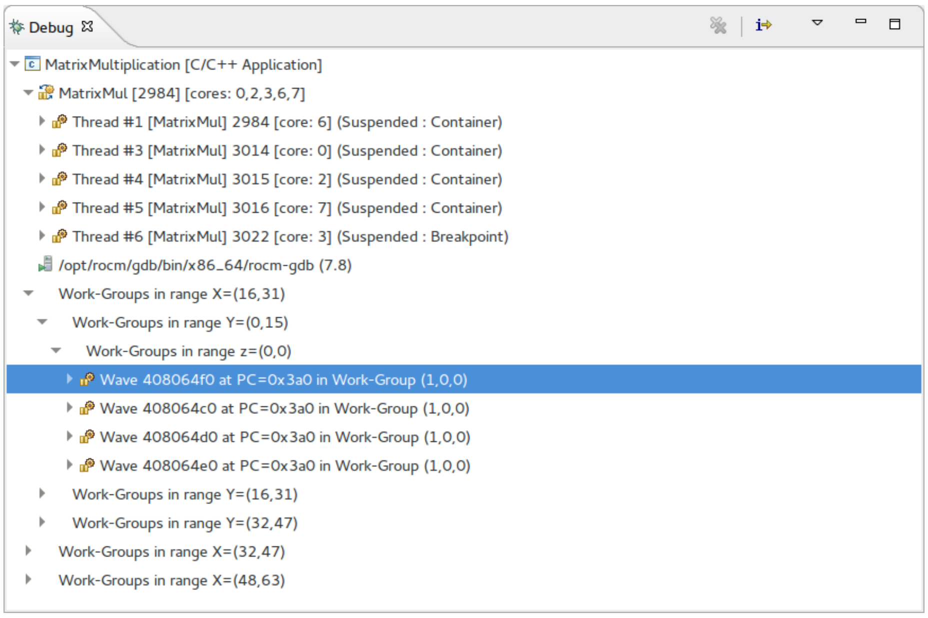
5.1 Tracing and conditional debugging

The first threat related to the fast dynamic tracing is that there is a minimum size to replace an instruction. Indeed, on architectures with instructions of variable size, such as Intel x86-64, the instruction must at least be the size of a jump. On Intel, that corresponds to a 5-byte instruction. Therefore, we cannot insert quick trace points anywhere, and this can be problematic. This limitation is known, as denoted in Chapter 3.



Axis	Min	Max
x	16	63
y	0	47
z	0	1

Fig. 16: GPU focus view to filter HSA waves



```
MatrixMultiplication [C/C++ Application]
  MatrixMul [2984] [cores: 0,2,3,6,7]
    Thread #1 [MatrixMul] 2984 [core: 6] (Suspended : Container)
    Thread #3 [MatrixMul] 3014 [core: 0] (Suspended : Container)
    Thread #4 [MatrixMul] 3015 [core: 2] (Suspended : Container)
    Thread #5 [MatrixMul] 3016 [core: 7] (Suspended : Container)
    Thread #6 [MatrixMul] 3022 [core: 3] (Suspended : Breakpoint)
    /opt/rocm/gdb/bin/x86_64/rocm-gdb (7.8)
  Work-Groups in range X=(16,31)
    Work-Groups in range Y=(0,15)
      Work-Groups in range z=(0,0)
        Wave 408064f0 at PC=0x3a0 in Work-Group (1,0,0)
        Wave 408064c0 at PC=0x3a0 in Work-Group (1,0,0)
        Wave 408064d0 at PC=0x3a0 in Work-Group (1,0,0)
        Wave 408064e0 at PC=0x3a0 in Work-Group (1,0,0)
    Work-Groups in range Y=(16,31)
    Work-Groups in range Y=(32,47)
  Work-Groups in range X=(32,47)
  Work-Groups in range X=(48,63)
```

Fig. 17: Eclipse CDT main debug view with GPU waves and an active GPU focus

5.2 Display of execution waves

The waves of execution on a graphic processor are grouped in a tree structure based on their position in the data grid. However, only the nodes of the first level of the tree are displayed, since there is a very large number of waves. The developers need to enlarge manually the groups they want to explore. This can complicate the use of the view, as the users have to navigate in the tree.

5.3 Filters

Filters allow the users not to be notified of events that occur outside the focus. However, the debugger still receives these events, and must manage them. That makes so that the program can be slowed even if the users do not receive an event. It is a similar limitation as the one presented by normal breakpoints.

5.4 Comparison to other tools

Having an open-source tool was an important requirement for the industrial partners. In order to use any possible tools developed as part of this project, they need to recompile and adapt the tools for their custom multi-core systems. Therefore, closed-source tools were not given as much consideration because they could not be adapted for their custom processors. This limitation was imposed by the initial requirements of the industrial partner.

6 Conclusion

In this paper, we have shown the limitations of current debuggers for large scale parallel systems. We have presented various methods currently available in both open and closed source software. We have analyzed the performance of the tracer available with GDB and the features offered in Eclipse CDT for multi-thread debugging.

An improved technique for dynamic tracing in GDB was proposed based on the results of the analysis. This architecture uses both GDB and LTTng-UST to ensure scalability of the dynamic user-space tracer. The performance of the proposed implementation has been compared to the performance of the default GDB fast tracer. The test results have shown that fast tracepoints using LTTng scale well, while fast tracepoints using the standard GDB implementation are greatly penalized by the lock protecting buffer access and data transfer. Tracing performance quickly deteriorates when multiple threads hit a breakpoint frequently, while the LTTng tracepoints performance is not affected by multi-threading.

We have evaluated the performance improvement of our tracing tool using a parallelized file compression tool. For instance, our tool creates an overhead of 90% when 15 threads are used to compress the file, while the default tracing tool causes an overhead of 3294%. As our tool does not cause the program to slow down by two orders of magnitude, it could conceivably be used to inspect software used in production. One could attach this tool to various functions in a web server application used in production, record variables at key points and understand how a problematic code path could be taken.

A view to group threads by their call stack has been presented and evaluated. The algorithm to build this view has been developed to ensure scalability on large multi-threaded programs. The main Eclipse debug view has been enhanced to include GPU waves for heterogeneous debugging. A filter was developped to exclude irrelevant waves and threads from the debugging views to reduce unneeded information. These contributions help the developer by providing a better view of the debugging context to focus on the problems to solve. It can be of use to analyze complex program. For instance, one could use this view to efficiently group the numerous threads used by a finite element solver and quickly understand what each thread is doing.

These contributions are a first step to adapt debugging tools for modern large scale heterogeneous systems. The tracing technique efficiently provides data to help the user. However, it still requires an instruction large enough to replace it by a jump. This limitation could be removed by displacing a function frame and instrumenting it. The proposed views and the filter simplify and enhance the user experience. The feedback was obtained from three software engineers working at Ericsson on open-source debugging tools (Eclipse and GDB). These engineers frequently interact with multiple teams internally at Ericsson and discuss with them to understand their needs. Therefore, these engineers were well-positioned to evaluate the appeal of the proposed solution.

The handling of excluded threads works in the same way as for conditional breakpoints, and could be improved as it involves costly context switches that carry an important overhead.

Acknowledgment

We would like to gratefully acknowledge the Natural Sciences and Engineering Research Council of Canada (NSERC), Prompt, Ericsson, and EfficiOS for funding this project.

References

1. Inc. Advanced Micro Devices. Gpuopen, 2016.
2. KB Antypas. Allinea ddt as a parallel debugging alternative to totalview. *Lawrence Berkeley National Laboratory*, 2007.
3. Tim Bird. Measuring function duration with ftrace. In *Proceedings of the Linux Symposium*, pages 47–54. Citeseer, 2009.
4. B Brandenburg and J Anderson. Feather-trace: A lightweight event tracing toolkit. In *Proceedings of the third international workshop on operating systems platforms for embedded real-time applications*, pages 19–28, 2007.
5. Jonathan Corbet. Uprobes in 3.5, 2012.
6. Mathieu Desnoyers and Michel R Dagenais. The lttng tracer: A low impact performance and behavior monitor for gnu/linux. In *OLS (Ottawa Linux Symposium)*, volume 2006, pages 209–224. Citeseer, 2006.
7. Mathieu Desnoyers and Michel R. Dagenais. Lockless multi-core high-throughput buffering scheme for kernel tracing. *SIGOPS Oper. Syst. Rev.*, 46(3):65–81, December 2012.
8. Naser Ezzati-Jivan and Michel Dagenais. Multi-scale navigation of large trace data : A survey. *Concurrency and Computation : Practice and Experience*, 2017.
9. HSA Foundation. Hsa programmer's reference manual: Hsail virtual isa and programming model, compiler writer, and object format (brig) 1.1. Technical Report 1.1, HSA Foundation, February 2016.
10. Kim Hazelwood. Dynamic binary modification: Tools, techniques, and applications. *Synthesis Lectures on Computer Architecture*, 6(2):1–81, 2011.

11. John L Hennessy and David A Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 2011.
12. Namhyung Kim. uftrace - function (graph) tracer for user-space, 2017.
13. L. Layman, M. Diep, M. Nagappan, J. Singer, R. Deline, and G. Venolia. Debugging revisited: Toward understanding the debugging needs of contemporary software developers. In *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 383–392, Oct 2013.
14. LLVM Project. The **LLDB** debugger, 2017.
15. Wolfgang E Nagel, Alfred Arnold, Michael Weber, Hans-Christian Hoppe, and Karl Solchenbach. Vampir: Visualization and analysis of mpi resources. 1996.
16. Andreas Olofsson and Marc Khouzam. Cdt and parallella multicore debugging for the masses, 2014.
17. Stan Shebs. Gdb tracepoints, redux. *Proceedings of the GCC Developers Summit. GCC Summit. Montréal, Canada*, pages 105–112, 2009.
18. Nathan Sidwell, Vladimir Prus, Pedro Alves, Sandra Loosemore, and Jim Blandy. Non-stop multi-threaded debugging in gdb. In *GCC Developers Summit*, volume 117, 2008.
19. A. Spear, M. Levy, and M. Desnoyers. Using tracing to solve the multicore system debug problem. *Computer*, 45(12):60–64, Dec 2012.
20. Richard Stallman, Roland Pesch, Stan Shebs, et al. *Debugging with GDB*. Free Software Foundation, 10 edition, 2017.
21. The LTTng Project. The lttng documentation, december 2016.
22. Trace Compass Project. Trace compass user guide, 2017.
23. Adrien Verg, Naser Ezzati-Jivan, and Michel R. Dagenais. Hardware-assisted software event tracing. *Concurrency and Computation: Practice and Experience*, pages e4069–n/a, 2017. e4069 cpe.4069.
24. Josef Weidendorfer. Sequential performance analysis with callgrind and kcachegrind. In *Tools for High Performance Computing*, pages 93–113. Springer, 2008.