Reducing the complexity of debugging parallel REPLICA programs with pluggable abstraction patterns

Jari-Matti Mäkelä
TUCS - Turku Centre for Computer Science / Dept. of Information Technology
University of Turku, Finland
jmjmak@utu.fi

Ville Leppänen
Dept. of Information Technology
University of Turku, Finland
ville.leppanen@it.utu.fi

Martti Forsell
VTT
Oulu, Finland
Martti.Forsell@VTT.Fi

ABSTRACT

Traditional debuggers focus on a single thread at a time or are better suited for concurrent programming with a low number of interacting threads and/or distributed memory, making it hard to monitor a massively data-parallel program on a shared memory multi-core system.

This work considers a globally step-synchronous model of computation. Compared to contemporary multi-core processors with indeterministic thread models, it significantly improves the language's safety guarantees, the easiness of programming and parallel debugging while also enabling a more efficient group synchronization. An implementation of the model is utilized by the REPLICA many-core processor.

The contribution of this work is to introduce a new, modular way of extending the debugger with pluggable abstraction patterns. Given a language which supports user defined control abstractions, the library code and its semantics while debugging can be defined in tandem without modifying the compiler toolchain. We identify a set of preliminary parallel programming abstractions relevant to debugging from the high level language, analyze the mismatch between the intent deducable from the resulting runtime code and the original source, and suggest ways for implementing the plugin framework along with a plugin sample for a core language feature.

Keywords

parallel programming, programming languages, debugging

1. INTRODUCTION

One of the largest problems related to software constructions is the inherent, growing complexity of the program code. More advanced programming methodologies and abstractions are invented to mitigate the complexity and to improve maintainability by modularizing code on different abstraction levels. Examples of these are high level programming abstractions such as procedural, object- and aspect-oriented programming, strong type systems, and development processes such as test-driven development.

Complexity also arises on another front. Previously most programming was sequential – now hardware architectures on a wide variety of domains (most notable exceptions being very low-powered micro-controllers and embedded processors) are multi-core. Consequently, utilization of the computational power requires more complex techniques. The most commonly used abstraction is indeterministic threading. Errors in multi-threaded code may lead to deadlocks and race conditions, among other hard to find bugs.

It can be argued, whether threads offer the right level of abstraction for implementing more complex parallel software [9]. Our approach is to support threading on the lowest abstraction level, but in a more limited form by assuming a configurable shared memory architecture [2, 4, 5] that uses a global, deterministic lock-step inter-thread synchronization after each instruction. This results in a very controlled execution model with deterministic progression of threads in an orchestrated manner, which results in much better guarantees of the safety and easiness of programming. It also enables a low-overhead group synchronization on top of the core threading model. For performance reasons, such an architecture requires a combination of several novel multi-core technologies such as low-overhead thread switching, latency hiding, and a synchronization wave mechanism [15]. We believe that these kind of restricted models for threading are only considerable option as the more indeterminate thread models became intractable from correctness point of view. For more information, we refer to the previous papers on our REPLICA architecture [11, 10, 13].

Debuggers are tools used to detect bugs by inspecting the operation of the program running either on a simulator or real hardware. Debuggers have gone long way from the early days of computing when errors in the computation could only be found by printing a memory dump or by measuring the hardware pins to read the data from the volatile memory storage. Modern debuggers have features such as source-level language integration via debugging metadata encoded in the executable, advanced control flow controlling techniques such as stepping, execution state manipulation, breakpoints, triggers for various conditions, memory inspection functionality, and visualization tools.

In parallel programming, the conventional sequential debugger functionality falls short. Large scale parallel systems such as GPGPUs (general purpose computing on graphics processing units) already utilize thousands of threads. Taming this multitude of threads is the state of the art in new debuggers as conventional debuggers focus on a single thread at a time. Multithreaded code can be further divided into data and task parallel categories. Debuggers exist for both approaches. For example, there are debuggers with functionality dedicated for a specific parallel framework such as MPI. Some data parallel debuggers are more hardware plat-

form specific, e.g. the ones for GPGPU platforms such as CUDA, OpenCL, and the NVIDIA PTX virtual machine.

While many parallel debuggers exist, due to the lock-step execution of threads, a more controlled inspection of execution phases is possible. Many debuggers are limited to basic step-wise execution either on instruction level or at statement level on a high level language or provide a fixed set of controls for a specific platform. The scope and contribution of this paper is to extend this control to new platforms and languages in a pluggable, technology agnostic manner. The approach also supports a possibility to support new user defined control flow library abstractions in the debugger without the need to change the compiler / debugger toolchain's code. We outline the general idea, explain how the framework affects the compiler and debugger implementations using our REPLICA toolchain as a target, and demonstrate the use of the framework by implementing the debugger logic for a simple parallel language feature.

The organisation of this paper is as follows. In Section 2 we describe how the abstractions map between high level code and debugging, discuss the modelling methodology, and proceed with a brief analysis of the language's features using the model. In Section 3 we outline a preliminary design for a framework, conceptually implement a parallel language feature and discuss other implementation details related to the debugger. In Section 4 we discuss related work on parallel debuggers and finally draw conclusions and suggest future work in the final Section 5.

2. PLUGGABLE ABSTRACTION FRAMEWORK FOR DEBUGGING

For programming parallel applications, several frameworks and languages have been implemented. However, from a debugger's point of view, these technologies are often a) custom built for a certain platform or b) resort to the least common denominator, that is e.g. the basic concepts of threads, thread local storage, synchronous and asynchronous stepwise execution of language or hardware primitives. In general, the debuggers do not assume a strictly synchronous threading model or even a threaded, shared memory multiprocessor. However, even with such a lax set of assumptions, a preliminary support for concurrent parallelism can be achieved by retrofitting a concept of thread grouping in SPMD or SIMD way to these debuggers as proposed in [13].

2.1 Propagating the intent of programming

Consider basic parallel programming abstractions such as splitting and merging a group of threads in the fork–join programming model or basic prefix sums, list ranking or sorting in classic PRAM literature. Each parallel abstraction introduces a new mechanism for managing and orchestrating a set of threads. How to actually convey the execution stepwise in a meaningful way depends on the language feature currently under execution for each group of threads.

Consider the process of compiling statically typed languages. The language expresses intent in a more or less declarative way. Accompanied with a set of type rules, the compiler extracts the static semantics from the algorithm for compile time analysis and transformations. The resulting executable contains the dynamic semantics of the program. Transformations between language abstraction levels and optimizations lose some of this data or make it harder to extract it in an automated way.

```
@myannotation(key1 = value1, key2 = value2)
while (true) {
  ...
}
```

Figure 1: Example of using annotations.

One of the pluggable abstraction framework's goals is to propagate the required information of the language abstractions from the high level code to the debugger. Enough information is required to reconstruct the high level intent of the feature, but also a conflict free support for feature composition needs to work as real life parallel programs might consist of intertwined parallel patterns.

The mechanism for propagating this data in the Replica language is the pragma annotation. Each syntactic node of the language can be associated with annotations that resemble a function call with named arguments. The value of each argument is passed as is or if it possible to statically further evaluate it using the language's constant folding semantics, the value is substituted with its result. The annotation is only visible to the compilation tools and does not affect the resulting binary code. An example of using annotations is shown in Figure 1. For each statement, the compiler by default generates an associated annotation for explicitly tracking the control flow of each thread. The library and/or user code can freely define additional arbitrary annotations.

2.2 Modelling abstractions

The goal of this work is to identify and extract useful and relevant parallel abstractions from the high level code, consider their resulting form in the executable, find a mapping between the two, and further deconstruct and analyze the abstractions to find sub-abstractions that are implementable without major issues and also support manual debugging work in a practical manner. For this, we need to identify, what kind of information about patterns is lost in the compilation process. Explicit annotations are used for conveying information about the patterns to the resulting code in order to make it available while debugging.

For the analysis, we considered the data-parallel core features of the Replica [10], Fork [8], and E [3] languages, a set of basic parallel abstractions from parallel programming literature [6]. All of these languages are built on ANSI C with small extensions discussed later, when appropriate. Our previous work [13] on debugger design also contributed to the results. While the REPLICA architecture is not inherently asynchronous, a software implementation of a high-level asynchronous task parallel framework [7] was also considered. We believe that such a system is necessary to coordinate low-overhead, high performance data-parallel computation on higher level.

2.3 Preliminary survey of abstractions

The main parallel abstraction in all three analyzed languages is the concept of thread. Each thread is associated with a number of attributes, e.g. its local state and identifiers for thread in a group, group, and the executing processor. Moreover, the group identifiers form a hierarchical tree. These all are important in parallel programming, since not only are they used to classify the purpose and role of each thread, but also participate in the semantics of control

structures.

2.3.1 Thread, groups and thread local state

In this work, we assume the concept of thread and its attributes as part of the core interface for debugging. The syntax tree nodes of the program and the associated annotations are also made available to the debugger. The attributes are implicitly accessible when debugging via the runtime stack and register contents. However, for a preliminary version we suggest a separate debugging mode option for the compiler that stores the thread local state beside the thread / processor id (which have fixed register slots) on stack to simplify the process of locating this data.

2.3.2 Core control constructs

In the core language constructs (`split`, `if`, `for`, `while`, and `do-while`), the control flow can proceed in various ways: each synchronous group of threads may or may not enter a specific branch (the first two branching statements), but the threads inside a branch may also diverge and finish out of sync. For loops, the number of iterations may differ between the threads. In the resulting binary code, the control flow is obscured in such a way that it becomes hard to deduce the exact mapping between the high level construct and the machine instructions. Moreover, the implicit statement level annotations (see Section 2.1) do not explicitly specify the exit point for the construct, only the entry point.

To improve this ad-hoc style, we suggest that the control constructs explicitly annotate each branch and also pairwise the entry and potential exit points to simplify the debugging. Further invariant definitions regarding the execution state are out of the scope of this work, but it seems possible to annotate the synchronization semantics of the control structure in order to monitor and verify the synchronicity properties at runtime. Also worth noting is that the `split` construct may be later implementable as a pure library feature. This way of modeling such abstractions ignores the distinction between library and built-in constructs.

2.3.3 Library provided higher order functions

The Replica standard library is currently only a planned feature for the REPLICA platform, which makes the treatise here highly speculative. The goal of the library is to provide a generic set of basic parallel data structures and algorithms for general purpose application design. On low level, the library would use data-parallel model inside the thread groups, possibly leveraging the fast multi-operations provided by the hardware. The actual implementation is abstracted away from the user. On high level, the routines would use dynamic scheduling and the task parallel runtime system to better balance the resource allocation throughout the whole program.

Since in practice the library code is often better tested and verified than user code, debugging the code step-wise on instruction or statement level seems to offer diminishing returns. The role of the debugger on this level would emphasize profiling the algorithm in a detailed way. The annotation metadata provided by the library would tell the low level hardware mechanism and algorithm type (e.g. complexity) assigned for the task. Also on high level, the library code would report about the scheduling of individual subtasks related to a parallel meta-task. The job of the debugger plugin would be to collect the information and present it in a visual tool using parallel algorithm visualization styles described e.g. in [6]. For stepping through the code, the debugger would offer special step options that consider the algorithm specific semantics.

2.3.4 Task parallel library

The previous work on task parallel libraries on top of a REPLICA style stricly synchronous shared memory architecture describes a preliminary runtime system for tasks and dependencies. REPLICA further complicates the issue by offering a set of especially fast parallel multi-operations for a limited number of threads. We have decided to associate these limited resources with parallel tasks [12]. Like in other task parallel systems, monitoring the execution deals with defining regions for task related code and task dependencies.

The task system has not been implemented yet, but we have plans to integrate the earlier model into the Replica language with either language or library assisted wrappers for data. The task dependencies would be determined statically, while the sheduling is dynamic. If the language's macros can be used to implement all wrappers, the related annotation metadata for the debugger may also be injected using macros using only library code. If the wrappers need language support, the annotations may need to be partially added by the compiler. Another possibility is to explicitly annotate the task parallel code.

In order to fully debug the task parallel model, we would use the annotation data to deduce the task sections and their static dependencies at call site. Monitoring the scheduling requires interaction with the runtime system and is outside this work's scope. The information about task specific resources (i.e. the synchronization token in the current implementation) would be available via the thread's local state in a fixed position.

The task relations seem to be easier to comprehend from a visual presentation format (e.g. [14]). For manipulation of the tasks, the debugger would offer functionality for starting, pausing, and ending tasks, removing tasks from and injecting tasks to the queue. Also the dynamic dependencies between tasks could be manipulated.

3. EVALUATION OF THE RESULTS

While studying the parallel programming on REPLICA architecture using the Replica language, we recognized sets of abstractions on various abstraction levels, starting from the thread level concepts such as threads with thread local identifier information, groups and group hierarchies, annotations and statements.

On top of the lowest level abstractions are language core constructs, higher order library routines and finally the task-parallel runtime library. We could find various abstractions helpful for debugging on these levels and each abstraction level seems to have ties to the next abstraction level above it. The higher the abstraction level, the more we also pose requirements for the debugger's advanced functionality. The proposed implementations do not utilize a scripting engine yet, but ideas for connecting with one were discussed.

4. RELATED WORK

The idea of extending debugger with domain specific functionality is not new. For example, the GNU debugger (GDB) [16] supports scriptable, user defined pretty printers for data

types. The idea of separating debugging from the application code is completely reduced in scripting languages such as JavaScript where the application code stays resident inside the interpreter process and can be inspected and stepped through without the need for external debuggers.

Numerous parallel debuggers also exist for parallel platforms. Many debuggers focus on more traditional technologies such as message passing concurrency with MPI. The use of traditional debuggers is suggested even for new multicore platforms such as Paraleap [17], Intel SCC (single chip cloud computer), and Intel MIC (many integrated core). GPGPU debuggers such as Allinea DDT, GPU Ocelot [1], and NVIDIA Nsight seem to consider the data parallelism, but are more tightly coupled with the warp based concept of parallelism which is less general than REPLICA's semantics.

Visualization, in relation to debugging in parallel, has also been widely studied. Among other important topics are projects for visualizing the results of performance profiling, task graphs and messaging in task parallel programming, data access in data parallel applications, and the parallel program state in general. Any of these approaches can be considered for inclusion in the Replica debugger, but since the semantics of the platform do not exactly match with any existing ones, the solutions may need to be re-engineered to take into account e.g. the step-wise synchronicity on hardware level and the differences between the semantics of control structures on the language level. However, a further treatise of visualization techniques is outside our scope.

5. CONCLUSIONS AND FUTURE WORK

We conclude that the presented preliminary framework can be used at least for expressing simple extensions to the basic paradigm comprising threads and groups. The expressiveness of the pluggable debugger functionality is largely constrained by the exposed interfaces. To graphically visualize the control flow, execution state or data structures, bindings to a graph library (e.g. JUNG [14]) are needed.

While the current implementation does not use a scripting language in the debugger like e.g. GDB for its pretty printing, a scriptable interface would allow extending the functionality hand in hand with the changes to the abstractions presented in the library code. Such an interface could be realized with an existing scripting engine.

As future work, we see much potential in parallel debugging. The parallel programming paradigm could benefit e.g. from a more disciplined model of computation with invariants, which could open up doors for more checking of code both at compile and run time. A debugger utilizing a cycle accurate hardware simulator, such as the Replica debugger, would be a natural framework for building the latter.

More work is needed to determine, considering the computational model, which conditions are terse enough for annotations to be practical, easily verifiable at compile time, and what information needs to be propagated to the debugger.

Acknowledgments: This work was funded by VTT.

6. REFERENCES

[1] G.F. Diamos, A.R. Kerr, S. Yalamanchili, and N. Clark. Ocelot: A Dynamic Optimization Framework for Bulk-synchronous Applications in Heterogeneous Systems. In *Proc. PACT'10*, pages 353–364. ACM, 2010.

[2] M. Forsell. A Scalable High-Performance Computing Solution for Network-on-Chips. *Micro, IEEE*, 22(5):46 – 55, sep–oct 2002.

[3] M. Forsell. E – A Language for Thread-Level Parallel Programming on Synchronous Shared Memory NOCs. *WSEAS Trans. on Computers*, 3(3):807–812, jul 2004.

[4] M. Forsell. Configurable emulated shared memory architecture for general purpose MP-SOCs and NOC regions. *Networks-on-Chip, International Symposium on*, 0:163–172, 2009.

[5] M. Forsell and M. Hiivala. Multi-core portability abstraction. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 778–785, 2012.

[6] Ja Ja Joseph. *An Introduction to Parallel Algorithms*. Addison–Wesley, 1992.

[7] C. Kessler and E. Hansson. Flexible Scheduling and Thread Allocation for Synchronous Parallel Tasks. In *Proc. of 10th Workshop on Parallel Systems and Algorithms (PASA'12)*, 2012.

[8] C.W. Kessler and H. Seidl. The Fork95 Parallel Programming Language: Design, Implementation, Application. *Int. Journal of Parallel Prog.*, 1997.

[9] E.A. Lee. The Problem with Threads. *Computer*, 39(5):33–42, May 2006.

[10] J-M. Mäkelä, E. Hansson, D. Åkeson, M. Forsell, C. Kessler, and V. Leppänen. Design of the Language Replica for Hybrid PRAM-NUMA Many-Core Architectures. In *10th IEEE International Symposium on Parallel and Distributed Processing with Applications, ISPA 2012*, pages 697–704. IEEE, 2012.

[11] J-M. Mäkelä, E. Hansson, M. Forsell, C. Kessler, and V. Leppänen. Design Principles of the Programming Language Replica for Hybrid PRAM-NUMA Many-Core Architectures. In *Proc. MCC 2011*, page 136. Linköping University, 2011.

[12] J-M. Mäkelä, V. Leppänen, and M. Forsell. Composable Hierarchical Synchronization Support for REPLICA. In Kiss Ákos, editor, *13th Symposium on Programming Languages and Software Tools*, pages 230–244. University of Szeged, 2013.

[13] J-M. Mäkelä, V. Leppänen, and M. Forsell. Towards a parallel debugging framework for the massively multi-threaded, step-synchronous REPLICA architecture. In Boris Rachev and Angel Smrikarov, editors, *To appear in the Proceedings of the 13th International Conference on Computer Systems and Technologies*, ACM ICPS. ACM, 2013.

[14] J. O'Madadhain, D. Fisher, S. White, and Y.B. Boey. The JUNG (Java Universal Network/Graph) Framework. Technical Report UCI-ICS 03-17, University of California, 2003.

[15] A.G. Ranade. How to Emulate Shared Memory. *Journal of Computer and System Sciences*, 42(3):307–326, 1991.

[16] R. Stallman, R.H. Pesch, S. Shebs, et al. *Debugging with GDB*. Gnu Press, 2002.

[17] X. Wen and U. Vishkin. FPGA-based Prototype of a PRAM-on-chip Processor. In *CF '08: Proceedings of the 2008 conference on Computing frontiers*, pages 55–66, New York, NY, USA, 2008. ACM.