SEVENTH FRAMEWORK PROGRAMME
THEME
FET proactive 1: Concurrent Tera-Device Computing (ICT-2009.8.1)

PROJECT NUMBER: 249013

Exploiting dataflow parallelism in Teradevice Computing

D4.5 – Optimized version of the compilation tools, with the invitation of third-party contributors

Due date of deliverable: 31/12/2011
Actual Submission: 31/12/2011

Start date of the project: January 1st, 2010 Duration: 48 months

Lead contractor for the deliverable: INRIA

Revision: See file name in document footer.

Project co-founded by the European Commission within the SEVENTH FRAMEWORK PROGRAMME (2007-2013)	
Dissemination Level: PU	
PU	Public
PP	Restricted to other programs participant (including the Commission Services)
RE	Restricted to a group specified by the consortium (including the Commission Services)
CO	Confidential, only for members of the consortium (including the Commission Services)

Change Control

Version#	Author	Organization	Change History
1.0	Albert Cohen	INRIA	First version
1.1	Albert Cohen	INRIA	Revision with feedback from Pedro Trancoso
1.2	Albert Cohen	INRIA	Integrated corrections from Salman Khan

Release Approval

Name	Role	Date
Albert Cohen	Originator	23/12/2011
Albert Cohen	WP Leader	23/12/2011
Roberto Giorgi	Project Coordinator for formal deliverable	30/12/2011

TABLE OF CONTENTS

The following list of authors will be updated to reflect the list of contributors to the writing of the document.

Riyadh Baghdadi, Albert Cohen, Feng Li, Antoniu Pop

INRIA

Rosa Badia

BSC

François Bodin, Laurent Morin

CAPS

Daniel Goodman, Salman Khan, Ian Watson, Mikel Luján
University of Manchester

Printed in Siena, Italy, Europe.

Part number: *please refer to the File name in the document footer*.

1 Glossary

Nothing specific for this deliverable.

2 Executive Summary

The tool flow for the TERAFLUX project has been evolving throughout the second year. The rationale for this evolution, and the associated semantic and algorithmic contributions have been reported in D4.4 (exploitation of multi-level parallelism, locality optimizations). This deliverable surveys the most important design and implementation information regarding the main components of the tool chain, and links to the online resources about the prototype component themselves. These components are being integrated following the communication scheme between the efficiency programming models' source-to-source compilers and the TERAFLUX GCC back-end, and the complete integrated flow will be demonstrated in the third year.

3 Introduction

The overall objective of WP4 is the development of compilation tools which are tailored to the TERAFLUX architecture and programming models.

This deliverable describes the design and implementation of the first version of the tool flow, mapping the productivity and efficiency programming layers to the TERAFLUX architecture. This flow is composed of fully operational components dealing with all aspects of the code generation, optimization and runtime support. It performs locality optimizations and handles multi-level parallelism as described in D4.4. But the different components are not yet integrated. The integration plans for the 3rd year will be discussed in the last part of this document.

3.1. Document structure

Section 4 outlines the main developments conducted on the TERAFLUX tool flow in the 2nd year of the project. These developments fill many gaps and complement the baseline tool flow presented in D4.1 and D4.3. A complete integration will take place in the 3rd year of the project. Section 5 surveys the design and implementation challenges and choices of the Scala component of the flow. Section 6 and 7 report on the StarSs and GCC backend implementation respectively. Section 7 summarizes our achievements and presents the plans for an integrated flow.

3.2. Relation to other deliverables

This deliverable extends the compilation tools described in D4.1 and D4.3. It complements D4.4, covering design and implementation aspects of the tool flow.

3.3. Activities referred by this deliverable

Work and software reported in this document corresponds to the current, intermediate progress status of Task 4.2. Implementation of the methods being investigated in Task 4.3 will take place in the 3rd year.

4 Compilation Tools

The prototypes available at the end of the second year can be sorted in 3 categories.

1. The first one is Scala-specific. It is based on run-time libraries from UNIMAN, described in D3.3. The departure from an integration of the Scala tool flow through GCJ is presented in this document. In the future, it is possible that some of this library-based support may be enhanced and implemented into the Java virtual machine itself.

   Technical information and code are available online and updated regularly: `http://apt.cs.man.ac.uk/projects/TERAFLUX/MUTS`

2. The different efficiency programming models all come with their source-to-source compilation framework. The TFlux model and tools have been extended to support transactions (see D3.3). Several locality optimizations and multi-level parallel programming extensions have been implemented in the HMPP Workbench 3.0. The Mercurium compiler for StarSs is now embedded into a comprehensive tool suite within the OmpSs infrastructure. Enhancements for multi-level parallelization have been integrated to the OmpSs tool flow, including the support libraries and code generators for different devices. Note that OmpSs and HMPP models the TERAFLUX architecture as an accelerator device.

   Technical information and code are available online and updated regularly: `http://nanos.ac.upc.edu`

3. The TERAFLUX back-end compiler is currently a collection of three independent patches to the mainline of GCC (development version 4.7). The first patch supports the direct compilation of OpenMP dataflow streaming pragmas to T* intrinsic functions, themselves compiled to the T* ISA (cf. Deliverable 6.2). The second one converts sequential programs through static analysis only. The original flow is modified, replacing pragma expansion with the automatic extraction of threads from arbitrary control flow; the run-time library may also be replaced with T* intrinsic functions. A paper will be presented at the MULTIPROG workshop in January 2012. The third one experiments with the late expansion of OpenMP pragmas, as a double attempt to facilitate the optimization of OpenMP streaming programs (task-level optimizations) and to combine the features of the two previous patches. The goal in the third year is to support a hybrid programming model with pragmas for coarse-grain parallelism and complex dependence patterns, and automatic parallelization for the finest grain of parallelism and scalar dependence patterns.

   Technical information and code are available online and updated regularly:

   `http://www.di.ens.fr/TerafluxProject`

The rest of this deliverable surveys the most important aspects and describes the current status of these tools. The Scala tool chain is discussed in greater detail given the important progress made in understanding its integration in a complete tool flow targeting a hardware-supported dataflow execution model with transactions.

5 Scala compilation and support libraries

Currently Scala and all the associated libraries that we have produced execute on the Java Virtual Machine. To tie this work to the work with lower level languages and to allow this code to execute on the simulations of the proposed hardware we have considered several approaches. While we are yet to decide the exact approach to be undertaken we will now detail the different approaches considered.

5.1 Running a JVM on the simulator

The simulated hardware will need to expose functionality to the software. In our case, this functionality is in two parts: the transactional memory mechanisms and the dataflow scheduler. In COTSon, this is done by expanding the instruction set by using the *cpuid* instruction. As JVM based languages cannot directly use native instructions the standard way of performing system-specific operations is through the Java Native Interface (JNI). This allows the program to link with native methods, which may be implemented in C or assembly. However, JNI calls come with a high overhead. This is mostly related with the passing of arguments and allowing native method to access process state. For small transactions, this overhead becomes unacceptably high. An alternative to this is modifying the JVM itself. The Hotspot virtual machine includes *intrinsic methods*. These are methods that, whenever they are called, are replaced by the JVM. This means, that with small modifications to Hotspot, calls to transactional memory or dataflow mechanisms can be replaced with the appropriate C and assembly code.

For transactions, another issue is that during execution on a JVM, events that should not be transactional may occur. For instance, class loading, recompilation of classes or garbage collection. The JVM must therefore be made aware of transactional execution, and either ensure that these events do not occur during transactions, or exclude their execution from the transactional mechanism. Separating the JVM code from the transactional code in the program becomes more complex if the code is interpreted. In this case, the interpreter must be differentiated by the hardware from the operations of the interpreted code. One option here is to not allow transactions in interpreted code and force compilation of any methods that include or are included in transactions. This issue needs to be explored further.

The JVM also requires modifications to interact with hardware dataflow scheduling. The dataflow scheduler, as proposed, relies on receiving a pointer to the section of code that is the computation in a dataflow thread. In C or C++, this is achieved through passing a function pointer to the scheduler device along with the directive to create a thread. When a thread is ready to execute, the associated code is invoked. However, the JVM may have compiled or recompiled the method code since the time of thread creation, meaning the code's location may have changed. This can be addressed by introducing a level of indirection that involves calling into the JVM to locate the right method using some consistent identifier other than the address. This need not involve changes to the hardware if the code invoked is always the handler in the JVM, the method identifier may be included as a token to the thread. This will add a small additional overhead to the scheduling process.

Finally this approach is also dependent on the underlying operating system for TERAFLUX being able to support a JVM. Possible solutions will be further explored in cooperation with WP6 and WP7 leaders and participants during the first half of the third year.

5.2 Compilation to other codes

If the code is not run directly on a JVM, then some form of compilation to either machine code or a language that is supportable by the other elements of the tool chain will be required. There are several ways this could be achieved.

Modifying Scalac

The last phase of the Scala compiler converts an abstract syntax tree into either JVM byte code or somewhat less well supported CLR byte code. These phases are plug-able and this raises the possibility of producing another backend that will produce either assembly or C++. Currently such a backend would have to operate without garbage collection, and because of its interface with the compiler this phase would be required to handle all the other features of Scala.

While the construction of such a phase is possible, the poor documentation of the compilers internal data structures would make it challenging and additional support would be required to replace the Java libraries and JNI calls. An alternative that overcome many of these issues would be to transform the resultant byte code as discussed next.

Byte code transformation

A more targeted and better documented approach is to convert Java byte code into either C++ or assembly. This approach has the advantage that we can avoid handing all cases by selectively deciding which classes we are going to convert, and avoid issues between Scala and Java code as all the code will be the same at this stage. It would be simplest to target C++ and then work on being able to feed this into the rest of the system tool chain. Again this approach would require the loss of the garbage collector in a basic implementation and how to handle JNI calls would need further thought, but we feel that this approach is worth further examination.

Compiling through GCJ

GCJ is an ahead-of-time compiler for the Java. It can compile Java source code to Java byte code or directly to native machine code. It can also compile Java byte code to native machine code. To avoid using a JVM, we considered the option of using GCJ to compile class files generated by the Scala compiler to native code. On examination we decided that this approach is not feasible for the following reasons:

1. GCJ is not yet fully compatible with Java 1.5 while Scala is dependent on Java 1.6 features.
2. GCJ relies on the byte code being compiled with the Eclipse Compiler and makes assumptions about the class files based on this. This means that GCJ does not accept every correct class file including most non-trivial class files generated by the Scala compiler.

We also evaluated a commercial tool, Excelsior JET. This, like GCJ, is an ahead of time compiler coupled with an interpreter for any portions of code that fail to compile. This tool has better compatibility, and can support Java 1.6, but it still fails to compile large parts of the Scala libraries to native code.

5.3 Summary

To summarize we feel that there is potential in both the JVM and the bytecode transformation options. The GCJ and the Scala compiler modifications are less likely to work for a sensible amount of effort so will not be pursued further at this point.

6 StarSs tool chain

BSC is currently devoting efforts to the OmpSs infrastructure, which combines the OpenMP standard with the StarSs ideas. OmpSs is composed of Mercurium compiler that converts the source code and links it to the NANOS++ runtime. The OmpSs infrastructure is depicted in Illustration1.

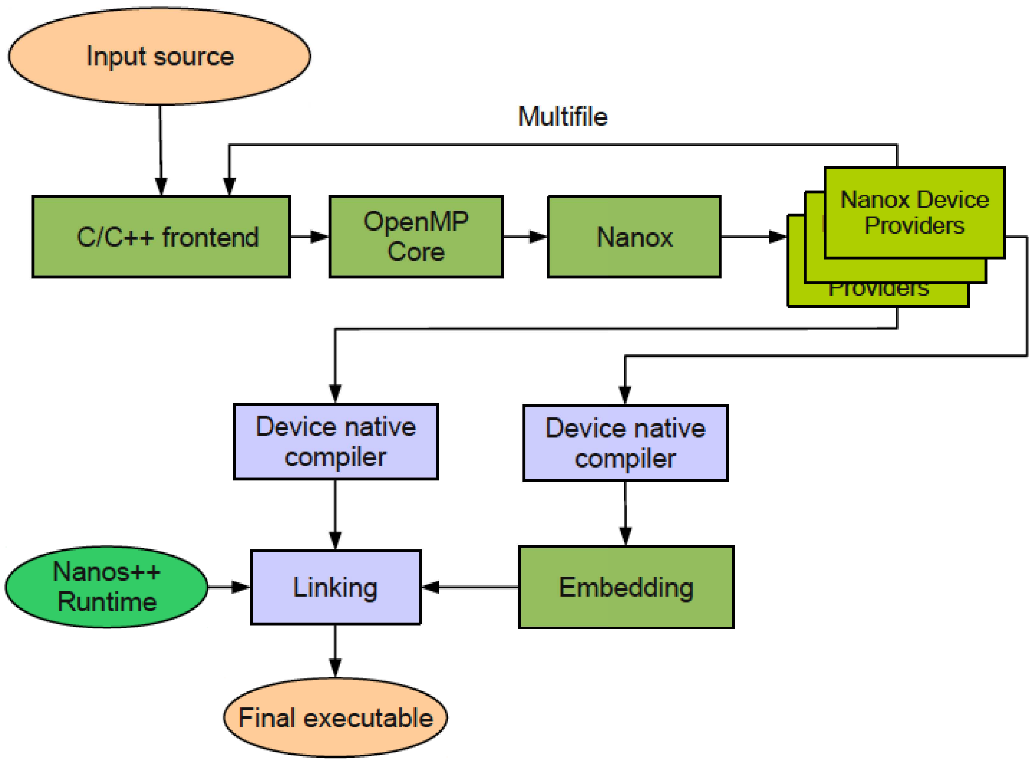

Illustration 1: OmpSs tool flow

The compiler recognizes the constructs and transforms them into calls to the runtime library. The `input/output/inout` clauses are transformed by generating a set of expressions that are evaluated when the application is executed. These expressions generate addresses of memory that will be passed to the runtime library.

The `target` clause is used to indicate that a task is targeting a specific device, like a dataflow thread of the TERAFLUX architecture, or a CUDA GPU. When the compiler generates the code for a `task` construct, it looks if it is annotated with a `target` directive or if another `target` directive is linked to this task construct by means of an `implement` clause. If so, the AST of the task is passed onto the Nanox tool, in charge of dispatching code generation to a device-specific provider and wrapping the device-specific compilation and linking flow for each non-SMP device.

Device providers generate the device-dependent code and metadata that must go associated with the task. They also, if necessary, generate a specialized outline, for the device which may need to be generated in a separated file. This additional file is reintroduced in the compiler pipeline, following usually a different compilation profile that will invoke different backend tools (e.g., the TERAFLUX backend compiler, gcc, or the NVidia compiler, nvcc for CUDA devices).

The binary output for these different files are merged together into a single object file that contains additional information about the different subobjects. This allows the compiler to maintain the traditional behavior of generating one object file per source file to enable compatibility with other tools (i.e., makefiles). The information is recovered at the linkage step to generate the final binary with all the objects.

7 TERAFLUX GCC backend compiler

The code generation pass has been developed as a middle-end pass in GCC 4.7.0 20110426, operating on three-address GIMPLE-SSA code. The traditional compilation flow is being modified according to a specialized adaptation of the builtin-based, late expansion approach described in D4.2 (first year deliverable). Builtins are used both to convey the semantics of input and output clauses in streaming pragmas to the compiler middle-end, and to capture the semantics of efficiency languages such as HMPP, OMPSs and TFLUX.

INRIA is working on software-only and hardware-assisted runtime libraries to support the TERAFLUX GCC backend compiler. These runtime libraries support the semantics of the different efficiency programming models on top of the TERAFLUX GCC's native support for T* dataflow instructions.

- We implemented a new machine description for the backend of GCC, generating T* dataflow instructions derived from the x86_64 ISA. The machine description expands the builtins inserted by the middle-end pass.

- For a hardware-assisted implementation, these instructions are detected by hooks in COTSon's SimNow, using a dedicated encoding of the T* instructions as a 32bit immediate field. This work was conducted as a collaboration between INRIA, UNISI and HP.

- To experiment with software-only implementation, and to implement a scalable dataflow middleware within COTSon, we developed a runtime system called `dfrt.` It is written in C++11 (the latest standard), making heavy use of the functional features such as lambdas and closures for readability. This runtime is inspired from BSC's previous implementation called `tfsim` and meant to be used within COTSon, but it has been completely reimplemented to experiment with a prototype of the memory model being designed for TERAFLUX. This runtime and memory semantics work will be reported in the third year.

8 Conclusion and Integration Plan

We presented the current state of the tool flow, mapping the TERAFLUX programming models to the TERAFLUX architecture. The different components are in place and implement the majority of the expected features and optimizations. Integration and communication between these components remains a challenge, however.

The proposed integrated flow is the following:

1. Scala programs will be mapped to the TERAFLUX architecture relying primarily on runtime libraries, mapping high-level constructs, parallel containers, and combinators, to dataflow and transactional primitives. This will demonstrate how a modern productivity language like Scala, provided with lightweight programming model and class library enhancements, can deliver scalable performance on a wide class of applications.

2. High-level dataflow synchronous programs will be compiled source-to-source to dataflow streaming extensions of OpenMP, to be compiled by the TERAFLUX GCC backend, and benefiting from all locality and parallelism-adapting transformations implemented therein.

3. Efficiency languages StarSs, HMPP and TFlux will be compiled source-to-source to an intermediate, unified representation, in the form of C code with builtin function calls. These builtins will be either dataflow and transactional primitives of the T* ISA itself, or mapped to runtime support functions. In particular, dynamic dependence resolution in StarSs and memory management in HMPP will be delegated to runtime support functions. This unified intermediate code will be further compiled with the TERAFLUX GCC backend. Many of the optimizations designed and implemented in the project will be applied to the unified intermediate code, relying on the builtin functions to avoid hiding the key semantic information for loop and task-level optimization (see D4.3). Some loss of information will not be avoidable however, resulting in missed optimization opportunities.

4. The dataflow streaming extension of OpenMP will be supported natively in GCC and will support all optimizations designed in WP4. The OpenMP pragmas will be converted internally (in the frontend of the compiler) into builtins and runtime library functions, merging with and integrating seamlessly with the flow from other efficiency languages.

5. The TERAFLUX GCC backend will generate T* ISA running on COTSon.

We believe this complete flow can be demonstrated in the 3rd year of the project, providing a unique toolkit for the exploration of dataflow computing principles on a full scale architecture simulator and on real-world applications. Optimizing this flow and extending the semantic expressiveness will be the focus of the 4th year.