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

PROJECT NUMBER: 249013

TERAFLUX

Exploiting dataflow parallelism in Teradevice Computing

D4.4 – Report on multi-level parallelization and locality optimization

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)

Deliverable number: D4.4 – **Dissemination Level: PU**

Deliverable name: **Report on multi-level parallelization and locality optimization**

Change Control

Version#	Author	Organization	Change History
1.0	**Albert Cohen**	**INRIA**	**First version**
1.1	**Albert Cohen**	**INRIA**	**Improved version, implementing important feedback from Pedro Trancoso (UCY)**

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.

Boris Arnoux, Riyadh Baghdadi, Albert Cohen, Feng Li, Antoniu Pop

INRIA

François Bodin, Laurent Morin

CAPS

Printed in Siena, Italy, Europe.

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

SPECIFICATION, WHETHER IN AN ACTION IN CONTRACT, TORT, STRICT LIABILITY, NEGLIGENCE, OR ANY OTHER THEORY, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

1. Glossary

OpenMP – Parallel programming pragma language on top of C, C++ and FORTRAN. In this deliverable, we refer to the OpenMP specification version 3.1.

http://www.openmp.org

HMPP – Hybrid Compiler for Many-core Applications, from CAPS entreprise, generally used as a shortcut for the HMPP pragma language on top of C, C++ and FORTRAN, and for the HMPP development workbench. HMPP exposes multi-level parallelism and memory management through a unique concept and language construct called a *codelet*.

http://www.caps-entreprise.com

StarSs – StarSs is a task-based programming model that enables the exploitation of the applications' inherent parallelism at the task level. To mark the tasks in a StarSs application, annotations (pragmas) similar to the OpenMP ones are used. A uniqueness of StarSs tasks are the input, output or inout clauses that applied to tasks' parameters enable the runtime to track tasks' data dependences.

http://nanos.ac.upc.edu

Graphite – Graphite is the name of a R&D project and a compilation pass of the GNU Compiler Collection (GCC). It implements polyhedral compilation algorithms, applied to automatic parallelization and loop nest optimization. In this deliverable, the unified representation enables us to sketch an extension of Graphite to task-level optimizations and to enhance its analysis with static semantics carried by annotations of the efficiency languages.

Http://gcc.gnu.org/wiki/Graphite

Program Dependence Graph (PDG) – The PDG collects the scalar data dependences and the control dependences between basic blocks of a function. It is a denser representation than the Static Single Assignment form, capturing the same information in a form that can be directly exploited for the conversion of control flow into data flow. The formal definition of the PDG and control dependences in terms of the post-dominance frontier can be found in any good textbook on optimizing compilation.

2. Executive Summary

This deliverable describes the compilation and run-time techniques enabling the adaptation of a parallel program expressing multiple levels and diverse forms of parallelism to the TERAFLUX architecture. Throughout this static and dynamic adaptation process, locality optimizations and tradeoffs between locality and parallelism are paramount.

We report on the multiple advances that took place in the second year of the project. Some of the contributions are generic and are being implemented in GCC, supporting all efficiency programming notations, while others are language-specific.

Four papers have been published: polyhedral compilation for locality optimization at POPL'11 [4] and IMPACT'12 [5], compiler support for multi-level dataflow parallelism and locality optimizations for streaming dataflow programs at HiPEAC'11 [3] and MULTIPROG'12 [2]. A PhD thesis has been defended [1].

3.Introduction

The overall objective of WP4 is the development of compilation and run-time support tools tailored to the TERAFLUX architecture and programming models. The compiler(s) need to map the parallelism and locality as available from the source program and programming model to the target execution model and architecture. The distribution of the roles among the compilation tools and the run-time tools is guided by the efficiency and robustness of handling the challenges statically or dynamically, respectively.

The source program exhibits high levels of concurrency, but it still has to be exploited effectively on the target. As any many-core processor, the TERAFLUX architecture template exposes parallelism in a non-uniform way. In this deliverable, the term multi-level parallelism refers to two complementary aspects:

- The hierarchy of parallel constructs expressed by a parallel program or exposed by the TERAFLUX architecture;
- The different forms of concurrency, expressed as task or data parallelism, pipelining, vector operations, and the heterogeneity of the computing resources associated with the exploitation of these different forms of parallelism.

To adapt the concurrency and data flow computations in the source program to this non-uniform architecture, the compiler and run-time tools implement a variety of code generation and optimization tasks:

- The first task consists in capturing the custom semantics of the different programming models, regarding inter-task dependencies and scheduling constraints (e.g., transactions).
- The second one is to map this semantics to the hardware primitives implemented by the TERAFLUX architecture.
- In addition, to guarantee some level of performance portability, and to increase productivity, the compiler and run-time tools apply various optimizations to coarsen the grain of synchronization, issue bulks of communications, overlap communication and computation, balance computation with communication bandwidth, and harness temporal locality of code and data, taking into account the features of the memory hierarchy.
- The compiler also needs to perform the usual scalar and array optimizations, generating tightly scheduled, fine-grain vectorized computations.

We report on semantical and algorithmic progress achieved during the second year. Our approach involves a combination of static and dynamic optimizations for locality, and language and compiler methods to exploit multi-level parallelism.

3.1. Document structure

Section 4 reports on the projects progresses in the expression and exploitation of multi-level parallelism. Section 5 reports on the compilation and run-time methods for locality optimization. Section 6 concludes and outlines future plans for Tasks 4.2 and 4.3 in WP4.

3.2. Relation to other deliverables

This deliverable extends the compilation algorithms described in D4.1 and D4.3 with enhanced code generation techniques. The associated D4.5 (prototype) deliverable presents the status of the software implementation, and discusses the integration roadmap and issues.

3.3. Activities referred by this deliverable

This deliverable is associated with and represents the intermediate progress status of Task 4.2.

INRIA invested most of its resources on the TERAFLUX GCC backend, developing and implementing a general and modular thread-level partitioning algorithm, combining it with pragma-based dependences expressed as streaming constructs in OpenMP, and contributing the work-streaming compilation and runtime method for coarse-grain dataflow synchronization in dynamic, multi-producer multi-consumer scenarios.

CAPS extended the HMPP Workbench to enhance its support for multi-level parallelism across multiple devices and supporting higher order functional operations. CAPS also designed and implemented language extensions and program transformations for locality optimization of loop nests across multiple dimensions.

Work on Task 4.3 has begun at UNIMAN, BSC and INRIA and has focused on language support and code generation, exploring different semantic integration and runtime support (hybrid transactional memory). The early results are mostly WP3-related and presented in D3.3. Task 4.3 will see direct WP4 contributions in the 3rd year.

4. Multi-level parallelization

Let us now describe the support for multi-level parallelism in the HMPP Workbench and in the TERAFLUX GCC back-end.

4.1 Multi-level parallelization in the HMPP compiler

Support for multi-level parallelism is a native and differentiating feature of HMPP. Compiling HMPP *codelets* to the different block/work-group and thread/work-item levels of CUDA/OpenCL exploits the multi-level annotations provided by the user, and more recently as internal heuristics to balance the exploitation of data parallelism and the exploitation of memory locality (discussed later in this deliverable). The HMPP compiler handles the translation of high-level parallelism annotations into the hardware-specific expression of data-parallel constructs, including low-level vectors intrinsics (i.e.,. builtins) and types supported by a majority of targets (besides NVidia).

HMPP also natively captures the heterogeneity of hardware accelerators; again user annotations (the `target` clause) provide explicit placement directives.

Since HMPP 3.0, the `device` and `nb_device` clauses allow to distribute computations over multiple accelerator devices. At the moment, only data parallel computations are supported; (dataflow) task parallelism is being investigated in collaboration with TERAFLUX partners in the newly established OpenHMPP consortium (and more specifically its language definition committee).

HMPP 3.0 supports currently the following level of parallelism, from the lowest level to the highest level:

- SIMT parallelism extracted from parallel nested loops in codelets.
- Data collection based parallelism that is provided via the specification of a `map` operation. A codelet is applied in parallel to a set of data distributed over the devices via a mirroring mechanism. A syntax example is given in the code fragment below, implementing a parallel `map` operation on an array `d[k]`.

```
#pragma hmpp <mgrp> parallel
for(k=0;k<n;k++) {
     #pragma hmpp <mgrp> f1 callsite
         myparallelfunc(d[k],n);
   }
```

4.2 Multi-level parallelization in the TERAFLUX GCC back-end

INRIA has been working on an algorithm and tool-chain for modular dataflow code generation. This deliverable surveys the main aspects of the design and implementation of this complete tool chain, starting from annotated C code down to a thread-level data-flow machine interface. The framework puts a strong emphasis on control flow generality, scalability and modularity. We propose new code generation algorithms and evaluate their implementation in GCC for automatic and annotation-driven parallelization.

Deliverable number: D4.4 – **Dissemination Level: PU**

Deliverable name: **Report on multi-level parallelization and locality optimization**

The dynamic data flow execution model follows a dynamically constructed precedence graph, where a node in may be executed only if all of its predecessors have finished their execution. To abstract from the ISA details and variations proposed in WP6 and WP7, we define an abstract dataflow interface serving as a target for code generation from parallelizing compilation or efficiency programmers developing low-level dataflow code. The interface defines two main components: the data-flow threads (dynamic task instances) together with their associated frames. The frame of a data-flow thread stores its input values, local variables and some thread meta-data. The producer thread knows the address of the dataflow frames of its successor consumer threads. A thread writes its output data directly into the dataflow frames of its dependent (consumer) threads. Each thread is associated with a Synchronization Counter (SC) to implement the precedence relation. The SC is initialized to the number of its predecessors, and decremented each time one of them provided its data. When the SC reaches 0, the thread is ready for execution.

Our algorithm operates on a low-level representation in Static Single Assignment (SSA) form and is implemented as a new parallelization pass of GCC's middle-end. An example of C code converted to SSA form is shown in Illustration 1 (GCC implement a form of SSA called loop-closed, with strong advantages for loop nest optimization).

```
S1  while (p != NULL) {
S2    x = p->value;
S3    if(c1) {
S4      x = p->value/2;
S5      ip = p->inner_loop;
L1      while (ip) {
L2        do_something(ip);
L3        ip = ip->next;
        }
      }
S6    ... = x;
S7    p = p->next;
    }
```

```
S1  while (p1 = Φ^loop(p0,p2)) {
S2    x1 = p1->value;
S3    if(c1) {
S4      x2 = p1->value/2;
S5      ip1 = p1->inner_loop;
L1      while (ip2 = Φ^loop(ip1, ip3)) {
L2        do_something(ip2);
L3        ip3 = ip2->next;
        }
      }
      x3 = Φ_c1^cond(x1, x2);
S6    ... = x3;
S7    p2 = p1->next;
    }
```

Illustration 1: Static Single Assignment form (SSA)

- The input is the Program Dependence Graph (PDG), collecting data and control dependencies of a given function in a compilation unit. Nodes of the PDG are single basic blocks; they are the finest-grain of dataflow threads achievable by the partitioning algorithm.

- An optimization heuristic coarsens the granularity of the tasks based on a generalization of the Parallel Stage Decoupled Software Pipelining (PS-DSWP); it has been presented in D4.1.

- Modular code generation: the externally visible functions in a compilation unit is cloned into a threaded version, preserving the original control flow as an alternative version. Thread creation is guarded by conditional execution predicates and always located on the post-dominance frontier of the thread's block, i.e., at the source of all control dependencies targeting the thread's block. This scheme avoids the need for speculative thread creation. On the other hand, the structure of dataflow frames is constructed from data dependence information only. But special care must be taken to propagate values and frame pointers among the control dependence paths in the Program Dependence Graph (PDG): the ultimate consumers may not be known at the production point, or the consumer threads may be created earlier than some of their producers. In the former case, values must be propagated along the

control dependence paths to the ultimate consumers, starting from the closest common ancestors of the producer and consumers, as shown in Illustration 2. In the latter case, frame pointers must be propagated instead, as shown in Illustration 3. Function calls are split into an asynchronous thread creation and a return continuation thread for modularity.

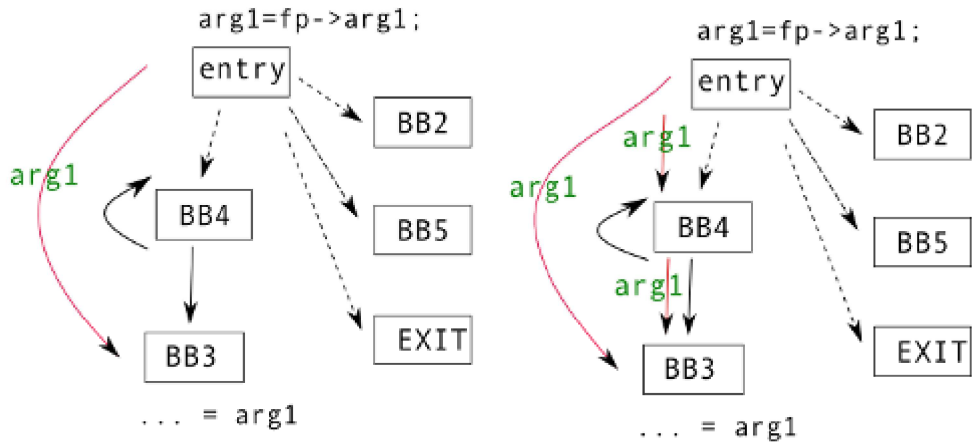


Illustration 2: Propagating values along control dependencies

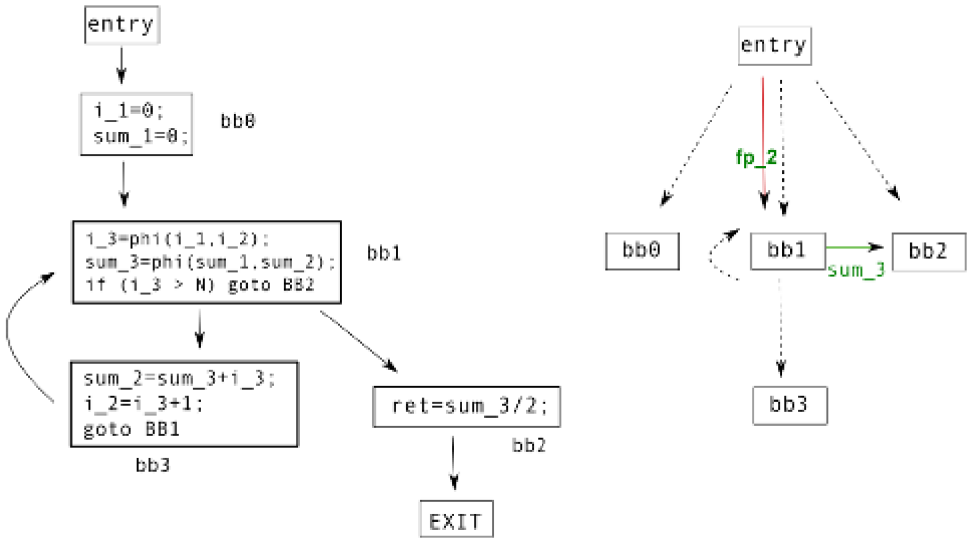


Illustration 3: Propagating frame pointers along control dependencies

Illustration 4 shows a template for modular code generation of imperative function calls. The caller thread will create the entry thread of the callee and the continuation thread for handling the return value. The entry thread creates post-dominating threads inside the callee, and the return thread pass the return value to the continuation in the caller. The algorithm is fully modular and can convert arbitrary control flow into dynamic dataflow threads at basic block granularity. More detail can be found in a MULTIPROG'12 paper by Li et al. [2].

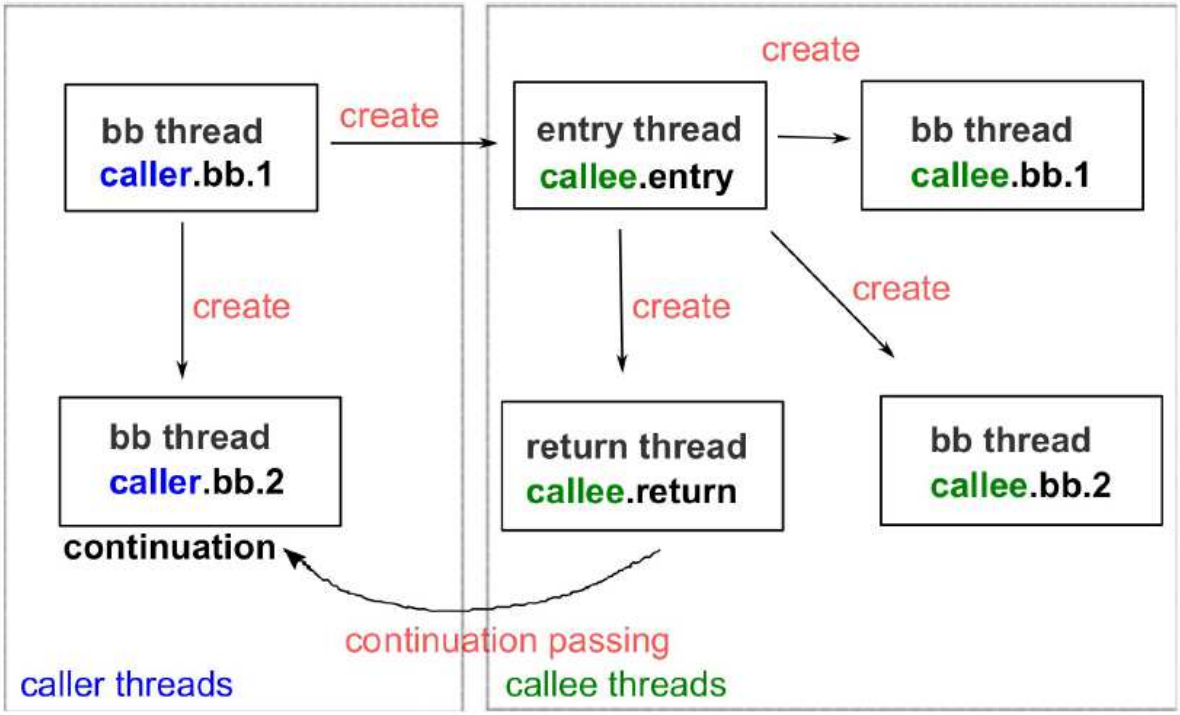


Illustration 4: Modular generation of dataflow threaded code

For pointers and arrays, we currently rely on a conservative scheme that only relies on the coarse but scalable, flow-insensitive, context-insensitive, and field-sensitive pointer analysis implemented in GCC. Our method aims for robustness and generality: arbitrary unstructured control flow with arbitrary memory accesses can be internally represented in GCC using virtual definitions (VDEF) and virtual uses (VUSE) in what is commonly referred to as mem-SSA (a concept introduced as HSSA in the SGI Pro64 compiler, now Open64). These virtual definitions and uses are associated with virtual PHI nodes to reconcile the memory-induced data flow from different control flow branches. Our partitioning and dataflow conversion algorithm can be extended to operate on these virtual operations and operands, at the cost of extra complexity to deal with the initialization of the synchronization counters; indeed, the number of producers is not known anymore at thread creation time in general. Implementation is in progress. We will report on the 3rd year on enhancements of the compilation methods and T* ISA (cf. Deliverable D6.2) will be proposed to efficiently support arbitrary memory-induced data flow and irregular control flow.

To go beyond the conversion of scalar dependencies and the conservative handling of pointer- and array-based dependencies, we leverage pragma annotations defined for our dataflow streaming extension of OpenMP (see D3.2 and D4.2 from the first year's deliverables). These annotations provide an explicit task partitioning and express data dependencies on pointer-based data structures and arrays that cannot be discovered through static analysis. The programmer use pragmas only when necessary to supplement static analysis; he/she is responsible for declaring live-in and live-out values, expressing task-to-task dependencies through intermediate stream variables. Note that this component of the compilation flow is fully functional but not yet integrated with the automatic partitioning and scalar dataflow conversion pass (the integration plan is discussed further in D4.5).

The second phase of the algorithm starts from the finest-grain data-flow threads and aggregates them into a hierarchy of tasks, driven by locality optimization, synchronization grain coarsening. Our heuristic extends Parallel-Stage Decoupled Software Pipelining (PS-DSWP) to operate on arbitrary control flow and SSA programs. It merges tasks hierarchically in a generalized typed loop fusion algorithm, preserving data parallelism of stateless threads and causality (causal scheduling in tasks).

Specific patterns can lead to more efficient, specialized code:

- DOALL: data-parallel loops can be compiled into a pair of splitter and merger tasks enclosing coarser grain independent blocks of iterations

- DOACROSS: loops with dependencies may be compiled into a splitter-merger pair enclosing pipelining stages resulting from the PS-DSWP algorithm.

Our current results come from two separate prototype implementations that are currently being merged into a consistent compilation flow. We target a 24-core Intel Xeon Dunnington server (6-core FSB architecture).

We studied three benchmarks: 1D-FFT, FMradio and 802.11a. Annotating the FMradio code with extended pragmas require little effort and provide up to 18.8x speedup. 802.11a is more unbalanced and harder to parallelize as the original version used global variables extensively; after a re-factoring step, annotating the program is straightforward and leads to 14.9x speedup. 1D-FFT scales up to 7.5x speedup.

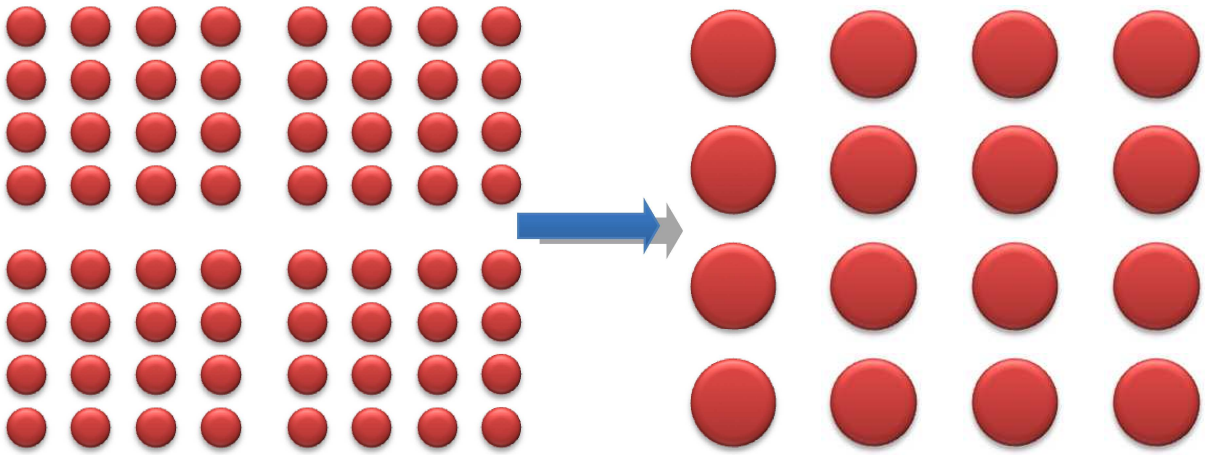
We evaluated our modular code generation algorithm on (tree-)recursive, scalar code only. The Fibonacci and Merge Sort kernels, written in plain C, can be automatically parallelized. This is just a proof of concept and we only measured performance on a small scale machine so far, an Intel Core i7-2720QM 4-Core machine, using our `dfrt` run-time. To coarsen the synchronization grain, we set a threshold to switch back to the sequential version. Fibonacci and Merge Sort reach 2.55x and 2.82x speedups respectively. We are currently working on a large-scale experiment using the same run-time for a cluster of workstations, and using the hardware-assisted implementation of T* instructions in COTSon.

This ongoing work will be presented at the MULTIPROG workshop associated with the HiPEAC'12 conference [2] and at the CGO'12 Student Research Competition.

5.Locality optimization

5.1 Locality optimization in the HMPP Workbench

HMPP provides a set of directives to implement unroll-and-jam, a dedicated form of loop tiling where the inner loops (i.e., the point loops) are fully unrolled (unroll-and-jam can also be seen as outer loop unrolling followed by fusion of the resulting copies of the inner loop). HMPP also provides directives to use device local memories effectively. The figure below describes unroll-and-jam: a square of 4x4 iterations in the source program are unrolled-and-jammed together to form a larger iteration (in the form of a big circle) in the transformed program. These directives are combined with the parallelism scheme ("gridification"); in such a way, the programmer has many ways to tune the mapping of the threads as well as memory accesses to improve data locality.



```
#pragma hmppcg gridify(j,i)
#pragma hmppcg unroll(4), jam(4)
  for( j = 0 ; j < p ; j++ ) {
    for( i = 0 ; i < m ; i++ ) {
     for (k = ...) { ...}
      vout[j][i] = alpha * ...;
    }
  }
```

These additional locality optimization directives are implemented in the HMPP Workbench 3.0 and have been applied to customer applications.

5.2 Locality optimization in the TERAFLUX GCC back-end

To harness the complexity of many-core architectures and complex memory hierarchies, powerful compiler optimizations and especially loop nest transformations are in high demand. We first survey ongoing work on polyhedral compilation for task-level optimizations, then we describe the evolution of our low-level locality optimization method for dataflow threads called "work-streaming". Both activities are conducted at INRIA.

5.2.1 Adapting polyhedral compilation for task-level optimization

The polyhedral framework is showing interesting success in this area. It is an algebraic abstraction for reasoning about loop transformations. It allows to model and apply complex loop nest transformations addressing most of the parallelism and locality-enhancing challenges. The loop control flow constraints, statement dependencies and the memory optimization constraints are used together to construct a unified system of constraints. Resolving this system allows to find a new schedule for the statements and thus to apply complex loop transformations maximizing parallelism and data locality.

INRIA is actively integrating the polyhedral optimization framework in the GCC compiler: the Graphite framework (INRIA also contributes to a mirror project in LLVM called Polly). We are also working towards using the polyhedral framework to target the optimization of dataflow programs and to generate aggressively optimized code starting from high-level languages.

This deliverable summarizes our progress on two challenges arising when adapting the polyhedral framework to work on the level of dataflow tasks: dealing with memory-based dependencies and optimizing very large kernels.

Part of it is a practical compiler construction issue, where upstream passes such as the transformation to three-address code and Partial Redundancy Elimination Constant Subexpression Elimination (PRE/CSE) introduce new scalar variables leading to additional memory-based dependencies. A second difficulty is specific to the compilation of task-based programs. General TERAFLUX tasks in our efficiency programming models may be stateful, although the programmer is discouraged to use state when not necessary for effective memory management. It is necessary to model the memory-based dependence constraints induced by this state. The last difficulty is to identify a profitable tradeoff between memory expansion (privatization, renaming) and parallelism. Memory-based dependencies not only increase the complexity of the optimization but most importantly, they reduce the degree of freedom available to express effective loop nest transformations, limiting the overall effectiveness of the polyhedral framework.

We designed and implemented a technique that solves this problem by allowing a compiler to relax the constraint of memory-based dependencies on loop nest transformations and that does not incur the memory footprint overhead of scalar and array expansion. The proposed technique is based on the concept of polyhedral live range interval interference. A paper was submitted to the ETAPS Compiler Construction (CC'12) conference.

We are also addressing scalability issues [4]. In order to find the most suitable optimization and parallelism opportunities, the polyhedral framework solves a system of constraints. Having a very large number of statements and dependencies make the system harder to solve. Polyhedral techniques are progressively extended to handle tasks as its atom of operation, and supporting larger classes of programs with data-dependent control flow, spanning over full functions. The complexity of the optimization algorithms becomes increasingly problematic.

To address this challenge, we introduce a technique called statement clustering. Motivated by a practical study of the 30 numerical kernels in the Polybench 2.0 suite, we found that statements that are a part of the same strongly connected component in the dependence graph can be lumped together, and that we can look for a schedule for the whole cluster instead of considering each statement individually. When building affine-by-statement schedules, the schedule of each statement at nesting depth *n* induces *n* unknown variables (schedule coefficients) in the constraint system, and possibly many times more when applying the affine form of the Farkas lemma to linearize the problem. With the proposed approach, many of these variables are being equated across the statements belonging to a given cluster at a given nesting depth. Furthermore, all intra-cluster dependencies are removed. This simplification leads to much smaller systems, and is particularly effective on low-level intermediate representations in three-address code. Applying this technique on the `3mm` kernel, for example, reduces the number of variables in the initial constraint system from 36 to 9 variables.

This ongoing work will be presented at the IMPACT'12 workshop [5] and at the CGO'12 Student Research Competition.

5.2.2 Work-streaming compilation for low-overhead dataflow execution

Our code generation framework aims to apply code transformations capable of transforming a streaming OpenMP program [3] into task-level dynamic dataflow threads whose synchronization and communication grain has been tailored to the target architecture.

In a nutshell, *work-streaming* converts general streaming OpenMP programs, with arbitrary dynamic dependence and dynamic task creation patterns, into low-overhead coarser-grain threads operating over generalized form of dataflow I-structures for very fast multi-producer multi-consumer communications. The background for these generalized I-structures consists of a run-time and data structure called Erbium and described in D4.3 (first year deliverable). In the second year, we greatly generalized the compilation method to these low-overhead synchronization and communication structures. We also proved the correctness of dynamic analysis algorithms to detect and eliminate resource deadlocks (buffer sizing), to detect functional deadlock (for debugging purposes).

In the 3rd year, we will further integrate the work-streaming compilation and run-time techniques with the TERAFLUX compilation flow and memory model, mapping the associated concurrent data structure to the Owner Writable Memory defined in D7.1 (first year deliverable).

Due to the length of these contributions and the large amount of background information, we refer to the PhD thesis of Antoniu Pop [1].

6. Conclusion

This deliverable presented the main contributions associated with the compiler and run-time support for the TERAFLUX programming models. Our work focused on Task 4.2, with numerous developments in the exploitation of multi-level parallelism and in the optimization for the memory hierarchy.

We believe that the main static and dynamic techniques are now in place for a complete tool flow to adapt the concurrency exposed in the efficiency programming models to the TERAFLUX architecture. A discussion of the implementation status and integration plan can be found in the conclusion section of D4.5.

7.References

[1] Antoniu Pop. Leveraging Streaming for Deterministic Parallelization : an Integrated Language, Compiler and Runtime Approach. *PhD thesis, MINES ParisTech,* Paris, France, September 2011.

[2] Feng Li, Boris Arnoux and Albert Cohen. A Compiler and Runtime System Perspective to Scalable Data-Flow Computing. *5th MULTIPROG Workshop (associated with HiPEAC'12)*, Paris, France, January 2012.

[3] Antoniu Pop and Albert Cohen. A stream-computing extension to OpenMP. In *Intl. Conf. on High Performance and Embedded Architectures and Compilers (HiPEAC'11)*, January 2011.

[4] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J. Ramanujam, P. Sadayappan, and Nicolas Vasilache. Loop transformations: Convexity, pruning and optimization. In *38*th *Symp. on Principles of Programming Languages (POPL'11)*, Austin, Texas, January 2011.

[5] Riyadh Baghdadi, Albert Cohen, and Konrad Trifunović. Using Live Range Non-Interference Constraints to Enable Polyhedral Loop Transformations . *In the 2nd International Workshop on Polyhedral Compilation Techniques (IMPACT'12)*, Paris, France, January 2012.