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

PROJECT NUMBER: 249013

Exploiting dataflow parallelism in Teradevice Computing

D4.6 – Thorough evaluation of the compilation tools, productivity and performance portability

Due date of deliverable: 31st December 2012
Actual Submission: 20th December 2012

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	**Mikel Lujan**	**UNIMAN**	**Scala compilation flow**
1.2	**Souad Koliai, Arne Garbade**	**UDEL, UAU**	**Corrections, comments**
1.3	**Rosa Badia, Albert Cohen**	**BSC, INRIA**	**Corrections, precisions**
1.4	**Albert Cohen, Roberto Giorgi**	**INRIA, UNISI**	**Final corrections**

Release Approval

Name	Role	Date
Albert Cohen	**Originator**	**04/12/2012**
Albert Cohen	**WP Leader**	**12/12/2012**
Roberto Giorgi	**Project Coordinator for formal deliverable**	**06/12/2012**

TABLE OF CONTENTS

Project: **TERAFLUX -** Exploiting dataflow parallelism in Teradevice Computing
Grant Agreement Number: **249013**
Call: FET proactive 1: Concurrent Tera-device Computing (ICT-2009.8.1)

Albert Cohen, François Gindraud, Feng Li, Antoniu Pop
INRIA

Rosa Badia, Guillermo Miranda
BSC

Ian Watson, Salman Khan, Daniel Goodman and Mikel Lujan
UNIMAN

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

OpenStream – Data-flow streaming extension of OpenMP, for the C language, implemented as a patch to GCC and a dedicate runtime system for data-flow tasks.

http://www.di.ens.fr/OpenStream

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://pm.bsc.es/ompss

2. Executive Summary

We report on the multiple advances that took place in the third year of the project towards the construction of a complete tool chain for TERAFLUX.

In particular, we describe the complete compilation flow from the StarSs efficiency language down to the TERAFLUX instruction set, and the associated runtime system components. The tool flow uses data-flow streaming extensions of OpenMP, called OpenStream, as an intermediate step in the adaptation of StarSs programs for the compilation on the T* feed-forward data-flow execution model.

This tool flow is distributed as free software as a patch to GCC 4.7.1, with a complete set of benchmarks selected from the comprehensive list of applications characterized in WP2, and with 19 tutorial examples introducing the language constructs of OpenStream.

We also report on the current state of the integration of the Scala tool flow with the TERAFLUX back-end compiler (GCC generating T* instructions).

Experiments currently target a software runtime running on native hardware. These experiments allowed us to perform a thorough validation of the design of the tool flow, comparing against state-of-the-art programming models and implementations. The experiments also helped identify missing features and a precise roadmap for the completion of the flow to enable the execution of larger StarSs applications. The detailed results can be found in the associated papers published at IEEE Micro, ACM TACO (HiPEAC 2013) and PPoPP.

Performance experiments on the TERAFLUX instruction set simulator, scaling to multiple node configurations, and with different timing models, will be conducted in the fourth year of the project.

3.Introduction

The overall objective of WP4 is the development of compilation and runtime 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 runtime 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. The compiler tools need 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. It also needs to generate tightly scheduled, fine-grain vectorized computation kernels, possibly targeting accelerators.

We report on major advances in the compilation of the data-flow streaming extension of OpenMP (now called OpenStream) to the TERAFLUX instruction set, in the design, experimentation, on the runtime support for this language, and on the ongoing implementation of a systematic translation of StarSs to OpenStream for execution on the TERAFLUX simulator.

3.1. Document structure

Section 4 reports on the back-end compiler, implemented in GCC, and mapping OpenStream to the TERAFLUX instruction set. A complete flow including the code generation method, runtime support, and experimental results are provided.

Section 5 reports on a formal and experimental study of the data-flow runtime system supporting the execution of data-flow tasks on conventional hardware. This thorough study will be useful for the design and implementation of future software/hardware interfaces.

Section 6 reports on the translation of StarSs to OpenStream and preliminary experiments in the direction of a complete integration of the tool flow for efficiency languages down to the TERAFLUX instruction set.

Section 7 reports on the integration of the Scala tool flow with the back-end compiler. While the main research route for Scala is based on the Java VM, an integrated route has been hard to implement due to technical limitations of the available tools. UNIMAN has been able to propose a new integration path leveraging the TERAFLUX backend through a different set of tools.

3.2. Relation to other deliverables

This deliverable extends the compilation algorithms described in D4.1, D4.3 and D4.4, and evaluates them on a range of representative benchmarks. It also complements D3.3 with an evaluation of the compilation a runtime support to execute StarSs applications on the TERAFLUX architecture.

3.3. Activities referred by this deliverable

This deliverable is associated with and represents the results of Task 4.2 and Task 4.3. Most of the work is related to Task 4.2. Indeed, during the course of the project it became apparent that the largest effort on transactional memory would be concentrated on the language support and on the architecture extensions (as reported in WP3 and WP6). Task 4.3 has been increasingly concentrating on compiler engineering to combine the data-flow and transactional memory compilation flow and runtime, with little resources left for research on optimizations.

4. Evaluation of the Back-End Compiler for TERAFLUX

We present a vastly remodeled compilation flow for the TERAFLUX architecture, implemented as a front- and middle-end extension to GCC 4.7.1, expanding streaming task directives into data-flow threads and point-to-point communications. It is the first complete, fully automatic compilation framework for OpenStream, the new name of the data-flow streaming extensions of OpenMP designed by INRIA in the TERAFLUX project.

We first recall the feed-forward data-flow execution model we are targeting in the project, and discuss why it challenges the classical compilation methods for parallel languages, and some constraints this model imposes on our stream programming model. Then we detail the code generation algorithm and the main features of the implementation. The reader may refer to [1] and [3] for details on the compilation of OpenStream and on the automatic extraction of fine-grain threads from arbitrary procedural control flow, respectively.

4.1. Feed-forward data-flow model: interfaces and challenges

Our code generation pass targets an abstract data-flow interface, designed after DTA (the data-driven execution model) [6] and the T* ISA [5] (as defined in the WP6 and WP7 deliverables). The interface defines two main components: *data-flow threads*, or simply *threads*, when clear from the context, together with their associated *data-flow frames*, or *frames*.

The frame of a data-flow thread stores its input values, and may also store local variables or thread metadata. The address of this data-flow frame also serves as an identifier for the thread itself, to synchronize producers with consumers. Communications between threads are single-sided: the producer thread knows the address of the data-flow frames of its dependent, consumer threads. A thread writes its output data directly into the data-flow frames of its consumers.

Each thread is associated with a *synchronization counter* (SC) to track the satisfaction of producer-consumer dependences: upon termination of a thread, the SC of its dependent threads is decremented. A thread may execute as soon as its SC reaches 0, which may be determined immediately when the producer decrements the SC. The initial value of the SC is equal to the amount of data that needs to be externally written to its frame plus the number of consumer threads to which it connects. In our implementation, the producer responsible of the last decrementation on a thread directly schedules the consumer for execution. This token-less driven execution is one of the strengths of this form of point-to-point synchronization.

In contrast, token-based approaches require checking the presence of the necessary tokens on incoming edges. This means that either (1) a scanner must periodically check the schedulability of data-flow threads, or (2) data-flow threads are suspendable. The former poses performance and scalability issues, while the latter requires execution under complex stack systems (e.g., cactus stacks) that may introduce artificial constraints on the schedule. The SC aggregates the information on the present and missing tokens for a thread's execution, allowing producer threads to decide when a given consumer is ready for execution.

Four primitives manage threads and frames. They are implemented as compiler builtins, recognized as primitive operations of the compiler's intermediate representation. In this deliverable, performance experiments use a software runtime for the timing measurements. But code generation for actual T* instructions and execution on COTSon is also supported, as reported in D7.4.

- `void *df_tcreate(void (*func)(), int sc, int size);` Creates a new data-flow thread and allocates its associated frame. `func` is a pointer to the argument-less function to be executed by the data-flow thread, `sc` is the initial value of the thread's synchronization counter, and `size` is the size of the data-flow frame to be allocated. It returns a pointer to the allocated frame. Once created, a thread cannot be canceled. Collection of thread resources is triggered by the completion of the thread's execution.
- `void df_tdecrease(void *fp, int num);` Marks the thread designated by frame pointer `fp` to be decremented by `num` upon termination of the current thread.
- `void df_tend();` Terminates the current thread and deallocates its frame.
- `void *df_tget_cfp();` Returns the frame pointer of the current thread.

Due to the underlying data-flow execution model, and its semantic requirement of writing stream data directly in the consumer's data-flow frame, we need to impose two simple restrictions on our programming model for this compilation path: (1) on a given stream, the horizon of all consumer tasks must be an integer multiple of the bursts of producer tasks (i.e., a producer's output window on a stream cannot be split between multiple consumer input windows); and (2) the burst of a consumer is always either 0 or equal to its horizon (i.e., when a task peeks on a stream, it cannot simultaneously advance the stream's read index).

The purpose of these restrictions is to ensure that any given output window on a stream cannot be split among multiple consumer windows. If a producer's output window must be split dynamically between multiple consumers, then each write access to the output window must be guarded by a conditional expression or made through an indirection. This would prevent us from generating optimized code where the producer writes its outputs directly in the data-flow frame of the consumer.

These constraints can be relaxed, but not without a performance overhead, or extending our target abstract data-flow interface and execution model. Other compilation paths have been explored and evaluated, where these restrictions do not apply, as described in D4.1 (see the section on work-streaming), but we preferred to focus on the complete automation of a compilation flow, supporting all the features of the programming model, even the most dynamic ones, and delivering an efficient execution on the TERAFLUX instruction set. Furthermore, these restrictions only bear on some advanced stream-oriented features of the language, like the ability to compute over sliding windows on a stream of data. We plan to support these features, and remove any restrictions, in future work.

4.2. Compiling streaming tasks to data-flow threads

The data-flow compilation path for OpenStream does not rely on streams for communication, but rather as a meeting point for producers and consumers of data. Streams record the production and consumption schedules, matching each producer with its consumer(s). *Before* it can start executing, a producer must acquire the locations, within its consumers' data-flow frames, where it needs to write its output data. While this adds some overhead, it is an essential part of our execution model that provides outweighing benefits, as evaluated in the next section. To illustrate the compilation process, we rely on some trivial examples of streaming tasks that exhibit the key characteristics required to explain the important parts of the code generation algorithm.

For each data-flow thread (or task instance), we keep track of the information required for the stream matching scheme and for the synchronization algorithm with a metadata block embedded within the thread's frame. Figure 1 shows an example of two streaming tasks and the two key data structures we use: the *frames* and the *views*. The former hold the metadata and the input data required for executing a data-flow thread, the latter are used to implement stream access windows.

```
int x __attribute__((stream));
#pragma omp task output (x)            // T1
  x = ...;
#pragma omp task input (x) output (y) // T2
  y = foo (x);
```

```
struct view {
  // pointer to the data
  // accessed through the window
  void *data;
  // pointer to owner frame
  // (always the consumer thread)
  frame_p owner;
} view_t, *view_p;
```

```
struct frame {
  int synchronization_counter;
  view_t view_x;
  view_t view_y;
  view_t ...

  void *data_block;
} frame_t, *frame_p;
```

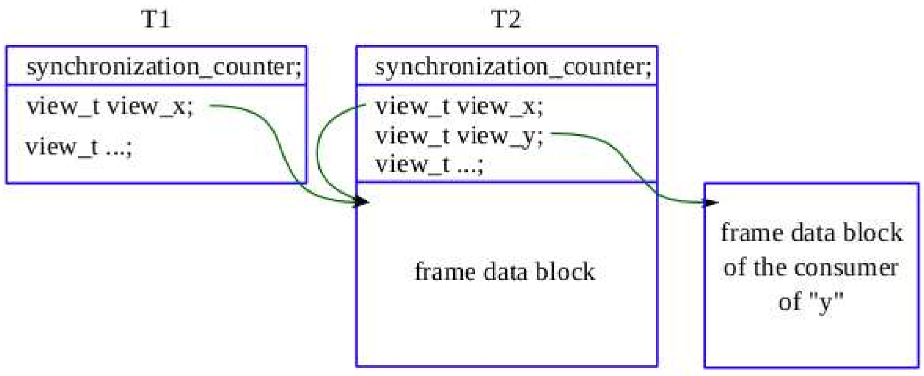
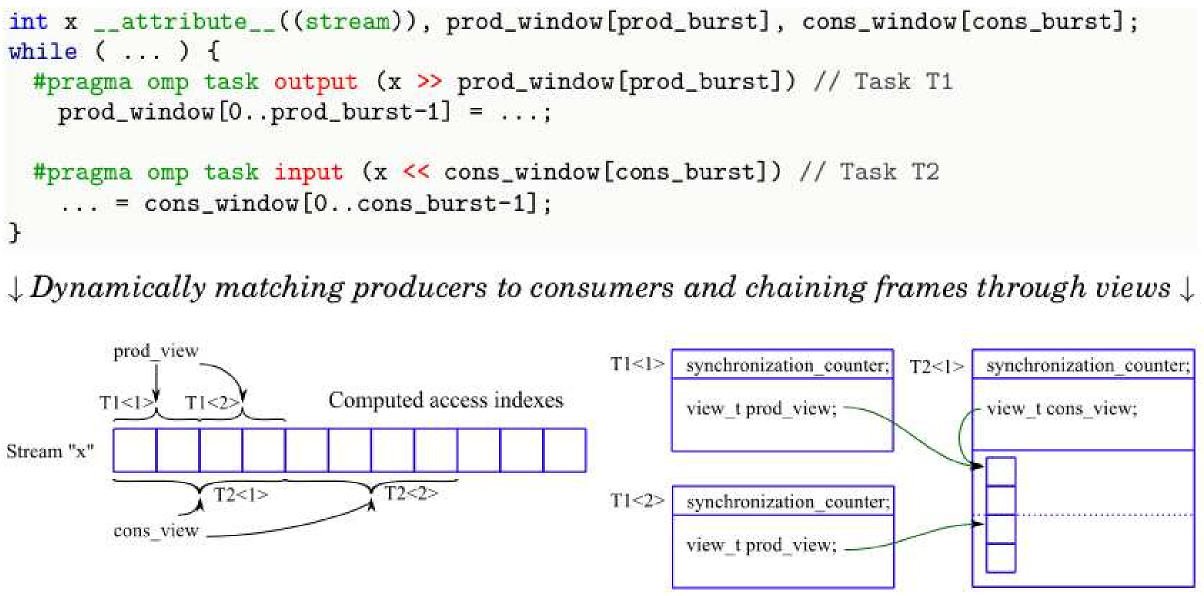


Figure 1: Streaming tasks communicating via data-flow frames

The top of Figure 1 shows an example where two tasks communicate through stream `x`. Task `T1` is the producer and `T2` the consumer, both using an implicit window to access the stream. The middle section of the figure shows (on the left) the `view` data structure. It contains a pointer to the actual data that a data-flow thread, which is an instance of a given task, is allowed to access within a conceptual stream through a window. In addition, the `view` data structure contains a pointer to the data-flow frame of the owner of the data, which is *always* the consumer thread. Indeed, for an output window, the view's "data" pointer gives access to a location within another thread's frame, while for an input window, this pointer points within the thread's own frame. On the right, the `frame` data structure shows a skeleton of what a frame might look like. Depending on a thread's inputs and outputs, each frame has a possibly unique structure, but respecting this layout: it always contains the synchronization counter, a set of views corresponding to the different stream access windows the task annotation uses and a `data_block`. The latter is not a pointer to a separately allocated buffer, but is just used as a marker for the beginning (offset) of the data block. The bottom of Figure 1 shows how the data-flow frames for tasks `T1` and `T2` will be chained at runtime, by means of the `view` metadata. The frame of `T1` contains a view for the (implicit) output window "x", which points to the data block of its consumer, within the frame of `T2`. Furthermore, the frame of `T2` contains a view for the input window "x", which points to its own data block, but also a view for the output window "y" pointing to its consumer's frame data block once it is determined.

The following step, presented in Figure 2, shows how the matching of producers and consumers is orchestrated by the runtime. The figure illustrates the work performed by the `stream_match_views` runtime function.



```
int x __attribute__((stream)), prod_window[prod_burst], cons_window[cons_burst];
while ( ... ) {
  #pragma omp task output (x >> prod_window[prod_burst]) // Task T1
    prod_window[0..prod_burst-1] = ...;

  #pragma omp task input (x << cons_window[cons_burst]) // Task T2
    ... = cons_window[0..cons_burst-1];
}
```

↓ Dynamically matching producers to consumers and chaining frames through views ↓

Figure 2: General scheme to match producers and consumers at run time using streams

```
int X[x_burst], Y[y_burst];

#pragma omp task input (x >> X[x_burst]) output (y << Y[y_burst])
   foo (X, Y);
```

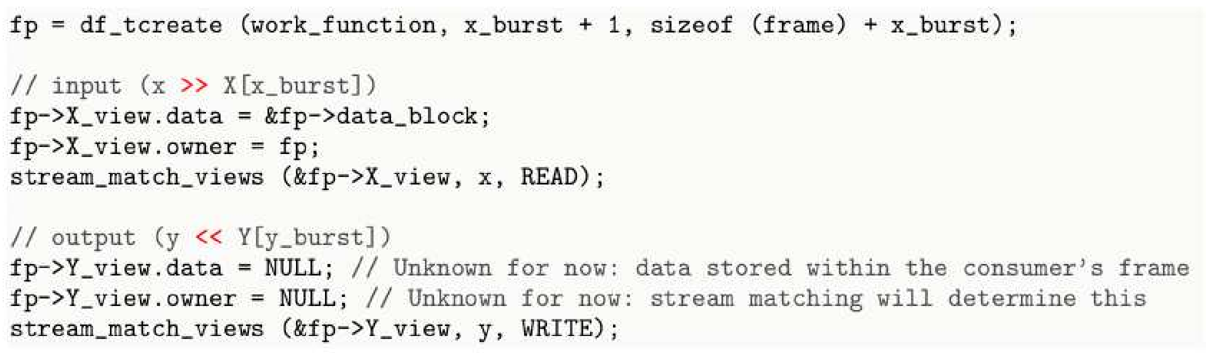
↓ *Work function code generation* ↓

```
void work_function (void) {
  frame_type *fp = df_tget_cfp ();

  foo (fp->view_X.data, fp->view_Y.data); // Typically inlined work-function

  df_tdecrease (fp->view_Y.owner, y_burst); // Owned by the consumer of stream ''y''
  df_tend ();
}
```

We rely here on a slightly more complex set of streaming tasks (top) which communicate through a stream `x` with explicit stream access windows, where the producer and the consumer bursts are non-trivial. We conceptually represent the stream and the stream accesses of both tasks on the bottom part of Figure 2, instantiating for the sake of illustration with *prod_burst* =2 and *cons_burst* =4. As shown on the right side (bottom) of the figure, two instances of the producer task, represented by the two frames `T1<1>` and `T1<2>` are necessary to produce the data for one instance of the consumer task. The stream matching not only sets the `owner` field of each view, but it also computes the appropriate offset in the frame of a consumer to ensure that the producer's view always points directly to the adequate memory location. For instance, thread `T1<2>`, which is the second instance of the producer task `T1`, produces the second half of the data accessed by the first instance `T2<1>` of the consumer task `T2` through its window.

First, the work function of a task is generated, with a streaming task that consumes data on a stream `x` and produces data on an output stream `y`, both accessed through their respective windows. The work function (bottom) consists of the body of the task annotation, outlined to a new function with no arguments. The input parameters are all stored within a thread's frame, which can be accessed through the frame pointer returned by a call to the `df_tget_cfp` runtime function. Within the body of the original task, each stream access window is replaced with an indirection through the `data` field of the corresponding view. Then, a call to `df_tdecrease` is issued, at the function's exit, for each output frame. This call is used to implement the synchronization algorithm: it atomically decrements the consumer's (owner of the view) synchronization counter by a `y_burst`, which represents the amount of data effectively produced and written for this consumer. This call further contains a test that schedules the consumer thread on the ready (work-stealing) queue if the synchronization counter reached zero. Finally, each work function contains a last call to the `df_tend` function to deallocate the frame and perform any necessary cleanup operations.

Finally, the code generation for the control program is presented in Figure 3.



```
fp = df_tcreate (work_function, x_burst + 1, sizeof (frame) + x_burst);

// input (x >> X[x_burst])
fp->X_view.data = &fp->data_block;
fp->X_view.owner = fp;
stream_match_views (&fp->X_view, x, READ);

// output (y << Y[y_burst])
fp->Y_view.data = NULL; // Unknown for now: data stored within the consumer's frame
fp->Y_view.owner = NULL; // Unknown for now: stream matching will determine this
stream_match_views (&fp->Y_view, y, WRITE);
```

Figure 3: Generated data-flow code from OpenStream pragmas

Figure 3 shows, on the same example, the code generated at the site of the original `pragma` annotation to allocate and prepare the data-flow frame. We first issue a call to `df_tcreate`, which allocates a data-flow frame for one instance of the task, passing a pointer to the work function, the initial synchronization counter and the size of the frame. The initial synchronization counter corresponds to the amount of input data required for a thread's execution (in this example `x_burst`, plus the number of output views of this thread, here 1). This additional synchronization value is decremented, by the `stream_match_views` function, every time a consumer view is matched to one of this thread's output views. The size of the frame is computed by adding the size of the application data stored in the frame, which is the amount of input data, to the size of the frame's metadata. After this, we generate, for each streaming clause, initialization code for the views created to implement the stream access windows. Finally, we issue, for each stream accessed by the task, a call to `stream_match_views`, which implements the matching algorithm, by setting the metadata for output views and decrementing the frame's synchronization counter by one for each output view that has been properly matched to a consumer.

4.3. Transactions within data-flow threads.

The compilation flow described above also support closed transactions, as long as they are restricted to the scope of data-flow tasks. The baseline support was implemented by recent versions of GCC, following the ABI defined by Intel and standardized across multiple compilers and software transactional memory libraries. A first implementation was designed initially through the support of the HiPEAC network, then redesigned and completed in the context of the VELOX FP7 project, and finally merged with the compilation flow and runtime system of TERAFLUX in the OpenStream patch of GCC 4.7.1.

Preliminary evaluation confirmed the smooth integration of the two compilation flows and runtimes (GCC's STM and OpenStream runtime). More extensive optimizations and experiments will be conducted in the consolidated tool flow during the fourth year of the project, validating the potential of combined data-flow and transactional memory semantics in applications, languages, compilers, runtime systems, and architectures.

4.4. Implementation

A full implementation of the code generation pass used for lowering OpenStream annotations to the data-flow runtime is publicly available, supporting all features of the stream-computing extension presented in this deliverable. This implementation builds on top of the GCC compiler's OpenMP expansion pass and targets a separate runtime library, which implements stream dynamic matching and the point-to-point synchronization scheme detailed above.

The compiler's front-end is modified to parse streaming annotations, as well as stream attributes, and lower them to GCC's intermediate representation, preserving stream typing information. This typing information is used both to enable modular compilation, with a clean interface between translation units, and to perform type checking providing compile-time feedback when stream types are incompatible.

Frame and view data structures are fully constructed and typed to facilitate debugging. This allows to dump the intermediate representation, using the classical GCC `-fdump-tree-*` flags, in a human-readable format where each structure's field accesses are easily identifiable rather than just an offset. This mitigates part of the drawback of not relying on a source-to-source compiler where the output can be directly checked.

On the OpenStream git repository, the code generation is integrated in the OpenMP expansion pass in the middle-end, and is activated with the same compilation flag, `-fopenmp`. The generated code does not target GCC's `libGOMP` OpenMP runtime but our own runtime library, `libwstream_df`.

To target the TERAFLUX instructions and the COTSon-based simulation of the architecture, an alternate branch of the OpenSteam tool flow must be downloaded and compiled. It is called OpenStream_TFX (`git branch origin/OpenStream_TFX`), developed also as part of WP7 (see D7.4).

To facilitate the build, test, and performance evaluation of OpenStream benchmarks on the TERAFLUX architecture simulator, a comprehensive Wiki page has been set up on the main repository of the tool flow:

http://sourceforge.net/p/open-stream/wiki/Home

More examples and automatic performance evaluation scripts will be added continuously until the end of the project.

Note that transactional memory has been demonstrated to smoothly coexist with data-flow tasks in OpenStream, but only in the context of the software runtime system implementation at this point.

4.5. Systematic performance evaluation

We evaluated the OpenStream tool flow targeting to our software runtime implementation to compare its performance (scalability and efficiency) against Cilk and native StarSs implementations on a selection of benchmarks.

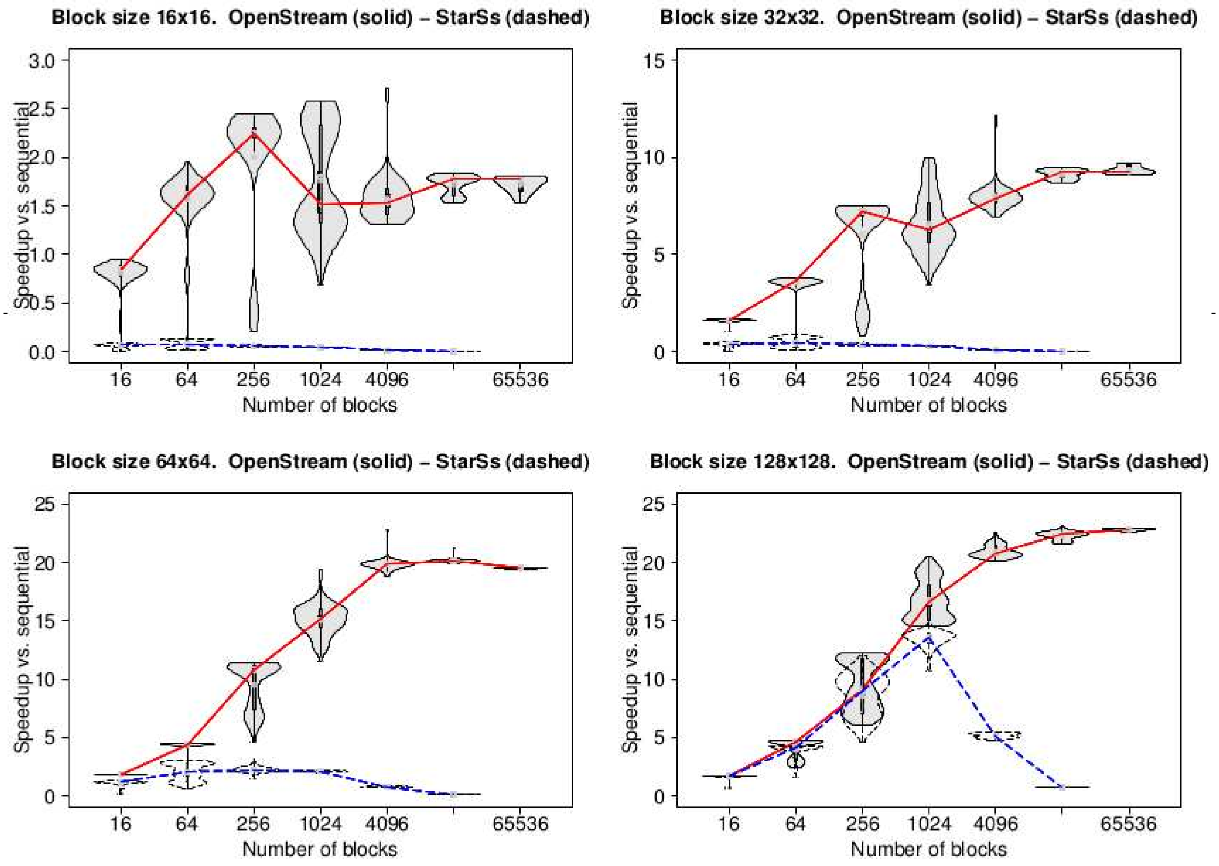


Figure 4: Sparse-LU benchmark using OpenStream and T vs. the conventional StarSs runtime*

Figure 4 compares an OpenStream-translated version of SparseLU with the native StarSs version, running with our runtime system on a 24 core Opteron.

Clearly, the benchmark scales much earlier with the data-flow execution model of TERAFLUX, even using a software runtime. Blocks of size 32x32 are sufficient, while StarSs using a more complex suspendable lightweight threading runtime still has scalability problems with blocks of size 128x128. This scalability penalty is by no means a limitation of the StarSs language. On the contrary, the experiment validates the benefits of converting the region-based task-to-task dependences of StarSs into streamlined, T* primitives and data-flow threads.

The generated code works on the COTSon implementation of the TERAFLUX instruction set.

We refer to [1] for additional and detailed performance experiments, including a systematic study of Gauss-Seidel illustrating the compilation and execution of a StarSs program with partially overlapping regions.

5. Thorough Study of a Runtime for Data-Flow Tasks

While designing a robust tool flow for the productivity and efficiency languages in TERAFLUX, we came across the challenge of designing and implementing a reliable, verified, and highly efficient runtime to implement the baseline data-flow execution model of the project. This runtime can be used on conventional hardware or on the TERAFLUX architecture (see D6.2), by supporting the TSU underneath the COTSon simulator interfaces. In both cases, correctness and performance are of utmost importance for the practicality of all the experiments and integration efforts.

At the time of writing, the TERAFLUX memory model has made much progress, with a partial implementation and formalization available for OWM regions (see the WP3 deliverable for the current status). Yet the state of the design and implementation of the model is still too preliminary to perform a thorough study of the correctness of our execution model implementation. We thus decided to focus on a very weak memory consistency model of an existing architecture, as a first experiment to validate our approach, and before moving to the full-scale implementation of the TERAFLUX memory model. We selected the very similar, weakly consistent memory models of the POWER and ARMv7 instruction set architectures.

We demonstrate that a high degree of confidence can be achieved for highly optimized, real-world concurrent algorithms such as a lightweight data-flow task scheduler on a weak memory model. More specifically, we study the Chase-Lev concurrent doubly-ended queue, an essential component of parallel programming language implementations, as the cornerstone of most work-stealing schedulers.

Until now, a formal correctness proof for a relaxed memory model has been missing for this concurrent algorithm. Furthermore, while work-stealing has met with wide success on x86, few experiments target weaker memory models. This is an important missing link for an implementation supporting the TERAFLUX instruction set and memory model.

We provide the first proof of correctness of this important concurrent data structure for a relaxed memory model. Specifically, the proof targets a very recent, experimentally and expert-verified axiomatic semantics for the POWER and ARMv7 memory models. This proof and the proof technique constitute our first contribution. We compare our optimized implementation with an x86 version and two portable ISO C 2011 variants (C11 for short): a canonical translation of the algorithm's sequentially consistent accesses, and an aggressively optimized version making full use of the acquire--release and relaxed semantics of C11 low-level atomics. We show that the POWER/ARM proof can be simplified and tailored to these alternative implementations. We also observe a slight mismatch between the C11 and POWER/ARM memory models, noting unrecoverable overheads in the interaction between atomic operations and the non-cumulativity of memory barriers. The study of similar possible mismatches and performance overheads in the context of the TERAFLUX memory model will be of particular interest in the fourth year of the project.

We evaluate all four doubly-ended queue implementations in the context of a work-stealing scheduler with diverse worker/thief configurations, including a synthetic benchmark with two different workloads, and standard task-parallel kernels. Our experiments demonstrate the impact of the memory barrier optimization on the throughput of our work-stealing runtime.

More details on the actual implementation of the deque, on the formalization and proof, and on the experimental results can be found in [2].

6. Compilation of StarSs Applications

StarSs relies on compiler directives for blocking asynchronous operations into coroutines similar to OpenMP tasks. It provides additional clauses to describe the memory accesses of each task, from which inter-task dependences are inferred. In its latest evolution, accesses are specified with *dynamic array regions*, providing a lot of flexibility to programmers and an incremental path to parallelize existing programs. The price for this rich, implicit dependence abstraction is paid through the need for a sophisticated runtime algorithm. A *runtime dependence resolver* detects the effective overlaps between the memory accesses of different task instances and the ordering constraints deriving from the task creation order.

We briefly recall the syntax and informal semantics of the StarSs programming model. We also analyze the workings of the array region support it provides. For more information, see the reference on StarSs in this deliverable's glossary.

In the deliverable D2.3, we illustrate the translation of a StarSs implementation of the Gauss-Seidel stencil kernel (e.g., heat transfer simulation) to OpenStream. The deliverable also highlights how partially overlapping regions (different array blocks with non-empty intersection) can be managed through the implementation of a dependence resolver capturing the exact data-flow dependences between tasks. In the present deliverable, we dig further into the systematic translation and construction of such a resolver on top of OpenStream. More technical details can be found in [1].

6.1. Translating StarSs into OpenStream

In this section, our objective is to show that OpenStream can form a common ground for the convergence of different efficiency languages into a common tool flow targeting the TERAFLUX instruction set. We show that OpenStream constructs can be used to express the dependencies between tasks working on shared data, using the dependence information provided by the StarSs resolver. We show that such an embedding can be implemented at compilation time, generating the adequate synchronizations with data-flow streaming constructs. We use StarSs to illustrate this embedding, but the same process applies more generally to any higher-level language for parallel-programming that handles dynamic dependences between tasks. In particular, the other efficiency languages studied in WP3, TFLUX from UCY and HMPP from CAPS, can be handled the same way.

Importantly, the resolver of dynamic dependencies expressed *implicitly* in StarSs and other high-level languages is a necessary component, provided by any language where dependencies are not specified by programmers. We do not address the design and optimization of such resolvers, but rather we use this particular example to show that task graphs can be built dynamically. This allows expressing the semantics of the dynamic constructs found in such high-level languages. We show how the *output* commonly available from such dependence resolvers can be used to lower StarSs constructs to OpenStream directives.

The key insight behind our translation scheme is that StarSs array regions, or any memory location, can be encoded by a stream as a sequence of versions, enforcing a form of *dynamic single assignment* on each version. To comply with the in-place update policy of StarSs, we restrict the live range of each stream to one single version/element: a single instance of the data is alive at any time in shared memory. For example, the degenerated case where array regions are guaranteed (e.g., by programming language semantics) to always either fully overlap or be disjoint, which means that each region can be assimilated to a scalar related to its dependencies, can be directly handled without additional support from a dependence resolver. Indeed, in our scheme, this case only requires of the resolver to perform an identity function.

The StarSs dependence resolver is marginally modified to attach a stream to each StarSs region and to return two sets of streams for each dynamic task instance *T*:

- The set of streams attached: (1) to any region that overlaps with the write regions of task *T* (`output` and `inout`); or (2) to any write region that overlaps with the read regions of task *T* (`input` and `inout`); or (3) to any of the own access regions of task *T*. We will call this set `streams_peek(T)`.
- The set of streams attached to the regions of task *T*, irrespectively of their type. We call this set `streams_out(T)`.

Implicitly, each stream attached to a StarSs region is initialized with an element representing the initial state of the region.

Figure 5 illustrates the translation of Gauss-Seidel to OpenStream.

```
for (iter = 0; iter < numiters; iter++)
  for (i = 1; i < N-1; i += B)
    for (j = 1; j < N-1; j += B) {
        starss_resolve_dependences (region_descriptors, &streams_peek, &streams_out,
                                    &num_streams_peek, &num_streams_out);

#pragma omp task peek (streams_peek >> peek_view[num_streams_peek][0])        \
                 output (streams_out << out_view[num_streams_out][1])
        {
          for (k = i; k < i + B; ++k)
            for (l = j; l < j + B; ++l)
              data[k][l] = 0.2 * (data[k][l] + data[k-1][l] + data[k+1][l]
                                  + data[k][l-1] + data[k][l+1]);
        }

        for (k = 0; k < num_streams_out; ++k) {
#pragma omp tick (streams_out[k] >> 1)
          }
      }
```

Figure 5: Example translation of StarSs to OpenStream: Gauss-Seidel

For each iteration of the outer loop on iteration tiles, the program first invokes the StarSs dependence resolver, passing a set of region descriptors built in the same way as in the StarSs compilation framework, and obtaining in return the two sets of streams `streams_peek` and `streams_out` as defined above, as well as the number of streams in each set. We then replace the original StarSs task annotation with our own task annotation with two clauses: (1) a variadic `peek` clause for all streams in the `streams_peek` array; and (2) a variadic `output` clause for all streams in the `streams_out` array. Finally, we issue a `tick` directive for each stream in `streams_out`.

The semantics of the code we generate is quite natural: the `peek` clauses request reading the current live version of a region, which enforces the order of the current tasks execution after the last task that invalidated that region, and the `output` clause generates a new version for each invalidated regions, followed by a `tick` directive to prevent any subsequent tasks from accessing the old version of the region.

6.2. Correctness of the translation

Let us now verify that all dependencies are properly enforced in our resulting code. Consider two regions *A* and *B*, accessed by two different tasks T_A and T_B . T_A and T_B are created in this order by the control program, such that the two regions overlap. There is a dependence $A \rightarrow B$ if at least one of the accesses is a write operation, so if either of the regions is `output` or `inout` on its respective task. We distinguish flow-, anti- and output-dependences (respectively read-after-write, write-after-read and write-after-write dependencies). As StarSs does not expand shared data into private copies, we enforce all of these dependencies.

Let s_A and s_B be the two streams attached to regions *A* and *B*. Streams inherently enforce flow dependences between their producers and consumers for each element, so in order for the dependence $A \rightarrow B$ to be properly enforced, it is necessary and sufficient for T_A to be producer of an element, of a stream *s*, consumed by T_B.

If T_A writes to region *A*, then it generates a new version on stream s_A. We deduce that the dependence resolver must return S_A belongs to `streams_peek`(T_B), irrespectively of the nature of the access to *B*, and therefore T_B reads that element from the stream s_A, preventing the execution of T_B before T_A completes. This means that both flow- and output-dependences are properly synchronized.

If T_A reads from region *A*, then we only need to enforce the dependence if T_B writes to *B*. As by definition s_A belongs to `streams_out`(T_A), T_A produces a new version on stream s_A. Similarly to the previous case, the resolver must return s_A belongs to `streams_peek`(T_B) because T_B writes to *B* and we deduce that T_B reads the element written by T_A in s_A, therefore synchronizing the anti-dependence.

We can also verify that we do not over-synchronize read-after-read dependencies by noting that s_A belongs to `streams_peek`(T_B), with one notable exception: if $A=B$. Indeed, in that case $s_A=s_B$ and we always have s_B belongs to `streams_peek`(T_B). This exception is necessary to enforce anti-dependencies transitively when successive read accesses to the same region discard older versions. Consider, for example, three task instances T_A, T_B and T_C such that all access the same region *A*, T_A and T_B read from *A* and T_C writes to *A*. In this case, enforcing the order T_A happens before T_B allows guaranteeing that T_A happens before T_C, which cannot be guaranteed otherwise as the version created by T_A in s_A is discarded by T_B's new version.

This over-synchronization can be avoided by creating a new region for T_B, that overlaps with A, but with its own stream. However, our performance results show no significant degradation without this optimization.

6.3. Implementation

This translation scheme is much simpler than one would anticipate given the semantic gap between dynamic array regions and data-flow streams. We also show in [1] that it is quite efficient.

Full automation is in process, in collaboration between INRIA and BSC. The following steps have been agreed upon by the partners. The first three of them have been completed at the time of writing:

1. Multiple communications (phone meetings, project meetings) between all the involved people at INRIA and BSC to establish a common, integrated flow from StarSs to the TERAFLUX architecture. The preliminary integration plan was presented at the second year project review.
2. One-to-one exchanges between Antoniu Pop (INRIA) and Guillermo Miranda (BSC), allowing Antoniu Pop to perform systematic performance experiments with different compilation and translation strategies.
3. Revision of the integration plan, resulting in the following three steps listed below.
4. First implementation of a StarSs→OpenStream translation in the OmpSs framework, following the new integration plan. It will only handle full region matches in the dependence resolver, translating region descriptors to streams directly. This is ongoing work at BSC.
5. Systematic experiments with StarSs applications, realized jointly between INRIA and BSC, for the subset that matches the above mentioned restriction. These experiments will aim to demonstrate the practicality and performance of the integrated flow on the TERAFLUX architecture simulator (based on COTSon). A joint publication could be aimed for in the fourth year of the project.
6. Extension of the translation to handle more complex, partial region matches, as described above in this section. Whether this step will be implemented in the OmpSs framework or in a dedicated StarSs-inspired extension of OpenStream remains to be decided and will in any case be implemented in collaboration between BSC and INRIA.

7. Scala to C++ Translation

To complete this deliverable, let us describe the status of the integrated tool flow for the TERAFLUX productivity language approach, which is based on the Scala language. Scala is normally compiled to Java bytecode and run on a Java Virtual Machine. The original TERAFLUX project plan envisaged that we would use the GCJ version of the GCC compiler to compile bytecode to native machine code. This way, we would have been able to make use of the wider project C++ compilation route tools as well as achieve higher performance.

Initial experiments with GCJ, using simple Scala programs, indicated that this route appeared viable. However, in order to add extra functionality, particularly Transactions, to the existing language we needed to modify the bytecode emitted by the Scala compiler. At this point, we discovered that GCJ was not able to handle general bytecode but was tailored specifically to sequences emitted by a particular Java compiler and would not handle our modified code.

As an alternative, we investigated the use of Java Virtual Machine (JVM) facilities which permit the static generation of machine code. This allows the configuration of time critical parts of a program to avoid the overhead of dynamic compilation each time the program runs. The WP2 and WP3 deliverable provide experimental results with this compilation flow and virtual execution environment for the Scala extensions of the TERAFLUX project. Although this route is still viable, it introduces added complexity and does not have the benefit of compatibility with the rest of the TERAFLUX tool chain.

We therefore have investigated the possibility of translating Scala to C++ to overcome these limitations. The Scala compiler permits the insertion of ‘plugins’ between stages of processing the Abstract Syntax Tree (AST) so that compilation experiments can be performed. The intention was to insert a plugin, in the final stages of AST processing which would traverse the AST and emit C++ code. During the initial stages of this investigation, we discovered that one of the EPFL Scala team, Miguel Garcia, had produced a plugin called ‘imp’ [4] which transforms the AST into something close to a three-address form with the specific intention of permitting compilation to high level imperative languages.

However, it does not appear that this work has progressed to produce any usable tools. It was therefore decided to make use of ‘imp’ but write an additional plugin to follow it and produce C++ code directly. Clearly any Scala program ultimately executes as low level machine instructions and therefore it must be possible to generate any intermediate form which is able to handle the complete Scala language. However, the normal compilation route already makes heavy use of the support provided within a JVM for Java, for example dynamic binding, and many complex issues are not handled at the compiler level. For that reason, it was decided that the simplest route would be to compile to a form mapping the Object Oriented (OO) structure of Scala onto the C++ one. However, there are number of significant differences between the OO structure of the two languages and it is necessary to accept that some features of Scala may not be easily translated.

The purpose of the productivity language route within TERAFLUX is to demonstrate the feasibility of using an advanced language to enable the production of programs which use the Dataflow plus Transactions model with minimal requirements on the programmer to understand the complexities of the underlying implantation. The project has defined a set of benchmarks to evaluate the approach and it was considered that a compilation route which was able to handle these benchmarks, together with some necessary runtime support code written in Scala, would be sufficient. This still requires support for most of the major language features but advanced issues such as reflection will not be addressed.

In addition, Scala has an extensive set of libraries and is also able to make use of all standard Java libraries. It would clearly be a major task to provide this and we intend to limit library support to that required for the benchmarks. Once the system is fully operational, it will be straightforward to add library code when needed.

7.1. Current Status

A version of the translator is operational and the following features are currently supported. They have so far been validated mainly by test programs.

1 All standard basic data types.

2 Lists, Strings and Multi-Dimensional Arrays.

3 Basic I/O.

4 Simple control structures.

5 Classes and Inheritance (including Traits).

6 Package Structure.

7 Some higher order function support.

Some of the above may prove incomplete when more complex programs are compiled and thus the development is ongoing. The major issues which have not yet been tackled include:

- Generics.
- Threads.
- Transactions.
- Exceptions.

In addition garbage collection has not been considered in any depth. It is possible that the facilities provided by GCC may be adequate. In the worst case, we may be able to run reasonably sized programs without GCC. However, Scala programs are likely to make significantly greater use of dynamic object creation that C++.

In order to both validate compilation and assess initial performance, a simplified version of our Lee benchmark written in Scala has been used. This is only a two dimensional version of the algorithm and is not parallel so does not require transactions. It is approximately 120 lines of Scala code. When compiled to C++ and then to native code using GCC the program runs approximately 50x faster than the standard bytecode version running on the HotSpot JVM.

7.2. Plans

We intend to provide initial support for a version which will run on conventional multi-core systems (probably using pthreads). We also intend to produce code which will interface to the run time support provided on the COTSon based TERAFLUX simulator.

This is ongoing work and there are still some complex issues to be addressed. However, it has progressed to a stage where we believe that it will provide a useful tool for evaluation of the productivity language system.

Notice that this integration path is independent from the generic research track conducted on data-flow and transactional memory programming in Scala. It is primarily intended at the validation of this research on the TERAFLUX architecture.

8. Conclusion

We presented the first integrated flow, combining compilation methods and tools, and runtime systems, mapping modern efficiency languages such as StarSs and OpenStream to the TERAFLUX execution model and instruction set. A strong publication output and open source tool distribution was achieved as a result of the activities of WP4. We will aim for direct scalability and efficiency measurements on benchmarks and a few larger applications selected and characterized in WP2 for the final deliverable of WP4.

9. References

[1] Antoniu Pop and Albert Cohen. OpenStream: Expressiveness and data-flow compilation of OpenMP streaming programs. *ACM Transactions on Architecture and Code Optimization (TACO), selected for presentation at the HiPEAC 2013 Conf.*, January 2013.

[2] Nhat Minh Lê, Antoniu Pop, Albert Cohen, and Francesco Zappa Nardelli. Correct and efficient work-stealing for weak memory models. In *Symp. on Principles and Practice of Parallel Programming (PPoPP)*, Shenzhen, China, February 2013.

[3] Feng Li, Antoniu Pop, and Albert Cohen. Automatic extraction of coarse-grained data-flow threads from imperative programs. *IEEE Micro*, 2012. Special issue on Parallelization of Sequential Code.

[4] “Moving Scala ASTs one step closer to C, by turning them into three-address form”, Scala Compiler Corner, http://lampwww.epfl.ch/~magarcia/ScalaCompilerCornerReloaded/

[5] Roberto Giorgi, "TERAFLUX: Exploiting Dataflow Parallelism in Teradevices", *ACM Computing Frontiers*, Cagliari, Italy, May 2012, pp. 303-304.

[6] Roberto Giorgi, Zdravko Popovic, Nikola Puzovic, "DTA-C: A Decoupled multi-Threaded Architecture for CMP Systems", *Proc. IEEE SBAC-PAD*, Gramado, Brasil, Oct. 2007, pp. 263-270

[7] COTSon official repository: http://cotson.sourceforge.net/