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

D3.3 – Report on the Lessons learned about the interaction of Programming Models with the Applications, Compiler and Architecture

Due date of deliverable: 31 December 2011
Actual Submission: 31 December 2011

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

Lead contractor for the deliverable: UNIMAN

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#	Date	Author	Organization	Change History
0.1	**8.11.2011**	**Daniel Goodman**	**UNIMAN**	**Initial document constructed**
0.2	**30.11.2011**	**Mikel Lujan**	**UNIMAN**	**Assembled the received contributions**
0.3	**1.12.2011**	**Mikel Lujan**	**UNIMAN**	**Contributions from Inria**
0.4	**5.12.2011**	**Mikel Lujan**	**UNIMAN**	**Complete document**
0.5	**7.12.2011**	**Mikel Lujan**	**UNIMAN**	**Feedback**
0.6	**14.12.2011**	**Mikel Lujan**	**UNIMAN**	**Updated author list**
0.7	**22.12.2011**	**Mikel Lujan**	**UNIMAN**	**Update based on internal review**

Release Approval

Name	Role	Date
Mikel Lujan	**Originator**	**22.12.2011**
Ian Watson	**WP Leader**	**22.12.2011**
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.

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

Albert Cohen, Léonard Gérard, Antoniu Pop
INRIA

Andreas Diavastos, Samer Arandi, Petros Panayi, Pedro Trancoso, Skevos Evripidou
University of Cyprus

Rahul Gayatri, Rosa M. Badia
BSC

Laurent Morin, Stéphane Bihan, François Bodin
CAPS Enterprise

Glossary

TM	Transactional Memory
Dataflow computation	A dataflow computation is defined by a graph where the nodes are side-effect-free computations (functional computation) and the arcs represent dependencies. A node is activated and executed when its input dependencies have been satisfied, generating seamlessly parallel execution.
Transaction	A set of individual operations that need to be executed atomically, with guarantees of consistency and isolation
Atomicity	Transactions must appear to other transactions as if they occur in a single operation, or do not occur at all.
Consistency	One transaction must take the program from one consistent state to another.
Isolation	Transactions must act on isolation of each other.
TM mechanisms	The implementation of a TM system normally requires a means for detecting conflicts among executing transactions, and a means for versioning data used within a transaction to allow restoring the system state back to its origin should one or more transactions conflict.
Conflict	Two transactions conflict when the two transactions cannot be executed in parallel preserving the atomicity, consistency and isolation properties. There are data dependencies across the transactions (e.g. read-after-write or write-after-write) which would invalidate the parallel execution of those two transactions
Eager conflict detection	The TM system has a choice about when to check whether a number of transactions have a conflict. Eager attempts to detect the conflict during the execution of the transaction.
Lazy conflict detection	Lazy attempts to detect conflicts among the executing transactions when one of these attempts to commit.
Eager versioning	Eager versioning modifies directly memory and requires an undo log to restore the original state.
Lazy versioning	Lazy versioning buffers memory modifications done by a transactions and only once such transaction is allowed to commit, these modifications are propagated to memory visible by other threads.
Nested transaction	A transaction is nested when its execution is contained within the context of another transaction. Flattening treats the nested transactions as a merged single transaction. Open nesting has been proposed as a means to reducing unnecessary conflicts by allowing nested transactions to commit before their parent transaction has been done so.
Strong vs weak isolation	Strong isolation is where nothing can see the state within a transaction while it is executing. Weak isolation is where only other transactions are unable to see intermediate state, but other threads will not be prevented by the programming model from viewing the intermediate state.

Executive Summary

This report contains descriptions of the current state of work within the programming model development, and is split into three parts covering the high productivity model, the synchronous dataflow model and the high performance models.

The specific achievements and discussions are:

High Productivity Model – Scala (Section 3)

- Manchester University Transactions for Scala (MUTS) extended to allow less invasive operation and better functionality.
- Dataflow library (DFLib) implemented providing core Scala dataflow functionality.
- Parallel collections library built on top of DFLib to allow dataflow computation to be added to existing code by changing which library structures are used.
- Lessons for the language design derived from the Teraflux architecture.

Synchronous Dataflow (Section 4)

- Lessons learned about the explicit control of the parallel desynchronization of a concurrent dataflow synchronous program.
- A simple language extension to explicit the desynchronization of a concurrent synchronous dataflow program.
- A systematic compilation method targeting concurrent “futures”, themselves expressible in terms of TStar dataflow primitives or dataflow streaming operations.

High Performance Model – C directives (Section 5)

- StarSs (from BSC) has extended their runtime system to incorporate software transactional memory code.
- TFLUX (from UCY) has proposed new pragma directives to incorporate transactional memory and has performed experiments combining their runtime system with a software transactional memory library.
- HMPP (from CAPS) has extended their pragma directives (reaching version 3) to support the software development of codes to execute on multi-GPU scenarios.
- A new OpenHMPP forum (led by CAPS) has been formed to establish an open standard for programming HPC many-cores. Teraflux provides input into dataflow directives; (visit http://www.openhmpp.org).

- Lessons learned (INRIA) from the refinement of the OpenMP dataflow streaming extensions, with a formal semantics, proofs of determinism, design of (runtime) deadlock detection and debugging algorithms.

- Generalization (INRIA) of the work streaming compilation algorithm to support arbitrary dynamic creation of tasks and dynamic communication rates (algorithm presented in D4.4).

- Lessons learned (INRIA) and new design of a simpler integration flow, leveraging OpenMP dataflow streams combined with Owner Writable Memory (OWM) frames to pass the dependence patterns of the other efficiency languages through the GCC-based backend compiler.

Overall, the programming models have been defined, initial experiments have been completed successfully and we have developed working prototypes able to execute on standard multi-core platforms. In these first two years, we have assumed certain simplifications for the usage of transactional memory within dataflow threads. Next we present a brief motivation for adding transactional memory to the dataflow computational model.

1 Introduction

This document is an update on the work carried out in WP3. It is split into three distinct sections covering the work carried out on the high productivity programming model, on the synchronous concurrency and on high performance models. Within the latter models, we cover progress with C-directive-based dataflow models (StarSs, TFLUX, HMPP, OpenMP).

The Need for Shared Mutable State in Dataflow Models

It is widely accepted that a functional programming approach together with dataflow execution can lead to the efficient exploitation of implicit parallelism. This computational model is usually associated with styles of programming based on pure functions as these map readily on to the execution model. However there are many computations which cannot be easily or efficiently expressed in a pure functional style as they have a structure where the parallel manipulation of shared mutable state is fundamental to the problem being solved. In Teraflux we argue for a new computational model which combines Transactional Memory (TM) and Dataflow to overcome this limitation. Using a simple example, we illustrate why the parallel manipulation of mutable state is necessary and how the addition of TM to Dataflow can provide an elegant and efficient parallel computational model.

Background on Dataflow

In the 1970s and 1980s, there was a strong belief, amongst some, that dataflow approaches would provide the solution to general purpose parallel systems which were easy to program. However, there were three main impediments:

1. Integrated circuit technology developed at a remarkable rate ensuring that serial instruction execution speeds were able to satisfy most performance requirements.

2. Dataflow and associated models have a requirement for high communication rates. The level of integration available in the 70s and 80s did not permit efficient implementations of the necessary functionality.

3. The absence of side effects in dataflow models (e.g. functional programming) permits easy parallelisation. Unfortunately, there are many cases in real programs where the use of state is either necessary for efficiency or is a fundamental part of the problem being solved. In these circumstances, functional approaches are unsuitable.

Nowadays only the third issue still needs to be addressed. Existing functional languages that have tried to address this need for shared state have the problem that either its introduction can rapidly destroy both the mathematical cleanliness of the language with implications to the ability of exploiting parallelism with dataflow (SML, F#, Haskell), or introduces an implicit lock to protect against concurrent accesses (M-structures) which quickly become unfit for purpose due to the lack of composition of locks. M-structures quickly result in deadlocks caused by different access patterns within different threads. To write correct code with M-Structures is difficult and M-Structures are widely noted as a failure including by many of those behind their original development.

An Example with Shared Mutable

The Dataflow plus Transactions model can best be understood by studying an example. Such a study is necessarily simplified but should serve to illustrate the essence of the approach. Assume a graph where a value at each node represents some changing local parameter. This parameter might represent a physical load on a particular local resource. We want to maintain a continuous histogram of load values in order to inform a load balancing system. In practice, it is likely that the structure would be a general graph but here we will assume a binary tree as this greatly simplifies the explanation.

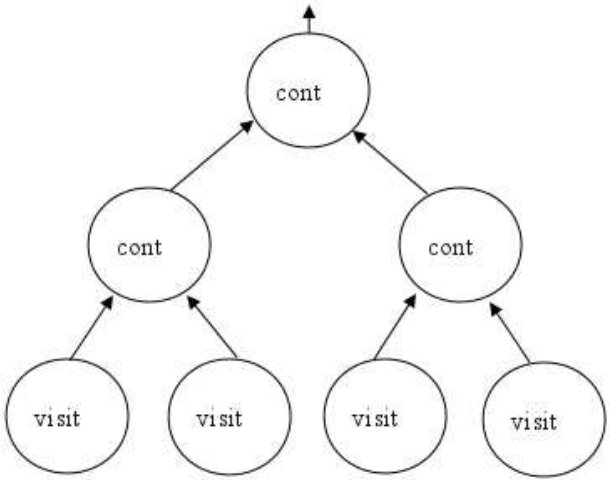
The natural way to express this is a recursive function which visits a node, updates a global histogram and then spawns new threads which visit the children. In addition to updating the histogram we want to know when an exploration of the tree has terminated. The basis of the computation might be a node definition and function of the following form:

```
typedef struct node {
   int old_value, new_value;
   struct node *left, *right;
} node;

int visit (node* t) {
   if (t == NULL) return 1;
   histogram[t->old_value]--;
   histogram[t->new_value]++;
   t->old_value = t->new_value;
   return visit(t->left) && visit(t->right);
}
while (visit (root));
```

We have used an imperative style using C to express the program but the overall computation has two distinct forms. The first is functional in style and can lead to a highly parallel execution. The second is an imperative update of histogram values.

We will assume a dynamic dataflow model where the dataflow nodes are threads executing the `visit' function at the level of a function body. The execution proceeds by constructing, for each function body, a new dataflow node for each new function call within the body together with a continuation node to receive the results of those newly created nodes. The continuation is subject to normal dataflow execution rules. This is essentially packet based graph reduction [8]. This will result in a parallel execution graph of the form shown below



The nodes at the bottom level are in the process of execution, this may either result in the generation of further nodes to explore more of the graph or, if the leaves have been reached, the return of values to a waiting continuation which is synchronised by dataflow firing rules.

However, we have so far ignored the updates to the state of the global array `histogram`, these can clearly occur from any of the parallel executing `visit' nodes. Unless the update action (add and subtract) is performed atomically, the resulting values may be incorrect. In general, the updates will consist of a memory read, an arithmetic operation and a memory write which cannot be separated. In addition, we would like to keep the overall state of the histogram consistent by performing the operation on two entries as a single action. We could achieve this by explicit global locking of the histogram array. This can be done by either using a single coarse grain lock or a set of finer grain locks to improve performance. In either case the addition of this explicit locking would both add significant complication to the code and potentially have an impact on the runtime parallelism.

Our proposal is to introduce transactions into the pure dataflow model. This involves a small modification to the visit function to indicate which parts of the code need to be executed atomically. Assuming the support of a TM system, our execution is going to proceed in one of two ways:

a) In the absence of conflict, i.e. there are no other threads attempting to perform an operation on the same elements of the histogram array, the parallel execution will proceed uninterrupted.

b) If conflict occurs, one of the conflicting threads will proceed uninterrupted but the others will serialize (e.g. by abandoning their execution and retrying)

An important property of transactions is their *isolation*. The detection of conflict and possible retrying is transparent to the application (other than a possible increase in execution time). This ensures that, in a model which is otherwise functional, it is possible to generate implicit parallelism via dataflow execution, but provide the ability to manipulate shared state where necessary.

Deficiencies of Alternative Formulations without Transactions

As already discussed, it is widely accepted that a functional programming approach together with dataflow execution can lead to the efficient exploitation of implicit parallelism. The drawback of the approach is that it cannot easily handle state based computation. In order to appreciate the issues, we will consider expressing the described histogram example using current functional approaches.

Infinite Histories − The classic way to deal with state in a functional program is to regard the updates to a variable as a sequence of state changes forming an infinite history. Any function wishing to change the state of a variable is passed it as a parameter and returns a new version which is the next in the sequence. This is passed in turn as a parameter to the next function which wants to update the state.

The fundamental problem with this approach is that it is essentially serial. It is clearly not possible to have branches in a history which would result if such a variable were passed to multiple functions in parallel. It is therefore not possible to use this technique to produce a parallel version of the above program.

Serialised State Manipulation − The infinite histories approach requires a function to take a parameter and produce an updated version as a result. In a purely functional world this requires the production of a new updated copy. If the variable is a monolithic structure, such as the histogram array in our example, it may be necessary to copy a large structure whilst updating only a small part of it. This can be highly inefficient.

However, as long as the serialisation of updates can be ensured, this is not strictly necessary and an updateable variable can be used. This is essentially the Monad approach where the type system is used to ensure the serial usage. Although this approach is powerful in achieving efficiency whilst maintaining clean properties of the language, it is again fundamentally serial.

Partial Histograms − Rather than construct a histogram directly, we could write our `visit' function to return a partial histogram which represented the contribution of itself and all its descendent nodes. Our program would then construct a tree-like dataflow graph where the partial results were combined by the continuation functions and eventually returned from the root. This would have the following disadvantages:

a) The complexity of the continuation would be increased significantly.
b) The copying of data between parent and child computations could add significant overhead.
c) The overall histogram would be updated only when the computation returned to the root. In a continuous system, such delays may well be unacceptable.

This formulation becomes even more unattractive if we were operating on a more general graph where the issue of when and where to combine partial results would be more complex.

1.1 Relation to other deliverables

This deliverable describes the existing work carried out to extend and implement dataflow and transactional models and it is a continuation of D3.1 and D3.2. The extensions of this model have relations with the Teraflux architecture (D6.1, D6.2 & D7.1) and fault tolerance (D5.1 & D5.2). The implementation has relations with the compilation processes described in D4.1.

1.2 Activities referred by this deliverable

This deliverable covers the work being carried out under WP3 (i.e. T3.1, T3.2 & T3.3).

2 Summary of Dataflow and Transactional Memory

We present an executive summary of the decisions taken to combine dataflow and transactional memory. Full details can be found in D3.1 & D3.2 and some key terms are defined in the Glossary of this deliverable.

Dataflow Threads

The architecture and semantics is simplified when a transaction executes only within a single thread. Once a good understanding of Transactional Memory and Dataflow has been achieved, we intend to look into weakening these constraints.

Versioning and Conflict Detection

Because the project is fundamentally interested in an extensible system, it is felt that the communication required to provide the global observation needed to implement eager conflict detection coupled with the complexity it adds in order to provide correct execution and progress guarantees mean that it is better to opt for lazy conflict detection. This lazy detection can always be strengthened by checks at specified points within the transaction.

Nesting

Although true closed nested transactions are preferred, due to finite hardware resources and after a given depth, it will be reverted to flattened transactions. The first TM prototypes will implement flattening. Because of its non intuitive semantics open nested transactions are not an option.

Syntax

Because of its clarity at a programmer level it is intended that TM syntax in the form of atomic blocks will be provided complete with supporting extensions.

Synchronization

In addition to providing atomic blocks it is intended that all forms of non transactional synchronization construct are excluded as they break the atomicity of transactions.

As an update to these decisions, we note that in WP6 we are investigating how to optimize the detection mechanism by taking advantage of the structure within a node (a set of cores) by having conflict detection options more frequent than lazy.

3 High Productivity Programming Model: Scala

In this section we will describe the work carried out on the development of a high productivity programming model based on extensions to the Scala programming language. Currently this work consists of three libraries which provide transactional memory, dataflow execution and collection objects that take advantage of dataflow execution for improved performance. Having introduced these we will discuss several compilation strategies that we are currently considering and the advantages and disadvantages of each of these. We will then conclude this section with a brief discussion of the architecture issues that affect these extensions to Scala.

3.1 Manchester University Transactions for Scala (MUTS)

In D3.2 we described the implementation of software transactional memory through modifications to the Scalac compiler and a byte-code rewrite of the code at runtime [1]. This has now been redesigned to allow the modifications to the Scala compiler to be replaced with a novel mechanism reliant on closures for marking the transactional areas of the code. The transactions provided by this new extended version are interoperable with the transactions added through the modified Scalac compiler. However, the closure based transactions allow all the required code for transactions to be distributed in a single jar file. This jar file contains both a library to support atomic blocks and a Java agent to perform the rewriting required to instrument the code. This removes the need for programmers using this model to use a special version of the Scala compiler, so making this work more widely applicable.

The syntax provided by the closures is very simple and an example can be seen below.

```
// Program code before a transaction
...
// The transaction
val id = atomic {
   threadId += 1
   threadId
}
// More none transactional code
```

Like with all other blocks in Scala, transactional blocks return a value that can be used as an argument to other expressions, in this case assigning the value of ‘id’.

3.2 Dataflow Library (DFLib)

To compliment MUTS and allow us to construct dataflow code for a number of the applications selected in work package 2 we have constructed a library to support the creation and execution of dataflow threads. Much of the functionality provided by this library will ultimately be provided by the underlying hardware, but this work allows us to experiment with different models in order to inform the design of the underlying hardware and to provide a location to support extended functionality that may be required but not supported directly by the hardware.

The DFLib library is constructed from two principle components a dataflow manager that controls the executions of a given dataflow graph and a set of dataflow thread classes. For any given graph there will be one manager and one or more threads (normally many more).

3.2.1 Thread Creation and Argument Setting

Threads are created via a call to the manager which takes the function to be executed as an argument. This call produces a thread that takes the arguments to the function as arguments. These arguments are strongly typed to ensure that a program is type correct. The arguments can be set in one of two ways: they can be set directly as seen in this example for a thread that executes the function `foo(x:Int, y:String)`,

```
val t = DFManager.createThread(foo)
\\ Other code with t potentially being passed into other functions before
\\ the arguments are set.
t.arg1 = 10
t.arg2 = bar
```

Alternatively because passing threads around can create problems with function type signatures and determining which argument should be set an alternative technique is provided where tokens can be retrieved from the thread. These tokens can then be passed around freely and set. While they are still strongly typed as tokens always take just one argument they can be passed into functions freely without constraining the function signature.

```
val t = DFManager.createThread(foo)
val token = t.token1
\\ Other code with token potentially being passed into other functions
\\ before the arguments are set.
token(10)
```

While the syntax of setting arguments and token values appears as a basic assignment, setter methods are used to ensure that no values escape a thread before the thread completes. This means no thread created or assigned to in another thread can start before the assigning and creating threads have successfully terminated.

3.2.2 Nested Dataflow Graphs

While only one manager may exist per graph, to allow the construction of component code it is necessary to be able to nest dataflow graphs inside one another. This is achieved through the creation of additional managers with each manager handling a single graph. Without this functionality it would not be possible for threads to include functions that execute in parallel and return a value to the thread. Instead the thread would have to be split at the function call. This would not only incur great expense if the thread contains large quantities of thread local data, but would also require that transactions are able to span multiple threads. Allowing transactions to span multiple threads is not desirable as there may be a long delay between the first thread executing and the last thread executing especially if many splits are needed. This not only increases the likelihood of collisions, but also increases the amount of data that has to be maintained both to commit the transaction, but also to be able to execute all the threads it has occupied in the event of a failure.

3.3 Dataflow Collections (DFCollections)

The insertion of parallel execution into a program should be assisted where possible by the program detecting through the stronger semantics of the functional language the possibility for parallelism and automatically inserting it. This can occur in two ways

1. By the compiler creating dataflow threads for the execution of functions within a piece of code, and using tokens to pass the arguments between these functions. This can be achieved by modifications to the parser similar to those undertaken to introduce the first version of transactional memory, by an additional phase in the compiler or by a byte code rewrite.
2. Replacement of existing sequential libraries with parallel code. This can be added for collections such as lists and maps, and ranges to automatically add parallelism. This can either be through the user explicitly requesting it, or by changing the contents of the scala-library jar file. The replacement of Range with a parallel version extends the parallelism to most foreach loops for free.

We have made a first attempt at the library approach through the construction of a set of dataflow collection libraries. These follow a similar structure to that of the Scala Parallel Collections, only instead of being built on top of a thread pool, and associated job queue they are built on a dataflow manager and construct dataflow threads to perform the required work when functions are invoked on them.

3.4 Architecture considerations in Scala

Currently we are aware of two architecture considerations that will ultimately affect the structure of the language. These are the allocation and typing of different categories of memory, and the extent of the support for nesting threads within other threads. We will look at each of these in turn.

3.4.1 Memory Types

There are various properties that we wish to enforce on the memory types of the proposed architecture. These include that transactional data can only be modified within a transaction and thread local data can only be modified by the owning thread. While these rules can be trivially enforced at runtime, by simply detecting incorrect accesses, their type needs to be determined before they are accessed to allow the allocation of the correct memory type. This problem is complicated further when it comes to data structures as different instances of the same data structure may require different data types, for example one instance may be used where members are required to be transactional while another is used where members are required to be local. Our first attempt at addressing this issue will be conducted through the use of the type system to track and enforce the correct memory creation and use at compile time.

3.4.2 Nested Threads

As discussed in the description of the dataflow code, in order to have efficient parallelizable component code without splitting threads at every function call it is necessary to be able to start threads within another thread and retrieve the result of its computation. Normally such behaviour would be prohibited by the dataflow programming model as it removes the guarantees of deadlock freedom. However allowing nested threads to execute a function is different, because they are merely an optimization of the splitting of a thread at the point that the function call is made. More specifically the difference is that there is a guarantee that all the inputs required to execute the nested thread are present when the thread is created, so the thread and its resultant graph will always run to its natural completion without further interaction with the spawning parent graph or any other threads.

At the architecture level this could be provided by either allowing the spawning of a new graph through the same mechanism that will allow different programs to interact with each other or by the prevision of a mechanism to efficiently backup and copy local state between threads to allow for efficient restarts.

4 Synchronous concurrency

This section reports on activities conducted in the context of Task 3.2.

The dataflow principle has well known advantages over most alternative parallel programming and execution models, including the determinism of the concurrent semantics and the scalability and latency-hiding properties of the execution model.

Yet it does not provide enough guarantees for a high-level programming language, and needs to be complemented with additional concurrency properties in four important areas:

- static resource allocation is not possible in general: resource deadlocks can only be eliminated at the expense of dynamic allocation, which may not be convenient on some real-time applications, and improper scheduling of dataflow threads may induce memory usage overhead;
- application deadlocks are not detected at compilation time, resulting in additional runtime bugs not present in sequential applications; our dataflow extension of OpenMP alleviates this problem by forcing a default sequential execution (deadlock-free by construction), but at the expense of expressiveness and programmer comfort; a better solution is expected for a productivity language;
- in the presence of cyclic dataflow graphs or when tasks are nested in complex control flow, the compiler is unable to perform static task fusion and scheduling for locality improvement and synchronization grain adaptation.

All three properties can be provided through the *synchronous principle* of concurrency, a restriction of the dataflow principle. In this context, synchronous execution does not mean blocking (the UNIX-derived interpretation of the term); synchronous execution means that all tasks *can* be scheduled w.r.t. a single global clock, task activations occurring instantaneously at discrete ticks of this clock. It is of course not the necessary way to execute a synchronous program, only a hypothesis that it must satisfy.

The synchronous principle can be enforced at compilation time with a dedicated type system, called a *clock calculus.* This calculus rejects dataflow programs where a clock type cannot be inferred or verified for all expressions in the program. Well-typed dataflow programs are called *dataflow synchronous.* Each stream in a dataflow synchronous program is associated with a unique clock, itself obtained by sampling a common, global clock. By construction of the clock type system and of the associated compilation methods, the program satisfies the three above-mentioned properties.

An introduction to synchronous programming was provided in D3.2. In this deliverable, we elaborate on language extensions to facilitate the expression of data parallelism in a dataflow synchronous language, and to provide the user with an explicit desynchronization construct to compile a dataflow synchronous program to an asynchronous dataflow execution model.

4.1 Controlled desynchronization

Dataflow synchronous expose a great amount parallelism, by their dataflow nature. Yet the traditional means of distributing a synchronous program over multiple processors remains a manual process. There are two main reasons for this:

- finding the right partitioning of fine-grain concurrent tasks into coarse-grain parallel dataflow threads is highly sensitive to dynamic execution and hardware parameters;
- since the main application of synchronous programming is for real-time embedded computing, the need to control the Worst-Case Execution Time and resource (memory) usage is paramount.

For these two reasons, language constructs are needed to let the programmer express the most appropriate way to desynchronize a given dataflow synchronous program.

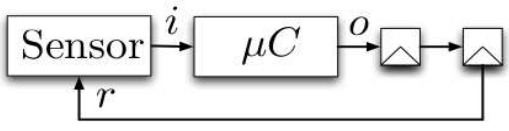
Our idea is to leverage the old concept of future from MultiLISP, now ubiquitous in modern concurrent shared-memory languages. Coupled with an asynchronous function call, it is a natural way to exploit parallelism in a computing pipeline. To the best of our knowledge, it has not been applied in the context of synchronous languages yet.

We demonstrate formally and via several examples that the interaction between the logically timed semantics of synchronous languages and the asynchronous computations expressed through futures allows for a precise control of the parallelism and memory resources. In particular, we propose simple language constructs to control the tradeoff between latency (exploitation of slack in a dataflow program), memory usage for intermediate waiting threads or buffers, and degree of parallelism.

Moreover, these asynchronous calls are proven to preserve the original program's functional semantics: adding or removing them does not change the program results. This allows verifying a program independently of its parallelization. The absence of deadlock (functional or resource-based) is also guaranteed on the synchronous program source.

Here is a simple illustrating example, called the “knock controller”. It consists of a feedback loop running on an embedded control unit for electronic gasoline injection.

```
i = sense(r);
o = compute(i);
r = 0 fby (0 fby o);
```



The syntax and intuitive semantics was introduced in D3.1: in this example, we define three equations on streams `i`, `r` and `o`. The `fby` operator delays its second (stream) argument by one logical instant, inserting its first argument (on the left hand side, here `0`) as the first stream element. To guarantee the causality of the feedback loop (the sensor depends on itself), we need at least one fby operator to break the instantaneous dependence. This example uses 2 `fby` operators: the automatic control algorithm in the µC task is designed to react accurately despite the longer delay. Expressed by 2 `fby` operators, this delay offers valuable slack in the feedback loop, giving us a chance to run a pair of activations of

the µC task in parallel. Our contribution consists in allowing the programmer to express this choice of exploiting this slack for asynchronous execution.

Desynchronization only happens when the program explicitly filters the output of a task with an `async` keyword. Its functional semantics consists in converting a stream of values into a stream of futures of these upcoming values. Downstream tasks depending on an `async` node may start running in parallel, until they need the value of the future. The `!` operator (pronounced "bang", or "get") makes explicit the consumption of the future values, acting as a dataflow synchronization point. In the following code, we desynchronize the main computation task (the async keywords in front of the constant `0` is only there for typing issues; it will be made implicit in future versions of the language via automatic boxing of values into futures).

```
i = sense(!r);
o = async compute(i);
r = (async 0) fby (async 0) fby o;
```

Now, the stream `o` becomes a stream of futures. The `fby` operators now operate on streams of futures; this does not impact the functional semantics of the program, and neither it changes the amount of memory needed to execute it (aside from the future descriptors themselves, whose individual size is constant). The good news is that it is now possible to run two activations of the `compute` task in parallel before being blocked by the synchronization on `r` in front of the `sense` task. In short, we are able to explicit the degree of parallelization of the computational task (here, 2), preserving the bounded memory usage of the program.

We fully formalized the structured operational semantics of our language extension as rewriting rules and a separate transition system to capture the asynchronous instantiation of future values. We proved a preservation theorem stating that any value computed by the original synchronous program is either the same in the asynchronous program, or replaced by a future with this value associated. As a corollary, the preservation theorem proves that the usual sanity checks of dataflow synchronous programming (clocking and causality) do not need any modification to cope with the explicitly desynchronized computations.

Our current implementation uses Java futures as a runtime execution environment. Here is the output of our Heptagon compiler for dataflow synchronous programs:

```
package Knock_control;
import java.util.concurrent.Future;
import java.util.concurrent.ExecutionException;
import jeptagon.AsyncNode;
import jeptagon.AsyncFun;

public class Main {
    protected Async_factory_compute n;
    protected Sense n_3;
    protected Future<Integer> r;
    protected Future<Integer> v;
```

```
    public Main () {
        this.n = new Async_factory_compute(1, 1);
        this.n_3 = new Sense();
        {
            this.r = new jeptagon.Pervasives.StaticFuture(0);
            this.v = new jeptagon.Pervasives.StaticFuture(0);
        }
    }
public void step () throws InterruptedException, ExecutionException
{
        int v_2 = 0;
        int i = 0;
        Future<Integer> o = null;
        v_2 = this.r.get();
        i = n_3.step(v_2);
        o = n.step(i);
        this.r = this.v;
        this.v = o;
        return ;
    }
    public void reset () {
        this.r = new jeptagon.Pervasives.StaticFuture(0);
        this.v = new jeptagon.Pervasives.StaticFuture(0);
    }
}
```

It is easy to recognize the sequential execution pattern resulting from the clock-directed compilation of the feedback control loop. The ability to generate efficient, bounded memory sequential code being one key strength of synchronous programming. The desynchronization constructs barely impact the structure of the code. The type of the `o` variable has been lifted from `int` to `Future<Integer>`, and the activations of the `compute` function are now run as asynchronous dataflow threads.

More general computations involving automata and subsampled clocks lead to more complex data dependence patterns. These are easily captured with futures, but mapping such dynamic computations to TStar dataflow primitives is more challenging. We are currently investigating different methods, from pure data-driven TStar implementations to streaming synchronization schemes that may be seen as specialized, highly scalable I-structures.

This Java-based implementation was the shortest path to a complete working system. We are now porting this implementation to generate streaming OpenMP pragmas, on which the semantics of futures is being mapped. This resulted in an extension of our streaming OpenMP proposal to more efficiently model completely dynamic dependences; this is work in progress and will be reported in the third year deliverables.

4.2 Expression of data parallelism

It is easy to complement the previous desynchronization constructs with reset operations, commonly found in synchronous languages. Resetting a task amounts to breaking any existing dependences on values of previous activations of this task, restarting from initial values in all `fby` operators instead. As a side-effect, resetting a task exposes data parallelism: activations of the task prior to the reset may run in parallel with activations of the task after the reset. While stateless tasks can be trivially data-parallelized, this idea allows some stateful tasks to be (partially) data parallelized as well, without extending the language semantical primitives.

4.3 Status of the work and perspectives

A paper will be submitted in January covering the formal semantics and implementation of the controlled desynchronization constructs in a synchronous program.

Future work will concentrate on revisiting the implementation the Heptagon source-to-source dataflow synchronous compiler, targeting the efficiency layer (streaming OpenMP pragmas). We will continue to evaluate our hypotheses and the expressiveness of the synchronous principle on TERAFLUX applications.

5 High Performance Developers: C pragmas

5.1 Construction of Software Transactional Memory in StarSs

StarSs is a task-based programming model that enables the exploitation of applications' inherent parallelism at task level. To annotate the tasks in a StarSs application, compiler directives similar to the OpenMP ones are used. While this is already found in OpenMP 3.0, a uniqueness of StarSs tasks are the input, output or inout clauses that applied to tasks' parameters enable the runtime to track tasks' data dependencies. A task dependence graph is dynamically built and scheduled for execution in the different devices. Also the clause "target device" to specify that a given task code is tailored to a specific device type (e.g., multiple GPUs) has been defined [7, 8]

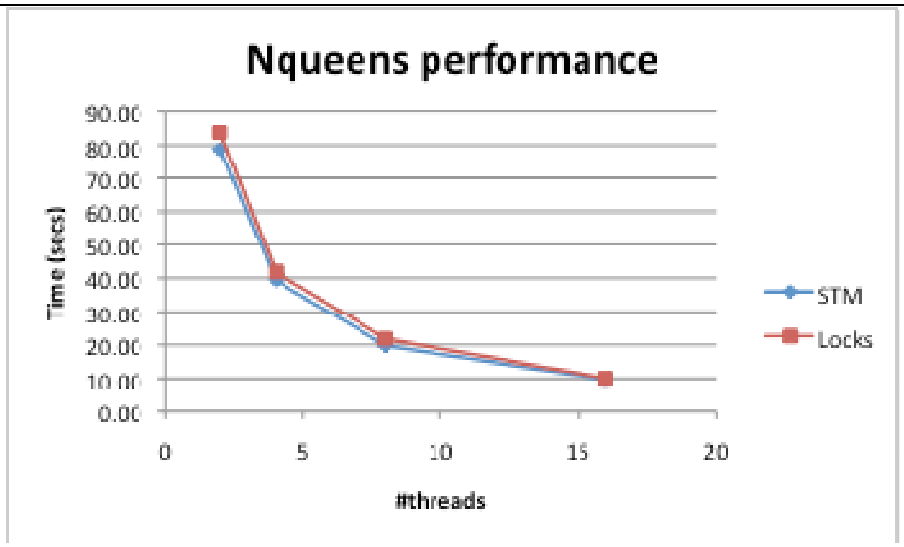
5.1.1 Use of TM to guarantee mutual exclusion in StarSs

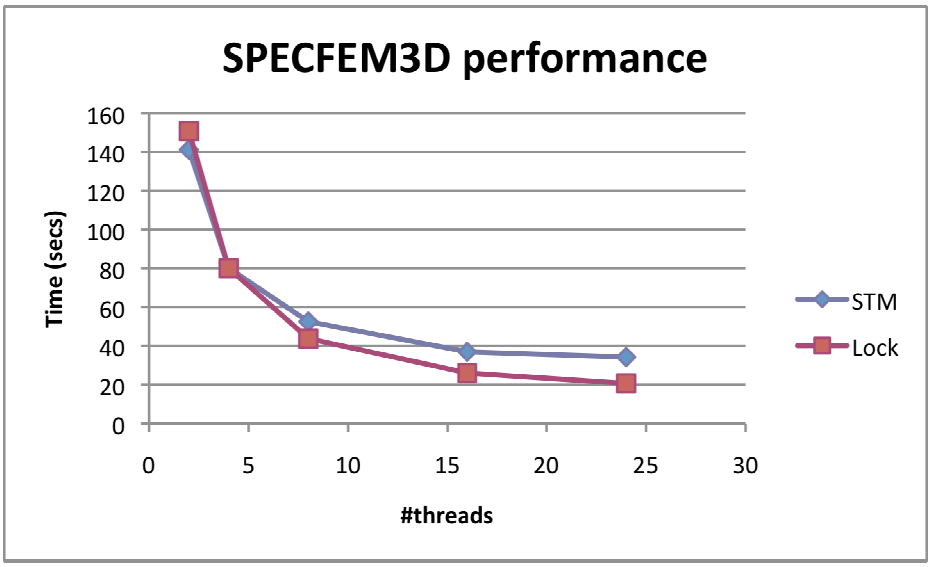
One of the activities performed in the framework of this WP is the integration of transactional memory with the StarSs programming model. Deliverable 3.2 reported the initial efforts in this direction, performed by integrating TinySTM, a STM library, with SMPSs, an implementation of StarSs with focus on SMP architectures.

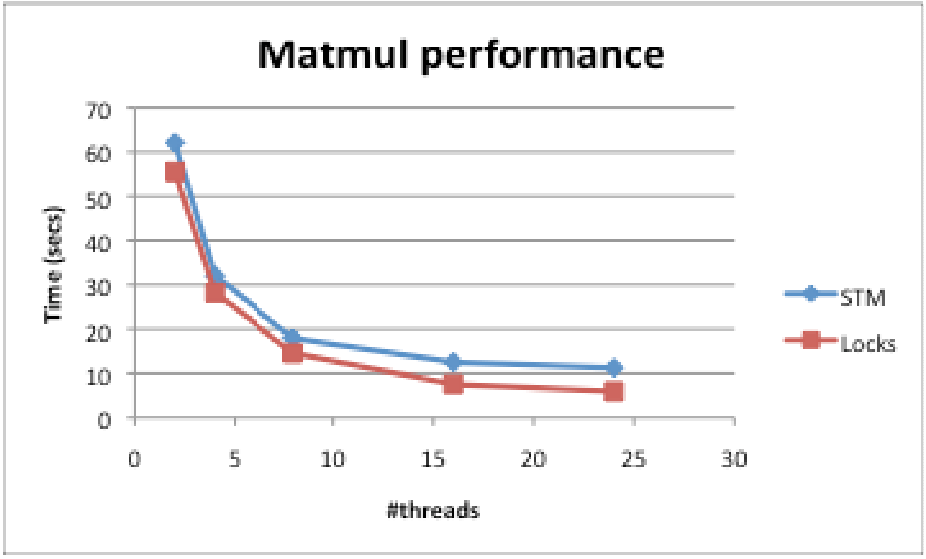
The initial experiments described in D3.2 were performed calling the STM calls from the SMPSs application code. In this second period, we have integrated in the SMPSs runtime code the TinySTM library calls that initialize the STM library, those that start a transaction, load a critical variable, store the variable back in the memory and the ones that commit the results. In order to indicate the regions that are to be executed atomically, we reused the compiler directives from SMPSs that guarantee mutual exclusion access (`#pragma css mutex lock` and `#pragma css mutex unlock`) by changing their implementation.

The behavior implemented in the SMPSs runtime is the following: whenever a lock on a variable is called, a transaction is started and the variable to be locked is loaded using the transactional calls. When the lock is released the value is stored back again using transactional commands and a commit operation is performed. If a conflict occurs then the transaction is restarted.

This implementation has been tested with some SMPSs benchmarks and applications, like the nqueens problem, the matrix multiply code and the specfem3D code. Results to such experiments are shown below:


Nqueens performance
90.00
80.00
70.00
60.00
50.00
40.00
30.00
20.00
10.00
0.00
Time (secs)
0
5
10
15
20
#threads
STM
Locks


SPECFEM3D performance
160
140
120
100
80
60
40
20
0
Time (secs)
0
5
10
15
20
25
30
#threads
STM
Lock



For the nqueens case, the original SMPSs mechanism (labeled as "Locks") is slightly worst that when we use STM (labeled as STM). For the SPECFEM3D the performance is also a bit worst when using STM than with the original lock mechanism, and the difference increases with the number of threads, but still comparable bit better with the new STM mechanism, and what is more, improving when increasing the number of threads which is promising. In the Matmul the results obtained with the STM mechanism are slightly worst, but we are still in the orders.

From these experiments we conclude that the use of STM can be better than the lock mechanism when a HW transactional memory mechanism is available.

5.1.2 Use of TM to guarantee perform task speculation in StarSs

One of the non-solved issues with StarSs is the synchronization with the main program code: while the task graph is able to drive the execution of the tasks in an asynchronous fashion whenever a result from a task needs to be read from the main code, an explicit synchronization is required. This synchronizations in the main code are sources of bottlenecks, load imbalance and also stop the generation of new work (new tasks) that reduce the chances of exploiting further parallelism in the application.

Work related to the use of STM combined with StarSs to reduce these issues are ongoing [2].

5.2 TFlux

Within the context of task T3.1 UCY has revised the TFlux pragma directives to support more applications and also new features in the runtime support. These revisions are minor and they have already been reported in D3.2. In the context of task T3.3 we have explored together with UNIMAN the augmentation of the data-flow model with the support for transactions. In this particular case we have done it as an extension to the TFlux system, which we call DDM+TM. We have used a software TM library (TinySTM) to extend TFlux with transactional support. We have also added new pragma directives for the support of transactions at the programming model level. These new directives are

presented in Table 1. The preliminary results of this collaboration work have been recently presented at the First Workshop on Data-Flow Execution Models for Extreme Scale Computing (DFM 2011) [3]. The following text has been extracted from that publication.

Table 1: TFlux DDM pragma directives for TM support.

`#pragma ddm atomic thread` *`ID`* `tvar(`*`NAME : READ/WRITE/READ_WRITE`*`)`	DDM+TM thread boundaries with identifier *ID* and the atomic variables to monitor for either *READ* or *WRITE*
`#pragma ddm atomic endthread`	
`#pragma ddm atomic for thread` *`ID`* `tvar(`*`NAME : READ/WRITE/READ_WRITE`*`)`	DDM+TM loop thread boundaries with identifier *ID* and the atomic variables to monitor for either *READ* or *WRITE*
`#pragma ddm atomic endfor`	
`#pragma ddm atomic transaction tvar(`*`NAME : READ/WRITE/READ_WRITE`*`)`	DDM+TM boundaries of a transaction that is smaller than a thread and the atomic variables to monitor for either *READ* or *WRITE*
`#pragma ddm atomic endtransaction`	
`#pragma ddm atomic tvar(`*`NAME : READ-/WRITE/READ_WRITE`*`)`	Declare an atomic variable to monitor either for *READ* or *WRITE*
`#pragma ddm atomic abort`	Manually abort a transaction

When adding support for transactions to TFlux an important decision concerned the granularity of transactions. The simplest approach would be to declare a whole thread as a transaction. With this option we would enhance the system by providing the programmer with two types of threads: pure Data-Flow threads or transactional threads. However, a thread may contain code that needs to be transactional combined with non-transactional code. Furthermore, it may be appropriate to specify several atomic regions within a thread. This could lead to potentially wasteful aborts when either a transaction is only a small portion of the thread or multiple atomic regions need to be aborted together. Therefore, we opted for providing new pragma directives to define the beginning and end of transactional sections within threads. These new TFlux directives are presented in Table 1.

Another design issue is how we identify variables, which are transactional. These variables will require that their read and write operations are observed to form the read-set and write-set during a speculative execution of the transaction. These sets are used to detect conflicts. For all these transactional variables we also need to version the results to allow a clean restart of the transaction if necessary. One option is to monitor every memory access that is performed within a transaction. However, this is not necessary for unshared variables, for example those that are thread local.

Therefore, in common with other TM approaches, we explicitly declare which variables are transactional. The directive

`#pragma ddm atomic tvar(`*`NAME : READ/WRITE`*`)`

offers such functionality. Note that each transactional variable is associated within a thread with a READ, WRITE or READ_WRITE qualifier. This qualifier provides information on the use of the variable within the thread, which can be used by the TM implementation to optimize the execution.

For DDM+TM, we decided to have a complete separation between transactional and non-transactional variables. Transactional variables must always be accessed within a transaction. Non-transactional variables are normally private to a thread during execution and thus cannot generate conflicts. Other non-transactional threads will only be allowed to access a non-transactional variable if the scheduling can guarantee independence. With this decision we avoid the problems of weak isolation. Note that we do not modify DDM by imposing this decision.

For transactional variables, we also provide the programmer with pragmas to define them within the declaration of a transactional thread. This is required to support the monitoring of variables that may have more than one alias (e.g. parameter variables inside the code of a function). The monitoring of these variables is specified as a parameter in the thread declaration (see Table 1).

As TFlux has two pragmas for declaring threads, the table contains

`#pragma ddm atomic thread` *`ID`* and

`#pragma ddm atomic for thread` *`ID`*

declaring a transactional thread and a transactional loop thread, respectively. The tvar(NAME : READ/WRITE) extension defines the thread variables that are transactional. The last proposed directive

`#pragma ddm atomic transaction`

allows the declaration of a transaction as a portion of a thread. For certain applications this offers better performance. These directives are a subset of the possible ones that have been defined for TM. However, they are enough to implement the applications we have implemented so far and we consider them to be the core directives. Extra transactional functionality can be added by declaring

`#pragma ddm atomic abort`

in the case the programmer wants to manually abort a transaction.

The newly proposed directives have been added to the TFlux chain as we have modified our TFlux preprocessor tool to automatically generate the calls to TinySTM based on the pragma directives.

5.3 HMPP: a Directive-based Programming Model

HMPP implements a Remote Procedure Call (RPC) of functions called codelets on GPU accelerators. A codelet takes several scalars and arrays as parameters, performs a computation on these data and returns the result in an argument passed as a reference of the codelet. The execution of a codelet is

considered atomic: it does not have an identified intermediate state or data. The execution has no side effects.

HMPP directives are safe meta-information added in the application source code that do not change the original code. They address the remote execution (RPC) of functions or regions of code as well as the transfers of data to and from the accelerator memory.

5.3.1 HMPP Accelerated Regions and Functions

The simplest form of programming with HMPP consists in either using one single directive to declare a GPU version of a region or two directives for functions: one to declare the codelet function and another one to annotate its call site. This way all input data are uploaded in the GPU before the execution of the region/function and the result is downloaded when the codelet has completed.

Device allocation and data transfers can be dissociated from codelets' calls using the appropriate directives. All directives are identified with a unique label indicating the codelet they are associated with. Directives of same label needs to be used in the same compilation unit.

For instance, in the code example 1 below, a `codelet` directive with a 'cuda_kernel' label is inserted line 2 to declare a CUDA version of the 'kernel' function to be generated by HMPP. Call to this codelet is indicated with a `callsite` directive of same label inserted just before the call to kernel line 31.

```
1   #pragma hmpp cuda_kernel codelet, target=CUDA, args[vout].io=inout
2   static void kernel(unsigned int N, unsigned int M,
3                      float vout[N][M], float vin[N][M]){
4     int i, j;
5     for(i = 2; i < (N-2); i++) {
6       for(j = 2; j < (M-2); j++) {
7         float temp;
8         temp = vin[i][j]
9              + 0.3f *(vin[i-1][j-1] + vin[i+1][j+1])
10             - 0.506f *(vin[i-2][j-2] + vin[i+2][j+2]);
11        vout[i][j] = temp * (vout[i][j]);
12      }
13    }
14  }
15  int main(int argc, char **argv){
16    unsigned int n = 100;
17    unsigned int m = 20;
18    int i, j;
19    float resultat = 0.0f;
20    float out[n][m];
21    float in[n][m];
22     ....
23    // init
24    for(i = 0 ; i < n ; i++){
25      for(j = 0 ; j < m ; j++){
26        in[i][j] = (COEFF) * (-1.0f);
27        out[i][j] = (COEFF) + (j * 0.01f) ;
28      }
29    }
30    #pragma hmpp cuda_kernel callsite
31    kernel(n,m,out,in);
32     ....
33    printf("result : %f\n",resultat);
34  }
```

Code example 1: basic HMPP programming.

5.3.2 HMPP Multi-GPU Partitioning

As the number of cores keeps growing, it is essential to help developers easily distribute data and computations over multiple CPUs and GPUs. The HMPP programming model version 3.0 supports multi-GPU programming by enabling developers to either perform array distribution or spread out a collection of data on multiple devices.

In the C example below, two CUDA devices are attached to a group of codelets. The loop in the main function is indicated parallel with a clause expression defining how data and computations in the region are to be distributed between the two devices. All the directive operations in the loop inherit from the distribution expression.

```
#pragma hmpp <my_grp> group, target=CUDA, nb_device=2
#pragma hmpp <my_grp> my_cdlt codelet, args[*].io=inout, args[*].mirror
void f(float a[10000])
{...}

#define N 20
float x[N][10000];

int main()
{
  int i;

#pragma hmpp <my_grp>  parallel, device="i%2"
  for (i = 0; i < N; ++i)
  {
    #pragma hmpp <my_grp> new, data["x[i]"]
    #pragma hmpp <my_grp> allocate, data["x[i]"], data["x[i]"].size={10000}

    #pragma hmpp <my_grp> my_cdlt callsite
    f(x[i]);
    #pragma hmpp <my_grp> release, data["x[i]"]
    #pragma hmpp <my_grp> delete, data["x[i]"]
  }
```

Code example 2: distributing codelet execution and data over multiple GPUs

5.4 Dataflow extensions to OpenMP and Interaction with WP4

Following the first design work on OpenMP dataflow extensions, INRIA has been conducting a more systematic effort on the compilation algorithms, formal semantics, implementation on different targets, and expressiveness experimentation. This deliverable highlights the main results and lessons learnt, while more details can be found in the PhD thesis of Antoniu Pop [4, 5], from which we plan to extract multiple papers in 2012.

5.4.1 Lessons learnt from the dataflow extensions to OpenMP

Our efforts concentrated on the integration of the streaming dataflow paradigm in a high-level, general-purpose parallel programming language, OpenMP. As introduced in the first year deliverables, our primary goal was to maximize the impact on existing programming practices, with no sacrifice in terms of semantical and syntactical expressiveness, preserving the OpenMP existing parallel and synchronization constructs. Our experience in the second year has been that this choice is fully compatible with code generation towards a low-level dataflow interface, including a hardware-based implementation with the TStar instruction set. Functional simulation of TStar generated from various streaming dataflow OpenMP programs is now possible.

In parallel, our effort to support stream operations (sliding windows, communication bursts, random-access in streams) has brought entire satisfaction. First, the performance and locality optimization benefits of the workstreaming algorithm we designed and implemented (reported in D4.4 and D4.5). Second, the integration potential of dataflow streams, generalizing I-structures and more dynamic data dependence patterns exposed in StarSs and TFlux. Third, the formal semantics and practical implementation described in Antoniu Pop's thesis [5] clearly demonstrate that arbitrary dynamic dependences, task creation, and communication rates are compatible with strong guarantees in terms of functional determinism, (runtime) deadlock detection and debugging facilities, and efficient compilation to scalable synchronization mechanisms (the workstreaming algorithm). Our semantical model is called Control-Driven Data-Flow. A trace-based operational semantics has been defined, and

its properties and associated compilation algorithms have been proven. Much of this remains in an early stage, implementation wise, and we will now focus our efforts on a complete implementation, widely disseminated with many examples and a comprehensive report.

Finally, driven by the University of Manchester, the partners of WP3 have been pushing a joint effort on refining the memory model of the TERAFLUX programming languages and TStar ISA. Two approaches are being investigated.

1. Focusing on object-level memory management, with a global object-level addressing space. This approach is favored by Microsoft and would be appropriate for Scala and other managed languages.
2. Focusing on word-level pointers, with a global address space in virtual memory.

In both approaches, TERAFLUX departs from the reasonably popular Partitionned Global Address Space (PGAS) approach of languages like CoArray Fortran, UPC (both word-level) or X10 (object-level). We are aiming for a non-partitioned global address space, leveraging dataflow synchronizations and explicit, software controlled actions to enforce coherence and consistency at the dataflow thread level. Based on a first experiment at BSC (tfsim, used as a low-level memory management framework for COTSon, cf. deliverable D7.1), INRIA implemented a runtime called dfrt for a word-level global address space. A first version was built on the VMS runtime platform designed at INRIA [6], and it was later optimized and scaled to clusters of multiprocessors as a native implementation in C++ with MPI. This runtime is currently being evaluated on increasingly complex applications. Based on this experiment, the partners of WP3 will define and implement an integrated plan to support the global address space memory model within the TERAFLUX efficiency languages and within the COTSon-based implementation of the TStar ISA; this is planned for the third year.

5.4.2 Lessons learnt in the integration plan and compilation flow

In the second year of the project, we realized that the implementation of the unified compilation interface proposed in D4.3 would not be ready on time to support a sufficient number of features of the different efficiency programming models. Indeed, supporting the expansion of high-level concurrency and control flow constructs on low-level SSA representation in GCC is a not for the faint of heart. We thus decided to explore alternative routes, still enabling every efficiency programming notation to be compiled with GCC to TStar ISA. Specifically, we studied the three following options.

- Support every programming model feature in the streaming OpenMP notation implemented in GCC.

The impact would be very high for the INRIA partner, as this option leads to the replication of the efforts at BSC, CAPS and UCY inside GCC. On the positive side, it would be possible to incrementally implement the unified representation of D4.3, such that the optimizations designed and implemented in GCC would become applicable to all efficiency programming models.

- Expose low-level TStar instructions as compiler builtins, allowing StarSs, HMPP and TFlux source-to-source compilers to target them directly. This is the proposed flow for Scala, and the current way TFlux operates. But it makes minimal use of the optimization framework being implemented in GCC. Also, it leads to duplication of effort at the different partners, who would need to design and implement complex compilation and runtime methods to map the general dynamic dependence patterns to the more constrained data-driven semantics of TStar. This would be particularly challenging for StarSs.
- By closely investigating the semantical differences of the different notations, we discovered a third, more elaborate option.

The idea is to expand the efficiency programming models into streaming OpenMP tasks, converting the high-level semantics for building complex dependence patterns into monotonic, stream operations. A clause in these programming models (e.g., `input`) is replaced with a stream of pointers in the streaming OpenMP extension. Each pointer in such a stream contains the base address of the region declared in the StarSs, TFlux or HMPP clause. Synchronization is a result of the streaming dataflow semantics, while communications are determined by explicit memory frame operations. Building on the memory model introduced in D7.1, we will rely on both Owner Writable Memory (OWM) and Transactional Memory (TM) frames, depending on the semantics of the clause being translated. The OWM frames being the most obvious candidates to implement I-structures and other dynamic dependence resolution mechanisms that naturally map to streaming semantics. Since neither OWM nor TM frames implement any dataflow synchronization by themselves, the GCC TStar backend will be responsible for maintaining the causal link between the coherence and communication actions associated with the OWM or TM frames, and the corresponding dataflow synchronizations.

Independently, the StarSs, TFlux and HMPP dynamic dependence resolvers will be implemented natively using streaming OpenMP constructs. The source-to-source compilers will also have to be modified to avoid any ad-hoc synchronization mechanism, relying only on their associated dependence resolver.

Option 3 is clearly more interesting as it leverages the optimizations being performed in WP4 and does not require CAPS, BSC and UCY to duplicate work being done at INRIA (unlike option 2), and it does not require INRIA to replicate all the work being done in WP3 by every other partner (unlike option 1).

A proof of concept implementation of Option 3 for StarSs, TFlux, and HMPP tasking and dependence concepts could be presented at the review meeting.

6 Summary

This document has described the research carried out in the Workpackage 3 of the Teraflux project during the second year. It is split into three distinct sections covering the work carried out on the high productivity programming model, on the synchronous concurrency and on high performance models. Within the latter models, we cover progress with C-directive-based dataflow models (StarSs, TFLUX, HMPP, OpenMP). The executive summary has presented the achievements obtained during this year. In addition, it does contain the rationale for needing a means for handling shared mutable state in dataflow models. This deliverable has covered the work being carried out in T3.1, T3.2 & T3.3).

Overall, the programming models have been defined, initial experiments have been completed successfully and we have developed working prototypes able to execute on standard multi-core platforms.

Next year, the projects partners will continue refining the dataflow computational models (i.e. T3.4 Consolidated Dataflow Models) and learn from the interactions with WP2 (applications), WP4 (compilation tools) and WP5 (execution on the Teraflux architecture). In addition we will start investigating how the combined model of TM and dataflow gets affected once we eliminate those restrictions used in years 1 and 2.

References

[1] Daniel Goodman, Behram Khan, Salman Khan, Chris Kirkham, Mikel Lujan and Ian Watson. MUTS: Native Scala Constructs for Software Transactional Memory. In Scala Days 2011.

[2] A. Diavastos, P. Trancoso, M. Lujan and I. Watson, “Integrating Transactions into the Data-Driven Multi-threading Model using the TFlux Platform” in Proc. of the Data-Flow Execution Models for Extreme Scale Computing (DFM) Workshop, Galveston, Texas, U.S.A., October 2011.

[3] Rahul Gayatri, "Integrating Transactional Memory in StarSs", Computer Architecture PhD program, Technical University of Catalonia, 2011.

[4] 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.

[5] Antoniu Pop. Leveraging Streaming for Deterministic Parallelization - an Integrated Language, Compiler and Runtime Approach. PhD Thesis MINES ParisTech, September 2011.

[6] Sean Halle and Albert Cohen. A mutable hardware abstraction to replace threads. In Languages and Compilers for Parallel Computing (LCPC’11), LNCS, Fort Collins, Colorado, September 2011.

[7] Alejandro Duran, Eduard Ayguadé, Rosa M. Badia, Jesús Labarta, Luis Martinell, Xavier Martorell and Judit Planas, "OmpSs: A proposal for programming heterogeneous multi-core architectures", in Parallel Processing Letter, Volume 21, Issue 2, pp. 173 - 193, June 2011.

[8] Javier Bueno, Judit Planas, Alejandro Duran, Rosa M. Badia, Xavier Martorell, Eduard Ayguadé, Jesús Labarta, Productive Programming of GPU Clusters with OmpSs, 26th IEEE International Parallel & Distributed Processing Symposium, 2012 (to be published).

[9] I. Watson, V. Woods, P. Watson, R. Banach, M. Greenberg, and J. Sargeant, “Flagship: A parallel architecture for declarative programming,” in ISCA, 1988, pp. 124–130.