Transactional Memory on a Dataflow Architecture for Accelerating Haskell

ROBERTO GIORGI
University of Siena
Department of Information Engineering and Mathematics
Via Roma 56, Siena
ITALY
giorgi@dii.unisi.it

Abstract: Dataflow Architectures have been explored extensively in the past and are now re-evaluated from a different perspective as they can provide a viable solution to efficiently exploit multi/many core chips. Indeed, the dataflow paradigm provides an elegant solution to distribute the computations on the available cores by starting computations based on the availability of their input data.
In this paper, we refer to the DTA (Decoupled Threaded Architecture) – which relies on a dataflow execution model – to show how Haskell could benefit from an architecture that matches the functional nature of that language. A compilation toolchain based on the so called External Core – an intermediate representation used by Haskell – has been implemented for most common data types and operations and in particular to support concurrent paradigms (e.g. MVars, ForkIO) and Transactional Memory (TM).
We performed initial experiments to understand the efficiency of our code both against hand-coded DTA programs and against GHC generated code for the x86 architecture. Moreover we analyzed the performance of a simple shared-counter benchmark that is using TM in Haskell in both DTA and x86. The results of these experiments clearly show a great potential for accelerating Haskell: for example the number of dynamically executed instructions can be more than one order of magnitude lower in case of Haskell+DTA compared to x86. Also the number of memory accesses is drastically reduced in DTA.

Key–Words: Multithreaded Architecture, Data-flow Architecture, Haskell, Transactional Memory

1 Introduction

It is widely accepted that combining dataflow execution with functional programming can provide enough implicit parallelism to achieve substantial gain compared to non-dataflow machines. Functional programming is a paradigm that treats computation as the evaluation of mathematical functions and avoids state and mutable data. It emphasizes the application of functions, in contrast with the imperative programming style that emphasizes changes in state [37]. However, real-world computations may require a mutable shared state as in the case of airline seat reservation or in bank account update by multiple clients.

Software Transactional Memory (STM) is a concurrency control mechanism analogous to database transactions for controlling access to shared memory [28], [15]. Transactions replace locking with atomic execution units, so that the programmer can focus on determining where atomicity is needed, rather than how to realize it. With this abstraction, the programmer identifies the operations within a critical section, while the STM implementation determines how to run that section safely. Moreover, the optimistic control over transactions allow the system to avoid unnecessary serialization of mutually exclusive operations. In this work we deploy a decoupled multithreaded architecture (DTA) [7] for an efficient symbiosis with Haskell and its TM API.

One contribution of this research is the creation of a tool to translate simple Haskell programs (using External Core intermediate language [25,26]) in DTA assembly language. A second contribution regards the development of a dataflow based implementation of STM (evaluated on the DTA). Examples and implementations are discussed in detail in this paper, while an initial discussion can be found in [4].

The rest of the paper is organized as follows: we briefly describe the DTA architecture (section 1.1) and the Haskell STM (section 1.2); then, we illustrate the tool we have developed for translating Haskell programs in DTA (section 2) and our first implementation of STM in DTA (section 3). We clarify our methodology in section 4.1 and we discuss the results of our experiments (section 5) and we conclude the paper.

1.1 Overview of the Decoupled Threaded Architecture

The DTA architecture [7] is a hybrid dataflow architecture that is based on the Scheduled DataFlow (SDF) execution paradigm [19], [31] and more recently has led to the TERAFLUX architecture [3], [30], [5], [6], [10], [16], which binds to coarse grained dataflow and multithreading. A DTA program is compiled into a series of non-blocking threads where all memory accesses are decoupled from the execution by using DataFlow Frames (DF-Frames), i.e., portions of memory managed at architectural level with the appropriate semantics (e.g., shared or private) by Thread Scheduling Units (TSUs, Figure 2). The instructions that manage DF-frames are reported in Table 1

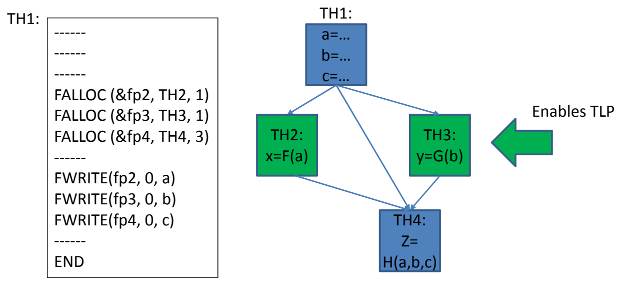
Table 1: DF-Frame Related Instruction.

Instruction	Meaning
FALLOC R1,R2,R3	Allocate a DF-frame (pointer in R3) for Thread (pointer in R1) with a synchronization count R2
FREAD R1,R2	Read from current DF-Frame at offset R1 and return the value in R2
FWRITE R1,R2,R3	Write in the DF-Frame (pointer in R1) at offset R2; the value is in R3
FFORKSP R1,R2	Switch control to Synchronization Pipe. based on condition in R1; the code is pointed by R2
FFORKEP R1,R2	Switch control to Execution Pipe. based on condition in R1; the code is pointed by R2

Starting from a Control Data Flow Graph of a program (or a portion of it) or from some intermediate representation as we are doing in this paper, each thread is isolated in such a way that it consumes a number of inputs and produces a number of outputs (Figure 1). In this example a program is subdivided into four threads. Thread 1 computes the variables a, b and c and sends them to other threads. Threads 2 and 3 execute the calculation of F and G, they can act in parallel because their bodies are independent. At last, Thread 4 waits for the results of the other threads, then it start its execution.

An important observation, is that the Threads are DataFlow Threads (DF-Threads [6]), i.e., they start executing when three conditions hold: i) a parent thread has to allocate a frame (FALLOC instruction); ii) all their inputs had been written (by some FWRITE instructions); iii) the data of the frame had been prefetched [8]; iv) an external unit (the TSU) has to decide that a suitable core for the execution is available.

In order to ensure that any thread will not start executing before all of its data is ready, a synchronization count (SC) is associated to each of them. This synchronization count is the number of inputs needed



```
------
------
------
FALLOC (&fp2, TH2, 1)
FALLOC (&fp3, TH3, 1)
FALLOC (&fp4, TH4, 3)
------
FWRITE(fp2, 0, a)
FWRITE(fp3, 0, b)
FWRITE(fp4, 0, c)
------
END
```

Figure 1: SDF/DTA execution paradigm. Dataflow execution enables Thread Level Parallelism (TLP).

by the thread. In this example, Thread 2 needs a, so its synchronization count is 1, while the SC for Thread 4 is 3. Whenever the data needed by a thread are stored (FWRITE instruction), the synchronization count is decremented, once it reaches zero, it means that the thread is ready to execute. DTA execution model uses DF-frames to pass data from one thread to other ones. DMA mechanisms and prefetching assist the processor to make sure that the data is in a local memory (e.g., the cache or a scratchpad [9]). DF-Frames are dynamically allocated whenever there is a need to execute a corresponding thread instance: this is similar to memory allocation, but the thread that issues a FALLOC does not need to wait much time: the frame can be preallocated and inserted in a local pool. The DF-frame has a fixed size and it can be written by other threads and read only by the thread that it belongs to. The FREAD instruction is used at the beginning of a thread to load the data from such local DF-frame: such phase is denoted as ”Pre-Load phase”. When the thread is ready to store its outputs, it issues several FWRITEs as needed: such phase is denoted as ”Post-Store phase”. In the original SDF architecture and in the DTA, each core is encompassing two or more pipelines [38] that can operate in parallel by partially overlapping the execution of the rest of instructions (Execution Pipeline) and Load/Store instructions (Synchronization Pipeline)

In other words, during the Pre-Load phase data is read from a local memory and possibly stored in local registers. In the Execution phase thread executes without any memory access and uses only the data that are in the registers. In the Post-Store phase after the computation is finished, a thread writes data to other threads' DF-frames and, if needed, it writes data to main memory.

The DTA is based on this execution paradigm and adds the concept of clustering resources and the Distributed Scheduler (DS) instead of a single centralized TSU as in the SDF, trying to address the on-chip scalability problem [7]. Each cluster in the architecture has the same structure and can be considered as a modular

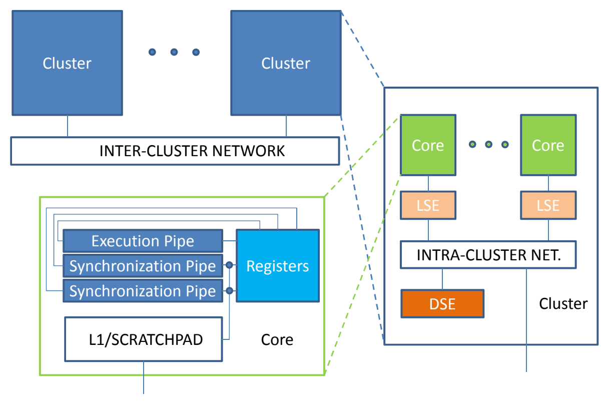


Figure 2: Overview of DTA architecture. LSE=Local Scheduling Unit. DSE=Distributed Scheduling Unit.

tile of the architecture. Scalability can be achieved by simply adding tiles.

DTA consists of several clusters, as seen in Figure 2. Each includes one or more cores and a Distributed Scheduler Element (DSE). Each core needs a Local Scheduler Element (LSE) to talk with the DSE. The set of all DSEs constitutes the Distributed Scheduler (DS).

This property of the cluster logically leads to the need of a fast interconnection network inside the cluster (intra-cluster network), while the network for connecting all clusters (inter-cluster network) can be chosen with more flexibility. The actual amount of processing elements that can fit into one cluster will depend on the technology that is used.

Other approaches to Dataflow Architectures attempt to map the program directly in the FPGA, such is in the ChipCflow project [2], in the MAXELER dataflow computers [22], exploit partial reconfiguration of the programmable logic to map multiple cores [29], or using a dataflow substrate that can be governed through fast reconfiguration of the dataflow graph and asynchronous execution [35, 36]. Recent efforts provide also first hardware evaluation of TSU-like architectural support [21].

1.2 Transactional Memory in Haskell

Haskell is a purely functional language born in 1990 and has been further specified in the Haskell98, Haskell2010, Haskell 2014. The Glasgow Haskell Compiler (GHC) [23] is the de-facto standard. It provides a compilation and runtime system for Haskell [17], a pure, lazy, functional programming language and supports strong static typing. Since version 6.6 (we used the version 6.8) GHC contains STM functions built into the Concurrent Haskell library [25], providing abstractions for communicating between explicitly-forked threads. STM is expressed elegantly in a declarative language and Haskell's type system (particularly the monadic mechanism) forces threads to access shared variables only inside a transaction. This useful restriction is more likely to be violated under other programming paradigms, for example direct access to memory locations [11], [12].

Listing 1: Code for the recursive Fibonacci algorithm in Haskell.

```
fib :: Int —> Int
fib 0 = 1
fib 1 = 1
fib n = fib (n−1) + fib (n−2)
```

Although the Haskell is very different from other languages like C# or C++ (see Listing 1), the actual STM operations are used with an imperative style, thanks to the monadic mechanism, and the STM implementation uses the same techniques used in mainstream languages [13]. The use of monads also grants a safe access to shared memory only inside a transaction and assures that I/O actions can be performed only outside a transaction. This guarantees that shared memory cannot be modified without the protection of the Haskell *atomically* function. This kind of protection is known as “strong atomicity” [20]. Moreover this context makes possible the complete separation between computations that have side-effects and the ones that are effect-free. Utilizing a purely-declarative language for TM also provides explicit read/writes from/to mutable cells (cells that contain data of different types). Memory operations that are also performed by functional computations are never tracked by STM unnecessarily, since they never need to be rolled back [13].

Threads in STM Haskell communicate by reading and writing transactional variables, or TVars, pointers to shared memory locations that can be accessed only within transactions. All STM operations make use of the STM monad [14], which supports a set of transactional operations, including the functions *newTVar*, *readTVar* and *writeTVar*, which perform the operation of creating, reading and writing transactional variables as shown in Table 2.

When a transaction is finished, it is validated by the runtime system by looking if it is executed on a consistent system state and no variable used in the transaction was modified by some other executed thread. In such case, the modifications of the transaction are committed, otherwise, they are discarded [24].

The other operation in the STM monad are the *retry* and *orElse* functions. The first blocks a transaction until at least one of the TVars it uses is modified. The second allows two transactions to be tried in sequence. If the first makes a retry then the second starts.

Table 2: Haskell STM operations.

STM Function	Haskell Type
atomically	STM a ->IO a
newTvar	a ->STM (TVar a)
readTVar	TVar a ->STM a
writeTvar	TVar a ->a ->STM()
retry	STM a
orElse	STM a ->STM a ->STM a

The Haskell STM runtime maintains a list of accessed transactional variables for each transaction, where all the modified variables are in the "writeset" and the read (but not written) ones are in the "readset" of the transaction. This list is maintained in a per-thread transaction log that records the state of the variables before the beginning of the transaction and every access made to those TVars. When *atomically* function is invoked, the STM runtime checks that these accesses are valid and that no concurrent transaction has committed conflicting updates. In case the validation turns out to be successful, then the modications are committed.

2 Compiling from Haskell to DTA Assembly

We developed a simple compiler prototype for Haskell programs able to create an equal representation in DTA assembly.

The Glasgow Haskell Compiler (GHC) uses an intermediate language, called "Core" [26] [27] as its internal program representation during some pass in the compiler chain. The Core language consists of the lambda calculus augmented with let-expressions (both non-recursive and recursive), case expressions, data constructors, literals, and primitive operations. Actually GHC's intermediate language is an explicitly-typed language based on System FC [32].

As a front end of our compilation toolchain, we use External Core (EC) [33], which is an external representation of this language created by GHC's developers to help people trying to write part of an Haskell compiler to interface with the GHC itself.

The utilization of External Core representation has some advantages compared to the use of Core, the internal representation of this Intermediate language. First, the existence of some instruments already using this language. In particular, we extracted an EC parser from the front end tool of an existing tool for translating EC to Java and found some Haskell library in this format, which helps the specifications of some data types. Another advantage is a more expressive specification of the data types that are found during the compilation of the program. This is extremely useful in case we have to work on structured data types, and in the management of functions and methods.

The External Core presents also some disadvantages. First, in the last GHC's versions this language is generated from the direct translation of internal Core in a series of steps using operational instruments within the GHC not completely up to date. So many existing instruments existing for this language, like the ones available in GHC itself, are not 100% reliable. The biggest problems of this EC representation is the absence of the specification of primitive functions execution, the computational behavior of external functions and the way to manage overloaded functions.

Our compiler is composed of two main parts. The front end has the task of analyzing the EC input file, performing the lexical and semantic analysis, then returning a data structure holding the lexical tree representing the entire examined program. Using this tree, a series of steps are performed to optimize the code generated in the first draft to simplify its structure. In particular, a rewriting phase applies a series of transformation to that tree to make sure the module is in canonical form, so it can not be reduced further, and it is ready for the next steps. Once this part is finished, the back end of our compiler analyzes the data tree, according to the features of the program's workflow. Then the corresponding DTA code is generated.

2.1 Translating standard elements

A Haskell program is made by a series of modules, which can be freely imported by each others, each of them specifies two categories of definitions:

1. **Type definitions:** definitions that are created from data and newtype keywords. They represent data types specified within the module where that keyword appear. A data type, in Haskell, can be seen as a set of values with common features. It can be *simple*, if the data is a primitive type, such as integer, character values or enumerations. On the other way, it could be a structured data type, if it is defined by using other types. The major examples of such data types, in Haskell, are lists or tuples. In our compiler the type management is made by creating data structures that are specific for the data type,

which hold a list of possible values of the specific type. Then, during the compilation, the assignment is made according with some simple rules:

- *Primitive types* are translated straightforwardly, whenever a value of this type is found is directly assigned to the container. Enumerations are coded in a very similar way; the possible values are assigned accordingly to the coded values in the data definition.
- *Structured data types* are implemented as tree-like structures that holds the data types used in their definition.

The Haskell's static type system assures that, within this model, errors never occur during the execution phase. Whenever an evaluation takes place, the result is guaranteed to have the correct data type, so the system is able to manage it.

2. **Top definitions:** they are a list of high level functions, that are bound to an expression that specifies the actual operations to execute. Top level definitions can be constant functions or non-constant functions. Constant functions, representing a fixed value within the program flow, can be seen as a global variable. Non-constant functions perform actual computation. The first are translated by our tool, according to the type of expression, into assignments. The second are translated in a series of DTA threads holding all the needed instructions, representing the expressions describing the execution of the function. Each expression is specifically computed according to its particular definition. The possible definitions of an expression are the following:

   - *Var:* it represent the invocation within a function of a variable, bound to a specific function. In our implementation a constant primitive or enumeration variable is simply translated in a series of direct assignment to values. Although this is different from Haskell general case, where every variable is stored into the Heap, for implementing its lazy behavior, we choose this solution that does not have side effects.

     On the other hand, if the variable represents a complex data type, an ad-hoc data structure is allocated in the memory, according to the data type definition, and the pointer to this area is returned to the thread. If the variable is bound to a function, which makes a computation, the program at this point creates a new thread that calls the needed function to be executed, then it ends its execution. In case the parent thread needs the result of that function, the subsequent code of such thread is placed in a sub thread, which collects the needed results. Such thread is called right before the parent thread exits.
   - *Let:* this statement creates a binding of the result of an expression to a specified variable. It is treated as a local variable within the function. It is managed in the same way as a top level definition, but with different scopes. If the binding represents a different function, it is translated in a different thread in the DTA code.
   - *App:* this definition represents the passing of a series of expressions to the main expression as arguments. Typically, it describes the invocation of a function with some parameters. In our tool every expression in evaluated separately, then the results of the argument evaluation are passed to the evaluation of the main expression. If some argument creates a separate thread, the main expression also generates a new thread, which receives the return values.
   - *Lambda:* this statement makes the binding to a variable, used in the main function, of a value coming from the outside of the function itself. It is translated in a series of load instruction according to the data types.
   - *Case:* the case statement has two main duties. First, as classical control flow. It evaluates the main expression, then compares it with the possible alternatives and, according with the results, chooses the right path. The comparison is direct if the data are simple and can be stored in the registers. If this is not possible, then the EC syntax makes a binding of the value according to its type to make the inner structure of the data available to the following expressions. In this case the compiler follows the same method. The second use of case involves the creation of an Haskell class' instance. It specifies, according with the data types involved in the computation, which methods are available in the expression. Currently, the system is not able to manage automatically the translation of such methods, but the needed instructions are directly created from the compiler for some of the simpler and most used class, like arithmetic and logic operations.

2.2 The Monadic System

Another important aspect of the translation from Haskell is the monadic mechanism. It makes possible to execute actions, defined within the monad itself, sequentially. In our case, this method is translated by the creation of a series of sub-threads, called one after another. This method makes possible to execute the specific actions in the right order. Many of those sub-threads, nevertheless, have a very small body. Often the only action they execute is to call a function that actually performs the computation and passing to it the data correctly. This creates a great overhead in the execution model. A similar problem is present in the management of polymorphic functions. In Haskell, those functions are resolved creating a middle function, which calls the correct function implementation, according to the data types of the parameters. In our compiler, in this situation, a thread is created, to represents the intermediate function. This thread calls the code for the actual compilation. Usually this is a single I/O instruction in our experiments.

Currently the HDTA tool can execute the automatic translation of simple Haskell programs, performing integer calculations, basic I/O operations, and management of the more common data type like enumerations and lists.

3 Mapping Transactional Memory to DTA

As a first step for porting the STM system in DTA, we have chosen a simple benchmark performing the increment of a variable that is shared between two threads, each performing a fixed number of iterations. This program translated in a DTA implementation by hand, trying to follow as closely as possible the methods specified for the compiler, with the goal of making it as generic as possible. We use this program as a starting point to have some ideas about the concurrency system, the performance, and the problems that the introduction of this model can generate.

We implemented in DTA three basic mechanisms that we illustrate below:

1. Generating concurrent threads.
2. Managing thread synchronization and communication.
3. Basic transactional memory system.

3.1 Generating concurrent threads

The Haskell concurrent paradigm involves the use of *forkIO* function to create concurrent threads, which takes as input an Haskell *IO* action (it can be a single action or, more likely, a sequence of actions) creating a thread operating concurrently with the main thread. To implement this mechanism, we use the threading system that is present in the DTA paradigm. We consider the whole input action as the body of the generated thread.

3.2 Managing thread synchronization and communication

In Haskell, communication among concurrent threads is supported by the use of particular data structures called *MVar*. Each MVar represents a reference to a mutable location, which can be empty or full with a value. The communication between threads makes use of those variables. If they are in an unsafe state (if a reading MVar is empty, or a writing MVar is full) the thread is blocked until the state is safe. It is important to point that Haskell thread are blocking, while DTA thread are not, so we had to introduce a way to translate the behavior of blocking threads in a non blocking environment, trying to maintain a close similarity with the original Haskell behavior. In our DTA implementation, when a thread should block, it saves the needed data for continuing its execution then it terminates itself. Such data are (this is similar to generate a new thread):

1. The parameters needed to the continuation of the execution.
2. The values needed to restart the execution of the thread from the right point, like the thread identifier, the pointer to the correct restarting thread and the number of the needed parameters.

3.3 Basic transactional memory system

For the implementation of the basic transactional system, we dealt with the reading and writing of TVars, the creation and management of logs, and the validation and commit of transactions. Currently our implementation is based in DTA code. The basic components of the model are the *Transaction Initialization* and the *Transaction Validation*.

- *Transaction Initialization:* In Haskell, a transaction is the body of the *atomically* function. Like in the case of *forkIO* function, the code within this function is translated into a series of threads, maintaining the correct dependency order between the specified STM actions. A starting thread has the duty tasks of activating the above function threads and creating the Transactional Log. This thread makes sure, according to the transactional variables that are present in the function writeset and readset, that the parameters passed to the function will go to write and read the log instead of the actual memory location, making the computation safe until it will be validated. This method makes possible to face another general feature of Haskell functional behavior, that is the passing of functions as arguments to other functions. This feature was treated in a very similar way to the thread synchronization problem. A memory structure is created, containing all the information needed to execute a thread. The management of the communication of parameters involving function passing is one of the greatest difficulties in our efforts to manage an automatic Haskell-DTA translation, and yet not fully resolved.

- *Transaction Validation:* The validation is done by a specific thread called at the end of the operations described in the transaction body. This thread performs the control needed to assure the correctness of the transaction, by checking that the value of the variables involved in the transaction is not changed. The validation, and eventually the commit phases, are serialized in order to assure the TVars consistency.

In our example we use only a single TVar, so we used the simplest mechanism available in the STM system, a global lock on the entire TVar set. We are working to extend this simple mechanism to manage more complex situations, like a layer TVar set for transactions, and the use of complex data types. To solve the above problems we are evaluating the possibility to support purely software mechanisms by using the underling architecture, like convenient System Calls or ad hoc instructions.

4 Experiments and Methodology

We make three different types of experiments.

The first experiment (see 5.1) involves the evaluation of a DTA program compiled by our tool from Haskell (here named *HDTA*, compared with an equivalent program, initially specified in C language and then hand-coded directly into DTA assembly. As benchmark we choose the recursive Fibonacci algorithm as it can easily generate a large number of threads, with n=15 as input. The purpose of this experiment is to validate the Haskell-to-DTA toolchain and evaluate its performance again the C-to-DTA (hand-coded) translation. In this case, we also test the strong scaling by varying the number of cores from 1 to 8 and keeping a fixed input.

The second experiment (see 5.2) aims at comparing the performance of the code generated by our tool (HDTA) against the code generated by the GHC on an x86 machine. We used also in this case the same recursive Fibonacci written in Haskell (see Listing 1). For a fair comparison, since we were not interested in optimizing our code, we compiled with no optimization also in GHC. In this case we also investigated the weak scaling by keeping the number of processors fixed to one and varying the value of the input (n=10,12,15,17).

In the third experiment (see 5.3) we choose a simple benchmark that is appropriate to test the Transactional memory. We choose the “counter” benchmark (Listing 2) where two concurrent threads try to update a shared counter protected by the Transactional Memory (atomically keyword), in order to create a situation of potential contention (this is typically a worst case since most likely there will be serialization of the transactions). In order to create a true contention we use in this case two active processors (cores), both in the simulator and in the actual x86 system (GHC has parallel execution enabled). The counter should update the counter up to 10000 (5000 for each of the two running threads.

Listing 2: Code of the “counter” benchmark.

```haskell
module Main where
import GHC.Conc
import Control.Concurrent

incTVar :: TVar Int -> STM ()
incTVar a = do
  tmp <- readTVar a
  writeTVar a (tmp+1)

loop n f = mapM_ (\_ -> f) [1..n]

main = do
  x <- newTVarIO 0
  n <- return 5000
  join1 <- newEmptyMVar
  join2 <- newEmptyMVar
  forkIO $ do
    loop n $ do
      atomically (incTVar x)
    putMVar join1 ()
  forkIO $ do
    loop n $ do
      atomically (incTVar x)
    putMVar join2 ()
  takeMVar join1
  takeMVar join2
  tmp <- atomically (readTVar x)
  print tmp
```

4.1 Methodology

All the experiments are executed on an Intel Core-2 (E8200) 2.66 GHz, with a 1.33 GHz front side bus. This is a superscalar processor with Wide Dynamic Execution that enables each core to complete up to four instructions per cycle.It has a 32KB L1 data cache and 32KB instruction 8-way associative cache, and a 6MB shared L2 32-way associative cache. This processor optimizes the use of the data bandwidth from the memory subsystem to accelerate out-of-order execution and uses a prediction mechanism that reduces the time in-flight instructions have to wait for data. New prefetch algorithms move data from system memory into L2 cache in advance of execution. These features help keep the processor pipeline full.

The performance of the DTA programs is analyzed by using a simulator with a 5-stage pipeline, 64 registers and 4096 frames of 256 64-bit entries. DTA uses a 128-bit intra-cluster single bus with 2-cycle latency and a 128-bit intra-cluster with 4-cycle latency. Since the purpose of these experiments is to validate the software side of the toolchain, we assume a perfect memory hierarchy. More in general, recent state-of-the-art for on-chip memory hierarchy should assume a Non-Uniform Cache Architecture (NUCA) [1]. Although the DTA architecture can work with a large number of cores and clusters [7] in this setup we are analyzing a single cluster.

The performance analysis of the Haskell programs, which are compiled for the x86 machine, is performed with the following methodology. The information about the cycles that the x86 processor spent while running the GHC-compiled Haskell programs are collected through the use of the RDTSC instruction [18] in order to read the 64-bit internal Time Stamp Counter (TSC), present in the x86 machines since the Pentium. The GHC-x86 memory footprint is obtained by the profiling options available in GHC (+RTS -s). The GHC-x86 cache statistics are captured by using the *cachegrind* tool [34], which allows to keep track of the user and library memory accesses, without the operating system (kernel) interference.

5 Analysis of the Experimental Results

In this section we present and discuss the results of the three experiments that we just introduced. The three experiments aim at validating the HDTA toolchain (Haskell-to-DTA) and in particular for what concern the support for the Transactional Memory on the Dataflow based execution model offered by the Decoupled Threaded Architecture (DTA).

5.1 Experiment-1

First, we evaluate the CPU cycles needed to perform the calculation of the Fibonacci number with input n=15, while varying the number of cores (Figure 3). We can see that the code generated automatically by our HDTA tool (Fib-HDTA) is more efficient than the manually coded version (Fib-manHDTA), indicating that the HDTA generates more optimized code for the Dataflow based architecture (DTA). The advantage of the HDTA version is more evident in Figure 4 where we took as reference the execution time of the manually generated DTA code: the HDTA code results from about 40% to about 90% faster than the manually generated version. This is likely due to the fact that, not only the Haskell program generates less threads, but it does so only when needed, while the manually generated version creates threads at the beginning of its execution then waits for all them to complete their computation, as confirmed in Figure 7.

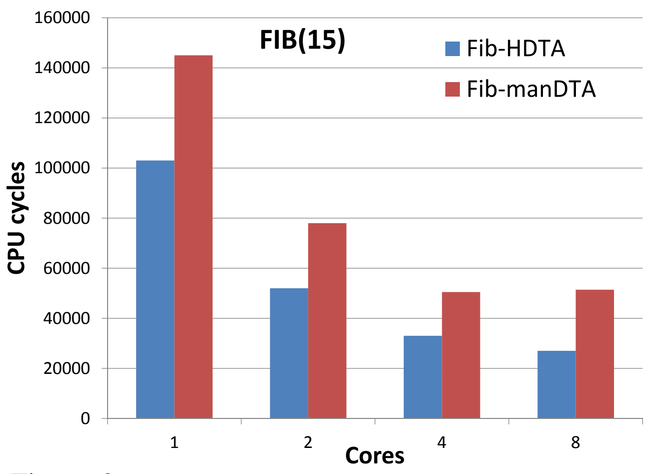


Figure 3: Number of CPU cycles when executing Fibonacci(15) in the our Haskell-generated DTA version (Fib-HDTA) and the manually coded DTA version (Fib-manDTA) while varying the number of cores.

In order to better understand the reasons of this advantage in case of the HDTA-generated code, we examine the breakdown of the instructions executed by our benchmark in the two cases, in a single core configuration (Figure 5). As we can see, the number of FALLOC instructions (DF-frame allocation) is lower in the HDTA-generated code compared to the manually generated code. This can be seen more clearly in Figure 6. In the same figure, the number of FREAD/FWRITEs tell us the data that is passed from one thread to another in a producer-consumer fashion. The number of arithmetic and logic operations (ALU) show that the HDTA coded version needs less ALU operations. Regarding the FFORKs, since they act as

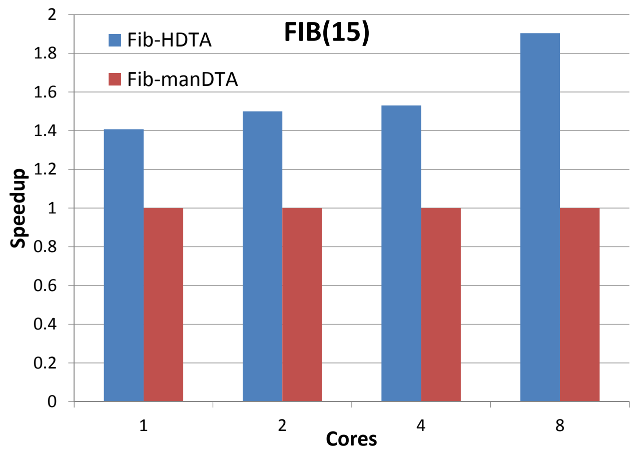


Figure 4: Speedup of our Haskell-generated DTA version (Fib-HDTA) versus the manually coded DTA version (Fib-manDTA) while varying the number of cores.

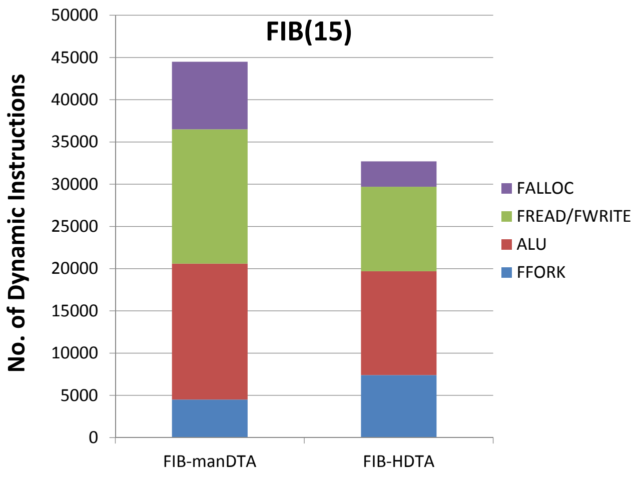


Figure 5: Number of instructions dynamically executed by the compilation of Fibonacci(15) and a single core.

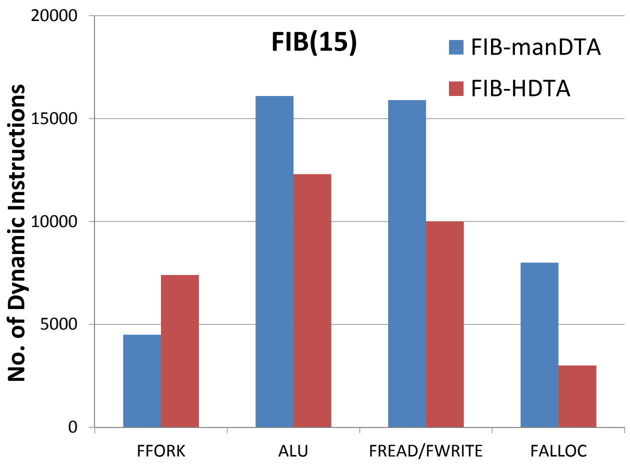


Figure 6: Comparing the number of instructions by type in case of Fibonacci(15) and a single core.

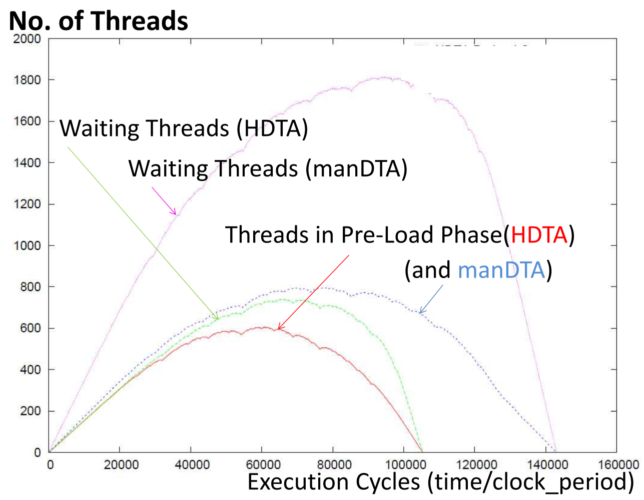


Figure 7: Number of threads in the pipeline queue and in the wait table generated by the compilation of Fibonacci(15).

a switch of the control between the execution and synchronization pipeline we observe an opposite behavior from the previous instruction categories. This means that the HDTA needs to switch more often among the two pipelines, since Haskell, because of its functional behavior, generates a less linear workflow.

We also examined the utilization of the pipeline during the execution of the two versions of the benchmark (not shown in graphs): in both cases (HDTA and manDTA we observed a high utilization, about 96% for manDTA and about 97% for the HDTA generated code. This tell us that the code is in both cases well suited for the DTA architecture. Finally, we show the number of threads that are generated dynamically (Figure 7) and that are either waiting for their inputs (Waiting Threads) or ready to be executed (they completed the Pre-load phase). Those results shows both cases (HDTA and manDTA) have a similar execution flow, but the HDTA version generates less threads, while the other seems to create early most of the threads, which therefore wait longer.

5.2 Experiment-2

In this case we compare the execution cycles for several inputs (n=10,12,15,17) in case of a single core (Figure 8) and contrast the execution obtained by the HDTA tool versus the x86 code running as generated by GHC and running on a real machine. According to the experimental data, the HDTA generated code shows a better behavior as it needs only a fraction of the time needed by the GHC generated code. The HDTA version shows a much better scalability with the input set, but we should recall that the DTA evaluation assumes a perfect memory hierarchy. However, this assumption should not impact much since the memory footprint is not so large as discussed in the following.

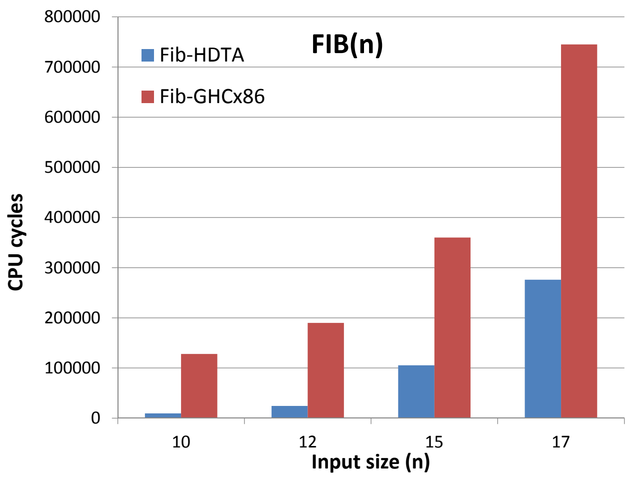


Figure 8: Number of CPU cycles of the execution of Fibonacci(n): DTA versus x86 GHC code.

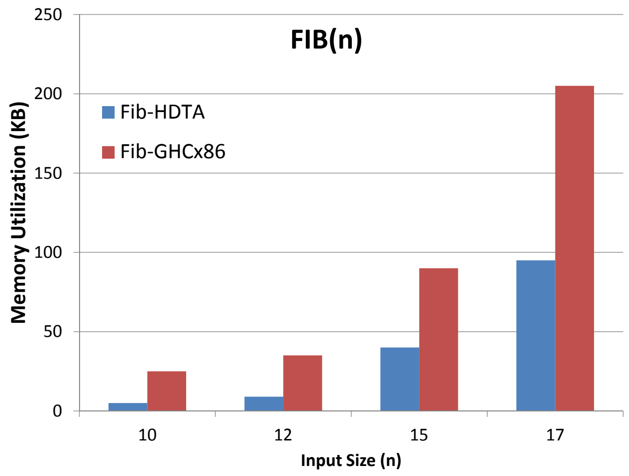


Figure 9: Memory Allocated (in KB) by the execution of Fibonacci(n): DTA versus x86 GHC code.

To evaluate the impact of the memory hierarchy, we collected the memory footprint size (Figure 9). It is important to point out that in the DTA, accesses are essentially due to loads and stores into the frame memory, not in the shared (main) memory (Figures 10 and 11). Instead, the GHC compiled program uses only the main memory during its execution. However, also the x86 GHC code the cache performance is very good: the miss-rate that we measured is under 2% (not shown), so we can assume the memory latency is not the main bottleneck.

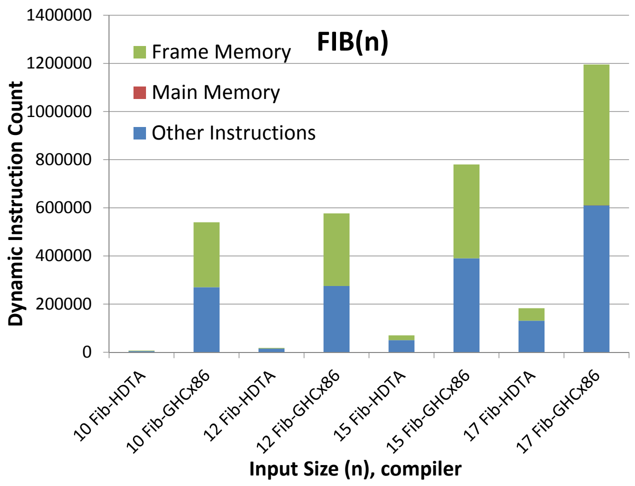


Figure 10: Dynamic Instruction Count for Fibonacci(n): DTA versus x86 GHC code.

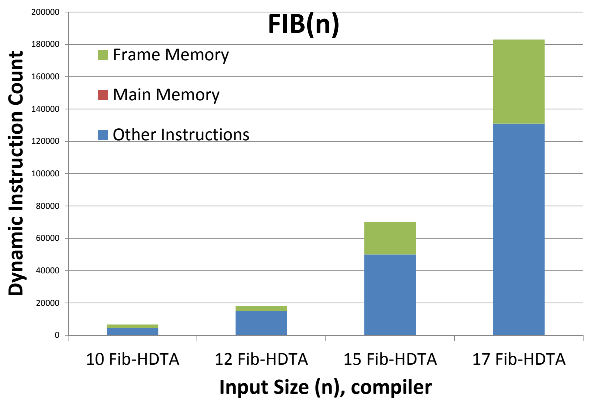


Figure 11: Dynamic Instruction Count for Fibonacci(n): DTA code (zoom).

More in detail, we see that the instructions dynamically executed by the program generated with the HDTA tool are much less than those related to the x86 GHC code (Figures 10 and 11), even by considering the inner parallelism of the processor that helps contain the execution time. The x86 GHC generated code makes a wide use of the main memory, about half of the instructions are memory accesses, while the HDTA code did not use the main memory at all and its number of memory accesses is lower than for GHC. Moreover, the dynamic instruction count does not depend by the memory hierarchy and it clearly indicates a large advantage for the DTA architecture.

In conclusion, also in this second experiment, we observed a good behavior of the HDTA tests compared to the x86 GHC generated code. It has to be noted that much of the complexity of the GHC compiled programs comes from the run time systems, which performs a great deal of work to implement the laziness of the program, the management of the heap, and the garbage collector functionality. While our program only performs pure calculation, without a lazy behavior. Nevertheless, those results show that the use of Haskell on DTA architecture is a promising way to exploit the capabilities of this paradigm, as it generates almost an order of magnitude less dynamic instructions in same cases.

5.3 Experiment-3

As before, we start evaluating the CPU cycles for different inputs as we contrast the HDTA generated code with the x86 GHC generated code.

In this case the input number is represented by the number of times we wish to increment the shared counter (Figures 12, 13, 14, 15). The counter total is reported on the horizontal axis. In this experiment the Transactional Memory protection is used to access the shared counter. As we can see, the behavior of the HDTA generated code is better. Also in this case, as in the previous experiment, we evaluated the impact of the memory footprint and the impact of the memory hierarchy for the case of the x86 code.

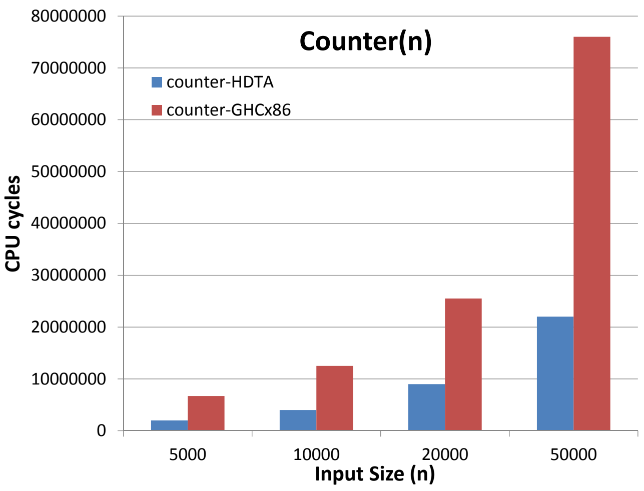


Figure 12: Number of CPU cycles for the shared-counter benchmark using TM (Listing 2): DTA versus x86 GHC code.

In Figure 13, the memory footprint is reported. We see that the HDTA code needs only a half of the memory needed by the GHC code. This is likely due to the lazy behavior of Haskell, which has a greater and more sophisticated use of the memory. Also, as shown in Table 3, the GHC compiled program has a very low miss rate both in the L1 (under 1%) and L2 (under 0.3%). This is in-line with the fact that the footprint is smaller than the last level cache size.

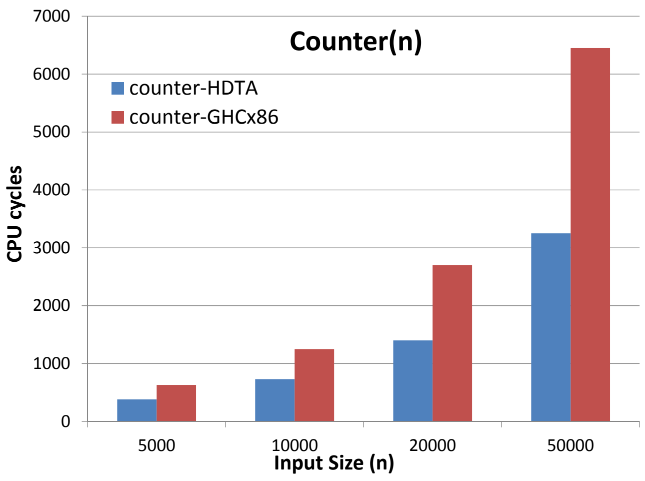


Figure 13: Memory Allocated (in KB) by the execution of Counter(n): DTA versus x86 GHC code.

Table 3: Cache miss rate for the GHC execution of the counter example (Listing 2).

	2500 It	5000 It	10000 It	25000 It
L1-I Miss Rate	0.02%	0.01%	7.5E-3%	3.3E-3%
L1-D Miss Rate	0.9%	0.8%	0.8%	0.8%
L2-I Miss Rate	0.01%	5.2E-3%	2.6E-3%	1.0E-3%
L2-D Miss Rate	0.3%	0.2%	0.1%	0.1%

Third, we show the dynamic instruction generated by the our DTA programs by the GHC compiled program (Figures 14 and 15). The comparison shows that both benchmarks use about half instructions to access the memory, but the HDTA code uses only a part of those instruction to access the main memory. This is certainly due to the simplicity of our Transactional Memory mechanism compared the GHC TM mechanism and the whole run time system.

Those results show us that even a really simple implementation of a TM mechanism in DTA environment has a good performance and show the good potential of the proposed approach, despite the limitations that we pointed out.

In this case, differently from the previous experiment, one important point is that the parallelism generated by the purely functional calculation, like Fibonacci appears much greater than in the case of a pro-

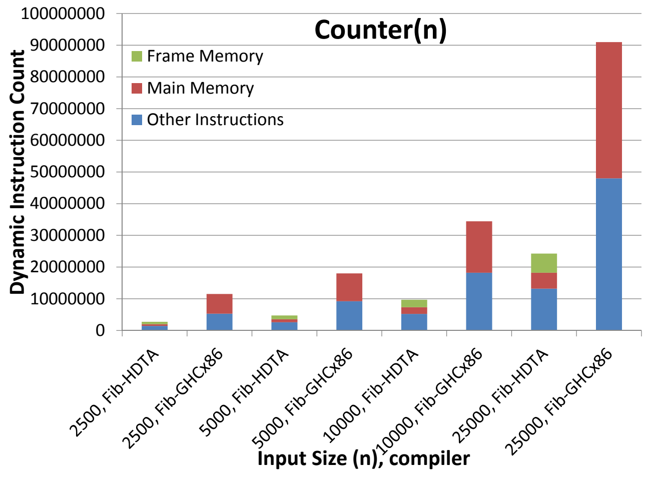


Figure 14: Dynamic Instruction Count for Counter(n): DTA versus x86 GHC code.

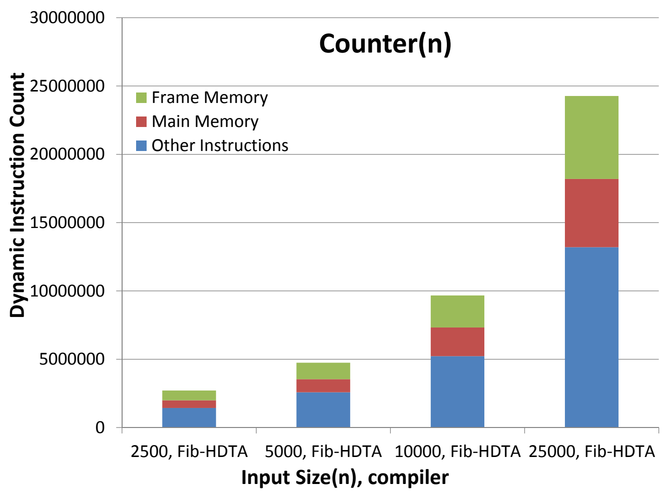


Figure 15: Dynamic Instruction Count for Counter(n): DTA code (zoom).

gram only using monad computations. In fact, in the latter case no more than three or four parallel threads were generated.

This work shows the possibility of combining Transactional Memory on a dataflow model of execution and that the obtained performance can be very competitive.

6 Conclusion

The main goal of this research is to demonstrate that it is possible to translate the functional behavior of the Haskell language in a dataflow based architecture like DTA. In particular, the Transactional Memory support has been proved possible and efficient as compared to the Software Transactional Memory support provided by the GHC.

We realized a first version of a compiler from Haskell to DTA assembly (called HDTA), that is available under request. We compared the performance of the code that was generated by HDTA both against manually translated C-code into DTA assembly and against the x86 code generated by the GHC. Despite some practical limitation of the experiments that can be easily overcome in the future, we have clearly found that HDTA largely outperforms (in same case of an order of magnitude) GHC in terms of instructions that are dynamically generated and also in terms of number of memory accesses.

The Transactional Memory implementation of HDTA could be even improved and it is also competitive with the one provided by GHC.

Dataflow models proved again their better efficiency.

Acknowledgments: We thank Roberto D'Aprile for setting up the initial framework. This research is partly funded by the EU through projects HiPEAC (id. 287759), TERAFLUX (id. 249013) and AXIOM (id. 645496).

References:

[1] S. Bartolini, P. Foglia, C. A. Prete, and M. Solinas. Coherence in the cmp era: Lesson learned in designing a llc architecture. *WSEAS Trans. on Computers*, 13:195–206, 2014.

[2] A. Fernandes Da Silva, J. Lopes, B. De Abreu Silva, and J. Silva. The chipcflow project to accelerate algorithms using a dataflow graph in a reconfigurable system. *WSEAS Trans. on Computers*, 11(8):265–274, 2012.

[3] R. Giorgi. TERAFLUX: Exploiting dataflow parallelism in teradevices. In *ACM Computing Frontiers*, pages 303–304, Cagliari, Italy, May 2012.

[4] R. Giorgi. Accelerating haskell on a dataflow architecture: a case study including transactional memory. In *Proc. Int.l Conf. on Computer Engineering and Application (CEA)*, pages 91–100, Dubai, UAE, February 2015.

[5] R. Giorgi et al. TERAFLUX: Harnessing dataflow in next generation teradevices. *Microprocessors and Microsystems*, 38(8, Part B):976 – 990, 2014.

[6] R. Giorgi and P. Faraboschi. An introduction to df-threads and their execution model. In *IEEE Proceedings of MPP-2014*, pages 60–65, Paris, France, oct 2014.

[7] R. Giorgi, Z. Popovic, and N. Puzovic. DTA-C: A decoupled multi-threaded architecture for CMP systems. In *Proceedings of IEEE SBAC-PAD*, pages 263–270, Gramado, Brasil, October 2007.

[8] R. Giorgi, Z. Popovic, and N. Puzovic. Exploiting DMA to enable non-blocking execution in decoupled threaded architecture. In *Proceedings of IEEE International Symposium*

on Parallel and Distributed Processing - MTAAP Multi-Threading Architectures and APplications, pages 2197–2204, Rome, Italy, may 2009. IEEE.

[9] R. Giorgi, Z. Popovic, and N. Puzovic. Introducing hardware tlp support for the cell processor. In *Proceedings of IEEE International Workshop on Multi-Core Computing Systems*, pages 657–662, Fukuoka, Japan, mar 2009. IEEE.

[10] R. Giorgi and A. Scionti. A scalable thread scheduling co-processor based on data-flow principles. *ELSEVIER Future Generation Computer Systems*, pages 1–10, 2015.

[11] T. Harris, A. Cristal, O. S. Unsal, E. Ayguade, F. Gagliardi, B. Smith, and M. Valero. iTransactional Memory: An Overview. *IEEE Micro*, 27(3):8–29, May 2007.

[12] T. Harris, J. Larus, and R. Rajwar. *Transactional Memory*. Morgan and Claypool Publishers, 2nd edition, 2010.

[13] T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy. Composable Memory Transactions. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, pages 48–60, New York, NY, USA, 2005. ACM.

[14] Haskell Documentation Library. Software transactional memory page. {https://hackage.haskell.org/package/stm-2.1.1.2/docs/Control-Monad-STM.html}. [Online; accessed March-2015].

[15] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-free Data Structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM.

[16] N. Ho, A. Mondelli, A. Scionti, M. Solinas, A. Portero, and R. Giorgi. Enhancing an x86_64 multi-core architecture with data-flow execution support. In *ACM Proc. of Computing Froniers*, pages 1–2, Ischia, Italy, May 2015.

[17] G. Hutton. *Programming in Haskell*. Cambridge University Press, New York, NY, USA, 2007.

[18] Intel Corporation. *Intel® 64 and IA-32 Architectures Developer's Manual: Vol. 2B*, January 2015.

[19] K. M. Kavi, R. Giorgi, and J. Arul. Scheduled dataflow: Execution paradigm, architecture, and performance evaluation. *IEEE Trans. on Computers*, 50(8):834–846, aug 2001.

[20] M. Martin, C. Blundell, and E. Lewis. Subtleties of Transactional Memory Atomicity Semantics. *IEEE Comput. Archit. Lett.*, 5(2):17–17, July 2006.

[21] G. Matheou and P. Evripidou. Architectural support for data-driven execution. *ACM Trans. Archit. Code Optim.*, 11(4):52:1–52:25, January 2015.

[22] M. Milutinovic, J. Salom, N. Trifunovic, and R. Giorgi. *Guide to DataFlow Supercomputing*. Springer, Berlin, April 2015.

[23] U. of Glasgow. GHC official site. {http://www.haskell.org/ghc/}. [Online; accessed March-2015].

[24] C. Perfumo, N. Sonmez, A. Cristal, O. S. Unsal, M. Valero, and T. Harris. Dissecting Transactional Executions in Haskell. In *Second ACM SIGPLAN Workshop on Transactional Computing*, August 2007.

[25] S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, PoPL '96, pages 295–308, New York, 1996. ACM.

[26] S. Peyton Jones and S. Marlow. Secrets of the Glasgow Haskell Compiler Inliner. *J. Funct. Program.*, 12(5):393–434, July 2002.

[27] S. L. Peyton Jones. Compiling Haskell by Program Transformation: A Report from the Trenches. In *Proceedings of the 6th European Symposium on Programming Languages and Systems*, ESOP '96, pages 18–44, London, UK, UK, 1996. Springer-Verlag.

[28] N. Shavit and D. Touitou. Software Transactional Memory. In *Proc. of the 12th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 204–213, 1995.

[29] J. Silva and J. Lopes. A dynamic dataflow architecture using partial reconfigurable hardware as an option for multiple cores. *WSEAS Trans. on Computers*, 9(5):429–444, 2010.

[30] M. Solinas et al. The TERAFLUX project: Exploiting the dataflow paradigm in next generation teradevices. In *Proceedings - 16th Euromicro Conference on Digital System Design, DSD 2013*, pages 272–279, Santander, Spain, 2013.

[31] K. Stavrou, D. Pavlou, M. Nikolaides, P. Petrides, P. Evripidou, P. Trancoso, Z. Popovic, and R. Giorgi. Programming abstractions and toolchain for dataflow multithreading architectures. In *IEEE Proceedings of the Eighth International Symposium on Parallel and Distributed Computing (ISPDC 2009)*, pages 107–114, Lisbon, Portugal, jul 2009. IEEE.

[32] M. Sulzmann, M. M. T. Chakravarty, S. P. Jones, and K. Donnelly. System F with Type Equality Coercions. In *Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, TLDI '07, pages 53–66, New York, NY, USA, 2007. ACM.

[33] A. Tolmach. An External Representation for the GHC Core Language. {http://www.haskell.org/ghc/docs/papers/core.ps.gz}, September 2001.

[34] VALGRIND Developers. Valgrind Tool Suite. {http://valgrind.org/info/tools.html}. [Online; accessed March-2015].

[35] L. Verdoscia, R. Vaccaro, and R. Giorgi. A clockless computing system based on the static dataflow paradigm. In *Proc. IEEE Int.l Workshop on Data-Flow Execution Models for Extreme Scale Computing (DFM-2014)*, pages 1–8, aug 2014.

[36] L. Verdoscia, R. Vaccaro, and R. Giorgi. A matrix multiplier case study for an evaluation of a configurable dataflow-machine. In *ACM CF'15 - LP-EMS*, pages 1–6, May 2015.

[37] P. Wadler. The Essence of Functional Programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, PoPL '92, pages 1–14, New York, NY, USA, 1992. ACM.

[38] Z. Yu, A. Righi, and R. Giorgi. A case study on the design trade-off of a thread level data flow based many-core architecture. In *Future Computing*, pages 100–106, Rome, Italy, sep 2011.