Accelerating Haskell on a Dataflow Architecture: a case study including Transactional Memory

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

Abstract: A possible direction for exploiting the computational power of multi/many core chips is to rely on a massive usage of Thread Level Parallelism (TLP). We focus on the Decoupled Threaded Architecture, a hybrid dataflow architecture which efficiently uses TLP by decoupling and scheduling threads on chip processing elements in order to provide on-chip scalable performance. The DTA architecture currently lacks a specific mapping to high level languages. Our idea is to use a functional language to match this execution paradigm because we think it is very fit for this environment. We choose Haskell as our language and in particular one of the features we want to implement is the concurrency control based on Transactional Memory, which is fully supported in Haskell. The main goal of this research is twofold. First, the study of a method to unite the functional paradigm of the Haskell programming language with the DTA execution paradigm. Second, the development of a Transactional Memory model for DTA architecture based on the STM (Software Transactional Memory) API. Our results show promising speedup of the Haskell based front-end for the DTA architecture.

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

1 Introduction

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 [1].

Software Transactional Memory (STM) is a concurrency control mechanism analogous to database transactions for controlling access to shared memory [2], [3]. 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.

We feel that a decoupled multithreaded architecture like DTA [4] could lead to an efficient symbiosis with Haskell and its TM API.

The advance of this research took two major steps. First, the creation of a tool to translate simple Haskell programs (using External Core intermediate language) in DTA language, looking for a good way to reproduce the program behavior. Second, the development of a DTA specific implementation of STM.

The following subsections will briefly describe the DTA architecture (section 1.1), the Haskell STM (section 1.2) and the Haskell Core (section 1.3). The rest of the paper is organized as follows: first, we describe the tool we have developed for translating Haskell programs in DTA (section 2). Second, we show our first implementation of STM in DTA (section 3). At last, we show some experiments made for evaluating the performance of our tools (section 4).

1.1 DTA

The DTA architecture [4] is a hybrid dataflow architecture that is based on the Scheduled Data-Flow (SDF) execution paradigm [5] and more recently has lead to the TERAFLUX architecture [7], [10], [6], [8], [9], which binds to coarse grained data flow and multithreading. A DTA program is compiled into a series of non-blocking threads where all memory accesses are decoupled from the execution.

Starting from a Control Data Flow Graph of a program (or a portion of it), 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 from the input and

ISBN: 978-1-61804-276-7

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.

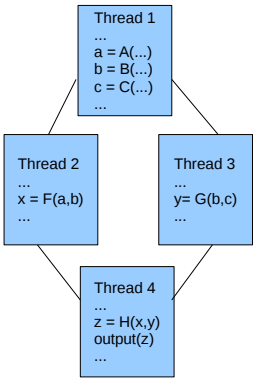


Figure 1: SDF execution paradigm.

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 by the thread. In this example, Thread 2 needs a and b, so its synchronization count is 2. Whenever the data needed by a thread are stored, synchronization count is decremented, once it reaches zero, it means that the thread is ready to execute. SDF execution model uses frames, a part of a frame memory, which is a small on-chip memory, for communicating data among threads. When a new thread is created, the system assigns a frame to it. The frame has a fixed size and it can be written by other threads and read only by the thread that it belongs to.

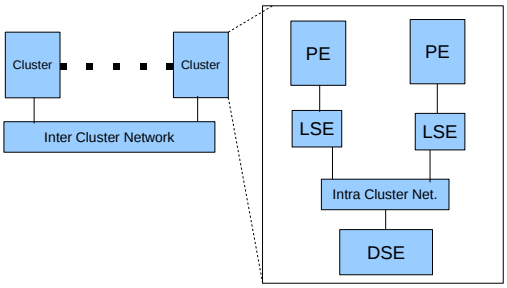


Figure 2: Overview of DTA architecture.

The DTA is based on this execution paradigm and adds the concept of clustering resources, while trying to address the on-chip scalability problem [4]. Each cluster in the architecture has the same structure and can be considered as a modular 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 Processing Elements (PE) and a Distributed Scheduler Element (DSE). Each PE need 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.

1.2 Transactional Memory

The Glasgow Haskell Compiler (GHC) [12] provides a compilation and runtime system for Haskell 98 [11], a pure, lazy, functional programming language. Since version 6.6 (we used the version 6.8) GHC contains STM functions built into the Concurrent Haskell library [14], 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 [19], [20].

Although the Haskell is very different from other languages like C# or C++, 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 [15]. 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 [17]. 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 [15].

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 [18], which supports a set of transactional operations, including the functions *newTVar*, *readTVar* and *writeTVar*, which perform the operation

ISBN: 978-1-61804-276-7

of creating, reading and writing transactional variables as shown in Table 1.

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 thread executed. In this case, the modications of the transaction are committed, otherwise, they are discarded [16].

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 start.

Table 1: 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 ones reads 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.

1.3 Haskell Core

The Glasgow Haskell Compiler (GHC) uses an intermediate language, called Core [21] [22] 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. We present a simple syntax of this language in Figure 3. Actually GHC's intermediate language is more complicated than that given here. It is an explicitly-typed language based on System FC [25].

In our tool, we use External Core (EC) [23], 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.

Program	*Prog*	→	$Bind_1$; ... ; $Bind_n$	$n \geq 1$
Binding	*Bind*	→	var = *Expr*	Non-recursive
		|	rec var_1 = $Expr_1$; ... ; var_n = $Expr_n$	Recursive $n \geq 1$
Expression	*Expr*	→	*Expr Atom*	Application
		|	*Expr ty*	Type application
		|	\ var_1 ... var_n -> *Expr*	Lambda abstraction
		|	/\ $tyvar_1$ ... $tyvar_n$ -> *Expr*	Type abstraction
		|	case *Expr* of { *Alts* }	Case expression
		|	let *Bind* in *Expr*	Local definition
		|	*con* var_1 ... var_n	Constructor $n \geq 0$
		|	*prim* var_1 ... var_n	Primitive $n \geq 0$
		|	*Atom*	
Atoms	*Atom*	→	*var*	Variable
		|	*Literal*	Unboxed Object
Literals	*Literal*	→	*integer* | *float* | ...	
Alternatives	*Alts*	→	$Calt_1$; ... ; $Calt_n$; *Default*	$n \geq 0$
		|	$Lalt_1$; ... ; $Lalt_n$; *Default*	$n \geq 0$
Constr. alt	*Calt*	→	*Con* var_1 ... var_n -> *Expr*	$n \geq 0$
Literal alt	*Lalt*	→	*Literal* -> *Expr*	
Default alt	*Default*	→	NoDefault	
		|	*var* -> *Expr*	

Figure 3: Core Syntax.

2 Haskell-DTA Compiler Tool

The first part of this work involved the development of a simple compiler prototype for Haskell programs able to create an equal representation in DTA assembly. Such application uses External Core (EC) intermediate language, produced by GHC compilation chain, as input. In this way, we exploit the front end of GHC compiler while we focus on the mapping to the DTA architecture.

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 [26], 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.

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 trans-

ISBN: 978-1-61804-276-7

formation 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.

Another important aspect to point in 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 the call to a function that actually performs the computation and passing 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. We are studying new methods to manage this situations, trying to reduce the overhead generated.

This first version of the program 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 Transactional Memory in DTA

As a first step for porting the STM system in DTA, we have chosen a simple example 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 three basic mechanism:

- Concurrent paradigm.
- Blocking threads.
- Basic transactional memory system.

Haskell concurrent paradigm involves the use of *forkIO* function, 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 realize 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.

The second point is the management of thread communication and synchronization. In Haskell, it is solved by the use of particular data structures called *MVar*. Each MVar represents a reference to a mutable location, which can be empty or full. 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. When the thread should block, it saves the data needed for continue its execution then it terminates itself. The data needed are:

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.

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 non 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 make 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 as the thread synchronization. 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 performed 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 executed in a serialized way, in order to assure the TVars consistency.

n 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 wide 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

We make three different types of experiments.

The first series involves the evaluation of a DTA program compiled by our tool from Haskell, compared with an equivalent program, initially specified in C language. The chosen program performs the calculus of the Fibonacci number, by using a recursive algorithm, with a specific input (15). First we execute this programs changing the number of processors for a fixed input, in order to show the performance of the programs on a multi-processor environment. Then we show some statistics of those executions in a single processor environment.

The second series of experiments shows the comparison between the Fibonacci program generated by our tool, and the same program compiled with GHC, with no optimization for fair comparison. The comparison is made by computing the number of Fibonacci with several inputs (10,12,15,17) for one processor, in order to compare our results on a reference implementation on a standard linux base single processor platform.

The last series of experiments tested the first version of our TM-DTA implementation. We use a counter benchmark (Figure 4) to study the potential of this work. This benchmark creates two threads, which concurrently access a shared variable protected by the TM mechanism, for a fixed number of times. The experiments are made with the same configuration of the previous experiment, only using a two process configuration both for the DTA, and for the GHC compiler (with parallel execution enabled).

```
module Main where
import GHC.Conc

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
```

Figure 4: Code of the counter benchmark.

4.1 Methodology

All the experiments are executed on an Intel Core 2 2,66 GHz, CPU E8200 with and 1,33 GHz front side bus. This is a superscalar processor with Wide Dynamic Execution that enables each core to complete up to four instruction per cycle, a 32KB L1 data cache and 32KB instruction cache 8 way associative, and a 6MB shared L2 cache 32 way associative. 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 pre-fetch algorithms move data from system memory into fast L2 cache in advance of execution. These functions help to keep the pipeline full.

ISBN: 978-1-61804-276-7

The DTA programs is analyzed using a DTA simulator with a 5 step pipeline, 64 register and 4096 frames of 64 bit per processing element. This architecture uses a 128 bit intra cluster single bus with 2 latency cycles, and an inter cluster with 128 bit and 4 latency cycles, no cache and a simulated memory access with perfect response.

The analysis of the Haskell programs compiled in the PC are obtained by the followings instruments. The statistics for memory utilization are obtained by the profiling options available in GHC.

The information about the CPU cycle spent by the Haskell programs are collected by a rdtsc library [**?**], which allows to read the Time Stamp Counter.

The cache statistic using the cachegrind tool of the valgrind profiler [28] which allows to keep track of the memory access of the user program and of the libraries it uses without involving the operation of the kernel.

4.2 Results

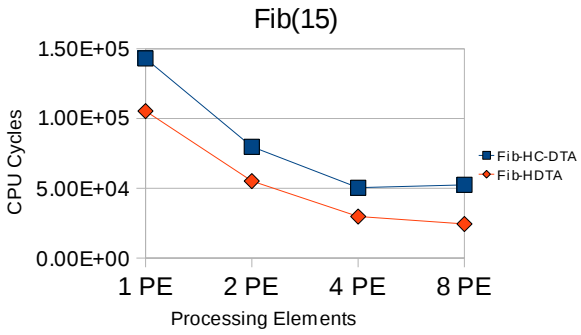


Figure 5: Number of CPU cycles of the execution of Fibonacci(15) in the Haskell generated DTA version (Fib-HDTA) and the general DTA version (Fib-HC-DTA) in a multiprocessor environment.

First, we evaluate the CPU cycles needed to end the calculation produced by the simulator (Figure 5) to compare the efficiency of those example. The experiments are made using a single cluster architecture, changing the number of Processing Elements. We see that the the implementation of the Haskell version of the program is more efficient than the general version, obtaining a reduction of CPU cycles of 30% in all the configuration of architecture. Those results shows that, for pure computational programs, the functional paradigm of Haskell has a better behavior, in this architectural environment, and it generates more optimized code. Moreover, we can see how, when augmenting the number of processing units, the version generated by our tool shows a better parallel behavior, although the Fib-HC-DTA program generates more execution thread at the same time. This is probably due to the fact that, not only the Haskell program generates less threads, but it generates them only at need, while the other program creates more threads in the beginning of its execution then waits for all them to complete their computation.

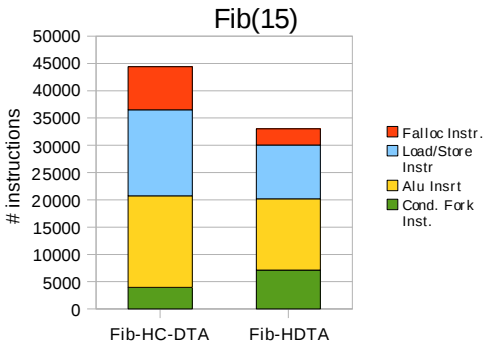


Figure 6: Number of instructions dynamically executed by the compilation of Fibonacci (15).

Second, we examine the detail of the instructions executed by the execution of our programs, in a single processor configuration (Figure 6). We begin examining the number of Falloc instructions (frame allocation), those instruction creates new threads, so we have the maximum number of threads generated from the compilation of the programs, therefore its complexity. The second comparison is between the number of load/store operations. Those instruction gives us the number of the argument passed by the various threads, not the access at the main memory, absent from those examples because they performs pure calculations. Every value refers to a couple of operation, a load and a store. The third set of instruction examined is the number of arithmetic and logic operations. Both those comparison shows the Haskell based program needs less instruction to execute correctly. In the end, we examine the use of conditional jumps inside a the threads, in this case the general program generates less jumps than the Haskell program. As expected, the Haskell programs, almost ever, uses less instructions and shows a better behavior. It manages to complete the same task generating only an half of the threads needed by the general version. Only for the jump instructions the general program outperforms the Haskell program. This shows that Haskell, because of its functional behavior, generates a less linear workflow during compilation.

At last we examine the utilization of the pipeline during the compilation, showing the maximum pipeline utilization for both the programs (Figure 7), and we can see this value is almost the same but the program compiled by our tool gains a few points. Then we show the number of threads active during

ISBN: 978-1-61804-276-7

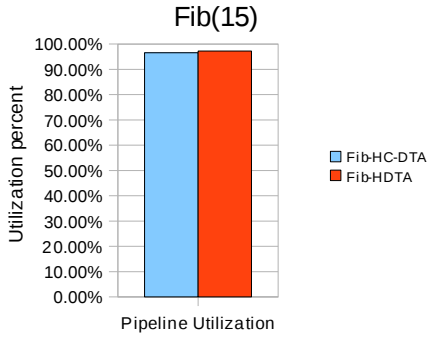


Figure 7: Maximum pipeline utilization generated by the compilation of Fibonacci (15).

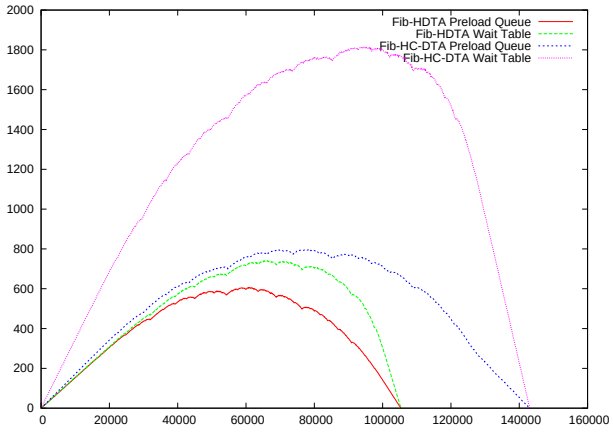


Figure 8: Number of threads in the pipeline queue and in the wait table generated by the compilation of Fibonacci(15).

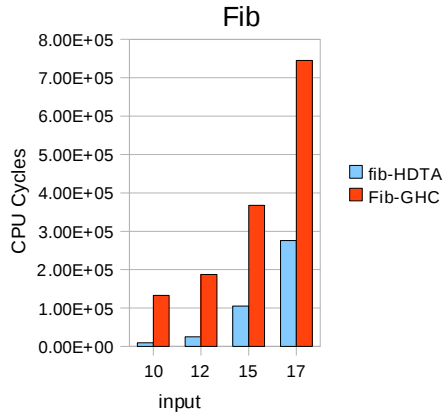


Figure 9: Number of CPU cycles of the execution of Fibonacci in the DTA simulated execution and the GHC compiled execution.

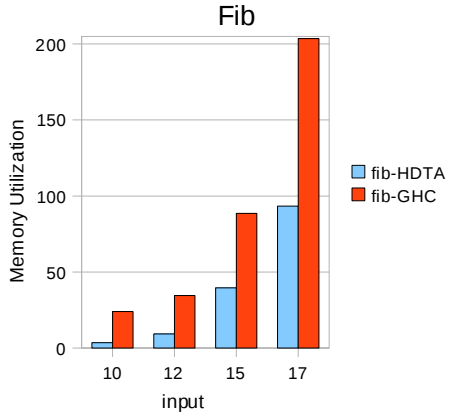


Figure 10: Memory Allocated (in KB) by the execution of Fibonacci in the DTA simulated execution and the GHC compiled execution.

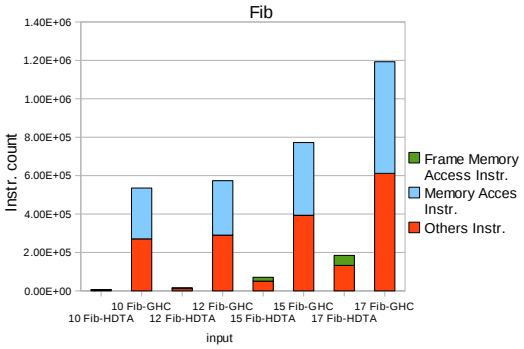
the compilation (Figure 8) both ready to executions (threads in the Pre-Load Queue), and waiting for some data to be stored (threads in the Wait-Table). Those results shows both the programs have a similar execution flow, but the Haskell programs generates less threads and only when needed, while the other seems to create early most of the treads and put them in the wait table.

Now we examine the second experiment. We begin examining the number of CPU cycles needed to complete the execution of those programs (Figure 9). According to those results, the program compiled under DTA architecture shows a better behavior. It needs only a fraction of the time needed by the Haskell program. Moreover, the DTA program, as seen in the last example, shows a very good scalability on multi processor environment, while the GHC generated execution uses a single thread.

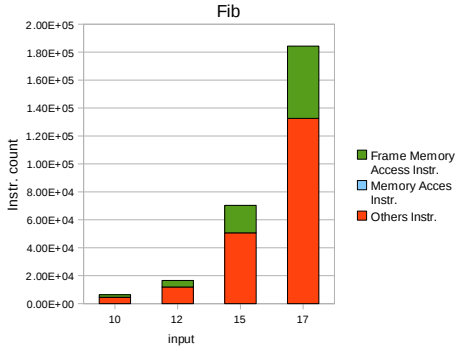
Second, we evaluate the memory request of the program expressed in KB (Figure 10). It is important to point that in the DTA program the memory refers only at the load and store phase on the frame memory, not at the main memory access. While the GHC compiled programs uses the main memory during its execution. But, according to our evaluations, the cache performance of those executions is very good. It generates a miss-rate under 2%, so we can assume the memory access is not the main bottleneck of those executions.

Third, we show the instruction generated by the DTA programs and the ones generated by the GHC compiled program (Figure 11). In this second case the number on instruction is widely bigger, and it can be pointed that the inner parallelism of the processor helps to contain the execution time. The comparison shows that in this example the GHC compilation generates an execution that makes a wide use of the main memory, about half of the instructions are memory access, while the DTA code did not use the main memory at all, and the few memory instruction are the load and store form the frame memory used for the data communication between threads.

ISBN: 978-1-61804-276-7



(a) Comparison between the GHC compilation and the DTA execution



(b) Instruction detail for the DTA compilation.

Figure 11: Instruction generated by the various executions of Fibonacci.

Those experiments show a good behavior of the DTA programs generated from Haskell, compared to the GHC generated executable. It has to be noted that many of the complexity of the Haskell 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.

Finally, we show the last series of experiments. As before, we start evaluating the CPU cycles usage, with a different number of iteration for each thread (Figure 12). In the picture is expressed the total number of iteration of the whole program. In every experiment the behavior of the DTA version of the program is better. This datum, nevertheless, is not completely accurate, because the DTA simulator didn't has a well established memory system yet. So, the calculation of the CPU cycles is not precise, and often under estimated. But, as shown in Table 2, the GHC compiled program has a good cache utilization. It generates a miss rate under 1% on the L1 cache and under 0.3% on the L2 cache. So, we assume the distance between

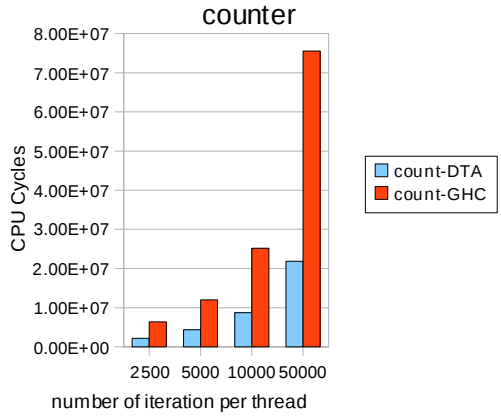


Figure 12: Number of CPU cycles of the execution of counter example (Figure 4) in the DTA simulated execution and the GHC compiled execution.

those values and a real configuration is not too large.

Table 2: Cache miss rate for the GHC execution of the counter example (Figure 4).

	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%

Second, we examine the memory request for those programs expressed in KB (Figure 13). In this case, we are comparing actual request to the main memory. We see the DTA program needs only an half of the memory needed by the GHC program. This is due to the lazy behavior of Haskell, which has a greater and more sophisticated use of the memory.

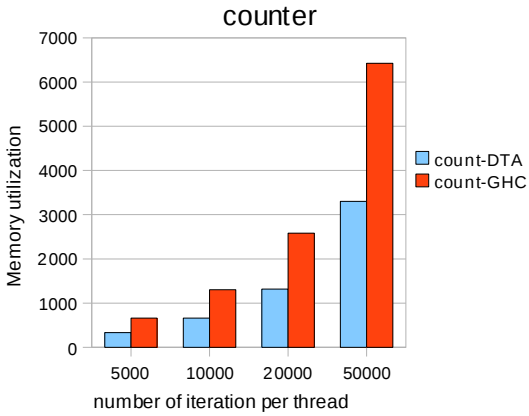
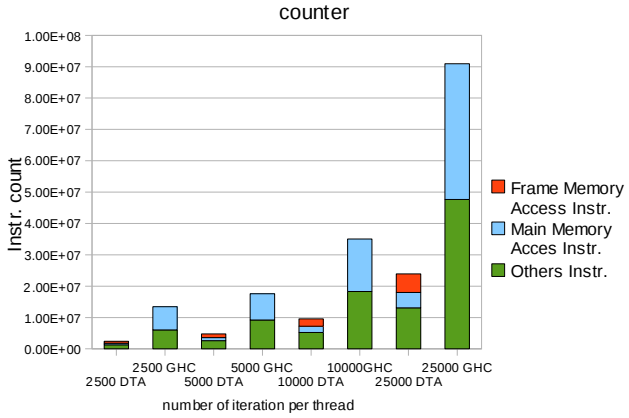


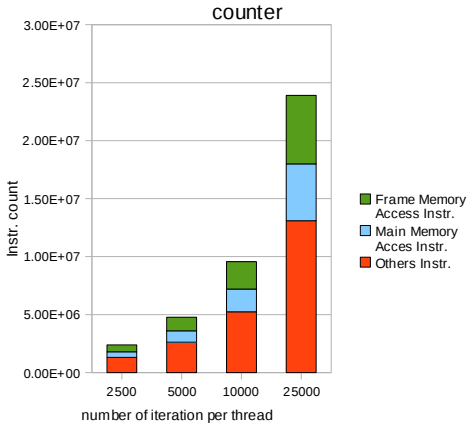
Figure 13: Max Memory Allocated at runtime (in KB) by the execution of counter example in the DTA simulated execution and the GHC compiled execution.

Third, we show the instruction generated by the our DTA programs by the GHC compiled program

ISBN: 978-1-61804-276-7



(a) Comparison between the GHC compilation and the DTA execution



(b) Instruction detail for the DTA compilation.

Figure 14: Dynamic instruction count generated by the various executions of counter (Figure 4).

(Figure 14). In this example can be used the same of the previous one. The comparison shows that both the examples use about half instructions to access the memory, but the DTA code uses only a part of those instruction to access the main memory. This is surely due to the simplicity of our Transactional Memory mechanism compared the the actual GHC TM mechanism and the whole run time system, but it seems a good starting point.

Those results show us that even a really simple implementation of a TM mechanism in DTA environment has a good performance. So, we think this project is very promising.

One important point those experiments display is that this program, differently from the previous one, didn't have a good parallel behavior, generating always no more than three or four parallel threads. This is important for showing the great difference between a program of pure functional calculation, like Fibonacci, and a program using only monad computation. The first has a good thread level parallelism. The second is mostly sequential, if not for the explicit forking of threads.

5 Conclusion

The main goal of this research is to find a way to translate the functional behavior of the Haskell language in the DTA architecture. Then, to realize a way to reproduce the Transactional Memory mechanism present in the GHC compiler into DTA.

The first part of the research involved the development of a simple tool to translate Haskell programs in DTA assembly. We made comparisons from code generated by this program with already existent DTA code generated from a general DTA version of the same program. Then, we compared the program generated by our tool with the same program compiled directly by GHC, and executed in an real machine. Both those experiments returned very good results, showing a promising behavior of the porting of this functional language in DTA.

In the second part, we developed a simple example of a Haskell program using Transactional Memory in DTA. We created a first simple example to make our experiments. This program, even in its simplicity, showed a good performance compared to the classical implementation.

Acknowledgments: This research is partly funded by the EU through projects HiPEAC (id. 287759), TERAFLUX (id. 249013) and AXIOM (id. 645496).

References:

[1] Philip Wadler, "The essence of functional programming", Annual Symposium on Principles of Programming Languages, Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 1992, pp. 1-14

[2] Nir Shavit, Dan Touitou, "Software transactional memory", Distributed Computing Volume 10, Number 2, February, 1997, pp. 99-116

[3] M. Herlihy and E. Moss, "Transactional Memory: Architectural Support for Lock-Free Data Structures", in 20th Annual International Symposium on Computer Architecture, May 1993, pp. 289-300.

[4] Roberto Giorgi, Zdravko Popovic, Nikola Puzovic, "DTA-C: A Decoupled multi-Threaded Architecture for CMP Systems", 19th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'07), 2007, pp. 263-270,

[5] K. M. Kavi, R. Giorgi, and J. Arul, "Scheduled Dataflow: Execution Paradigm, Architecture, and Performance Evaluation," IEEE Transaction on Computers, vol. 50, 2001, pp. 834-846.

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

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

[8] 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.

[9] R. Giorgi and A. Scionti. A scalable thread scheduling co-processor based on data-flow principles. *Future Generation Computer Systems*, Jan. 2015, pp.1–9.

[10] Solinas et al. M. 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.

[11] Graham Hutton, "Programming in Haskell", Cambridge University Press, January 2007.

[12] GHC Official Site, www.haskell.org/ghc/

[13] Hal Daume III, "Yet Another Haskell Tutorial", www.cs.utah.edu/ hal/docs/daume02yaht.pdf

[14] S. Peyton-Jones, A. Gordon, and S. Finne, "Concurrent Haskell", ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (PoPL), 1996, pp. 295308.

[15] 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, June 2005, pp. 48-60.

[16] C. Perfumo, N. Sonmez, O. S. Unsal, A. Cristal, M. Valero, and T. Harris. "Dissecting transactional executions in Haskell." In The Second ACM SIGPLAN Workshop on Transactional Computing (TRANSACT), 2007.

[17] Blundell, Colin and Lewis, E Christopher and Martin, Milo M. K., "Subtleties of Transactional Memory Atomicity Semantics", Computer Architecture Letters, Vol 5, Number 2, November 2006, pp. 17-17.

[18] Haskell library documentation, Software Transactional Memory page, http://www.haskell.org/ghc/docs/latest /html/libraries/stm/Control-Monad-STM.html

[19] Tim Harris, Adrin Cristal, Osman S. Unsal, Eduard Ayguade, Fabrizio Gagliardi, Burton Smith, Mateo Valero, "Transactional Memory: An Overview", IEEE Micro, vol. 27, no. 3, May/Jun, 2007, pp. 8-29.

[20] J, Larus , R. Rajwar, "Transactional Memory (Synthesis Lectures on Computer Architecture)", Morgan & Claypool Publishers, 2007.

[21] Simon Peyton Jones, Simon Marlow, "Secrets of the Glasgow Haskell Compiler inliner", Journal of Functional Programming, 12, Jul 2003, pp. 393-434.

[22] Simon Peyton Jones, "Compiling Haskell by program transformation a report from the trenches", Programming Languages and Systems ESOP '96, January 2006, pp. 18-44.

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

[24] GHC developer Wiki, SimpleCore page, http://hackage.haskell.org/trac/ghc/ wiki/Commentary/Compiler/CoreSynType

[25] Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, Kevin Donnelly. "System F with Type Equality Coercions", Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation, 2007, pp. 53-66.

[26] Eric Parsons' CPSC 510, Project http://pages.cpsc.ucalgary.ca/ parsonse/510/

[27] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, C. R. Moore, "Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture", 30th Annual International Symposium on Computer Architecture (ISCA'03), 2003, pp.422bitemrdtsc RDTSC library for Haskell GHC, http://uebb.cs.tu-berlin.de/ magr/projects/rdtsc/

[28] Valgrind Tool Suite, http://valgrind.org/info/tools.html