Int J Parallel Prog (2016) 44:257–277
DOI 10.1007/s10766-015-0369-2

Integrating Transactions into the Data-Driven Multi-threading Model Using the TFlux Platform

Andreas Diavastos[1] · Pedro Trancoso[1] · Mikel Luján[2] · Ian Watson[2]

Received: 28 March 2013 / Accepted: 8 June 2015 / Published online: 19 June 2015

Abstract The introduction of multi-core processors has renewed the interest in programming models which can efficiently exploit general purpose parallelism. *Data-Flow* is one such model which has demonstrated significant potential in the past. However, it is generally associated with functional styles of programming which do not deal well with shared mutable state. There have been a number of attempts to introduce state into Data-Flow models and functional languages but none have proved able to maintain the simplicity and efficiency of pure Data-Flow parallelism. Transactional memory is a concurrency control mechanism that simplifies sharing data when developing parallel applications while at the same time promises to deliver affordable performance. In this paper we report our experience of integrating Transactional Memory and Data-Flow within the TFlux Platform. The ability of the Data-Flow model to expose large amounts of parallelism is maintained while Transactional Memory provides simplified sharing of mutable data in those circumstances where it is important to the expression of the program. The isolation property of transactions ensures that the exploitation of Data-Flow parallelism is not compromised. In this study we extend the TFlux platform, a Data-Driven Multi-threading implementation, to support

✉ Andreas Diavastos
diavastos@cs.ucy.ac.cy

Pedro Trancoso
pedro@cs.ucy.ac.cy

Mikel Luján
mikel@cs.manchester.ac.uk

Ian Watson
watson@cs.manchester.ac.uk

1 Department of Computer Science, University of Cyprus, P.O. Box 20537, 1678 Nicosia, Cyprus

2 School of Computer Science, University of Manchester, Oxford Road, Manchester M13 9PL, UK

transactions. We achieve this by proposing new pragmas that allow the programmer to specify transactions. In addition we extend the runtime functionality by integrating a software transactional memory library with TFlux. To test the proposed system, we ported two applications that require transactional memory: Random Counter and Labyrinth an implementation of Lee's parallel routing algorithm. Our results show good opportunities for scaling when using the integration of the two models.

Keywords Programming models · Future multi-cores · Transactional Memory · Data-Flow model · Data-Driven Multi-threading

1 Introduction

As technology delivers higher integration of devices into processors, the multi-core design has become the de-facto standard for processor architecture. It promises to deliver high performance whilst maintaining an acceptable complexity and power budget. The trends show a continuous increase in the number of cores and it is expected that by 2020 processors will include 1000 s of cores [23]. This will lead to new challenges, one of them being the programmability of such large-scale systems. If their power is to be harnessed on the solution of a wide range of problems it will be necessary to develop new parallel programming models which are both efficient and easy to use.

There has recently been a re-surgence of interest in the Data-Flow model as a way to efficiently exploit large-scale parallelism. Even though the original implementations of Data-Flow were not efficient, more recent developments have overcome this [5,8,20]. However, the models are suited largely to the implementation of programming styles which are essentially purely functional. Indeed it is the absence of side effects in functional models which permits easy parallelisation. Unfortunately, there are many cases in real programs where the use of shared mutable state is either necessary for efficiency or is a fundamental part of the problem being solved. In these circumstances, functional approaches are unsuitable.

This limitation has long been recognised and there have been a number of attempts to integrate state into both Data-Flow models and functional languages. The early work on M-structures in Id [3] added implicit locking to data items to avoid the explicit manipulation of synchronisation. However, this merely hid the complexity of the synchronisation rather than removed it and, in many ways, made the writing of shared state programs more error prone. Functional languages such as SML [14] and F# [22] have introduced mutable variables in an attempt to extend their practicality. Unfortunately this state can rapidly destroy both the mathematical cleanliness of the language and the ability to exploit parallelism with Data-Flow like execution models. Haskell originally introduced state in a more disciplined way by the use of MONADS [13]. However, although this enables the isolation of state via the type system and hence preserves mathematical properties, the state manipulation is serialised and thus does not address the problems of writing parallel programs.

The Transactional Memory (TM) model facilitates sharing data in a manner which isolates individual sharers from the complexities of synchronisation. It was originally

proposed as a way of simplifying parallel programming in conventional languages but has been shown to provide a clean and simple way to add the sharing of state to a functional language [18]. Transactional Haskell uses the MONAD approach to allow the expression of explicit threads within Haskell programs by defining transactional variables which are manipulated serially within a thread but interact in parallel across threads. It has been shown that parallel state based programs can be specified while maintaining much of the purity of functional programming. It is our belief that a more general and more usable programming model can be produced by adding transactions to Data-Flow using more pragmatic programming approaches and avoiding the complexity of MONADS.

This paper reports our first experiences of integrating transactions into a thread based Data-Flow model. We use TFlux [21] as the Data Driven Multi-threading (DDM) implementation and TinySTM [17] for transactional support. This merged implementation will be referred to as DDM+TM, hereafter. We propose additional TFlux directives for defining transactional threads and variables. We show how it is possible to program applications with the combined model and present a preliminary performance evaluation study. We also analyse the overheads of the proposed model through an evaluation process using a synthetic application.

This paper is organized as follows, Sect. 2 discusses the related work. Section 3 introduces the two models of Data-Flow and TM used in this work. Section 4 describes the implementation we propose using the TFlux system and TinySTM model. In Sect. 5 we analyze the applications we implemented and used for our experimental evaluation. Section 6 describes the experimental setup while Sect. 7 describes the performance scaling of two applications developed using DDM+TM and our overhead analysis of the proposed model. In Sect. 8 we discuss the conclusions and future work.

2 Related Work

There has recently been renewed interest in the Data-Flow approach to computation that was pioneered in late 1970s and 1980s [4,9,12,16,27]. Those projects demonstrated that it was feasible to express sufficiently large amounts of parallelism that a significant problem was how to throttle and schedule it. Subsequent projects, for example TFlux [21] showed that a coarsening of the granularity (from instructions to tasks) could result in more controllable and more efficient parallel execution.

Numerical Linear Algebra (NLA) is one area where Data-Flow ideas have recently been adopted. This is apparent both in LAPACK and BLAS functionality (see, e.g. PLASMA project [24]) and for sparse matrices [11]. NLA constitutes one of the main kernels for scientific computing and their main next challenge is how to scale to petaflop-scale high-performance systems. The new generation of NLA algorithms are moving towards expressing parallelism but leaving the scheduling to the runtime trying to harness the available combination of resources (multi-cores, many-cores, clusters, GPUs). It has been demonstrated that a Data-Flow execution using Parallel Linear Algebra for Scalable Multi-core Architectures (PLASMA) can easily generate millions of tasks.

There is a clear relation between Data-Flow computations and parallelization of functional languages. Another highly prominent use of functional programming techniques can be observed in the MapReduce frameworks [6]. Provided the map and reduce operations are side-effect free, we can automatically parallelize their execution using a Data-Flow approach.

Although the benefits of Data-Flow can be demonstrated by the above applications, it is clear that there are cases where shared state is essential. This may be because it is a fundamental part of the program, it facilitates software development or a pure functional execution will be inefficient (e.g. due to memory management overheads). To overcome these limitations, it is desirable to introduce shared mutable state into the Data-Flow approach. However, this needs to be done in a way which does not require the specification of explicit synchronisation between parts of the program. This both introduces significant programming complexity and can often lead to unnecessary serialisation of the execution. Therefore approaches which introduce conventional locking are unlikely to lead to models which are either easy to use or efficient.

Transactional Memory (TM) [10] is a model for manipulating mutable shared data which attempts to reduce complexity by eliminating the need for explicit synchronisation. It works by allowing the programmer to specify that certain sections of a program must be executed atomically but without the need to consider any of the synchronised control that might be required. Execution of atomic sections takes place optimistically, that is with an assumption that any shared data within the section will not be changed by any other concurrent execution. If such a conflict does occur, the underlying runtime system ensures that only one execution succeeds while others are transparently re-executed. This leads to the important property of isolation. A thread always proceeds as though it has exclusive access to any shared data within an atomic section. All synchronisation complexity is removed and it is unnecessary to serialise accesses to achieve correct execution. Although, in practice, some serialisation may occur due to the resolution of conflicts, the optimistic nature of the model ensures that maximal parallelism is achieved. Although TM was originally proposed to reduce the complexity in the context of conventional threaded programming languages, the isolation property makes it an ideal way of introducing mutable state into Data-Flow or functional approaches.

A common example used for motivating TM illustrates the need for mutable shared state [18]. Consider a computation which is trying to perform concurrent credit/debits between bank accounts. Firstly, the state is fundamental to the problem. The account balances must be globally accessible variables which can be updated and persist. The credit/debit operations must be atomic to preserve the overall correctness of the balances. Assuming that we do not know the identity of the accounts when specifying the problem, it is clear that the accounts might overlap and conflicts could occur. A conventional locking approach would need to deal with the cases where overlap might occur by taking explicit locks and dealing with complex interactions such as deadlock. The problem could, of course, be greatly simplified by serialising all the operations but this would defeat the desire to exploit parallelism. However, in many cases, there will be no overlap and an optimistic approach can proceed with maximal parallelism. We can envisage a Data-Flow solution where threads have been generated to perform calculations on each account using a purely functional approach and then invoking a

transaction to perform a balance transfer. Any number of such threads can be generated to operate in parallel without any need to consider how they interact.

In the TERAFLUX project [23] we are investigating how to combine different variants of Data-Flow models (including synchronous Data-Flow) with Transactional memory. This paper reports our experiences with a first combination of the data-driven runtime system, TFlux, and software TM driven by the implementation of two applications.

3 Data-Flow and Transactional Memory

In this section we describe the two individual parallel programming models, emphasising on the strengths and weaknesses of each one separately. We also discuss the combination of the models as proposed in this work.

3.1 Data-Flow

Data-Flow is known to be the model that is able to exploit the most parallelism in an application. In the recent years it has been revisited as the solution for scaling the performance of applications. Data-Flow is a computation model that does not follow the classical program counter model but instead relies the execution of each operation on the availability of its input data. Each operation can be considered as a Data-Flow node and each node can be executed independently of all other nodes.

A Data-Flow program can be described using a directed acyclic graph (Data-Flow Graph), where each node represents an operation and every edge the data dependencies between nodes. Figure 1 shows a Data-Flow graph for a reduction operation. The first level of nodes can start execution immediately as there are no input dependencies, while the rest of the nodes must wait for their input to arrive from previous nodes as depicted by the directed edges of the graph.

Strengths The Data-Flow model has the ability to exploit the maximum available parallelism in an application while avoiding the high synchronization overheads and memory latencies of other parallel implementation (e.g. locks, barriers). Data-Flow is a side-effect free model as each node depends only upon its inputs and can be independently executed. A Data-Flow program can be easily implemented as a distributed

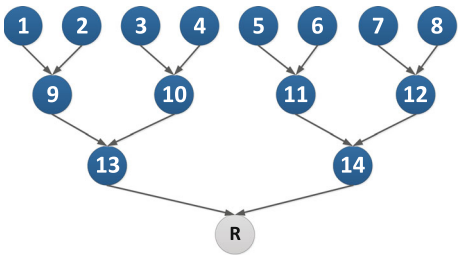

Fig. 1 A Data-Flow graph for calculating a reduction operation

program as the communication between nodes is explicit and done only at the end of each nodes execution.

Weaknesses The Data-Flow model lacks the ability of providing shared state in programs where this is a fundamental operation. To overcome this limitation it is important to be able to provide shared mutable state in a Data-Flow program. However, it is also essential that this is done without introducing explicit synchronization in the program as this will lead to unnecessary serialisation of the execution that will eventually eliminate the Data-Flow principles from the execution.

3.2 Transactional Memory

Originally proposed as a way of simplifying parallel programming in conventional languages, Transactional Memory [10] attempts to reduce complexity of manipulating mutable shared data by eliminating explicit synchronization. It allows the programmer to specify that certain sections of a program will be executed atomically, without the need of considering any synchronization control. Atomic sections are executed fully parallel and if a conflict occurs on shared data, the underlying runtime will ensure that only one transaction succeeds, while others are re-scheduled for execution. Using Transactional Memory, parallel state programs can be specified while maintaining much of the purity of functional programming.

Strengths The atomicity offered by TM will allow concurrent access to shared data structures, consequently allowing parallel execution of transactions that share these data. By monitoring these data structures the TM runtime system will detect any conflicts that may arise during the execution. In such a case the conflict transactions will abort and the runtime system will reschedule them.

Weaknesses The overheads imposed by existing software implementations of TM are significant. These overheads come from monitoring data structures, aborting and rescheduling conflicting transactions. Although TM is good at synchronizing and monitoring memory access to shared data, it does not offer any mechanism to prevent conflicts. These conflicts alone impose a large overhead as the runtime system will need to reschedule them, thus re-executing them all over again. Some of the overheads of a software TM implementation can be resolved when using a hardware TM system. In this work though, we focus on a software implementation as to keep it portable, as this is a first work of combining the two models and we wanted a platform that would allow us to be able to debug and experiment with the system as much as possible.

3.3 Data-Flow and Transactional Memory Combined

Our proposed combined model allows both approaches to be integrated in one system. Data-Flow is used to parallelise a program at the level possible, thus retaining the Data-Flow principles in a parallel application. Where shared state is needed, or to explore even more parallelism in the parallel application, the TM system is introduced.

Strengths Combining the two models results in several benefits. We manage to retain the strengths of both models and minimize some of their weaknesses. Introducing mutable state we improve the Data-Flow model by allowing data structures to be easily shared among Data-Flow threads, without having to explicitly merge them as you would in a purely Data-Flow program. We also improve the TM model by reducing the number of conflicts. By using both Data-Flow threads and transactions in the same program we manage to reduce the total number of transactions a parallel program will have. Eventually this means that the possibility of transactions to conflict will also be reduced. Overall, this will reduce the overhead produced by a software TM implementation. The Data-Flow model cannot explore parallelism in sections that share common data structures. Integrating transactions into the Data-Flow model we add the ability of handling shared state, thus increasing the level of parallelism that can be explored. Data-Flow threads are stateless and can be re-executed since they have no side effects. This reduces the work that has to be done by a TM system upon recovery in a conflict situation, by simply re-executing the conflicting Data-Flow threads.

Weaknesses On the other hand, the combination of the two models has some weaknesses. To provide mutable shared state on data structures for the Data-Flow model we must monitor the shared data from the TM runtime system. This will provide atomicity, isolation and consistency for the shared data and the execution itself. To do so we need to indicate those data structures that are to be monitored. In the proposed model this task is left to programmer. To ease the programmers effort though we offer a more efficient way to express these dependencies on the shared data structures between threads as described in Sect. 4.

4 Proposed DDM+TM Implementation

4.1 TFlux System

There has recently been a resurgence of interest in the Data-Flow model as a way to efficiently exploit large-scale parallelism. Nevertheless, the original Data-Flow implementations suffered from serious limitations as it required specialized hardware [9]. New hardware designs though, make the Data-Flow model a possible solution for scaling the performance of applications. Consequently, researchers have proposed new alternatives based on this model that are able to exploit the parallelism effectively. DDM is one such alternative, that can exploit the maximum available parallelism of an application by exploiting data dependencies, while at the same time overcomes overheads by increasing the granularity of the execution blocks [5,20]. Each execution block (i.e. thread) in DDM is scheduled in a Data-Flow way at runtime, but the instructions inside a thread are executed in a control-flow way. In some DDM implementations, the scheduling unit that handles the synchronization of the execution is implemented in software, thus guarantees portability of the implementation to any new hardware design.

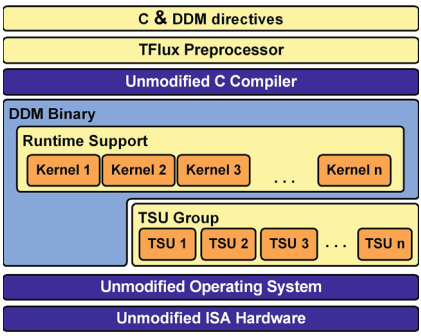

Fig. 2 The layered design of the TFlux system [21]

This study uses the Data-Driven Multi-threading model and in particular its TFlux implementation [21]. We chose TFlux as it is a complete platform that includes a programming environment for DDM applications using compiler directives, a source-to-source preprocessor that will translate the application augmented with the directives into DDM parallel code and a Thread Scheduling Unit (TSU) that handles the Data-Flow scheduling of the threads at runtime. In Fig. 2 we show the different modules of TFlux. An important advantage of TFlux is that it is not build for a specific machine but rather works as a virtualization platform for DDM program execution on a variety of computing systems. Another reason we chose the TFlux implementation is that the produced parallel code from the platform is in ANSI-C which complies with the supported programming languages of most systems.

In order to program a DDM application with TFlux, directives must be added to regular C code. The most relevant directives are the ones which enable a set of instructions to be defined as a thread (see Table 1). In addition it is necessary to define the inputs and outputs of a thread as well as the producer and consumer relationships between threads. Using this information the system is able to form the code for the threads as well as the thread dependency graph, which is the structure that needs to be loaded into the TSU for the scheduling to be executed in a Data-Flow manner. The task of the scheduling unit is to manage the counters that control the firing of threads. Each time a producer thread terminates its execution, the consumer's counter is decremented by one. When the counter reaches zero, all needed results have been produced and thus the thread is ready for execution. All these operations are part of the TFlux runtime system, which includes all data structures required to manage the thread scheduling as well as the scheduling code itself. The operations of the runtime system are depicted in Fig. 3. There are hardware and software versions of the TFlux platform. In this study we use the software version, also called TFluxSoft, without losing generality of the system. In TFluxSoft the TSU's execution is handled by one core of the multi-core system we are using.

4.2 TinySTM Software Library

In order to support the transactional execution, *i.e.* the monitoring of the updates to variables, the conflict detection and the restarting of the execution in case of abort, we

Table 1 TFlux DDM pragma directives

`#pragma ddm startprogram`	Define the start and the end of a DDM program
`#pragma ddm endprogram`	
`#pragma ddm block` *ID*	Define the start and the end of a block of threads with identifier *ID*
`#pragma ddm endblock`	
`#pragma ddm thread` *ID* `kernel` *NUMBER*	Define the boundaries of a DDM thread with identifier *ID* and the kernel *NUMBER*
`#pragma ddm endthread`	
`#pragma ddm for thread` *ID*	Define the boundaries of a DDM loop thread with identifier *ID*
`#pragma ddm endfor`	
`#pragma ddm kernel` *NUMBER*	Declare the number of kernels to be used
`#pragma ddm var` *TYPE NAME*	Declare a shared variable with *NAME* and *TYPE*
`#pragma ddm private var` *TYPE NAME*	Declare a private variable with *NAME* and *TYPE*

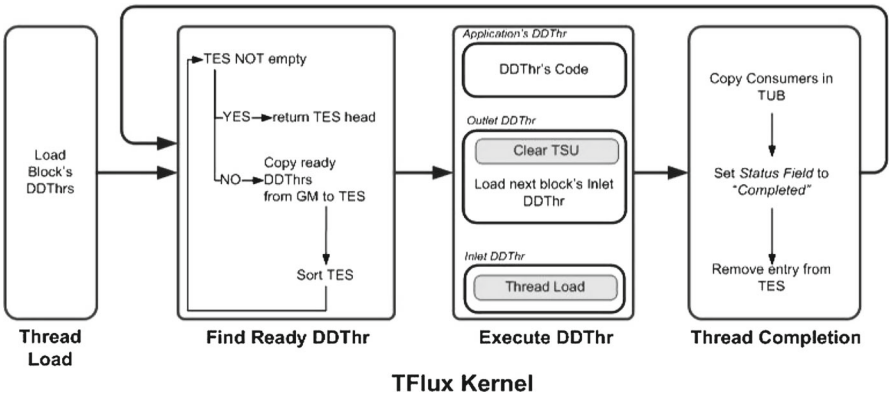

Fig. 3 TFlux Runtime system

need to extend the TFlux runtime. Rather than developing from scratch we preferred to extend the TFlux runtime system with an existing software TM implementation. We chose the TinySTM [17] as it appeared to provide a simple approach to the integration. However, we could have used other Software TM systems such as TL2 [7] or RSTM [19]. Within TERAFLUX, we intend to investigate other TM implementations, including hardware support, for the scalability of the execution required for large-scale Data-Flow applications.

4.3 DDM+TM

A program in DDM+TM consists of both Data-Flow threads and transactions. Each of the two models though has it's own runtime system for scheduling either threads

or transactions, thus when combining the two we had to define responsibilities for each runtime. In this work, we kept the two runtime systems independent from one another but we enforced a hierarchical structure for the execution of Data-Flow threads and transactions. The TFlux runtime system (the TSU) runs on top of the TM runtime library and is responsible only for scheduling Data-Flow threads, while the TM runtime is only responsible for aborting/committing transactions. The scheduling of a Data-Flow thread may impose the start of a transaction but the TSU will not interfere with the monitoring of the transactions. When a transaction starts execution the TM runtime will take over and monitor for conflicts. If such a conflict occurs, the TM runtime will reschedule the transaction without the TSU noticing any changes. As soon as a transaction commits and the Data-Flow thread is finished, the TSU will take over again and schedule the next ready thread.

In this first attempt at adding TM to TFlux, we have opted not to make changes in the TSU to support transactional behavior. However, we are investigating the possibility of offloading the re-scheduling of an aborted transactional thread to the TSU instead of the TinySTM system as previous work on TM [1,2] has shown that controlling the scheduling using information about transactions can improve the performance and reduce wasted work due to aborts.

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

The responsibility of finding the dependencies in a Data-Flow program and the variables to be monitored in a TM program fall upon the programmer. In DDM+TM the programmer must decide for both the dependencies of Data-Flow threads and the variables to be monitored within transactions. To ease the effort of the programmer in developing DDM+TM programs we introduce new pragma directives that allow the declaration of Data-Flow threads along with their dependencies and the declaration of transactions with the variables to be monitored. These new TFlux directives are presented in Table 2.

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

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

Table 2 TFlux DDM+TM pragma directives

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

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

For DDM+TM, we decided to have a complete separation between transactional and non-transactional variables. Transactional variables must always be accessed within a transaction. Non-transactional variables are normally private to a thread during execution and thus cannot generate conflicts. Other non-transactional threads will only be allowed to access a non-transactional variable if the scheduling can guarantee independence. With this decision we avoid the problems of weak isolation. Note that we do not modify DDM by imposing this decision. DDM+TM could be implemented without speculation by performing a scheduling where the transactional variables are treated as inputs and outputs of the threads that are read and written. This will result in the sequential execution of the code in case of threads accessing shared data where we cannot determine the dependencies before runtime.

For transactional variables, we also provide the programmer with pragmas to define them within the declaration of a transactional thread. This is required to support the monitoring of variables that may have more than one alias (e.g. parameter variables inside the code of a function). The monitoring of these variables is specified as a parameter in the thread declaration (see Table 2).

As TFlux has two pragmas for declaring threads, the table contains
`#pragma ddm atomic thread ID` and
`#pragma ddm atomic for thread ID`
declaring a transactional thread and a transactional loop thread, respectively. The `tvar(NAME : READ/WRITE)` extension defines the thread variables that are transactional.

Table 3 TFlux DDM+TM pragma directives correspondence with TinySTM runtime calls

DDM+TM Prama Directives	TinySTM runtime support
`#pragma ddm atomic thread ID tvar (NAME : READ/WRITE/READ_WRITE)`	`stm_init_thread()`
`#pragma ddm atomic endthread`	`stm_exit_thread()`
`#pragma ddm atomic for thread ID tvar(NAME : READ/WRITE/READ_WRITE)`	`stm_init_thread()`
`#pragma ddm atomic endfor`	`stm_exit_thread()`
`#pragma ddm atomic transaction tvar(NAME : READ/WRITE/READ_WRITE)`	`sigjmp_buf *_e = stm_start (&_a) if (_e != NULL) sigsetjmp(*_e, 0)`
`#pragma ddm atomic endtransaction`	`stm_commit()`
`#pragma ddm atomic tvar(NAME : READ/WRITE/READ_WRITE)`	`int temp = (int) stm_load ((stm_word_t *)&NAME) OR stm_store((stm_word_t *)& NAME, (stm_word_t *)temp)`
`#pragma ddm atomic abort`	`stm_abort()`

The last proposed directive

```
#pragma ddm atomic transaction
```

allows the declaration of a transaction as a portion of a thread. For certain applications such as those considered in this work, this offers better performance (see Sect. 7).

These directives are a subset of the possible ones which have been defined for TM [10]. However, they are enough to implement the applications described in this paper and we consider them to be the core directives. Extra transactional functionality can be added by declaring

```
#pragma ddm atomic abort
```

in the case the programmer wants to manually abort a transaction. In Table 3 we show the correspondence of the proposed pragmas with the calls to the TinySTM runtime that will automatically be generated by the preprocessor.

5 Workloads

For this proposal of transaction integration with the Data-Flow model we have tested two applications. The Random Counters and the Labyrinth implementation of Lee's algorithm from the STAMP Benchmark suite [15]. These were originally implemented in a conventional language using TM and are not naturally Data-Flow applications. However, for this study we are focussing mainly on the functionalities and overheads of the implementation rather than the added benefits of the integration such as exploiting implicit parallelism.

5.1 Random Counters

In the Random Counters application we use multiple threads to increment the values of random array positions. We create and initialize all elements of the array to zero.

```
#pragma ddm atomic for thread 1
        tvar(counters : READ_WRITE)
for(cv00 = 0; cv00 < THREADS; cv00++)
 {
  for (j = 0; j < numOfCounters; j++)
        indexTM[j] = rand() % arraySize;
  for (j = 0; j < numOfCounters; j++)
      counters[indexTM[j]]++;
 }
#pragma ddm atomic endfor
```

Fig. 4 Random Counters implementation with TFlux DDM+TM pragma directives

We then spawn multiple threads to execute loop thread code. Each loop iteration is executed in parallel and each thread will do the same work: generate a random sequence of numbers—that represent index positions in the array and increment by 1 the values located in those positions. In the code shown in Fig. 4 we present the implementation of this algorithm, where the

`#pragma ddm atomic for` and

`#pragma ddm atomic endfor`

directives define the DDM+TM thread boundaries, showing that all the code of the thread is considered to be transactional. The purpose of this algorithm is to create concurrent accesses to memory. This will in turn create memory conflicts between threads trying to access the same memory locations. In an unsynchronised parallel implementation this execution would create a false result.

In the DDM model this code would have to be executed sequentially since the dependencies cannot be defined prior to the execution (due to the random memory accesses). By using TM we protect the values of the shared variables and ensure that, at the end of the execution, we will get the correct result while running the threads in parallel.

5.2 Labyrinth Implementation of Lee's Algorithm

Lee's Algorithm is used in the process of producing an automated interconnection of electronic components. It guarantees to find a shortest interconnection between two points using the Expansion-Backtracking technique shown in Fig. 5.

Starting from the source point *S*, the grid points are numbered by expanding a wavefront until the destination is reached (Fig. 5a–e). At each phase during the expansion stage each grid point in the wavefront marks its unnumbered neighbors with an increment of its value. Once the destination *D* is reached, a route is traced back to the source by following any decreasing sequence of numbered grid points that are not used by others. Once the shortest route has been determined, the grid points reserved for this route cannot be used by others [26].

For the purpose of this work we used the Labyrinth TM implementation of Lee's algorithm from STAMP [15] as a guideline and ported it to DDM+TM. In Fig. 6 we present the integration of TFlux DDM+TM pragmas into the application. Using

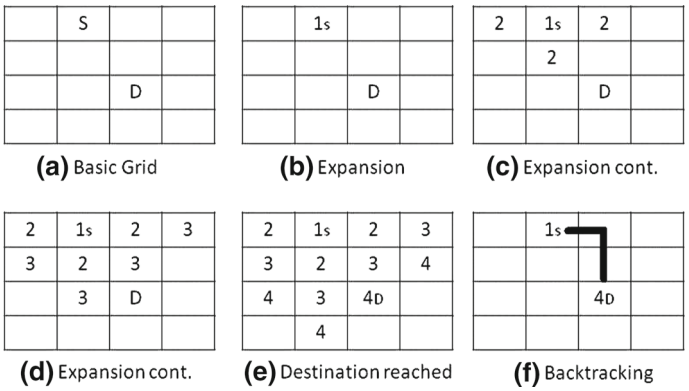

Fig. 5 Lee's algorithm example

```
#pragma ddm atomic for thread 1
for(cv00 = 0; cv00 < THREADS; cv00++)
{
#pragma ddm atomic transaction
        tvar(queue : READ_WRITE)
    [1] Get a (S, D) pair from the
         work queue
#pragma ddm atomic endtransaction

#pragma ddm atomic transaction
        tvar(global_grid : READ)
    [2] Copy global grid to threads
        local grids
#pragma ddm atomic endtransaction

    [3] Expansion stage
    [4] Backtracking stage

#pragma ddm atomic transaction
        tvar(global_grid : WRITE)
    [5] Write selected route back to
         global grid
#pragma ddm atomic endtransaction
}
#pragma ddm atomic endfor
```

Fig. 6 Lee's Algorithm pseudo-code with TFlux DDM+TM pragma directives

`#pragma ddm atomic for` and
`#pragma ddm atomic endfor`
directives we declare the boundaries of a DDM+TM thread. In step 1 each thread will take a *(S, D)* pair from the global work queue as an atomic action. In step 2 each thread will copy the global grid to its local memory space, and in steps 3–4 each thread will execute the algorithm's expansion and backtracking phase locally. Finally in step 5 each thread will write the selected route back to the global grid as an atomic action. In order for steps 1, 2 and 5 to be executed correctly we must ensure that no data conflicts

occur on the work queue or global grid. Declaring steps 1, 2 and 5 as transactional code with the

`#pragma ddm atomic transaction` and

`#pragma ddm atomic end transaction`

we provide an atomic execution for those steps ensuring that the final result will be correct. As for steps 3 and 4 the execution is done locally using only data local to each thread. This does not require this part of the code to be transactional and lets it execute in parallel within the boundaries of the thread, thus avoiding the need to re-execute non-transactional code in the event of a conflict. This is therefore an example where not all the code of a thread is transactional and hence need not be declared with the transactional version of the pragma.

6 Experimental Setup

We have evaluated the integration of TM with the Data-Flow model using the Random Counters and Lee's algorithm described in Sect. 5. For Random Counters we used a conventional multi-threaded TM implementation as a baseline for our implementations of the DDM+TM version of the application. In this preliminary study we are interested in the overheads of the DDM+TM model over the TM implementation rather than detailed performance. For Lee's algorithm we used the Labyrinth implementation from [15] that uses TM calls from TinySTM and integrated the TFlux directives to create the DDM+TM version. The results are for the parallel DDM+TM implementation of the applications on a 12-core machine with 2 6-core AMD Opteron(tm) Processors 2427 running at 2200 MHz. The available memory of the system is 31 GB of main memory, 512 KB of L2 and 6144 KB of L3 cache. The system is running an Ubuntu SMP x86-64 operating system.

The porting of the applications for the Data-Flow model was performed using the pragma directives from [25] of the TFlux Soft system [21] version 1.5.0. To integrate the transactions in the applications we used the TinySTM library. The execution time measurements were collected using the *gettimeofday* system call to measure the execution time of the section of code that has been parallelized. The input data sizes used for each application are depicted in Table 4. All the results collected are for the Data-Flow model implementing transactions over the baseline execution. The baseline execution was considered to be the sequential execution of the applications implementing transactions with the TinySTM library. To calculate the results while avoiding any statistical errors we used the average of 10 executions removing the largest and the smallest execution time.

Table 4 Experimental workloads problem sizes

Benchmark	Problem size		
Random counters	10 Updates	100 Updates	1000 Updates
Labyrinth	$256 \times 256 \times 3 - n256$	$256 \times 256 \times 5 - n256$	$512 \times 512 \times 7 - n512$

7 Experimental Results

7.1 DDM+TM Applications Evaluation

In Table 5 we present the statistics of the Random Counter application both for TM and DDM+TM implementations. It shows the number of commits for executions with a different number of threads and the average number of aborts for 10 executions. For each execution the statistics are different due to the use of *random* in the application. Since each execution of the application is random the statistics will follow a uniform random distribution. Consequently, the results demonstrate that the integration of TM with DDM behaves similarly to a normal TM implementation but cannot be used to extract conclusions about detailed performance.

In Fig. 7 we present the difference in the execution time of the Labyrinth benchmark when reducing the size of the transactional code inside a thread. From now on we reference the execution of a whole transactional thread as *large* and the execution of a part of the thread as transactional as *small*. We observe that on average the execution time is smaller when we declare only a part of the thread as transactional. As we increase the number of threads we see this difference decreasing. Correlating these results with Table 6, where we present the numerical results of the previous executions, we see that when the number of aborts is the same for the two executions the *small* implementation is faster and when the number of aborts for the *small* implementation exceeds the number of aborts of the *large* implementation the execution times are

Table 5 Random counter statistics for TM and DDM+TM implementations

Number of threads	Commits	Aborts	
		TM	DDM+TM
2	200	38	47
4	400	157	479
8	800	920	1283

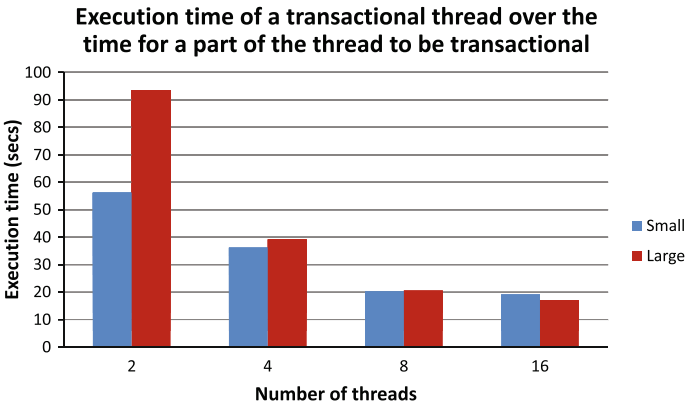

Fig. 7 Comparison of complete and partially transactional threads in the Labyrinth implementation

Table 6 Labyrinth statistics for TM and DDM+TM implementations

Number of threads	Commits	Aborts	
		Small	Large
2	1028	18	18
4	1032	47	46
8	1040	101	99
16	1056	182	175

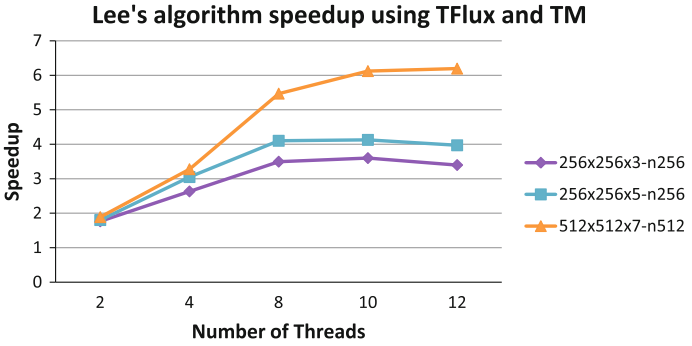

Fig. 8 Labyrinth's TM routing algorithm—scalability of TFlux extended with TM

almost the same. From this correlation we conclude that the overhead of rescheduling a whole transactional thread is higher than rescheduling a small part of the thread. This is because the rescheduled code to be executed will be more in the former case.

Finally we present the experimental results that indicate the speedup for the DDM+TM implementation of the Labyrinth application over the TM sequential implementation; these are depicted in Fig. 8. From these results it is easy to observe that for the two smaller input files ($256 \times 256 \times 3 - \text{n256}$ and $256 \times 256 \times 5 - \text{n256}$) the application scales well up to 10 threads and then starts degrading. For the largest input file it scales beyond 10 threads with a maximum speedup of 6.2× over the sequential implementation with TM at 12 threads. The reason for this degradation in the speedup for more than 10 threads is that although we use a 12-core machine we also have the TSU and the OS simultaneously running with the application and occupying one core each [21]. As seen from Fig. 8 the scalability of the DDM+TM model is good as it seems that at this early stage of the study there are no obvious extra overheads.

7.2 DDM+TM Overheads Analysis

To get a clear view of the integration of transactions into the Data-Flow model we created simple scenarios where we test the TM runtime system and record the overheads produced by monitoring shared mutable data, as well as the overheads of aborting and rescheduling conflicting transactions. We created synthetic applications with multiple threads that interact with shared data structures in order to record the overheads produced.

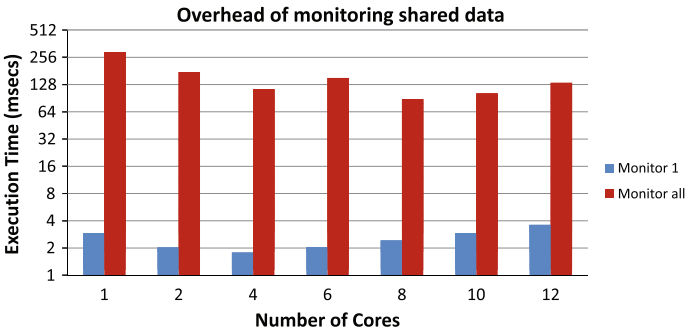

Fig. 9 Overheads of monitoring one shared variable versus monitoring all shared variables of the parallel application

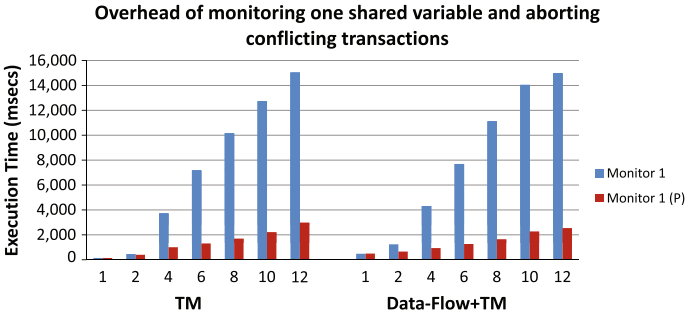

Fig. 10 Overheads of aborting conflicting transactions while monitoring one shared variable when a whole Data-Flow thread is declared as a transaction versus a part of a Data-Flow thread to be declared as a transaction

In the first scenario we executed a parallel application that uses shared data but will not create any conflicts during the execution. The data of this application are shared only for read operations, so no conflicts that require aborting of parallel transactions will arise. The purpose of this scenario is to record the overheads when monitoring only one such shared data structure, as opposed to monitoring all the shared data structures of the application. Monitoring a large number of shared data structures not only requires more hardware resources (e.g. memory) but, as shown in Fig. 9, it also results in a large overhead in execution time. The more data structures we monitor, the more work a TM runtime system will have to do during the execution. Although there are no conflicts during the execution the runtime system will still consume resources (mainly CPU time) in order to monitor the execution and check for possible conflicting transactions.

In another scenario we tested the granularity of a transaction relative to a Data-Flow thread. Figure 10 shows the overhead produced when monitoring one shared variable for two granularity levels. In the first case, we consider that a whole Data-Flow thread will be transaction. This means that in case of a conflict the whole Data-Flow thread will be aborted and re-executed. In the second case, we declare as a transaction only the portion of the Data-Flow thread that actually uses the shared data structure for a write

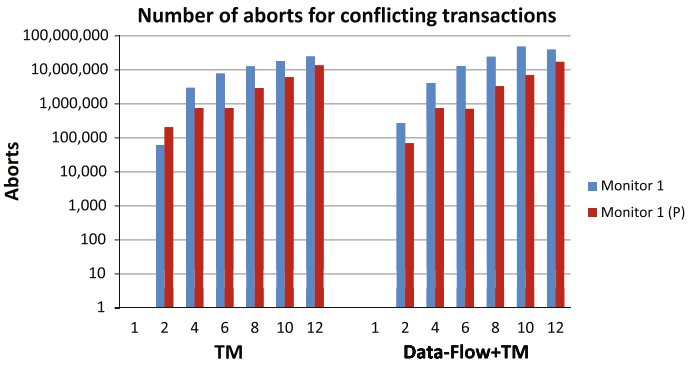

Fig. 11 Number of aborts of conflicting transactions while monitoring one shared variable when a whole Data-Flow thread is declared as a transaction versus a part of a Data-Flow thread to be declared as a transaction

operation. This is the only part of the Data-Flow thread that a conflict may arise. In this case, if we have a conflict during the execution only that portion of the thread will be aborted and rescheduled for execution, retaining any operations done previously in the Data-Flow thread. This avoids re-executing any operations done previously and did not affect the shared data monitored in any way. The results in Fig. 10 show significant reduction of the overhead imposed when rescheduling only the portion of the thread that uses the shared data, instead of rescheduling the whole Data-Flow thread.

What actually happens when we reduce the granularity of a transaction is that we reduce the scope of a conflict detection by the TM runtime system. This means that as soon as an operation on shared data structures is executed the transaction will commit, allowing other transactions to execute correctly and without conflicting. The number of aborts for these two test cases are presented in Fig. 11. The results clearly state that when reducing the scope of a conflict detection (i.e. the granularity of a transaction) the number of conflicts and consequently the number of aborted transaction will reduce significantly.

In the previous two figures we presented results for both the TM implementation and the proposed implementation of combining the Data-Flow model with transactions. The small variance in the results show an instability of the execution that is affected mainly by the TM runtime system. The nature of a TM implementation is that every execution is different from any other. The important thing that should be taken from this is that adding Data-Flow to a software TM implementation does not add any significant extra overhead to the performance of the TM system used. This means that by combining the two models we do not necessarily combine the overheads of the two models to the new proposed model.

8 Conclusions

We have reported our lessons from integrating, and performance experiments of, Data Driven Multi-threading (DDM) with Transactional Memory (TM). DDM offers the

benefit of discouraging sharing of mutable data while expressing large amounts of parallelism. However, it is not always possible to avoid sharing mutable state and that reduces parallelism in the DDM model. By adding TM to DDM, we allow further parallelism to be exploited while introducing mutable shared state in a composable manner.

We have extended the TFlux directives to develop Data-Flow applications with new pragmas for TM. We have also described the extended TFlux runtime system with functionality provided by a software TM. We have tested the new runtime system by developing two applications that require TM: Random Counter and an implementation of Lee's parallel routing algorithm.

The experiments and our reported experience indicates that, from the runtime integration point of view, existing software implementations can be integrated without major problems. The biggest interaction and point for further exploration comes with the scheduling. When a complete DDM thread is a single transaction, the TSU could take ownership of the restart mechanism and reschedule the thread. In future work we will address this and other potential optimizations as well as explore further parallelism in Data-Flow applications by using transactions.

We also plan on testing the overhead analysis on a hardware TM system as we recognize that many of the overheads imposed by the software TM implementation may be resolved when using a hardware TM system. The porting of our implementation to a hardware TM system is a trivial task as we can directly replace the software TM instructions with hardware TM instructions within the preprocessor, thus code for a hardware TM system would be generated automatically.

Acknowledgments The authors would like to thank HiPEAC Network of Excellence for the collaboration Grant and TERAFLUX project founded by the EC with Grant Agreement Number 249013. Dr. Luján is supported by a Royal Society University Research Fellowship.

References

1. Ansari, M., Kotselidis, C., Jarvis, K., Luján, M., Kirkham, C.C., Watson, I.: Advanced concurrency control for transactional memory using transaction commit rate. In: Euro-Par, pp. 719–728 (2008)
2. Ansari, M., Luján, M., Kotselidis, C., Jarvis, K., Kirkham, C.C., Watson, I.: Steal-on-abort: improving transactional memory performance through dynamic transaction reordering. In: HiPEAC, pp. 4–18 (2009)
3. Barth, P.S., Nikhil, R.S., Arvind: M-structures: Extending a parallel, non-strict, functional language with state. In: Proceedings of the 5th ACM conference on functional programming languages and computer architecture, pp. 538–568. Springer, London, UK (1991). http://dl.acm.org/citation.cfm?id=645420.652538
4. Cann, D.: Retire Fortran? A debate rekindled. Commun. ACM **35**, 81–89 (1992)
5. Costas, K., Evripidou, P., Trancoso, P.: Data-driven multithreading using conventional microprocessors. IEEE Trans. Parallel Distrib. Syst. **17**, 1176–1188 (2006)
6. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. In: Proceedings of the 6th Conference on Symposium on Opearting Systems Design and Implementation (2004)
7. Dice, D., Shalev, O., Shavit, N.: Transactional locking II. In: Proceedings of the 20th International Symposium on Distributed Computing (DISC) (2006)
8. Giorgi, R., Popovic, Z., Puzovic, N.: Dta-c: A decoupled multi-threaded architecture for CMP systems. In: 19th International Symposium on Computer Architecture and High Performance Computing, 2007. SBAC-PAD 2007, pp. 263–270 (2007)

9. Gurd, J.R., Kirkham, C.C., Watson, I.: The manchester prototype dataflow computer. Commun. ACM **28**, 34–52 (1985)
10. Harris, T., Larus, J., Rajwar, R.: Transactional memory. Synth. Lect. Comput. Archit. **5**(1), 1–263 (2010)
11. Hogg, J.D., Reid, J.K., Scott, J.A.: Design of a multicore sparse cholesky factorization using dags. SIAM J. Sci. Comput. **32**(6), 3627–3649 (2010)
12. Johnston, W.M., Hanna, J.R.P., Millar, R.J.: Advances in dataflow programming languages. ACM Comput. Surv. **36**, 1–34 (2004)
13. Jones, S.P., Gordon, A., Finne, S.: Concurrent haskell. In: Annual symposium on principles of programming languages, pp. 295–308. ACM (1996)
14. Milner, R., Tofte, M., Harper, R.: The Definition of Standard ML. MIT Press, Cambridge (1990)
15. Minh, C.C., Chung, J., Kozyrakis, C., Olukotun, K.: Stamp: Stanford transactional applications for multi-processing. In IISWC'08: Proceedings of The IEEE International Symposium on Workload Characterization (2008)
16. Papadopoulos, G.M., Culler, D.E.: Monsoon: an explicit token-store architecture. In: Proceedings of the 17th Annual International Symposium on Computer Architecture, ISCA'90, pp. 82–91 (1990)
17. Pascal, F., Christof, F., Torvald, R.: Dynamic performance tuning of word-based software transactional memory. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP) (2008)
18. Peyton Jones, S.L.: Beautiful Concurrency (2007). http://research.microsoft.com/Users/simonpj/papers/stm/index.htm
19. Sreeram, J., Cledat, R., Kumar, T., Pande, S.: Rstm : A relaxed consistency software transactional memory for multicores. In: International Conference on Parallel Architectures and Compilation Techniques (2007). http://doi.ieeecomputersociety.org/10.1109/PACT.2007.62
20. Stavrou, K., Evripidou, P., Trancoso, P.: DDM-CMP: data-driven multithreading on a chip multiprocessor. Embed. Comput. Syst. Archit. Model. Simul. **3553**, 205–224 (2005)
21. Stavrou, K., Nikolaides, M., Pavlou, D., Arandi, S., Evripidou, P., Trancoso, P.: Tflux: a portable platform for data-driven multithreading on commodity multicore systems. In: 37th International Conference on Parallel Processing, pp. 25–34 (2008)
22. Syme, D., Granicz, A., Cisternino, A.: Expert F# (Expert's Voice in.Net)
23. Giorgi, R., Badia, R.M., Bodin, F., Cohen, A., Evripidou, P., Faraboschi, P., et al.: TERAFLUX: Harnessing dataflow in next generation teradevices. Microprocess. Microsy. **38**(8),976–990 (2014)
24. Tomov, S., Nath, R., Ltaief, H., Dongarra, J.: Dense linear algebra solvers for multicore with GPU accelerators. In: International Symposium on Parallel and Distributed Processing (2010)
25. Trancoso, P., Stavrou, K., Evripidou, P.: DDMCPP: the data-driven multithreading C pre-processor. In Proceedings of the 11th Interact-11, pp. 32–39 (2007)
26. Watson, I., Kirkham, C., Luján, M.: A study of a transactional parallel routing algorithm. In: PACT'07 Proceedings of the 16th international conference on parallel architecture and compilation techniques, pp. 388–398 (2007)
27. Watson, I., Woods, V., Watson, P., Banach, R., Greenberg, M., Sargeant, J.: Flagship: a parallel architecture for declarative programming. In: ISCA, pp. 124–130 (1988)