Integrating Dataflow Abstractions into Transactional Memory

Vladimir Gajinov
Barcelona Supercomputing Center
Universitat Politecnica da Catalunya
vladimir.gajinov@bsc.es

Milos Milovanovic[†]
Microsoft Development Center Serbia
milmil@microsoft.com

Osman Unsal
Barcelona Supercomputing Center
osman.unsal@bsc.es

Adrian Cristal
IIIA - Artificial Intelligence Research Institute
CSIC - Spanish National Research Council
adrian.cristal@bsc.es

Eduard Ayguade
Barcelona Supercomputing Center
Universitat Politecnica da Catalunya
eduard.ayguade@bsc.es

Mateo Valero
Barcelona Supercomputing Center
Universitat Politecnica da Catalunya
mateo.valero@bsc.es

ABSTRACT

Many concurrent programs require some form of conditional synchronization to coordinate the execution of different program tasks. Programming these algorithms using transactional memory (TM) often results in a high conflict rate between transactions. In this paper we propose an Atomic dataflow model - ADF, which aims to reduce transaction conflicts by incorporating dataflow scheduling principles into transactional memory. The ADF model is based on the execution of atomic units of work called ADF tasks. A programmer explicitly defines data dependencies for the ADF task using the trigger set extension. Trigger set data is implicitly tracked by the TM runtime system, which detects changes and enables the re-execution of a transaction when its dependencies are satisfied. In this paper we fully describe the ADF model, present its syntax and show advantages of the model on a practical example.

Categories and Subject Descriptors
D.1.3 [**Concurrent Programming**]: Parallel Programming.

General Terms
Design, Languages.

Keywords
Transactional Memory, Dataflow, OpenMP, Parallelization.

1. INTRODUCTION

The trend towards incorporating more cores in Chip Multiprocessors (CMP) continues, with the potential of reaching hundreds of cores in future chip generations. However, factors like inefficient data access ordering and unmanageable synchronization primitives, such as locks, already limit application performance and programmer's productivity. Transactional Memory (TM) [5,9,15,6] is a synchronization mechanism that aims to solve these problems by abstracting away the complexities associated with concurrent access to shared data. Each thread optimistically executes transactional code in isolation and relies on the TM runtime system to detect conflicting memory changes. If no conflicts were detected during its execution, a transaction can safely commit its tentative changes to the memory. Otherwise, it has to roll back and re-execute from the beginning until it succeeds.

In order to provide blocking synchronization for transactional memory, Harris et.al [7] proposed the retry mechanism [8,12,17,3]. A transaction can retry at any point upon which its execution is blocked until some other transaction commits and invalidates its read set (a set of data read by the transaction before retry). The intention is to avoid useless re-execution of a transaction which is bound to retry if the data used to evaluate the retry condition has not changed. While the benefits of the retry mechanism are evident, the read set size of the retried transaction can undermine the system performance. Moreover, if the data that has triggered the re-execution of a transaction doesn't change the retry condition, the transaction will again retry and block. Hence, the idea to extend the retry mechanism to watch only a subset of data from a read set seems sound. This was first recognized by McDonald et al. [12] in the form of a watch-set. A transaction can add a variable to a watch set any time before retry. The intended use for a watch set is to enable conditional synchronization for transactional memory. We argue that this can be further extended to support expressiveness of the dataflow programming.

In dataflow programming [1,2,4,10,11,16,18] a program is represented as a directed graph consisting of named nodes, which represent units of work, and arcs, which represent data dependencies between nodes. Data values propagate along the arcs in the form of data packets, called tokens. Beside the data they carry, tokens are usually associated with tags. Dependencies between data are translated into tag matching and transformation. A node is enabled when its data is ready i.e. when a set of input tokens is matched. When a node fires, it consumes input tokens, computes a result and produces a result token on each output arc, thus enabling successive nodes. The dataflow model is asynchronous and self-scheduling since the execution of nodes is constrained only by data dependencies.

The Atomic Dataflow model (ADF) borrows the ideas behind the retry mechanism and the watch set, and extends them further to enable dataflow scheduling between transactions. The basic unit of work in the ADF model is the ADF task for which data dependency is explicitly defined using the trigger set. The trigger set is a set of data that controls the re-execution of a transaction which is encapsulated inside an ADF task. Each data from the trigger set is associated with an implicit buffer that stores input tokens. Buffering enables asynchronous execution of producer and consumer tasks, since producers are able to generate many tokens without waiting on consumers to consume them. Consequently, the only limitation which may result in blocking of ADF tasks is the size of implicit token buffers. Therefore, the

† This work was performed while the author was associated with Barcelona Supercomputing Center

ADF model conceptually resembles a dynamic dataflow model which allows any number of tokens to reside on a given arc. The difference is that the ADF model doesn't rely on tags for token matching, but instead employs token buffers.

The motivation for this work is the notion that implementations of the producer-consumer class of problems using transactional memory often perform poorly. In general, any data that represents inter-task coordination in the TM environment causes a large number of conflicts between transactions. On the other hand, these coordination variables are identified easily, because they stem directly from algorithm definitions. A programming environment that can better express these patterns can increase the determinism and the performance of concurrent programs. The ADF model delivers this expressiveness to the TM programming paradigm by allowing a programmer to explicitly identify coordination variables for each task. The ADF runtime schedules tasks based on the availability of the trigger set data. A consumer task blocks if its input dependencies are not satisfied and waits for the producer task to commit new data tokens. When the input is ready, a consumer ADF task atomically processes the data, commits the changes and returns to the beginning of the task to wait for new data. This automatic transaction re-execution is conceptually different from an ordinary "exactly-once" semantics of transactional memory. It presents the first foundation for the dataflow semantics of the ADF model. The second is the token flow between ADF tasks. During the commit, an ADF task can produce the data needed by some other ADF task, which then processes the data further. Since each ADF task is triggered only when a new data appears, a program exhibits a dataflow behavior. An important aspect of the ADF model is that it separates the transactional execution from the dataflow coordination. In case of abort, the task restarts a new transaction using the same set of input tokens.

The syntax for the Atomic dataflow model, its implementation and the execution model are described in Section 2. Section 3 shows the advantages of the ADF model using a parallel implementation of Dijkstra's algorithm for solving the single source shortest paths problem in directed graphs. Final remarks are given in Section 4.

2. ATOMIC DATAFLOW MODEL

The retry mechanism and the watch set proposal are two extensions that increase the programmability of transactional memory. Inspired by these ideas, we present Atomic dataflow model that employs dataflow scheduling principles to coordinate the execution of atomic tasks.

Figure 1 illustrates the use of retry for the implementation of a bounded buffer. If the buffer is full when a producer wants to insert a new value, or empty when a consumer wants to get buffered value, the appropriate transaction retries and blocks until the buffer state has changed. This example is straightforward: the read set of the retried transaction consists only of `itemCount` data. However, the read set of a large transaction can grow substantially before retry is executed. This can lead to spurious wakeups and useless transaction re-execution. The watch set extension can improve the performance in such cases, but a broad class of problems can be expressed more naturally using explicit coordination between program tasks based on inter-task

```
void producer() {
   while (!done) {
      transaction {
         if (itemCount == BUFFER_SIZE)
            retry;
         item = produceItem();
         putItemIntoBuffer(item);
         itemCount = itemCount + 1;
      }
   }
}

void consumer() {
   while (!done) {
      transaction {
         if (itemCount == 0)
            retry;
         item = getItemFromBuffer();
         itemCount = itemCount - 1;
         consumeItem(item);
      }
   }
}
```

Figure 1. Implementation of a bounded buffer using TM retry mechanism.

dependencies. On the other hand, programs that exhibit irregular parallelism are hard to reason about. Data dependencies and the behavior of complex tasks can differ from one execution to the other. Atomic dataflow model is able to explicitly express apparent inter-task dependencies as well as to handle obscure and hidden dependencies implicitly. The model is based on execution of atomic units of work called ADF tasks. The ADF task is scheduled when its input data dependencies are satisfied, while the body of the task is executed atomically.

This is the pseudocode syntax for the ADF task:

```
        adf task     [trigger_set (<set>)]
                     until (exit_condition)
        {  < task_body>   }
```

The `trigger_set` clause is used to explicitly declare input data dependencies for the task, while the `until` clause acts like a control input which is used to gracefully finalize the task. The `trigger_set` clause is optional because a task might not have any input dependencies. For example, a producer task in the implementation of an unbounded buffer doesn't need to check if the buffer is full before it produces the next item; hence its execution is independent.

2.1 Implementation

When the compiler encounters the ADF task declaration, it transforms the enclosed code region into the body of the ADF task. Additionally, it inserts runtime calls for task creation and dataflow support. Figure 2 demonstrates the code transformation. The `_adf_AddTriggers` runtime call registers the input data dependencies for the task. For each data from the trigger set the ADF runtime creates an implicit buffer to hold input tokens. The `_adf_ConsumeTokens` runtime call tries to obtain input tokens and blocks the task if some tokens are missing.

```
{
   int task_finish = 0;
   taskId = _adf_CreateTask();
   _adf_AddTriggers(taskId , <set>);
   while (!task_finish) {
      _adf_ConsumeTokens();
      transaction {
         if (!cond) {
            <task_body>
         }
         else
            task_finish = 1;
      }
   }
}
```

Figure 2. Code transformation for the adf task construct.

```
#define TRIGGER_SET(...)                          \
   _adf_AddTriggers(taskId , __VA_ARGS__);
#define UNTIL(X) X
#define ADF_TASK(TRIGGER, COND, TASK_BODY) { \
   int task_finish = 0;                       \
   taskId = _adf_CreateTask();                \
   TRIGGER                                    \
   while (!task_finish) {                     \
      _adf_ConsumeTokens();                   \
      transaction {                           \
         if (!COND) {                         \
            TASK_BODY                         \
         }                                    \
         else                                 \
            task_finish = 1;                  \
         }                                    \
      }                                       \
}
```

a)

```
ADF_TASK(TRIGGER_SET(var), UNTIL(cond),  {
      <task_body>
} )
```

b)

Figure 3. a) Definition of the ADF_TASK macro, b) Example of the macro invocation.

```
void *Producer() {
   ADF_TASK(    TRIGGER_SET(&produceToken),
                UNTIL(stopProduction),
   {
      item = produceItem();
      putItemIntoBuffer(item);
      // generate a new token for consumers
      consumeToken = 1;
    } )
}

void *Consumer() {
   ADF_TASK(    TRIGGER_SET(&consumeToken),
                UNTIL(stopConsumption),
   {
      item = getItemFromBuffer();
      consumeItem(item);
      // generate a new token for producers
      produceToken = 1;
    } )
}

int main () {
   // Initialize buffer
   BufferInit();
   // create producers
   for (i=1; i<=NumProducers; i++)
      ThreadCreate(Producer);
   // create consumers
   for (i=1; i<=NumConsumers; i++)
      ThreadCreate(Consumer);
   // generate initial tokens for producers
   ADF_TASK(    UNTIL(tokenNum == BUFF_SIZE), {
      tokenNum++;
      produceToken = 1;
   } )
   ...
   // stop the producer-consumer execution
   transaction {
      stopProduction = 1;
      stopConsumption = 1;
   }
   ThreadJoin();
}
```

Figure 4. The ADF implementation of a bounded buffer.

Ideally, the proposed ADF task syntax should be supported as a proper language construct. This would allow a compiler to inspect the code and collect all the information relevant to the ADF program execution. Namely, the compiler could statically determine the trigger set data for all ADF tasks and insert code to create token buffers for each input data. For example, it could insert a runtime call for token buffer creation before the user code. This would guarantee that no data is lost even if the producer task starts producing tokens before the consumer task is created. The similar effect can be achieved using `pragma` directives. However, programmers usually expect the program to work even when `pragma` directives are ignored, which would not be the case in the ADF model. Therefore, until the ADF tasks are properly supported in the compiler, the ADF_TASK macro from the Figure 3 can be used, although with some restrictions. Namely, each task has to be created by spawning a dedicated thread. Additionally, since token buffers for a given task are created at same time when the task itself is created, the program must be written such that no producer task is enabled before all the consumer tasks are created. This can be achieved by organizing a program in such a way that all ADF tasks are created before the main thread generates initial tokens that will start the dataflow execution. This method is illustrated with the ADF implementation of the bounded buffer in Figure 4. The main thread first creates a number of `Producer` and `Consumer` tasks, by spawning a new thread for each of them. Then, it creates a task whose purpose is to generate initial `produceToken` tokens. Finally, when the program is about to finish the execution, the main thread sets the exit condition for `Consumer` and `Producer` tasks and waits until all threads are terminated.

2.2 The ADF execution model

Atomic dataflow model is based on execution of ADF tasks which are scheduled according to the dataflow principles, when their input dependencies are satisfied. As such, the ADF task acts as a macro dataflow actor which repeats the execution as long as the tokens are present on its inputs or until the exit condition is met. The exit condition acts as a control input for the task and can be set either by the task itself, or by some other program thread. This models an aspect of the dataflow programming which assumes that the task processes one set of data, produces output data and then waits for the new matching set of input data.

The ADF runtime system creates an implicit buffer for each trigger set data. When the task is created, it is put into the ready queue. The thread that picks the task from the ready queue tries to consume input tokens for the task by checking token buffers (the `_adf_ConsumeTokens` runtime call). If all input data is ready, the thread proceeds with the task execution. Otherwise, it puts the task into the waiting queue for the token that is missing. Eventually, when the producing task generates the required token, the ADF runtime system selects the task from the waiting queue and passes the token to it. Then it checks if the remaining data dependencies for the task are satisfied and if so, it puts the task into the ready queue. Otherwise it moves the task into the waiting queue for the next missing token. Therefore, the token is passed directly to the task that is waiting for it, and put into the buffer only if all consuming tasks are busy.

When the task is scheduled, its body is executed atomically. This allows a task to safely share the state with other running tasks. In case of a conflict, the task transaction aborts and starts a new execution. However, this does not affect the dataflow execution since the task privatizes the tokens when it consumes them (it becomes the only task that has a reference to the token). The task restarts the transaction with the initial state of the input tokens. On the other hand, the shared state is certainly changed. Therefore, Atomic dataflow model doesn't impose any ordering between the tasks. The only guarantee provided by this model is that it matches the tokens by the order of their arrival to the buffers.

When the task wants to commit, the ADF runtime system checks if any data from the write set of the task transaction is in fact a trigger set data. If that is the case, it modifies the write operation and uses the data to create a new token. Then, it passes the token to the first task that is blocked in the waiting queue for that input. Otherwise, if the queue is empty, it puts the token into the buffer. If there are *N* different ADF tasks that are consumers of a given token, there will exist a separate buffer for each of those tasks. Therefore, the ADF runtime system will create *N* copies of the token and repeat the previous sequence of actions for each of the *N* consumer tasks.

Figure 5 illustrates the execution of the ADF model on an example of multiple producers and multiple consumers that synchronize on data *x*. Shaded circles denote busy tasks, while white circles are idle tasks. Figure 5a shows two producers that are concurrently producing a new value for *x*. Producer P_1 commits first and passes the new token directly to the consumer C_1. Notice that in any TM implementation, from the point of view of a single data, commits are always serialized. This means that, even if there are multiple concurrent producers in the system, the access to the token buffer is serialized. Next, the producer P_2 commits a new token and passes it to the consumer C_2 – Figure 5b. Then, in Figure 5c the producer P_1 produces a new token, but this time all consumers are busy. Therefore, the ADF runtime system puts the new token x_1 into the buffer. The same action is performed for the second token produced by the producer P2 – token x_2 in Figure 5d. In the meantime, the consumer C_1 has finished the previous execution and is ready to start a new one. Hence, it checks the buffer and finds the token x_1 ready, takes it from the buffer (thus consuming the token) and immediately continues with the execution – Figure 5e. Finally, the consumer C_2 consumes the token x_2 and continues the execution as well - Figure 5f. Thus, a consumer only needs to block if there are no input tokens in the buffer. This may improve the performance of the system due to a decrease in the frequency of the task queue operations.

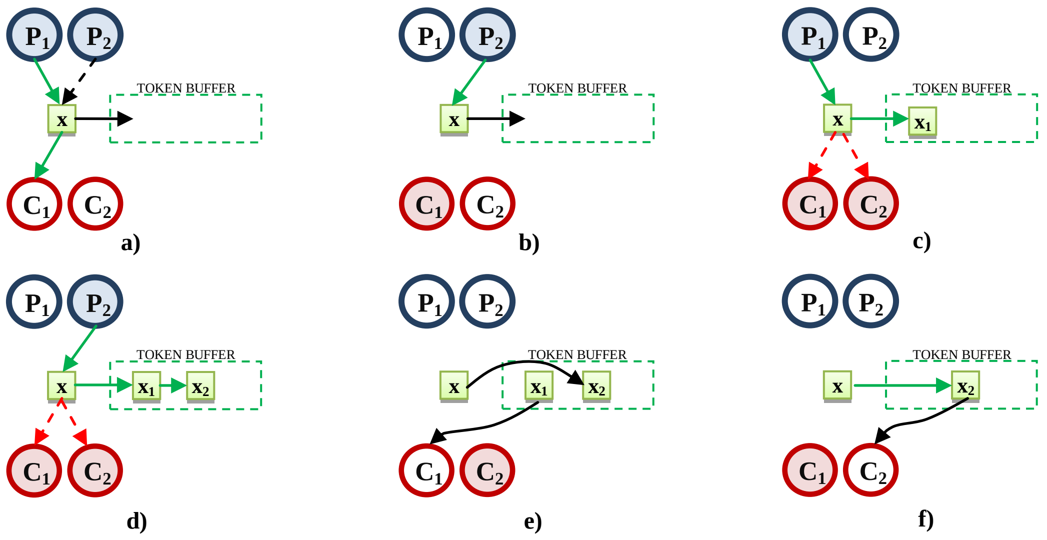


Figure 5. Illustration of the ADF execution model on the example of multiple producers (P) and multiple consumers (C) that synchronize on data x: a) P1 produces a token for C1, b) P2 produces a token for C2, c) P1 produces a token x1 and puts it into the buffer, d) P2 produces a token x_2 and stores it into the buffer, e) C1 consumes token x_1 and f) C2 consumes token x_2.

```
void Dijkstra_seq (int numV, int source) {
   int      u, new_dist;
   node_t  *v;

   // Update the source vertex
   dist[s] = 0;
   DecreaseKey(PQ, s, new_dist);
   // Find shortest paths to all other nodes
   while (!Empty(PQ)) {
      u = ExtractMin(PQ);
      v = adjlist[u];
      while (v) {
         new_dist = dist[u] + v->weight;
         if (new_dist < dist[v->nodeId]) {
            DecreaseKey(PQ, v->nodeId, new_dist);
            dist[v->nodeId] = new_dist;
            pred[v->nodeId] = u;
         }
         v = v->next;
      }
   }
}
```

a)

```
void Dijkstra_ADF (int numV, int source) {
   int i, u;

   for (i = 0; i < numV; i++) {
      // create task for each vertex
      ADF_TASK( TRIGGER_SET(&dist[i]), UNTIL(1), {
         int      new_dist;
         node_t  *v;
         v = adjlist[u];
         while (v) {
            new_dist = dist[u] + v->weight;
            if (new_dist < dist[v->nodeId]) {
               dist[v->nodeId] = new_dist;
               pred[v->nodeId] = u;
            }
         }
      } )
   }
}

int main() {
   Dijsktra_ADF(num_vertices, 0);
   // start the dataflow execution
   transaction {
      dist[0] = 0;
   }
   ...
   // wait for the end of execution
}
```

b)

Figure 6. Dijsktra's algorithm: a) Sequential algorithm, b) Parallel algorithm based on ADF tasks

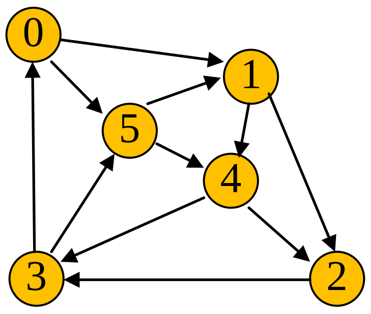
3. USE CASE

In order to illustrate the advantages of the ADF model we compare the sequential and the parallel, atomic dataflow implementation of Dijsktra's algorithm. This is a graph search algorithm that solves the single-source shortest path problem for a graph with nonnegative edges, producing a shortest path tree. The algorithm finds the path with the lowest cost (i.e. the shortest path) between a given source vertex and every other vertex in the graph. It is used in a variety of applications, like routing, VLSI design, robotics and transportation.

Given a directed graph *G* and a set of its vertices *V*, the algorithm visits all nodes of *G* in non-decreasing distance from a given source $s \in V$. The algorithm [14] starts from *s* and iteratively builds the set of visited nodes *S*. Additionally, the priority queue *PQ* is maintained to sort unvisited nodes by their provisional distance *dist[u]* from *S*. Initially, the distance for the node *s* is set to 0, and distances of all other nodes are set to ∞. In each iteration, a node *u* with the minimum distance from *S* is selected from the priority queue *PQ* and inserted into *S*. Then, all distances from *u* to its neighbors are updated. The edge relaxation is done for each edge $(u, v) \in E$ by checking the new distance to vertex *v*, $dist[u] + w(u, v)$, against the old distance $dist[v]$. If the new value is smaller, the vertex *v* is either added to *PQ* or, if *v* is already in the queue, its priority is decreased. The algorithm runs until all nodes have been visited. The `Dijkstra_seq` function in Figure 6a shows the sequential implementation of this algorithm.

The algorithm is characterized by a nested loop where the outer loop iterates over all the nodes in the graph while the inner loop updates the distances from the selected node to all its neighbors. The priority queue serializes the algorithm and parallelization strategies mostly aim to extract parallelism from the outer loop by working with multiple nodes from the unvisited set in parallel. In addition, there are some strategies that aim to parallelize inner loop by allowing concurrent accesses to the priority queue.

Our parallel algorithm is shown in Figure 6b. It applies the first parallelization strategy, but it doesn't use a priority queue. Instead, a single ADF task is created for each vertex $v \in V$ and put into the waiting state. The trigger set for the given vertex *u* consists of the distance *dist[u]* from the source vertex *s*. Initially, distances for all vertices are set to ∞. The execution starts when the main thread sets the distance of a source vertex *s* (in our case vertex 0) to 0, which triggers the ADF task for vertex *s*. The task for the vertex *s* updates distances to its neighbors which effectively triggers the execution of the neighbors' ADF tasks. An ADF task for a given vertex *u* is triggered each time the distance of the vertex *dist[u]* is updated. The execution is finished when all ADF tasks are in the waiting state.



Edge weights:

----

0 – 1	0.41	3 – 5	0.38
0 – 5	0.29	4 – 2	0.32
1 – 2	0.51	4 – 3	0.36
1 – 4	0.32	5 – 1	0.29
2 – 3	0.5	5 – 4	0.21
3 – 0	0.45		

Figure 7. Example of a directed graph with edge weights

Step	Enabled tasks	Distance and path arrays	1	2	3	4	5
0	**0^{0}(0)**	dist	-	-	-	-	-
		pred	-	-	-	-	-
1	**1^{0}(0.41)** **5^{0}(0.29)**	dist	0.41	-	-	-	0.29
		pred	0	-	-	-	0
2	**2^{01}(0.92)** **4^{01}(0.73)** 1^{05}(0.58) **4^{05}(0.5)**	dist	0.41	0.92	-	0.5	0.29
		pred	0	1	-	5	0
3	**3^{012}(1.42)** 2^{014}(1.05) **3^{014}(1.09)** **2^{054}(0.82)** **3^{054}(0.86)**	dist	0.41	0.82	0.86	0.5	0.29
		pred	0	4	4	5	0
4	0^{0123}(1.87) 5^{0123}(1.8) 0^{0143}(1.54) 5^{0143}(1.47) 3^{0542}(1.32) 0^{0543}(1.31) 5^{0543}(1.24)	dist	0.41	0.82	0.86	0.5	0.29
		pred	0	4	4	5	0
5	-	dist	0.41	0.82	0.86	0.5	0.29
		pred	0	4	4	5	0

Figure 8. One possible execution of the parallel Dijkstra's algorithm.

Figure 7 shows the graph with six vertices and its adjacency list with edge weights. The steps of one possible execution of parallel algorithm are shown in Figure 8. The second column shows enabled tasks in each step, where tasks in bold represent those which execution triggers the activation of a new tasks in the next step of the algorithm. The tasks are represented using the *u* ^{p}*(d)* format, where *u* denotes the vertex, *p* denotes the path from the source vertex *s* to vertex *u*, and *d* denotes tentative distance from the source vertex *s* to vertex *u*. The third column shows the distance and path arrays calculated in each step. Of course, the algorithm steps have only descriptive purposes and Figure 8 shows one of many possible executions of the algorithm. Actual execution might look quite different and in general depends on task scheduling.

4. CONCLUSION

This paper proposes the Atomic dataflow model which aims to combine two seemingly disjoint worlds of the shared memory and the dataflow parallel programming. The model is based on the dataflow execution of atomic tasks. Our work is motivated by two goals. The first goal is to enhance the programmability of transactional memory by exposing the expressiveness of the dataflow directly to a programmer. Our second goal is to improve the performance of transactional memory execution by eliminating unnecessary conflicts. We show the potential of the model on the example of parallel Dijsktra's algorithm. Once we finish the implementation of the model we are planning to evaluate its performance and scalability using a set of real world applications.

5. ACKNOWLEDGEMENTS

We would like to thank reviewers for their comments and feedback. This work is supported by the cooperation agreement between the Barcelona Supercomputing Center and Microsoft Research, by the Ministry of Science and Technology of Spain and the European Union (FEDER funds) under contract TIN2007-60625 and TIN2008-02055-E, by the European Network of Excellence on High-Performance Embedded Architecture and Compilation (HiPEAC) and by the European Commission FP7 project VELOX (216852). Vladimir Gajinov is also supported by a scholarship from the Government of Catalonia.

6. References

1 Arvind and Culler, D. E. *Annual review of computer science vol. 1, 1986, Dataflow architectures*. Annual Reviews Inc. , (Palo Alto, CA, USA 1986), 225-253.

2 Arvind and Iannucci, R. A. A critique of multiprocessing von Neumann style. In *ISCA '83: Proceedings of the 10th annual international symposium on Computer architecture* (New York, NY, USA 1983), ACM, 426-436.

3 Carlstrom, B. D., McDonald, A., Chafi, H., Chung, J., Minh, C. C., Kozyrakis, C., and Olukotun, K., The Atomos transactional programming language. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA 2006), ACM, 1-13.

4 Dennis, J. B. and Misunas, D. P. A preliminary architecture for a basic data-flow processor. *SIGARCH Comput. Archit. News*, 3, 4 (1974), 126-132.

5 Harris, T. and Fraser, K.. Language support for lightweight transactions. In *Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications* (New York, NY, USA 2003), ACM, 388-402.

6 Harris, T., Larus, J., and Rajwar, R. *Transactional Memory (Second Edition)*. Morgan & Claypool Publishers, 2010.

7 Harris, T., Marlow, S., Peyton-Jones, S., and Herlihy, M. Composable memory transactions. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming* (New York, NY, USA 2005), ACM, 48-60.

8 Harris, T., Plesko, M., Shinnar, A., and Tarditi, D. Optimizing memory transactions. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA 2006), ACM, 14-25.

9 Herlihy, M. and Moss, J. E. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on Computer architecture* (New York, NY, USA 1993), ACM, 289-300.

10 Johnston, W. M., Hanna, J. R., and Millar, R. J. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36, 1 (2004), 1-34.

11 Lee, B. and Hurson, A. R. Dataflow Architectures and Multithreading. *Computer*, 27, 8 (1994), 27-39.

12 McDonald, A., Chung, J., Carlstrom, B. D., Minh, C. C., Chafi, H., Kozyrakis, C., and Olukotun, K. Architectural Semantics for Practical Transactional Memory. In *Proceedings of the 33rd annual international symposium on Computer Architecture* (Washington, USA 2006), IEEE Computer Society, 53-65.

13 Milovanovic, M., Ferrer, R., Gajinov, V., Unsal, O. S., Cristal, A., Ayguade, E., and Valero, M. Multithreaded software transactional memory and OpenMP. In *Proceedings of the 2007 workshop on Memory performance: Dealing with Applications, systems and architecture* (New York, NY, USA 2007), ACM, 81-88.

14 Nikas, K., Anastopoulos, N., Goumas, G., and Koziris, N. Employing Transactional Memory and Helper Threads to Speedup Dijkstra's Algorithm. In *ICPP '09: Proc. 38th International Conference on Parallel Processing* (sep 2009).

15 Shavit, N. and Touitou, D. Software transactional memory. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing* (New York, NY, USA 1995), ACM, 204-213.

16 Silc, J., Robic, B., and Ungerer, T. Asynchrony in parallel computing: From dataflow to multithreading. 1997.

17 Spear, M. F., Sveikauskas, A., and Scott, M. L. Transactional memory retry mechanisms. In *Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing* (New York, NY, USA 2008), ACM, 453-453.

18 Veen, A. H. The misconstrued semicolon: reconciling imperative languages and dataflow machines. *Centrum voor Wiskunde en Informatica*, Amsterdam, The Netherlands, 1986.