Integrating Dataflow Abstractions into the Shared Memory Model

Vladimir Gajinov$^{*†}_{1}$, Srdjan Stipić$^{*†}_{1}$, Osman S. Unsal$^{†}_{1}$, Tim Harris$^{‡}_{2}$, Eduard Ayguadé$^{*†}_{1}$, Adrián Cristal$^{§}_{1}$
* Universitat Politecnica de Catalunya, †Barcelona Supercomputing Center, ‡ Oracle Labs in Cambridge UK(1),
§ Artificial Intelligence Research Institute (IIIA) - Spanish National Research Council
1 {name}.{surname}@bsc.es, 2 timothy.l.harris@oracle.com

Abstract—**In this paper we present Atomic Dataflow model (ADF), a new task-based parallel programming model for C/C++ which integrates dataflow abstractions into the shared memory programming model. The ADF model provides pragma directives that allow a programmer to organize a program into a set of tasks and to explicitly define input data for each task. The task dependency information is conveyed to the ADF runtime system which constructs the dataflow task graph and builds the necessary infrastructure for dataflow execution. Additionally, the ADF model allows tasks to share data. The key idea is that computation is triggered by dataflow between tasks but that, within a task, execution occurs by making atomic updates to common mutable state. To that end, the ADF model employs transactional memory which guarantees atomicity of shared memory updates. We show examples that illustrate how the programmability of shared memory can be improved using the ADF model. Moreover, our evaluation shows that the ADF model performs well in comparison with programs parallelized using OpenMP and transactional memory.**

Keywords - Parallel programming, dataflow, shared memory, transactional memory.

I. INTRODUCTION

Following the Moore's law, the number of transistors on integrated circuits keeps doubling approximately every two years. However, power and energy issues prohibit further frequency scaling of single processor cores. As a result, the trend in the processor architecture today is to increase the number of cores on the chip. But, in order to completely utilize the power of multicore processors, new programming models are needed that can offer an easier way to exploit parallelism, not only by expert programmers but also by the general programming community.

Current multicore processor architectures largely support a shared memory programming model. This is mainly because shared memory programming resembles the sequential style of programming since it provides a single view of data to all program threads. Still, multithreaded programming often requires complex coordination of threads which can introduce subtle and difficult-to-find bugs due to the interleaving of processing on data shared between threads. Synchronization mechanism based on mutual exclusion locks is the conventional method for controlling accesses to the shared data. However, problems such as deadlocks, race conditions, priority inversions and non-composability make programming with locks difficult.

Transactional Memory (TM) [10,12] 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.

Nevertheless, mutual exclusion alone is not sufficient to express the desired behavior of many applications. Threads may need to wait until certain condition holds true before attempting an operation on a data item. Lock-based programming provides the solution to this problem in the form of condition variables. Conceptually, a condition variable is a queue of threads, associated with a monitor, on which a thread may wait for a condition to become true. Similarly, the *retry* mechanism proposed by Harris et al. [11] provides blocking synchronization for transactional memory. The idea is that 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 the transaction which is bound to retry if the data used to evaluate the retry condition has not changed. McDonald et al. [16] extend this idea with the proposal for a *watch-set*. The authors provide a means for a programmer to explicitly add a variable to the watch set at any time before retry, thus defining the set of data that may trigger the re-execution of the transaction after retry.

While condition variables primarily model event-driven synchronization, the watch set extension models data-driven execution. Namely, a transaction is re-executed when a new value is produced for some of the data from the watch set. There is also an analogy with the dataflow program execution in which changing the value of a variable automatically forces recalculation of the values of variables which depend on it. Thus, changing a value of the watch set variable forces recalculation of the retry condition and potentially re-execution of the entire transaction.

In this paper we argue that the idea for the watch-set extension can be generalized and we show that the integration of dataflow abstractions can improve the programmability of shared memory model. The contributions of this work can be summarized as follows:

- We present Atomic Dataflow programming model for C/C++ which is based on a dataflow execution of atomic tasks. The model provides pragma directives that allow a programmer to organize a program into a set of tasks and to explicitly define data dependencies for each task.

(1) This work was done while the author was affiliated with Microsoft Research Cambridge.

- We provide examples that illustrate how the programmability of shared memory can be improved using the ADF model to organize a program as a dataflow task graph.
- Our evaluation also shows that the added programmability of using the ADF model has little to none performance cost. Moreover, the performance can even be improved in some cases compared to the pure shared memory implementation.

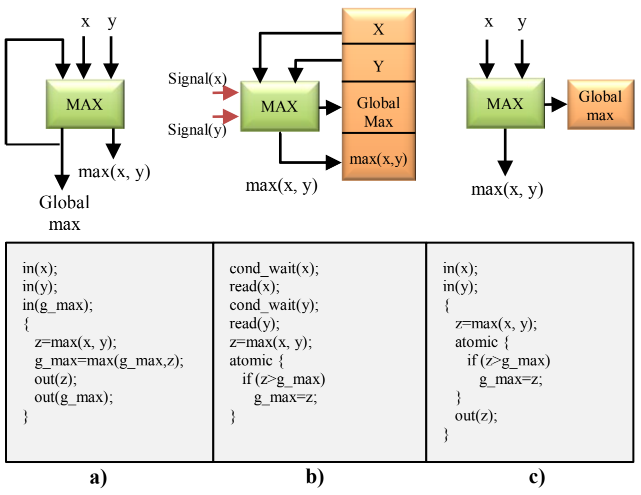
The rest of the paper is organized as follows. Section II explains the motivation for this work and gives an overview of the related work. In Section III we present the ADF model. In Section IV we show on two examples how the ADF model can improve the programmability of the shared memory model. Section V describes the evaluation while the results are presented in Section VI. In Section VII we conclude and give short discussion of future work.

II. Motivation

Since the beginning of the multicore era, there has been a resurgence of interest into the dataflow model due to its ability to efficiently exploit parallelism. The main characteristic of the dataflow model is that the execution of an operation is constrained only by the availability of its input data. A dataflow 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, which coordinate the program execution. A node, which can be a single operation or a block of operations, is enabled when its input data is ready. 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. Furthermore, it is inherently parallel since no hidden state is shared between operations.

Early dataflow projects have concentrated on dataflow scheduling of single instructions [3,7]. However, it was soon realized that the abundance of parallelism created with this approach can quickly consume available resources and hinder the performance [6]. Later, the research effort was put towards limiting the parallelism which resulted in many proposals for hybrid dataflow designs [19]. The individual dataflow node was no longer a simple operation but a thread of control-flow execution. Different hybrid dataflow designs have been proposed but most of them can be classified as coarse-grained dataflow model.

A recently proposed task-based parallel programming model, StarSs [17], also belongs to the class of coarse-grained dataflow. StarSs relies on programmer's annotations of input and output operands of program functions to dynamically construct the inter-task data dependency graph, and extract task parallelism at runtime. It follows a single assignment rule for the data shared between tasks.



```
in(x);
in(y);
in(g_max);
{
  z=max(x, y);
  g_max=max(g_max,z);
  out(z);
  out(g_max);
}
```

```
cond_wait(x);
read(x);
cond_wait(y);
read(y);
z=max(x, y);
atomic {
  if (z>g_max)
    g_max=z;
}
```

```
in(x);
in(y);
{
  z=max(x, y);
  atomic {
    if (z>g_max)
      g_max=z;
  }
  out(z);
}
```

Figure 1: Max function with the global maximum:
a) dataflow, b) shared memory and c) ADF implementation.

Scheduling is implicit and determined by the data dependency between tasks. However, it assumes that tasks have no side-effects and restricts annotations to the level of functions. The ADF model relies on programmer's annotations in a similar fashion as StarSs, but it does not impose the same limitations. Thus, the ADF model allows a programmer to define a task from an arbitrary region of the code and allows tasks to have side-effects. Moreover, we argue that stateful dataflow tasks provide additional flexibility and that there are numerous applications that can benefit from this abstraction. Finally, there are many applications where a state is fundamental to the problem. Bank account balances, state machines or reductions are examples of such applications.

As an illustration let us consider a simple example from Figure 1. Given two input streams of integers, *x* and *y*, the goal is to discover which of the current values of these two streams is larger, but we would also like to find the global maximum of all values received on both streams. Dataflow model handles the input streams naturally, but in order to maintain the global maximum the state of the previous execution has to be provided as an input for the next execution - Figure 1a. This means that the state has to be copied after each invocations of the MAX node. On the contrary, shared memory model treats the global maximum as the mutable state which can easily be updated in-place - Figure 1b. Still, we need to synchronize the access to this data in order to allow safe updates by concurrent threads. The issue with the shared memory implementation is how to deal with input data streams. A typical solution is to rely on some form of conditional waiting, i.e. conditional variables or transactional memory retry mechanism, through which we implement coordination between different threads of execution. Ultimately, using atomic dataflow tasks provided by the ADF model we can handle both of the above problems naturally. Figure 1c illustrates the idea: we handle the streams and find the current maximum in a dataflow fashion, while the global maximum is treated as a mutable state.

We are not the first to recognize the potential of including the state into the dataflow model. The work on M-structures [4] was one of the early attempts in this direction. Similarly, the concept of Monads [13] introduced the state into the pure functional programming language, Haskell.

Recently, transactional memory researchers have investigated the ways to provide more robust cooperation between concurrent transactions. Luchango and Marathe introduced transaction communicators [15], as special objects through which concurrent transactions can communicate and interact. The system tracks dependencies between transactions that access communicators and assures that mutually dependent transactions commit or abort together as a group. Similarly, transactional events [8] combine properties of synchronous message-passing events with the isolation property of transactions. Central to this abstraction is a sequencing combinator which enables sequential composition of events. Dependence-aware transactional memory (DATM) [18] is an implementation of transactional memory based on conflict serializebility. The runtime system tracks data dependencies between transactions and serializes the commit of dependent transactions. Finally, a model of Communicating Memory Transactions (CMT) [14], combines the Actor model with the software transactional memory in order to allow transactions to coordinate across actors or threads. In this model, concurrent transactions that cooperate by exchanging tentative messages form a set which effectively assures that either all transactions commit successfully or they all abort. Compared to these proposals, in the ADF model we decouple the dataflow and transactional execution: data dependency information is explicitly defined by a programmer and used to construct the dataflow graph that governs the scheduling of tasks based on the availability of task input data. Transactional execution is managed by the TM runtime system and it does not influence the dataflow execution. It is used only to protect mutually shared data.

III. The ADF model

In this section we describe language support for the ADF model in the form of C/C++ pragma directives and introduce the application programming interface.

A. *Language support*

Atomic dataflow model allows a programmer to delineate a program into a set of tasks, to explicitly express some of their dependencies and, at the same time, benefit from data sharing. The runtime system coordinates the execution of tasks based on the flow of data, and guarantees that each task accesses shared data atomically.

A task is defined using the *adf_task* pragma directive with the following syntax:

```
#pragma adf_task [ trigger_set (<set>) ]
                 [ until (exit_condition) ]
                 [ instances (<num>) ]
                 [ relaxed ]
                 { < task_body > }
```

Data dependencies for the task are defined using the *trigger_set* clause. The trigger set is a set of data that controls the firing of a task. Each data from the trigger set is associated with an implicit token 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. Conceptually, this resembles a dynamic dataflow model [3] which allows any number of tokens to reside on a given arc.

The ADF runtime system 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 task atomically processes the data, commits the changes and returns to the beginning of the task to wait for new data. During commit, the consumer task can produce the data needed by some other task, which then processes the data further. Since each ADF task is triggered only when a new data value is produced, the program is thus executed in a dataflow fashion. Finally, in case of abort, as a result of conflicting changes to the shared mutable state, the task restarts a transaction using the same set of input values.

An important aspect of the ADF model is that it decouples the transactional execution from the dataflow coordination. Namely, the ADF model guarantees that a given value of a dataflow token is only visible to the task that is using this value. Thus, from the dataflow perspective, the task execution is isolated from other concurrent tasks. This closely corresponds to the isolation property of transactional memory which ensures that the tentative changes to the shared state performed by a running transaction are not visible to other concurrent transactions. Therefore, transactional memory is an ideal mechanism for handling mutable state in combination with dataflow.

Usually, dataflow execution stops once it achieves the goal of the calculation; e.g. when the product of two matrices has been found. However, there are cases when the execution should proceed until some external signal is received. For example, a game server runs indefinitely processing clients' requests and terminates only when it is explicitly shutdown. In such case, the exit condition for a task can be defined using the *until* clause.

By default, the system creates a single instance of the task. A programmer can create more instances of the same task (that operate on the same set of input token buffers) using the *instances* clause.

Finally, using the *relaxed* option it is possible to switch off the implicit TM synchronization of the task body. The implicit transaction guarantees the atomicity of shared memory operations executed inside the task body, but in some cases a programmer can provide more optimal synchronization.

When the main thread initializes the dataflow region it unblocks worker threads and starts the dataflow execution using the second ADF directive:

```
#pragma adf_start
```

From there on, the execution of ADF tasks is self-scheduled and governed only by the production of data. The main thread waits for the end of the dataflow execution using the third and final ADF directive:

```
#pragma adf_taskwait
```

B. *The ADF model API*

The application programming interface that the ADF model exposes to the programmer and the compiler is shown in Figure 2. Only the first two routines are exposed to a programmer, while the remaining five runtime calls serve as the support for the expansion of the ADF pragma directives.

Every program has to call *adf_init* at the beginning of the main function to initialize the ADF runtime system. This function creates the task queues, initializes transactional memory support, and creates a pool of worker threads.

The main thread calls the *adf_terminate* routine at the end of the program to stop worker threads and to destroy the task graph and the runtime support for task scheduling.

The first two internal runtime calls serve as support for *adf_start* and *adf_taskwait* pragma directives. The final three routines from Figure 2 are used for the transformation of the *adf_task* directive. To create a task the program calls *adf_create_task* routine. This call accepts four parameters: 1) the number of task instances that should be created for this task, 2) the number of input tokens, 3) the pointer to the list of consumed input tokens which is provided by the runtime system, and 4) the name of the outlined procedure that represents the task. During dataflow execution, when the task instance consumes a set of input tokens, it becomes their exclusive owner. The tokens are used for the single execution of the task instance, after which they are destroyed. Finally, when a worker thread finds a ready task, it executes the task by calling an outlined procedure *fn*, passing the list of tokens as the parameter to the call.

The task calls *adf_pass_token* routine to pass the output tokens to their consumer tasks. The runtime system uses *addr* parameter to access the map that associates token addresses with input token buffers of consumer tasks. A copy of the token is provided for each of these token buffers.

When the exit condition of the *until* clause is set, the task calls *adf_task_stop* routine to stop its execution. This means that the task stops consuming input tokens and removes itself from further dataflow execution.

C. *Discussion*

For simplicity, in current implementation of the ADF model we use pragma directives to support the syntax of the model. The disadvantage of this approach is that removing ADF pragmas does not result with the correct sequential code. Alternatively, the model could be supported by extending the language or by using C++ template metaprogramming. Additional advantage of the later approach is that it enables compiler to check if dataflow tokens are used correctly in the code. Namely, the ADF model differentiates dataflow tokens from regular variables

```
/* programmer API */
void adf_init (int num_threads);
void adf_terminate();

/* internal API */
void adf_start ();
void adf_taskwait();
void adf_create_task (int num_instances,
        int num_tokens, void *tokens[],
        std::function <void (token_t *)> fn);
void adf_pass_token (void *addr, void *token,
                    size_t token_size);
void adf_task_stop();
```

Figure 2: The ADF model API.

in that the token address is never used to read or write the value of the token but only to identify the token. Declaring dataflow tokens using templates or type attributes would enable the compiler to issue an error if the address operation is applied to the token variable or if the token is used outside of the scope of the task. Another compile time check that is possible in the ADF model is to test if all tasks are reachable in the code which may help in ensuring the correctness of the ADF program. We leave further discussion of this topic for the future work.

IV. PROGRAMMING WITH ATOMIC DATAFLOW TASKS

In this section we show how the ADF model can improve programmability using two example applications: parallel bounded buffer implementation and game engine simulation.

A. *Bounded Buffer*

Let us first consider a multiple producer, multiple consumer bounded buffer. Figure 3 shows implementation of this application using OpenMP and ADF. In the OpenMP version (Figure 3a), the main thread creates a number of producer and consumer tasks. Each task executes an infinite loop which iteration represents a single operation on a bounded buffer. The task has to check the buffer condition before it can proceed with the operation. Hence, producers check if the buffer is full before they produce a new item and store it into the buffer, while the consumers check if the buffer is empty before they try to take the item from it. These operations are executed inside a transaction, but in order to avoid unnecessary re-executions of a transaction until a given operation is possible, we utilize retry mechanism.

In the ADF implementation (Figure 3b), the main thread creates a number of instances of Producer and Consumer tasks, which operate on a shared bounded buffer. We utilize ADF task directives and *num_instances* clause for this purpose. All instances of the Producer task share a single token buffer to store *produce* input tokens. Similarly, all instances of the Consumer task share the same *consume* token buffer. Next, the main thread creates a number of initial *produce* tokens which initiate a dataflow execution. The execution that follows is self-scheduled and coordinated by the production of *consume* and *produce* tokens by corresponding Produce and Consume tasks. For example, a Producer task produces an item, stores it into the buffer and generates the

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

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

int main() {
   …
   #pragma omp parallel
   {
      #pragma omp single
      for (int i=0; i<num_tasks; i++) {
         #pragma omp task
            producer();
         #pragma omp task
            consumer();
}}}
```

a)

```
void producer() {
   item = produceItem();
   putItemIntoBuffer(item);
   consumeToken = 1;
}

void consumer() {
   item = getItemFromBuffer();
   consumeItem(item);
   produceToken = 1;
}

void create_initial_tokens() {
   static int token_cnt = 0;
   #pragma adf_task until(token_cnt >= BUFFER_SIZE)
      produceToken = 1;
      token_cnt++;
}

int main() {
   …
   #pragma adf_task trigger_set(produceToken)
                    num_instances(num_producers)
      producer();

   #pragma adf_task trigger_set(consumeToken)
                    num_instances(num_consumers)
      consumer();

   create_initial_tokens();
}
```

b)

Figure 3: Bounded buffer: a) OpenMP implementation, b) implementation with atomic dataflow tasks.

consume token that will enable a single Consume task. Since the tasks are enabled only when the tokens are available, there is no need to check the state of the buffer. For example, if the buffer is empty, there will be no *consume* tokens available and all Consume tasks will be blocked. As a result, in the ADF implementation there is no need to maintain *itemCount* counter which effectively eliminates transactional memory conflicts caused by this counter.

B. Game engine

The second example is the simulation of the multiplayer game engine. Typically, multiplayer games are based on client-server architecture. Clients connect to the server and send requests that reflect players' actions in the game world. The server receives the requests and processes the clients' actions keeping the state of the game world consistent.

Parallelization is usually done by calculating the maximum area that player's action can affect and conservatively locking the entire area while the action is processed. This ensures atomicity of the world state updates, but at the same time it creates a coarse grained mechanism that penalizes the performance and has a negative effect on load balancing [1]. Using transactional memory instead of locks results in a more fine grained synchronization, but the overhead of running software transactional memory causes a sub-optimal performance of such implementation [9].

The ADF model allows a different approach to the problem. Instead of calculating the entire action in a single thread of execution, we can partition the game world in a number of areas and effectively divide each player's action into a series of smaller actions that are executed in different areas of the game world - Figure 4. A similar approach is applied in a distributed environment in [5]. Now, we can define a number of dedicated ADF tasks for each game area, and orchestrate the dataflow execution between them. Effectively, we restrict the calculation of the object's action to a single area at the time. If the action spans further, we generate a new dataflow token that describes the object and its action, and send the token to the destination task. Consequently, this triggers the execution of the destination ADF task that simulates the continuation of the player's action in the corresponding game area - Figure 4. As a result, the server provides finer-grained action processing which can increase the fairness in the game.

Additionally, each time a task executes a part of the player's action it needs to update the game world state. Since the ADF model executes each dataflow task atomically, it allows a programmer to handle the state naturally. Therefore, in the game world update example, the game world state is kept consistent regardless of concurrent accesses from different tasks.

Figure 5 shows the ADF implementation of the game engine. The main thread creates Process and Receive tasks and then waits for the external command for the server shutdown. In the meantime, Receive task receives clients request and forms the *action* token for the appropriate Process task. For a given area, the dedicated task consumes input *action* tokens that carry the object and its action, processes the object's action, calculates the direction for the next move and produces output *action* tokens accordingly.

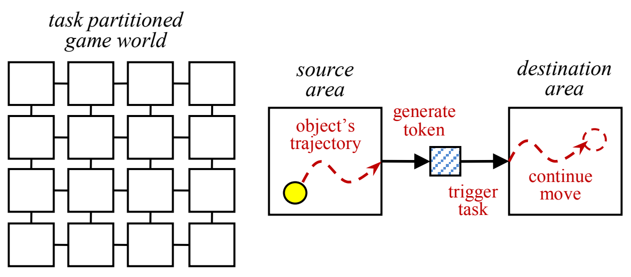


Figure 4: The ADF task partitioning of the game world.

```
/* tokens */
struct action_s **actionToken;

void CreateProcessTasks(int num_instances) {
  for (Ti=0; Ti<gamesize; Ti++)
    for (Tj=0; Tj<gamesize; Tj++) {

      #pragma adf_task trigger_set (actionToken[Ti][Tj])
                       until (stopGame == true)
                       instances(num_instances)
      {
          struct action_s act = actionToken[Ti][Tj];
          ProcessAction(act);
          GetNextDestination(obj, &X, &Y);
          actionToken[X][Y] = PackAction(obj, act);
          UpdateGameState();
      }
    }
}

void CreateReceiveTask(int num_receivers) {
  #pragma adf_task until (stopGame == true)
                   instances(num_receivers)
  {
      req = ReceiveRequest();
      struct action_s act = MakeAction(req);
      GetNextDestination(obj, &X, &Y);
      actionToken[X][Y] = PackAction(obj, act);
  }
}

int main () {
  adf_init();
  GameInit();
  /* create ADF tasks for each game area */
  CreateProcessTask (num_instances);
  /* create a task for receiving client's requests */
  CreateReceiveTask(num_receivers);

  #pragma adf_start
  WaitForServerShutdown(); /* external action */
  atomic{stopGame = true;}
  #pragma adf_taskwait

  DestroyGame();
  adf_terminate();
}
```

Figure 5: The ADF implementation of the game engine.

V. Evaluation

We have implemented the ADF model as a library written in C/C++. The code is compiled using the gcc compiler version 4.7 which supports transactional memory. We have used default GCC-TM runtime configuration in this evaluation. We evaluate the model on two different systems. The first machine is Dell PowerEdge 6850, with four dual-core 64-bit Intel® Xeon™ processors running at 3.2GHz. Each processor unit has 16MB L3 cache memory. The machine is running SUSE LINUX 10.1 operating system. The second machine is Apple Mac Pro with two 6-Core 64-bit Intel Xeon CPU X5650 Westmere processors running at 2.67GHz. Each processor unit has 12MB L3 cache memory. In addition, the cores are two way SMT-capable, giving a total number of 24 hardware threads. The machine is running Scientific Linux 6.2 (Carbon) operating system.

For each test application we analyze two ADF implementations: one with implicit task transactions (default) and the other with manually placed transactions (relaxed). We then compare the results of the ADF implementations against the results of corresponding sequential implementation as well as two OpenMP versions implemented using tasks – one in which the whole task body is enclosed in a transaction and the other in which transactions are placed optimally. We run each experiment ten times and report the average result. The evaluation is done using four applications with different characteristics. Bounded buffer application is a standard kernel application for testing producer-consumer systems which we covered in Section IV.A. The Game application is a simplified version of the game engine, similar to the one described in Section IV.B. Matrix multiplication is another kernel application used widely for testing parallel programming models. Finally, Lee's routing algorithm is a natural fit for the shared memory models. We use it to evaluate the performance of the ADF model with this type of applications.

VI. Results

We begin this section with the analysis of the overhead introduced by the ADF model. We measure the overhead using the built-in per-thread support for statistics gathering which is based on *rdtsc* instruction. Since the ADF tasks are created only once, before the beginning of the dataflow execution, the ADF runtime overhead can be broken into the task creation overhead and the task handling overhead. First, we measure the time spent in the task creation. Our profiling shows that the overhead of creating a single task is 1.95us or 6250 cycles, while the time needed to create a task instance is 0.62us or 2000cycles. This is expected since different instances of a task share some of the task support structures which are created during the creation of the first task instance. Next, we measure the task handling overhead which consists of the time wasted on all other task related operations except the task creation. Thus, we compare the single threaded execution of the ADF implementation with optimized sequential code that executes the same task inside for loop. In both cases we measure the time to execute 10000 tasks. We find that the ADF task handling overhead is approximately 2.1us or 6700 cycles per task. This gives us the lower bound for the duration of the task after which the performance is dominated by the task handling overhead.

A. Game simulation

To obtain the results for the Game application, we divide the game world into 8x8 domains and run the application for two seconds counting the number of executed tasks. In the ADF version each game domain is operated by a dedicated task. The communication between domains is strictly

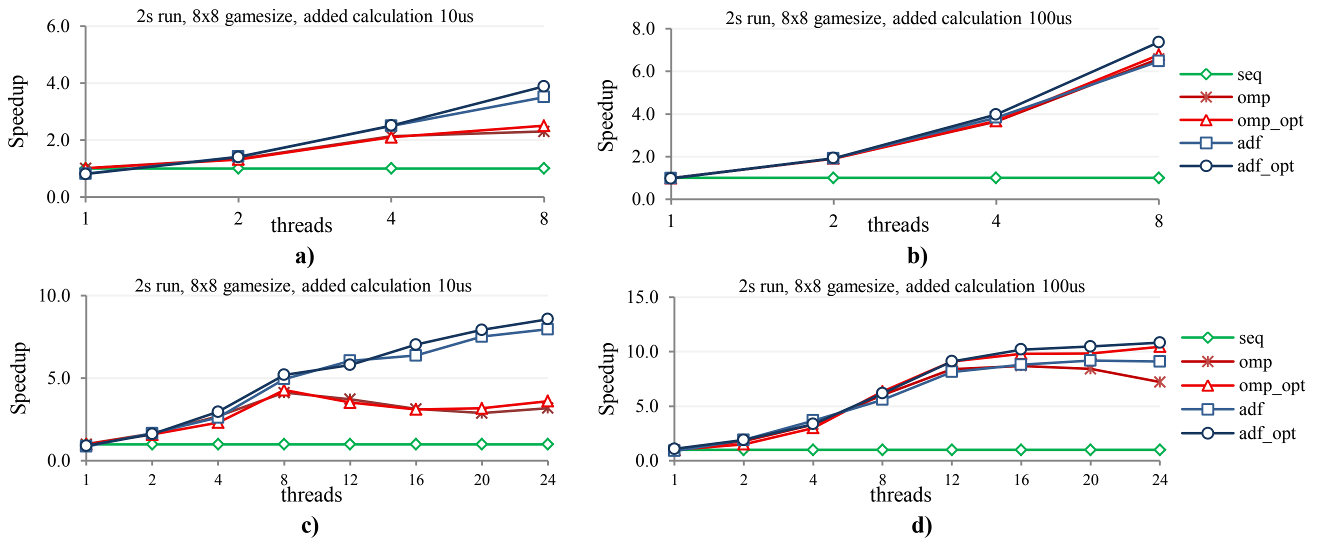


Figure 6: Game application: results for default and relaxed ADF and OpenMP implementations normalized against the results

through dataflow tokens. Data sharing happens only inside the domain when the object's action is processed and when the state of the game world is updated. In the OpenMP implementation a new task is dynamically created each time a game object moves and the runtime system is responsible for scheduling tasks to threads. In the sequential implementation the tasks are executed as loop iterations.

We simulate the useful computation by adding an arbitrary calculation time (implemented using an unoptimized for loop in order to prevent the thread to yield the processor). The rationale for adding the calculation time in the benchmark is that in the real game engine, processing of an object's action takes a significant time. However, only a portion of this time is spent in operations that access the shared data. The rest is spent in calculation of physical aspects of the action. In the default ADF implementation all of these operations are executed inside a transaction. In the relaxed version physic calculation is done outside of the transaction.

The first two graphs in Figure 6a and Figure 6b show the results when the application is run on the Dell machine, and when the added calculation time is 10us and 100us respectively. The next two graphs in Figure 6c and Figure 6d show similar results when the application is run on the Apple machine. All results are normalized against the results of the sequential implementation for the appropriate setup.

All ADF configurations show significant speedup and scalability. As expected, the optimized (relaxed) ADF implementation delivers the best performance. The maximum speedups achieved are 3.88 and 7.37 on the Dell machine, and 8.57 and 10.83 on the Apple machine. However, the difference in performance between two ADF versions is not big because the ADF model keeps the contention between tasks low by load balancing the execution to different game regions. This performance difference depends on the level of contention in the same game region. While in non-optimized version the entire action of a given object is processed inside a transaction, the optimized version processes the object's actions outside of the transaction and only updates the shared state from within the transaction. Thus, the more the server is loaded and the number of objects in the game world increases, this difference in the performance is expected to increase as a result of the lower number of conflicts and transaction aborts in the case of the optimized ADF implementation.

At the same time we see that the OpenMP implementations do not scale as well as the ADF versions for higher thread counts which is most noticeable in Figure 6c. To discover the cause for the impaired performance, we turn back to the OpenMP implementation of the Game application. The main thread starts the parallel region and creates a single task for each game object. Thereafter, at the end of each move, a given task creates a new child task that encapsulates the object and its next action. This nesting of tasks requires constant memory allocation for task structures while the parent tasks are still alive. Instead, the ADF model creates the tasks beforehand and reuses task structures throughout the dataflow execution.

B. Bounded Buffer

Figure 7 shows the results of the bounded buffer application which we use to test the throughput of the ADF model. Compared to the sequential execution, concurrent tasks in parallel implementations have to synchronize the access to the bounded buffer which adds contention overhead to parallel executions. Also, parallelism depends on the amount of time that each task spends in calculation before it accesses the buffer. From Figure 7 it is clear that the contention for the buffer access is the performance bottleneck in default parallel implementations, because the useful calculation is part of the task transaction. Thus, the calculation has to be repeated each time the transaction aborts. On the other hand, relaxed parallel implementations avoid repeating the calculation and are thus able to deliver better performance.

The synchronization bottleneck of the bounded buffer application is best illustrated by the diagram in Figure 7c.

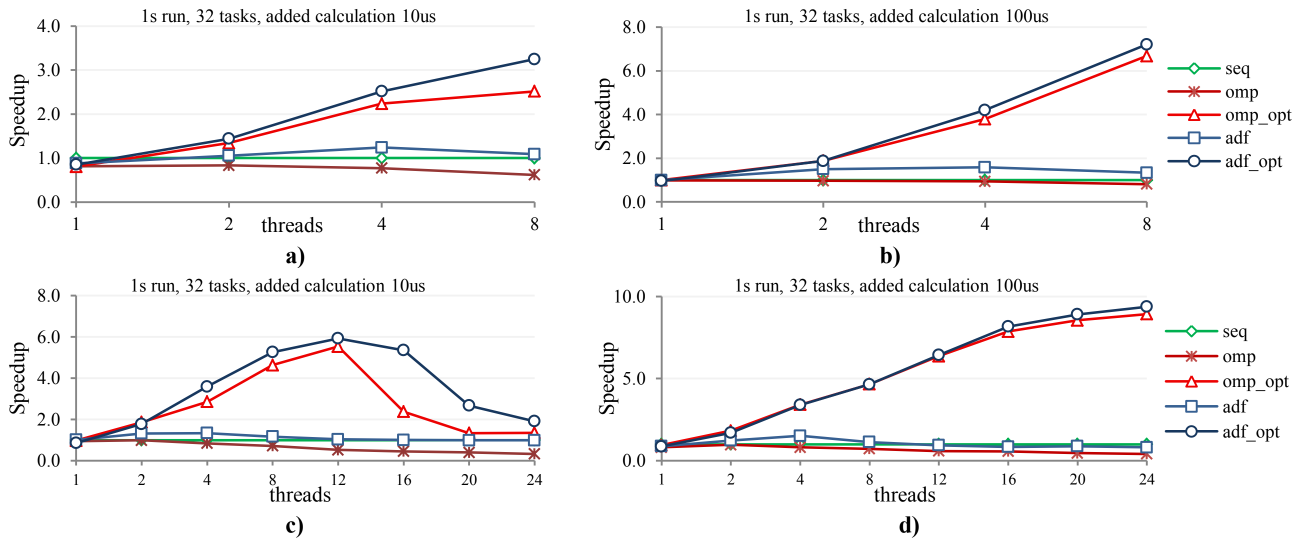


Figure 7: Bounded Buffer: results normalized against the results of the sequential implementation, for 10us and 100us added time.

For the small number of threads, 10us of useful calculation is sufficient to hide performance penalties of buffer synchronization and runtime task manipulation, but after 12 threads the performance drops as a result of increased contention. Increasing the useful calculation to 100us effectively hides more performance penalties, but we can expect the same behavior as in the previous case, only for larger thread counts.

Importantly, the ADF implementations outperform their OpenMP counterparts. This is because the contention involved in testing the empty and the full conditions of the bounded buffer is avoided by scheduling tasks only when they are ready. This further supports our premise that by integrating dataflow principles, the ADF model offers better support for producer-consumer class of problems than the shared memory model.

C. *Matrix Multiplication*

This application performs block matrix multiplication of two squared matrices of the size 2048x2048. The matrices are divided into 256 blocks, each with 128x128 elements. The block layout of matrices is appropriate for the dataflow execution: the multiplication task consumes two input blocks, one from each input matrix, and produces one output block used to calculate the resulting matrix. Figure 8 shows the speedup of the ADF and OpenMP implementations. The relaxed ADF implementation performs marginally better than the default ADF implementation due to the absence of the implicit transaction. Due to its support for dataflow, the ADF model is a better fit for this application than OpenMP which is evident from the results in Figure 8.

D. *Lee's Routing*

This application is an adapted version of Lee-TM benchmark which is used for testing transactional memory performance [2]. The benchmark is based on the well-known Lee's algorithm used in circuit routing. Lee's routing algorithm is attractive for parallelization as each route can potentially be treated as an independent transaction. The conflicts occur when any of the grid cells used by one route are concurrently used by another route. The ADF implementation defines two types of dataflow tasks: one that generates tokens which represent a single source-target cell pair, and the second that uses these tokens to find the route.

Figure 9 shows that the ADF version performs on par with the OpenMP version which is a natural fit for this application. In the lack of the real dataflow execution, the ADF task handling incurs additional overhead which explains a slight advantage of the OpenMP implementation. The performance of both parallel versions drops for high thread counts which can be accounted to the increased contention between different threads that concurrently try to include the same path into their routes. This has been studied previously in [2].

VII. CONCLUSION

This paper describes the ADF model which integrates dataflow abstractions into the shared memory model. The resulting model enhances the programmability by exposing the expressiveness of dataflow directly to a programmer. Additionally, it can improve the performance of transactional memory applications by eliminating unnecessary conflicts. Our evaluation shows that the model is a good fit for Producer-Consumer type of problems and for problems that exhibit irregular parallelism. In this paper, we describe the basic semantics of the ADF model which further opens many opportunities for improvement and the future work. The results are promising given that this is the first implementation of the proposed model. Many existing techniques can be applied in order to improve the model implementation and decrease the overhead of the ADF task handling. Despite the simple and unoptimized implementation, the results show that the ADF model mostly outperforms task-based OpenMP implementations of our test applications. We plan to develop a representative benchmark suite for evaluation of the ADF model which will help us to improve the implementation and enrich the set of programing abstractions.

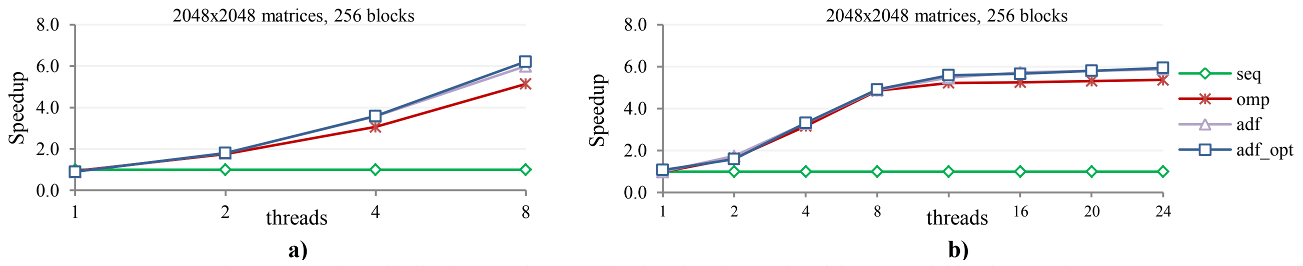


Figure 8: Matrix multiplication: results normalized against the results of the sequential implementation.

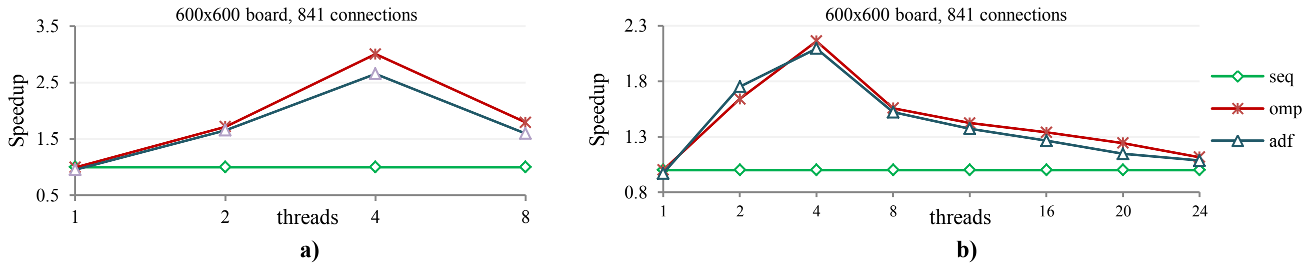


Figure 9: Lee's routing: results normalized against the results of the sequential implementation.

VIII. ACKNOWLEDGMENTS

We thankfully acknowledge the support Microsoft Research though the BSC-Microsoft Research Centre, the European Commission through the HiPEAC-3 Network of Excellence, the Spanish Ministry of Education (TIN2007-60625, and CSD2007-00050) and the Generalitat de Catalunya (2009-SGR-980).

IX. BIBLIOGRAPHY

1 Abdelkhalek, A. and Bilas, A. Parallelization and Performance of Interactive Multiplayer Game Servers. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS 2004)* (2004), IEEE Computer Society.

2 Ansari, M., Kostelidis, C., Jarvis, K., Lujan, M., Kirkham, C., and Watson, I. Lee-TM: A Non-trivial Benchmark for Transactional Memory. In *In* (Aiya Napa, Cyprus June 2008).

3 Arvind and Culler, D. E. Dataflow architectures (1986), 225-253.

4 Barth, P. S., Nikhil, R. S., and 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* (London, UK 1991), Springer-Verlag, 538-568.

5 Bharambe, A., Pang, J., and Seshan, S. Colyseus: a distributed architecture for online multiplayer games. In *Proceedings of the 3rd conference on Networked Systems Design and Implementation - Volume 3* (Berkeley, CA, USA 2006), USENIX Association, 12-12.

6 Culler, D. E. and Arvind. Resource Requirements of Dataflow Programs. In *ISCA '88: Proceedings of the 15th Annual International Symposium on Computer architecture* (Los Alamitos, CA, USA 1988), IEEE Computer Society Press, 141-150.

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

8 Donnelly, K. and Fluet, M. Transactional events. In *Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming* (New York, NY, USA 2006), ACM, 124-135.

9 Gajinov, V., Zyulkyarov, F., Unsal, O. S., Cristal, A., Ayguade, E., Harris, T., and Valero, M. QuakeTM: parallelizing a complex sequential application using transactional memory. In *Proceedings of the 23rd international conference on Supercomputing* (New York, NY, USA 2009), ACM, 126-135.

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

11 Harris, T., Marlow, S., Peyton-Jones, S., and Herlihy, M. Composable memory transactions. In *Proceedings of the 10th ACM SIGPLAN symposium on Principles and practice of parallel programming* (New York, NY, USA 2005), ACM, 48-60.

12 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.

13 Jones, S. P., Gordon, A., and Finne, S. Concurrent Haskell. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages, POPL'96* (St. Petersburg Beach, Florida, USA 1996), ACM.

14 Lesani, M. and Palsberg, J. Communicating memory transactions. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming* (New York, NY, USA 2011), ACM, 157-168.

15 Luchangco, V. and Marathe, V. J. Transaction communicators: enabling cooperation among concurrent transactions. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming* (New York, NY, USA 2011), ACM, 169-178.

16 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, DC, USA 2006), IEEE Computer Society, 53-65.

17 Perez, J. M., Badia, R. M., and Labarta, J. A dependency-aware task-based programming environment for multi-core architectures. In *Proceedings of the 2008 IEEE International Conference on Cluster Computing* (2008), 142-151.

18 Ramadan, H. E., Rossbach, C. J., and Witchel, E. Dependence-aware transactional memory for increased concurrency. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture* (Washington, DC, USA 2008), IEEE Computer Society, 246-257.

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