Recursion Support for the Data-Driven Multithreading Model

George Matheou
Department of Computer Science
University of Cyprus
Nicosia, Cyprus
Email: geomat@cs.ucy.ac.cy

Ian Watson
Department of Computer Science
University of Manchester
Manchester, United Kingdom
Email: watson@cs.man.ac.uk

Paraskevas Evripidou
Department of Computer Science
University of Cyprus
Nicosia, Cyprus
Email: skevos@cs.ucy.ac.cy

***Abstract*—Data-Driven Multithreading (DDM) is a threaded data-flow model that schedules threads for execution based on data availability on sequential processors. DDM utilizes a Thread Scheduling Unit (TSU) for the management of the threads on sequential processors. In this work we provide recursion support for the DDM model. This will provide more flexibility to the programmers. As a proof of concept, we have implemented the Fibonacci and NQueens algorithms in DDM. We also compared DDM with the OmpSs framework. The evaluation shows that DDM performs better on both applications.**

***Index Terms*—Data-Driven Multithreading, Recursion Support, Multi-core Systems**

I. Introduction

Data-Driven Multithreading (DDM) [1] is an execution model that allows Data-Driven scheduling on conventional processors. The core of the DDM model is the Thread Scheduling Unit (TSU) which schedules threads dynamically at runtime based on data availability. In DDM, a program is divided into a number of threads. For each thread DDM collects meta-data that enables the TSU to manage the dependencies among the threads and determine when a thread can be scheduled for execution. Data-Driven scheduling enforces only a partial ordering as dictated by the true data-dependencies which is the minimum synchronization. This is very beneficial for parallel processing because it exploits the maximum possible parallelism.

The DDM model was evaluated by three different software implementations: the Data-Driven Network of Workstations (D^2Now) [1], the Thread Flux Parallel Processing Platform (TFlux) [2] and the Data-Driven Multithreading Virtual Machine (DDM-VM) [3], [4]. DDM was also evaluated by two hardware implementations. In the first one, the TSU was implemented as a hardware peripheral in the Verilog language and it was evaluated through a Verilog-based simulation [5]. The second one [6] was the full hardware implementation on an 8-core system.

The main contribution of this work is the implementation of recursion support for the DDM model. This will provide more flexibility to the DDM programmers. Also, it allows more parallelism to be exploited in DDM applications/benchmarks. We explored the mechanisms that are needed for supporting recursion in DDM by studying two different data-flow models, the U-Interpreter [7] and the Packet Based Graph Reduction (PBGR) [8], [9], [10].

As a proof of concept we have implemented two famous recursive algorithms: Fibonacci and NQueens. DDM is compared with the OmpSs framework [11], [12], [13] on a 32-core AMD processor. The evaluation shows that DDM outperforms OmpSs on both applications. This is justified by the fact that OmpSs builds the dependencies graphs at runtime. On the other hand, DDM builds the dependencies graphs of the programs at compile-time which incurs less overheads.

The rest of the paper is organized as follows: An overview of Data-driven Multithreading is presented in Section II. Section III briefly reviews how the U-Interpreter model and the PBGR approach handle recursive calls. Section IV describes the recursion support for the DDM model. The experimental results are presented in Section V. Section VI describes the related work and Section VII concludes this paper.

II. Data-Driven Multithreading

The Data-Driven Multithreading (DDM) [1] is a non-blocking multithreading model that allows data-driven scheduling on sequential processors. A DDM thread (called DThread) is scheduled for execution after all of its required data have been produced, thus no synchronization or communication latencies are experienced after a DThread begins its execution. In the DDM model, the DThreads have producer-consumer relationships. DThreads' instructions are executed by the CPU sequentially in a control-flow manner. This allows the exploitation of control-flow optimizations, either by the CPU at runtime or statically by the compiler.

In DDM, a program consists of the DThreads' code, the Thread Templates and the Dependency Graph. A Thread Template holds the meta-data of a DThread. The Dependency Graph describes the consumer-producer dependencies amongst the DThreads. DDM is utilizing the Thread Scheduling Unit (TSU), a special module responsible for scheduling the DThreads in a data-driven manner. The TSU uses the Thread Templates and the Dependency Graph to schedule DThreads for execution when all of their producer-threads completed their execution. This ensures that all data needed by a DThread is available, before it is scheduled for execution.

A. The DDM Dependency Graph

The DDM Dependency Graph is a directed graph where the nodes represent the DThreads and the arcs represent the data dependencies amongst the DThreads. Each DThread is paired with a special value called Ready Count (RC) that represents the number of its producers. A simple example of a Dependency Graph is shown in Figure 1 which is composed of six DThreads. The DThreads T2, T3 and T4 have one producer, the T1, as such their RC is set to 1. The RC value of T5 and T6 is equal to 2 because they have two producers. The RC value is initiated statically and is dynamically decremented by the TSU each time a producer completes its execution. In DDM, the operation used for decreasing the RC value is called *Update*. A DThread is deemed executable when its RC value reaches zero, such as the DThread T1 of the Figure 1.

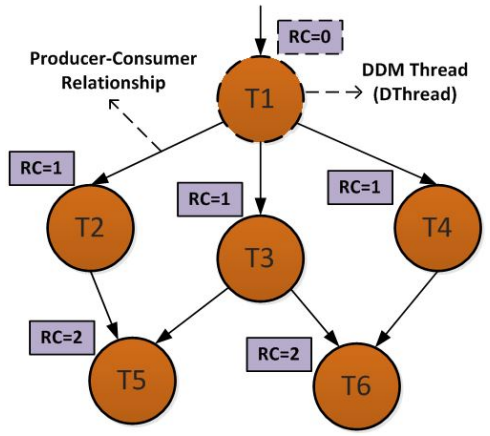

Fig. 1: Example of a DDM Dependency Graph.

B. The Context attribute

The Context attribute is a 32-bit value that enables multiple instances of the same DThread to co-exist in the system and run in parallel. This is essential for programming constructs such as loops and recursion. This idea was based on the U-Interpreter's tagging system [7] which provides a formal distributed mechanism for the generation and management of the tags at execution time. This system was used in Dynamic data-flow architectures to allow loop iterations and subprogram invocations to proceed in parallel via the tagging of data tokens [14].

Figure 2 depicts a simple example of using multiple instances of the same DThread through the Context attribute. The for-loop shown on the top of the figure is fully parallel, thus it can be executed by only one DThread (DThread 1). Each instance of the DThread is identified by the Context and it executes the inner command of the for-loop. The for-loop is executed 16 times, thus 16 instances are created with Contexts from 0 to 15.

C. The Thread Template

A DThread is identified by the *ThreadID* (TID) and *Context* and is paired with its *thread template* (or meta-data). The TSU uses the thread templates to schedule the DThreads dynamically at runtime. A *thread template* contains the following information:

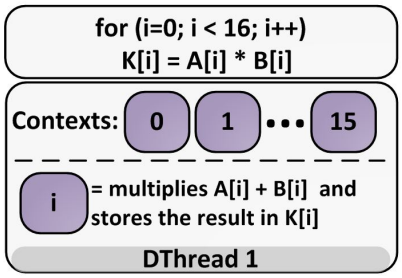

Fig. 2: Spawning multiple instances of the same DThread.

- **Instruction Frame Pointer (IFP):** a pointer to the address of the DThread's first instruction.
- **Ready Count (RC):** a value that is equal to the number of the producer-threads of the DThread.
- **Nesting:** The DDM model allows the parallelization of nested loops that can be mapped into a single DThread by using the Nesting attribute [3], [4], [6]. Three nesting levels are supported, i.e. the DThreads are able to implement one-level (Nesting-1), two-level (Nesting-2) or three-level (Nesting-3) nested loops. The indexes of the loops are encoded into the 32-bit Context value. The TSU uses the Nesting attribute to manage the Context value properly, during the update operations. An example of one-level loop (Nesting-1) is depicted in Figure 2 where the Context is equal to the index of the loop.
- **Scheduling Policy:** the method that is used by the TSU to map the ready DThreads to the cores. The Scheduling Policy consists of two fields: the scheduling method and the scheduling value. Two basic scheduling methods are provided by the TSU, dynamic and static. The dynamic method distributes the thread invocations to the cores in order to achieve load-balancing. In the static method, the DThread instances are assigned to a specific core. For this purpose, the scheduling value is used to hold the identity of the specific core.
- **Consumer Threads:** a list of the DThread's consumers that is used to determine which RC values will be decreased, after the DThread completes its execution.

D. The DDM Node

Figure 3 depicts a DDM processing node which consists of a commodity multi-core processor and the TSU. Each processor's core communicates with the TSU via two queues: the Acknowledgment Queue (AQ) and the Firing Queue (FQ). The AQ contains information of executed DThreads while the FQ contains the DThreads that are ready for execution. The TSU uses three main units for the storage, the Graph Memory (GM), the Consumer List (CL) and the Synchronization Memory (SM). The GM contains the thread template of each DThread while the CL holds the consumers of each DThread. The SM contains the Ready Count (RC) values for each DThread.

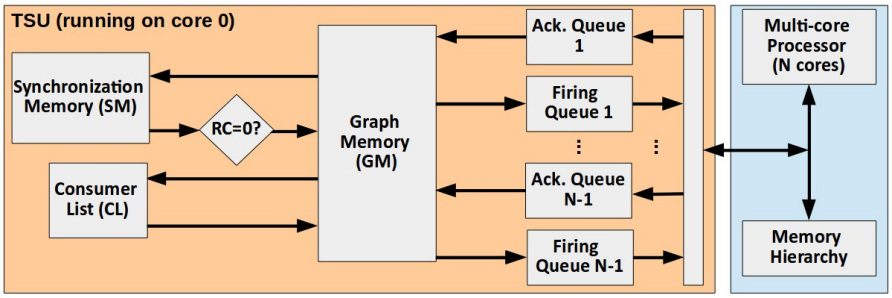

Fig. 3: A DDM processing node.

Each core reads the Instruction Frame Pointer (IFP) of the next DThread that will be executed from its FQ. After the core completes the execution of a DThread, it stores in its AQ the Thread ID and Context of the completed DThread. The TSU's control unit fetches the completed DThreads from the AQs, locates their consumers in the CL and then updates the RC of the corresponding consumer-threads in the SM. If the RC value of any of these consumers reaches zero, then they will be deemed ready to be executed and so they are stored in the appropriate FQs (using their Scheduling Policy) and wait for their execution.

III. Handling Recursion in Data-flow

A. The U-Interpreter model

The DDM model is based on the U-Interpreter [7] and uses token tagging to distinguish between different instantiations of a static code template. Current implementations have focused on iteration using locations in memory to match tagged tokens. This is an implementation of the L and D operators in the U-Interpreter model.

In the U-Interpreter model, each instance of the execution of an operator is called an *activity* which has a unique name. Each token/value carries the name of its destination activity. An activity name consists of four fields <u.c.s.i> where *u* is the context field, *c* is the code block name, *s* is the instruction number and *i* is the initiation/iteration number. Each value is combined with its destination activity name into a packet which is called *tagged token*.

A procedure activation is implemented by using four basic operators:

- A: it creates a new context u'.
- BEGIN: it receives the A's output and it replicates tokens for each fork.
- END: it sends the result to A^{-1}.
- A^{-1}: it replicates its output for its successors.

Figure 4 depicts a simple program which outputs the square of a number. This doesn't have any parallelism or recursion and could therefore be handled by the simple technique of inlining. However, it helps with the explanation of more general mechanisms if we start with a simple case. There are three features that we need from a general mechanism:

- The ability to create a separate thread for a function call.
- The ability to return the result from a particular function call to different call sites.
- The ability to distinguish different instantiations of parallel function calls.

```
int main() {
 s = square(4);
 printf("result:%d\n",s);
}
```

Call

```
int square(int n) {
 int result;
 result = n*n;
 return result;
}
```

Fig. 4: A program that computes the square of a number.

Figure 5 depicts the U-Interpreter graph of the square function of Figure 4. The A node is a representation of the action at the call site. The function *square* and the argument 4 are shown as two tokens <u.sq> and <u.n> both with a context u. The A operation constructs a new token <u',n> which is directed to the input of the square function (sq).

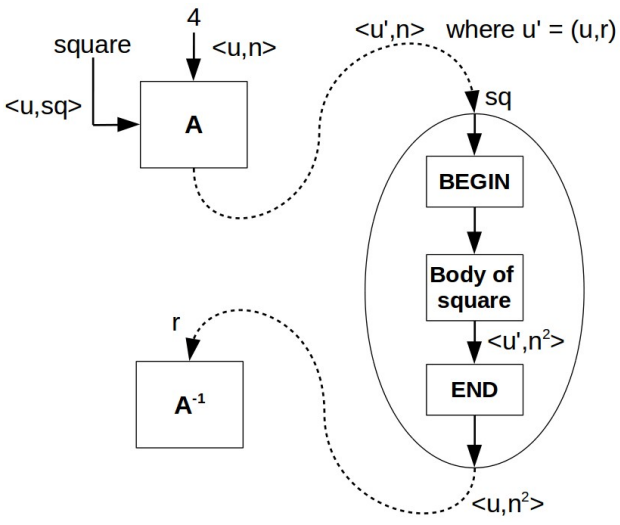

Fig. 5: U-Interpreter graph of the square function call.

The U-interpreter description uses the (u,r) pair as a new context. The call site has the information of the location r (the address of the node to which the result will be directed) and can clearly construct the new context from the information which it has locally. The reason it does this is threefold:

- The new context must be unique and contexts constructed from this data must satisfy this.
- It is necessary to pass the return link to the function so that it knows where to return its result.
- It is necessary to pass the old context to the function so that it can restore this old context to the return value.

Having executed the body of the function in the new context, the *END* operator takes the result, replaces the old context and directs the resulting token to the return link address.

In order to explore further the mechanisms that are needed for supporting recursion in DDM, it is worth looking at the Fibonacci algorithm (Listing 1). Although this function it's not an efficient implementation of the Fibonacci algorithm, it has the complexity of double recursion.

```
int fib(int n){
 if (n == 0 || n == 1)
  return n;
 else
 return fib(n-1) + fib(n-2);
}
```

Listing 1: Implementation of Fibonacci series using recursion.

The basic U-interpreter representation of the Fibonacci algorithm is shown in Figure 6. The recursive calls to the Fibonacci function are depicted as dotted instantiations. If we want to generate new threads to execute the recursive calls in parallel, it will be necessary to split the final add operation from the rest of the body as a continuation as we cannot execute it until the recursive calls have returned. The graph shows how the computation could be split into two threads: T1 and T2. The execution of T1 will cause the creation of the continuation T2 and two new frames will be created for the recursive calls of T1.

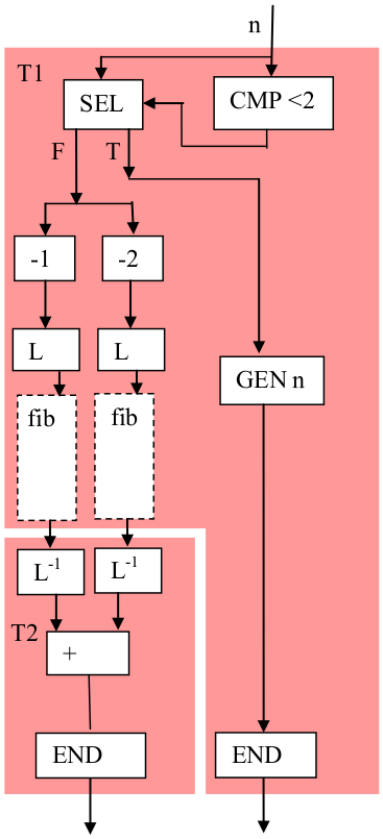

Fig. 6: U-Interpreter graph for the Fibonacci algorithm.

B. The PBGR approach

The Packet Based Graph Reduction (PBGR) approach was used in the ALICE [8] and Flagship [9], [10] projects back in the 1980s. PBGR generates new sections of Data-flow graph dynamically to instantiate the body of a function. In PBGR a *packet* (or thread descriptor) contains the following information:

- A function pointer (f)
- Arguments (a1, a2, ...)
- A return pointer (ra)
- A suspension count (sc)

The computation is in the form of a graph of packets held in memory which is shared between cores which can perform computation. A packet is *active* if its suspension count (sc) is zero. A scheduling queue contains the addresses of all active packets. The process of computation (or graph reduction) involves cores taking addresses of active packets from the scheduling queue and operating on their contents as determined by the code referenced by the function pointer. This may involve simply performing computation on the arguments and returning a value (to the place pointed to by ra) or it may involve the construction of a new piece of graph.

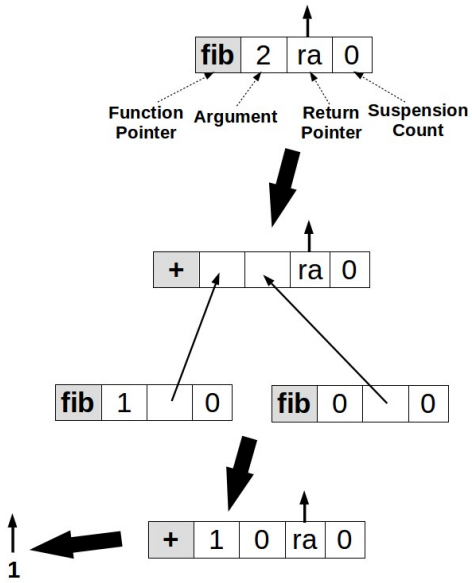

Fig. 7: PBGR evaluation of fib(2).

Figure 7 illustrates the PBGR evaluation of the Fibonacci algorithm for n=2. The evaluation starts with a single active packet. We will assume that this call will return a value to

an outer level computation, e.g. a print function, and we will omit any detail of this. The fib(2) example involves only one packet reduction with the creation of two calls which will in turn execute but return a value and generate no further packets.

The first *fib* packet generates three new packets: two *fib* packets and one *add* packet. The *add* packet is suspended on two values which are to be provided via the ra fields of the two *fib* packets which are immediately created as active (i.e. with sc=0 and an entry placed on the scheduling queue). Assume that the two *fib* packets execute in parallel. Because of their argument values, they immediately return a value (the one packet will return the value one and the other packet will return the value zero). As they do so, they decrement the suspension count of the *add* packet. When it becomes zero, the *add* packet becomes active. The *add* packet then executes to produce the value 1 ($1 + 0 = 1$) which is then returned to the outer level computation.

IV. Recursion Support for the DDM model

According to the U-Interpreter graph of Figure 6, two different threads (T1 and T2) are required for supporting the parallel execution of the Fibonacci algorithm. The T1 will be responsible for spawning the recursive calls while the T2 will be responsible for summing/reducing the return values of the children-calls and return the results to the parent-calls.

More specifically, in the case of $n > 2$, an instance of T1 (the parent-call) will spawn two additional instances of T1 (the children-calls). When the children-calls finish their execution, an instance of T2 will be responsible for summing the return values of the children-calls and sending the result ($fib(n-1)+fib(n-2)$) to the parent-call. In this section we will discuss the basic functionalities and data-structures that are needed for implementing the Fibonacci algorithm in DDM. Our approach can also be used for solving more complex recursion problems.

A. Implementing the thread T1 in DDM

For implementing the thread T1 in DDM the following functionalities are required:

*1) **Allowing multiple instances of the same thread**:* In DDM this functionality can be provided by utilizing the Context attribute which is based on the U-Interpreter model (see Section II-B).

*2) **A mechanism that will allow the spawning of recursive function calls**:* This can be done by using a single DThread where its instances are responsible for executing the recursive function calls. An instance can spawn additional recursive calls by using the Update command. An Update command consists of two basic attributes: the ThreadID and Context.

*3) **A mechanism that will allow T1 to behave as regular function**:* Currently, in the DDM model, each DThread has a block of instructions. The Instruction Frame Pointer (IFP) is used to point to the address of the first instruction of the block. When an instance of the DThread is ready for execution the TSU uses the IFP to execute its code. The code takes as input data only the Context of the instance.

Each recursive instance of T1 needs to behave as regular function, i.e each instance needs to have an argument list (AL) and a return value (RV). These attributes are also used in the packets of the PBGR approach. A special data-structure (called *RData*) is provided to hold the AL and RV of each recursive instance. An array can be used for recursive functions where their number of instances is known at compile-time (Figure 8). Each element of the array corresponds to a different instance. An instance is able to manage (read/write) its RData entry by using its Context value. Notice that in case of the Fibonacci algorithm, the AL consists of a single integer value (n). The RV is also an integer value.

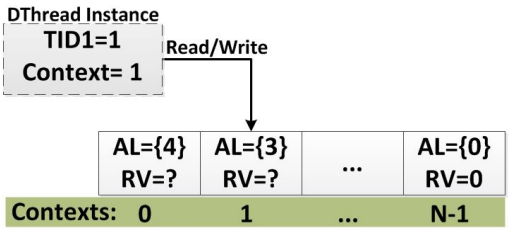

Fig. 8: RData implemented as a fixed-size array.

For recursive functions where their number of instances is not known at compile-time, a hash-map data-structure can be used (Figure 9). Accessing an RData entry is an associative operation based on the Context value. The allocation/deallocation of the RData entries will be performed as the execution proceeds by the instances in parallel. Thus, the hash-map has to allow concurrent insert and delete operations from several instances without the need to block them.

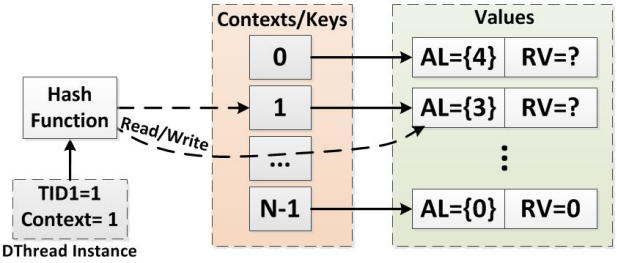

Fig. 9: RData implemented as a hash-map.

*4) **A mechanism that will guarantee that all recursive instances of T1 will have unique Context values, at runtime**:* The instances of T1 run concurrently and each instance accesses its RData entry by using its Context value. As such, a mechanism is needed to assign unique Context values to the instances. For this functionality we are proposing two different methods:

Method 1: Assign Context values to the children-calls based on the Context of the parent-call. In the case of Fibonacci, the recursive calls construct a binary tree of executions. We can simply borrow an idea from the binary heap when it is implemented in an array where the root is at index zero. Thus, the children-calls will have the following Contexts:

1) Child_1's Context = $2 * Parent's\ Context + 1$

2) Child_2's Context = $2 * Parent's\ Context + 2$

This method cannot be used as a general solution for assigning unique Context values to the instances since it depends exclusively on the algorithm.

Method 2: Use a global atomic variable/counter that holds the next available Context value. When a parent-instance spawns a child-instance, the Context value of the child-instance will be equal with the value of the counter. After that, the counter's value will be increased by one. This method is algorithm independent but has the potential of blocking threads.

*5) **Create a Thread Template for T1**:* Finally, a new thread template has to be stored in the TSU's Graph Memory which will be corresponded to the T1 DThread. The Thread Template has the following characteristics:

- **IFP =** the pointer to the T1's code.
- **Ready Count (RC) =** 1. This is because each instance has only one producer, its parent-instance.
- **Nesting =** Nesting-1.
- **Scheduling Policy =** Dynamic.

B. Implementing the thread T2 in DDM

Each instance of T2 corresponds to one parent-call of T1. As such, a T2's instance and its corresponded T1's instance can have the same Context value. This simplifies the procedure of assigning unique Context values to the T2's instances. A T2's instance is responsible for summing the return values of the children-calls of a parent-call. The result of the sum operation is actually the return value of the parent-call. For this functionality the T2 thread needs to access the RData data-structure of T1 in order to write the return value of the parent-call to its RData entry.

Finally, a new thread template for T2 has to be stored in the TSU with the following characteristics:

- **IFP =** the pointer to the T2's code.
- **Ready Count (RC) =** 2. This is because each instance has two children-instances of T1 as producers.
- **Nesting =** Nesting-1.
- **Scheduling Policy =** Dynamic.

C. The DDM Dependency Graph of the Fibonacci algorithm

Figure 10 depicts the Dependency Graph of the Fibonacci algorithm with $n = 4$. The graph consists of two DThreads, the T1 and T2. The solid arrows indicate Update operations and the dotted arrows indicate write operations to the *RV* attribute of the T1's instances.

D. DDM code of the Fibonacci algorithm

The DDM pseudo-code for the Fibonacci algorithm is depicted in Listing 2. The pseudo-code includes all the necessary functions and data-structures that are needed for supporting recursion in the DDM model. In this example, we are using an RData data-structure implemented as a fixed-size array. The size of the RData is equal to 2^n. This is because the total number of nodes of the binary tree of Fibonacci(n) is about 2^n.

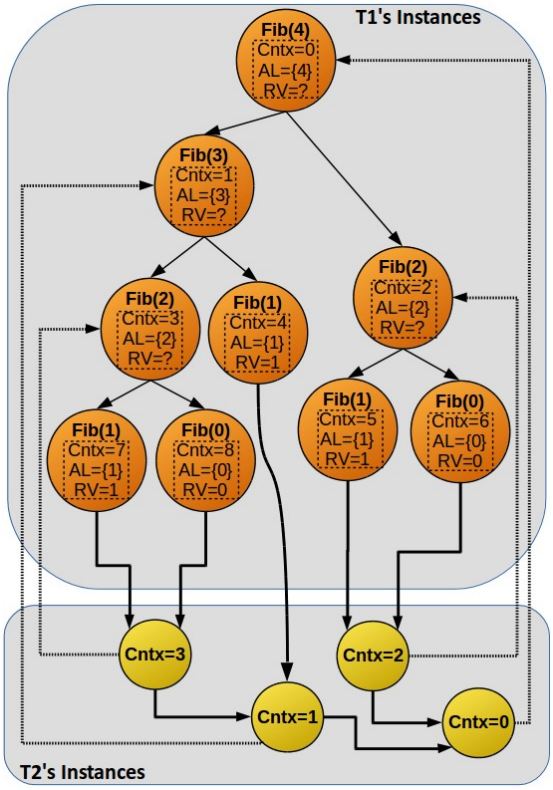

Fig. 10: Fibonacci's DDM Dependency Graph.

For generating unique Context values for the T1's instances we are using the formula that was described in Section IV-A4 (Method 1). The *GET_PARENT* macro is used to calculate the Context of a parent-instance based on the Context value of a child-instance. This formula is also borrowed by the binary heap implementation. This special macro is required for sending Update commands to the parents of the currently executed instances. Another solution to this requirement is to store the parent's Context of a child-instance in its RData entry. This solution will increase the memory consumption of the program.

```
#define GET_PARENT(C) floor((c-1) / 2)

RData rdata; // The RData (implemented as an array)

// The Code of the T1 DThread (ThreadID=2)
T1_code(Context cntx) {
 // Get the AL of the current instance, i.e n
 n = GET_ARG_LIST(rdata, cntx);

 // Return n to the parent of the current instance
 if (n == 0 || n == 1) {
  // Set the RV of the current instance
  SET_RETURN_VALUE(rdata, cntx, RV=n);
  // Update the T2 instance of the parent
  UPDATE(TID=3, Context=GET_PARENT(cntx));

  return;
 }

 // Call fib (n-1)
 SET_ARG_LIST(rdata, Context=2*cntx + 1, n-1);
```

```
 UPDATE(TID=2, Context=2*cntx + 1);

 // Call fib (n-2)
 SET_ARG_LIST(rdata, Context=2*cntx + 2, n-2);
 UPDATE(TID=2, Context=2*cntx + 2);
}

// The Code of the T2 DThread (ThreadID=3)
T2_code(Context cntx) {
 // Get the RV of the first child
 c1_RV = GET_RETURN_VALUE(rdata, 2*cntx + 1);
 // Get the RV of the second child
 c2_RV = GET_RETURN_VALUE(rdata, 2*cntx + 2);
 // Set the return value (RV) of the parent
 SET_RETURN_VALUE(rdata, cntx, RV=c1_RV + c2_RV);
 // Update the parent instance
 UPDATE(TID=3, Context=GET_PARENT(cntx));
}

// The main function
main(){
 // The maximum number of instances
 size = power(2, n);

 // Allocate elements for the RData
 ALLOCATE(rdata, size);

 // Load T1 and T2 DThreads in TSU
 ADD_T1_DTHREAD(TID=2, IFP=T1_code's address, RC=1,
      Nesting=1, Scheduling=Dynamic, Consumers={});

 ADD_T2_DTHREAD(TID=3, IFP=T2_code's address, RC=2,
      Nesting=1, Scheduling=Dynamic, Consumers={});

 // Call the intance with Context=0 (Root instance)
 SET_ARG_LIST(rdata, Context=0, n);
 UPDATE(TID=2, Context=0);

 START_DDM_SCHEDULING();

 // The RV of Context 0 (the root) holds the result
 result = GET_RETURN_VALUE(rdata, 0);
 PRINT result;
}
```

Listing 2: DDM pseudo-code for the Fibonacci algorithm.

E. Differences between DDM and PBGR

The major differences between the DDM model and PBGR approach are the following:

- PBGR holds the synchronization (or suspension) count in the packet/frame allocated to hold the arguments and context of the particular function instantiation. DDM uses a separate synchronization memory area, called Synchronization Memory (SM).
- DDM has a Thread Template for the function where the pointer to the executable code is held. PBGR stores a pointer to code directly. All the knowledge about how to construct any new graph associated with a function call is embedded in the code.

V. Experimental Results

To evaluate the recursion support of DDM we have used a 64-bit HP server machine that supports 32 threads. The machine has the following characteristics:

- *Clock Speed:* 1.4GHz
- *L1 Data Cache:* 16KB (4-way set associative)
- *L1 Instruction Cache:* 64K (2-way set associative)
- *L2 Cache:* 2MB (16-way set associative)
- *L3 Cache:* 6MB (64-way set associative)
- *Main Memory:* 48GB DDR3 RAM clocked at 1333MHz

Out of the 32 threads, one is used to run the TSU, while the rest are used for executing DThreads. The execution time measurements were collected using the *gettimeofday* system call.

For the performance evaluation we compare DDM with OmpSs (version 15.04 [11]) for the Fibonacci benchmark (Figure 11). The results show that our model has lower parallel execution time than OmpSs.

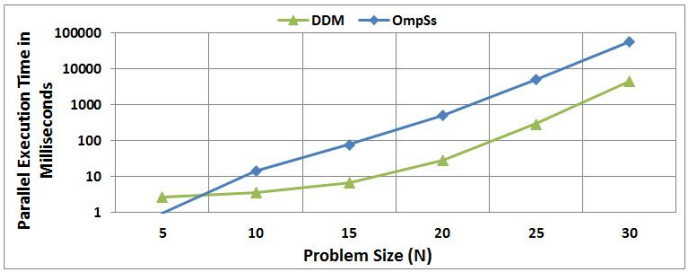

Fig. 11: DDM vs OmpSs for the Fibonacci benchmark.

Moreover, the NQueens benchmark is used to compare DDM with OmpSs (Figure 12). The comparison shows that DDM outperforms OmpSs for all problem sizes. In Figure 12, speedup represents how many times a certain parallel execution is faster than the corresponding sequential execution. The baseline for the speedup is the original sequential one, i.e without any DDM overheads.

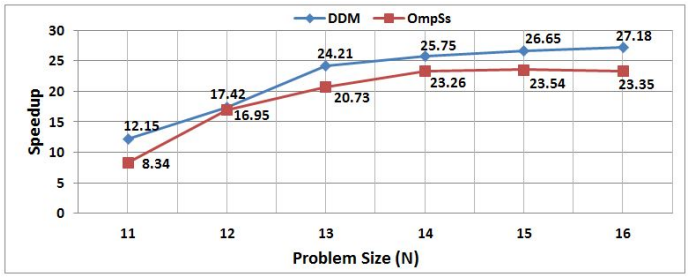

Fig. 12: DDM vs OmpSs for the NQueens benchmark.

VI. Related Work

OmpSs [11], [12], [13] is a programming model that provides the features of the StarSs [15], [16] using OpenMP directives. This framework allows to express data-dependencies between tasks using the *in, out and inout* clauses. The Nanos++ runtime system is used to support task parallelism using synchronizations based on data-dependencies. Also, the Mercurium source-to-source compiler is used. Mercurium recognizes constructs and transforms them to calls to the runtime. The task dependency graph is built at runtime, hence this approach incurs extra overheads. The DDM model provides

support for both Static and Dynamic dependency resolutions [17].

The Scheduled DataFlow (SDF) [18], [19] is a multi-threaded data-flow architecture that decouples the synchronization from the computation of non-blocking threads. In SDF a processor consists of two pipelines, the execution pipeline and the synchronization pipeline. The execution pipeline is responsible for executing the threads. The synchronization pipeline is responsible for scheduling the non-blocking threads. The main difference between DDM and SDF is that in SDF the computation is carried out by a custom designed processor while in DDM the computation is carried out by an off-the-shelf processor.

The Decoupled Threaded Architecture - Clustered (DTA-C) [20] is an SDF-based architecture with the addition of the concept of clusterizing resources. The architecture is composed of a set of clusters where each cluster consists of one or more Processing Elements (PEs) and a Distributed Scheduler Element (DSE). The Distributed Scheduler (DS) consists of all the system's DSEs and it's responsible for assigning threads at runtime.

SWARM (SWift Adaptive Runtime Machine) [21] is a software runtime that uses an execution model based on codelets [22]. The codelets model was based on the EARTH project [23]. Codelet is a collection of instructions that can be scheduled "atomically" as a unit of computation which is run until completion. SWARM divides a program into tasks with runtime dependencies and constraints that can be executed when all runtime dependencies and constraints are met. The runtime schedules the tasks for execution based on resource availability. Also, SWARM utilizes a work-stealing approach for on-demand load-balancing. SWARM provides support only for Static dependency resolution while DDM provides support for both Static and Dynamic dependency resolutions [17].

All the frameworks described above provide functionalities for supporting recursion.

VII. Conclusions

In this paper we provide the basic functionalities for supporting recursion in the Data-Driven Multithreading (DDM) model. Our approach is based on the U-Interpreter model and the Packet Based Graph Reduction (PBGR) approach. As a proof of concept we have implemented and evaluated two recursive algorithms: Fibonacci and NQueens. We also compared our model with the OmpSs framework on a 32-core AMD processor. The evaluation shows that DDM outperforms OmpSs on both applications.

Acknowledgment

This work was partially funded by the IKYK foundation through a scholarship for George Matheou.

References

[1] C. Kyriacou, P. Evripidou, and P. Trancoso, "Data-driven multithreading using conventional microprocessors," *IEEE Trans. on Parallel and Distributed Systems*, vol. 17, no. 10, pp. 1176–1188, Oct. 2006.

[2] K. Stavrou and et al, "TFlux: a portable platform for data-driven multithreading on commodity multicore systems." IEEE, Sep. 2008, pp. 25–34.

[3] S. Arandi and P. Evripidou, "Programming multi-core architectures using data-flow techniques," in *SAMOS-2010*. IEEE, 2010, pp. 152–161.

[4] ——, "DDM-VMc: the data-driven multithreading virtual machine for the cell processor," in *Proc. of the 6th Int. Conf. on High Performance and Embedded Architectures and Compilers*, 2011, pp. 25–34.

[5] G. Matheou and P. Evripidou, "Verilog-based simulation of hardware support for data-flow concurrency on multicore systems," in *SAMOS XIII, 2013*. IEEE, 2013, pp. 280–287.

[6] ——, "Architectural support for data-driven execution," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 4, pp. 52:1–52:25, Jan. 2015. [Online]. Available: http://doi.acm.org/10.1145/2686874

[7] Arvind and Gostelow, "The u-interpreter," *Computer*, vol. 15, no. 2, pp. 42–49, Feb. 1982.

[8] P. G. Harrison and M. J. Reeve, "The parallel graph reduction machine, alice," in *Graph Reduction*. Springer, 1987, pp. 181–202.

[9] P. Watson and I. Watson, "Evaluating functional programs on the flagship machine," in *Functional Programming Languages and Computer Architecture*. Springer, 1987, pp. 80–97.

[10] I. Watson, V. Woods, P. Watson, R. Banach, M. Greenberg, and J. Sargeant, "Flagship: a parallel architecture for declarative programming," in *ACM SIGARCH Computer Architecture News*, vol. 16, no. 2. IEEE Computer Society Press, 1988, pp. 124–130.

[11] BSC, "The ompss programming model," 2015. [Online]. Available: https://pm.bsc.es/ompss

[12] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "Ompss: a proposal for programming heterogeneous multi-core architectures," *Parallel Processing Letters*, vol. 21, no. 02, pp. 173–193, 2011.

[13] V. K. Elangovan, R. M. Badia, and E. A. Parra, "Ompss-opencl programming model for heterogeneous systems," in *Languages and compilers for parallel computing*. Springer, 2013, pp. 96–111.

[14] I. Watson and et al, "A prototype data flow computer with token labelling," in *Managing Requirements Knowledge, International Workshop on*. IEEE Computer Society, 1989, pp. 623–623.

[15] J. M. Pérez, P. Bellens, R. M. Badia, and J. Labarta, "Cellss: Making it easier to program the cell broadband engine processor," *IBM Journal of Research and Development*, vol. 51, no. 5, pp. 593–604, 2007.

[16] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta, "Hierarchical task-based programming with starss," *International Journal of High Performance Computing Applications*, vol. 23, no. 3, pp. 284–299, 2009.

[17] S. Arandi, G. Michael, P. Evripidou, and C. Kyriacou, "Combining compile and run-time dependency resolution in data-driven multithreading," in *Data-Flow Execution Models for Extreme Scale Computing (DFM), 2011 First Workshop on*. IEEE, 2011, pp. 45–52.

[18] J. M. Arul and K. M. Kavi, "Scalability of scheduled data flow architecture (sdf) with register contexts," in *Algorithms and Architectures for Parallel Processing, 2002. Proceedings. Fifth International Conference on*. IEEE, 2002, pp. 214–221.

[19] K. M. Kavi, R. Giorgi, and J. Arul, "Scheduled dataflow: Execution paradigm, architecture, and performance evaluation," *Computers, IEEE Transactions on*, vol. 50, no. 8, pp. 834–846, 2001.

[20] R. Giorgi, Z. Popovic, and N. Puzovic, "Dta-c: A decoupled multithreaded architecture for cmp systems," in *Computer Architecture and High Performance Computing, 2007. SBAC-PAD 2007. 19th International Symposium on*. IEEE, 2007, pp. 263–270.

[21] C. Lauderdale, M. Glines, J. Zhao, A. Spiotta, and R. Khan, "Swarm: A unified framework for parallel-for, task dataflow, and distributed graph traversal," *ET International Inc., Newark, USA*, 2013.

[22] S. Zuckerman, J. Suetterlein, R. Knauerhase, and G. R. Gao, "Using a codelet program execution model for exascale machines: position paper," in *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*. ACM, 2011, pp. 64–69.

[23] H. H. Humy, O. Maquelin, K. B. Theobald, X. Tian, X. Tang, G. R. Gao, P. Cupryky, N. Elmasri, L. J. Hendren, A. Jimenez *et al.*, "A design study of the earth multiprocessor," 1995.