FREDDO: an efficient Framework for Runtime Execution of Data-Driven Objects

George Matheou
Paraskevas Evripidou

TR-16-1 **January 2016**

University of Cyprus

Technical Report

FREDDO: an efficient Framework for Runtime Execution of Data-Driven Objects

George Matheou

UCY-CS-TR-16-1

January 2016

Table of Contents

Abstract

The switch to multi-core and many-core systems has created the need for efficient parallel processing in mainstream computing. To achieve that, efficient parallel programming models and architectures must be developed. Such a model is the Data-Driven Multithreading (DDM) model of execution. DDM is a non-blocking multithreading model that schedules threads for execution based on data availability on conventional processors.

In this work we introduce FREDDO, an efficient object oriented implementation of DDM, and its evaluation on a 32-core machine using a suite of nine benchmarks. FREDDO is a C++ framework that supports efficient data-driven execution on conventional processors. FREDDO incorporates new features like recursion support and simpler programming methodology to the DDM model. Our aim is to create a robust system that can scale to HPC level. In order to evaluate the ability of our framework to support recursion, we have implemented the Fibonacci and NQueens algorithms in DDM.

The evaluation shows that FREDDO scales well and tolerates scheduling overheads and memory latencies effectively. We have also compared our system with the DDM-VM, the OpenMP and the OmpSs frameworks on high-complexity applications like LU and Cholesky. Our framework outperforms the OpenMP and OmpSs on these benchmarks. We also compared FREDDO with the SWARM framework for the Fibonacci algorithm and with the Intel's TBB framework for the Swaptions benchmark. The evaluation shows that FREDDO performs better on both applications. Finally, we show that our framework achieves similar results with DDM-VM despite the fact that the former provides more functionalities and the size of the DDM applications is reduced to half.

1 Introduction

Since multi/many-core architectures are the default, researchers and processor vendors are aggressively seeking for execution models that will be able to efficiently keep all the available resources busy [1]. Conventional multiprocessor systems based on control-flow architectures suffer from long memory latencies and waits due to synchronization events [2].

Data-Driven Multithreading (DDM) [3] 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 enable 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. Unlike GPU programming models, the DDM model supports the execution of multiple concurrent threads where each thread is able to execute its independent group of instructions.

The DDM model was evaluated by three different software implementations: the Data-Driven Network of Workstations (D^2Now) [3], the Thread Flux Parallel Processing Platform (TFlux) [4] and the Data-Driven Multithreading Virtual Machine (DDM-VM) [5, 6]. In this work, we present the FREDDO framework, an optimized object oriented implementation of DDM. The main contributions of this work can be summarized as follows:

- We introduce a new optimized implementation of the DDM model which provides several improvements over the previous implementations.
- We are providing the basic functionality for supporting recursion in the DDM model.
- We improve the programmability of DDM programs.

The TSU and a Runtime system were implemented in the C++ language. The Runtime system is a software support that allows the execution of DDM programs on any commodity Operation

System (OS). More specifically, the Runtime provides communication functionalities between the TSU and the processor's cores.

FREDDO is evaluated using a suite of nine benchmarks on a 32-core AMD processor. The evaluation showed that the platform scales well and tolerates synchronization and scheduling overheads efficiently. We have also compared our system with the DDM-VM, the OpenMP [7] and the OmpSs [8, 9, 10] frameworks. Our framework outperforms the OpenMP and OmpSs especially in the case of high-complexity applications. FREDDO is compared with the SWARM framework [11, 12] for the Fibonacci algorithm and with the TBB framework [13, 14] for the Swaptions benchmark. The evaluation shows that FREDDO performs better on both applications. Finally, we show that FREDDO achieves similar results with DDM-VM, despite the fact that the former provides more functionalities. Also, due to the new programming interface, the size of the DDM applications implemented in FREDDO is reduced to half, compared to the size of the DDM-VM applications.

The rest of the report is organized as follows: An overview of Data-driven Multithreading is presented in Section 2. Section 3 describes the FREDDO architecture. The programming methodology for our implementation is presented in Section 4. The experimental results are presented in Section 5. Section 6 describes the related work. Future work and conclusions are presented in Sections 7 and 8 respectively.

2 Data-Driven Multithreading

The Data-Driven Multithreading (DDM) [3] 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, 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.

2.1 DDM Implementations

The DDM model was evaluated, by several software implementations. The first implementation, the Data-Driven Network of Workstations (D^2Now) [3], was targeting Networks of Workstations. It has illustrated the major components of DDM such as the TSU and CacheFlow [15]. The evaluation was done using execution driven simulations. That was followed by two other implementations, the TFlux [4] and the Data-Driven Multithreading Virtual Machine (DDM-VM) [5, 6]. Both TFlux and DDM-VM were targeting data-driven concurrency on sequential multiprocessors. DDM-VM supported distributed multi-core systems for both homogeneous and heterogeneous systems [16]. TFlux also developed the TFlux directives and a source-to-source compiler. The TFlux compiler was gradually extended to support all DDM systems.

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 [17]. The results show that the TSU module can be implemented on an FPGA device with a moderate hardware budget. The second one [18] was the full hardware implementation with an 8-core system. A software API and a source-to-source compiler were provided for developing DDM applications. For evaluation purposes, a Xilinx ML605 Evaluation Board with a Xilinx Virtex-6 FPGA was used. This implementation showed that data-driven

execution can be implemented on sequential multi-core systems with very small hardware budget and negligible overheads.

2.2 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 five DThreads. The RC values are depicted as shaded values next to the nodes. The DThreads T2, T3 and T4 have one producer, the T1, as such their RC is set to 1. The T5's RC is equal to 2 because it has 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.

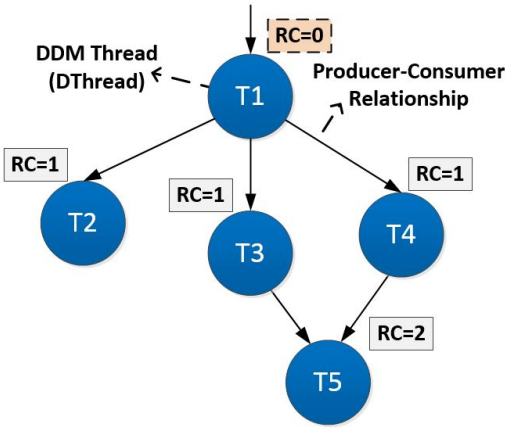

Figure 1: Example of a DDM Dependency Graph.

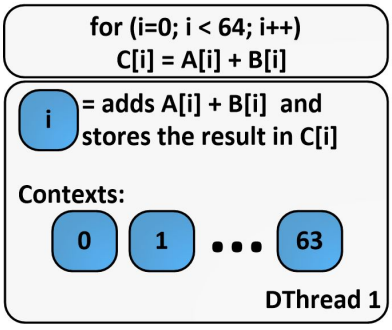

Figure 2: Example of using multiple instances of the same DThread.

2.3 The DDM Tagging System

2.3.1 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 [19] 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 [20].

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. 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 64 times, thus 64 instances are created with Contexts from 0 to 63.

2.3.2 The Nesting attribute

The DDM model allows the parallelization of nested loops that can be mapped into a single DThread by using the Nesting attribute. This attribute is a small number that indicates the loop nesting level for the DThreads that implement loops. In the latest DDM implementations [5, 6, 18] 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. If a DThread does not implement a loop, its Nesting attribute is set to zero (Nesting-0).

The Nesting attribute is used in combination with the Context. 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. An example of one-level loop is depicted in Figure 2 where a Context value holds an index of the loop. In Figure 3 an example of two-level nested loop is shown. Each instance of the DThread will execute the basic block of the nested loops. A Context in this case will include indexes of the inner (in the lower 16-bits) and the outer (in the upper 16-bits) loops. Similarly, an example of a three-level nested loop is shown in Figure 4. An index of the inner loop is stored in the lower 12-bits of a Context value, an index of the middle loop is stored in the bits 12-21 and an index of the outer loop is stored in the upper 10-bits.

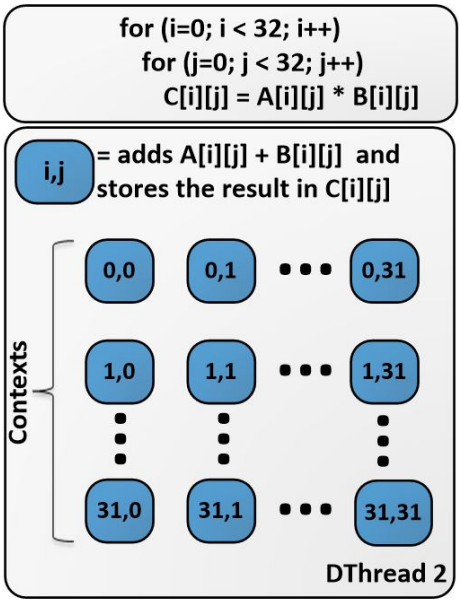

Figure 3: Example of a DThread that supports two-level nested loop.

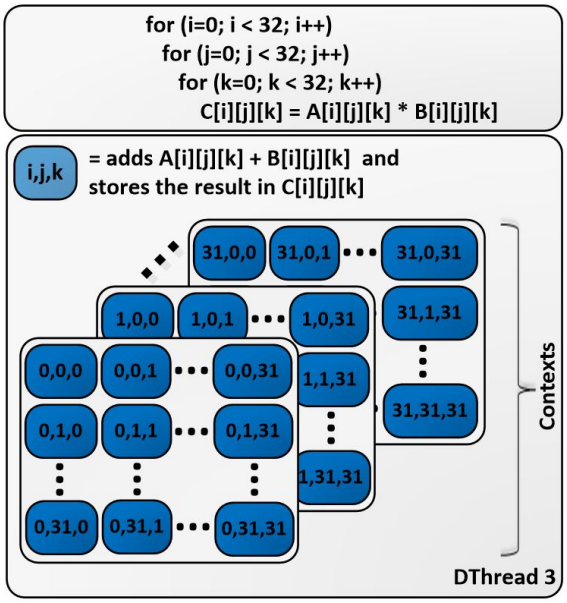

Figure 4: Example of a DThread that supports three-level nested loop.

2.4 The Thread Template

A Thread Template contains the following information:

- **Thread ID (TID):** an identification number that identifies uniquely each 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 Nesting attribute.
- **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.

3 The FREDDO Architecture

FREDDO is a robust implementation of DDM that provides extra functionalities like, support of recursion, larger Context values and better programming interface. FREDDO has been developed

and tested with the same methodology used in commercial software. C++11 was selected as the programming language in order to take advantage of object oriented techniques such as maintainability, re-usability, data abstraction and encapsulation. Also, new features of C++11 are used like range-based loops, Lambda expressions and atomic operations [21]. FREDDO allows efficient DDM execution on multi-core and many-core systems by utilizing three different components: the TSU, the Kernels and the Runtime Support.

3.1 New features and Improvements

3.1.1 Extending the Context Attribute:

In the previous DDM systems, the Context attribute was only 32-bit long. This prohibits the execution of DDM programs that have nested loops with large indexes. For example, consider a three-level nested loop in which the bounds of its outer loop is in range of 0 to 65535. This loop is unable to be parallelized since the indexes of the outer loop are stored in the upper 10-bits of the Context values. In this work we are evaluating the TSU performance by using three different Context sizes: 32-bit, 64-bit and 96-bit. The evaluation shows that we don't have any loss in performance.

3.1.2 Avoid recompiling the TSU code

In the past, all the TSU structures were implemented using static memory allocation in order to increase the performance. This forces the programmer to recompile the TSU code in the following cases:

- When a user wants to change the number of cores that will be utilized by the TSU. In this work, through a new API (Application Programming Interface), we allow the programmers to change the number of cores at runtime. This functionality is vital for the HPC programming.
- When the queues that hold the update commands are full. In this situation the DDM execution was aborted and the user had to manually increase the size of the queues. In FREDDO we avoid this situation, by introducing an auxiliary data structure, the Unlimited Input Queue (UIQ).
- When the data-structure that is used for holding the Ready Count (RC) values, called Synchronization Memory (SM), is full. To solve this issue we allocate a different SM for each DThread.

3.1.3 Improving the DDM programmability

In the previous DDM implementations the programs were implemented using C macros [5, 6]. This approach was very restrictive and complicated since special macros had to be used for: (i) marking the boundaries of the DThreads, (ii) creating the DThreads' IFPs and (iii) extracting the index of the loops from Context values, regarding the Nesting value. In this work, these functionalities are provided automatically without the need of using special macros/functions by the programmers. Our API provides C++ functions and classes that are included in a C++ namespace, called *ddm*. The user is able to create and manage DThreads by just creating and accessing objects of special C++ classes (*SimpleDThread, MultipleDThread, MultipleDThread2D*, etc.). The development process is more efficient which reduces the programming effort.

3.1.4 Supporting Recursion

FREDDO is the first framework that supports recursion in the DDM model. We are providing additional components for this functionality through special C++ template classes. As an example, we show how we can parallelize the recursion implementation of the Fibonacci algorithm in DDM.

3.1.5 Allowing DDM code anywhere

In DDM-VM, two special macros are used to mark the boundary of the DThreads, the *DVM_THREAD_START* and *DVM_THREAD_END*. The former macro is used to identify the first instruction of a DThread by using a unique *label* statement. The latter macro is used to fetch the next ready DThread from the TSU and execute its code. For this purpose, a *goto* statement is used to jump to the label of the next ready DThread.

The C programming language does not permit a jump to a label contained within another function [22]. Thus, the approach of executing DThreads using *label* and *goto* statements is feasible when all DThreads are declared in the same function/place. This restricts the programmers from having parallel code in different files as well as in different functions. FREDDO allows DThreads' code to be embodied in standard C++ functions, in Lambda expressions and in functors. As such, the *label* and *goto* statements are avoided which gives more flexibility to the programmers.

3.1.6 Automatic computation of the RC values

In the previous DDM systems the RC value of each DThread was required to be specified by the programmer. FREDDO uses the consumers of each DThread to compute automatically the RCs. For this purpose, a special data-structure, called Pending Template Memory (PTM), is introduced in TSU.

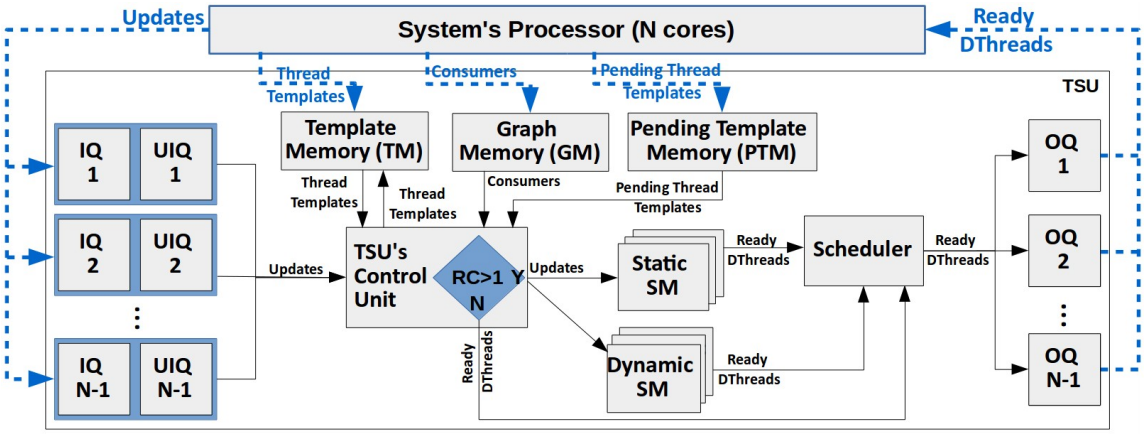

Figure 5: Block diagram of the FREDDO's TSU.

3.2 The Thread Scheduling Unit

The block diagram of the TSU, that supports N cores, is shown in Figure 5. Each block of the diagram is a C++ object that may consist of several internal objects.

3.2.1 The TSU's storage units

The TSU uses four main units for the storage, the Template Memory (TM), the Pending Template Memory (PTM), the Graph Memory (GM) and the Synchronization Memory (SM). The TM contains the Thread Template of each DThread. The PTM contains the Thread Templates that their RC values will be computed at runtime using the consumers of each DThread. The GM contains the consumers of each DThread. The SM contains the Ready Count (RC) values of the different instances of the DThreads. A DThread that implements a loop has multiple instances, one for each iteration. FREDDO supports static and dynamic SM:

- **Static SM:** It's used when the number of instances of a DThread is known at compile time. Each instance allocates a unique entry in the Static SM. The allocation of all RCs is performed

at the time of creating the Thread Template. Accessing an RC entry at runtime is a direct operation that uses the Context of the DThread's instance.

- **Dynamic SM**: It's used when there is no information about the number of instances of a DThread. A hash-map is used to allocate the SM entries where the *keys* are the Contexts and the *values* are the RCs. The allocation of RCs is performed as the execution proceeds. The hash-map is allocated at the time of creating the Thread Template. Moreover, accessing an RC entry is an associative operation.

Instead of having one global SM for each type (static/dynamic), we allocate a different SM for each DThread, for two reasons. Firstly, in the case of the Static SM, we allocate exactly the amount of RCs that is required for each DThread. This is an improvement over the previous DDM implementations. Secondly, in the case of the Dynamic SM, we are using simpler/smaller key-value pairs. For instance, if we had a global Dynamic SM, a possible *key* would be the tuple <Thread ID, Context> instead of <Context>. Furthermore, the rehashing operation of a DThread's Dynamic SM will not cause the rehashing of the other DThreads' Ready Counts.

3.2.2 TSU-Cores Communication

The communication between the TSU and the computation cores is done through the Output Queues (OQs), the Input Queues (IQs) and the Unlimited Input Queues (UIQs). A triplet of IQ, UIQ and OQ is attached in each core. The TSU dispatches the ready DThreads to the cores, through the OQs.

After a core completes the execution of a DThread, it sends update commands to the consumers of the completed DThread. An update consists of the TID and the Context of the DThread that is going to be updated. An update operation indicates that the RC value that corresponds to the TID and Context attributes will be decreased by one. The updates are stored in the core's IQ. If the IQ is full, then the updates are stored in the UIQ. The UIQ is an efficient queue data-structure implemented using a linked-list, thus, it's not fixed-size.

3.2.3 The TSU's Control Unit

The TSU's control unit fetches the updates from the IQs in a round-robin fashion. If an IQ is empty, the TSU fetches updates from the associated UIQ. For each update, it locates the Thread Template of the DThread from the TM and decrements the RC in the SM which is associated with the DThread (Static or Dynamic). If the RC value of any DThread's instance reaches zero, then it is deemed executable and is sent to the Scheduler. An instance of a DThread that is ready for execution it's called Ready DThread and consists of the TID, the IFP and the Context attributes. In our implementation the DThreads that their RC value is equal to one are managed differently. More specifically, they are scheduled immediately without the need of allocating a SM. This approach, decrements the memory usage of DDM applications as well as it accelerates the update operations of DThreads with RC=1.

Before the TSU starts the DDM scheduling, its control unit fetches the Pending Thread Templates from PTM and performs three algorithmic steps:

- **Step 1:** For each Pending Thread Template create an RC value and set it to 0
- **Step 2:** Get the consumers of each DThread from GM
  - For each consumer of each DThread
    1. If the consumer is located in PTM, increase its RC value by one
    2. If the consumer is not located in PTM, ignore it
- **Step 3:** For each Pending Thread Template (PTT) of PTM
  1. If the RC value of the current PTT is zero, set it to 1
  2. Remove the PTT from PTM and store it in TM
  3. If the PTT's RC value > 1, allocate a new SM (static or dynamic)

3.2.4 Scheduling the Ready DThreads

The Scheduler is responsible for assigning the ready DThreads to the Output Queues (OQs). The thread invocations are distributed to the cores in order to achieve load-balancing.

3.2.5 Memory allocation of the TSU's structures

The IQs and OQs are implemented as fixed-size circular buffers that are allocated statically at compile time in order to accelerate the enqueue and dequeue operations. The other TSU's data structures are allocated dynamically at runtime. This avoids the need of recompiling the TSU code in the cases we mentioned in Section 3.1.2.

3.3 The Kernels and the Runtime Support

Our implementation allows data-driven execution on top of any commodity OS, like the previous software DDM implementations [4, 5, 6]. This allows the execution of DDM and non-DDM applications simultaneously. This key-feature is supported by the Kernels and the Runtime system. The overall architecture of the FREDDO framework is depicted in Figure 6.

A Kernel is a POSIX Thread (PThread) that is pinned in a specific core until the end of the DDM execution. This eliminates the overheads of the context-switching between the Kernels in the system. The Kernel is responsible for executing the ready DThreads that are stored in the Output Queue (OQ) of its core. Also, it is responsible for storing the Update commands in its core's IQ/UIQ. In FREDDO, m Kernels are created, where m is the maximum number of DThreads that can be executed in parallel in a system. Usually, m is equal to $N - 1$, where N is the number of cores of the system. This is because one of the cores is reserved for the execution of the TSU code.

The Runtime system enables the communication between the cores and the TSU through the Main Memory. It is responsible for loading the Thread Templates, the Pending Thread Templates and the Consumers of the DThreads, for creating and running the Kernels, and for deallocating the resources allocated by DDM programs. The Runtime, enqueues the Update commands in the IQs/UIQs and it dequeues the ready DThreads from the OQs and forwards them to the Kernels.

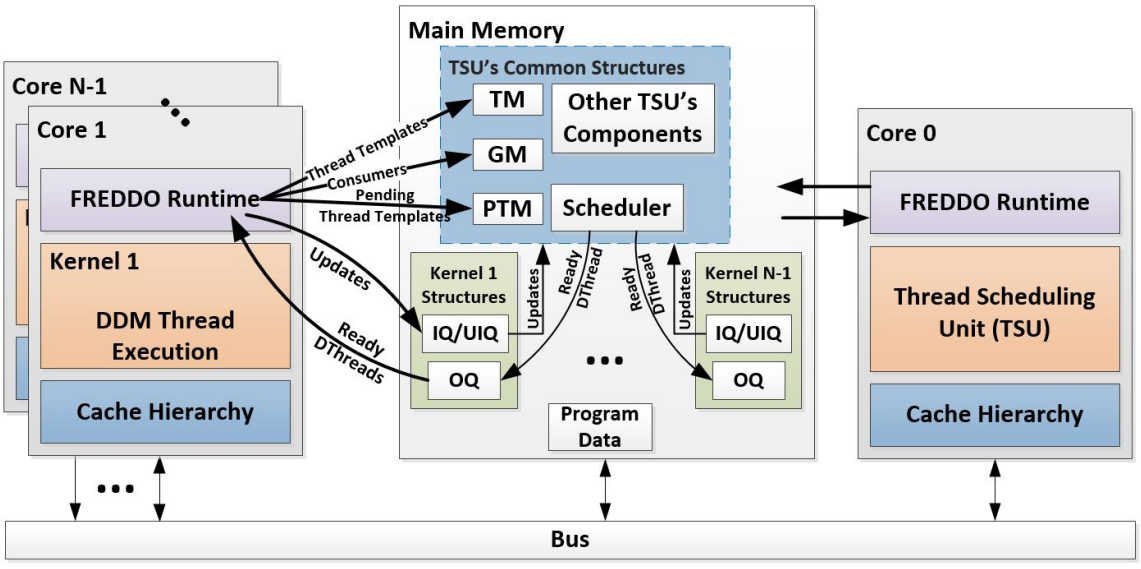

Figure 6: The FREDDO's overall architecture.

4 Programming Methodology

In this work we provide an API (Application Programming Interface) that enables the programmers to develop DDM applications. The API is a C++ library that includes a set of runtime functions and classes which are grouped together in a C++ namespace called *ddm*. The API enables the programmers to manage the DDM execution environment and the Dependency Graph (create and remove DThreads) as well as to perform updates.

4.1 Runtime Functions

The basic runtime functions of FREDDO are the following:

- **void init(unsigned int numKernels)**: initializes the DDM execution environment and it starts numKernels Kernels.
- **void run(void)**: computes the RC values of the pending Thread Templates and starts the scheduling of DThreads. The scheduling will finish if the TSU has no updates to execute (the Input Queues and Unlimited Input Queues are empty) and there are no pending ready DThreads (the Output Queues are empty). All the initial updates have to be sent to the TSU before this command. If this does not happen, the *run* function will finish the scheduling immediately since the TSU has no updates to execute. Thus, every DDM program should have update commands before the *run* function.
- **void finalize()**: stops the Kernels and releases all the resources allocated by FREDDO.

4.2 DFunctions

The code of the DThreads can be embodied in any callable target (called *DFunction*) like: (i) standard C++ functions, (ii) Lambda expressions and (iii) functors. This methodology allows us to have parallel code anywhere in a DDM program, without the need of using *goto* and *label* statements. The DDM Kernels execute the code of the DThreads at runtime. Each *DFunction* has at most one input argument, the *Context*. Four different DFunction types are provided according to the Nesting value of the DThread:

1. **SimpleDFunction**: for DThreads with Nesting-0. It has no argument(s) since the Context value is always zero.
2. **MultipleDFunction**: for DThreads with Nesting-1. It has as input argument, the *Context* data type, which is a single value that contains an index of the one-level loop.
3. **MultipleDFunction2D**: for DThreads with Nesting-2. It has as input argument, the *Context2D* data type. This data type contains indexes of the outer and the inner loops.
4. **MultipleDFunction3D**: for DThreads with Nesting-3. It has as input argument, the *Context3D* data type. This data type contains indexes of the outer, the middle and the inner loops.

4.3 DThread Classes

In FREDDO, we are providing eight special C++ classes that enable creating, updating and removing DThreads with different characteristics. These classes are derived-classes of the *DThread* class. A programmer is able to create and load Thread Templates in the TSU by just using constructors of the special classes. The DThreads are removed from the TSU using the *delete* operator of C++ (in each case the appropriate destructor is called).

4.3.1 DThread Class

The base class of our framework. The user is not able to create objects of this class, i.e. there is no constructor. All the methods of this class are accessible by the special classes mentioned in the next subsections. This class is used only for inheritance and has the following methods:

- **void updateAllCons()**: decrements the RC of the DThread's consumers with Nesting-0. I.e., it decreases the RC that corresponds to the Context=0, for all consumer-DThreads.
- **void updateAllCons(Context context)**: decrements the RC that corresponds to the input context of the DThread's consumers with Nesting-1. For example, *updateAllCons(3)*, decreases the RC value that corresponds to the Context=3, by one, for all consumers of this DThread.
- **void updateAllCons(Context2D context)**: decrements the RC of the DThread's consumers with Nesting-2. For example, *updateAllCons(*$\{2, 3\}$*)*, decreases the RC value that corresponds to the Context with *outer index*=2 and *inner index*=3, by one, for all consumers of this DThread.
- **void updateAllCons(Context3D context)**: decrements the RC of the DThread's consumers with Nesting-3. For example, *updateAllCons(*$\{2, 5, 3\}$*)*, decreases the RC value that corresponds to the Context with *outer index*=2, *middle index*=5 and *inner index*=3, by one, for all consumers of this DThread.
- **void updateAllCons(Context context, Context maxContext)**: decrements the RC of multiple instances of the DThread's consumers with Nesting-1. For example, *updateAllCons(0,13)*, decrements the RCs that correspond to Contexts 0 to 13, of all consumers of this DThread.
- **void updateAllCons(Context2D context, Context2D maxContext)**: decrements the RC of multiple instances of the DThread's consumers with Nesting-2. For example, *updateAllCons(*$\{0, 0\}$*,* $\{0, 4\}$*)*, decrements the RCs that correspond to Contexts $\{0, 0\}$ to $\{0, 4\}$, of all consumers of this DThread. More specifically, the instances with Contexts $\{0, 0\}$, $\{0, 1\}$, $\{0, 2\}$, $\{0, 3\}$ and $\{0, 4\}$ will be updated, for each consumer.
- **void updateAllCons(Context3D context, Context3D maxContext)**: decrements the RC of multiple instances of the DThread's consumers with Nesting-3. For example, *updateAllCons(*$\{0, 0, 1\}$*,* $\{0, 0, 3\}$*)*, decrements the RCs that correspond to Contexts $\{0, 0, 1\}$ to $\{0, 0, 3\}$, of all consumers of this DThread. More specifically, the instances with Contexts $\{0, 0, 1\}$, $\{0, 0, 2\}$ and $\{0, 0, 3\}$ will be updated, for each consumer.
- **unsigned int getTID()**: returns the Thread Identifier (TID) of the DThread. The TID of each DThread is created at runtime by the framework. In the previous DDM implementations, the user has to specify the TID manually.
- **void setConsumers(Consumers consList)**: set the list of consumers. The *consList* variable is a C++ vector that contains DThread pointers (DThread*).

4.3.2 SimpleDThread Class

SimpleDThread is a DThread with Nesting-0 and only one instance with Context=0. The user has to specify the RC value. It has only one constructor and only one method:

- **SimpleDThread(SimpleDFunction sDFunction, ReadyCount readyCount)**: inserts a SimpleDThread in the TSU. The user has to specify the DFunction and the RC value.
- **void update()**: decrements the RC value of this DThread.

4.3.3 MultipleDThread Class

MultipleDThread is a DThread with Nesting-1 and multiple instances. The user has to specify the RC value. If the number of instances is specified a Static SM will be allocated, otherwise a Dynamic SM will be allocated. It has the following constructors and methods:

- **MultipleDThread(MultipleDFunction mDFunction, ReadyCount readyCount, UInt numOfInstances)**: inserts a MultipleDThread in the TSU where a static SM will be used. The user has to specify the DFunction, the RC value and the number of instances of the DThread.
- **MultipleDThread(MultipleDFunction mDFunction, ReadyCount readyCount)**: like the previous constructor but the number of instances is not specified. Thus, a Dynamic SM will be allocated.
- **void update(Context context)**: decrements a RC value of this DThread.
- **void update(Context context, Context maxContext)**: decrements the RC value of multiple instances of this DThread.

4.3.4 MultipleDThread2D Class

Similar with MultipleDThread, but it has Nesting-2. It has the following constructors and methods:

- **MultipleDThread2D(MultipleDFunction2D mDFunction2D, ReadyCount readyCount, UInt innerRange, UInt outerRange)**: inserts a MultipleDThread2D in the TSU where a static SM will be used. The user has to specify the DFunction, the RC value and the number of instances of the DThread. *outerRange* indicates the dimension of the outer-level loop where *innerRange* indicates the dimension of the inner-level loop. For example, if *outerRange*=4 and *innerRange*=3, then a Static SM (as a 4x3 matrix) will be allocated with 12 RC entries.
- **MultipleDThread2D(MultipleDFunction2D mDFunction2D, ReadyCount readyCount)**: like the previous constructor but the dimensions are not specified. Thus, a Dynamic SM will be allocated.
- **void update(Context2D context)**: decrements a RC value of this DThread.
- **void update(Context2D context, Context2D maxContext)**: decrements the RC value of multiple instances of this DThread.

4.3.5 MultipleDThread3D Class

Similar with MultipleDThread, but it has Nesting-3. It has the following constructors and methods:

- **MultipleDThread3D(MultipleDFunction3D mDFunction3D, ReadyCount readyCount, UInt innerRange, UInt middleRange, UInt outerRange)**: inserts a MultipleDThread3D in the TSU where a static SM will be used. The user has to specify the DFunction, the RC value and the number of instances of the DThread. *outerRange* indicates the dimension of the outer-level loop, *middleRange* indicates the dimension of the middle-level loop and *innerRange* indicates the dimension of the inner-level loop. For example, if *outerRange*=4, *middleRange*=3 and *innerRange*=2, then a Static SM (as a 4x3x2 matrix) will be allocated with 24 RC entries.
- **MultipleDThread3D(MultipleDFunction3D mDFunction3D, ReadyCount readyCount)**: like the previous constructor but the dimensions are not specified. Thus, a Dynamic SM will be allocated.
- **void update(Context3D context)**: decrements a RC value of this DThread.

- **void update(Context3D context, Context3D maxContext)**: decrements the RC value of multiple instances of this DThread.

4.3.6 FutureDThread Classes

FutureDThread Classes are derived-classes of the "Regular" DThreads (SimpleDThread, MultipleDThread, MultipleDThread2D and MultipleDThread3D). They have the same constructors and methods with the "Regular" DThreads. The only difference is that the user does not specify the RC value in the constructors. As a result, their RC value will be evaluated at runtime (initially they will be stored in Pending Template Memory) using the producer-consumer relationships of the program. The following FutureDThread Classes are provided:

- **FutureSimpleDThread Class**: a derived-class of SimpleDThread.
- **FutureMultipleDThread Class**: a derived-class of MultipleDThread.
- **FutureMultipleDThread2D Class**: a derived-class of MultipleDThread2D.
- **FutureMultipleDThread3D Class**: a derived-class of MultipleDThread3D.

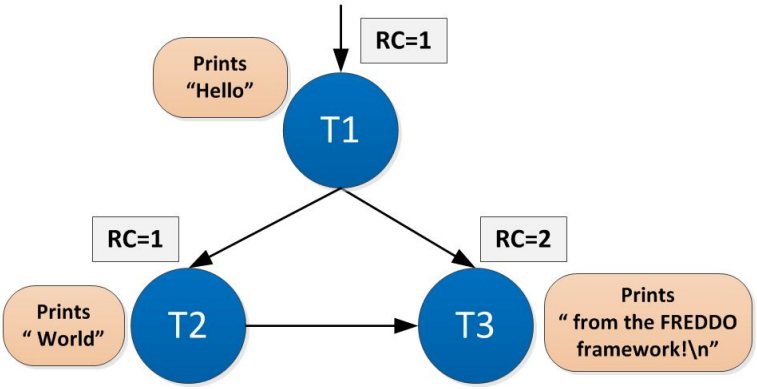

Figure 7: The DDM Dependency Graph of a simple application.

4.4 A simple programming example

In this section we present the mapping of a simple application to a DDM program. The Dependency Graph of the application is shown in Figure 7 which is composed of three DThreads. The RC values are depicted as shaded values next to the nodes. The rounded rectangles illustrate the functionalities of the DThreads. The DThread T1 has two consumers, which are the DThreads T2 and T3, and it prints the string "Hello". The DThread 2 has one consumer, the T3, and it prints the string " World". The DThread T3 prints the string " from the FREDDO framework!\n". Thus, the DDM program will output the result: *Hello World from the FREDDO framework!*.

Listing 1 depicts the DDM code of the simple example. For this example we are using *Future DThreads*. The codes of DThreads t1 and t2 are embodied in Lambda expressions where the code of DThread t3 is embodied in a standard C++ function. All DThreads have Nesting-0 since they don't implement loops or recursion, thus are type of *FutureSimpleDThread*. An alternative solution is to use *SimpleDThreads*. In that case, we have to specify the RC value of each DThread manually. In line 31, an initial update is sent to DThread t1, since it's the root of the DDM dependency graph (it has no producers). Listing 2 depicts the DDM code of the same example. In this case, we are using *SimpleDThreads* and the code of each DThread is embodied in a standard C++ function.

```
#include <iostream>
#include <future_dthreads.h>
using namespace std;
using namespace ddm;

void t3_code(){ // The t3's code
 cout << " from the FREDDO framework!\n";
}

void main(){
 ddm::init(NUM_KERNELS); // Initializes the DDM execution environment

 FutureSimpleDThread *t1, *t2, *t3; // The DThread objects

 t1 = new FutureSimpleDThread([&] (){ // The t1's code
  cout << "Hello";
  t1->updateAllCons();
 });

 t2 = new FutureSimpleDThread([&] (){ // The t2's code
  cout << " World";
  t2->updateAllCons();
 });

 t3 = new FutureSimpleDThread(t3_code);

 // Specify the Consumers
 t1->setConsumers({t2, t3});
 t2->setConsumers({t3});

 t1->update();  // Initial Update
 ddm::run(); // Start the DDM scheduling
 delete t1; delete t2; delete t3; // Remove the DThreads
 ddm::finalize(); // Stops the Kernels and releases the resources
}
```

Listing 1: A simple DDM example: solution with FutureSimpleDThreads.

```
// Includes goes here ...
#include <dthreads.h>
using namespace ddm;

SimpleDThread *t1, *t2, *t3; // The DThread objects

void t1_code(){ // The t1's code
 cout << "Hello";
 t2->update(); // Update t2 DThread
 t3->update(); // Update t3 DThread
}

void t2_code(){ // The t2's code
 cout << " World";
 t3->update(); // Update t3 DThread
}

void t3_code(){ // The t3's code
 cout << " from the FREDDO framework!\n";
}

void main(){
 ddm::init(NUM_KERNELS); // Initializes the DDM execution environment

 t1 = new SimpleDThread(t1_code, 1);
 t2 = new SimpleDThread(t2_code, 1);
 t3 = new SimpleDThread(t3_code, 2);

 t1->update(); // Initial Update
```

```
30  ddm::run(); // Start the DDM scheduling
31
32  // Remove the DThreads
33  delete t1;
34  delete t2;
35  delete t3;
36  ddm::finalize(); // Stops the Kernels and releases the resources
37 }
```

Listing 2: A simple DDM example: solution with SimpleDThreads.

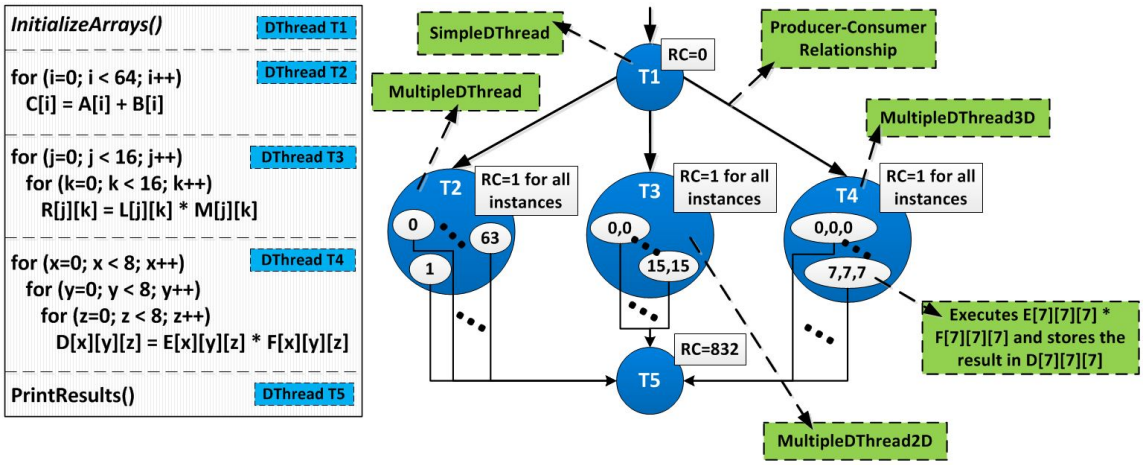

Figure 8: Example of a DDM program with different for-loops.

4.5 A DDM example with different for-loops

An example of a DDM program with different for-loops is shown in Figure 8. On the left side of the figure, the pseudo-code of a synthetic application and its partitioning into five DThreads are depicted. From this code it is possible to observe a number of dependencies. DThreads T2, T3 and T4 depend on T1 which is responsible for initializing the data. Also, T5 depends on T2, T3 and T4 since T5 prints the output results generated by them. These dependencies form the DDM Dependency Graph of this application which is presented on the right side of Figure 8. The RC values are depicted as shaded values next to the nodes.

The three for-loop blocks are fully parallel and are mapped into three different DThreads. Each instance of a DThread is identified by the Context and it executes the inner command of the block. T2 has 64 instances (with Contexts from 0 to 63), T3 has 256 instances (with Contexts from 0,0 to 15,15) and T4 has 512 instances (with Contexts from 0,0,0 to 7,7,7). DThread T5 depends on all the instances of DThreads T2-T4, thus its RC is equal to 832. Moreover, the instances of DThreads T2-T4 have RC=1 because they have only one producer (T1). When T1 finishes its execution a *Multiple Update* will be sent in each consumer-thread. As a result, all the instances of DThreads T2-T4 will be executed concurrently.

Listing 3 depicts one possible implementation of the algorithm. In this example, we place the DThreads' code in standard C++ functions (for T1 and T4) and in Lambda expressions (for T2, T3 and T5). Furthermore, we are using a combination of *Regular DThreads* (T3 and T5) and *Future DThreads* (T1, T2 and T4).

```
// Includes goes here ...
#include <dthreads.h>
#include <future_dthreads.h>
using namespace ddm;

// Declate DThread objects
FutureSimpleDThread *t1;
FutureMultipleDThread *t2;
MultipleDThread2D *t3;
FutureMultipleDThread3D *t4;
SimpleDThread *t5;

// Declare Global Variables (Arrays, etc.)

void t1_code(){ // The t1's code
 // Initializing Arrays ...

 // Update the instances of consumers
 t2->update(0, 63); // Multiple Update
 t3->update({0,0}, {15,15}); // Multiple Update
 t4->update({0,0,0}, {7,7,7}); // Multiple Update
}

void t4_code(Context3D c){ // The t4's code
 auto x = c.Outer, y = c.Middle, z = c.Inner;
 D[x][y][z] = E[x][y][z] * F[x][y][z];
 t4->updateAllCons();
}

void main(){
 // Initializiations goes here ...
 ddm::init(NUM_KERNELS);

 // DTheads declarations using standard functions
 t1 = new FutureSimpleDThread(t1_code);
 t4 = new FutureMultipleDThread3D(t4_code);

 // DTheads declarations using Lambda expressions
 t2 = new FutureMultipleDThread([&](Context cntx){ // The t2's code
  C[cntx] = A[cntx] + B[cntx];
  t5->update();
 });

 t3 = new MultipleDThread2D([&](Context2D cntx){ // The t3's code
  auto j = cntx.Outer, k = cntx.Inner;
  R[j][k] = L[j][k] * M[j][k];
  t5->update();
 }, 1); // 1 at this point is the RC value

 t5 = new SimpleDThread([&](){ // The t5's code
  // Print Results ...
 }, 832); // 832 at this point is the RC value

 // Set the consumers of each DThread
 t1->setConsumers({t2, t3, t4}); t2->setConsumers({t5});
 t3->setConsumers({t5}); t4->setConsumers({t5});

 t1->update(); // Decrease the RC of T1
 ddm::run(); // Start the DDM scheduling
 delete t1; ...; delete t5;
 ddm::finalize(); // Deallocate Resources
}
```

Listing 3: DDM code for an application with different for-loops.

4.6 A complex example - The Blocked LU Decomposition

In this section we present the mapping of a complex application, the blocked LU decomposition, to a DDM program. The code of the original program is shown at the bottom-right corner of Figure 9. The code is composed of five nested loops that perform four basic operations on a blocked matrix: *diag*, *front*, *down* and *comb*.

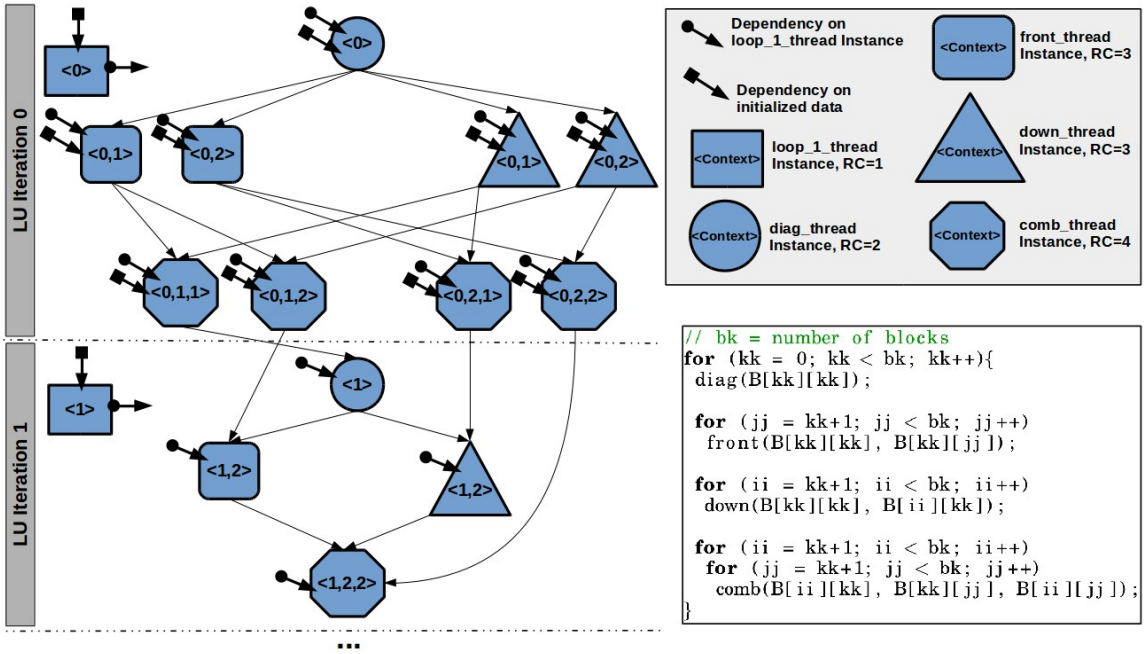

Figure 9: The Blocked LU Decomposition and its DDM Dependency Graph.

In every iteration of the outermost loop, the *diag* operation takes as input the diagonal block that corresponds to the iteration number to produce its new value. The *front* operation produces the remaining blocks on the same row as the diagonal block. For each one of those blocks, it takes as input the result of the *diag* in addition to the current block to produce its new value. Similarly, the *down* operation produces the remaining blocks on the same column as the diagonal block. The *comb* operation produces the rest of the blocks for that LU iteration. For every block it produces, it takes as input three blocks: the current block, the block produced by the *front* operation and the block produced by the *down* operation. It multiplies the second and third blocks and adds the result to the first block to produce the final resulting block. This computational pattern is repeated in the next LU iteration on a subset of the resulting matrix that excludes the first row and column and continues for as much iterations as the diagonal blocks of the matrix.

4.6.1 The DDM Dependency Graph

The loops implementing the control-flow in the original application are mapped into five DThreads, called *loop_1_thread*, *diag_thread*, *front_thread*, *down_thread* and *comb_thread*. The first DThread implements the outermost loop of the algorithm while the other DThreads are responsible for executing the four operations. The following data dependencies are observed:

1. The DThreads that execute the operations depend on the *loop_1_thread* since the index of the outermost loop is used in the four operations.
2. The *front_thread* and *down_thread* DThreads depend on the *diag_thread*.
3. The *comb_thread* depends on the *front_thread* and *down_thread* DThreads.

4. The next LU iteration depends on the results of the previous iteration. As a result, the instances of the DThreads that execute the operations in the next iteration, depend on the instances of those DThreads, from the previous iteration.

The Dependency Graph shown in Figure 9 illustrates the dependencies among the instances of the DThreads for the first two iterations of LU. For simplicity, we show the operations on only two blocks of the matrix. Each DThread's instance is labeled with the value of its Context.

```
1  // Includes, Function prototypes and Global Variables goes here ...
2  #include <dthreads.h>
3  #include <future_dthreads.h>
4  using namespace ddm; // Use the ddm name-space
5
6  void main(){
7   // Initializiations goes here ...
8
9   ddm::init(NUM_KERNELS);  // Initializes the DDM execution environment
10
11  // The DThreads' objects
12  FutureMultipleDThread *loop_1DT, *diagDT;
13  FutureMultipleDThread2D *frontDT, *downDT;
14  FutureMultipleDThread3D *combDT;
15
16  /////////// DTheads declarations using Lambda expressions /////////
17
18  // loop_1_thread
19  loop_1DT = new FutureMultipleDThread([&](Context kk) {
20   diagDT->update(kk);
21
22   if (kk < bk - 1) {
23    frontDT->update({kk, kk + 1}, {kk, bk - 1});
24    downDT->update({kk, kk + 1}, {kk, bk - 1});
25    combDT->update({kk, kk + 1, kk + 1}, {kk, bk - 1, bk - 1});
26   }
27  });
28
29  // diag_thread
30  diagDT = new FutureMultipleDThread([&](Context kk) {
31   diag(A[kk][kk]); // Execute Diag operation
32
33   if (kk < (bk - 1))
34    diagDT->updateAllCons({kk, kk + 1}, {kk, bk - 1});
35  });
36
37  // front_thread
38  frontDT = new FutureMultipleDThread2D([&](Context2D context) {
39   auto kk = context.Outer, jj = context.Inner;
40   front(A[kk][kk], A[kk][jj]); // Execute Front operation
41   combDT->update({kk, kk + 1, jj}, {kk, bk - 1, jj});
42  });
43
44  // down_thread
45  downDT = new FutureMultipleDThread2D([&](Context2D context)  {
46   auto kk = context.Outer, jj = context.Inner;
47   down(A[kk][kk], A[jj][kk]); // Execute Down operation
48   combDT->update({kk, jj, kk + 1}, {kk, jj, bk - 1});
49  });
50
51  // comb_thread
52  combDT = new FutureMultipleDThread3D([&](Context3D context)  {
53   auto kk = context.Outer, ii = context.Middle, jj = context.Inner;
54   comb(A[ii][kk], A[kk][jj], A[ii][jj]); // Execute Comb operation
55
56   if (ii == kk+1 && jj == kk+1)
57    diagDT->update(kk + 1);
58   else if (ii == kk+1)
59    frontDT->update({ii, jj});
```

```
60      else if (jj == kk+1)
61       downDT->update({jj, ii});
62      else
63       combDT->update({kk + 1, ii, jj});
64    });
65
66    // Set the consumers of each DThread
67    loop_1DT->setConsumers({diagDT, frontDT, downDT, combDT});
68    diagDT->setConsumers({frontDT, downDT});
69    frontDT->setConsumers({combDT});
70    downDT->setConsumers({combDT});
71    combDT->setConsumers({diagDT, frontDT, downDT, combDT});
72
73    // Updates resulting from data initialization
74    loop_1DT->update(0, bk - 1);
75    diagDT->update(0);
76    frontDT->update({0, 1}, {0, bk - 1});
77    downDT->update({0, 1}, {0, bk - 1});
78    combDT->update({0, 1, 1}, {0, bk - 1, bk - 1});
79
80    ddm::run(); // Start the DDM scheduling
81    // Remove the DThreads
82    delete loop_1DT; delete diagDT; delete frontDT; delete downDT; delete combDT;
83    ddm::finalize(); // Stops the Kernels and releases the resources
84  }
```

Listing 4: DDM code of the LU benchmark.

4.6.2 The DDM code

Listing 4 depicts the DDM implementation of the algorithm. In this example, we show how the LU decomposition can be implemented using Lambda expressions and *Future DThreads*. For each DThread object we specify only the DFunction. Thus, the Dynamic SM will be used. Each call of an update function corresponds to one dependency arrow in Figure 9. Moreover, the update functions at the end of the *comb_thread* implement a switch actor (lines 56-63), which depending on the Context of the instance, it updates a different consumer. After the DThreads' declarations, the consumers of each DThread are defined, using the *setConsumers* function (lines 67-71). The initial updates are sent to the TSU (lines 74-78). These updates correspond to the arrows of Figure 9 that describe dependencies on initialized data.

4.7 An example with Reduction

Reduction refers to the process of combining the results of several sub-calculations into a final result. FREDDO allows parallel reduction by utilizing DThreads. As a proof-of-concept we show how we can implement parallel reduction for the vector dot-product algorithm (Listing 5). The serial implementation of the algorithm is defined in lines 9-16. The *sum* variable holds the dot-product of two arrays, *a* and *b*. For the parallel implementation we are using one *MultipleDThread*, the *sum_dt*, and one *SimpleDThread*, the *red_dt*. Each instance of *sum_dt* computes the dot-product of a subset of the two arrays (*a* and *b*) and it stores the result in a global array, the *sumPerInstance*. Each instance owns a unique entry in *sumPerInstance* which is accessed by its Context value. The workload is divided equally in the instances of *sum_dt*. This is done by the *distributeWorkLoad* function. This auxiliary function distributes the workload based on the Context of each instance. For example, if the arrays have 100 elements and *sum_dt* has two instances with Contexts 0 and 1, then: the instance with Context=0 will calculate the dot-product of the first 50 elements (the indexes 0-49) and the instance with Context=1 will calculate the dot-product of the last 50 elements (the indexes 50-99).

red_dt is responsible for summing the results calculated by the instances of *sum_dt*. Thus, *red_dt* is a consumer of *sum_dt* (line 53). The *red_dt* DThread has to wait all the instances of *sum_dt* to finish their execution. As such, the RC value of *red_dt* is equal to the number of instances of *sum_dt*

(NUM_INST). Moreover, *sum_dt* has RC=1 and NUM_INST instances. The update command in line 54 is responsible for spawning all the instances of *sum_dt* in parallel.

```
1  #include <dthreads.h>
2  using namespace ddm;
3
4  // Gloabal Variables
5  float a[SIZE], b[SIZE];
6  float sumPerInstance[NUM_INST]; // Holds the sum of each instance
7
8  // Serial Implementation
9  int serial(){
10  float sum = 0.0;
11
12  for (unsigned int i = 0; i < SIZE; i++)
13   sum = sum + (a[i] * b[i]);
14
15  printf("Serial Result = %f\n", sum);
16 }
17
18 // DThreads
19 MultipleDThread *sum_dt;
20 SimpleDThread *red_dt;
21
22 void sum_code(Context c) { // The summing code
23  float mySum = 0.0;
24  unsigned int beg, end;
25
26  distributeWorkLoad(c, SIZE, NUM_INST, &beg, &end);
27
28  for (unsigned int i = beg; i < end; i++)
29   mySum = mySum + (a[i] * b[i]);
30
31  sumPerInstance[c] = mySum;
32  sum_dt->updateAllCons(); // Update the consumers, i.e. the red_dt
33 }
34
35 void reduction_code(){ // The reduction code
36  float sum = 0.0;
37  for (unsigned int i = 0; i < NUM_INST; i++)
38   sum += sumPerInstance[i];
39
40  printf("Parallel Result = %f\n", sum);
41 }
42
43 int main() {
44  init_arrays(); // Initialize arrays
45  serial(); // Execute serial implementation
46
47  ddm::init(NUM_KERNELS); // Initialize FREDDO envrinoment
48
49  // Create the DThreads
50  sum_dt = new MultipleDThread(sum_code, 1, NUM_INST);
51  red_dt = new SimpleDThread(reduction_code, NUM_INST);
52
53  sum_dt->setConsumers({red_dt}); // Create dependency
54  sum_dt->update(0, NUM_INST - 1); // Spawn the instances
55  ddm::run(); // Start the scheduling
56  // Other code follows ...
57 }
```

Listing 5: Parallel Reduction (Vector Dot-Product).

4.8 Recursion Support

The envelopment of DThreads' code in DFunctions enables us to create parallel DThreads for function calls. This is useful for supporting recursion in the DDM model. FREDDO is the first DDM implementation that supports recursion. We show how we can parallelize a recursive function by using the Fibonacci function:

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

Although this function it's not an efficient implementation of the Fibonacci algorithm, it has the complexity of double recursion. This allows a detailed examination of the functionalities needed for supporting recursion in DDM. In the Fibonacci example, in the case of $n > 2$, each recursive call (parent-call) spawns two additional recursive calls (children-calls) and it returns the sum of their results ($fib(n-1) + fib(n-2)$). This implies the need of some kind of synchronization between the recursion calls since the parent-call waits the return values of its children-calls. In order to implement the Fibonacci in DDM, the following functionalities are required:

1. Mechanisms for creating/updating/deleting DThreads that support recursion.
2. A mechanism that will allow the spawning of recursive function calls.
3. A mechanism that will allow synchronization between the recursive function calls.
4. A mechanism that will allow DThreads to behave as regular C/C++ functions, i.e. to have an argument list and a return type. Notice that, in FREDDO, the functions that contain the DThreads' code have at most one input argument (the *Context*) and the same return type (*void*).

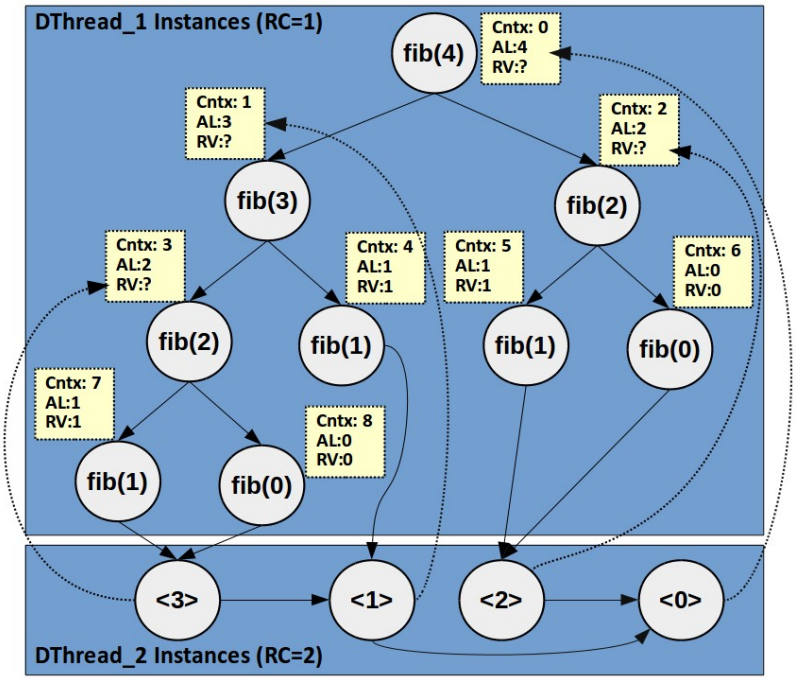

Figure 10: Fibonacci's DDM Dependency Graph.

4.8.1 The DDM Dependency Graph

Figure 10 depicts the Dependency Graph of the Fibonacci algorithm with $n = 4$. The graph consists of two DThreads, the DThread_1 (or Recursive-DThread) and DThread_2 (or Continuation-DThread). The DThread_1's instances are responsible for executing the recursive function calls of the Fibonacci algorithm. Each DThread_1's instance has a unique Context and two additional attributes, the Argument-List (AL) and the Return-Value (RV). The AL attribute holds the arguments of each recursive instance (in this example the AL contains a single value, the n). The RV attribute holds the return value of each recursive instance. In Figure 10, the solid arrows indicate update operations and the dotted arrows indicate write operations to the RV attribute of the DThread_1's instances. In DDM the threads are non-blocking. As such, the DDM does not provide any kind of synchronization between the instances of DThreads. In the Fibonacci example, we need synchronization between a parent instance and its children instances because the parent waits for the results of its children.

We are providing synchronization by introducing a helper/continuation DThread. Each instance of this DThread corresponds to one parent-call and its RC is equal to the number of its children-calls (RC=2 for double recursion). When a child-call returns, it updates the continuation instance that corresponds to its parent-call. When the RC of a continuation instance becomes zero, it means that its associated parent-call can continue its execution. In Figure 10 the continuation DThread is the DThread_2. Each instance of DThread_2 is responsible for calculating and storing the sum of the RV attribute of two children-calls to the RV of their parent-call.

4.8.2 DThread Classes for Recursion Support

FREDDO provides three different classes for recursion support, the *RecursiveDThreadWithContinuation*, the *RecursiveDThread* and the *ContinuationDThread*.

RecursiveDThreadWithContinuation Class: This special template class provides functionalities for algorithms with multiple recursion. It has two template parameters, the *T_ARGS* and the *T_RETURN*. *T_ARGS* indicates the type of the argument(s) of each recursive call. If a recursive function has more than one argument in its argument list, a struct can be used for holding them. *T_RETURN* indicates the type of the return value.

RecursiveDThreadWithContinuation is responsible for creating and managing a Recursive-DThread and a Continuation-DThread. These DThreads are MultipleDThreads with additional functionalities. The provided class holds the AL and RV attributes of each recursive instance. Also, it assigns unique Contexts to the instances of the Recursive-DThread and the Continuation-DThread. The AL and RV attributes of all instances are stored in a dynamically allocated array (in the heap section of the program). Thus, this special class is used when the number of instances of the recursive function is known at compile-time. Additionally, *RecursiveDThreadWithContinuation* has a constructor and several methods:

- **RecursiveDThreadWithContinuation(MultipleDFunction dFunction, UInt maxNumInstances, MultipleDFunction rFunction, UInt numOfChildren)**: inserts a DThread in the TSU that implements multiple recursion. The user has to specify the DFunctions of the recursive code (*dFunction*) and the continuation code (*rFunction*). Also, the maximum number of instances (*maxNumInstances*) of the DThreads has to be specified. In the Fibonacci example this number is equal to 2^n. Finally, the user has to specify the maximum number of children (*numOfChildren*) of a parent. For example, in a double recursion algorithm, like Fibonacci, the maximum number of children is 2.
- **T_ARGS* getArguments(RInstance instance)**: returns the Argument List (AL) of a specific recursive instance. In this function, as well as, in the functions below, *RInstance* is actually the Context of an instance.
- **T_RETURN getReturnValue(RInstance instance)**: returns the Return Value (RV) of a specific instance.

- **T_RETURN getRootReturnValue()**: returns the Return Value (RV) of the root instance.
- **void callChild(RInstance parentInstance, T_ARGS& args)**: spawns a child-instance, where *parentInstance* is the child's parent-instance and *args* is the child's Argument List (AL).
- **void callRoot(T_ARGS& args)**: spawns the root recursive call with its arguments.
- **vector<RInstance>& getMyChilds(RInstance instance)**: returns the instances (Contexts) of children of a specific instance.
- **void returnValueToParent(RInstance instance, T_RETURN value)**: returns the value of a child (instance) to its parent. Also, it updates the continuation instance of the parent-call.
- **void updateContinuationInstance(RInstance instance)**: updates an instance of the Continuation-DThread directly.

4.8.3 Fibonacci implementation using the *RecursiveDThreadWithContinuation* Class

The DDM implementation of the Fibonacci algorithm is shown in Listing 6.

```
1  #include <recursive_dthreads.h>
2  using namespace ddm;
3  RecursiveDThreadWithContinuation<int, int> *fib_dt;
4
5  void fib_code(RInstance context) { // The Fibonacci Code
6   int n = *(fib_dt->getArguments(context)); // Get the arguments (n)
7
8   if (n == 0 || n == 1) {
9    fib_dt->returnValueToParent(context, n); // Return n
10   return;
11  }
12
13  int n1 = n-1, n2 = n-2;
14  fib_dt->callChild(context, n1); // Call fib (n-1)
15  fib_dt->callChild(context, n2); // Call fib (n-2)
16 }
17
18 void continuation_code(RInstance context) {  // The Continuation Code
19  int result = 0; // Holds the sum of the Return Values of the children-instances
20
21  // Get the Return Value of each child and add it to the result variable
22  for (auto i : fib_dt->getMyChilds(context))
23   result += fib_dt->getReturnValue(i);
24
25  fib_dt->returnValueToParent(context, result); // fib(n) = fib(n-1) + fib(n-2)
26 }
27
28 // The main program
29 void main(int argc, char* argv[]){  // Initializiations goes here ...
30  ddm::init(NUM_KERNELS); // Initializes the DDM execution environment
31  int maxInstances = pow(2, n);
32
33  // Create the DThread for recursion support
34  fib_dt = new RecursiveDThreadWithContinuation<int, int>(fib_code, maxInstances,
        continuation_code, 2);
35
36  fib_dt->callRoot(n); // Call the root instance, i.e. the instance with Context=0
37  ddm::run(); // Start DDM scheduling
38  cout << fib_dt->getRootReturnValue() << endl; // Print the result
39  delete fib_dt;
40 }
```

Listing 6: The Fibonacci algorithm implemented in FREDDO.

4.8.4 The *RecursiveDThread* and *ContinuationDThread* Classes

For recursive functions that their number of instances is not known at compile time, we are providing two special classes: the *RecursiveDThread* and the *ContinuationDThread*. These classes allocate/deallocate the arguments and the return values of the instances at runtime. A DDM user can use the *RecursiveDThread* along with the *ContinuationDThread* to implement an algorithm with multiple recursion (or any similar algorithm). Also, the *RecursiveDThread* can be used standalone for other types of recursion, such as linear, tail, and so on. The functionalities of these classes are under development, thus, we are not providing any example in this report.

5 Experimental Results

5.1 Experimental Setup

To evaluate FREDDO we have used an HP server machine with 2 AMD Opteron 6276 processors running at 1.4GHz that supports 32 threads. Each processor is an 8-core 64-bit Clustered Multi-Threaded (CMT) architecture with the capacity of running 16 threads simultaneously. Each core has a 16KB 4-way set associative L1 data cache, a 64K 2-way set associative L1 instruction cache and a 2MB 16-way set associative L2 cache. Also, each processor utilizes a 6MB 64-way set associative L3 cache. The server is equipped with a shared 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. Moreover, the server runs the Ubuntu 14.04 OS (server edition).

Our benchmarks suite has nine applications. Figure 11 illustrates the characteristics of all the benchmarks. The problem sizes are separated into six categories: *Tiny, Small, Medium, Large, XLarge and XXLarge*. The execution time measurements were collected using the *gettimeofday* system call.

Benchmark	Description	Granularity	Problem Sizes					
			Tiny	Small	Medium	Large	XLarge	XXLarge
BMMULT	Blocked Matrix Multiplication	32x32 block	512x512	1Kx1K	2Kx2K	4Kx4K	6Kx6K	8Kx8K
LU	Blocked LU Decomposition	32x32 block	512x512	1Kx1K	2Kx2K	4Kx4K	6Kx6K	8Kx8K
Cholesky	Blocked Cholesky Factorization	64x64 block	512x512	1Kx1K	2Kx2K	4Kx4K	6Kx6K	8Kx8K
Blackscholes	Financial analysis application	variable	4 options	16 options	4K options	16K options	64K options	10M options
Conv2D	9x9 convolution filter	32x32 block	512x512	1Kx1K	2Kx2K	4Kx4K	6Kx6K	8Kx8K
Swaptions	Financial analysis application	102400 simulations	128 Swaptions	256 Swaptions	512 Swaptions	1024 Swaptions	2048 Swaptions	4096 Swaptions
NQueens	Recursive implementation of the NQueens problem	variable	N=11	N=12	N=13	N=14	N=15	N=16
Trapez	Trapezoidal rule for integration	variable	2^{26} steps	2^{27} steps	2^{28} steps	2^{29} steps	2^{30} steps	2^{31} steps
Fibonacci	Recursive implementation of the Fibonacci algorithm	variable	N=5	N=10	N=15	N=20	N=25	N=30

Figure 11: The benchmark suite characteristics.

	BMMULT	LU	Cholesky	Blackscholes	Conv2D	Swaptions	NQueens	Trapez	Fibonacci
Tiny	0.53	0.16	0.03	7.42E-04	0.37	141.92	0.04	0.24	1.55E-06
Small	4.26	1.26	0.28	2.84E-03	1.47	283.63	0.24	0.47	3.22E-06
Medium	34.07	10.11	2.23	0.53	5.88	567.23	1.48	0.94	1.45E-05
Large	273.34	80.91	17.94	2.11	23.43	1134.54	9.66	1.88	9.40E-05
XLarge	922.35	274.02	60.71	8.55	52.94	2268.53	65.76	3.76	8.52E-04
XXLarge	2190.00	647.41	143.94	1359.60	93.72	4537.16	469.75	7.52	1.08E-02

Figure 12: Sequential execution time (in seconds) for all problem sizes of each benchmark.

For the performance evaluation, all the experimental results are reported as speedups. 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, without any FREDDO overheads. For example, if a certain parallel application has speedup equal to 30 this means that it's 30 times faster than its sequential application. Figure 12 depicts the sequential execution time (in seconds) of each problem size of all benchmarks.

5.2 Experimenting with the Context size

In this section we are evaluating three different Context configurations: 32-bit (the one that was used in the previous DDM implementations), 64-bit and 96-bit. Figure 13 describes how the indexes of the loops are encoded into the Context value for all possible combinations, for each Context size. Notice that for the Nesting-0 the Context value is always zero and its width is equal to the Context size.

	32-bit Context Value			64-bit Context Value			96-bit Context Value		
Nesting Attribute	**Outer**	**Middle**	**Inner**	**Outer**	**Middle**	**Inner**	**Outer**	**Middle**	**Inner**
Nesting-1	*Unused*	*Unused*	32-bits	*Unused*	*Unused*	64-bits	*Unused*	*Unused*	32-bit
Nesting-2	16-bits	*Unused*	16-bits	32-bits	*Unused*	32-bits	32-bit	*Unused*	32-bit
Nesting-3	10-bits	10-bits	12-bits	16-bits	16-bits	32-bits	32-bit	32-bit	32-bit

Figure 13: Context encoding according to the Nesting attribute for each Context size.

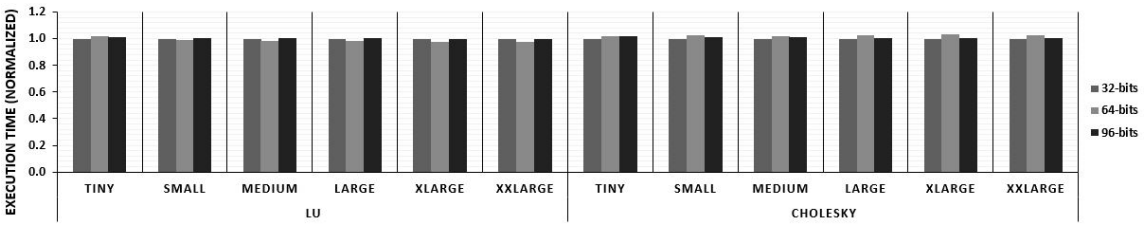

Figure 14: Normalized execution time based on different Context sizes for LU and Cholesky benchmarks.

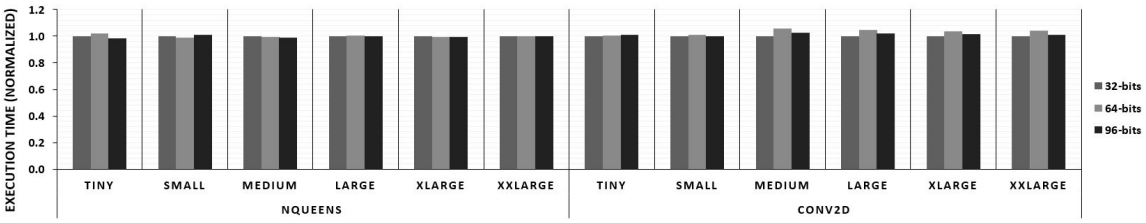

Figure 15: Normalized execution time based on different Context sizes for NQueens and Conv2D benchmarks.

To evaluate how the Context size affects the TSU performance, we measured the normalized parallel execution time of four of our benchmarks: the LU and Cholesky (Figure 14) which have high-complexity Dependency Graphs, the NQueens which implemented using recursion, and the Conv2D which contains DThreads only with RC=1, i.e. there is no allocation of Synchronization Memory (Figure 15). For the three configurations, the results are normalized based on the execution time of the 32-bit Context.

The evaluation shows that the TSU performance is almost the same. The 32-bit and 64-bit configurations require the extraction of loop indexes using shift operations. In the case of the 96-bit Context, the value consists of three 32-bit integers, thus no shifting operations are performed. The

Context size of 96-bit provides more flexibility to the programmers without substantial overheads. Notice that in all experimental results of this work the 96-bit Context is used.

5.3 Performance Evaluation

The speedup results of the eight applications of our benchmark suite are depicted in Figure 16. Fibonacci is not included in the speedup results since the achieved speedup is very low. This is because the sequential execution time is very low, thus, the overhead of parallelizing is more than the benefit gained. For the Fibonacci benchmark we will compare the parallel execution time of our framework with the parallel execution time of two other frameworks: OmpSs and SWARM.

For the *Tiny*, *Small* and *Medium* problem sizes, all the benchmarks apart from Blackscholes, achieved good speedups ranging from 5.77 to 30.74. The Blackscholes achieves speedups from 1.96 to 18.62. For the *Tiny* problem size, the achieved speedup of Blackscholes is negligible because the serial execution time is very low. The problem sizes used for this benchmark are taken from the PARSEC benchmark suite [23].

The evaluation on larger problem sizes (*Large*, *XLarge* and *XXLarge*) shows that FREDDO scales very well across the range of the benchmarks achieving almost linear speedup. This is justified by the fact that, as the benchmark's execution time increases, the parallelization overhead is amortized.

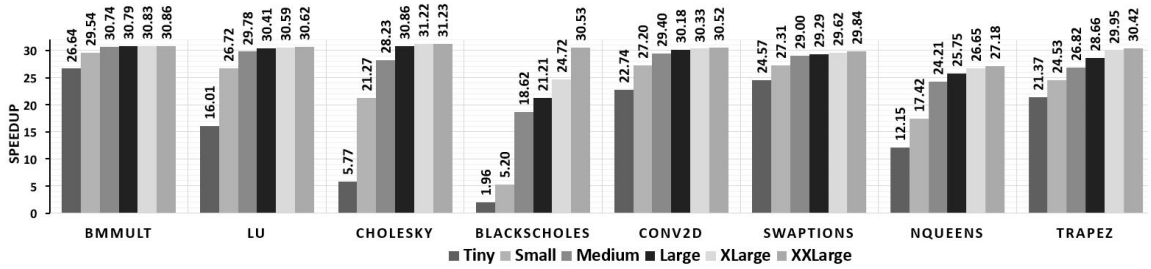

Figure 16: Speedup Results of FREDDO.

We have also evaluated the performance of our framework on different number of cores. Figure 17 illustrates the speedup results of the benchmarks for the *Large* problem size. Four different Kernel configurations are used (4, 8, 16 and 31) where each Kernel reserves a different core. Also, an addition core is reserved by the TSU. The blocked algorithms (BMMULT, Cholesky, Conv2D and LU), Swaptions and Trapez achieved almost the theoretical peak in each configuration. For instance, the LU benchmark, achieves the following speedups: 3.95 out of 4, 7.84 out of 8, 15.57 out of 16 and 30.41 out of 31. Blackscholes and NQueens ended up with lower speedups due to their low sequential execution time (2.11 and 9.66 seconds respectively).

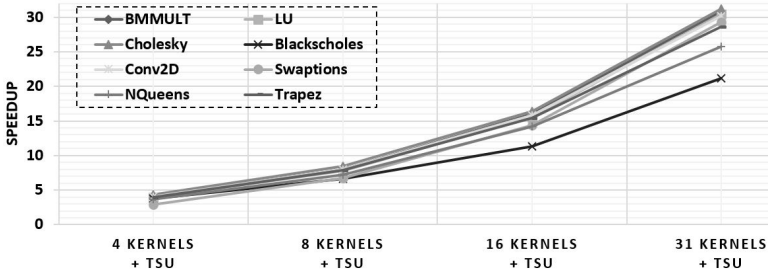

Figure 17: Evaluating FREDDO on different numbers of cores for the *Large* problem size.

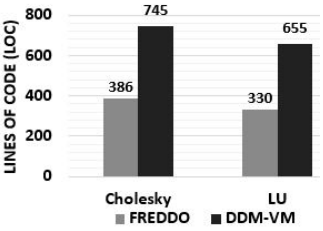

Figure 18: FREDDO vs. DDM-VM on LOC.

5.4 Comparisons with other frameworks

FREDDO is compared with three different frameworks, the DDM-VM [6], the OpenMP (version 3.1) and the OmpSs (version 15.04 [8]), for the LU and Cholesky benchmarks (Figure 19). Cholesky is a complex application with strict data dependencies like the LU benchmark. The DDM systems achieve much better performance results compared to the OpenMP and OmpSs implementations. This is due to the DDM ability to tolerate synchronization latency by decoupling the synchronization from computations, and allowing the TSU core to operate asynchronously from the computation cores.

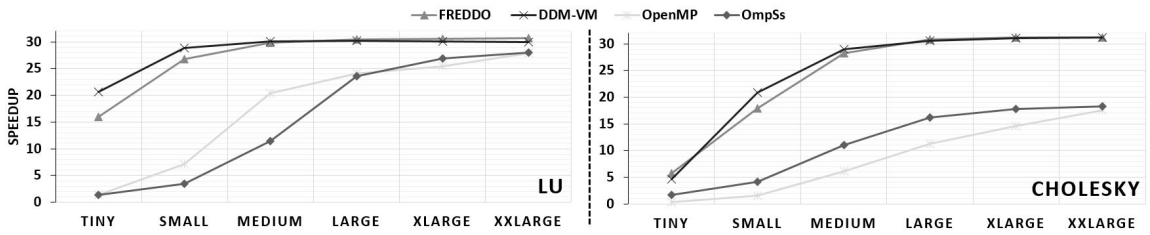

Figure 19: Framework comparisons for the LU and Cholesky benchmarks.

DDM-VM achieves slightly better results than FREDDO, for the *Tiny*, *Small* and *Medium* problem sizes due to three factors:

- The DDM-VM uses only static memory in contrast with FREDDO which uses both static and dynamic memory.
- In DDM-VM, all DThreads are declared in the same function using *label* statements. The DDM-VM's runtime executes the ready DThreads by simply jumping to their *labels* using the *goto* statements. FREDDO, places the DThreads' code in any callable target (functions, Lambda expressions and functors). The FREDDO's runtime executes the ready DThreads by calling the callable targets which incurs more overheads than the DDM-VM's approach.
- In FREDDO, functionalities such as creating, updating and removing DThreads are provided by creating and accessing objects of the *DThread* classes. DDM-VM provides the same functionalities by using C macros which incurs less overheads.

On the other hand, the factors above, restrict the programmers from writing productive DDM-VM applications (see Sections 3.1.2, 3.1.3 and 3.1.5). For the larger problem sizes (*Large*, *XLarge* and *XXLarge*), the overheads derived from the new features of our framework are amortized. Thus, FREDDO achieves similar or even better results compared with DDM-VM, for the larger problem sizes. This is due to optimizations performed in the TSU's structures as well as FREDDO does not allocate Synchronization Memories for DThreads with RC=1. It is worth noting that the lines of code (LOC) of both LU and Cholesky benchmarks in FREDDO is reduced to half (Figure 18) compared to DDM-VM.

In Figure 20 we compare FREDDO with OpenMP for the Blackscholes benchmark and with Intel's Threading Building Blocks (TBB) [14] for the Swaptions [23] benchmark. Both benchmarks are embarrassingly parallel with low data sharing and exchanging between threads. FREDDO achieves better performance than OpenMP and TBB taking into consideration that OpenMP and TBB utilize an additional core for computation. This is primarily due to the low thread switching overheads of our framework.

Additionally, we compare FREDDO with OmpSs and SWARM (SWift Adaptive Runtime Machine) [12] for the Fibonacci benchmark (left side of Figure 21). The results show that our system has lower parallel execution time than the other frameworks. Moreover, the NQueens benchmark is used to compare FREDDO with OmpSs (right side of Figure 21).

The results achieved for all benchmarks show that FREDDO effectively leverages the decoupling of synchronization and execution for the maximum tolerance of synchronization overheads.

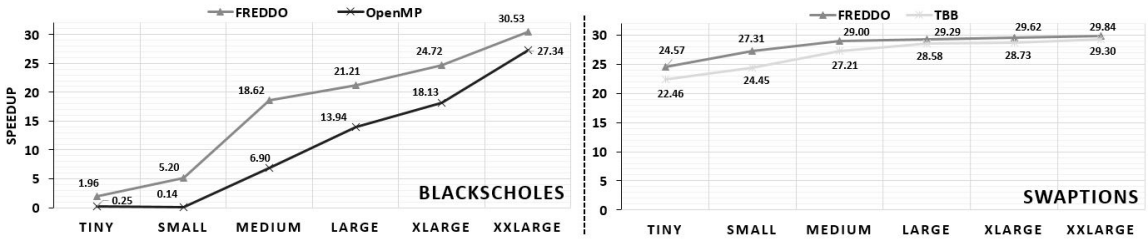

Figure 20: Framework comparisons for the Blackscholes and Swaptions benchmarks.

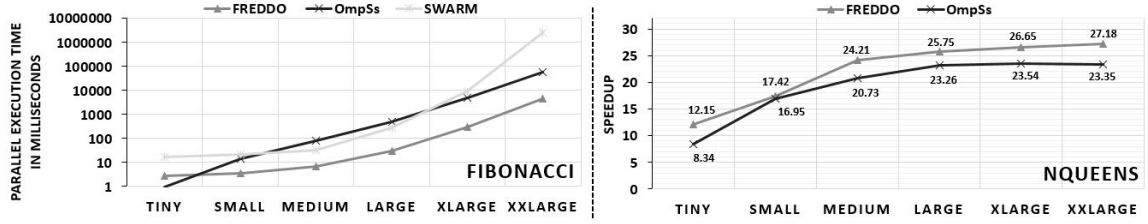

Figure 21: Framework comparisons on recursive benchmarks (Fibonacci and NQueens).

5.5 Dynamic Synchronization Memory (SM) implemented with hash-map

The Dynamic Synchronization Memory (SM) is used for scheduling DThreads which the number of their instances (invocations) cannot be determined at compile-time. The Dynamic SM is implemented as a hash-map data-structure. Thus, choosing the proper hash-map implementation is critical for the TSU performance. We evaluate the TSU performance by using three different hash-map implementations: the unordered_map (UMAP) of the C++11's standard library [24], the boost's unordered_map (BUMAP) [25] and the Google's C++ B-Tree (GBTREE) [26].

For the evaluation of Dynamic SM we used the high complexity benchmarks, LU and Cholesky, because they have DThreads that their RC is greater than one. Notice that, for DThreads with RC=1, the SM (Static or Dynamic) is not used. The normalized execution time of the two benchmarks is shown in Figure 22. The results are normalized based on the timings of UMAP. According to the results, the three hash-map implementations have similar performance. As such, we recommend the use of the UMAP implementation because the users will not need to install third-party libraries on their machines.

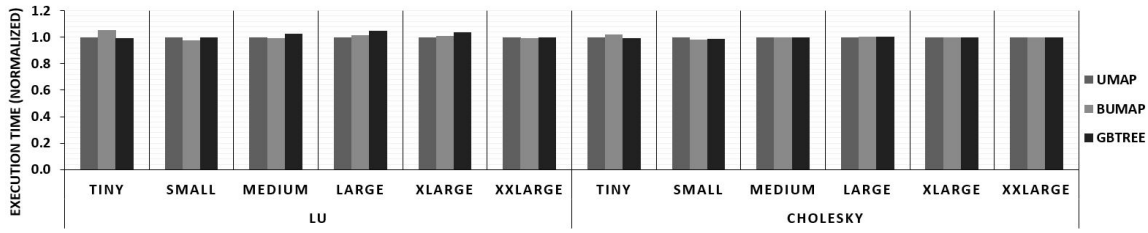

Figure 22: LU and Cholesky: normalized parallel execution time based on different implementations of Dynamic SM.

We also compared the performance of Static SM and Dynamic SM for the LU and Cholesky benchmarks. The results are normalized based on the parallel execution time of Static SM (Figure 23). As expected, the Static SM is faster than Dynamic SM, since accessing an RC entry in Static SM is a direct operation. The Static SM is up to 20% faster than Dynamic SM in the case of LU and up to 6% in the case of Cholesky. The Static SM's performance is much better in LU

compared to Cholesky because the number of updates that are performed in LU are about 20 times more than in the case of Cholesky (Figure 24).

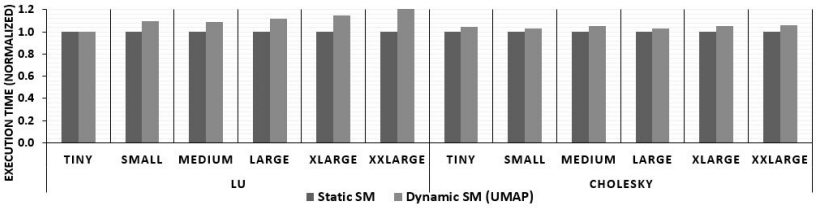

Figure 23: Static SM vs Dynamic SM (UMAP) for the LU and Cholesky benchmarks.

Problem Size	LU	Cholesky
Tiny	5,712	308
Small	44,704	2,280
Medium	353,600	17,360
Large	2,812,544	135,072
XLarge	9,473,984	451,440
XXLarge	22,435,072	1,064,768

Figure 24: Total updates of LU and Cholesky.

6 Related Work

Open Multi-Processing (OpenMP) [7] is a widely-utilized parallel programming API that supports shared-memory programming on multiple platforms. OpenMP consists of a run-time library and a set of compiler directives utilized to identify sections of code that can be parallelized. Traditionally, it targets loop-based parallelism, but the standard was recently extended with the concept of tasks to accommodate irregular applications. OpenMP does not allow the explicit specification of dependencies amongst tasks, which our model naturally does.

Star Superscalar (StarSs) [27, 28, 29] is a parallel programming platform that targets a variety of models, symmetric multi-cores, the Cell processor, GPUs etc. It builds a data dependency graph at runtime where each node represents an instance of an annotated function and edges between nodes denote data dependencies. StarSs has two major components, a source-to-source compiler and a runtime system. The task dependency graph is always built at runtime, hence this approach incurs extra overheads. Moreover, StarSs exposes only a part of the dependency graph available to the scheduler and consequently a fraction of the concurrency opportunities in the applications is visible at any time. The DDM model provides support for both Static and Dynamic dependency resolutions [30]. For this work we have not implemented the Dynamic dependency resolution. It is on our to do list.

OmpSs [8, 9, 10] is a programming model that provides the features of the StarSs 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 Superscalar [31] is an out-of-order pipeline that dynamically detects inter-task data dependencies and executes tasks out-of-order. Task Superscalar addresses similar issues to our work for the StarSs programming model. In this work, an execution time dependency analysis is used. In DDM, we are using static dependency analysis.

Gupta and Sohi [1] have also produced a software system that implemented data-flow/data-driven execution of sequential imperative programs on multi-core systems. They apply their techniques on the function level where the DDM applies its techniques at non-blocking threads.

The Scheduled DataFlow (SDF) [32, 33] is a multithreaded 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) [34] 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) [11] is a software runtime that uses an execution model based on codelets. The codelets model was based on the EARTH project [35]. 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 [30].

Cilk is a parallel programming extension to the C language that adds a few keywords for facilitating parallelism [36]. It uses a fork-join paradigm on top of the existing threading model. The keywords are used to spawn functions as asynchronous parallel tasks and synchronize amongst the tasks using a barrier-like join method. Cilk programs are preprocessed to C and then compiled and linked to a runtime library. The runtime manages the scheduling of the tasks using a work-stealing scheduling policy. The fork-join approach adopted by Cilk is well-suited for expressing recursive algorithms. However, unlike our approach, Cilk does not rely on data dependencies for the scheduling of tasks, which misses part of the potential parallelism.

Cilk Plus [37, 38] is a language extension that provides both task and data parallelism constructs. It's a commercial implementation of Cilk provided by Intel and supports both C and C++. The user can use three simple keywords to express task parallelism: *cilk_for*, *cilk_spawn* and *cilk_sync*. Cilk Plus also provides array notations for expressing data parallelism, reducers for eliminating contention for shared data and SIMD-enabled functions.

Thread Building Blocks (TBB) is an API developed by Intel that relies on C++ templates to facilitate parallel programming [13]. It provides a set of data structures and algorithmic skeletons that supports the execution of tasks. Also, a set of concurrent containers (hash-maps, queues, etc.) and synchronization constructs (mutex constructs, atomic operations, etc.) is provided. The TBB runtime implements a tasks-stealing scheduling policy and adopts a fork-join approach for the creation and management of tasks, similarly to Cilk approach. As such, it suffers the same shortcomings as Cilk.

7 Future Directions

Software DDM implementations, like FREDDO, can be used for efficient data-driven scheduling on conventional multi/many-core processors. Hardware implementations of DDM [18] can achieve better results than software implementations due to two main reasons. The first reason is that the TSU communicates directly with the processor's cores. This allows faster scheduling of the DThreads since the TSU receives faster the Update operations. In software implementations, the TSU communicates with the cores/Kernels through the system's main memory. The second reason is that the different modules of the hardware TSU (Template Memory, Synchronization Memory, Scheduler, etc.) can work concurrently. Thus, basic functionalities like updating the Ready Counts and scheduling the ready DThreads can perform faster.

The hardware implementation of [18] was based on DDM-VM. We would like to integrate the new functionalities of this work into the hardware implementation. Also, the users will develop DDM applications using the FREDDO's programming interface which is based on C++ objects.

Moreover, we will be focused on the implementation of a heterogeneous system that will target High Performance Computing. This system will consist of two main components: a multi/many-core processor and an FPGA device. The processor will consist of conventional cores. The FPGA will hold a low-power and low-complexity many-core processor paired with a hardware TSU [18]. FREDDO will be responsible for executing DDM applications on this architecture. Such a system can be emulated in a Xilinx Zynq MPSoC [39]. Currently, the latest Zynq version integrates a

quad-core ARM processor and a large FPGA device. It is expected that future Zynq devices will contain more ARM cores.

Finally, an efficient distributed DDM system will be implemented. Each node of the system will be a heterogeneous architecture that is described above. A Global Shared Address Space (GAS) system will be implemented in order to allow efficient data transfers between the DDM nodes.

8 Conclusions

In this work we presented FREDDO, an object oriented software implementation for the Data-Driven Multithreading (DDM) model, that targets multi/many-core systems. It provides improvements and new functionalities over the previous DDM implementations.

FREDDO performance has been evaluated on a 32-core server with nine benchmarks, and compared with the performance achieved with OpenMP, OmpSs, TBB, SWARM, and the latest implementation of DDM, the DDM-VM. The evaluation shows that FREDDO scales well and tolerates scheduling overheads and memory latencies effectively. Our framework outperforms the OpenMP and OmpSs especially in the case of high-complexity applications. We also show that FREDDO achieves similar results with DDM-VM despite the fact that the former provides more functionalities. The size of the DDM applications implemented in FREDDO is reduced to half, compared to the size of the DDM-VM applications, due to the improvements in the programming interface.

Acknowledgment

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

References

[1] G. Gupta and G. S. Sohi, "Dataflow execution of sequential imperative programs on multicore architectures," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2011, pp. 59–70.

[2] Arvind and R. A. Iannucci, "Two fundamental issues in multiprocessing," in *4th International DFVLR Seminar on Foundations of Engineering Sciences on Parallel Computing in Science and Engineering*. New York, NY, USA: Springer-Verlag New York, Inc., 1988, pp. 61–88.

[3] 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.

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

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

[6] ——, "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.

[7] O. Board, "Openmp application program interface version 3.0," in *The OpenMP Forum, Tech. Rep*, 2008.

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

[9] 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.

[10] 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.

[11] 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.

[12] ETI, "Swarm," 2015. [Online]. Available: http://www.etinternational.com/index.php/products/swarmbeta

[13] J. Reinders, *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. O'Reilly Media, Inc., 2007.

[14] Intel, "Threading building blocks," 2015. [Online]. Available: https://www.threadingbuildingblocks.org/

[15] C. Kyriacou, P. Evripidou, and P. Trancoso, "Cacheflow: A short-term optimal cache management policy for data driven multithreading," in *Euro-Par 2004*, 2004, vol. 3149.

[16] G. Michael, S. Arandi, and P. Evripidou, "Data-flow concurrency on distributed multi-core systems," in *In Proceedings of the 2013 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA13)*, 2013.

[17] 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.

[18] ——, "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

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

[20] 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.

[21] B. Stroustrup, *The C++ programming language*. Pearson Education, 2013.

[22] S. B. Lippman, J. Lajoie, and B. E. Moo, *C++ Primer*. Addison-Wesley, 2005.

[23] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: characterization and architectural implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM, 2008, pp. 72–81.

[24] cppreference.com, "std::unordered_map," 2015. [Online]. Available: http://en.cppreference.com/w/cpp/container/unordered_map

[25] boost.org, "Class template unordered_map," 2015. [Online]. Available: http://www.boost.org/doc/libs/1_38_0/doc/html/boost/unordered_map.html

[26] Google, "C++ b-tree," 2013. [Online]. Available: http://code.google.com/p/cpp-btree/

[27] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta, "Cellss: a programming model for the cell be architecture," in *SC 2006 Conference, Proceedings of the ACM/IEEE*. IEEE, 2006, pp. 5–5.

[28] 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.

[29] 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.

[30] 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.

[31] Y. Etsion, F. Cabarcas, A. Rico, A. Ramirez, R. M. Badia, E. Ayguade, J. Labarta, and M. Valero, "Task superscalar: An out-of-order task pipeline," in *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on*. IEEE, 2010, pp. 89–100.

[32] 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.

[33] 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.

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

[35] 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.

[36] C. F. Joerg, Robert D Blumofe, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: an efficient multithreaded runtime system," in *Proc. Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1995.

[37] Intel, "Intel cilk plus," 2015. [Online]. Available: https://software.intel.com/en-us/intel-cilk-plus

[38] A. D. Robison, "Cilk plus: Language support for thread and vector parallelism," *Talk at HP-CAST*, vol. 18, 2012.

[39] Xilinx, "All programmable socs and mpsocs," 2015. [Online]. Available: http://www.xilinx.com/products/silicon-devices/soc.html