Architectural Support for Data-Driven Execution

GEORGE MATHEOU and PARASKEVAS EVRIPIDOU, University of Cyprus

The exponential growth of sequential processors has come to an end, and thus, parallel processing is probably the only way to achieve performance growth. We propose the development of parallel architectures based on data-driven scheduling. Data-driven scheduling enforces only a partial ordering as dictated by the true data dependencies, which is the minimum synchronization possible. This is very beneficial for parallel processing because it enables it to exploit the maximum possible parallelism. We provide architectural support for data-driven execution for the Data-Driven Multithreading (DDM) model. In the past, DDM has been evaluated mostly in the form of virtual machines. The main contribution of this work is the development of a highly efficient hardware support for data-driven execution and its integration into a multicore system with eight cores on a Virtex-6 FPGA. The DDM semantics make barriers and cache coherence unnecessary, which reduces the synchronization latencies significantly and makes the cache simpler. The performance evaluation has shown that the support for data-driven execution is very efficient with negligible overheads. Our prototype can support very small problem sizes (matrix 16×16) and ultra-lightweight threads (block of 4×4) that achieve speedups close to linear. Such results cannot be achieved by software-based systems.

Categories and Subject Descriptors: C.1.4 [Processor Architectures]: Parallel Architectures; D.1.3 [Programming Techniques]: Parallel Programming

General Terms: Prototyping, Thread Scheduler, Parallel Architectures, Parallel Programming

Additional Key Words and Phrases: Data-Driven Multithreading, FPGA

ACM Reference Format:

George Matheou and Paraskevas Evripidou. 2014. Architectural support for data-driven execution. *ACM Trans. Architect. Code Optim.* 11, 4, Article 52 (December 2014), 25 pages.
DOI: <http://dx.doi.org/10.1145/2686874>

1. INTRODUCTION

The end of the exponential growth of the sequential processors has facilitated the development of multicore systems. Thus, any growth in performance must come from parallelism [Fuller and Millett 2011]. 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 nonblocking multithreading model based on the Decoupled Data Driven model of execution [Evripidou and Gaudiot 1990; Evripidou 2001]. A DDM thread is scheduled for execution in a data-driven manner, that is, after all of its required data have been produced. As a result, no synchronization or communication latencies are experienced after a thread begins its execution. DDM combines data-driven concurrency with efficient sequential execution by utilizing a Thread Scheduling Unit (TSU) for the management of the threads.

This work was partially funded by the University of Cyprus through a scholarship for George Matheou, by the IKYK foundation, and by the EU TERAFLUX project.

Authors' addresses: G. Matheou and P. Evripidou; emails: {geomat, skevos}@cs.ucy.ac.cy.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2014 ACM 1544-3566/2014/12-ART52 \$15.00

DOI: <http://dx.doi.org/10.1145/2686874>

The DDM model was evaluated in the past by several implementations. The first implementation of DDM was targeting Networks of Workstations [Kyriacou et al. 2006], called the Data-Driven Network of Workstations (D^2 Now). It has illustrated the major components of DDM such as the TSU and CacheFlow [Kyriacou et al. 2004]. The evaluation was done using execution-driven simulations. That was followed by two other implementations, TFlux [Stavrou et al. 2008] and the Data-Driven Multithreading Virtual Machine (DDM-VM) [Arandi and Evripidou 2010, 2011]. Both TFlux and DDM-VM were targeting data-driven concurrency on sequential processors (multicore chips). DDM-VM has extended it into distributed multicore systems for both homogeneous and heterogeneous systems. The heterogeneous version had outperformed some similar systems [Arandi and Evripidou 2011]. TFlux also developed the TFlux directives [Trancoso et al. 2007] and a source-to-source compiler. The TFlux compiler was gradually extended to support all DDM systems. All major components of the DDM were presented in all systems. However, each system has extended the state of the art of DDM. Also, a preliminary project [Matheou and Evripidou 2013] shows that a hardware TSU can be implemented on an FPGA device with a moderate hardware budget.

With this work, we are moving to the next step by developing a hardware system that supports data-driven execution. The major contribution of the submitted work is the full hardware implementation of the TSU. In order to demonstrate the efficiency and functionality of our design, an eight-core shared memory system has been developed that is managed by the TSU. A software Application Programming Interface (API) and a source-to-source compiler are provided for developing DDM applications. For evaluation purposes, a Xilinx ML605 Evaluation Board is used, which is equipped with a Xilinx Virtex-6 FPGA [xilinx.com 2014]. The results of the TSU's performance evaluation showed that data-driven execution can be implemented on sequential multicore systems with a very small hardware budget and negligible overheads.

The rest of the article is organized as follows: an overview of Data-Driven Multithreading is presented in Section 2. Section 3 describes the TSU and the API that is used for developing DDM applications. An example of a DDM program is presented in Section 4. Section 5 describes how the TSU can be ported in a real multicore system. The performance analysis is presented in Section 6. Lessons learned and future work are presented in Sections 7 and 8, respectively. The related work is presented in Section 9. Finally, the concluding remarks are presented in Section 10.

2. DATA-DRIVEN MULTITHREADING

The Data-Driven Multithreading [Kyriacou et al. 2006] is a nonblocking multithreading model that schedules threads based on data availability on sequential processors. A DDM program consists of several threads of instructions, called DThreads, that have producer-consumer relationships. The instructions within the DThreads are fetched and executed by the CPU sequentially in a control-flow manner. This allows the exploitation of a plethora of control-flow optimizations, either by the CPU at runtime or statically by the compiler.

The core of the DDM model is the Thread Scheduling Unit, which is responsible for the management of the DThreads. For each DThread, the TSU collects metadata (also called Thread Templates) that enable the management of the dependencies among the DThreads and determine when a DThread can be scheduled for execution. In particular, TSU schedules a DThread for execution when all its producer threads have completed their execution. This ensures that all the data that this DThread needs are available.

2.1. The DDM Dependency Graph

In DDM, a program is composed of a number of re-entrant, interdependent DThreads along with their DDM Synchronization/Dependency Graph. The Dependency Graph

Fig. 1. Example of a DDM program.

Fig. 2. Example of using multiple instances of the same DThread.

is in the form of $G(N, A, R)$, where nodes in N represent the DThreads, arcs in A represent the data dependencies, and Ready Counts (RCs) in R represent the number of the producer threads of each DThread.

An example of a DDM program is shown in Figure 1. 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. Particularly, DThread 1 and DThread 2 write the variables a and c , respectively, which are then used by DThread 3. As such, DThread 1 and DThread 2 are producers of DThread 3. Similarly, it is possible to observe that there is a dependency between DThread 2 and DThread 4 through the variable c , between DThread 3 and DThread 5 through the variable h , and finally between DThread 4 and DThread 5 through the variable l . These dependencies form the DDM Dependency Graph of this application, which is presented on the right side of Figure 1. The RC values are depicted as shaded values next to the nodes. The RC value is initiated statically and is dynamically decreased 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.

3. TSU: HARDWARE SUPPORT FOR DDM

In this work, the TSU is developed as a hardware peripheral using the Verilog HDL. It uses Thread Templates for the data-driven scheduling of the DThreads. A DThread is identified by the Thread ID (TID) and the *Context* and is paired with a Thread Template.

3.1. The *Context* and Nesting Attributes

3.1.1. The *Context* Attribute. TSU has been based on the U-Interpreter's tagging system [Arvind and Gostelow 1982]. The tagging system enables multiple instances of the same DThread to coexist in the system. More specifically, it maps the tag of the U-Interpreter into a unique 32-bit integer, called the *Context*. This enables concurrency in re-entrant constructs, such as loops, function calls, and recursion. DDM supports implicit dependency resolution among the DThreads. Figure 2 illustrates a simple example of using multiple instances of the same DThread through the *Context* attribute. The for-loop that is shown on the top of the figure is fully parallel; thus, one DThread can be used. The DThread's instances will execute the inner command of the for-loop. In this example, we named this DThread as Thread 1. Each instance of the DThread

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

Nesting Attribute	Context Value
Nesting-0	0 31 0
Nesting-1	Inner Loop 31 0
Nesting-2	Outer Loop Inner Loop 31 16 15 0
Nesting-3	Outer Loop Middle Loop Inner Loop 31 22 21 12 11 0

Fig. 4. Context encoding according to the Nesting attribute.

is identified by the *Context*. The for-loop is executed 32 times; thus, 32 instances of the DThread are created with *Contexts* from 0 to 31.

3.1.2. The Nesting Attribute. The Nesting attribute is a small number that indicates the loop nesting level for the DThreads that implement loops. This is useful for nested loops that can be mapped into a single DThread. In our implementation, we allow up to three nesting levels; that is, 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 a one-level loop is depicted in Figure 2, where the *Context* is equal to the index of the loop. Similarly, in Figure 3, an example of a two-level nested loop is shown. Each instance of the DThread will execute the basic block of the nested loops. The *Context* in this case will include the indexes of the inner (in the lower 16 bits) and the outer (in the upper 16 bits) loop. Figure 4 describes how the indexes of the loops are encoded into the 32-bit *Context* value for all possible combinations. Notice that for the Nesting-0, the *Context* is always zero.

3.2. The Thread Template

The Thread Template is a collection of the following attributes:

- Thread ID (TID):** identifies uniquely a DThread
- Instruction Frame Pointer (IFP):** a pointer to the address of the DThread's first instruction
- Ready Count (RC):** the number of producer threads
- Nesting:** the Nesting attribute
- Scheduling Policy:** the method that is used by the TSU to map the ready DThreads to the cores
- Consumer Threads:** a list of the consumer threads of the DThread

Fig. 5. Block diagram of the TSU microarchitecture supporting eight cores.

Fig. 6. Block diagram of the Template Memory.

3.3. TSU Microarchitecture

The block diagram of TSU’s microarchitecture, which supports eight cores, is shown in Figure 5. Each block of the diagram is a Verilog module that consists of several internal hardware modules. The communication of the cores with the TSU is done through the Output and Input queues. The TSU dispatches the DThreads to the cores through the *Output Queues*. The cores send DDM commands to the TSU through the *Input Queues*. The DDM commands are categorized into six different types:

- (1) **Update_Cmd:** decreases the RC value of a specific DThread
- (2) **Mult_Update_Cmd:** decreases the RC value of multiple instances of a specific DThread
- (3) **Update_Cons_Cmd:** decreases the RC value of the consumers of a specific DThread
- (4) **Mult_Update_Cons_Cmd:** decreases the RC value of multiple instances of the consumers of a specific DThread
- (5) **Store_Cmd:** stores a Thread Template of a specific DThread
- (6) **Remove_Cmd:** removes a Thread Template from TSU

3.3.1. The Template Memory (TM). The Template Memory contains the Thread Template of each DThread. In Figure 6, the block diagram of the TM is illustrated. It is a standard hash table that is used to allocate the TM entries. Each TM entry consists of the Thread Template’s attributes, as well as the Valid field, which indicates if the entry contains valid data. The TM is a fully associative data structure; that is, a Thread Template can be in any entry. The TID is used as the hash table’s key. The TM is implemented as a dual-port RAM, which enables access to two entries simultaneously, and thus the

Fig. 7. Block diagram of the Graph Memory.

processing time is reduced to half. The TM's Finite State Machine (FSM) is responsible for executing the TM's operations:

- Read:** It requests from the Read Lookup module the index of the Thread Template that corresponds to the input TID. After that, it uses the index to output the fields of the Thread Template.
- Write:** It requests from the Write/Invalid module the index of an available entry (Valid = 0). After that, it stores the input Thread Template in that index and it sets the entry's Valid field to 1.
- Invalid:** It requests from the Write/Invalid module the index of an existing Thread Template (Valid = 1) that corresponds to the input TID. After that, it removes the Thread Template by setting its Valid field to 0.

3.3.2. The Graph Memory (GM). The Graph Memory (Figure 7) contains the consumers of each DThread. The consumer threads are kept separately from the TM to facilitate simultaneous access by the TSU. The GM consists of two internal modules, the Consumers Memory (CM) and the Consumer List (CL).

The CL holds lists of consumer threads. To achieve concurrency, the CL's data structure is also a dual-port RAM. Each CL entry points to the next one through the Next field and the entry of the last consumer has the value of Next, set to 0. This means that the address 0 is never used. For example, in Figure 7, there is one list that consists of three consumers with TIDs 3, 4, and 5. This list begins from the address 1 and ends at the address 3. The address of the first entry of a list is called List_Head. The CL's FSM supports three operations:

- Read:** It requests from the List Reader module to find the list of consumers that starts from the address that is equal to the input List_Head. After that, the consumers are returned.
- Write:** It requests from the List Allocator to store the input consumers to a new list. The Valid fields of the list's entries are set to 1 and the List_Head of the new list is returned.

Fig. 8. Block diagram of the Synchronization Memory.

—**Invalid:** It requests from the List Invalidator module to remove an existing list from the CL that has as head the input List_Head. The Valid fields of the deleted list's entries are set to 0.

The CM has exactly the same front-end functionality as the TM. The only difference is that the entries of the CM hold the TID, Cons1, and Cons2 attributes. The TID is also used as the key of the hash table. The Cons1 and Cons2 attributes hold the consumers of a DThread. If a DThread has more than two consumers, the Cons1 attribute becomes zero while the Cons2 attribute is a pointer to a list of consumers; that is, it holds the List_Head of the list that is stored in the CL (see the CM entry in the address 3). The GM's FSM also supports three operations:

- Read:** The consumers of the input TID are retrieved. The CM module is accessed. If a DThread has more than two consumers, then the CL is also accessed.
- Write:** The consumers of the input TID are stored in the GM. If the DThread has more than two consumers, then the appropriate data is stored in the CM and the CL. Otherwise, the consumers are stored only in the CM.
- Invalid:** The consumers of the input TID are removed from the GM. The data is removed from the CM. If a DThread has more than two consumers, then the appropriate data is also removed from the CL.

3.3.3. The Synchronization Memory (SM). The Synchronization Memory contains the RC values for each DThread. A DThread that implements a loop has multiple instances, one for each iteration. The TSU holds a separate entry for each instance of a DThread in the SM. The RC values are allocated and deallocated dynamically by the SM in the form of blocks, based on the applications' needs. The SM (Figure 8) consists of two different modules, the SM Indexer (SMI) and the Ready Count Memory (RCM).

The RCM is a one-dimensional array that holds the RC values in blocks of 32 cells. The RCM supports the following operations:

- Allocate:** Searches for a free RCM block. If the block is found, the RCM module initializes all the entries of the block with the value of the RC that is stored in the TM. For instance, in Figure 8, the first 32 entries of the RCM are set to 2.
- Write:** Modifies an RCM entry.
- Read:** Returns the contents of an RCM entry.
- Invalid:** Invalidates an RCM entry.

The SMI holds the RCs of the DThreads. If a DThread has Nesting-0, the RC field holds the value of the DThread's RC. In our example, the entry in the address 2 holds the RC (with value 3) of the DThread with TID = 4. If a DThread has Nesting > 0, then the RC field holds the address of an RCM block that corresponds to a specific TID and *Context* between the Min_Iteration and Max_Iteration. In the address 0 of the SMI module, 32 RC values of the DThread with TID = 1 are allocated, which correspond to the *Contexts* from 0 to 31. The Valid_Cells attribute indicates how many RC entries are not zero. If its value becomes zero, the SMI entry will be deallocated as well as the RCM entry that is referred by the RC field. The SMI supports the following operations:

- Search:** Searches for a specific SMI entry, that is, an entry that is mapped to the input TID and the input *Context*, which is between its Min_Iteration and Max_Iteration fields. The two ports of the dual-port RAM are used in parallel to decrease the search time. If the SMI entry is not found, a new entry is allocated.
- Write:** Modifies an SMI entry.
- Invalid:** Invalidates an SMI entry.

The SM supports two operations:

- Clear:** Removes the SMI and RCM entries that correspond to the input TID.
- Update:** Decreases the RC value of a specific DThread with a specific *Context*. The Update operation consists of three algorithmic steps:
 - STEP 1:** Search the SMI module for the specific entry. If the entry is found, go to STEP 2; otherwise, go to STEP 3.
 - STEP 2:**
 - If the DThread has Nesting-0, the RC value in the SMI entry is decreased. If the RC becomes zero, the ready DThread and its scheduling information are stored in the Ready Queue (RQ) and the SMI entry is invalidated.
 - If the DThread has Nesting > 0, the value of the RCM entry in the address $RC + offset$ is decreased, where RC is the field of the SMI entry and the $offset$ is equal to $Context - Min_Iteration$. If the RCM entry's value becomes zero, the ready DThread and its scheduling information are stored in the RQ and the Valid_Cells attribute is decreased by one. If the Valid_Cells value becomes zero, the SMI entry and its RCM block are invalidated.
 - STEP 3:** This step is responsible for allocating resources in SM.
 - If the DThread has Nesting-0, a new SMI entry is allocated with the following values: Valid_Cells = 1, Min_Iteration = $(Context/32) * 32$, Max_Iteration = Min_Iteration, and RC = the TM's RC.
 - If the DThread has Nesting > 0, a new RCM block is allocated. After that, a new SMI entry is allocated with the following values: Valid_Cells = 32, Min_Iteration = $(Context/32) * 32$, Max_Iteration = Min_Iteration + 31, and RC = the address of the new RCM block.
 - Finally, go to STEP 2.

3.3.4. The Fetch Unit. The Fetch Unit is shown in Figure 9. The Input Queue Readers dequeue the DDM commands from their corresponding Input Queues and send them to the Round-Robin arbiter, which forwards them to the Command Manager. The

Fig. 9. Block diagram of the Fetch Unit.

Fig. 10. Block diagram of the Update Unit.

Command Manager decodes the DDM commands and sends them to the appropriate modules. The Store_Cmd or Remove_Cmd commands are sent to both the TM and GM Loaders/Invalidators. The Store_Cmd is used for storing the Thread Templates and the consumers to the TM and the GM, respectively. Similarly, the Remove_Cmd is used for removing the Thread Templates and the consumers. Finally, the Update commands are stored in the Update Queue.

3.3.5. The Update Unit. The Update Unit (Figure 10) decrements the Ready Counts (RCs) of the DThreads in the SM. The Resolver receives the Update commands from the Update Queue and processes the Update_Cons_Cmd and the Mult_Update_Cons_Cmd commands. When these commands are processed, the consumers of the DThread that are going to be updated are retrieved from the GM, through the GM Reader. For each consumer, a separate Update command (Update_Cmd or Mult_Update_Cmd) is stored in the Update Buffer.

The Processing Unit is responsible for sending the Update commands to the SM. For each Update command, the Thread Template of the DThread that is going to be updated is located from the TM through the TM Reader. If an Update_Cmd is processed, an Update signal is sent to the SM. If a Mult_Update_Cmd is processed, a separate Update signal is sent to the SM for each instance of the DThread. For the update operation, the *TID* and *Context* attributes are sent to the SM.

3.3.6. The Scheduling Unit. The *Scheduling Unit* enforces the *Scheduling Policy* by assigning a ready DThread (*TID*, *IFP*, and *Context*) to the corresponding Output Queue.

Fig. 11. Block diagram of the Scheduling Unit.

The *Scheduling Policy* consists of two fields: (1) the scheduling method and (2) the scheduling value. Two scheduling methods have been implemented: dynamic and static. The dynamic method distributes the thread invocations to the cores in order to achieve load balancing (the scheduling value is not used). Furthermore, in the static method, the instances of a DThread are assigned to a specific core. For this purpose, the scheduling value is used to hold the identity of the specific core. For instance, if a user wants to execute a DThread only on the core with $id = 1$, then a DThread with $method = static$ and $value = 1$ has to be created (recall that we are supporting an eight-core implementation, which means that each core has a unique id between 0 and 7).

Figure 11 depicts the block diagram of the Scheduling Unit. The Map Thread Unit (MTU) dequeues the ready DThreads along with their scheduling information (scheduling method and scheduling value) from the SM's RQ. The MTU uses the scheduling information to forward the ready DThreads to the proper Output Queues through the Output Queue Writers.

3.4. TSU Timings

Table I depicts the minimum and maximum cost (in cycles) of the TSU operations. It is worth noting that the majority of these operations can be executed concurrently. For instance, the Fetch Unit can read from Input Queues while the Update Unit decreases the RC values in the SM. Since the TSU operates dynamically, the majority of its operations depends on the size of the structures. For instance, the maximum cost of the TM Write/Invalid operation is equal to its minimum cost plus the size of the TM structure in cycles. The cost of the operations that manage consumers, such as the GM Write operation, depends on the number of the consumers (# of consumers) that they will manage.

The dynamic allocations/deallocations of the RC values affect the performance of the SM. To mitigate this issue, we allocate blocks of RC values (32 RC values each time) in order to avoid frequent allocations/deallocations. This technique improves the TSU performance, since the SM Update operation is the most frequent operation.

3.5. The TSU's API

The TSU API is the interface between the DDM applications and the TSU. The API is a C library that includes a set of functions (Table II) that allow the programmers to (1) initialize/reset the hardware TSU, (2) send the DDM commands to the TSU via the Input Queues, and (3) fetch the ready DThreads from the Output Queues and execute their codes.

Table I. Timings of the TSU Operations

Operation	Minimum Cycles	Maximum Cycles
Graph Memory (GM) [GM Size = 16]		
<i>Write</i>	$(16 * \# \text{ of consumers}) + 10$	$(16 * \# \text{ of consumers}) + 10 + (2 * \text{GM size})$
<i>Read</i>	$(3 * \# \text{ of consumers}) + 7$	$(3 * \# \text{ of consumers}) + 7 + \text{GM size}$
<i>Invalid</i>	$\# \text{ of consumers} + 10$	$\# \text{ of consumers} + 10 + \text{GM size}$
Template Memory (TM) [TM Size = 16]		
<i>Write</i>	10	10 + TM size
<i>Invalid</i>	10	10 + TM size
<i>Read</i>	7	7 + TM size
Fetch Unit		
<i>Read from Input Queues</i>	4	4 + $\# \text{ of consumers}$
<i>Store Thread Template</i>	13	13 + TM size
<i>Store Consumers</i>	$13 + (16 * \# \text{ of consumers})$	$13 + (16 * \# \text{ of consumers}) + (2 * \text{GM size})$
<i>Remove Thread Template</i>	13	13 + TM size
<i>Remove Consumers</i>	$13 + \# \text{ of consumers}$	$13 + \# \text{ of consumers} + \text{GM size}$
<i>Forward Update Command</i>	2	2
Other [SM Size = 1024]		
<i>Input Queue: Enqueue/Dequeue</i>	1	1
<i>Output Queue: Enqueue/Dequeue</i>	1	1
<i>Synchronization Memory (SM) Update</i>	14	$28 + (2 * \text{SM size})$
<i>Scheduling Unit: schedules a ready DThread</i>	8	9
<i>Update Unit: receives data from Fetch Unit and executes an update command</i>	25	$46 + (2 * \text{SM size}) + \text{TM size} + (3 * \# \text{ of consumers}) + \text{GM size}$

4. A DDM PROGRAM EXAMPLE

In this section, we show how a programmer is able to develop the matrix multiplication application (Listing 1) in DDM using the TSU's API. In this simple example, the outer for-loop of the algorithm is able to be parallelized by using only one DThread.

One possible transformation of this application to a DDM program is shown in Figure 12. The outer for-loop is mapped into a DThread called *thread_1*. Since *thread_1* implements a one-level loop, its Nesting equals 1 (Nesting-1). Each instantiation of *thread_1* calculates one row of the C matrix; that is, it executes the two nested for-loops of the algorithm. Each instantiation is labeled with its *Context* as $\langle \text{Context} \rangle$. Initially, the N instantiations of *thread_1* are spawned in parallel because the *thread_1*'s instances are independent. Finally, when N instantiations (from 0 to N-1) are executed, the program is completed.

4.1. Declaring the DDM Threads (DThreads)

Listing 2 depicts the code of the DThreads. In this simple implementation, we declare two DThreads, *thread_1* and *thread_2*. In our programming model, the code of DThreads is embodied in standard C functions. So, the names of the C functions are the DThreads' IFPs. Each function has one input argument, which is a type of DContext. The DContext is a C structure that contains the *inner*, *middle*, and *outer* members. These members hold the indexes of the loops according to the Nesting attribute. The members are set by the API at runtime before the execution of the DThread. The API decodes the *Context* of the ready DThread and fills the members properly.

Table II. The Functions of the TSU's API

Function	Description
System Management	
tsu_init ()	Initializes the TSU.
tsu_clear ()	Cleans up the resources that TSU allocates like the Thread Templates and the Ready Count Values.
tsu_start ()	Starts the TSU and executes the first ready DThread.
DDM Dependency Graph Management	
dthread_create (THREAD_NAME, RC, SCH.METHOD, SCH.VALUE, NESTING, CONS_LIST)	Loads the thread template in the TSU. The thread template consists of the thread name, the ready count value, the scheduling policy (method and value), the loop nesting level, and the consumers of the DThread.
dthread_remove (THREAD_NAME)	Removes a DThread given its specific name and its related data from the TSU.
DDM Dependency Graph Maintenance	
dthread_update (THREAD_NAME, RC_VAL, CONTEXT)	Decreases the RC value by RC_VAL of a DThread's specific instance.
dthread_update_cons (THREAD_NAME, RC_VAL, CONTEXT)	Decreases the RC value by RC_VAL of a specific instance of a DThread's consumers.
dthread_mult_update (THREAD_NAME, RC_VAL, CONTEXT, MAX_CONTEXT)	Decreases the RC value by RC_VAL of a DThread's multiple instances.
dthread_mult_update_cons (THREAD_NAME, RC_VAL, CONTEXT, MAX_CONTEXT)	Decreases the RC value by RC_VAL of multiple instances of a DThread's consumers.
Managing Ready Threads	
dthread_get_next ()	Dequeues the next ready DThread from the TSU's Input Queue and it executes its code.

```

// Global Variables
float* A;
float* B;
float* C;

int main()
{
  // Memory Allocations
  A = (float*) malloc(N*N * sizeof(float));
  B = (float*) malloc(N*N * sizeof(float));
  C = (float*) malloc(N*N * sizeof(float));

  // Data initialization goes here ...

  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
      for (k = 0; k < N; k++)
        C[i][j] += A[i][k] * B[k][j];

  // Post execution code goes here ...
}

```

Listing 1: The original Matrix Multiplication.

Fig. 12. Matrix Multiplication: the dynamic instantiations of *thread_1*.

thread_1 is responsible for executing the two nested for-loops of the algorithm (lines 13–15) using the *Context* value as the *i* index of the outer loop. The *i* index is stored in the *inner* member of the *cntx* variable, that is, in *cntx.inner*. After that, each instantiation of *thread_1* updates *thread_2* (line 18). The *thread_2* is used to print the

Table III. The DCONTEXT_CREATE Macros

Nesting	Macro
Nesting-0	DCONTEXT_CREATE_Z ()
Nesting-1	DCONTEXT_CREATE_S (INNER INDEX)
Nesting-2	DCONTEXT_CREATE_D (OUTER INDEX, INNER INDEX)
Nesting-3	DCONTEXT_CREATE_T (OUTER INDEX, MIDDLE INDEX, INNER INDEX)

```

1 // The function prototypes
2 void thread_1(DContext);
3 void thread_2(DContext);
4
5 // Declare the code of thread_1
6 void thread_1(DContext cntx)
7 {
8     // Local Variables
9     int j, k;
10    float res;
11
12    // Executes the algorithm using the context as i
13    for (j = 0; j < N; j++)
14        for (k = 0; k < N; k++)
15            C[cntx.inner][j] += A[cntx.inner][j] * B[k][j];
16
17    // Update the thread_1's consumers, i.e thread_2
18    dthread_update_cons(thread_1, 1, DCONTEXT_CREATE_Z());
19
20    // Execute the next ready thread
21    dthread_get_next();
22 }
23
24 // Declare the code of thread_2
25 void thread_2(DContext cntx)
26 {
27     // Print results goes here ...
28
29     // Remove the threads
30    dthread_remove(thread_1);
31    dthread_remove(thread_2);
32
33    // Clean-up the TSU
34    tsu_clear();
35 }

```

Listing 2: The code of the DThreads.

results, to release the resources that are allocated by the two DThreads (lines 30–31) and to clean up the TSU (line 34).

At line 21 of the *thread_1*'s code, the *dthread_get_next()* function is called. This function is a blocking command that waits for a ready DThread from the TSU. When a ready DThread is received by the API, the *TID*, *Context*, and *IFP* attributes are extracted. After that, the *Context* is decoded according to the Nesting attribute and the members of the DContext input argument of the ready DThread's function are set. Finally, the IFP attribute is used to execute the ready DThread's code.

Furthermore, in line 18, the *DCONTEXT_CREATE* macro is used to encode the indexes of the loop levels into the *Context* value, as we mentioned in Section 3. Table III illustrates the variants of the *DCONTEXT_CREATE* macro that correspond to the Nesting attribute.

4.2. DDM Dependency Graph Creation and Execution

Listing 3 illustrates the main function of the DDM program. First, the TSU has to be initialized (line 10). After the arrays (A, B, and C) are allocated and initialized,


```

1 // Global Variables
2 float* A;
3 float* B;
4 float* C;
5
6 // The main program
7 int main()
8 {
9     // Initialize the TSU
10    tsu_init();
11
12    // Memory Allocations
13    A = (float*) malloc(N*N * sizeof(float));
14    B = (float*) malloc(N*N * sizeof(float));
15    C = (float*) malloc(N*N * sizeof(float));
16
17    // Initializations goes here ...
18
19    // Create Thread Template for thread_1
20    DConsumer consumers_1[1] = {thread_2};
21    dthread_create(thread_1, 1, sch_dynamic, core_dynamic, nesting_one, consumers_1);
22
23    // Create Thread Template for thread_2
24    DConsumer consumers_2[1] = {};
25    dthread_create(thread_2, N, sch_static, core_0, nesting_zero, consumers_2);
26
27    // Multiple Updates (from 0 to N-1)
28    dthread_mult_update(thread_1, 1, DCONTEXT.CREATE_S(0), DCONTEXT.CREATE_S(N-1));
29
30    // Start the TSU
31    tsu_start();
32
33    return 0;
34 }

```

Listing 3: The main program: initializations, dependency graph creation/execution.

the Thread Templates are created and loaded in the TSU (lines 20–25) using the *dthread_create* function.

thread_1's ready count equals 1, its scheduling policy is dynamic, and its Nesting is set to 1 (*nesting_one*). Also, it has one consumer, the *thread_2* DThread. Moreover, *thread_2*'s RC equals N because it needs to wait for the N instantiations of *thread_1* to finish their execution. *thread_2* is scheduled to be executed in the core with id 0 (scheduling method = static and scheduling value = 0) and it has no consumers. Also, *thread_2*'s nesting is set to 0 (*nesting_zero*) because it does not implement a loop.

After the DThreads are loaded, the N instantiations of *thread_1* are released by using the *dthread_mult_update* command (line 28). This command is provided to send a special request to the TSU for decrementing multiple consecutive instances of a specific thread. The TSU manages this special request internally in an optimized manner, which reduces overheads significantly. Finally, the *tsu_start* function is called in line 31 in order to enable the TSU and also to execute the first ready DThread.

4.3. A Source-to-Source Compiler for the DDM Model

In order to allow easy programming, we provide an extension of the TFlux source-to-source compiler [Trancoso et al. 2007]. The primary target of this compiler is to hide the details of the API from the programmer. To develop a DDM application, the programmer only needs to describe the parallel sections of the application using the TFlux directives, as in the OpenMP API. Functionalities such as loading/unloading the TSU and managing the *Context* are handled automatically. Listing 4 depicts the Matrix Multiplication application in DDM style, using the C directives. In this case,

```

1 // Global Variables
2 float* A;
3 float* B;
4 float* C;
5
6 // The main program
7 int main()
8 {
9     // Memory Allocations
10    A = (float*) malloc(N*N * sizeof(float));
11    B = (float*) malloc(N*N * sizeof(float));
12    C = (float*) malloc(N*N * sizeof(float));
13
14    // Initializations goes here ...
15
16    #pragma ddm for thread 1 start (0 : N-1) nesting 1
17    for (i = 0; i < N; i++)
18        for (j = 0; j < N; j++)
19            for (k = 0; k < N; k++)
20                C[i][j] += A[i][k] * B[k][j];
21    #pragma ddm endfor update (2)
22
23    #pragma ddm thread 2 kernel (static, core_0) readycount N end
24    // Print results goes here ...
25    #pragma ddm endthread
26
27    return 0;
28 }

```

Listing 4: The Matrix Multiplication application using C directives.

our TFlux compiler extension will take as input the C application and will produce the code that targets the DDM model, that is, the code that we described in Listings 2 and 3.

The *#pragma ddm for thread* directive in line 16 will create the DThread 1 that will parallelize the outer for-loop of the algorithm. The *start* keyword will release the N instantiations of the DThread 1 (from 0 to $N-1$) as mentioned earlier. The *nesting* keyword defines the loop nesting level of the DThread. When this keyword is omitted, the TFlux compiler sets the DThread's Nesting to zero. The *#pragma ddm endfor* directive in line 21 closes the *#pragma ddm for thread* directive and updates the DThread 2 in each for-loop iteration. The DThread 2 is declared in line 23 by the *#pragma ddm thread* directive. The *kernel* keyword enforces the scheduling policy. For the DThread 2, the static method is used in core with $id = 0$. When the *kernel* keyword is omitted, the dynamic scheduling will be enforced. Furthermore, the *readycount* keyword defines the ready count of the DThread. Finally, the *end* keyword indicates the last DThread of the program. This keyword notifies the compiler to add the appropriate functions for removing the DThreads and cleaning up the TSU.

5. PORTING THE TSU ON A MULTICORE SYSTEM

To test the functionality and the performance, we developed a multicore system that is paired with our TSU implementation. Since we are working with Xilinx's FPGAs, we are able to build the multicore system using the Xilinx's Intellectual Property (IP) blocks such as processors, buses, memory controllers, and so forth. The entire design was developed using the Xilinx Platform Studio (XPS).

The block diagram of the multicore architecture is shown in Figure 13. It consists of eight cores, each of them featuring a Xilinx MicroBlaze soft core [xilinx.com 2014] with its caches and local memory. The MicroBlaze is a 32-bit RISC Harvard architecture that operates at 100MHz. It has a 128KB local memory for data and instructions, a 32KB L1 D-Cache, and a 16KB L1 I-Cache, implemented using Block RAM (BRAM)

Fig. 13. Block diagram of a multicore architecture paired with the hardware TSU.

[xilinx.com 2014]. The sizes of the local memory and the caches are selected based on the available BRAM of the Virtex-6 FPGA.

The cores share a DDR3 SDRAM Controller via a shared AXI Bus, which provides access to a 512MB DDR3 SDRAM chip. Furthermore, the cores share a standard set of peripherals through an AXI Lite Bus. These peripherals provide basic functionality such as an interrupt controller, a UART interface for accessing an RS-232 port, a timer, and a MicroBlaze Debug Module (MDM) [xilinx.com 2014], which enables JTAG-based debugging to one or more MicroBlaze cores.

The TSU communicates with each core with an Input and an Output Queue, implemented with the Fast Simplex Link (FSL) Bus [xilinx.com 2014]. FSL is a very fast 32-bit-wide interface that provides unidirectional FIFO-based communication.

5.1. Developing and Executing DDM Applications

DDM applications that target the multicore processor are developed using ANSI-C/C++ augmented with the set of the API's functions. The TFlux source-to-source compiler is also available for the same procedure. The DDM binary is produced by the MicroBlaze GCC compiler. The DDM binary includes the user's application, the drivers of the Xilinx peripherals (such as the drivers of timer, buses, memory, etc.), and the TSU's API. The users are able to write their programs using the Xilinx Software Development Kit (SDK).

For the development of a multithreaded application in Xilinx's programming environment, an instance of the DDM binary has to be loaded in each MicroBlaze. The flow of each program instance is managed by the TSU dynamically at runtime through the API. The Xilinx Microprocessor Debugger (XMD) [xilinx.com 2014] is used for downloading the DDM binary into the MicroBlazes' local memory as well as for executing the code.

Table IV. The Benchmark Suite Characteristics

Benchmark	Description	Source	Problem Size						
			Tiny	XXSmall	XSmall	Small	Medium	Large	XLarge
MMULT	Matrix Multiplication	Kernel	16×16	32×32	64×64	128×128	256×256	512×512	1,024×1,024
BMMULT	Blocked Matrix Multiplication	Kernel	16×16	32×32	64×64	128×128	256×256	512×512	1,024×1,024
LU	Blocked LU Decomposition	SPLASH-2	16×16	32×32	64×64	128×128	256×256	512×512	1,024×1,024
Conv2D	9×9 convolution filter	Kernel	16×16	32×32	64×64	128×128	256×256	512×512	1,024×1,024
Trapez	Trapezoidal rule for integration	Kernel	2 ¹⁶ steps	2 ¹⁸ steps	2 ¹⁹ steps	2 ²⁰ steps	2 ²¹ steps	2 ²³ steps	2 ²⁵ steps
Blackscholes	Financial analysis application	PARSEC	8 options	12 options	16 options	-	-	-	-
SMMULT	Sparse Matrix Multiplication	Kernel	16×16	32×32	64×64	128×128	256×256	512×512	1,024×1,024

5.2. The System's Memory Model

In our system's design, the DThreads can share the memory through their inputs and not through any special synchronization in the memory. For instance, if a producer thread passes an array to a consumer thread, the array has to be filled before the consumer thread is invoked to consume it. This functionality allows DDM execution without the need of synchronization barriers. Furthermore, the execution according to the DDM dependency graph eliminates race conditions.

The MicroBlaze's caches are simple and very fast modules due to three factors: (1) they are implemented by using BRAM, which is a very fast memory unit; (2) they support the Direct Mapped scheme; and (3) they don't implement cache coherency. The DDM semantics, which are implemented in the hardware TSU, guarantee that the consumer threads will be activated only after the producer threads terminate; as such, there is no need for a cache coherency protocol. This feature allows us to use the MicroBlaze caches. For data consistency, we are using the write-through storage method.

6. PERFORMANCE ANALYSIS

For evaluating the correctness and performance of the TSU, we use a suite of seven different benchmarks. Table IV illustrates the characteristics of the benchmarks along with the different problem sizes tested. The problem sizes are separated into seven categories: *Tiny*, *XXSmall*, *XSmall*, *Small*, *Medium*, *Large*, and *XLarge*. The thread size of the blocked algorithms is 4×4 for the *Tiny*, *XXSmall*, and *XSmall* sizes; 16×16 for the *Small* size; and 32×32 for the *Medium*, *Large*, and *XLarge* sizes. The column labeled "Source" indicates the benchmark suite where each benchmark is originated. The Source named "Kernel" indicates general kernels widely used in scientific and image processing applications.

In the case of the *Blackscholes* benchmark [Bienia et al. 2008], our system is unable to run the standard problem sizes (from 4K to 10M options). The reason is that our FPGA device does not have enough BRAM to model the bigger MicroBlazes' local memories for holding the *Blackscholes* binary with the standard problem sizes. Despite this limitation, we did run the *Blackscholes* benchmark for very small problem sizes (from eight to 16 options) in order to study its behavior.

Benchmark	DThread	Nesting	Ready Count Value	Consumers
BMMULT	Thread_1	Nesting-3	1	Thread_1, Thread_2
	Thread_2	Nesting-0	1	None
MMULT	Thread_1	Nesting-1	1	Thread_1, Thread_2
	Thread_2	Nesting-0	8	None
Blackscholes	Thread_1	Nesting-1	1	Thread_1, Thread_2
	Thread_2	Nesting-0	1	None
Trapez	Thread_1	Nesting-1	1	Thread_2
	Thread_2	Nesting-0	8	None
Conv2D	Thread_1	Nesting-1	1	Thread_2
	Thread_2	Nesting-2	1	Thread_1, Thread_3
	Thread_3	Nesting-0	1	None
LU	Thread_1	Nesting-1	1	Thread_2, Thread_3, Thread_4, Thread_5
	Thread_2	Nesting-1	2	Thread_3, Thread_4, Thread_6
	Thread_3	Nesting-2	3	Thread_5
	Thread_4	Nesting-2	3	Thread_5
	Thread_5	Nesting-3	4	Thread_2, Thread_3, Thread_4, Thread_5
	Thread_6	Nesting-0	1	None
SMMULT	Thread_1	Nesting-1	1	Thread_1, Thread_2
	Thread_2	Nesting-0	8	None

Fig. 14. The DThreads' characteristics of each benchmark.

Fig. 15. Speedup for the DDM execution.

The execution time measurements were collected using the hardware Timer of the system. The experimental results are given in the form of speedups in relation to the sequential/serial execution on one core of the system. Notice that all cores are identical.

Figure 14 depicts the DThreads' characteristics of each benchmark. For each DThread, we show its Nesting and RC attributes as well as its consumer threads. If a DThread has no consumers, its Consumers field is set as "None." The *LU* [Woo et al. 1995] is the most complex benchmark. Its Synchronization Graph consists of six DThreads, where all the Nesting types are used. Also, the Thread_1, Thread_2, and Thread_5 DThreads have at least three consumer threads.

6.1. Performance Evaluation

The speedup results of all seven applications are depicted in Figure 15. We have evaluated the ability of the system to handle small problem sizes and ultra-lightweight threads (*Tiny*, *XXSmall*, *XSmall*, and *Small* sizes). The four applications, *MMULT*, *BMMULT*, *Conv2D*, and *Trapez*, achieve speedups from 6.74 to 7.96. The high complexity of the *LU* ended up with smaller speedups on the order of 1.93 to 6.79. Although we evaluate the *Blackscholes* on very small problem sizes, our system achieves speedups from 4.71 to 7.61. Also, the *SMMULT* benchmark achieves speedups from 2.9 to 7.42.

The evaluation on larger problem sizes (*Medium*, *Large*, and *XLarge*) shows that the multicore system scales very well across the range of the benchmarks achieving almost linear speedup. From these results, it is easy to observe that for the *MMULT*,

BMMULT, *SMMULT*, and *LU* benchmarks, the speedup increases for larger problem sizes. This is justified by the fact that, as the benchmark's execution time increases, the parallelization overhead is amortized. Furthermore, for the *Conv2D* and *Trapez* benchmarks, the speedup reaches the theoretical peak performance for all the sizes.

MMULT is an embarrassingly parallel application but suffers from a large number of misses, limiting it from achieving the ideal speedup. *BMMULT* exploits the data locality by using blocking techniques; thus, it achieves better performance than *MMULT*. *SMMULT* suffers from poor cache utilization due to the lack of spatial locality. For the *SMMULT* benchmark, the Compressed Sparse Row (CSR) format is used. *LU* achieves the lowest speedup, compared to the other applications, due to the complexity of its Synchronization Graph, which leads to reduced parallelism. *Blackscholes* is a data-parallel benchmark with low data sharing and low data exchange. Finally, *Conv2D* and *Trapez* have very few data transfers between DThreads, which allows them to achieve the optimal speedup on all problem sizes.

Our system achieves very good performance in both small and large sizes, since the overheads of the TSU are negligible. There are four factors that contribute to this:

- The communication between the TSU and the DDM applications is made through nonblocking commands (apart from the command that is responsible for fetching the ready DThreads).
- The TSU is an asynchronous hardware module that uses optimization techniques like pipelining for increasing its performance.
- The TSU's data structures (like TM, GM, etc.) are implemented using dual-port RAM, which reduces the accessing time to half.
- The TSU communicates directly with the cores. This allows faster scheduling of the DThreads.

The efficiency of the hardware support allows us to reach linear speedups with smaller problem sizes and thread granularity. DDM-VM [Arandi 2012; Michael et al. 2013] had to use a problem size of $2,048 \times 2,048$ and block size of 64×64 to get similar speedups. We have four applications in common: *BMMULT*, *LU*, *Conv2D*, and *Trapez*. The DDM-VM applications achieve speedups close to 8/9, 7.2/9, 8/9, and 7.2/9, respectively. Notice that in DDM-VM, one core is reserved for the execution of the TSU, and thus nine cores are used for the evaluation. Our small problem size (128×128) gets roughly similar results with DDM-VM at $1/_{256}$ of the problem size and $1/4$ of the block size.

6.2. Comparing Our System with the OpenMP Framework

In our final experiment, we compare our system with the OpenMP framework. For the evaluation of the OpenMP implementation, a 12-core AMD Opteron processor has been used with the following features: 2.2GHz clock, 128KB L1 Cache, 512KB L2 Cache, and 6MB L3 Cache. The results of the very small problem/thread sizes (Tiny, XXSmall, and XSmall) are shown in Figure 16. As expected, the hardware support of our system provides a big advantage over the software-based implementation for the very small program/thread granularity.

Figure 17 depicts the results of the comparison for the larger granularity (*Small*, *Medium*, *Large*, and *XLarge*). The OpenMP gets much better speedups for the coarser granularity and achieves similar results compared to our system for the *Large* and the *XLarge* size. The largest problem sizes allow the software-based system to amortize better the extra overhead that is incurred from the software concurrency management.

Fig. 16. Our system versus OpenMP on Tiny, XXSmall, and XSmall problem sizes.

Fig. 17. Our system versus OpenMP on Small, Medium, Large, and XLarge problem sizes.

Table V. Virtex-6 FPGA Resource Utilization and Power Consumption Estimations

Component	Flip Flops	LUTs	BRAM	Power (W)
Core x 8	28,340 (9%)	31,782 (21%)	384 (92%)	0.27
TSU	2,055 (0%)	3,307 (2%)	8 (1%)	0.01
FSL x 16	1,000 (0%)	2,128 (1%)	-	0.02
Other	1,311 (0%)	1,574 (1%)	0 (0%)	0.17
Total	32,706 (10%)	38,791 (25%)	392 (93%)	0.47

6.3. Comparing Our System with the SMP Superscalar (SMPSs)

We also compared our work with the published work of BSC SMPSs [BSC 2014] for two benchmarks: Blocked Matrix Multiplication (BMMULT) and LU. The BMMULT achieves a speedup of 5 out of 8 for all three problem sizes (1,024, 2,048, 4,096). For the LU, the small size (1,024) achieves speedup of 3, the medium size (2,048) achieves a speedup close to 4.5, and the large size (4,096) achieves a speedup of 4.8. Our system gets a speedup of 8 for BMMULT and 7.91 for LU for the size of 1,024.

The SMPSs schedules annotated tasks at runtime based on data dependencies, as in our model. DDM creates the Dependency Graph statically, while the SMPSs build it at runtime. This can cause more delay to the critical path of the application than in our model. Moreover, SMPSs makes only a part of the Dependency Graph available to the scheduler, and consequently, a fraction of the concurrency opportunities in the applications are visible at any time. BSC runs its experiments on two Power5 processors, with two FPU's, at 1.5GHz.

6.4. FPGA Resource Utilization and Power Consumption Estimations

Table V depicts the FPGA resource utilization and power consumption estimations of the multicore processor. The utilization percentage of each component is shown in parentheses. The component labeled “other” includes the clock generator, the MDM, the timer, the AXI Buses, and so forth, which are all necessary for the proper functionality of the system but outside the scope of this work.

The hardware device utilization of our prototype is rather low, which will allow us to extend the functionality of our system in the future. The BRAM utilization, on the other hand, is quite high at 93%. We choose to utilize as much as possible BRAM to model big caches and private memories, in order to increase the system's performance. The maximum operating frequency of the TSU peripheral is 200MHz. To be fair, we degrade the TSU's frequency at 100MHz, which is the MicroBlaze's frequency.

The results show that the hardware TSU can easily fit on the Virtex-6 FPGA since it utilizes about 3% of its resources; thus, the TSU can also be applied to FPGAs with lower capacity. Furthermore, the TSU utilizes a small proportion (2.12%) of the overall power of the system.

7. LESSONS LEARNED

The FPGA prototyping is much more challenging than software simulation and development of virtual machines, but it provides accurate modeling on timings, resource utilization, power consumption estimations, and so forth. The FPGA design is not just a hardware simulation. Xilinx provides services that turn the FPGA into an ASIC with higher performance.

The TSU was tightly integrated into a multicore processor for testing its functionalities and efficiency. We have realized that the limited amount of BRAM, which was used for memory and caches, did not allow us to run applications with large footprints. We are looking into ways of remedying this situation.

In our design, a Dynamic SM has been used. In this approach, the programmer provides no information about the *Context* bounds. Also, there is no bound on the size the *Context* can reach. The dynamic allocation/deallocation of RC values at runtime increases the TSU overheads. To mitigate this problem, we reduce the allocation/deallocation overheads of the SM by grouping the RC values into blocks. Each block contains 32 RCs [Stavrou et al. 2008]. Another approach of the SM is the static/direct, where the allocation occurs at the time of creating the Thread Template, which accelerates the updates of the RCs. In this approach, the programmer is required to provide information on the maximum value the *Context* of the DThread would reach. We plan to create a hybrid combination of static and dynamic SM and evaluate its performance.

The hardware TSU is able to manage more than eight cores. It can be extended by just changing a few parameters in the TSU's Verilog code. However, in this project, we were not able to have more than eight cores because the AXI bus and the MicroBlaze Debug Module can only support eight cores.

The software TSU has more overheads and higher latencies than the hardware implementation. The hardware implementation gets much better speedups in the entire range of the experiments since the overheads of the TSU are negligible. Finally, the hardware TSU is a low-complexity and low-power system that can be expanded for a larger number of cores in the future.

8. FUTURE WORK

The FPGA prototype presented in this article is a starting block that will be extended for high-performance computing (HPC). Figure 18 illustrates the future DDM implementation. We plan to replace the data cache of each core with an automated Scratch-Pad Memory (SPM). The SPM's controller will deterministically fetch the data needed for a DThread into the SPM only when the DThread will be scheduled for execution. We expect that this will increase the locality by sending DThreads to cores that have their input data. We have developed such a system in software for the CELL processor with very good results [Arandi and Evripidou 2011]. The next level will be a distributed implementation, in which the TSU of each processor will be able to collaborate with TSUs of other processors by moving data.

processors. The design of DDM avoids the use of a new ISA or extensions to an ISA because it adopted an evolutionary approach by just designing an add-on unit, the TSU. This unit can easily fit into a multicore system without any other changes to the architecture.

The Manchester Data-flow Machine [Gurd et al. 1985] is a dynamic dataflow system that focused on constructing a powerful processing element. The Manchester system implements a tagged-token dataflow model of computation in order to increase the parallelism for re-entrant graphs. This implementation combines the concept of streams with conventional arrays.

Capalija and Abdelrahman [2013] proposed a coarse-grained superscalar processor that allows parallel and out-of-order execution of tasks. This processor is implemented on an FPGA platform and uses two microarchitectural techniques: register renaming and dynamic task scheduling. On the other hand, in our work, we don't use out-of-order execution.

Star Superscalar (StarSs) [Bellens et al. 2006; Pérez et al. 2007; Planas et al. 2009] is a parallel programming platform that targets a variety of models, symmetric multicores, the Cell processor, GPUs, and so forth. 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, and 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 are visible at any time. The DDM model provides support for both static and dynamic dependency resolutions [Arandi et al. 2011]. For this work, we have not implemented the dynamic dependency resolution. It is on our to-do list.

The Task Superscalar [Etsion et al. 2010] is an out-of-order pipeline that dynamically detects intertask 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 [2011] have also produced a software system that implemented dataflow execution of sequential imperative programs on multicore systems. This work is more related to the TFlux and DDM-VM because it is a software system; however, the techniques used are different. They apply their techniques on the function level, whereas the DDM applies its techniques at nonblocking threads. Moreover, in DDM, a set of pragmas has been used for thread creation/management.

Vandierendonck et al. [2013] studied several schemes for dynamic dependence tracking in the task dataflow model. This model utilizes a task graph where nodes represent dynamic task instances and edges represent task dependencies. A set of memory usage annotations is used to define the dependencies between the tasks. Tasks are ordered by true, anti-, and output dependencies. In DDM, the data-driven execution enforces only a partial ordering as dictated by the true data dependencies, which is the minimum synchronization. Thus, it has the potential of being more efficient.

EARTH (Efficient Architecture for Running THreads) [Theobald 1999] is an event-driven fine-grained multithreaded execution model. EARTH runs on top of parallel machines built with off-the-shelf processors and implements a two-layer hierarchy of fibers and threaded procedures. A fiber is a sequentially executed, nonpreemptive, atomically scheduled set of instructions. A threaded procedure groups interacting fibers that share data. EARTH was never implemented in hardware. It is a software implementation, like the DDM implementations of TFlux and DDM-VM.

10. CONCLUSIONS

In this article, we present the design, development, and evaluation of a hardware Thread Scheduling Unit (TSU) that provides architectural support for data-driven execution based on the Data-Driven Multithreading (DDM) model. The TSU is a low-power and low-complexity hardware unit that schedules threads based on data availability on sequential processors.

The TSU implements concurrency at many levels, and as a result, scheduling, data management, and transfer operations are interleaved with the execution of DThreads, which reduces latencies. Furthermore, an API has been developed that provides all the functionality needed for managing the hardware TSU. The programmer can use the TFlux pragma-based source-to-source compiler for program development.

As a proof of concept, the TSU has been integrated into a multicore system with eight cores on a Virtex-6 FPGA. The performance evaluation of the system has shown that the architectural support for data-driven execution can be implemented in multicore systems with negligible overheads. We are very encouraged by the overall results and especially with the ability of our system to get almost linear speedups for very small problem sizes (16x16 matrices) and ultra-lightweight threads of the order of 4x4 blocks. The hardware support we developed utilizes about 3% of the total resources of the Virtex-6 FPGA. Thus, we are confident that our architectural support can support larger number of cores and even be expanded to hierarchical configurations.

ACKNOWLEDGMENTS

The authors would like to thank Dr. Pedro Trancoso and Andreas Diavastos for developing the TFlux directives for this work.

REFERENCES

- Samer Arandi. 2012. *The Data-Driven Multithreading Virtual Machine*. Ph.D. Dissertation.
- Samer Arandi and Paraskevas Evripidou. 2010. Programming multi-core architectures using Data-Flow techniques. *IEEE*, 152–161. DOI: <http://dx.doi.org/10.1109/ICSAMOS.2010.5642072>
- Samer Arandi and Paraskevas Evripidou. 2011. DDM-VMc: The data-driven multithreading virtual machine for the cell processor. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*. ACM, ACM Press, 25–34. DOI: <http://dx.doi.org/10.1145/1944862.1944869>
- Samer Arandi, George Michael, Paraskevas Evripidou, and Costas Kyriacou. 2011. Combining compile and run-time dependency resolution in data-driven multithreading. In *2011 First Workshop on Data-Flow Execution Models for Extreme Scale Computing (DFM)*. *IEEE*, 45–52.
- Arvind and Kim P. Gostelow. 1982. The U-Interpreter. *Computer* 15, 2 (Feb. 1982), 42–49. DOI: <http://dx.doi.org/10.1109/MC.1982.1653940>
- Pieter Bellens, Josep M. Perez, Rosa M. Badia, and Jesus Labarta. 2006. CellSs: A programming model for the Cell BE architecture. In *Proceedings of the ACM/IEEE SC 2006 Conference*. *IEEE*, 5–5.
- Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*. ACM, 72–81.
- BSC. 2014. SMP Code Examples. (2014). Retrieved August 27, 2014, from <http://www.bsc.es/computer-sciences/programming-models/smp-superscalar/code-examples>.
- Doug Burger, Stephen W. Keckler, Kathryn S. McKinley, Mike Dahlin, Lizy K. John, Calvin Lin, Charles R. Moore, James Burrill, Robert G. McDonald, and William Yoder. 2004. Scaling to the end of silicon with EDGE architectures. *Computer* 37, 7 (2004), 44–55.
- Davor Capalija and Tarek S. Abdelrahman. 2013. Microarchitecture of a coarse-grain out-of-order superscalar processor. *IEEE Transactions on Parallel and Distributed Systems*, 24, 2 (2013), 392–405.
- Yoav Etsion, Felipe Cabarcas, Alejandro Rico, Alex Ramirez, Rosa M. Badia, Eduard Ayguade, Jesus Labarta, and Mateo Valero. 2010. Task superscalar: An out-of-order task pipeline. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. *IEEE*, 89–100.
- Paraskevas Evripidou. 2001. D3-Machine: A decoupled data-driven multithreaded architecture with variable resolution support. *Parallel Computing* 27, 9 (2001), 1197–1225.

- Paraskevas Evripidou and Jean-Luc Gaudiot. 1990. Decoupled graph/computation data-driven architecture with variable resolution actors. In *Proceedings of the 1990 International Conference on Parallel Processing*.
- Samuel H. Fuller and Lynette I. Millett. 2011. Computing performance: Game over or next level? *Computer* 44, 1 (Jan. 2011), 31–38. DOI: <http://dx.doi.org/10.1109/MC.2011.15>
- Gagan Gupta and Gurindar S Sohi. 2011. Dataflow execution of sequential imperative programs on multicore architectures. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 59–70.
- John R. Gurd, Chris C. Kirkham, and Ian Watson. 1985. The manchester prototype dataflow computer. *Communications of the ACM* 28, 1 (1985), 34–52.
- Costas Kyriacou, Paraskevas Evripidou, and Pedro Trancoso. 2004. CacheFlow: A short-term optimal cache management policy for data driven multithreading. In *Euro-Par 2004 Parallel Processing*, Marco Dane-lutto, Marco Vanneschi, and Domenico Laforenza (Eds.). Lecture Notes in Computer Science, Vol. 3149. Springer Berlin Heidelberg, 561–570. DOI: http://dx.doi.org/10.1007/978-3-540-27866-5_73
- Costas Kyriacou, Paraskevas Evripidou, and Pedro Trancoso. 2006. Data-driven multithreading using conventional microprocessors. *IEEE Transactions on Parallel and Distributed Systems* 17, 10 (Oct. 2006), 1176–1188. DOI: <http://dx.doi.org/10.1109/TPDS.2006.136>
- George Matheou and Paraskevas Evripidou. 2013. Verilog-based simulation of hardware support for data-flow concurrency on multicore systems. In *Proceedings of the 2013 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII)*. IEEE, 280–287.
- George Michael, Samer Arandi, and Paraskevas Evripidou. 2013. Data-flow concurrency on distributed multi-core systems. In *Proceedings of the 2013 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'13)*.
- Josep M. Pérez, Pieter Bellens, Rosa M. Badia, and Jesús Labarta. 2007. CellSs: Making it easier to program the Cell Broadband Engine processor. *IBM Journal of Research and Development* 51, 5 (2007), 593–604.
- Judit Planas, Rosa M. Badia, Eduard Ayguadé, and Jesus Labarta. 2009. Hierarchical task-based programming with StarSs. *International Journal of High Performance Computing Applications* 23, 3 (2009), 284–299.
- Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Doug Burger, Stephen W. Keckler, and Charles R. Moore. 2003. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*. IEEE, 422–433.
- Kyriakos Stavrou, Marios Nikolaidis, Demos Pavlou, Samer Arandi, Paraskevas Evripidou, and Pedro Trancoso. 2008. TFlux: A portable platform for data-driven multithreading on commodity multi-core systems. *Parallel Processing, 2008. ICPP'08. 37th International Conference on*. IEEE, 25–34. DOI: <http://dx.doi.org/10.1109/ICPP.2008.74>
- Steven Swanson, Ken Michelson, Andrew Schwerin, and Mark Oskin. 2003. WaveScalar. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'36)*. IEEE Computer Society, Washington, DC, 291.
- Kevin Bryan Theobald. 1999. *Earth: An Efficient Architecture for Running Threads*. Ph.D. Dissertation. Montreal, Quebec, Canada, Canada. Advisor(s) Gao, Guang R. AAINQ50269.
- Pedro Trancoso, Kyriakos Stavrou, and Paraskevas Evripidou. 2007. DDMCPP: The data-driven multithreading C pre-processor. In *Proceedings of the 11th Workshop on Interaction between Compilers and Computer Architectures*. Citeseer, 32.
- Hans Vandierendonck, George Tzenakis, and Dimitrios S. Nikolopoulos. 2013. Analysis of dependence tracking algorithms for task dataflow execution. *ACM Transactions on Architecture and Code Optimization (TACO)* 10, 4 (2013), 61.
- Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. 1995. The SPLASH-2 programs: Characterization and methodological considerations. In *ACM SIGARCH Computer Architecture News*, Vol. 23. ACM, 24–36.
- xilinx.com. 2014. All Programmable Technologies from Xilinx Inc. (2014). Retrieved August 28, 2014, from <http://www.xilinx.com/>.

Received July 2014; revised October 2014; accepted October 2014