Data-Driven Multithreading Programming Tool-chain

Andreas Diavastos
George Matheou
Paraskevas Evripidou
Pedro Trancoso

TR-17-3 **September 2017**

University of Cyprus

Technical Report

Data-Driven Multithreading Programming Tool-chain

Andreas Diavastos, George Matheou, Paraskevas Evripidou and
Pedro Trancoso

UCY-CS-TR-17-3

September 2017

Table of Contents

Abstract

The increasing parallelism offered by the parallel architectures introduced by processor vendors, coupled with the need to extract more parallelism out of the applications, has led the community to examine more efficient programming and execution models. The Dataflow Multithreading model is known to be the model that can exploit the most parallelism out of a wide range of applications. The Dataflow Multithreading model is not new to the community and one of the main reasons it wasn't widely spread among the user community is the programming effort it requires in order to exploit the maximum parallelism out of a Dataflow implementation. The wide spread of parallel machines today though has led many researchers to re-examine its applicability. In this work we explore the programmability of the Data-Driven Multithreading (DDM) model, a non-blocking execution model that applies the principles of Dataflow execution at a coarser granularity, that of sequence of instructions. In this paper we present a tool-chain that promises to ease the effort needed by the programmers to write efficient DDM applications that exploit the Dataflow execution model and in particular two implementations of the DDM model, the TFlux Architecture and the Data-Driven Multithreading Virtual Machine (DDM-VM). The tool-chain includes a set of programming directives, developed under the DDM C Preprocessor project and an Eclipse Plug-in suite that enables content assist while integrating directives in applications, a side panel with the available directives, their arguments and explanation for each one. Finally, the preprocessing procedure is integrated in the Eclipse editor to enable easy DDM application development regardless of the platform used.

Index terms— dataflow; programming model; Data-Driven Multithreading;

1 Introduction

Unsustainable power consumption and ever-increasing design and verification complexity have driven the microprocessor industry to the multicore architectures as an architectural solution, sustaining Moores' law [8]. Researchers and processor vendors are investigating architectures, compilers, and programming models for manycore processors with hundreds or even thousands of cores on a single platform. The major challenge today is to find programming and execution models that will efficiently keep all the available resources busy while maintaining a high level of energy efficiency.

The Dataflow model was proposed by Jack Dennis in the early 70s [9, 10] as an alternative to the control-flow (or von Neumann) model of execution. In Dataflow the instructions can only be executed when all of their input operands are available, as in an asynchronous model. Moreover, computations has no side-effects, which makes Dataflow a functional model. A Dataflow program is compiled into a Dataflow graph, that is a data dependency graph representing the order of execution imposed by data dependencies. A Dataflow graph specifies only a partial order for the execution of instructions and thus provides opportunities for parallel and pipelined execution at the level of individual instructions. The main advantage of the Dataflow model is its ability to expose the maximum amount of parallelism to the hardware since the only condition for an instruction to start executing is the availability of its input data. Also a Dataflow program works in a distributed concurrency control, due to the fact that there is no central point of control (no need for Program Counter). These advantages allow Dataflow architectures to tolerate the synchronization and memory latencies and extract more parallelism compared to traditional control-flow models. As such, the Dataflow model is suitable for multicore and manycore systems.

The Data-Driven Multithreading (DDM) model of execution is a non-blocking multithreading model that was inspired by the Dataflow and the Decoupled Data Driven (D^3) models of execution. The DDM model decouples the synchronization from the computation portions of a program allowing them to execute asynchronously in a Dataflow manner. This is achieved by scheduling a thread for execution only when its input data have been produced. As such, it exploits the benefits of the Dataflow model and no synchronization or communication latencies are experienced after a thread begins execution. Moreover, the DDM exploits the sequential processing of the commodity

microprocessors due to the fact that instructions within a thread are fetched by the CPU sequentially in a control-flow order. Hence, the CPU can exploit the advantages of pipelining, branch prediction and out-of-order execution.

The sequence by which the processors execute the threads is defined either statically or dynamically using a specialized module called Thread Scheduling Unit (TSU). The TSU is responsible for scheduling the threads for execution based on data availability. Effectively, scheduling based on data availability can tolerate synchronization and communication latencies. The synchronization information, i.e. the information regarding the data dependencies among the threads, is stored into the TSU at compile time. The dependencies among the threads are expressed via producer-consumer relationships. Hence, a thread can be scheduled for execution only when all of its producers have completed their execution. Additionally, the DDM model currently has three implementations: the Data-Driven Network of Workstations (D^2NOW) [3], the Thread Flux platform (TFlux) [1] and the Data-Driven Multithreading Virtual Machine (DDM-VM) [2].

The threads definition, the scheduling policy that the execution will follow, the producer-consumer relationships among threads and all the synchronization information in any programming paradigm that follows the Dataflow scheduling must be explicitly defined by the user. Different implementations also have different interfaces, which means that the user must be familiar with all implementations in order to parallelize an application using Dataflow principles. The lack though of programming tools that can effectively help the user, makes the programming of Dataflow applications preventive.

In this work we present a single tool-chain that promises to ease the effort of the user community in developing Dataflow applications that follow different platforms of the DDM model of execution. This tool-chain supports the different implementations of the TFlux and the DDM-VM platforms. Programming an application using this tool-chain we can produce, from a single program, parallel code for the hardware implementation of TFlux (*TFluxHard*) as well as for the software implementation of TFlux (*TFluxSoft*). Using the tool-chain to write a single program, we can also produce code for the software implementation of DDM-VM, the *DDM-VMs* and also its hardware implementation. This tool-chain consists of a set of programming directives that can be used to define the Dataflow principles in a DDM program and a DDM C Preprocessor that, based on the directives inserted and the target platform chosen, will produce the appropriate DDM parallel code. Finally, it introduces an Eclipse Plug-in suite that provides an easier way for the programmer to integrate the programming directives and a one-step generation of DDM executables.

The rest of the paper is organized as follows. Section 2 present the DDM model and the two platforms we are studying in this work (TFlux and DDM-VM). Section 3 describes the most important directives of the tool-chain presented. In Section 4 we describe the Eclipse Plug-in suite and its different modules introduced in this paper. In Section 5 we give an incremental development of a DDM application using the directives and the Eclipse Plug-in. Finally, in Section 6 we conclude with the main contributions of this work and what is our plan for the future regarding the further development of the tool-chain.

2 Data-Driven Multithreading Architectures

In this work we focus on two *Dataflow Multithreading* approaches that implement the DDM model. The TFlux architecture [1] and the DDM-VM [2]. In order to overcome some of the limiting overheads of the Dataflow Multithreading these architectures follow a coarse-grain granularity approach, applying the Dataflow principles on sequence of instructions, called Dataflow Threads (DThreads). Both these architectures schedule the DThreads in a dataflow-like way based on the data availability of each DThread.

2.1 TFlux Architecture

The TFlux architecture serves as a virtualization platform for the execution of DDM programs on top of any commodity multicore computer system. Its objective is to offer the DDM programming

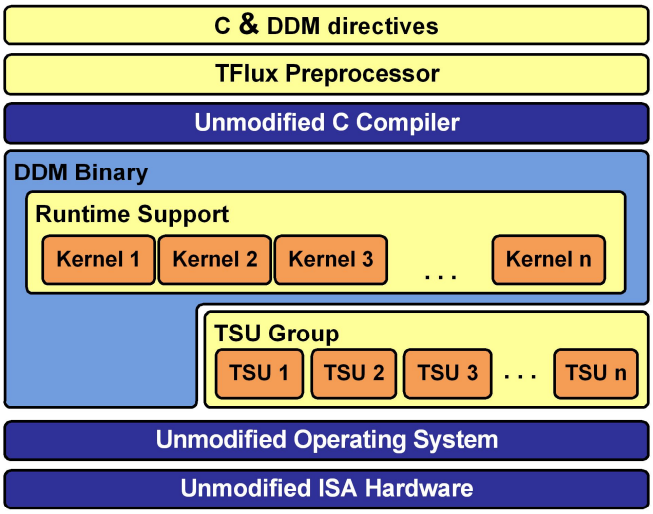

Figure 1: The layered design of the TFlux architecture [1].

model to any system architecture by virtualizing the details of the underlying system. In particular, it implements a runtime support system that is build on top of a commodity Operating System with a software or hardware TSU responsible for scheduling the DThreads.

In Figure 1 we present the different modules of the TFlux system in a layered design. A DDM binary executable of the application also invokes the TFlux Runtime Support allowing the application to execute under the DDM model. The Runtime support hides all DDM-specific implementation and execution details from the programmer. The primary responsibility of the runtime is to dynamically load the application DThreads into the scheduling unit and invoke the TSU scheduling operations that will eventually allow the DDM execution of the application. The runtime of the TFlux architecture also provides a mechanism for the DDM application DThreads to access the shared data in the producer-consumer relationships characterizing a Dataflow-like application and also provides an efficient communication layer between the application and the TSU.

The scheduling unit in the TFlux architecture is also called a *TSU Group*. The reason is that, unlike the previous implementation of the DDM (D^2NOW) [3]) that required each processor (which was an independent machine) to have its own TSU, in TFlux, these TSUs were grouped in a single unit (TSU Group) with their operations split in two categories; Those that are common to all CPUs and those that only correspond to each CPU serving a TSU. This reduced the overhead of the TSU-to-TSU communication since now all the communication operations are handled internally and, even more importantly, this allowed the implementation of the scheduling unit in software, by emulating the TSU functionality, thus allowing a multiprocessor implementation of the TFlux architecture.

As a proof-of-concept the TFlux architecture was implemented both in hardware and software. The hardware implementation (*TFluxHard*) was tested on a Simics-based full-system simulator using a Sparc multicore system. TFlux has also two software implementations, the (*TFluxSoft*) that can be used on any commodity x86 homogeneous multicore system and the *TFluxSCC* that runs on a 48-core Intel Single-chip Cloud Computer [12].

2.2 Data-Driven Multithreading Virtual Machine

The DDM-VM is a virtual machine that supports the DDM model of execution on homogeneous and heterogenous multicore systems. It virtualizes the resources of an underlying parallel system and uses a general, unified representation for DDM programs. The DDM-VM runtime system,

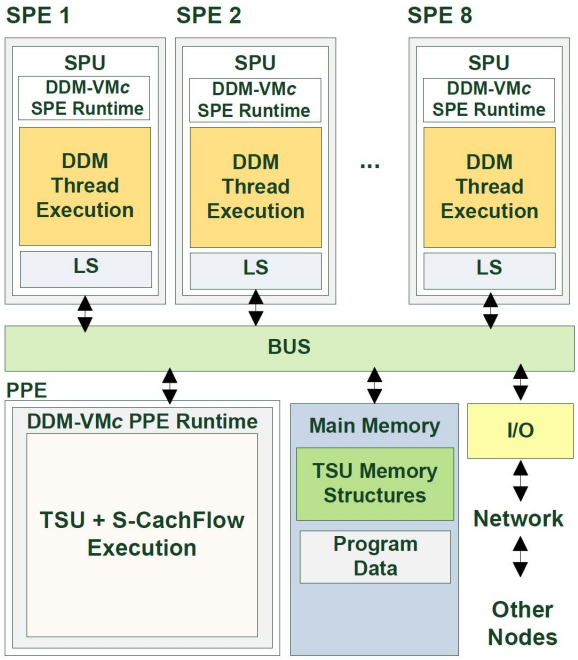

Figure 2: The Cell/BE implementation of the DDM-VM [2].

mainly composed of the TSU, handles the tasks of thread scheduling, execution instantiation and data prefetching implicitly.

The scheduling of threads is orchestrated by the TSU, which is implemented as a software module running on one of the cores. The TSU is aided by the runtime that supports DDM execution on the rest of the cores. The DDM-VM implementation uses a cache management policy, called CacheFlow [2], that ensures that all data that a thread needs for execution is in the cache before the thread is scheduled for execution. This technique helps improve the performance of DDM applications running under the DDM-VM implementation. CacheFlow is a fully automated prefetching software cache with variable block sizes that is extended with many optimizations like adaptive multi-buffering, data re-use and reference-counting.

Using specialised I-Structures [11], the DDM-VM supports parallel execution of code that contains producer-consumer dependencies that are only resolved at runtime while utilizing compile-time resolution at the same time. This takes advantage of the strengths of both approaches and expands the class of programs that can be mapped to the DDM model to a wider range. It also has the potential to improve the programmability and optimize the compilation methods generating Dataflow code.

There are two versions of the DDM-VM implementation, one that is optimised for homogeneous symmetric multiprocessors and one that is tailored for the Cell Broadband Engine [4]. The latter version was developed for heterogeneous multicore architectures with a host/accelerator organization and a software-managed memory hierarchy. It provides a fully-automated software prefetching cache with variable cache block sizes and explicit data locality optimizations for handling explicitly-managed memory hierarchies. As seen on Figure 2, the TSU responsible for scheduling threads at run-time is implemented as a software module running primarily on the Power Architecture based (Power Processor Element - PPE) which is a two-way multithreaded core, while the execution of the threads takes place on the eight fully-functional SIMD co-processors (Synergistic Processing Elements - SPE). This mapping is an efficient utilization of the Cell heterogeneous resources; as the code of the TSU that heavily uses branches and control-flow structures, is more suited to run

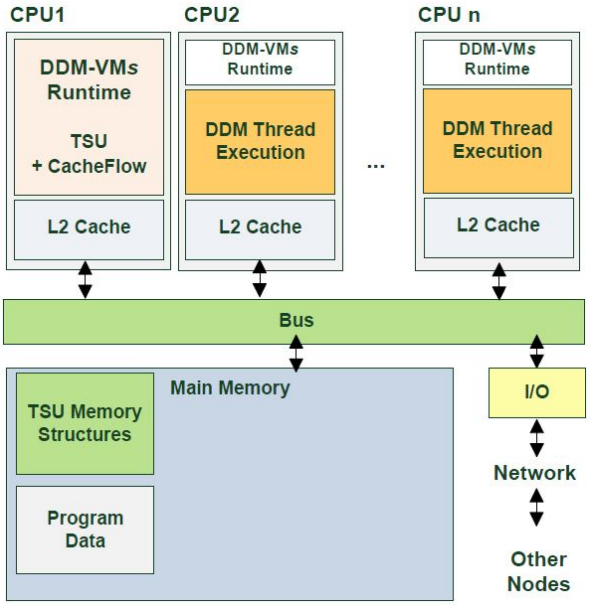

Figure 3: The Symmetric implementation of the DDM-VM [2].

on the general purpose PPE core originally designed for control tasks, while the threads are more suited to run on the SIMD SPE cores optimized for computational loads. In Figure 3 we depict the second version of the DDM-VM implementation, that supports symmetric multi-processors. The TSU in this case does not utilise special hardware to execute and uses any processor to aid its tasks and uses the shared memory architecture to control the execution.

3 Tool-chain Programming Directives

In this section we present the programming style of the DDM implementations of TFlux and DDM-VM. As discussed later, programming a parallel application using the Dataflow principles requires significant effort and expertise from the programmer, regardless of the implementation used. To ease the effort of the programmer and remove any unnecessary implementation specific details from the programming level we use programming directives along with the DDM C Pre-Processor (DDMCPP) [5]. The objective of the directives is to allow the programmer to define the boundaries, the type and the dependencies among the application DThreads. In this section we present the most significant directives and explain their usage.

In Table 1 we show the most relevant directives that allow the programmer to define a set of instructions either as a simple DThread or as a Loop DThread for the TFlux implementation. For the DDM-VM implementation though some parameters of the directives from Table 1 have a different structure. In Table 2 we describe these differences in the structure of the directives as well as any new parameters that we introduced in order to fully support the new operations of the DDM-VM implementation through the DDM C Preprocessor.

A simple DThread is defined by enclosing its code in a **#pragma ddm thread** and a **#pragma ddm endthread**. These directives mark the start and the end of a DThread and also define the unique identifier of each DThread (DThread ID). Currently the TFlux implementation supports a static scheduling technique, thus the programmer must also define the *Kernel* that the declared DThread will execute on. In the case of the DDM-VM implementation the *Kernel* parameter is a pair of numbers that defines the scheduling policy the programmer wants to use on that specific

Table 1: DDM pragma directives.

`#pragma ddm startprogram` `#pragma ddm endprogram`	Define the start and the end of a DDM program
`#pragma ddm block` *ID* `#pragma ddm endblock`	Define the start and the end of a block of threads with identifier *ID*
`#pragma ddm thread` *ID* `kernel` *NUMBER* `import (`*TYPE NAME*`) export (`*NAME*`)` `#pragma ddm endthread`	Define the boundaries of a DDM thread with identifier *ID* and the kernel *NUMBER* to execute on and the data to import/export

`#pragma ddm for thread` *ID* `ilc`
`[NUM,NUM,NUM,NUM,NUM,NUM] depends (`*ID*`)`
`#pragma ddm endfor`
	Define the boundaries of a DDM loop thread with identifier *ID*, its consumers with *ilc* and its producers with the *depends*

`#pragma ddm kernel` *NUMBER*	Declare the number of kernels to be used
`#pragma ddm var` *TYPE NAME*	Declare a shared variable with *NAME* and *TYPE*
`#pragma ddm private var` *TYPE NAME*	Declare a private variable with *NAME* and *TYPE*

Table 2: New parameters for DDM-VM support.

`kernel (`*SCHEDULING_POLICY* `:` *SCHEDULING_VALUE*`)`	Define the scheduling policy that the specified DThread will follow
`arity` *NUMBER*	Define the number of nested loops that a DThread represents
`readycount` *NUMBER*	Explicitly define the number of consumers of DThread
`update (THREAD_ID : START : END)`	Update the ending threads' consumers
`import (address : size : flag : expression : reference variable)`	Define the variables that will be imported in the specified DThread
`export (address : size : flag : expression : reference variable)`	Define the variables that will be exported by the specified DThread

DThread and a value that might be needed, depending on the scheduling policy the user chooses (see Table 2).

The definition of loops is supported in a similar way. By enclosing the code of a for loop in **#pragma ddm for** and **#pragma ddm endfor** directives, we define a *Loop DThread*, that all its iterations will be executed in parallel. In the case of the TFlux implementation, the *kernel* statement, that defines where each DThread will be executed, is not necessary as the loop iterations will be evenly distributed to all kernels automatically by the preprocessor. In the case of the DDM-VM implementation though the *kernel* statement is needed as to define the scheduling policy that the user wishes to apply on the execution of the loop iterations. For the DDM-VM implementation we also use the *arity* parameter that describes the depth of the nested loops that are to be parallelized. Currently this is used on the declaration of a simple DThread while the user removes the *for* loop statements from the code and only keeps the body of the loop. This parameter is used to expand the parallelism to the internal loops in the case of nested loops.

In order for the runtime system to know when a DThread is ready for execution a counter value is kept that denotes the number of consumers a DThread is waiting upon being scheduled for execution. This counter is called *readycount*. Note that for the TFlux implementation this field is not mandatory, thus it does not appear in Table 1, as the number of consumers will be automatically inferred by the declaration of the explicit dependencies we will discuss later in this section.

In the TFlux implementation a different approach is pursued. The preprocessor allows dependencies between loop iterations of different Loop DThreads using the *ilc* statement. As depicted in Table 1 the *ilc* statement is a six-tuple entity of a type, the consumer loop identifier, three numeric values and a value indicating the scheduling type (Chunk or Round-Robin scheduling) of the consumer and producer loops. The type and the three numeric values are used to calculate the iteration of the consumer loop based on the iteration id of the producer Loop DThread. In this case the *readycount* statement must be used to explicitly define the Ready Count value of the consumer Loop DThread.

To define the producer/consumer relationships between the DThreads in a DDM application we need to consider the data that the DThreads consume and produce. Using the *import* and *export* statements on a DThread directive the preprocessor will know which variables each DThread will consume and produce. Thus, it will automatically create a dependence between the DThread that produced that specific variable and the DThread that will eventually consume it. In some cases expressing the data dependencies through the produced and consumed data is not possible. For example, when we have arrays as produced or consumed data. To explicitly define dependencies between DThreads in TFlux we use the *depends* statement that denotes the producers of a DThreads. Note that for a Loop DThread in TFlux only the *depends* statement can be used to define the producers of that loop.

In the case of the DDM-VM implementation though, instead of the producers, we denote the consumers of a specific DThread (either simple or loop) by using the *update* statement on the *#pragma ddm endthread* and the *pragma ddm endfor* directives as described in Table 2. The *START* and *END* will be used in the case we want to update multiple iterations of the consumer loop.

Any type of DThread in a DDM program must be enclosed in a DDM block at all times. This can be done by enclosing the definitions of DThreads in a set of **#pragma ddm block** and **#pragma ddm endblock** directives. Any number of blocks can be used. The DThreads within a block will be executed in parallel as long as the dependencies among them allow it but blocks are strictly executed sequentially between them. To define a complete DDM program enclose all DDM blocks in a set of **#pragma ddm startprogram** and **#pragma ddm endprogram** directives. These directives define the complete DDM application. Finally, before executing the Preprocessor to create the DDM application, the user must also define the number of kernels that will be used in the execution using the **#pragma ddm kernel** directive.

4 Eclipse Plug-in Suite

Apart from the significant programming effort needed to develop Dataflow applications today, we also consider the lack of programming tools that help the users develop such applications. In this section, we present a plug-in suite for the Eclipse platform that provides significant help in using the DDM programming directives presented in Section 3. The plug-in provides support for both DDM implementations presented in this paper. The main purpose of this plug-in is to give an easier way to the programmers to use the DDM directives and a one-step generation of DDM executables. When creating a new DDM application in Eclipse, the user will be prompt to choose the target implementation, between TFlux and DDM-VM as shown in Figure 4. This will automatically load the modules of the chosen implementation and provide the necessary information concerning the syntax of the directives to be used. Also, by choosing the target platform the preprocessor will also add all the necessary initializations each platform needs.

The DDM Eclipse plug-in is composed of three modules:

1. The *Content Assistant*, that provides a drop-down list of the available directives while the user is coding;

2. The *Side Panel*, that displays a side panel next to the code that shows the available directives and their arguments;

3. The *Pre-processor integration*, that calls the DDM C Pre-processor through a button at the top of the tools bar. This process will produce the DDM executable based on target platform chosen when creating the project;

The DDM directives are mostly start-end directives. Thus, the user must define the start and the end of a code that will be enclosed in a DDM directive. For this reason, the plug-in also provides an auto-closing functionality that will auto-close an open directive when pressing the ENTER key at the end of the corresponding start directive.

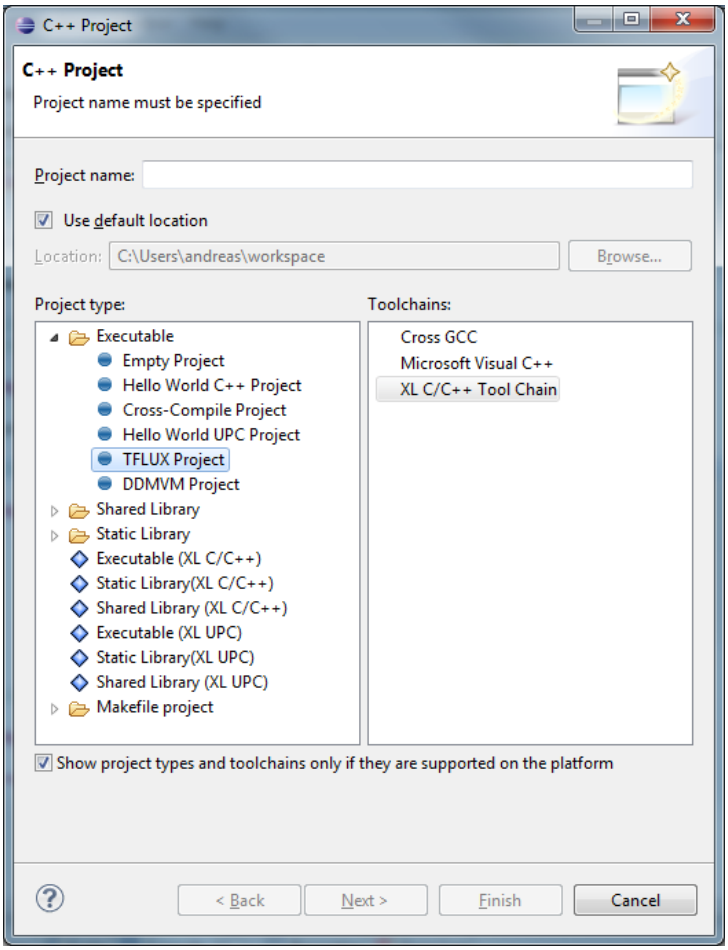

Figure 4: Create a new DDM project window in Eclipse.

4.1 The Content Assistant

Figure 5 illustrates the basic functionality of the content assistant module. When pressing the CTRL+SPACE keys at any point of writing a directive, a proposal window will appear with all the possible arguments that specific directive has. By selecting one of the possible arguments the module will automatically insert it at the end of the current directive.

The content assistant module also provides all the predefined arguments for any directive. An example is the scheduling policy options the implementation supports, as seen in Figure 6.

4.2 The Side Panel

Figure 7 shows the DDM side panel module. This module consists of two lists, the *Sample View* list and the *Property* list. The Sample View contains the directives that are available to the user. A user can insert into the code a specific directive by just clicking on a directive from the Sample View list. The Property list, as the name suggests, contains the properties of each directive along with the available parameter values. Using the Property list the user can define the parameter values without having to hand-code them. Also, if the user moves the cursor an already complete directive, the values of the parameters for the directive will appear in the Property list.

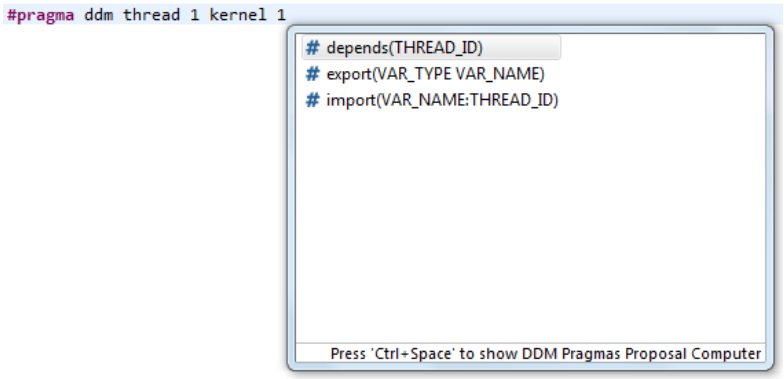

Figure 5: The content assistant module listing the available parameters of a DDM directive.

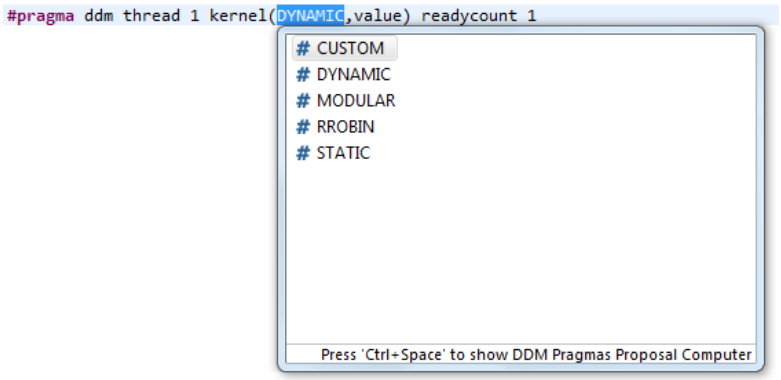

Figure 6: The content assistant module listing the available predefined argument parameters.

5 Case Study

In this section we will show the benefits of using the programming directives through the Eclipse platform, going through a step-by-step development of a simple DDM application for both the TFlux and the DDM-VM implementations. We will use Eclipse screenshots as a proof of concept of our implementation and show that using the DDM programming tool-chain can make easier and less time consuming the programming of DDM applications for a wide range of architectures.

The first step and the only one that the user will have to do by hand is identifying the potentially parallel sections of the sequential code and insert the appropriate directives. As a sequential code, we will use the Matrix Multiplication paradigm, where we will use one DThread for the initialization of the input data and one Loop DThread that will execute the parallel loop of the application.

In Figure 8 we depict the syntax of a TFlux DThread where we declare the boundaries of the DThread, the identity number of the specific DThread and the kernel it will execute on. Notice that we use neither the *import/export*, nor the *depends* statements on this DThread. Being the first DThread in the application means that it has no producers, thus this DThread will be executed first.

In Figure 9 we depict the syntax of a DDM-VM DThread. The difference with the previous TFlux example is that we declare the scheduling policy that this DThread will follow upon execution and we explicitly define its *readycount* value. As mentioned before, in contrast to the TFlux implementation where we declare the producers of the thread, in the DDM-VM we declare its consumers. Thus, at the *#pragma ddm endthread* directive we use the *update* statement to define that at the end of this

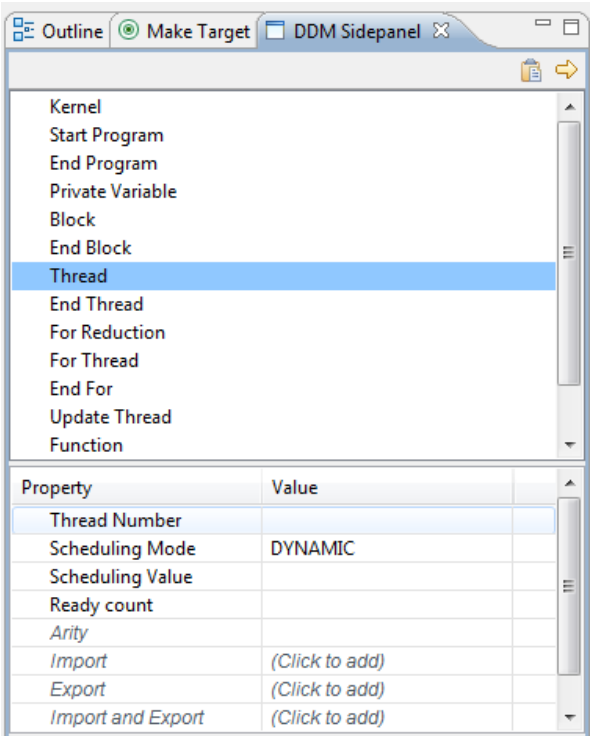

Figure 7: The side panel module providing all necessary information about the arguments of all the directives and their values.

```
#pragma ddm thread 1 kernel 1

//INITIALIZE ARRAYS
for(i=0;i<NROW;i++)
{
    for(j=0;j<NCOL;j++)
    {
        inputArrayA[i][j]= i*NCOL+j;
        inputArrayB[i][j]= j*NCOL+i;
    }
}

#pragma ddm endthread
```

Figure 8: Example of a TFlux DThread declaration.

DThread, DThread with identity 2 (THREAD_2) must by updated. As we show later, THREAD_2 is a Loop DThread and multiple iterations of that DThread must be updated after THREAD_1 finish. For this reason, we use the extra fields of *START* and *END* that are defined as the first and last iterations of the loop respectively. The reason we use the first and last iterations of the loop is that the loop iterations have no dependencies between them. In any other case we would use the only the iterations that were ready for execution.

```
#pragma ddm thread 1 kernel (MODULAR,MASK_INDX) readycount 1

//INITIALIZE ARRAYS
for(i=0;i<NROW;i++)
{
    for(j=0;j<NCOL;j++)
    {
        inputArrayA[i][j]= i*NCOL+j;
        inputArrayB[i][j]= j*NCOL+i;
    }
}

#pragma ddm endthread update (THREAD_2 : START : END)
```

Figure 9: Example of a DDM-VM DThread declaration.

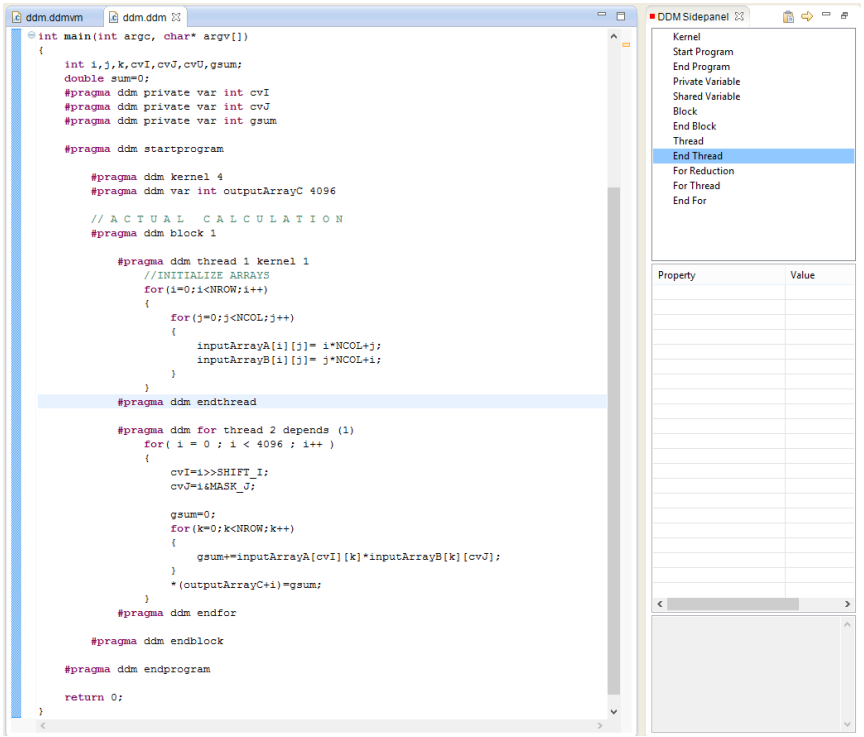

Figure 10: Matrix Multiplication using the directives for the TFlux Implementation.

Figure 11 shows the next DThread in our example for the TFlux implementation. This is a Loop DThread, thus we use the *pragma ddm for* directive to define its boundaries. As this loop will use data from the previous DThread, we have to declare its dependence on the corresponding DThread. The use of the *depends* statement shows that the execution of the current DThread will start after the execution of the DThread with identity 1 (*depends(1)*) finishes. In the case of a Loop DThread, the outer-most loop is the one to be parallelized. This suggests that the outer-most loop iterations will be evenly shared to all executing kernels.

```
#pragma ddm for thread 2 depends (1)

    for( i = 0 ; i < 4096 ; i++ )
    {
        cvI=i>>SHIFT_I;
        cvJ=i&MASK_J;

        gsum=0;
        for(k=0;k<NROW;k++)
        {
            gsum+=inputArrayA[cvI][k]*inputArrayB[k][cvJ];
        }
        *(outputArrayC+i)=gsum;
    }

#pragma ddm endfor
```

Figure 11: Example of a TFlux Loop DThread declaration.

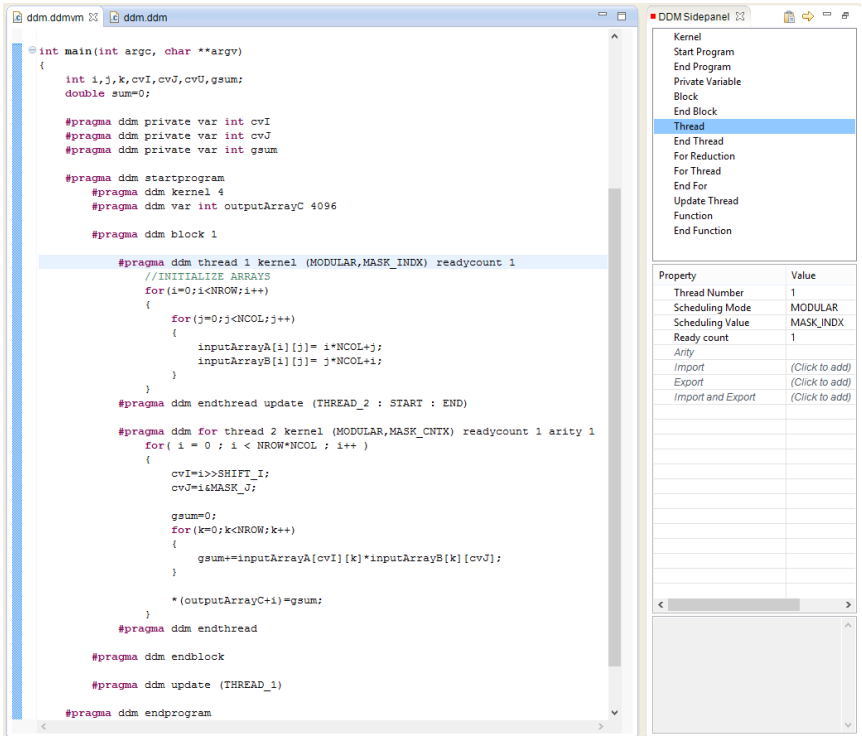

Figure 12: Matrix Multiplication using the directives for the DDM-VM Implementation.

For the DDM-VM implementation, as in the DThread example, we must declare the scheduling policy that the loop iterations will follow during the execution, the number of producers the current thread has (*readycount*) and since it is a Loop DThread, the number of nested loops that are to be parallelized (*arity*). In this example we are only parallelizing the outer-most loop so the value for the arity is 1. The declaration of the Loop DThread for the DDM-VM implementation is depicted in Figure 13.

```
#pragma ddm for thread 2 kernel (MODULAR,MASK_CNTX) readycount 1 arity 1

for( i = 0 ; i < NROW*NCOL ; i++ )
{
    cvI=i>>SHIFT_I;
    cvJ=i&MASK_J;

    gsum=0;
    for(k=0;k<NROW;k++)
    {
        gsum+=inputArrayA[cvI][k]*inputArrayB[k][cvJ];
    }

    *(outputArrayC+i)=gsum;
}

#pragma ddm endfor
```

Figure 13: Example of a DDM-VM Loop DThread declaration.

In Figures 10 and 12 we show a screenshot of the complete applications implemented with the directives using the Eclipse Platform and the DDM Plug-in.

The second step in the process of developing a DDM application is using the preprocessor. The C code augmented with the directives cannot be directly compiled with a compiler as the directives don't mean anything for a commodity C compiler. The DDM C Preprocessor will transform the code with the directives into a pure parallel C code that can then be compiled. Thus, produce the final DDM executable code.

To show the benefits of the tool-chain we compare the number of code lines of the implementations. We selected two applications, both implemented with directives for both the DDM implementations. In Figure 14 we demonstrate the lines of code (LOC) a user would have to write either using C augmented with DDM directives or natively write DDM parallel code. The number of code lines that have to written using the directives is significantly smaller compared to the output code of the preprocessor for both implementations. For the DDM-VM the code lines are also less than those of the TFlux implementations. This is mostly due to the code inserted to application, that will communicate with the runtime system and mainly the scheduling unit. It is only an implementation issue.

6 Conclusions and Future Work

In this paper we presented a programming tool-chain for Dataflow Multithreading and specifically for the DDM model that follows the principles of Dataflow at a coarser granularity. This tool-chain invokes a set of programming directives with which the user can expose the available parallelism by expressing the data dependencies among the different DThreads in an application. To translate these directives into usable parallel code we presented the DDM C Preprocessor that will transform the C code augmented with the DDM directives into DDM parallel code that can later be compiled to a binary executable using any commodity C compiler. Using this tool-chain, we can produce

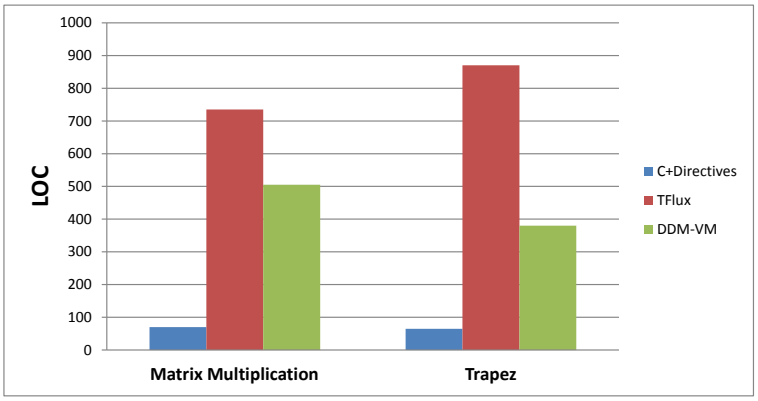

Figure 14: Lines of code using the DDM directives compared to the output of the preprocessor for two distinct applications.

dataflow parallel code for different platforms, in this paper the TFlux Platform and the DDM-VM, while significantly reduce the programming effort.

As future work we aim to support more platforms and different parallel architectures through this tool-chain. The architecture followed to build the tool-chain and the DDM C Preprocessor enables the support of new Dataflow platforms with changes only to the back-end of the platform, that is the produced parallel code. The front-end, that is the DDM directives, is not necessary to be changed in the case of a new platform. We also consider the possibility of translating OpenMP [7] directives to DDM directives, thus translating OpenMP code to Dataflow code.

References

[1] Stavrou, Kyriakos, et al., *TFlux: A portable platform for data-driven multithreading on commodity multicore systems.*, Parallel Processing, 2008. ICPP'08. 37th International Conference on. IEEE, 2008.

[2] Arandi, Samer, and Paraskevas, Evripidou, *Programming multi-core architectures using dataflow techniques.*, SAMOS'10: Proceedings of the 10th International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation, 2010.

[3] Kyriacou, Costas and Paraskevas, Evripidou, and Pedro, Trancoso, *Data-driven multithreading using conventional microprocessors.*, Parallel and Distributed Systems, IEEE Transactions on 17.10 (2006): 1176-1188.

[4] Chen, Thomas, et al., *Cell broadband engine architecture and its first implementationa performance view.*, IBM Journal of Research and Development 51.5 (2007): 559-572.

[5] Trancoso, Pedro and Kyriakos, Stavrou and Paraskevas, Evripidou, *DDMCPP: The data-driven multithreading C pre-processor.*, The 11th Workshop on Interaction between Compilers and Computer Architectures, 2007.

[6] Kyriakos, Stavrou, *The TFlux Platform: Dataflow Multithreading Execution on Commodity Multiprocessor Systems.*, PhD Dissertation, University of Cyprus, 2009.

[7] Dagum, Leonardo, and Ramesh, Menon, *OpenMP: an industry standard API for shared-memory programming.*, Computational Science & Engineering, IEEE 5.1 (1998): 46-55.

[8] Moore, Gordon E., *Cramming more components onto integrated circuits.*, Proceedings of the IEEE 86.1 (1998): 82-85.

[9] Dennis, Jack B. and David, P. Misunas, *A preliminary architecture for a basic data-flow processor.*, ACM SIGARCH Computer Architecture News 3.4 (1974): 126-132.

[10] Dennis, Jack., *First version of a data flow procedure language.*, Programming Symposium, Springer Berlin/Heidelberg, 1974.

[11] Nikhil, Rishiyur S., and Keshav, K. Pingali., *I-structures: Data structures for parallel computing.*, ACM Transactions on Programming Languages and Systems (TOPLAS) 11.4 (1989): 598-632.

[12] J. Howard and et al., *A 48-core ia-32 message-passing processor with dvfs in 45nm cmos.*, Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International, pages 108 –109, Feb. 2010.