Programming Abstractions and Toolchain for Dataflow Multithreading Architectures

Kyriakos Stavrou[1], Demos Pavlou[2], Marios Nikolaides[3], Panayiotis Petrides, Paraskevas Evripidou[4], and Pedro Trancoso[4]
Department of Computer Science
University of Cyprus, Cyprus
{tsik, cs04dp1, cs04mn1, csp7pp5, skevos, pedro}@cs.ucy.ac.cy

Zdravko Popovic and Roberto Giorgi[4]
Department of Information Engineering
University of Siena, Italy
{popovic, giorgi}@dii.unisi.it

Abstract

The need to exploit multi-core systems for parallel processing has revived the concept of dataflow. In particular, the Dataflow Multithreading architectures have proven to be good candidates for these systems. In this work we propose an abstraction layer that enables compiling and running a program written for an abstract Dataflow Multithreading architecture on different implementations. More specifically, we present a set of compiler directives that provide the programmer with the means to express most types of dependencies between code segments. In addition, we present the corresponding toolchain that transforms this code into a form that can be compiled for different implementations of the model. As a case study for this work, we present the usage of the toolchain for the TFlux and DTA architectures.

1 Introduction

The emerging multi- and many-core processors have been proposed in order to overcome the serious performance, power and design limitations of the traditional monolithic single-core processors [3, 4, 9]. While we observe an increasing number of cores on every new generation of such processors, the efficient utilization of such parallel resources is already a major challenge for future program developers. Traditional programming models and languages that are based on barrier and lock paradigms are not able to exploit parallelism at finer-grain, loosing some of the opportunities for concurrent execution. This issue becomes more serious as the scale of the machines increases.

In order to address this challenge, the dataflow model has been revived by some research groups [1, 6, 14, 15]. It is known that with dataflow, it is possible to exploit the maximum amount of parallelism that exists in an application. Nevertheless, the first implementations suffered from very large overheads limiting the applicability and consequently the success of the model [2]. In order to overcome these issues, in the recent approaches, dataflow is enforced at a coarser granularity, that of sequences of instructions or Threads [1, 5, 6, 14]. These approaches follow the *Dataflow Multithreading* model. Two such approaches are TFlux [14] and DTA [6]. Both architectures are based on execution models that schedule threads in a dataflow-like way, *i.e.* Threads are scheduled for execution based on data availability.

The major issue in introducing a new model is that either a compiler tool needs to be developed in order to transform the existing applications for execution under the new model, or alternatively, applications need to be re-written according to that model. The former is a solution that would result in a better adoption of the new model. Such an automated parallelizing compiler is usually a very complex tool which requires a significant effort for its implementation. The latter, requesting the users to completely re-write their applications, does not usually lead to a success of the model as it is an extremely time-consuming process.

Consequently, one feasible option is to provide ways to

[1]Currently at Intel Barcelona Research Center (IBRC), Intel Labs
[2]Currently at Universitat Politecnica de Catalunya (UPC)
[3]Currently at University of Illinois at Urbana-Champaign (UIUC)
[4]Members of the HiPEAC FP7 Network of Excellence.

augment programs written in conventional languages with some sort of special library function calls, new data types and/or keywords and compiler directives. Many of the existing approaches, such as OpenMP [10] and CUDA [8], have adopted this solution. For the dataflow models mentioned before we decided to also adopt this approach, in particular to provide a set of compiler directives that allows the programmer to augment existing application codes.

The objective of this work is to define a set of directives adequate for expressing the concepts of a generic *Dataflow Multithreading* model. This approach allows for the same augmented program to be used by different architectures.

Since the proposed solution is to have the programmers augment their code with compiler directives, there is a need for a new toolchain that transforms this code into a program that may execute on the target architectures as depicted in Figure 1. Once the input code is the same for different architectures, the first component of this toolchain will be common. The basic operation of this tool is to extract the information added by the programmer through the directives. Obviously, the second component will be architecture-dependent and will output code for the corresponding target architecture. The third component will be used to produce the executable binary during the last step of the process.

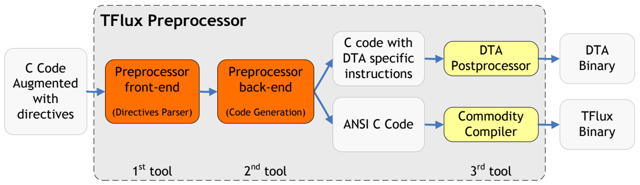

Figure 1. Dataflow Multithreading Toolchain.

The contributions of this work are twofold. Firstly, this work defines a set of compiler directives that are able to describe *any* type of dependency in the abstract Dataflow Multithreading model. The second contribution regards the toolchain which, given augmented code, produces code for execution for different Dataflow Multithreading architectures. Currently this tool supports the TFlux and DTA Dataflow Multithreading systems. Notice that adding support in the toolchain for a new architecture requires minimum effort.

The rest of this paper is organized as follows. Section 2 presents the Dataflow Multithreading model of execution and its basic program constructs. Section 3 presents the details of the TFlux and DTA. In Section 4 we discuss the proposed compiler directives and the components of the toolchain. Section 5 shows as case study the augmented code for two simple application kernels and finally, Section 6 concludes this work.

2 Dataflow Multithreading

The principal element in the *Dataflow Multithreading* model is the *Thread*. Each Thread corresponds to a different portion of the application's static code and can be of any size (regarding the number of static or dynamic instructions). This code can include any type of programming constructs such as function calls, loops and control flow operations. In addition to their code, Threads are characterized by their *input* and *output* data, *i.e.* data that is used and produced by the Thread, respectively. The input and output data of the different Threads form a *Thread Dependency Graph*. Threads are executed in parallel unless a data dependence exists among them. Inside each Thread, instructions are executed in control flow order allowing the processor or the compiler to apply any optimizations (e.g. out-of-order execution).

The concept of Threads is also applicable to loops, where usually each iteration is represented and executed by a different Thread following the dynamic dataflow model. The dependencies for the loops can either be at the loop level (no iteration of the dependent loop can proceed unless *all* iterations of the producer loop have been completed) or at a finer-grain level, that of loop iterations. This means that an iteration of the dependent loop can proceed without the need to wait for *all* iterations of the producer loop to complete, but only wait for those iterations it truly depends on. As we will explain later in Section 4.1.4, this feature has an important contribution to the performance of the system.

In this work we investigate two *Dataflow Multithreading* models: the Data-Driven Multithreading (DDM) model supported by the TFlux architecture and the Scheduled Data-Flow (SDF) model supported by the DTA architecture.

For both DDM and SDF, as well as any other non-blocking Dataflow Multithreading model, the application Threads are scheduled for execution only when all their inputs are ready. Threads execute in a *non-blocking* way, *i.e.* until completion. If the output results of a Thread are used as input for other Thread(s) then the completion of the former may trigger the execution of the latter.

The management of the scheduling is performed by the *Thread Scheduler*, which may be implemented in either software (TFlux) or hardware (TFlux and DTA). The Thread Scheduler assigns the execution to a pre-determined compute element (*static scheduling* as for TFlux) or to any of the free compute elements in the system (*dynamic scheduling* as for DTA). The scheduling decision is performed based on the value of per-thread counters. These counters are initialized as the number of input variables necessary for the corresponding Thread. Each time a Thread produces a result that is used by another Thread, the counter of the latter is decremented. When it reaches 0 the Thread is deemed executable.

Regarding the data exchanged between Threads, it may be passed through the regular memory hierarchy (as for TFlux) or using a special memory component (Frame for DTA). The use of the latter requires special instructions which must be added to the ISA of the system.

3 Dataflow Multithreading Architectures

In this Section we provide a brief description of the two Dataflow Multithreading platforms targetted by the proposed toolchain. Notice that despite their differences, the toolchain supports both of them transparently.

3.1 TFlux Portable Platform

TFlux does not refer to a specific implementation on a certain machine but rather to a collection of abstract entities that allow execution under the DDM model on a variety of computer systems. As such, the main objective of TFlux is to serve as a virtualization platform for the execution of DDM programs.

Figure 2 depicts the layered design of the TFlux system. In particular, the top layer, which is the one programmers use to develop DDM applications, abstracts all details of the underlying machine. The Runtime Support runs on top of an unmodified Unix-based Operating System and hides all DDM-specific details such as the particular implementation of the Thread Scheduler. One of the primary responsibilities of the TFlux Runtime Support is to dynamically load the Threads onto the Thread Scheduler and invoke all related operations required for DDM execution.

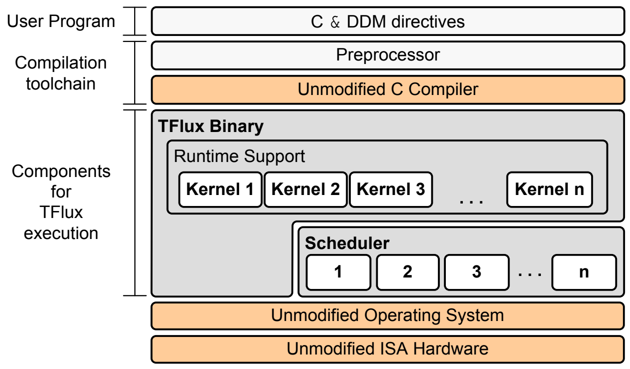

Figure 2. The layered design of the TFlux system.

3.2 DTA Platform

DTA [6] is an architecture that derives from the SDF [7] model. Threads communicate with each other in a producer-consumer fashion and a thread will start its execution only when all its data are ready in a local memory. As can be seen from Figure 3 , the Processing Elements (PE) in DTA are grouped into nodes. The number of PEs in a node is limited by the wire delay on the chip. In comparison to SDF, DTA adds the concept of clustering the resources in order to address the wire-delay problem. A fully distributed scheduler and a communication protocol for exchanging synchronization messages are implemented.

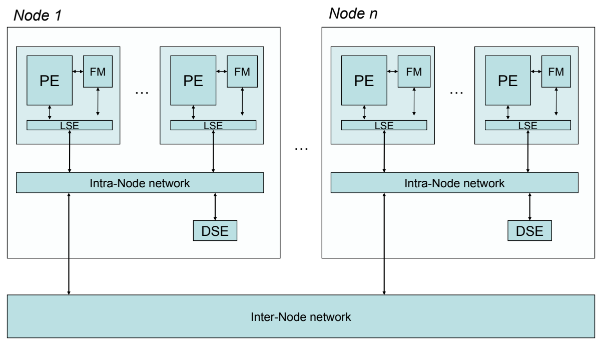

Figure 3. High level view of DTA architecture.

DTA requires three specific hardware structures and a small number of ISA extensions. The hardware structures are: (i) frame memory - fast memory, associated with each processing element; (ii) Distributed Scheduler Element (DSE) - responsible for dynamically distributing the workload among the processors in the node and for forwarding the workload to other nodes when no internal resources are available; (iii) Local Scheduler Element (LSE) - manages local frames and forwards requests for resources to a DSE. The high level view of the DTA architecture is shown in Figure 3. In order to manage the lifetime of each thread, DTA requires a few additional instructions in the ISA (creating a thread, read and write to a frame memory, and releasing a frame).

4 Compiler Directives and Toolchain

4.1 Compiler Directives

The main purpose of the compiler directives is to allow the user to define the boundaries, the type and the dependencies among the application's Threads. This Section presents the most important directives together with simple examples of their usage. The complete set of the Compiler Directives is listed in [13].

4.1.1 Threads

A Thread is defined by enclosing its code in a set of ***#pragma ddm thread*** and a ***#pragma ddm endthread*** directives (Figure 4). These directives define the *begin* and *end* points of the Thread respectively. Moreover the ***#pragma ddm thread*** directive defines the unique identifier of the Thread (Thread Id) which in the example depicted in Figure 4 is equal to 4. The current version of TFlux applies a static scheduling technique, *i.e.* the Threads each TFlux Kernel executes are defined statically. As such, the Preprocessor requires the user to define which Kernel executes a particular Thread (notice the *Kernel 2* part of the directive in the example of Figure 4). Given that for DTA the Threads are scheduled dynamically to the different nodes, this parameter is ignored during generation of DTA code.

```
#pragma ddm thread 4 kernel 2
   x=sin(y);
#pragma ddm endthread
```

Figure 4. Example of Thread declaration using compiler directives.

Data Import/Export These directives provide support for the definition of the data produced and consumed by a Thread. This is achieved using the ***import*** and ***export*** parts of the ***#pragma ddm thread*** directive (Figure 5). The preprocessor automatically creates a dependence between the Thread that produces a variable (*e.g export x*) and the Thread that consumes this variable (*e.g. import x*).

```
#pragma ddm thread 4 kernel 2 import(double y) \
                              export(x)
   x=sin(y);
#pragma ddm endthread

#pragma ddm thread 5 kernel 1 import(double x) \
                              export(z)
   z=x*x*x-x*x;
#pragma ddm endthread
```

Figure 5. Example of Thread declaration using *import/export* statements.

Explicit Dependencies. The producer/consumer relationship between Threads can also be defined explicitly using the ***depends*** statement of the ***#pragma ddm thread*** directive (Figure 6). This feature is useful for cases where the *import*/*export* statements can not express the data dependence. Such a situation is when these dependencies regard complete arrays.

Notice that although having *arbitrary* dependencies between the Threads comes natural for dataflow architectures, this is not the case for existing models of execution like the OpenMP standard. In particular, as explained by Sinnen *et al.* [12], for OpenMP to support and benefit from such arbitrary dependencies the standard needs to be extended.

```
#pragma ddm thread 4 kernel 2
  *x=sin(*y);
#pragma ddm endthread

#pragma ddm thread 5 kernel 1 depends(4)
   *z=(*x)*(*x)*(*x)-(*x)*(*x);
#pragma ddm endthread
```

Figure 6. Example of Thread declaration using the *depends* statement.

All-Kernels Threads The directives also allow the programmer to define Threads that are to be executed by *all* Kernels or nodes, *i.e.* the preprocessor creates multiple instances of this Thread and each such Thread is assigned for execution to a different Kernel. This is achieved by replacing the ***kernel kernelID*** statement of the ***#pragma ddm thread*** directive with the ***kernel all*** statement (Figure 7). This feature is very helpful for Threads that initializate private variables or execute segments of the code performing Single-Program-Multiple-Data (SPMD) operations.

```
#pragma ddm thread 4 kernel all
   mySum=0;
#pragma ddm endthread
```

Figure 7. Example of Thread declaration using the *kernel all* statement.

4.1.2 Loops

Definition of loops is also supported by the proposed directives. The basic declaration of a Loop, which has always the form of a *for-loop* is depicted by Figure 8. The code of this Loop is enclosed in a set of ***#pragma ddm for*** and ***#pragma ddm endfor*** directives. Notice that this declaration regards a *parallel* for-loop, *i.e.* a loop where *all* its iterations can proceed in parallel (*DO-ALL* loop).

```
#pragma ddm for thread 4
   for(cv=0;cv<1024;cv++)
   {
      a[cv]=sin(cv);
   }
#pragma ddm endfor
```

Figure 8. Example of a Loop.

Notice that loops are always executed by *all* Kernels of the system with the preprocessor distributing the iterations to the different Kernels as evenly as possible. When the number of loop iterations is not a multiple of the number of Kernels, the preprocessor distributes iterations to Kernels in such a way that the imbalance is never greater than one loop iteration.

Unrolling The preprocessor provides automatic unrolling for Loops which is achieved with the use of the ***unroll*** statement. For example the Loop depicted in Figure 9 is set to be unrolled 8 times. Unrolling the loop is often very helpful as increasing the size of the Threads helps to better amortize the parallelization overheads. The preprocessor automatically handles all side-effects of unrolling, such as, replicating the code, adjusting the increase of the control variable and the corresponding change in the value of the upper bound of the loop.

```
#pragma ddm for thread 4 unroll 8
  for(cv=0;cv<1024;cv++)
  {
    a[cv]=sin(cv);
  }
#pragma ddm endfor
```

Figure 9. Example of a Loop with unrolling.

Reduction A common operation performed by parallel loops is for all loop iterations to lead to a single value which is called "reduction". The preprocessor provides special support for such Loops, *i.e.* Loops that perform a reduction operation (Figure 10). This is achieved through the use of the ***reduction*** statement.

```
#pragma ddm for thread 1 reduction sum + double totalSum
  for(i=0;i<1024;i++)
  {
    localSum+=i;
  }
#pragma ddm endfor
```

Figure 10. Example of a Loop with reduction.

Referring to the example depicted in Figure 10, each Kernel will calculate the summation of the iterations it executes on the *localSum* variable. To find the total sum, *i.e.* the summation for *all* iterations of the loop, it is necessary to add all these *localSum* variables. The necessary code to perform these operations is automatically generated by the the preprocessor.

In addition to the simple reduction operations (summation, subtraction and multiplication), the preprocessor allows its user to *express* more complex reduction operations using custom functions. The only responsibility for the user is to define this function; all other details are handled by the preprocessor automatically. As an example, Figure 11 depicts a Loop that calculates the minimum and maximum value of an array of integer values.

```
#pragma ddm for thread 1 reduction \
        redFun(Max, Min, int localMax, int localMin)
  for(i=0;i<128;i++)
  {
    if(A[i]>localMax)
      localMax=A[i];

    if(A[i]<localMin)
      localMin=A[i];
  }
#pragma ddm endfor

void redFun(int* Max,int* Min,int localMax,int localMin)
{
  if(*Max<localMax)
    *Max=localMax;

  if(*Min>localMin)
    *Min=localMin;
}
```

Figure 11. Example of a Loop with reduction with function.

4.1.3 Loop Dependencies

Loops can depend on other Loops and Threads. When a Loop depends on another Loop, no Thread of the second loop can start its execution unless *all* Threads of the first loop have completed. Similarly, when a Loop depends on a Thread, no Loop Thread can start its execution unless the Thread they depend on has completed. As for the situation where a Thread depends on a Loop, the Thread can start is execution only when *all* Loop Threads have completed. Notice that it is possible for a Loop to depend on multiple Threads or Loops. All these dependencies can be expressed with very simple compiler directives as depicted in Figure 12 .

```
#pragma ddm for thread 3
  for(cv=0;cv<1024;cv++)
  {
    a[cv]=b[cv]*c[cv];
  }
#pragma ddm endfor

#pragma ddm for thread 4 \
        depends(3)
  for(cv=0;cv<1024;cv++)
  {
    a[cv]=sin(cv);
  }
#pragma ddm endfor
```

(a)

```
#pragma ddm for thread 3
    x=sin(y);
#pragma ddm endfor

#pragma ddm for thread 4 \
        depends(3)
  for(cv=0;cv<1024;cv++)
  {
    a[cv]=a[cv]*x;
  }
#pragma ddm endfor
```

(b)

Figure 12. Declaration of Loop dependencies. (a) Dependencies between Loops. (b) Dependency between a Loop and a Thread.

4.1.4 Iteration Level Dependencies

The preprocessor allows expressing dependencies at the iteration-level of loops, *i.e.* dependencies between Loop Threads. As depicted in Figure 13, this is done by using the ***ilc*** statement (Iteration Level Consumers). Each ***ilc*** statement is a six-tuple entity consisting of a type, the consumer Loop identifier, three numeric values (a, b and c) and a value indicating the scheduling type (Chunk scheduling or Round-Robin) of the consumer and producer loops. The type and the three numeric values (a, b and c) are used to calculate the Iteration Id of the consumer Loop based on the Iteration Id of the Loop Thread that has completed according to the expressions depicted in Figure 13. Notice that for the second Loop the Ready Count value has been set explicitly by the programmer to be equal to 2 as each Thread of this loop depends on two Threads of the first Loop.

```
#pragma ddm for thread 1 ilc [2 2 2 0 0 0]
  for(i=0;i<1024;i++)
  {
    A[i]=i*i;
  }
#pragma ddm endfor

#pragma ddm for thread 2 readyCount 2
  for(i=0;i<512;i++)
  {
    B[i]=A[2*i+1]-A[2*i];
  }
#pragma ddm endfor
```

Figure 13. Example of Loops with Iteration Level Dependencies.

4.2 OpenMP directive comparison

Here we will just give a qualitative comparison of the DDM C preprocessor directives with the well known OpenMP [11] directives. The main advantage of the directives we used is the ability of Dataflow Multithreading to explicitly define the true data dependencies between threads, in contrast to OpenMP which, even though it supports the *nowait* clause, introduces a barrier when we have the need to synchronize the execution of each thread. More specifically, if we had the scheme of execution depicted in Figure 14-(a), the dependence between threads 2 and 4 can be explicitly declared using the Dataflow Multithreading model. On the other hand, OpenMP introduces barriers at each level of execution, meaning that if a thread has a bigger computational load at the first level, then no threads from the next level can start their execution unless all of the threads of the previous level have completed their execution (even though they have no real data dependencies between them). This gives the ability to the programmer to express high levels of parallelism exploiting the true data dependencies among threads.

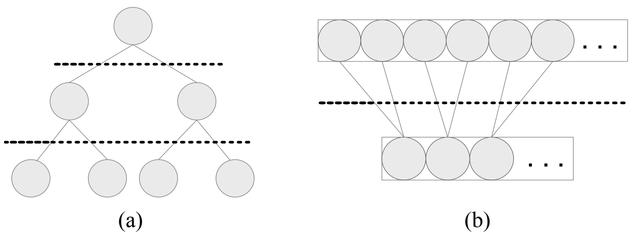

Figure 14. Example synchronization graphs with fine-grain dependencies.

Another big advantage of DDM is the pragma for iteration level dependencies. It gives the ability to express iteration level dependencies among loops. More specifically, when iterations 11 and 12 (Figure 14-(b)) of the first loop finish their execution, the data needed for iterations 21 and 22 of the second loop are produced and ready. Having iteration level dependencies enables the execution of the iterations of the second loop whose data are produced, without the need to wait for the execution of the first loop entirely in order to proceed to the iterations of the second loop.

Multiple applications are known to exist with dependencies at the level of loop iterations. Examples include the Conjugate Gradient (CG), LU decomposition, the deblocking filter of H.264 encoding, and the Smith-Waterman algorithm. Expressing the dependencies between two loops at the iteration level, when this is applicable, instead of having them depend at the iteration level, removes unnecessary synchronization point with a consequent increase in performance. Stavrou *et al.* [13] showed this performance improvement to be in the order of 10-20%. Although iteration-level dependencies are not inherently supported by widely used parallel models, such as the OpenMP standard, they are natural for dataflow architectures.

4.3 Preprocessor

The Preprocessor developed within the context of this work was based on the DDM-C-Preprocessor [16]. This is a tool that takes as input a regular C code program along with compiler directives (presented in the previous sections) and outputs a C program that includes all runtime support code necessary for the program to execute on TFlux or DTA.

The Preprocessor front-end is a parser tool which is independent of the Dataflow Multithreading implementation, i.e. its task is to parse the compiler directives and then pass the information to the back-end to produce the code corresponding to the target architecture. The back-end is built as the actions of a grammar for the compiler directives and is dependent on the target architecture and implementation. Its task is to generate the code required for the runtime support.

4.4 Porting the TFlux Preprocessor to DTA

Since TFlux and DTA architectures share the concept of *Dataflow Multithreading* at the high level it was easy to port the TFlux preprocessor to support the DTA architecture as well. As explained earlier the Preprocessor operates in two phases, pragma parsing and code generation. It is only this second step that has been extended to produce code for DTA. As an example, consider the code that needs to be invoked when a Thread completes its execution. When during the code generation phase the preprocessor finds a ***#pragma ddm endthread*** directive it inserts the appropriate architecture-specific code.

Although for TFlux the output code can be directly compiled by a commodity C compiler, this is not the case for DTA due to the assembly instructions that need to be inserted into the code.

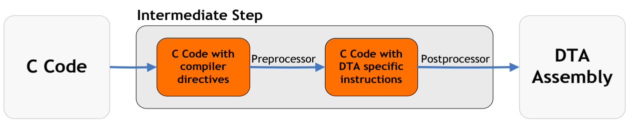
Intermediate Step
C Code
C Code with compiler directives
Preprocessor
C Code with DTA specific instructions
Postprocessor
DTA Assembly

Figure 15. DTA translation flow.

In particular, for DTA, the output of the preprocessor is ANSI C code with DTA-specific instructions which are represented as function calls. As depicted by Figure 15, we extended the back-end of the C compiler in order to have support for specific ISA instructions (notice that this requirement does not affect the compilation process for TFlux). As such, the toolchain presented in this work is able to produce binary for both the TFlux and DTA architectures from the same high-level code, *i.e.* the application's code with the compiler directives.

5 Case Study

In this section we will show, on several simple application kernels, the benefits of using this programming abstraction in both mentioned Dataflow Multithreading architectures. We use these examples only as a proof of concept and not for any performance comparison. The purpose is to show that with this kind of abstraction, programming for a wide range of architectures can be easier and less time consuming.

First, we will describe the complete program flow, from C code to binaries. The first step in the program flow is identifying potentially parallel code portions and inserting appropriate pragmas (thread, for loop...). This is the only step that the programmer needs to do by hand, and it is not architecture dependant. The next step is using the preprocessor. In order to use the preprocessor for different architectures, we need to have one implementation per target architecture. In other words, each implementation of the preprocessor processes the code in a way that is suitable for the target architecture. For now, we have two implementations of the preprocessor for TFlux and DTA architectures. As a product of this step we have a pure C code that can be compiled, which finally produces the binary ready for execution. Depending on the target architecture, the C compiler may need to be changed. TFlux relies on "off the shelf" processors, so the ISA is like in standard processors. DTA has some specific ISA instructions, and therefore some small changes in the compiler backend need to be done.

In order to demonstrate the benefits of this programming abstraction we have used the number of code lines as a metric. We have selected several simple application kernels (the synchronization graphs of which are depicted in Figure 16) that use the most important constructs with pragmas. For each application we have three numbers. One is the number of code lines for the C code with pragmas, the other is the number of C code lines for the TFlux architecture and the last is the same but for the DTA architecture. Results are shown in Figure 17.

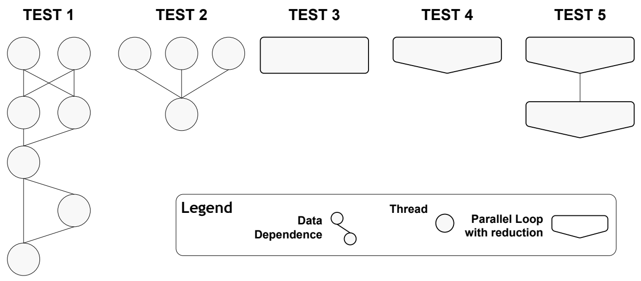

Figure 16. The synchronization graphs of the example programs.

The numbers for DTA and TFlux are different for several reasons. The most important reason is the way that these architectures schedule threads. In TFlux it is done statically, so it has to be defined in software, and in DTA it is done completely in hardware. Another reason is that DTA has some more overhead because of explicitly passing parameters between the threads - for each parameter one load and one store instruction. One more thing is that DTA has some specific ISA instructions for thread management which, in the case of TFlux, can't be done with only one instruction.

Figure 17 depicts how much effort (in terms of lines of code) a programmer can save by using this programming abstraction. The baselines are TFlux and DTA generated code, and C code with pragmas is used to see how much is actually saved. We can see that in all the cases savings are higher for TFlux. This is because TFlux requires the scheduling to be explicitly coded, and therefore there are more lines of code.

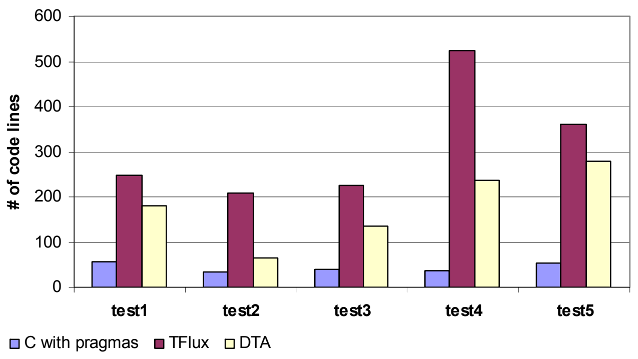

Figure 17. Programming abstraction advantages: lines of code.

In all the test cases we can see a clear benefit for the programmer when using this programming abstraction. The effort required by the programmer in coding is significantly smaller, and the modifications are done at the higher level of abstraction.

6 Conclusions

In this paper we have presented a programming abstraction that can be used in Dataflow Multithreading architectures. This abstraction is represented as a set of directives that the programmer can use to expose parallelism of the application. Together with this, we have presented a toolchain that transforms C code augmented with the above mentioned directives to binary code. The toolchain can produce binaries for different target architectures that share the same concept of dataflow at the thread level. For now, there are two such target architectures: TFlux and DTA. As demonstrated for both architectures, there is a significant reduction in programming effort when using these directives.

With this paper we wish to demonstrate that the directives and the toolchain can be efficient for architectures that share the Dataflow Multithreading paradigm. As a future work we want to measure performance and compare it to other solutions and add support for other architectures.

7 Acknowledgments

This project was partially funded by HiPEAC (EU FP6 IST 004408, FP7 IST 217068), SARC (EU FP6 FET 27648) and the Cyprus Research Promotion Foundation under the grant $\Delta\Pi E$/0505/25.

References

[1] J. Arul and K. Kavi. Scalability of Scheduled Dataflow Architecture (SDF) with Register Contexts. *ICA3PP*, 2002.

[2] Arvind and R. A. Iannucci. Two fundamental issues in multiprocessing. In *4th International DFVLR*, 1988.

[3] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, University of California, Berkeley, Dec 2006.

[4] K. D. Bosschere, W. Luk, X. Martorell, N. Navarro, M. OBoyle, D. Pnevmatikatos, A. Ramirez, P. Sainrat, A. Seznec, P. Stenstrom, , and O. Temam. High-Performance Embedded Architecture and Compilation Roadmap. *Transactions on HiPEAC*, 1:5–29, 2007.

[5] P. Evripidou and C. Kyriacou. Data driven network of workstations (D^{2}NOW). *J. UCS*, 2000.

[6] R. Giorgi, Z. Popovic, and N. Puzovic. Dta-c: A decoupled multi-threaded architecture for cmp systems. In *Proceedings of IEEE SBAC-PAD*, pages 263–270, Gramado, Brasil, Oct. 2007.

[7] K. M. Kavi, R. Giorgi, and J. Arul. Scheduled dataflow: Execution paradigm, architecture, and performance evaluation. *IEEE Transaction on Computers*, 50(8):834–846, Aug. 2001.

[8] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with cuda. In *SIGGRAPH '08: ACM SIGGRAPH 2008 classes*, pages 1–14, New York, NY, USA, 2008. ACM.

[9] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a Single-Chip Multiprocessor. In *Proceedings of the 7th ASPLOS*, 1996.

[10] OpenMP Architecture Review Board. OpenMP Application Program Interface. Vension 2.5, May 2005.

[11] OpenMP Architecture Review Board. OpenMP Application Program Interface. Vension 2.5, 2005.

[12] O. Sinnen, J. Pe, and A. V. Kozlov. Support for fine grained dependent tasks in openmp. In *IWOMP '07: Proceedings of the 3rd international workshop on OpenMP*, pages 13–24, Berlin, Heidelberg, 2008. Springer-Verlag.

[13] K. Stavrou. *The TFlux Platform: Dataflow Multithreading Execution on Commodity Multiprocessor Systems*. PhD dissertation, University of Cyprus, in preparation.

[14] K. Stavrou, M. Nikolaides, D. Pavlou, S. Arandi, P. Evripidou, and P. Trancoso. Tflux: A portable platform for data-driven multithreading on commodity multicore systems. In *ICPP '08: Proceedings of the 2008 37th International Conference on Parallel Processing*, pages 25–34, Washington, DC, USA, 2008. IEEE Computer Society.

[15] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin. WaveScalar. In *Proceedings of the 36th MICRO*, 2003.

[16] P. Trancoso, K. Stavrou, and P. Evripidou. "DDMCPP": The data-driven multithreading c pre-processor. pages 32–39, February 2007.