THE TFLUX PLATFORM: A PORTABLE PLATFORM FOR DATA-DRIVEN MULTITHREADING ON COMMODITY MULTIPROCESSOR SYSTEMS

Kyriakos Stavrou

University of Cyprus, 2009

Kyriakos Stavrou

This work presents the *Thread Flux* (***TFlux***) *Parallel Processing Platform*, a complete system that offers an efficient dataflow-like thread-based model of execution, the *Data-Driven Multithreading* (DDM), to its users using commodity components, *i.e.* unmodified Operating System, unmodified compiler and unmodified ISA hardware making it applicable to *off-the-shelf* systems. ***TFlux*** provides a complete solution from the programming toolchain to the hardware implementation. The abstraction layer ***TFlux*** exports to its users hides all the details of the underlying machine allowing different hardware configurations to support its model of execution transparently to the programmer. One key component of TFlux is the TFlux Scheduler that is responsible for Thread Scheduling based on data-availability.

The user of ***TFlux*** can develop applications using a set of simple but powerful compiler directives. Then the TFlux-C-Preprocessor converts this code to an ANSI C program that includes the Runtime Support for ***TFlux*** and all calls to the system's scheduler. This code can be compiled with a commodity C compiler resulting in a binary that is executable by any commodity Operating System on any commodity CPU processor. The layered design of ***TFlux*** has been tested on different Unix-based multiprocessor systems. Moreover, this design enabled the porting of ***TFlux*** to different machines with minimum effort.

In this work, two ***TFlux*** designs are presented: ***TFluxHard*** and ***TFluxSoft***. For ***TFluxHard*** the Thread Scheduler is a hardware unit whereas for ***TFluxSoft***, the Scheduler's functionality is

provided at the software level. As such, ***TFluxHard*** is applicable to systems that offer the ability to augment the machine with a small hardware module while ***TFluxSoft*** is directly applicable to any existing, *off-the-shelf* system.

To evaluate the ***TFlux*** designs, a benchmark suite based on real-life and synthetic applications was developed. The applications in this suite were carefully chosen in order to have different characteristics both in terms of their dynamic behavior and complexity of their dataflow graph.

For the applications of the evaluation suite, both ***TFluxHard*** and ***TFluxSoft*** show remarkable speedup and scalability. Although for most applications both achieve almost the same performance, ***TFluxHard*** shows an advantage over ***TFluxSoft*** arising from offloading the Scheduler's functionality to the hardware module. In addition, the experimental results also show that both ***TFluxHard*** and ***TFluxSoft*** are able to exploit more parallelism for applications with complex dependency graphs, compared with traditional parallel programming model approaches.

Overall, ***TFlux*** is a platform able to deliver high-performance by exploiting dataflow-like Thread scheduling on *off-the-shelf* systems through augmentation of the source code with simple compiler directives.

Kyriakos Stavrou

THE TFLUX PLATFORM: A PORTABLE PLATFORM FOR DATA-DRIVEN MULTITHREADING ON COMMODITY MULTIPROCESSOR SYSTEMS

Kyriakos Stavrou

A Dissertation

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Doctor of Philosophy

at the

University of Cyprus

Recommended for Acceptance

by the Department of Computer Science

January, 2009

Kyriakos Stavrou

© Copyright by

Kyriakos Stavrou

All Rights Reserved

2009

Kyriakos Stavrou

APPROVAL PAGE

Doctor of Philosophy Dissertation

Presented by

Kyriakos Stavrou

Research Supervisor ______________________
Pedro Trancoso

Committee Member ______________________
Marios Dikaiakos

Committee Member ______________________
Paraskevas Evripidou

Committee Member ______________________
Lasse Natvig

Committee Member ______________________
Costas Pattichis

Committee Member ______________________
Theo Ungerer

University of Cyprus

January, 2009

Kyriakos Stavrou

ACKNOWLEDGEMENTS

This thesis would not have been possible without many people's help. The first person I would like to thank is my advisor Dr. Pedro Trancoso who was always there in the good and bad times. His help was not limited to guiding and supervising this work but also at the personal level. He has supported all my professional steps but also he was there as a friend during all these years. In addition to my advisor, I would like to thank my other advisor, Professor Paraskevas Evripidou who had an important role on the design of the TFlux Platform through his comments, ideas and experience.

For parts of this work I need to acknowledge and thank other researchers. Demos Pavlou and Marios Nicolaides have contributed to the implementation of the TFluxSoft platform and the extension of the TFlux Preprocessor. The help of Demos and Marios, who I had the luck to work with for almost two years, also covered the evaluation of the TFlux platform, design and plan of optimizations for TFluxSoft and the authorship of a number of papers published during this work. Moreover, I want to thank Samer Arandi who had an important contribution on the implementation of TFluxCell. I would also like to thank Panayiotis Petrides for his help on benchmarks.

In addition, I would like to thank my committee members, Professor Constantinos Pattichis, Professor Marios Dikaiakos, Professor Lasse Natvig and Professor Theo Ungerer for their valuable comments.

I am also grateful to the organizations that partially funded this work, the Cyprus Research Promotion Foundation and the HiPEAC network of excellence.

Kyriakos Stavrou

Professor Stamatis Vassiliadis was the distinguished collaborator in the project that gave us the larger portion of funding for this work. Although this is something for which I am really grateful, what I would really like to thank him about are some other, more important things; for showing the road, for inspiring, and for reminding that kindness, wisdom and success can coexist.

Last but not least I want to thank my family; my father Avramis, my mother Xenia, my brother Stavros and of course Avra. It is not possible to define the things I thank them for just because I want to thank them for everything they did for me.

Kyriakos Stavrou

TABLE OF CONTENTS

Chapter 1: Introduction 1

1.1 The TFlux Parallel Processing Platform 4

1.2 Contributions 7

Chapter 2: Related Work 9

2.1 The Shift to Multicore Systems 9

2.2 Parallel Programming Models 11

2.3 The Dataflow Model of Computation 17

2.3.1 Dataflow Architectures 18

2.3.2 Dataflow Limitations 20

2.3.3 Hybrid Dataflow 22

2.3.4 Non-blocking Multithreading 23

2.3.5 Recent Dataflow Developments 24

2.4 Data-Driven Multithreading (DDM) 27

2.5 Data-Driven Multithreading Network of Workstations (D^2NOW) 30

2.6 Data-Driven Multithreading Chip Multiprocessor(DDM-CMP) 31

Chapter 3: The TFlux Parallel Processing Platform 32

3.1 TFlux Layered Design 32

3.1.1 Runtime Support 34

3.1.2 TFlux Kernel 35

3.1.3 The TFlux Scheduler 36

3.1.4 Compilation Toolchain 38

Kyriakos Stavrou

3.2 Basic Execution Components . . . 39
3.2.1 Data-Driven Threads (DThreads) . . . 39
3.2.2 TFlux Loops . . . 40
3.2.3 Thread Recycling . . . 50
3.2.4 Blocks . . . 51
3.3 TFlux Scheduler Basic Operations . . . 53
3.3.1 Thread Load . . . 53
3.3.2 Thread Completion . . . 53
3.3.3 Thread Update . . . 56
3.3.4 Find Ready Thread . . . 56
3.3.5 Clear TSU . . . 56
3.4 TFlux Incarnations . . . 56
Chapter 4: TFlux Preprocessor **58**
4.1 Structure of a TFlux Application Code . . . 59
4.2 Phases of the TFlux Preprocessor . . . 61
4.2.1 Phase 1: Parsing . . . 61
4.2.2 Phase 2: Creation of Output Code . . . 63
4.3 Basic TFlux directives . . . 64
4.3.1 DThreads . . . 64
4.3.2 TFlux Loops . . . 66
4.3.3 Thread Recycling . . . 72
4.4 TFlux directives Expressibility . . . 73
4.5 Limitations . . . 76

Kyriakos Stavrou

Chapter 5: TFluxHard **78**

5.1 The TFluxHard System . . . 78

5.2 TFluxHard Scheduler . . . 79

5.2.1 TFluxHard Scheduler Units . . . 80

5.2.2 Implementation of Basic Operations . . . 83

5.3 TFluxHard Scheduler Interface . . . 90

5.3.1 Load TSU . . . 90

5.3.2 Clear this TSU . . . 91

5.3.3 Flush Scheduler . . . 92

5.3.4 Thread Completed Execution . . . 92

5.4 TFluxHard Scheduler Hardware . . . 94

5.4.1 Logic Units . . . 94

5.4.2 Memory Units . . . 98

5.5 Hardware Budget Estimation . . . 101

5.6 TFluxHard Implementation Issues . . . 103

5.6.1 Current Design . . . 103

5.6.2 Possible Future Implementations . . . 104

Chapter 6: TFluxSoft **106**

6.1 SoftScheduler . . . 108

6.1.1 Implementation of Basic Operations . . . 109

6.1.2 Mutual Exclusion . . . 112

6.1.3 Upper Bound of Basic Operations Cost . . . 113

6.2 SoftScheduler Design Issues . . . 118

Kyriakos Stavrou

6.2.1 Thread to Kernel Indexing (TKI) 118

6.2.2 TUB Segmentation 120

6.2.3 Local TUB 121

6.2.4 TUB Buffers 122

6.2.5 TUB Ranges 123

6.2.6 Summary 124

6.3 TFluxSoft Scalability Issues 124

Chapter 7: TFlux Evaluation Suite **126**

7.1 Introduction 126

7.2 Real-life Applications 128

7.2.1 Matrix Multiply (MMULT) 129

7.2.2 Trapezoidal Rule for Integration (TRAPEZ) 131

7.2.3 Susan Smoothing (SUSAN) 133

7.2.4 Sorting using qSort (SORT) 135

7.2.5 Runge-Kutta (RK) 139

7.2.6 Fast Fourier Transformation (FFT) 141

7.2.7 Conjugate Gradient Method (CG) 144

7.2.8 LU 150

7.2.9 Summary 156

7.3 Synthetic Applications 158

7.3.1 Parallel Threads 161

7.3.2 Basic Loops 162

7.3.3 TFlux Advantage of Dataflow Scheduling 166

Kyriakos Stavrou

7.3.4 TFluxSoft Scalability Study . . . 171

Chapter 8: Experimental Setup **179**

8.1 Experimentation Infrastructure . . . 179

8.1.1 Virtutech Simics Full System Simulator . . . 180

8.1.2 TFluxHard Simulation: Modeling the Scheduler . . . 181

8.1.3 Systems used for Performance Evaluation . . . 183

8.1.4 Systems used for Studying Virtualization . . . 186

8.1.5 Summary . . . 186

8.2 Compilation . . . 187

8.3 Scheduling Policy . . . 187

8.4 Unrolling . . . 188

8.5 Metrics . . . 188

8.6 Collecting Statistics . . . 189

8.7 Statistical Significance . . . 190

Chapter 9: Performance Evaluation **191**

9.1 Minimum DThread Size . . . 191

9.2 Real Life Applications . . . 194

9.2.1 TFluxHard . . . 195

9.2.2 TFluxSoft . . . 204

9.3 Synthetic Applications . . . 206

9.3.1 TFlux Loops Dependencies . . . 207

9.3.2 Applications with Complex Dataflow Dependencies . . . 212

9.4 TFluxSoft Scalability - Operation with Multiple Updaters . . . 218

Kyriakos Stavrou

9.4.1 Potential of using Multiple Updaters 219
9.4.2 Performance Evaluation with Multiple Updaters 223
9.5 TFluxHard Scheduler Delay 225

Chapter 10: Virtualization and Portability 228

10.1 TFlux Virtualization 228
10.2 TFlux Portability 231
10.2.1 From TFluxHard to TFluxSoft 231
10.2.2 TFluxCell 231
10.2.3 Execution on Distributed Memory Environments 232

Chapter 11: Conclusions and Future Work 235

11.1 Conclusions 235
11.2 Future Work 236
11.2.1 TFlux Platform 237
11.2.2 TFluxHard 237
11.2.3 TFluxSoft 238
11.2.4 TFlux Preprocessor 238

Appendix A: Example of using the interface of TFluxHard-Scheduler 240

Appendix B: TFlux Directives 246

B.1 Notations 246
B.2 Program Control 248
B.3 Variables Declaration 249
B.4 Block Declaration 249

Kyriakos Stavrou

B.5 DThread Declaration 250

B.6 TFlux Loop 254

Bibliography **258**

Kyriakos Stavrou

LIST OF TABLES

1 Explanation of the operation of DThreads 41

2 The API of the TFlux Scheduler 54

3 Examples of TFlux Loops scheduling 68

4 Consumer List usage example 85

5 The configuration of the memory units of TFluxHard Scheduler. 102

6 Access pattern of SoftScheduler units for the Basic Operations 113

7 Theoretical Upper bound cost for the SoftScheduler Operations 114

8 Size of the SoftScheduler units. The size of the TUB regards a configuration with 27 TFlux Kernels. 124

9 Cache setup for the memory statistics of the TFlux Evaluation Suite 128

10 Summary of the characteristics of *MMULT* 131

11 Summary of the characteristics of *TRAPEZ* 133

12 Summary of the characteristics of *SUSAN* 136

13 Summary of the characteristics of *SORT* 139

14 Summary of the characteristics of *RK* 142

15 Summary of the characteristics of *FFT* 145

16 Summary of the characteristics of *CG* 149

17 Summary of the characteristics of *LU* 157

18 Summary of the characteristics of real-life applications 159

19 Number of dynamic instructions in synthetic applications' computational load . . 161

20 Applications used for the evaluation of execution with multiple *Updaters* 178

21 Memory hierarchy configuration for TFluxSim 184

Kyriakos Stavrou

22 Memory hierarchy of Intel Xeon E5320 185

23 Summary of the machines of the experimentation infrastructure 187

24 Machines used for virtualization study 230

Kyriakos Stavrou

LIST OF FIGURES

1 Example of a Data-Driven Multithreading program . . . 28
2 The Thread Synchronization Unit (TSU) . . . 29
3 The layers of the TFlux Platform . . . 33
4 Operation of the TFlux Kernel . . . 36
5 The TFlux Scheduler . . . 37
6 Explanation of the operation of DThreads . . . 40
7 Explanation of TFlux Loops: TFlux Loop - DThread dependency . . . 43
8 Explanation of TFlux Loops: TFlux Loop - TFlux Loop . . . 44
9 Explanation of TFlux Loops: Iteration Level Dependencies . . . 45
10 Execution of TFlux Loops . . . 45
11 Execution of TFlux Loops with large number of iterations . . . 47
12 Example of TFlux Loop that performs a reduction operation . . . 50
13 Execution of Reduction TFlux Loops . . . 50
14 Thread Recycling . . . 51
15 The structure of TFlux programs . . . 60
16 Inlet/Outlet Synchronization . . . 63
17 TFlux directives: DThread declaration . . . 65
18 TFlux directives: *import*/*export* statements . . . 65
19 TFlux directives: *depends* statement . . . 65
20 TFlux directives: *kernel all* statement . . . 66
21 TFlux directives: TFlux Loop declaration . . . 66
22 TFlux directives: The *schedule 1* statement . . . 68

Kyriakos Stavrou

23 TFlux directives: TFlux Loop with unrolling 69

24 TFlux directives: TFlux Loop with reduction 69

25 TFlux directives: TFlux Loop with reduction using a custom function 70

26 TFlux directives: TFlux Loop dependencies 71

27 TFlux directives: TFlux Loops with Iteration Level Dependencies 71

28 TFlux directives: Thread Recycling 72

29 TFlux directives expressibility: Example program 74

30 TFlux directives expressibility: Code of the example program 75

31 TFlux directives expressibility: Example with iteration level dependencies 75

32 TFlux directives expressibility: Code of the iteration level dependencies example . 76

33 Abstract TFluxHard configuration with 4 cores 79

34 TFluxHard Scheduler 80

35 Example of Thread Loading 84

36 Consumer List usage example 86

37 TFluxHard Scheduler Interface for *Load TSU* 90

38 TFluxHard Scheduler Interface for *Clear TSU* 91

39 TFluxHard Scheduler Interface for *Flush Scheduler* 92

40 TFluxHard Scheduler Interface for *Execution Completion* 92

41 TFluxHard Scheduler Interface for *L-DThread Recycle* 93

42 TFluxHard Scheduler Interface for *Recycle Execution* 93

43 TFluxSoft multicore 107

44 SoftScheduler 108

45 Thread to Kernel Indexing (TKI) 119

46 TUB Segmentation 120

Kyriakos Stavrou

47 TUB Buffers . . . 123

48 Operation of *MMULT* . . . 129

49 Real-life applications: MMULT . . . 129

50 Operation of the parallel version of TRAPEZ . . . 132

51 Real-life applications: TRAPEZ . . . 132

52 Real-life applications: SUSAN . . . 134

53 Operation of the parallel version of *SORT* . . . 136

54 Real-life applications: *SORT* - Synchronization Graph . . . 137

55 Real-life applications: SORT - TFlux Code . . . 138

56 Real-life applications: RK . . . 140

57 Real-life applications: FFT . . . 144

58 Real-life applications: CG - Synchronization Graph . . . 147

59 Real-life applications: CG - TFlux Code . . . 148

60 Details of the code of *CG* . . . 150

61 Real-life applications: CG with iteration level dependencies - TFlux Code . . . 151

62 Dependencies of the critical loops of *LU* . . . 152

63 The Synchronization Graph of LU. . . . 153

64 Details of the code of *LU* . . . 155

65 Scheduling for LU . . . 155

66 Synchronization Graphs of the real-life applications . . . 158

67 Computational load for synthetic the applications . . . 160

68 The *load()* function used for synthetic applications . . . 160

69 Synthetic applications: Parallel Threads . . . 162

70 Synthetic applications: Baseline for *Parallel Threads* . . . 162

Kyriakos Stavrou

71 Synthetic applications: L1 163

72 Synthetic applications: Baseline for *L1* 163

73 Synthetic applications: L2 164

74 Synthetic applications: Baseline for *L2* 164

75 Synthetic applications: L4 165

76 Synthetic applications: Baseline for *L4* 165

77 Synthetic applications: L2R 166

78 Synthetic applications: Baseline for *L2R* 167

79 The load executed by the $ILD2_x$ synthetic application 168

80 The $ILD2_x$ synthetic application and its baseline 168

81 Synthetic applications: the code of the TFlux and baseline versions for ILD2_x . . . 169

82 The *BINARY TREE* synthetic application and its baseline 170

83 Synthetic applications: the code of the TFlux and baseline versions for ILD2_x . . . 171

84 Synthetic applications: ILD4 172

85 The baseline for the *DIAGONAL* synthetic application 172

86 Synthetic applications: L2-T1 173

87 Synthetic applications: L4-T3 174

88 Synthetic applications: ILD2 175

89 Synthetic applications: ILD4 176

90 Synthetic applications used to study the potential of using multiple *Updaters* . . . 176

91 Conceptual view of the generic simulated machine 180

92 Conceptual view of the TFluxHardSim simulator 182

93 Timing of executing Scheduler operations for TFluxHard 183

94 Conceptual view of the TFluxHardSim simulator 185

Kyriakos Stavrou

95 Quantification of the minimum DThread size for TFluxHard 193

96 Quantification of the minimum DThread size for TFluxSoft 193

97 Number of DThread for the real-life applications 194

98 DThread sizes for the real-life applications 195

99 L1-data cache miss rate for the real-life applications 196

100 TFluxHard performance for the real-life applications 196

101 TFlux Kernels usage for *MMULT* 198

102 TFlux Kernels usage for *TRAPEZ* 198

103 TFlux Kernels usage for *QSORT* 199

104 TFlux Kernels usage for *SUSAN* 200

105 TFlux Kernels usage for *RK* 202

106 TFlux Kernels usage for *FFT* 203

107 TFlux Kernels usage for *CG* 204

108 TFlux Kernels usage for *LU* 205

109 TFluxSoft performance for the real-life applications (simulation results) 207

110 TFluxSoft performance for the real-life applications (native execution results) . . . 207

111 TFluxHard performance for the synthetic applications with TFlux Loops 209

112 TFluxHard efficiency for the synthetic applications with TFlux Loops 210

113 TFluxSoft performance for L1, L2, L2R and L4 (simulation results) 210

114 TFluxSoft efficiency for the synthetic applications with TFlux Loops 211

115 TFluxSoft performance for L1, L2, L2R and L4 (native execution results) 212

116 TFlux performance for ILD2_x 213

117 TFlux performance for BINARY TREE 215

118 TFlux performance for DIAGONAL 217

Kyriakos Stavrou

119 Potential performance benefit of using 2 *Updaters* 220

120 Update-requests for execution with different number of Udpaters 221

121 Potential performance benefit of using 2 *Updaters* 222

122 Performance comparison of system with 2 versus a system with 1 *Updater* 224

123 Performance comparison of system with 4 versus a system with 1 *Updater* 225

124 TFluxHard performance for the real-life applications 226

125 The implementation of TFlux on the Cell/BE processor 232

126 Performance of TFluxCell 233

127 TFluxHard Scheduler Interface Example Program 241

Kyriakos Stavrou

Chapter 1

Introduction

Kyriakos Stavrou

It is a known fact that traditional techniques used to improve microprocessor performance have lead to diminishing returns. Such techniques include, among others, packing more transistors into the same chip [76], clocking the chips at higher frequencies [1, 82], the introduction of on-chip caches [48], exploitation of Instruction Level Parallelism [10, 110, 111, 119], speculation [66, 65, 107, 109] and prefetching [106]. The main reasons that limited the performance improvement delivered by these techniques are twofold. On the one hand the complexity of the design and the power consumed have reached such high levels, that make the extension of the above techniques unfeasible [11, 19, 80]. On the other hand, the performance increase that these techniques could deliver has, to a large extent, already been exploited [1, 11, 19].

To overcome these limitations, the CPUs have entered a new era, that of the *multicore* chips. These processors consist of a number of interconnected cores on the same die. These cores are usually simpler than the traditional monolithic designs leading to lower complexity designs and more power efficient systems. Currently, all major manufacturers have multicore products in the

market. Examples include Sun's T1 and T2 (Niagara) that support up to 64 hardware threads, IBM's Cell/BE [56] with 8+1 cores, Intel's Xeon 7400 with 6 cores, Intel's Quad Core [51] with 4 cores, and AMD's Opteron [2] with also 4 cores. Moreover, more ambitious designs like the Intel Teraflops chip [50] with 80 cores and the IBM Cyclops-64 (C64) [31] with 160 cores, are under development.

There are two major ways to explore the parallelism offered by the multicore processors: *"throughput"* and *"concurrency"*. *Throughput* refers to executing multiple *independent* applications on the system, one on each of its cores. Although this approach can keep the resources busy it is unable to improve the performance of each particular instance of an application. Moreover, it is not likely that users will have as many compute intensive tasks as the number of on-chip cores especially when this number of cores increases. On the other hand, *"Concurrency"* is the approach that has been traditionally used for the execution of compute intensive applications. In particular, the target of the *"Concurrency"* approach, is to split a *single* program into parallel entities, which are to be executed by different computation nodes. The main objective is to decrease the execution time of the application. If enough parallelism from the application is exposed to the hardware, it is possible to utilize the available resources, therefore decreasing the application's execution time as much as possible. While throughput is easily exploited in current small-scale multicore processors, the support for efficient *concurrency* will be the key issue to exploit the parallelism for the future large-scale multicore processors.

In addition to the parallel hardware, the key components for exploiting concurrency are the *execution model*, which allows the application to utilize multiple processors for its execution and the *programming model* in order for the programmers to develop such applications. An execution model appropriate for multicore processors must be a good match to the particular characteristics of the architecture. It must provide an efficient scheduling of the concurrent tasks, and at the same

Kyriakos Stavrou

time hide the details of the underlying system providing to the user and programmer a virtualized environment. This *virtualization* allows for programs to be developed without a particular system as target.

The programmability of a parallel system is another major issue. Although numerous research projects have focused in the past on building fully automated parallelizing compilers, such as Polaris [18], SUIF [128], Parafrase [41] and NCI [77], none of them is widely used today. In contrast, programmers use semi-automated solutions like the compiler directives of OpenMP [81] or the libraries for MPI [74] or pthreads [21]. This semi-automated approach is what seems to be the programming solution to be followed for the multicores architecture [11, 19]. To achieve high performance the programming model should enable the programmer to express a high degree of parallelism at a fine-grained level. In addition, this model should not require the programmer to develop applications with a particular machine configuration in mind.

According to the previous discussion, the main characteristics that a platform must have in order to allow for the applications to efficiently exploit the parallelism offered by a large-scale multicore are the following:

Virtualization: The platform must offer to its user and programmer a virtualized environment that hides the details of the underlying system both at the hardware and software level. Consequently, programs do not need to be developed for a specific machine configuration. Instead, programmers must be able to develop and execute on all implementations of the proposed platform.

Performance: It is required for the proposed platform to be able to achieve high levels of performance even for non explicitly parallel applications. In order to achieve this, the parallelization overheads incurred must be kept at minimum levels.

Kyriakos Stavrou

Scalability: The performance of the proposed platform must scale to multicore systems with larger number of cores. In order to achieve this, the parallelism exposed by the model must not be limited in any way by the number of execution units.

Portability: The proposed platform must require minimum effort for it to be ported to a new system. The system-specific code must be limited to only a small number of components of the proposed platform.

Programmability: Programs for such a platform are to be developed using a fully- or semi-automated parallelization approach, *i.e.* with the use of a parallelizing compiler or by augmenting sequential code with special compiler directives. The programming model must allow the programmer to express any parallel construct in an intuitive way.

1.1 The TFlux Parallel Processing Platform

This work presents the ***Thread Flux (TFlux)*** Parallel Processing Platform, a parallel execution system that aims to deliver high performance on *commodity* multicore systems. ***TFlux*** is a complete system, from the hardware to the programming toolchain and includes a number of key components that allow it to meet the targets set for a successful parallel processing platform.

TFlux achieves high performance and scalability mainly due to its scheduling policy that follows the Data-Driven Multithreading (DDM) model of execution. This dataflow-like scheduling policy enables ***TFlux*** to exploit more parallelism compared to other models that enforce synchronization using barriers and locks. The careful design of the several components of ***TFlux*** reduce the parallelization overheads in a way that allows for applications with fine-grained parallel segments to scale well on systems with a large number of processing elements.

Kyriakos Stavrou

It is a main goal of ***TFlux*** to provide dataflow-like scheduling to *commodity* systems. By using *commodity components* ***TFlux*** not only widens its impact but also is able to benefit from the improvements introduced in those components. In order to achieve its goal, ***TFlux*** follows a layered design. The top layer, which is the one programmers use to develop TFlux applications, abstracts all details of the underlying machine. TFlux applications are developed using ANSI-C together with TFlux compiler directives. The TFlux compiler directives are used to express the code of the parallel segments and the dependencies among those segments. The code of the TFlux program passes through the TFlux Preprocessor that outputs a C program, which may be compiled by a *commodity* C compiler resulting in an executable binary. The binary invokes the operations of the TFlux Runtime Support allowing execution under TFlux. The Runtime Support runs on top of an *unmodified* Unix-based Operating System and hides all TFlux specific details such as the particular implementation of the TFlux Thread Scheduler. More details about the purpose, design and operation of the different layers of TFlux will be given in Chapter 3 that focuses on the presentation of the TFlux Platform. As for the TFlux directives and the TFlux Preprocessor they are presented in Chapter 4.

This work presents two incarnations of TFlux: ***TFluxHard*** and ***TFluxSoft***. The key component that differentiates these two incarnations is the TFlux Thread Scheduler. In particular, whereas for ***TFluxHard*** the functionality of the Scheduler is provided by a hardware unit, for ***TFluxSoft*** this functionality is provided at the software level. As such, although ***TFluxHard*** uses only commodity components, it requires the system to be augmented with extra hardware. ***TFluxSoft*** on the other hand is directly applicable to *off-the-shelf* multiprocessor systems. The details of these two systems are presented in Chapter 5 for ***TFluxHard*** and in Chapter 6 for ***TFluxSoft***.

The key component of TFlux is its ability to provide the dataflow model of execution to its programmer. As such, evaluation of the platform needs to be made with a representative collection

Kyriakos Stavrou

of benchmarks with different dependency graphs. For this TFlux Evaluation suite we selected a number of real-life and synthetic applications which are presented in detail in Chapter 7.

The evaluation of TFlux was made on a number of different systems which included both simulated and off-the-shelf machines. In Chapter 8 we present the details of these configurations and explain the simulation process. Moreover, we analyze the metrics we used and explain the methodology for the experiments.

The experimental results are presented in Chapter 9. These results show that TFlux incarnations are able to deliver almost linear speedup and good scalability. A comparison of the two TFlux systems shows that offloading the Scheduler's functionality to the hardware unit allows ***TFluxHard*** to deliver better performance compared to ***TFluxSoft*** especially for applications with very fine-grained parallel constructs. Both systems however, were found to outperform "traditional" parallel execution models.

The last part of this work, which is presented in Chapter 10, explores the ability of TFlux to virtualize the details of the underlying system. This qualitative study was performed by executing both TFlux incarnations on a number of different systems. Towards this direction, to study the *portability* of ***TFlux***, *i.e.* the effort that is required to port the system to a machine with key differences, we detail the implementation of TFlux on the Cell/BE processor.

The main outcome of this work is a system that brings *dataflow* scheduling to *commodity multicore systems*. ***TFlux*** achieves this goal through its layered design which enables to the Data-Driven Multithreading (DDM) model of execution to operate on unmodified components.

Kyriakos Stavrou

1.2 Contributions

The main contribution of this work is the design, the implementation and the evaluation of the ***TFlux*** *Parallel Processing Platform*. In the list that follows we present each contributing factor in more detail.

1. **TFlux Platform**

   (a) **Abstract design of the TFlux Platform:** The design includes strict description and specification for all entities required in order for multiprocessor systems to execute ***TFlux*** applications. These entities include the TFlux compiler directives, the Runtime Support, the TFlux Kernels and the TFlux Scheduler.

   (b) **Virtualization and Portability:** The different layers of TFlux have been designed in such a way that allows them to hide the details of the underlying machine. To achieve this goal it was also necessary to define efficient interfaces between the different layers.

2. **TFlux Preprocessor**

   (a) **Definition of the TFlux directives:** This work defined a set of *general* compiler directives appropriate for expressing dataflow dependencies between the parallel segments. These directives are not specific for the TFlux Platform but have also been used for other modern architectures capable of supporting dataflow scheduling.

   (b) **TFlux Preprocessor:** Another part of this work was the development of a tool that converts TFlux applications, *i.e.* applications augmented with the aforementioned compiler directives, to ANSI C programs able to compile with commodity C compilers.

Kyriakos Stavrou

3. **TFlux Incarnations**

   (a) **TFluxHard:** Another contribution of this work is the identification of the Basic Operations of the TFlux Scheduler and their description as hardware components. Moreover, this task includes the efficient design and implementation of the interface between this Scheduler and the Runtime System as well as the interconnection of the Scheduler to the system's network.

   (b) **TFluxSoft:** A second incarnations of ***TFlux***, the ***TFluxSoft*** system is also a contribution of this work. The main difference of ***TFluxSoft*** compared with ***TFluxHard*** is that the former is applicable to off-the-shelf machines. For this incarnations, this work presents the design and implementation of the TFlux Scheduler at the Software level.

4. **TFlux Evaluation Suite:** As part of this work we needed to form a collection of real-life and synthetic applications in order to evaluate the two incarnations. Notice, that this suite is general enough to be applied to other dataflow architectures.

5. **Publicly available distribution of TFlux:** TFluxSoft is publicly available upon request and may be found at www.cs.ucy.ac.cy/carch/casper/tflux. With this contribution, anyone interested in executing applications using the DDM model of execution may do so using TFluxSoft on any off-the-shelf system.

Kyriakos Stavrou

Chapter 2

Related Work

This Chapter presents a representative subset of the related work relevant to the concepts of the TFlux Platform. Section 2.1 discusses the factors that forced the shift from the monolithic single-chip microprocessor design to the multicores whereas Section 2.2 presents models of execution that target these systems. In Section 2.3 we introduce the dataflow model of computation and in Section 2.4 the Data-Driven Multithreading (DDM) model of execution. Finally, in Section 2.5 we present the Data-Driven Network of Workstations (D^2NOW) and in Section 2.6 the Data-Driven Multithreading Chip Multiprocessor (DDM-CMP).

2.1 The Shift to Multicore Systems

The performance increase observed for single-chip microprocessors was until recently a result of combining three different factors. The first was related to the implementation technology. In particular, as the technology feature size decreases it is possible to pack not only more, but also faster transistors, into the same chip [1, 11, 19, 82, 131]. This technology shrink, in addition

Kyriakos Stavrou

to the performance increase due to higher clock rates, also allows the implementation of more sophisticated microarchitectural mechanisms.

The second factor is related to microarchitectural techniques. Maybe the most important such technique is pipelining which leads to a significant increase of the throughput of microprocessors. Today all known microprocessors include deep pipelines. Other techniques that are known to improve the instruction-level parallelism are the register renaming [108], out-of-order execution [119] and branch prediction [75].

Whereas these techniques improved significantly the performance of microprocessors this was not the case for the memory system. In reality, most advances have lead to the increase of what is known as the *"CPU-Memory gap"* [82], *i.e.* the gap between the time to execute an instruction and the time to get a value from memory. Therefore, the third factor towards increasing the performance of single-chip microprocessors regards techniques that target this *"Memory Wall"*. These techniques include the introduction of on-chip caches [48], hardware and software prefetching schemes [106] as well as microarchitectural structures such as the load-store queue [108] or the trace cache [55].

An orthogonal technique to the ones mentioned above is *Simultaneous Multithreading* [122]. This technique, which was implemented in commercial chips such as the Intel Pentium 4, allows for instructions from different programs to co-exist in the same pipeline. To achieve this the processor is equipped with additional registers and wider pipelines. Although this technique is able to increase the throughput of the processor often comes with a negative impact on power consumption.

In the last years, the use of the above techniques to exploit better performance started to result in diminishing returns [1, 11, 19, 80], *i.e.* little improvement was obtained for the additional complexity in the design. This is mainly due to the fact that the potential of most of these approaches

Kyriakos Stavrou

has, to a large degree, been already exploited [1, 11, 19]. Moreover, no new, radical approach has been included in the commercial chips of last years. Although making the existing mechanisms more aggressive (e.g. larger caches, wider pipelines, multilevel speculation) seemed to be a possibility, the *"Power Wall"* became a serious limitation. In particular, the complexity of these designs increased the power consumption to such level that managing the dissipated temperature was only possible up to a point. Vendors tried to mitigate this problem through various techniques both at the physical and microarchitectural level [67, 69, 70, 103, 115]. Despite the contribution of these techniques to decreasing both power and temperature, the *Power Wall* is now the major limiting factor to the microprocessor design. Today, computer architects design chips for maximum performance for a given power budget [103].

Major processor vendors realized that the traditional approaches were not the way that would allow them to meet the performance increase rate projected by *Moore's Law*. As such, instead of trying to increase the performance of each single CPU core, the effort focused on packing multiple cores onto the same chip leading to the Chip Multiprocessor (CMP) paradigm [80]. The trend is for the number of these cores to increase and for their complexity to decrease in each generation [11, 19, 131]. The major challenge today is to find programming models that will be able to efficiently keep all the available resources busy.

2.2 Parallel Programming Models

This Section presents recent parallel programming models specifically designed for multicore systems. Due to the large number of research efforts that target parallel processing it is not possible for this list to be exhaustive. However, we believe that the works presented in this Section form a representative subset.

Kyriakos Stavrou

Thread Level Speculation:

Thread Level Speculation (TLS) [79], implemented by the Stanford's *Hydra* CMP, allows the programmer to break a sequential program into non-overlapping sections of code called *Threads* without the need to statically identify data-dependencies among them. The hardware attempts to execute the Threads in parallel while tracking all memory accesses to detect dependencies. As such, parallelization of difficult programs is made easier. In case of miss-speculation Thread rollback/restart functions are activated. The Hydra CMP requires extra hardware which is reported to consume the real estate of a pair of L1 caches per CPU.

Compared to Hydra, TFluxSoft achieves similar speedup without the need for extra hardware; and as for the TFluxHard solution, this hardware is smaller and simpler. In addition, in the Hydra solution TLS targets only loop bodies and subroutine calls whereas in TFlux parallelism can be of much finer grain. Currently in TFlux dependencies are statically defined but this issue is expected to be covered in future versions of our compilation tool chain. Moreover, in contrast to Hydra, the extra hardware required for TFluxHard does not affect the design of the CPU cores making a potential implmentation much simpler.

Multiscalar Processors:

The idea behind the *Multiscalar* processors [111] is to split a program into fine-grained tasks and execute these tasks in parallel. What makes Multiscalar Processors differ compared to other parallelization approaches is the fact that it is not required to define the dependencies between these tasks statically. The hardware enforces the necessary Control and Data dependencies dynamically. To achieve this, the Multiscalar processors execute the tasks speculatively and in the case of miss-speculation invoke a recovery mechanism. Speculation does not regard only control flow dependencies but also data-dependencies. As such, the ultimate goal of Multiscalar processors is to achieve a partial ordering for the execution by applying only the true data dependencies.

Kyriakos Stavrou

According to the experimental results, the Multiscalar processors are able to achieve large speedup values. However, to deliver this speedup the Multiscalar processors need to modify the CPU cores, the memory hierarchy, the compiler and extend the ISA.

Compared to Multiscalar processors the TFlux architecture does not support neither data nor control-flow speculation which is likely to limit the amount of parallelism it is able to exploit. Moreover, for TFlux the data dependencies need to be identified by the user statically. However, as opposed to Multiscalar Processors, TFlux achieves all its benefits without requiring any modification to any component (CPU cores, OS, compiler, ISA). As such, TFlux is able to deliver its benefits on off-the-shelf systems and, as such exploit any benefits delivered by future generations of these components whereas this will not always be true for the Multiscalar architecture.

Carbon:

In the Carbon [60] project, the authors propose augmenting the CMP with additional hardware queues to exploit data and loop level parallelism. The results are promising, as in the TFlux system, but the solution is not applicable to existing multiprocessors.

In contrast, TFlux can achieve high performance without any modifications to the hardware or to the OS. In addition, Carbon requires extensions to the ISA and as such, modifications to the CPU cores and the compilation toolchain whereas TFlux requires neither of these.

Synchronization State Buffer (SSB):

The target of the *Synchronization State Buffer* (SSB) [131] approach is to provide the means for fine-grain synchronization on many-cores systems. The rationale of SSB is based on the observation that at any instance during the execution of an application, only a small fraction of memory locations are actively participating in synchronization. Based on this observation the SSB design is able to provide the illusion that the entire memory is tagged at word-level, and therefore can be considered as a "virtually tagged memory design".

Kyriakos Stavrou

For efficient fine-grain parallelism, it is necessary to have all the memory locations tagged, which imposes very large hardware requirements. Given that future multicores will have significantly smaller *per core* caches, such an approach would further increase the problem of reduced memory. For SSB to be implemented there are certain hardware requirements. In particular, SSB adds a small hardware device which is attached to the memory controller of each memory bank. In addition, SSB assumes a non-preemptive execution model. According to [131], future chips are likely to have this characteristic.

Although the architecture is promising it can not be applied to current multicores. This is not only due to its requirement for hardware extensions in the CPU but also due to requirement for non-preemptive execution model and lack of programming tool-chain. TFlux, on the other hand, is able to operate using commodity components whereas at the same time it has a dedicated programming toolchain.

MapReduce:

MapReduce is a programming model and an associated implementation proposed by Google for processing and generating large data sets that targets large clusters of commodity machines [30]. This model consists of two simple operations, *map* that processes a key/value pair to generate a set of intermediate key/value pairs and a *reduce* function that merges all intermediate values associated with the intermediate key. The main advantage of this model is its simplicity and the fact that it can be applied to commodity computers. In addition to high performance, this model also provides support for dynamic load balancing, fault tolerance, locality optimizations and also backup tasks (*stragglers*). According to [87], which studies the applicability of this programming model on multicore processors under an architecture named *"Phoenix"*, although promising, MapReduce is not general enough to cover all application domains. This is due to the fact that the MapReduce model targets mainly *data parallel* applications.

Kyriakos Stavrou

Comparing MapReduce to TFlux's programming model, *i.e.* Data-Driven Multithreading, it is possible to conclude that TFlux is able to exploit more parallelism as it is not limited to data parallel applications. The runtime system of the two implementations of the MapReduce programming model (the one used by Google for clusters and the *Phoenix* system) however, are more mature that the runtime system of TFlux as they provide functionality which is still under development for TFlux such as support for load balancing and fault tolerance.

Compute Unified Device Architecture (CUDA):

The Compute Unified Device Architecture (CUDA) [78] is a programming environment proposed by Nvidia for developing high-performance parallel programs to be executed on Graphics Processors. Although using Graphics processors for general purpose computing (GPGPU) [38] is a very promising approach, application development is not trivial. In particular, the programmer needs to manage the data explicitly in order to exploit data locality and consequently high performance. Moreover, similar to MapReduce, CUDA is limited to data parallel applications. As such, CUDA can lead to extremely high performance only for applications that fit to its execution style. TFlux, in contrast, has wider applicability and targets conventional processing elements.

Micro-threading:

Micro-threading [71] is not a processor architecture but rather a model of concurrency. The model is fine-grain and provides synchronization in a distributed register file, which allows it to efficiently scale to designs with a large number of cores. Although this model provides backwards compatibility, it requires ISA extensions to describe the concurrency in the program.

In a *micro-threaded microprocessor* the threads are obtained from a single context exploiting both vector and instruction level parallelism (ILP). This approach employs vertical and horizontal transfer in a simple pipeline. The horizontal transfer is referred to as the normal scalar pipeline processing used in most microprocessors. Vertical transfer is a context switch, which allows the

Kyriakos Stavrou

code to tolerate any latency from undetermined data and control dependencies. A micro-thread can be a small sequence of code, which may be a basic block, a loop iteration or even just a few instructions.

Being more fine-grained than TFlux allows Micro-threading to exploit larger amount of parallelism. However, Micro-threading requires complete redesign of the microprocessor and also ISA extensions. In contrast, TFlux is able to operate with commodity components.

Cell Superscalar:

Cell Superscalar (CellSs) [15] is a programming model for the Cell/BE architecture. CellSs has two major components: a source-to-source compiler and a runtime system. Similar to TFlux, CellSs requires the user to annotate the code with directives describing the boundaries and dependencies of parallel sections. Using this information, the source-to-source compiler of CellSs builds the application's dependency graph and also partitions the execution to the SPE and PPE processors. Using two lists, a ready task list and a list of the available resources, the runtime schedules tasks for execution based on the dataflow firing rule. Moreover, the runtime takes into consideration data locality, applies techniques to minimize the communication needed between the SPEs and main memory and to decrease load imbalance.

Although both TFlux and CellSs follow the same task scheduling policy there are some important differences between the two parallel models. On the one hand CellSs has a number mechanisms that do not exist or are not relevant for TFlux. Such techniques include the load balancing mechanism, which could apply for TFlux as well, and the handling of heterogeneity between the SPEs and PPEs which is not relevant for TFlux. On the other hand, the tasks for CellSs are more coarse-grained compared to TFlux as they need to be at the level of function calls, which decreases the amount of exploitable parallelism. In contrast, for TFlux the tasks are not limited to a particular programming construct. Moreover, CellSs has only been tested and evaluated on the

Kyriakos Stavrou

Cell/BE processor whereas TFlux has been proven able to execute on a variety of multiprocessor architectures.

2.3 The Dataflow Model of Computation

Conventional computer architectures are based on the control-flow model of execution. In this model, instructions are scheduled statically by the programmer or compiler. A program counter is used to issue the next instruction. The control-flow model exploits program locality via a memory hierarchy. Multiprocessor systems based on control-flow architectures suffer from memory latency and synchronization overheads [7]. Furthermore, the control-flow model inhibits parallelism by imposing artificial dependencies. If an instruction stalls, then the entire program must wait for the stalled instruction to resume execution.

The dataflow model of computation was proposed in the early 1970s by Jack Dennis [32] as an alternative to the control-flow model of execution. In the dataflow model, instructions are scheduled dynamically at runtime based on data availability. An instruction becomes executable only when all of its input operands are available to it. A dataflow program is represented by a graph consisting of nodes and arcs. The nodes represent the instructions of the program, while the arcs represent the data dependencies among instructions. Data propagates along the arcs in the graph in data packets called the tokens.

Dataflow is known to expose the maximum amount of parallelism to the hardware [7]. This is due to the fact that dataflow graphs, as opposed to conventional machine languages, specify a *minimum* order for the execution of instructions and thus provide the opportunity to the hardware to exploit *all* the parallelism available in an application. This key characteristic, the ability to expose to the hardware *all* the available parallelism, is of major importance for any multiprocessor

Kyriakos Stavrou

system as it allows for applications which have limited amount of parallelism to benefit from the multiple execution nodes.

Dataflow architectures can be classified as *Static* or *Dynamic* [88]. In a *Static* architecture the nodes of a program graph are loaded into memory before the computation begins and at most one instance of a node is enabled for firing at a time. A *Dynamic* architecture facilitates the firing of several instances of a node at a time and these nodes can be created at runtime. For example, a loop body in a program can be represented as a single node. A *Dynamic* architecture unfolds the loop at runtime by creating multiple instances of the node representing the loop body and attempts to execute the instances concurrently. For the same loop, a *Static* architecture would require a different node for each iteration of the loop. *Static* architectures are known to be less complex in comparison to the *Dynamic* ones but less efficient for a wide range of applications.

2.3.1 Dataflow Architectures

2.3.1.1 Static Dataflow Architectures

The first dataflow machines were static dataflow architectures. These machines were known as the *single-token-per-arc* dataflow machines as only a single token could reside on an arc. A graph, in a static dataflow machine, is represented by *activity templates* containing the opcode of the instruction, the operand slots and the destination address fields.

Static dataflow machines have difficulties when a graph is reentrant, such as a loop body or a function invocation. To avoid these problems, an instruction is enabled for execution if tokens are present on its input arcs and there are no tokens on any of its output arcs. This *single-token-per-arc* constrain is achieved by using acknowledgment signals in the *activity templates*. These signals are shown on the dataflow graph with additional arcs. When an instruction is fired for execution, it sends a token on the acknowledgment arc indicating that it is ready to accept a new token.

Kyriakos Stavrou

These acknowledgment signals propagate from the consuming nodes to the producing nodes on the dataflow graph.

One of the early static dataflow machines was the *MIT Static Dataflow Architecture* [53]. This machine was composed of memory units and processing elements. Each memory unit stored the operation, operands and destinations addresses of a node of the program's graph which was loaded to the memory cells by the host. Two different networks were responsible for the necessary communication between the processing elements and the memory units. The main drawback of this architecture was that consecutive loop iterations could only partially overlap in time, limiting parallelism. Additions to the static model, such as queues, have addressed this limitation in subsequent designs by Dennis and others.

Another static dataflow architecture is the *Language Assignation Unique (LAU)* [85] which targeted programs written in a special Single Static Assignment language (Single Static Assignment Languages are functional languages that prevent side effects and often favor the dataflow model of execution). The LAU machine was built out of four major units: memory, execution, control and interface units. A prototype with 32 processors was built and execution of some aerospace programs showed that a linear speedup could be achieved.

Texas Instruments' *Data-Driven Processor* (DDP) [25] was designed for the execution of Fortran programs using some of the principles of the aforementioned MIT's static architecture. In this architecture a special algorithm was used to partition the program into subgraphs which were dynamically loaded to the memory. Another novelty of this design, was the use of *counters* to determine when a node of the program was ready to execute. A machine with four such processors was built in the early 80's.

Kyriakos Stavrou

2.3.1.2 Dynamic Dataflow Architectures

Dynamic dataflow architectures allow loop iterations and subprogram invocations to proceed in parallel. This is achieved by allowing multiple tokens on each arc. A token includes a tag that carries information about the context of the token. Dynamic dataflow architectures are also called *tagged-token* dataflow architectures. A node is enabled if tokens with identical tags are present at each of its input arcs. To distinguish between different instances of an instruction, the activity template is extended with two extra fields: the first carries the loop iteration index, and the second its function context.

Two of the early tagged-token dataflow architectures are the *Manchester Dataflow Computer* [43] and the *MIT Tagged-Token Dataflow Machine* [9, 8]. The main drawback of tagged-token dataflow machines is the need of a large associative memory for the implementation of the token matching unit.

The concept of *Explicitly Token Store* (ETS) has been proposed to eliminate the need for associative memory. A separate memory frame is allocated for each active loop iteration or subprogram invocation. The explicitly address token store concept was initially developed in the Monsoon [39, 40] project. A memory location, within the activation frame of each function allocation is established where each synchronization takes place. In general, the allocation of activation frame has to be done dynamically at runtime.

2.3.2 Dataflow Limitations

The benefits of the dataflow model are the tolerance to long memory latency events and overcoming the synchronization overheads, as defended by Arvind and Iannoucci [7]. Although the authors concluded that the dataflow microprocessors could overcome these limitations, reality proved less favorable for the dataflow model of execution. The reasons for this are very well

Kyriakos Stavrou

presented in a work published by Culler *et al.* in 1993 [29]. In particular, the authors observe that Arvind's and Iannucci's arguments neglected significant limiting factors related to the storage hierarchy.

Arvind and Iannucci based their reasoning on the latency tolerance ability of dataflow multiprocessors. Specifically, they argued that if the physical processor includes a sufficient number of *Virtual Processors*, each executing a different task, synchronization and communication latencies can be tolerated by fast switching between tasks. More specifically, when a task blocks due to a remote reference, the physical processor switches to another enabled task (*Scheduling Event*) thus hiding the penalty this event would cause. When the remote reference completes its execution, the task that issued it, is deemed enabled, *i.e.* ready for execution (*Synchronization Event*). According to this reasoning, the efficiency of dataflow multiprocessors depends on their ability to effectively handle *Scheduling* and *Synchronization* events.

However, both events impose certain requirements on the hardware. Specifically, fast synchronization, can only be achieved with associative memories, which size and cost increases with the number of concurrently executed tasks. On the one hand, the dataflow architecture requires a large number of such tasks to tolerate latencies. On the other hand, the larger the number of concurrent tasks, the longer the time required to perform a synchronization event. Due to the inability to build the required hardware structures, this issue was one of the major factors that limited the success of early dataflow systems.

Scheduling events require switching the state of the processor. This switching, can be efficient only if the data to be stored and loaded reside in the higher levels of the memory hierarchy. However, fast memories are small, which contradicts with the requirement of large number of concurrently enabled tasks. The limitations imposed by the efficiency of the memory hierarchy, was another reason that limited the success of the dataflow systems.

Kyriakos Stavrou

To decrease the hardware complexity, some functionality was offloaded to the compilers of special dataflow languages. Examples of those languages include the *ID* [6], *LUCID* [12], *LAU* [24], *Sisal* [42], *Cajole* [45], *VAL* [73], and *Tdfl* [125]. Although excelling in expressing parallelism in dataflow machines, these languages were unable to support side effects, mutable data structures and other programming constructs that were present in widely used programming languages [37, 54, 57, 117, 127].

Another limiting factor for dataflow was its inability to efficiently manipulate complex data structures. Although it was easy to pass simple arithmetic values from one instructions to another, this issue was getting more complicated when structured data was to be passed [57]. Moreover, dataflow machines did not handle arrays of data very efficiently due to their emphasis on fine-grain, operation-level concurrency [37].

2.3.3 Hybrid Dataflow

Pure dataflow does not perform very well with sequential code because of: (a) the inefficient use of the pipeline since an instruction of the same thread can be issued only after its predecessor instruction is completed, (b) no usage of registers since a context switch occurs at fine-grain (after the execution of each instruction) and (c) the excessive overheads due to the per-instruction token matching. Combining dataflow with control-flow mechanisms could eliminate the problems of pure dataflow. Such hybrid architectures are classified as *macro-actor Threaded dataflow*, or Large-grain dataflow [94]. In such architectures a node in a dataflow graph is a sequential instruction stream referred as a *thread* of instructions. Since a thread consists of several instructions, data can be stored in registers, the token matching overhead is reduced and pipeline bubbles can be avoided.

Kyriakos Stavrou

Large grain dataflow architectures decouple the token matching stage from the execution stage. The two stages communicate with each other via FIFO buffers. One of the earliest architectures that utilize the decoupling principle is the USC Decoupled Graph/Computation architecture [34]. A node in this architecture consists of two basic units (the graph unit and the execution unit) that operate in an asynchronous manner. The graph unit is responsible for updating the dataflow graph and determining whether a graph node is executable, while the computation unit is responsible for the execution of the instructions of the graph's nodes.

2.3.4 Non-blocking Multithreading

Architectures that have evolved from the large-grain dataflow architectures are called *non-blocking multithreaded* architectures, since a thread is fired only if its data dependencies are resolved. This ensures that a fired thread will execute to completion without encountering long latency events due to remote memory, communication or synchronization operations. Non-blocking multithreaded architectures have the advantage that they can be built using conventional off-the-shelf microprocessors. The representative examples of non-blocking multithreaded architectures are the StarT [3], the EARTH [47] and the TAM [28].

StarT (or *T) [3] is the successor of the Monsoon [39] project. This is a multithreaded architecture designed to support non-blocking threads. Each *T node consists of three processors: the data, the memory-memory request and the synchronization processors, all sharing a local memory. The data processor is the processor that executes threads. The memory-memory request processor is responsible for incoming remote load/store requests. The synchronization processor is responsible for handling returning load responses and join operations.

Another non-blocking multithreading project that has its origins in the dynamic dataflow sequencing is the EARTH *(Efficient Architecture of Running Threads)* [47] multiprocessor. The

Kyriakos Stavrou

EARTH architecture was implemented on top of the MANNA multiprocessor. Each EARTH node consists of two processing units: the execution unit and the synchronization unit, linked together by queues and sharing the same local memory and network processor. The execution unit executes threads and passes information to the synchronization unit by placing messages in an event queue. The synchronization unit fetches these messages from the event queue, determines which threads are ready for execution, and places their identification numbers in a ready queue. Ready threads are executed by the execution unit to completion, since they are non-blocking threads.

The Threaded Abstract Machine (TAM) [28] is a fine-grain execution model where thread synchronization, scheduling and storage management are placed under the compiler control. Synchronization is explicit and occurs only at the top of a thread. A synchronizing thread is associated with a frame slot that contains its *entry count*. The *entry count* is initialized by the compiler according to the input arcs of the thread in the graph. Whenever a thread is forked, its *entry count* is decremented. A thread is enabled for execution when its *entry count* reaches zero. Finally, notice that TAM was implemented for a variety of existing processors and platforms.

2.3.5 Recent Dataflow Developments

Dataflow principles are currently used in commercial high performance processors. Out-of-order processors employ dataflow principles to rearrange the execution order of instructions in order to achieve better utilization of the processor's resources. This reordering is achieved with the support of hardware units that determine data dependencies among the instructions stored in the instruction window dynamically at runtime.

Recently, several research projects like SDF [5], EDGE [20, 90], DDM [61] and DDM-CMP [112], have adopted the dataflow principles. Their success is mainly due to two reasons, the usage of today's mature hardware technology and the fact that the majority of them apply

Kyriakos Stavrou

the dataflow model to a coarser-grained level, that of the thread, instead of the fine-grain instruction. These issues minimize the hardware requirements and make their efficient implementation feasible.

Explicit Data Graph Execution (EDGE) [20] proposes a new instruction set architecture (ISA) that enables scaling to window sizes of thousands of instructions to hold large code blocks. The EDGE ISA provides a richer interface between the compiler and the micro-architecture by directly expressing the dataflow graph generated by the compiler to the micro-architecture of the processor. This removes from the hardware the task of rediscovering data dependencies dynamically at runtime. EDGE uses direct instruction communication, *i.e.* a producer instruction delivers its produced data directly to its consumer instructions. This enables instructions to execute in dataflow order, with instructions firing as soon as all of their operands are available. A major difference between EDGE and other RISC or CISC architectures is that in EDGE the compiler does not encode the source operands of instructions. EDGE instructions specify only their targets or consumers. The TRIPS processor [90] is an instance of the EDGE architecture which uses large cores consisting of a matrix of execution units (ALUs with input operands, buffers and output routers). Code blocks are mapped by the compiler to an array of execution units and are scheduled dynamically based on branch predictors. Instructions within code blocks are executed based on the dataflow order set by the compiler.

Another threaded dataflow project is the Scheduled Dataflow (SDF) [58] architecture. SDF is a multithreaded architecture that decouples the synchronization from the computation of non-blocking threads. SDF has its origins in the PL/PS-Machine (Pre-Load/Post-Store), a multithreading machine that decouples memory accesses from thread execution. An SDF processor consists of two pipelines: the execution pipeline and the synchronization pipeline. The synchronization pipeline is responsible for scheduling the non-blocking threads whereas the execution pipeline for

Kyriakos Stavrou

the execution of threads. Each thread is assigned a register context. The synchronization pipeline preloads the data needed by each thread in its register context in the execution pipeline before firing the thread for execution. The results from an execution are stored by the execution pipeline in registers and then post-stored in the memory by the synchronization pipeline. This decoupling eliminates stalls due to memory accesses and cache misses.

Weng and Chapman [126] present the benefits of translating OpenMP code to low-level parallel code using a dataflow execution model. Using dataflow principles, they manage to remove unnecessary dependencies between potentially parallel sections, improve data locality and reduce the synchronization overheads.

Kim and Sair [59], use a dataflow architecture for the design of an efficient decoder for the H.264 video standard. The authors point out that System-on-Chip (SoC) designs are known to be inadequate for data-intensive tasks such as video encoding/decoding. Their design was based on co-locating computation and data in the physical space, achieving a significant decrease in the communication latencies. Finally, based on their experimental results, they conclude that dataflow architectures are the most appropriate for data-dominated applications.

In the work by Swanson *et al.* [118], the authors investigate the area/performance tradeoffs of a tiled dataflow architecture, that of the WaveScalar [117] processor. Specifically, this work compares the WaveScalar's area efficiency to that of an aggressive out-of-order superscalar and to that of a Sun's Niagara chip multiprocessor. The main conclusion of this work is that, the dataflow nature of WaveScalar provides substantially more performance per unit area and better area scaling compared to the other two systems.

Finally, Balakrishnan and Sohi [14], proposed a novel technique for significant performance improvements based on dataflow principles. This execution paradigm *demultiplexes* the execution of methods (functions or subroutines) from the rest of the program. Based on dataflow analysis, a

Kyriakos Stavrou

method for converting the *total* ordering of a sequential program's functions to *partial* ordering is presented. This allows them to speculatively execute functions prior to the point they would have been invoked in the sequential execution of the program. When the program reaches the call point of those functions, the correctness of their speculative execution is checked and their results are either committed or discarded. Although the results presented are limited, the authors managed to prove the potential of their approach.

2.4 Data-Driven Multithreading (DDM)

The *Data-Driven Multithreading* (DDM) model of execution [61, 63] has been proposed as a solution to overcome the limitations of dataflow whereas at the same time keeping its ability to exploit very high degrees of parallelism. The characteristic of DDM that allows it operate with significantly smaller hardware requirements compared to dataflow is the fact that is uses coarser-grained scheduling units following the Hybrid Dataflow approach. In particular, whereas dataflow applies the dataflow firing rule at the level of *individual instructions*, DDM applies this rule at the level of *sequences of instructions*. As for the other limitations of dataflow, DDM can be programmed with imperative languages, such as C. This allows it to provide the common load/store memory semantics explicitly and also to efficiently handle any type of data structure in a generic way.

DDM programs are composed of non-overlapping sections of code called *Data-Driven Threads* (DThreads) that may contain an arbitrary number of static or dynamic instructions. A producer/consumer relationship exists between DThreads. The dependencies among the DThreads in a DDM program are expressed by the input and output data of each DThread and are represented by a *Synchronization Graph*. The nodes of this graph correspond to the program's DThreads while its arcs to data dependencies between the different DThreads.

Kyriakos Stavrou

As an example, refer to Figure 1-(a) that depicts the high-level code of a synthetic application and its partitioning into DThreads (the particular partitioning is for illustrative purposes only). From this code it is possible to observe a number of dependencies. In particular, DThread 1 writes the variable z which is then used by DThread 3. As such, DThread 1 is a producer for DThread 3 (equivalently, DThread 3 is a Consumer of DThread 1). Similarly, it is possible to observe that there is a dependency between DThread 2 and DThread 3 through variable t, between DThread 2 and DThread 4, also through t, between DThread 3 and DThread 5 through f and finally between DThread 4 and DThread 5 through g. These dependencies form the Synchronization Graph of this application which is presented by Figure 1-(b). The number of producers for each DThread is named the "Ready Count" value and is depicted in this Figure as a shaded value next to each node. The Ready Count value is initiated statically and is dynamically decreased each time a producer completes. A DThread is deemed executable when its Ready Count value reaches zero.

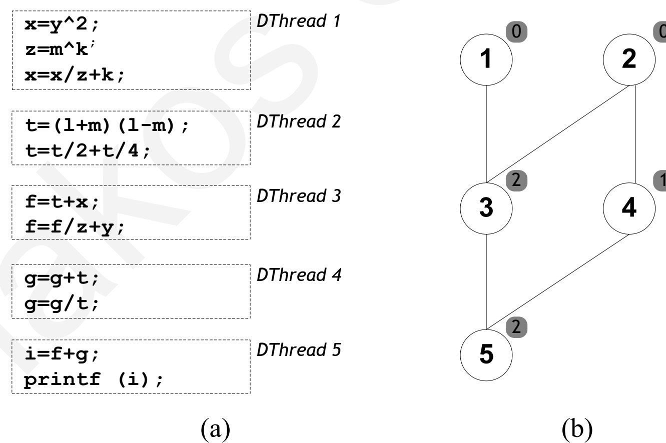


Figure 1: Example of a Data-Driven Multithreading program.

Scheduling a DThread for execution is done dynamically at runtime in a data-driven manner, *i.e.* only when all its producers have completed their execution. The instructions within a DThread are executed by the CPU in a control flow order and any optimization, either by the CPU at runtime

Kyriakos Stavrou

or statically by the compiler, are exploited. Notice that the code of the DThreads can contain any programming construct like simple instructions, control flow instructions, function calls or loops.

DThread scheduling is achieved with the help of the *Thread Synchronization Unit* (TSU) which abstract layout is depicted in Figure 2 (more details about the operation of the TSU can be found in [33]). Prior to the execution of any DThread, the *metadata* of the application's DThreads are loaded into the TSU. For each DThread these metadata consist of a unique identifier for the DThread (*Thread Template*), the DThread's Ready Count value and the Thread Templates of its Consumers.

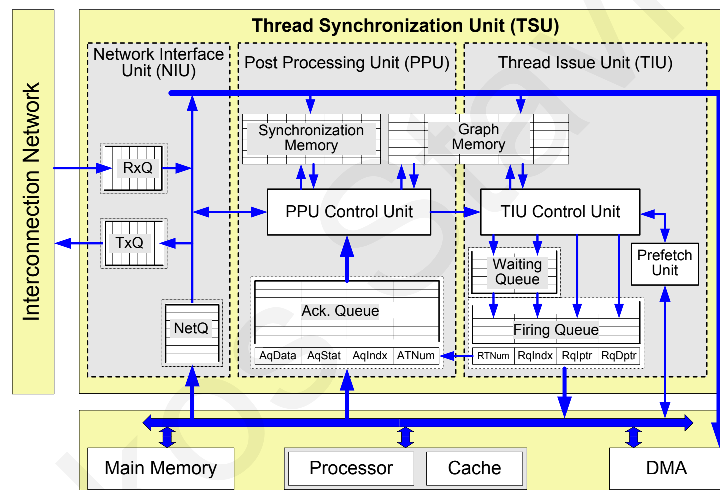


Figure 2: The Thread Synchronization Unit (TSU). *This Figure was taken from [61].*

According to the DDM model the CPUs continually performs a 3-steps process. The first step is to query the TSU for a ready DThread, the second step is to execute this DThread's code and the third step, which is invoked when execution of the DThread completes, is to notify the TSU about this event. Notice that during the first step, in case no ready DThread exists, the TSU will force the CPU to wait.

Kyriakos Stavrou

Each time the TSU is notified about a DThread completion event, it invokes the *Post-Processing Phase*. During this phase the *Ready Count* value of the Consumers of the completed DThread is decreased. When the *Ready Count* of a DThread reaches zero it will be deemed executable and the TSU can schedule it for execution when a CPU becomes available.

To allow applications with arbitrarily large Synchronization Graphs, without requiring equally large TSU storage, DDM programs can be split into *DDM Blocks* or simply *Blocks*. Each Block contains a subset of DThreads of the original program. The maximum number of DThreads a Block may contain is limited by the size of the TSU. In addition to the application's DThreads, each DDM Block has two other DThreads, the *Inlet* and *Outlet* DThread. The purpose of the former is to load the TSU with the metadata of all the DThreads belonging to that Block whereas the purpose of the latter is to clear the allocated resources. When *all* the DThreads of a DDM Block complete their execution, the Outlet DThread is executed. After releasing the allocated resources the Outlet DThreads load the TSU with the Inlet DThread of the next Block to allow execution to proceed.

2.5 Data-Driven Multithreading Network of Workstations (D^2NOW)

The Data-Driven Network of Workstations (D^2NOW) [35, 61, 63] is the first implementation of the DDM model. D^2NOW consists of a collection of interconnected workstations each of which is augmented with a Thread Synchronization Unit (TSU). For the communication of the TSUs, D^2NOW uses another dedicated network.

A major characteristic of D^2NOW is its cache management scheme which is based on the "Cacheflow" [62] policy. According to this policy, the scheduling information of the TSU is used to manage the data present in the L2 cache. Cacheflow has been found to provide significant reduction of the cache misses with an important consequent performance increase.

Kyriakos Stavrou

Overall, D^2NOW proved to have very high performance and good scalability when evaluated using a subset of the SPLASH-2 [129] benchmarks. In addition, a major contribution of D^2NOW is the fact that it proved DDM to be implementable using commodity CPU cores.

Although D^2NOW and TFlux are based on the same execution model, DDM, the two systems differ significantly. The major advantage of TFlux over D^2NOW is the fact that it provides to its user a *virtualized* environment that hides all the details of the underlying system. Programming for TFlux is done at the high-level by augmenting ANSI-C code with compiler directives. Another major advantage of TFlux is that it introduced an implementation, TFluxSoft, where no extra hardware is required. As such, this allows the DDM model to be provided on *off-the-shelf* systems.

2.6 Data-Driven Multithreading Chip Multiprocessor(DDM-CMP)

The main target of the Data-Driven Multithreading Chip Multiprocessor (DDM-CMP) [112] design was to explore the potential of applying the DDM model of execution to the Chip Multiprocessors architecture. In addition to the potential performance, this design studied the power consumption, the hardware cost as well as ways to benefit from the particular characteristics of CMPs architecture.

The DDM-CMP design proved able to deliver not only high performance [112, 113, 120] but also high power efficiency [120]. As for the hardware cost of providing data-driven scheduling on a commodity CMP chip, this was found to be in the order of 1M transistors per CPU [114]. By taking into advantage the fast on-chip communication of the CMP architecture, DDM-CMP examined several alternatives for the scheduling unit for space savings [112]. Moreover, the shared memory environment that is provided by most CMPs allowed implicit communication between the execution units and easier programmability [116].

Kyriakos Stavrou

Chapter 3

The TFlux Parallel Processing Platform

Thread-Flux ***TFlux*** is a parallel processing platform that allows its user to exploit the benefits of the Data-Driven Multithreading (DDM) model of execution on *commodity* multiprocessor systems. ***TFlux*** is composed of a collection of abstract entities that provide the necessary virtualization for the execution of TFlux programs on a variety of computer systems.

This Chapter starts with Section 3.1 presenting the different layers of the platform, *i.e.* the programming toolchain, the Runtime Support, the TFlux Kernels and the Scheduler. Section 3.2 focuses on the basic execution components of the TFlux Platform whereas Section 3.3 on details of the Scheduler. Finally, this Chapter concludes with a brief presentation of the two major incarnations of the TFlux Platform: *TFluxHard* and *TFluxSoft*.

3.1 TFlux Layered Design

The main target of the TFlux platform is to provide the necessary *virtualization* for the parallel execution of programs under the DDM model on *commodity* multiprocessor systems. Figure 3

Kyriakos Stavrou

depicts the layered design of TFlux. The top layer is the one programmers use to develop TFlux applications and its purpose is to abstract *all* details of the underlying machine. TFlux applications are developed using ANSI-C together with TFlux directives [121], which are used to express the DThreads' code and the dependencies among them. The code of the TFlux program (ANSI-C augmented with TFlux directives) passes through the TFlux Preprocessor which outputs a C program able to be compiled by a *commodity* C compiler. As a result we can obtain an executable binary for any ISA. In addition to the code of the initial application, the binary also includes calls to invoke the operations of the *TFlux Runtime Support* allowing execution under the DDM model. 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 *TFlux Scheduler* (throughout the text the term *"Scheduler"* with a capital "S" refers to the scheduler of TFlux).

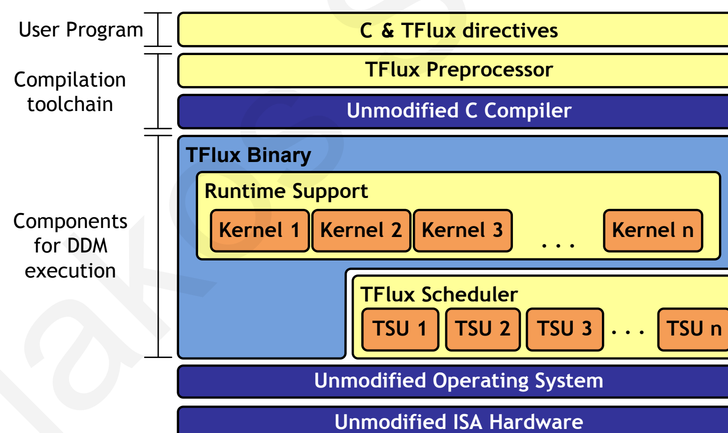


Figure 3: The layered design of the TFlux Platform.

The different layers of the TFlux Platform are presented in detail in the following Sections in an order different to their position on the design for easier explanation.

3.1.1 Runtime Support

The virtualization TFlux provides is mainly due to its *Runtime Support* . The Runtime Support executes on top of an *unmodified* Unix-based Operating System (OS) and allows for the DDM execution to be interleaved with the execution of non-DDM binaries by means of simple OS *context switch* operations.

For the use of a regular OS for DDM execution, the Runtime Support has to satisfy two important requirements. First, when an application is executed in parallel, the runtime has to provide a way for the different DThreads to access the shared variables used in the producer-consumer relationships. Second, to achieve DThread scheduling according to the DDM model, the Runtime Support needs to communicate with the TFlux Scheduler, in order to invoke the appropriate operations on each event.

The Runtime Support meets these requirements with the help of a simple user-level process, the *TFlux Kernel* , which is described in the next Section. The Runtime Support starts its execution by launching n TFlux Kernels, where n is the maximum number of DThreads that can execute in parallel in the machine. Usually, this number is limited by the number of CPUs of the system. The Runtime Support spawns the TFlux Kernels in the same address space which allows the different DThreads of the application to exchange data through shared variables. Another requirement for the Runtime Support is that the DThreads must be scheduled according to the DDM model. This scheduling is performed by the *TFlux Kernels* as will be explained in the following Section. Notice that the Runtime Support terminates its execution and exits when *all* TFlux Kernels have completed their execution, *i.e.* executed all DThreads assigned to them. It is relevant to mention

Kyriakos Stavrou

that the Runtime Support for the TFlux applications and the code of the TFlux Kernels is embedded into the code at compile time. Therefore, each TFlux binary is self-contained, requiring no extra software for its execution such as OS patches or demons.

3.1.2 TFlux Kernel

The main target of the TFlux Kernel is to communicate with the TFlux Scheduler in order to schedule DThreads according to the DDM model.

Figure 4 depicts the operation of the TFlux Kernel. Its first task is to transfer the execution to the first instruction of the *Inlet DThread* (Section 2.4) of the first Block in order for the *metadata* of this Block's DThreads to be loaded into the TFlux Scheduler. At the end of that Inlet DThread, as well as of any other DThread, the control jumps to the Kernel code and more specifically to the *FindReadyThread* loop. At that point the Kernel requests from the Scheduler the next DThread for execution. When this call returns a valid ready DThread, the execution control jumps to the first instruction of that particular DThread. If more than one ready DThreads exist, the TFlux Scheduler returns the one with the subsequent (regarding the DThread that has just completed its execution) Thread Template. When the execution of a DThread completes, control is transfered again to the TFlux Kernel, which notifies the Scheduler about this completion event. The Scheduler will then execute the *Post-Processing Phase* for the completed DThread during which the Ready Count values of its consumers are decreased. Then the TFlux Kernel repeats the same process by querying the Scheduler for the next ready DThread.

To allow execution of multiple Blocks, the *Outlet DThreads*, in addition to deallocating the Scheduler resources, also load the Scheduler with the Inlet DThreads of the next Block. When these Inlet DThreads are executed they load the Scheduler with the metadata of the corresponding Block's DThreads, therefore allowing for the execution to proceed. As for the Outlet DThreads of

Kyriakos Stavrou

the *last* Block, their operation is to force their Kernels to exit. Consequently, the Runtime Support exits and the application completes when *all* DThreads assigned to each Kernel complete their execution.

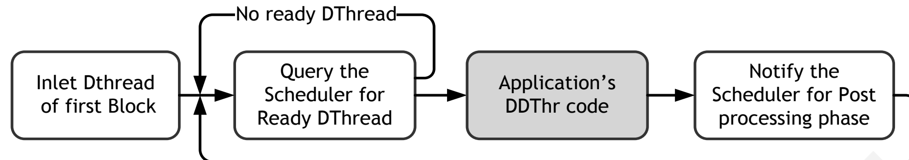


Figure 4: Operation of the TFlux Kernel. Application's code is shown shaded.

The Kernel executes at the user-level. This is achieved by inlining the code of the different functions of the TFlux Kernel (instead of having function calls) inside the program's code wherever such a function invocation is required. The transition from the Kernel to the application's code, and vice-versa, occurs with minimal overhead and is completely transparent to the OS. As these transitions may be frequent, this contributes to minimizing of the runtime overheads.

3.1.3 The TFlux Scheduler

As explained earlier, the purpose of the TFlux Scheduler is to enable DThread execution according to the DDM model. As can be seen from Figure 5 that depicts the layout of the Scheduler, it consists of a set of units that are private to each TFlux Kernel and three units that are shared between all Kernels of the system. Notice that the set of *private* units form the Kernel's *Thread Synchronization Unit (TSU)* [33] and one such TSU exist for each Kernel of the system. The units which are private for each Kernel are the *Graph Memory* (GM), the *Synchronization Memory* (SM) and the Thread Execution Stack (TES) whereas the units shared among the Kernels are the *Threads-to-Update Buffer* (TUB) the *Consumer List* (CL) and the Iteration-level Consumers List (ILCL). The purpose of these units is as follows:

Kyriakos Stavrou

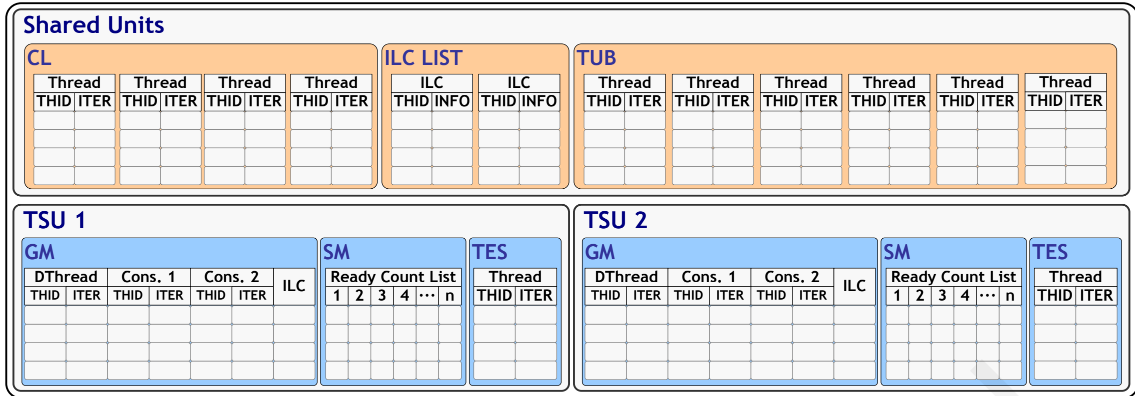


Figure 5: *TFlux Scheduler* configured with 2 TSUs.

***Graph Memory* (GM):** The GM stores the *static* information of the DThreads. For each DThread this information consists of its Thread Template (Section 2.4), the Thread Templates of its Consumer DThreads and an index to the Iteration-level Consumers List that contains information about the Iteration-level Consumers of this DThread.

***Synchronization Memory* (SM):** SM stores the DThread's Ready Count value *i.e.* the DThread's *dynamic* information. As explained earlier, the *Ready Count* value is equal to the number of producers for this DThread that have not yet completed their execution. For DThreads with multiple instances, *i.e.* DThreads executing loops (see Section 3.2.2), it is necessary to store a Ready Count value for each DThread instance. As such, each SM entry provides storage for multiple Ready Count values.

***Thread Execution Stack* (TES):** This unit holds the thread templates of the *ready* DThreads, *i.e.* of the DThreads with Ready Count equal to zero. The DThread that is currently executing is the one at the top of the TES.

Kyriakos Stavrou

***Consumer List* (CL):** CL is shared among all TSUs of the system and its purpose is to allow DThreads with more than two consumers. In particular, if a DThread has up to two consumers their Thread Templates are stored in the GM whereas in a different case their templates are stored in the CL.

***Iteration-level Consumers List* (ILCL):** ILCL is also shared among all TSUs of the system. Its purpose is to store the information that describes the Iteration-level Consumers of the different DThreads.

***Threads-to-Update Buffer* (TUB):** TUB is used during the *Post-Processing Phase*, *i.e.* when the Ready Count values of the Consumers of the completed DThreads are decreased. Whenever a DThread completes its execution the Thread Templates of its consumers are inserted into the TUB (*update-request*). In a later phase (*Thread Update*) another entity reads these *update-requests* from the TUB and decreases the corresponding Ready Count counters.

3.1.4 Compilation Toolchain

Application development for TFlux is done at the high-level by augmenting ANSI-C code with the *TFlux directives*. These directives allow expressing the boundaries of the application's DThreads and dependencies among them. The *TFlux C Preprocessor* (TFluxCpp), which is an extended version of the Data-Driven Multithreading C Preprocessor (ddmCpp) [121], takes the application's code as input and outputs a C program that includes the code of the Runtime Support and of the TFlux Kernels as well as the TFlux interface calls necessary for the program to execute on a TFlux architecture under the DDM model. This program can be compiled into an executable binary using a commodity C compiler.

Further details about TFluxCpp will be given in Chapter 4, which is dedicated to the presentation, analysis and evaluation of TFluxCpp and its directives.

Kyriakos Stavrou

3.2 Basic Execution Components

This Section presents the basic execution components of the TFlux Platform. These components regard the programming constructs provided to the programmer for the development of applications to be executed on TFlux under the DDM model.

3.2.1 Data-Driven Threads (DThreads)

The principal element of TFlux programs is the *DDM Thread* (DThread). Each DThread 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. Inside each thread instructions are executed in control flow order allowing the processor or the compiler being able to apply any optimization (*e.g.* out-of-order execution). The different DThreads of the application can be executed in parallel unless a data dependence exits among them.

A DThread is characterized by its *static code*, its *Thread Template*, its *Ready Count* and its *Consumers*. Notice that the tuple Thread Template, Ready Count and Thread Templates of the Consumers is what we call the *"metadata"* of a DThread. To better explain these notions we will use the example depicted in Figure 6.

Figure 6-(a) presents the necessary operations for calculating the Binomial probability for n experiments with k successes and Figure 6-(b) the corresponding Synchronization Graph. Recall that the nodes of this graph represent DThreads and arcs the data dependences between them.

The *Thread Template* is a *unique* identifier for each DThread, in this example the Synchronization Graph of the program consists 8 DThreads each with a different Thread Template (1, 2, ..., 8). Notice that in general the Thread Template is not just a single number but a two-fields tuple consisting of the the *Thread Id* (THID) and the *Iteration Id* (ITER). The purpose of this second field,

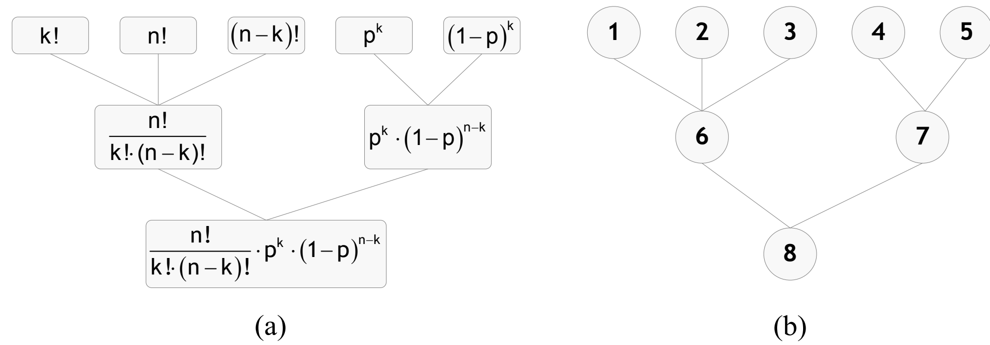


Figure 6: Calculation of the Binomial probability using DDM threads.

Iteration Id, as will be explained in Section 4.3.2 regards the execution of loops. For the analysis presented here, what is used as the Thread Template is just the *Thread Id*.

According to the DDM model, the Consumers of DThread X are the DThreads that can not start their execution unless DThread X has completed its execution; equivalently, a DThread can start its execution only when *all* its *Producers* have completed their execution. As an example, the Producers of DThread 6 are DThreads 1, 2 and 3 whereas the Consumer of DThread 3 is DThread 6. Finally, the *number of Producers* for a DThread is equal to its *Ready Count* value. In the previous example the Ready Count value of DThread 6 is 3 whereas the Ready Count of DThread 7 is 2. It is important to notice that DThreads that have no dependence between them (*e.g.* DThreads 4 and 5) can be executed in parallel. Table 1 summarizes the characteristics of the DThreads of this particular example.

3.2.2 TFlux Loops

Loops are known to be a very common structure in compute-intensive applications. For performance improvement, the algorithms of many applications are modified to allow implementations

Kyriakos Stavrou

Table 1: The DThreads of the example of Figure 6.

Thread Id	***Ready Count***	***Consumers***	***Producers***	***Code***
1	0	6	-	$k!$
2	0	6	-	$n!$
3	0	6	-	$(n-k)!$
4	0	7	-	p^k
5	0	7	-	$(1-p)^{n-k}$
6	3	8	1,2,3	$\frac{n!}{k!\cdot(n-k)!}$
7	2	8	4,5	$p^k \cdot (1-p)^{n-k}$
8	2	-	6,7	$\frac{n!}{k!\cdot(n-k)!} \cdot p^k \cdot (1-p)^{n-k}$

with *parallel* loops, *i.e.* loops for which there are no data dependencies among the different iterations. This is due to the fact that it is very easy to parallelize these loops leading to significant performance benefits.

A common case is for these loops to depend on each other. These dependencies can either be at the *loop level*, *i.e.* no iteration of the dependent loop can proceed unless *all* iterations of the producer loop have completed thei execution, or at the *loop iteration level* meaning that an iteration of the dependent loop can proceed after particular iterations of the producer loop have completed.

Most parallel programming models do not provide native support for the execution of loops with dependencies at the *loop iteration level*. Examples of such models include OpenMP [81] and MapReduce [30] paradigms. Consequently, in order to enforce the necessary synchronization between the loops they introduce barriers. This leads to an implementation that often inserts unnecessary synchronization points with a consequent negative impact on the performance.

TFlux provides special support for the execution of parallel loops which, in addition to supporting *fully* parallel loops, also covers loops with dependencies at the iteration level. We use the general term *"TFlux Loop"* to cover both categories. Formally, a *TFlux Loop* is a loop *each* iteration of which may be deemed executable based on a *different* data dependency condition.

Kyriakos Stavrou

This feature is of major importance as it allows converting the barriers between loops into dependencies between loop iterations. As such, synchronization is enforced only when it is necessary according to the true data dependencies. As will be explained in Chapter 9, that presents the quantitative evaluation of TFlux, this feature is highly relevant for the performance of TFlux.

The TFlux Loops are executed by a special category of DThreads named *"Loop DThreads"* (*L-DThreads*) and each such L-DThread executes a *different* iteration of the TFlux Loop. The L-DThreads executing the same loop have the same static code, the same Ready Count value and the same Thread Id. What differentiates them is the second field of the Thread Template, *i.e.* the *Iteration Id* (ITER). As such, the L-DThreads should be seen as instances of the same entity; each different instance however, executes a different iteration of the loop. As for the TFlux Scheduler, the Runtime Support and the TFlux Kernels, notice that they handle both the DThreads and the L-DThreads in the same way.

For the representation of L-DThreads we use the convention *T/I* where *T* refers to the Thread Id whereas *I* to the Iteration Id. For example, *2/7* is the Thread Template of the L-DThread with Thread Id equal to 2 and Iteration Id equal to 7. For the Thread Template of "normal" DThreads (*i.e.* not L-DThreads) we often use a shorter representation which consists of the Thread Id only (e.g. 2). Another convention that is used throughout the text regards the TFlux Loops which are symbolized with only one number which is the Thread Id of the L-DThreads executing it (as explained earlier the Thread Id is common for all L-DThreads executing the same loop).

Kyriakos Stavrou

3.2.2.1 Dependencies of TFlux Loops

The dependencies of TFlux loops can be partitioned into three categories: between a TFlux Loop and a DThread, between TFlux Loops and between loop iterations.

Dependency between TFlux-Loops and DThreads

The first category regards the dependency between TFlux-Loops and DThreads. An example of this dependency is depicted in Figure 7-(a). When a TFlux-Loop (x) depends on a DThread (y), no L-DThread of TFlux Loop x can start its execution unless DThread y has completed. Similarly, when a DThread (y) depends on a TFlux-Loop (x), DThread y can not start its execution unless *all* L-DThreads of TFlux Loop x have completed their execution. This dependency, *i.e.* between DThreads and TFlux-Loop is represented as shown in Figure 7-(b).

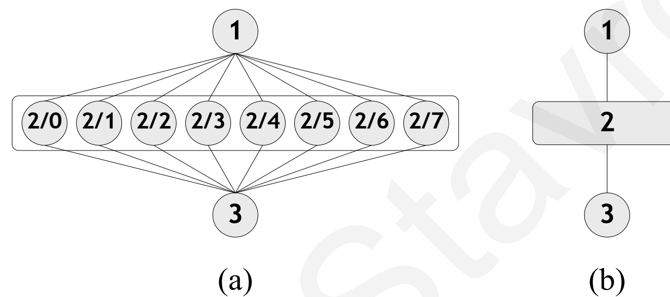


Figure 7: Dependencies between a TFlux Loop and DThreads.

Kyriakos Stavrou

Dependency between TFlux-Loops

The second category, depicted in Figure 8-(a), refers to the dependency between TFlux Loops. This dependency means that no L-DThread of the consumer TFlux-Loop (TFlux Loop 3 in the example) can start its execution unless *all* L-DThreads of the producer TFlux Loop (TFlux Loop 2 in the example) have completed their execution. As such, there is a dependency between *each pair* of L-DThreads of the two TFlux Loops. Figure 8-(b) shows how the dependency between TFlux Loops is represented.

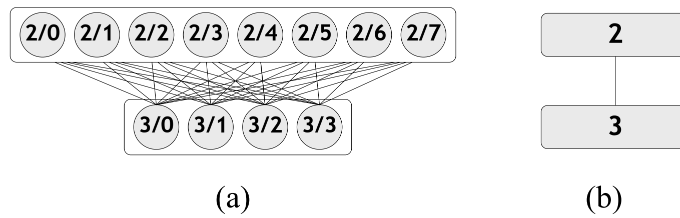


Figure 8: Dependency between TFlux Loops.

Dependency at the level of loop iterations

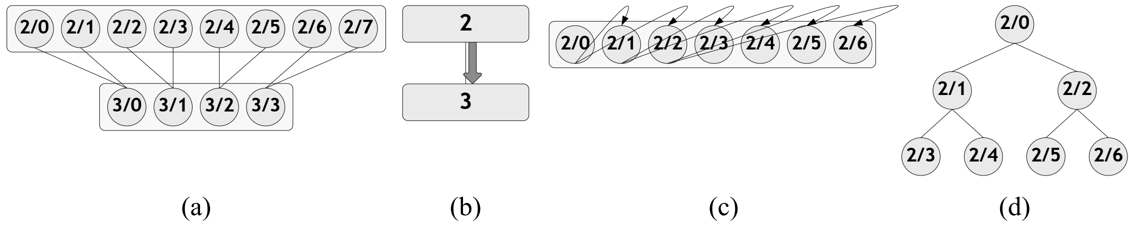
Finally, the third category refers to dependencies at the iteration level of TFlux Loops (Iteration Level Dependencies - ILD) *i.e.* dependencies between L-DThreads of the same or different loops (Figure 9-(a)). More specifically, if L-DThread y depends on L-DThread x, L-DThread y can start its execution after L-DThread x has completed *regardless* the progress of the *other* L-DThreads of the TFlux Loops.

An example of this situation is depicted in Figure 9-(a) where L-DThread 3/0 can start its execution after L-DThreads 2/0 and 2/1 of Loop DThread 2 have completed. When two TFlux Loops depend at the iteration level, the dependency is shown using a thick arrow as depicted in Figure 9-(b).

As mentioned before it is possible to have Iteration Level Dependencies between L-DThreads of the *same* TFlux Loop. For example, the loop depicted in Figure Figure 9-(c) corresponds to the Synchronization Graph shown by Figure 9-(d).

3.2.2.2 Execution of TFlux Loops

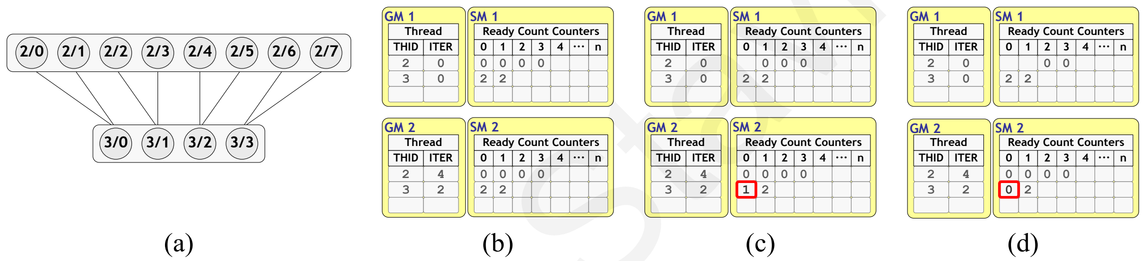
As mentioned earlier, each iteration of a TFlux Loop is executed by a different L-DThread and all L-DThreads executing the same TFlux Loop have the same *initial* Ready Count value (dynamically the Ready Count value of the different L-DThreads will be different according to



(a) (b) (c) (d)

Figure 9: Iteration-Level Dependencies.

the execution scenario), the same static code and the same Thread Id. This is the case depicted in Figure 10 where all L-DThreads of TFlux Loop 2 have Ready Count equal to 0.



(a) (b) (c) (d)

Figure 10: Execution of TFlux Loops.

As all *static* information is common for all L-DThreads executing the same TFlux Loop it is possible to store their metadata in the *same* Graph Memory row (Figure 10-(b)). As for the Ready Count counters, they are kept in a single row of the Synchronization Memory (Figure 10-(b)). Recall that a single Synchronization Memory entry provides multiple Ready Count counters allowing one such counter per L-DThread.

The L-DThreads of the TFlux Loops are distributed to the different Kernels of the system. As such, each TSU will have an entry in its internal structures corresponding to each TFlux Loop. Notice that for the current version of TFlux the assignment of DThreads and L-DThreads to the

Kyriakos Stavrou

different execution nodes is static, however in the future dynamic assignment policies are to be applied.

Referring to Figure 10, when the first TFlux Loop is to be executed by a system with 2 nodes, each Kernel will be assigned 4 iterations (this loop has 8 iterations). The GM entry corresponding to the L-DThreads of this TFlux Loop in the first TSU has Iteration Id (ITER) equal to 0. As for the corresponding SM entry it uses 4 Ready Count counters, which means that this row has 4 L-DThreads and consequently the particular Kernel will execute 4 iterations of the loop. Notice that all Ready Count counters have been initialized with 0 which is the Ready Count value of the L-DThreads of the TFlux Loop. For the second TSU, the corresponding Iteration Id is equal to 4 and again 4 Ready Count counters are used. In general, the Iteration Id of an L-DThread is equal to the sum of the ITER field of its corresponding GM entry plus its offset in the SM. For this example the L-DThreads of first TSU will execute loop iterations 0,1,2 and 3 whereas the L-DThreads of the second TSU the iterations 4, 5, 6 and 7.

Providing a different Ready Count counter for each L-DThread allows exploiting parallelism at the level of loop iterations. As an example, consider the Synchronization Graph depicted in Figure 10-(a). In this particular case, L-DThread $3/0$ can start its execution after L-DThreads $2/0$ and $2/1$ have completed (Figure 10-(b) to 10-(d)) regardless the progress of the other L-DThreads. As such, it is possible for L-DThreads of TFlux Loop 3 to start their execution without waiting for *all* L-DThreads of TFlux Loop 2 to complete their execution.

3.2.2.3 Execution of loops with large number of iterations

An important issue for the execution of TFlux Loops regards the case of loops with a very large number of iterations compared to the number of available Ready Count counters. This is due to the fact that providing as many Ready Count counters as the number of loop iterations would

significantly increase the Scheduler's space requirement. To avoid this situation, TFlux applies a technique named *"L-DThread Recycling"* which is explained through the example presented in Figure 11. For this example assume that TFlux Loop 2 has 32768 iterations and it is executed in a system with 2 TFlux Kernels. Moreover, assume that each SM row has 32 Ready Count counters.

When the metadata of the of L-DThreads executing this TFlux Loop are *initially* loaded into the TSU *only one* Graph Memory (GM) and *only one* Synchronization Memory (SM) row are allocated in each TSU of the Scheduler (first generation L-DThreads). As can be seen from Figure 11-(a) initially the Iteration Id for DThread 2 is equal to 0 for TSU 1 and 32 for TSU 2. Given that all 32 Ready Count counters of the SM row are used, the L-DThreads loaded into the TSU 1 correspond to the iterations 0-31 whereas for TSU 2 to the iterations 32-63.

However, as mentioned earlier the TFlux Loop has 32768 iterations which is larger than the number of L-DThreads that have been initially created (which are only 64 *i.e.* 32 per row). In order to allow for the execution of the loop, the TFlux Preprocessor adds the necessary code to inform the TFlux Kernels about the number of iterations of this loop. Consequently, each time an L-DThread completes the execution of a loop iteration, the TFlux Kernel will apply the *L-DThread Recycling* technique to that DThread.

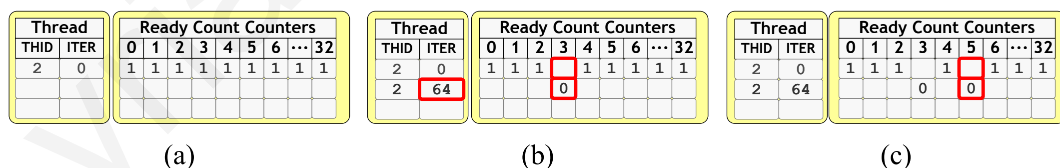


Figure 11: Execution of TFlux Loops with large number of iterations.

According to this technique, when an L-DThread completes its execution it *"recycles"* itself creating a new entry in the TSU structures (that corresponds to a new L-DThread), which will

Kyriakos Stavrou

execute a different iteration of the TFlux Loop. The iteration this new instance is going to execute is the sum of the Iteration Id of the L-DThread that created it and the *parallelism* of the loop execution, *i.e.* the number of L-DThreads that were originally loaded to execute the particular TFlux Loop. In this example, 32 L-DThreads were created per TFlux Kernel for the execution of the TFlux Loop, as such the *parallelism* is equal to 64.

As an example of the *L-DThread Recycling* technique, consider the situation depicted in Figure 11-(b). Here the L-DThread that completed had Iteration Id equal to 3. As such, its new instance will have Iteration Id equal to 67 ($3 + 64$) (Figure 11-(b)). For this new L-DThread to be handled by the Scheduler, its metadata need to be stored in the internal structures of its TSU. As can be seen from the Figure, this new instance is stored in a separate GM / SM row. More specifically, for this new row the Iteration Id is equal to 64 and the Ready Count counter used for this L-DThread is the 3rd one (64+3=67). However, it is not for all new L-DThreads that a new GM/SM is created but instead only when this is necessary. Consider for example the completion of the L-DThread with Iteration Id equal to 5, which will create a new L-DThread with Iteration Id equal to 69. As depicted in Figure 11-(c), the metadata of this L-DThread can be saved in the same row that has been previously created when the metadata of L-DThread 2/67 was previously saved. This saves valuable space in the Scheduler's structures.

As can be seen from Figures 11-(b) and 11-(c) whenever an L-DThread recycles, its Ready Count counter is deallocated. As such, when all the L-DThreads of a row recycle, that row can be reused. This technique is what enables the execution of TFlux Loops with arbitrarily large number of iterations to be completed with limited storage resources. When the desired number of iterations has been covered, instead of performing the *L-DThread recycle* operation, the DThreads will perform the conventional *Thread Completion* operation. These L-DThreads are called the *last-generation L-DThreads*.

Kyriakos Stavrou

L-DThread Recycling also allows reducing the Ready Count values used in TFlux applications. In particular, if a TFlux Loop with 32768 iterations depends on a DThread, this DThread does not need to have as many Consumers. In contrast, the DThread will have as Consumers only the *first generation* of L-DThreads, *i.e.* the ones initially loaded into the TSU. This also applies for the consumers of the TFlux loop, *i.e.* only the *last generation* of L-DThreads updates the subsequent DThreads.

3.2.2.4 Reduction TFlux Loops

Reduction Loops is a special category of TFlux Loops which perform a reduction operation that can be calculated in parallel. An example of such a loop is depicted in Figure 12 and regards the calculation of the sum of all elements of array A. In addition to the L-DThreads that execute the loop iterations, these TFlux Loops have one extra DThread per Kernel to perform the reduction operation (Figure 13). For each Kernel this extra DThread is executed only after *all* its L-DThreads that correspond to this TFlux Loop have completed. The extra DThread for one of the Kernels (main TFlux Kernel - usually TFlux Kernel 0) in addition to waiting for all L-DThreads of its Kernel regarding the execution of this loop to complete, also waits for the extra DThreads of the other Kernels to complete in order to safely perform the global reduction operation (Figure 12). Notice that to allow better scalability, for configurations with large number of TFlux Kernels it might be beneficial to have additional DThreads that are organized in a multilevel structure for the reduction operation. Nevertheless, at this point, multilevel reduction is not supported automatically by the Preprocessor tool, therefore requiring the manual creation of the DThreads and their code.

Finally, it must be noted that the dependencies of the Reduction TFlux Loops have the same properties as the dependencies for the TFlux Loops as they have been presented in Section 3.2.2.1.

Kyriakos Stavrou

```
sum=0;
for(i=0;i<n;i++)
{
  sum=sum+A[i];
}
```

Figure 12: An example of TFlux Loop that performs a reduction operation.

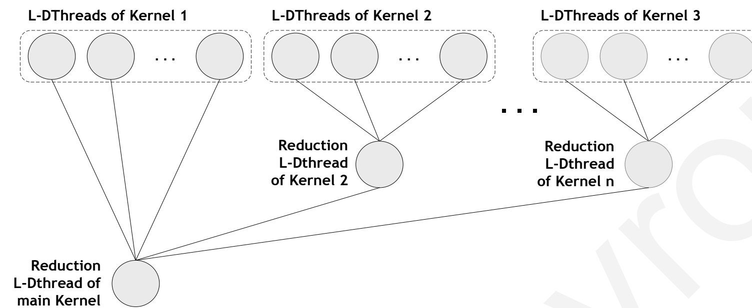


Figure 13: Reduction TFlux Loops.

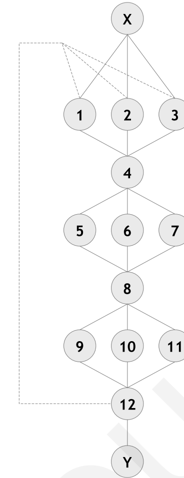
3.2.3 Thread Recycling

TFlux provides support for repeating the execution of one or more parts of the application's Synchronization Graph through a technique name *"Thread Recycling"*. The rationale of this technique is similar to that of *L-DThread Recycling*. An example of a program using this technique is presented in Figure 14 and regards the the calculation of Elliptic Integrals.

As can be seen from Figure 14-(b) which depicts the Synchronization Graph of this example, the body of the while-loop (DThreads 1-11) is executed multiple times according to the condition evaluated by DThread 12. Notice that this technique allows these DThreads to be executed multiple times without the requirement to add their templates multiple times into the Scheduler, *i.e.* one single instance is able to execute many times.

Kyriakos Stavrou

```
do
{
  sX=sqrt(xt);                    //DThread 1
  sY=sqrt(yt);                    //DThread 2
  sZ=sqrt(zt);                    //DThread 3
  alamb=sX(sY+sZ)+sY*sZ;          //DThread 4
  xt=0.25*(xt+alamb);             //DThread 5
  yt=0.25*(yt+alamb);             //DThread 6
  zt=0.25*(zt+alamb);             //DThread 7
  ave=THIRD*(xt+yt+zt);           //DThread 8
  X=(ave-xt)/ave;                 //DThread 9
  Y=(ave-yt)/ave;                 //DThread 10
  Z=(ave-zt)/ave;                 //DThread 11
}
//DThread 12 evaluates the condition
while(MAX(MAX(abs(X),abs(H)),abs(dZ))>err);
```



(a) (b)

Figure 14: (a) An example program for Thread Recycling. The code comes from [86] and is used for the calculation of Elliptic Integrals. (b) The Synchronization Graph of this program.

To enable recycling, the Scheduler handles the DThreads that belong to the "recycle-group" in a special way. In particular, each time such a DThread completes its current execution, instead of invalidating its metadata, the Scheduler keeps these entries in order to allow it to execute again. When the DThread that controls the execution of the recycle-group (DThread 12 in the example) completes, based on the condition it evaluated, it either "wakes-up" the DThreads of the recycle-group (so that these DThreads will execute again) or the DThreads that follow (so that the DThreads of the recycle-group will not execute again), *i.e.* DThread *Y*.

3.2.4 Blocks

As explained earlier, for the DThreads to be executed under the DDM model, their metadata first needs to be loaded into the Scheduler. The storage of this unit is limited, therefore to allow programs with arbitrarily large Synchronization Graphs it is necessary to dynamically load and clear the Scheduler. This dynamic metadata management is done through the use of *DDM Blocks*.

Kyriakos Stavrou

A *DDM Block* (or simply *Block*) is an entity that encloses DThreads whereas *all* DThreads of an application must be inside a DDM Block. The maximum number of DThreads in a DDM Block is defined by the size of the TSU. Each program must have at least one DDM Block whereas the number of DDM Blocks in a program is not limited to any maximum value.

In addition to the application's DThreads each DDM Block has two additional DThreads *for each TFlux Kernel*, the *Inlet* and *Outlet* DThreads. The *"Inlet DThread"* loads into the TSU the metadata of all DThreads executed by its TFlux Kernel which belong to the specific DDM Block.

As for the *"Outlet DThread"* it performs two operations. The first is to release the resources allocated for the execution of the DThreads of its Block. The second operation is performed when there is another Block that follows. If this is the case, the Outlet DThread loads the Scheduler with the metadata of the next Block's Inlet DThreads. As already explained, the Inlet DThread will load the metadata of this Block's DThreads into the TSU and therefore enable the execution to proceed.

Finally notice that whereas the DThreads inside a DDM Block are executed in a Data-Driven manner, the different DDM Blocks are executed sequentially, *i.e.* the execution of a DDM Block starts only after *all* DThreads of the previous Blocks have completed their execution.

Inter-block Sequential Code

The code that exists between Blocks, is always executed by one Kernel only, *i.e.* it is executed sequentially. Notice that for the sequential code to be executed, *all* the DThreads of the previous Block need to have completed their execution. Similarly, the DThreads of the Block that follows can start their execution only after the sequential code has completed.

Kyriakos Stavrou

3.3 TFlux Scheduler Basic Operations

To enable execution under the DDM model the Scheduler is required to provide to the TFlux Kernels five operations: (1) the *"Thread Load"* operation which enables the TFlux Kernels to load the Scheduler with the metadata of the DThreads to be executed; (2) the *"Find Next Thread"* operation which is used by the Kernel to find the next ready DThread to execute; (3) the *"Thread Completion"* operation by which the Scheduler is notified each time a DThread completes each execution; (4) the *"Thread Update"* operation by which the Ready Count values of the consumers of the completed DThreads is decreased and finally, (5) the *"Clear TSU operation"* which clears the resources allocated onto the several units of the Schedulers. These five operations are named, the Scheduler's *Basic Operations*.

In the Sections that follow we present more details for each of these Basic Operations. Notice that the interface for these *Basic Operations*, which is summarized in Table 2, forms the API of the TFlux Scheduler.

3.3.1 Thread Load

The *Thread Load* operation is performed by the TFlux Kernel during the execution of the *Inlet DThread*. This operation loads the metadata of the DThreads into the Graph Memory and Synchronization Memory structures. If a DThread has more than two consumers, their Thread Templates are written in the Consumer List (CL). Moreover, if the DThreads have Iteration-level Consumers, their information is stored in the Iteration-level Consumers List (ILCL).

3.3.2 Thread Completion

The Thread Completion operation has three different versions which are analyzed in the paragraphs that follow.

Kyriakos Stavrou

Table 2: The API of the TFlux Scheduler

1. Load Thread	
Input	Metadata of the DThreads to be loaded
Output	-
Invoked By	Inlet DThread
When	During Inlet DThread's Execution
2. Thread Completion	
2.1. Execution Completion	
Input	-
Output	-
Invoked By	TFlux Kernel
When	DThread completes
2.2. L-DThread Recycle	
Input	Offset for the new Iteration Id
Output	-
Invoked By	TFlux Kernel
When	L-DThread Recycles
2.3. Thread Recycle Execution	
Input	New Ready Count
Output	-
Invoked By	TFlux Kernel
When	DThread Recycles
3. Thread Update	
Input	Content of the TUB
Output	-
Invoked By	-
When	Runs continually
4. Find Ready Thread	
Input	-
Output	Thread Template of Ready DThread
Invoked By	TFlux Kernel
When	The CPU request a new DThread for execution
5. Clear TSU	
Input	-
Output	-
Invoked By	Outlet DThread
When	During Outlet DThread's Execution

3.3.2.1 Execution Completion

When a DThread completes its execution its Kernel invokes the *Execution Completion* operation which is a two steps process. First the Kernel inserts the Thread Templates of the Consumers of the completed DThread in the *"Threads-to-Update Buffer"* (TUB) in order for their Ready

Kyriakos Stavrou

Count values to be decreased. The second step is to remove the completed DThread from the *Thread Execution Stack* (TES).

3.3.2.2 L-DThread Recycle

As explain in Section 3.2.2.2, during the execution of TFlux Loops, all L-DThreads except those of the last generation recycle themselves to execute a new iteration. For this purpose, instead of performing the *Thread Completion* operation, these L-DThreads execute the *L-DThread Recycle* operation.

During *L-DThread Recycle* operation the TFlux Kernel removes the completed L-DThread from the Thread Execution Stack (TES) and inserts the new instance of this L-DThread in the Graph and Synchronization Memory structures. For L-DThreads with Iteration Level Consumers this operation also inserts the identifiers of these consumers into the TUB.

3.3.2.3 Thread Recycle Execution

Another variation of the *Thread Completion* operation is *Thread Recycle Execution* which enables executing of a part of the application's Synchronization Graph multiple times (Section 3.2.3). In particular, when the DThreads that belong the part of the Synchronization Graph that is repeated complete their execution, instead of executing the *Execution Completion* operation, they execute the *Thread Recycle Execution* operation. In addition to the actions taken by the *Execution Completion*, *Thread Recycle Execution* reinserts the metadata of the completed DThread into the Graph and Synchronization Memory structures. The only difference of *Thread Recycle Execution* compared to *L-DThread Recycle* is that whereas the former reinserts the DThread with exactly the same metadata the latter modifies the Iteration Id field in order for the new instance of the L-DThread to execute a different iteration.

Kyriakos Stavrou

3.3.3 Thread Update

The purpose of this operation is to decrease the Ready Count value of the Consumers of the completed DThreads which have been inserted by the TFlux Kernels in the TUB during the *Thread Completion* operation. During this operation the TUB is traversed and for each valid entry the corresponding Ready Count counter is decreased.

3.3.4 Find Ready Thread

The task of this operation is to return to the CPU a ready DThread. In the common case, where the *Thread Execution Stack* (TES) is *not* empty, the operation completes by returning the DThread at the head of the TES. If the TES *is* empty, the Graph and Synchronization Memory structures are first traversed and the Thread Templates of all ready threads are copied into the TES. If these structures contain no ready DThreads the process will repeat until such a DThread is found.

3.3.5 Clear TSU

This operation is performed by the TFlux Kernel during the execution of the *Outlet DThread* and its result is to release the resources allocated onto the several TSU units for the execution of the DThreads of the particular Block.

3.4 TFlux Incarnations

Currently TFlux has two incarnations, *TFluxHard* and *TFluxSoft* which are presented in detail in Chapters 5 and 6 respectively. These two systems differ only to the implementation of the Scheduler and the corresponding interface. In particular, for *TFluxHard*, the Scheduler is a hardware component and is attached to the on-chip system's network as a memory mapped device whereas for *TFluxSoft*, the Scheduler's functionality is provided at the software level. As for the

Kyriakos Stavrou

other components of the system, *i.e.* the TFlux Preprocessor, the Runtime Support and the TFlux Kernels they are common for both incarnations.

The basic characteristic of *TFluxSoft* is its ability to execute using *off-the-shelf* components which makes it *directly* applicable to *existing* multiprocessors systems. The fact for that *TFluxHard* the Scheduler is provided by a *hardware* unit does not allow it to be used today by *off-the-shelf* systems. In contrast, *TFluxHard* targets multicores that will allow extension of the system with such modules.

Kyriakos Stavrou

Chapter 4

TFlux Preprocessor

The ***TFlux Preprocessor*** (*TFluxCpp*) is the tool with which the users develop programs for the TFlux Platform. Using TFluxCpp it is possible to develop TFlux programs by augmenting ANSI C code with dedicated compiler directives, named the *"TFlux directives"*. With these directives, which are presented in Section 4.3, the user defines the boundaries of DThreads, their type (DThreads or L-DThreads), and the dependencies among them.

To better explain the operation of TFluxCpp we will first present the structure for the code of TFlux applications in Section 4.1. Then Section 4.2 analyzes the operation of TFluxCpp. Section 4.3 presents TFlux directives and simple example programs. What follows is a qualitative evaluation of the expressibility of the TFlux directives in Section 4.4 and finally Section 4.5 presents the limitations of this tool.

Kyriakos Stavrou

4.1 Structure of a TFlux Application Code

The code of a TFlux application consists of the code of the original program together with the code added to allow execution under the DDM model. The main purpose of this additional code is to allow the Runtime Support system to interact with the Scheduler in order to perform data-driven Scheduling.

Figure 15 depicts the most important parts of code of a TFlux program. The first element shown in this Figure regards initializations of structures necessary for the execution of TFlux programs (*TFlux Initializations*). These initializations, among others, include the number of TFlux Kernels and the size of the fields in the Thread Template (Thread Id and Iteration Id).

The second component, regards the creation of TFlux Kernels (*TFlux Kernels Creation*). Upon creation, TFlux Kernels load the Scheduler with the metadata of the Inlet DThread of the first Block (*Load Inlet of First Block*) with a Ready Count value equal to zero (this is to make these DThreads immediately executable). Then the Kernels are redirected to a special loop, the purpose of which is to request from the Scheduler a ready DThread and redirect execution to its first instruction (*Thread Select Loop*). If the Scheduler has nothing to return, *i.e.* no DThread is ready for execution, this process is repeated.

As mentioned several times, DThreads are represented by their Thread Template. However, when the execution of a DThread is to be initiated, the Runtime Support is required to know the *address of the first instruction* of this DThread. This information, *i.e.* translation from the Thread Template to the address of the corresponding DThread's first instruction, is provided by a code segment of the application's code. This code segment, which is part of the *Thread Select Loop*, consists of a *switch* statement, each element of which contains a branch to a different label (*Thread*

Kyriakos Stavrou

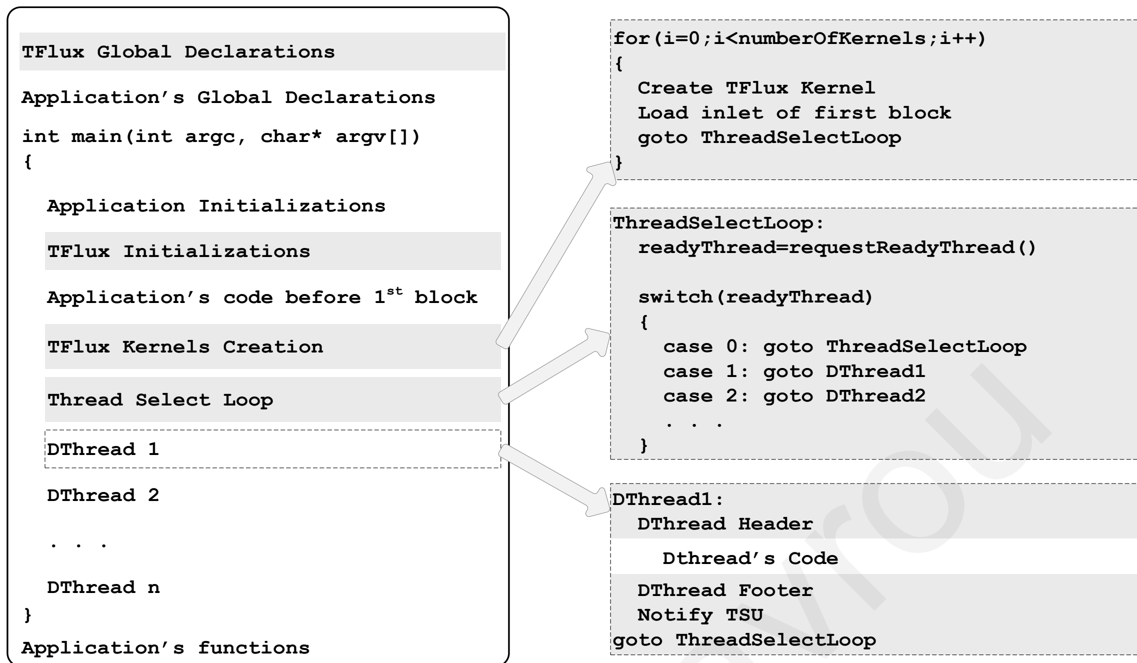


Figure 15: The structure of the TFlux program.

Label), and such a label exists for each DThread. This label points to the first instruction of the corresponding DThread.

The *Thread Label* is the first information of the header of DThreads (*Thread Header*). For regular DThreads the Label is the only information in the Header. For L-DThreads the *Thread Header* also includes information to set the control variable to its proper value in case the lower bound of the loop is calculated dynamically, or it is not equal to zero, or if the loop has been unrolled. Similar adjustment of the control variable needs to be made for the different iteration scheduling policies (see Section 4.3.2.1).

The code of the DThread is followed by the *Thread Footer*. The *Thread Footer* includes a call to notify the Scheduler that the execution of the DThread has completed and is followed by an unconditional branch to the *Thread Select Loop* (*goto threadSelectLoop*). For the case of L-DThreads, the footer also includes code to define the termination condition. In particular, this code

Kyriakos Stavrou

examines if the completed L-DThread should recycle itself to execute another iteration of the loop or if it belongs to the last generation (Section 3.2.2.2).

4.2 Phases of the TFlux Preprocessor

The purpose of TFluxCpp is to take as input the user program written in ANSI C augmented with the TFlux directives and output the equivalent ANSI C program that operates under the DDM model. This code can then be compiled using a *commodity* C compiler. Notice that parts of this output program may differ for different TFlux incarnations (TFluxHard and TFluxSoft).

TFluxCpp operates in two phases. During the first phase TFluxCpp parses the TFlux program and builds the Synchronization Graph of the application whereas during the second phase, it creates the equivalent output code for this program. As the TFlux directives used in the input code are the same for different target incarnations, the first phase is common. As for the second phase, the back-end that produces the output program, there is a different version of this back-end depending on the target implementation. This is mainly due to the fact that for different implementations, the Scheduler's interface may differ. Nevertheless, the large majority of the output program is common for both TFlux incarnations.

4.2.1 Phase 1: Parsing

To create the Synchronization Graph of the input program, TFluxCpp parses the directives in order to identify the type and the dependencies between the DThreads. Understanding the *type* of the DThreads is necessary as some directives declare DThreads that have multiple instances (*e.g.* TFlux Loops). Each time TFluxCpp finds the declaration of a DThread it inserts it in an internal list (DThread List). For the case of TFlux Loops, the preprocessor inserts into this list multiple DThreads, one per TFlux Kernel. The same stands for DThreads defined by the user

Kyriakos Stavrou

to be executed by *all* TFlux Kernels (Section 4.3.1.3). Another situation for which additional DThreads are created regards the DThreads performing the reduction operation of TFlux Loops (Section 4.3.2.3). After all DThreads have been created and data dependencies have been defined, TFluxCpp creates the program's Synchronization Graph.

The next step for TFluxCpp is to create the *Inlet* and *Outlet* DThreads. As explained in Section 3.1.2, each Block has one *Inlet* and one *Outlet* DThread *per* TFlux Kernel. The main purpose of the *Inlet* DThreads is to load onto the TSU all DThreads to be executed by their TFlux Kernel. The DThreads each *Inlet* loads onto the TSU are easily found by TFluxCpp by traversing the DThreads List. Notice that all DThreads of a Block that do not have Producers are set to depend on the *Inlet* DThreads. Similarly, the DThreads that do not have any Consumer are set to have as Consumer the *Outlet* DThread of their Kernel for their block.

The final step is to set the dependencies between the *Inlet* and *Outlet* DThreads to enable the execution of consecutive Blocks and also of the sequential code between Blocks. Notice that the sequential code is only executed by one Kernel, which we name the *First Kernel*. In order to achieve the execution of consecutive Blocks, all *Outlet DThreads* are set to load into the TSU the *Inlet DThread* of the next Block for their Kernel. In addition, to enforce the necessary synchronization (a DThread can start its execution only after *all* DThreads of the previous Block have completed their execution) the *Inlet* of the *First Kernel* has as consumers all the other *Inlets* of the same Block (Figure 16). Similarly, the *Outlet* of the *First Kernel* is a consumer of all other *Outlets*. Finally, the *Inlet* of the *First Kernel* of a Block is a consumer of the *Outlet* of the *First Kernel* of the previous Block. As for the *Outlet* of the last Block, it is set by TFluxCpp to force its Kernel to exit.

During this first step, in addition to creating the Synchronization Graph, TFluxCpp creates a list of variables that need to be declared in a special way. These variables are used for the TFlux

Kyriakos Stavrou

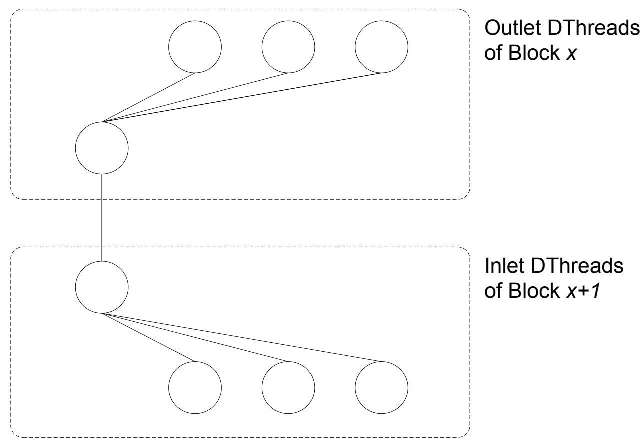


Figure 16: Synchronization of Inlet / Outlet DThreads of consecutive DThreads.

Loop reduction operation which needs to be shared among all TFlux Kernels as well as global memory addresses needed for the communication between the TFlux Kernels and the Scheduler.

4.2.2 Phase 2: Creation of Output Code

Based on the Synchronization Graph created in Phase 1, in this second phase TFluxCpp creates the output code for the input program. The code-segments added during this second phase include code: (1) that initializes TFlux execution (*TFlux Initializations*); (2) that creates the system's TFlux Kernel *Create TFlux Kernel*); (3) that defines the entry and exit points of DThreads (*Thread Header* and *Thread Footer*); (4) that performs the necessary calls the to TSU operations; (5) of the TFlux Kernels; and (6) of the Runtime Support.

Notice that adding these components into the application's code results in a binary that is self-contained. This means that the TFlux application can be executed on the host system without the need to install any patches to the Operating System or run specialized services.

Kyriakos Stavrou

All the code that is necessary to support the several features provided for the execution of TFlux Loops, such as scheduling (Section 4.3.2.1), unrolling (Section 4.3.2.2) and reduction (Section 4.3.2.3), is also generated by TFluxCpp in this second step. In particular, when the user defines the scheduling mode for TFlux Loops, TFluxCpp may need to insert special code at the header of L-DThreads in order to adjust the control variable. This is also true for the case of TFlux Loops the lower bound of which is different than zero. As for unrolling, the code of the loop body is replicated during this second preprocessing step and the control variable is again adjusted accordingly. As for the case of reduction, TFluxCpp inserts the proper code in reduction DThreads which have been created in the first step.

4.3 Basic TFlux directives

The main purpose of the TFlux directives is to allow the user to define the *boundaries*, the *type* and the *dependencies* among the application's DThreads. This Section presents the most important TFlux directives together with simple examples of their usage. The complete set of the TFlux directives is listed in Appendix B.

4.3.1 DThreads

A DThread is defined by enclosing its code in a set of ***#pragma ddm thread*** and a ***#pragma ddm endthread*** directives (Figure 17). These directives define the *begin* and *end* point of the DThread respectively. Moreover the ***#pragma ddm thread*** directive defines the unique identifier of the DThread (Thread Id) which in this example is equal to 4. The current version of TFlux applies a static scheduling technique, *i.e.* the DThreads each TFlux Kernel executes are defined statically. As such, TFluxCpp requires the user to define the Kernel that will execute this particular DThread (notice the ***Kernel 2*** part of the directive in the example of Figure 17).

Kyriakos Stavrou

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

Figure 17: Example of DThread declaration using TFlux directives.

4.3.1.1 Data Import/Export

TFluxCpp allows the user to define the data produced and consumed by a DThread. These definitions are achieved using the ***import*** and ***export*** parts of the ***#pragma ddm thread*** directive (Figure 18). TFluxCpp automatically creates a dependence between the DThread that produces a variable (*e.g export x*) and the DThread that consumes this variable (*e.g. import double 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 18: Example of DThread declaration using *import/export* statements.

4.3.1.2 Explicit Dependencies

The producer/consumer relationship between DThreads can also be defined explicitly using the ***depends*** statement of the ***#pragma ddm thread*** directive (Figure 19). 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.

```
#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 19: Example of DThread declaration using the *depends* statement.

Kyriakos Stavrou

4.3.1.3 All-Kernels DThreads

TFluxCpp allows its user to define DThreads that are to be executed by *all* Kernels, *i.e.* TFlux-Cpp will create multiple instances of this DThread and each such DThread will be assigned for execution to a different TFlux Kernel. This is achieved by replacing the ***kernel kernelID*** statement of the ***#pragma ddm thread*** directive with the ***kernel all*** statement (Figure 20). This feature is very helpful for DThreads 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 20: Example of DThread declaration using the *kernel all* statement.

4.3.2 TFlux Loops

TFluxCpp provides to its user multiple options for defining TFlux Loops (Section 3.2.2). The basic declaration of a TFlux Loop, which has always the form of a *for-loop*, is depicted in Figure 21. The code of this TFlux 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 *all* iterations of which can proceed in parallel (*DO-ALL* loops).

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

Figure 21: Example of a TFlux Loop.

Notice that TFlux Loops are always executed by *all* Kernels of the system with TFluxCpp distributing the iterations to the different Kernels as evenly as possible (Section 4.3.2.1). When

Kyriakos Stavrou

the number of loop iterations is not a multiple of the number of Kernels TFluxCpp distributes iterations to TFlux Kernels in such a way that the imbalance is never larger than one loop iteration as explained in the Section that follows.

4.3.2.1 Iterations Scheduling

TFluxCpp assigns loop iterations to Kernels using chunks the default size of which is 32 (this size is equal to the number of Ready Count counters per Synchronization Memory row) following a *Chunk Scheduling* [68, 130] scheme. As an example, if a system with two Kernels is executing a TFlux Loop with 256 iterations, TFlux Kernel 1 will be assigned iterations 0-31, 64-95, 128-159 and 192-223 whereas TFlux Kernel 2 iterations 32-63, 96-127, 160-191 and 224-255. The rationale behind this scheduling scheme is to allow the TFlux Kernels to better exploit temporal and spatial data locality. The reason for which TFluxCpp does not split the loops into parts of *consecutive* iterations and assign these parts to the Kernels (*i.e.* iterations 0-127 for Kernel 0 and 128-255 for Kernel 1) is to avoid load imbalance when the computational load of an iteration depends on the control variable. An example of this situation exists in the *CG* benchmark that is presented in Section 7.2.7.

In addition to the scheduling type described above, TFluxCpp supports *Round-Robin* scheduling [68, 130] *i.e.* assigning consecutive iterations to different Kernels. To select *Round-Robin* for a loop the programmer needs to use the ***schedule 1*** statement in the ***#pragma ddm for*** directive (Figure 22). On a 2-Kernels system, execution of a TFlux Loop with 256 iterations, Round-Robin scheduling would result in Kernel 1 executing iterations 0, 2, 4, ..., 254 and Kernel 2 executing the iterations 1, 3, 5, ..., 255. Notice that currently, Round-Robin Scheduling (schedule 1), supports only applying the round-robin iteration assignment on a *single*-iteration basis (this is why we use the number ***1*** in the ***schedule 1*** statement).

Kyriakos Stavrou

```
#pragma ddm for thread 4 schedule 1
  for(cv=0;cv<1024;cv++)
  {
    a[cv]=sin(cv);
  }
#pragma ddm endfor
```

Figure 22: Example of a TFlux Loop with Round-Robin scheduling.

Table 3 provides several examples for the explanation of these two scheduling policies. These examples assume a configuration with 4 TFlux Kernels.

Table 3: Examples of scheduling the iterations of TFlux Loops

Loop Iterations	**TFlux Kernel 1**	**TFlux Kernel 2**	**TFlux Kernel 3**	**TFlux Kernel 4**
CHUNK SCHEDULING (Default)				
8	0, 1	2, 3	4, 5	6, 7
13	0, 1, 2, 3	4, 5, 6	7, 8, 9	10, 11, 12
128	0-31	32-63	64-95	96-127
256	0-31 128-159	32-63 160-191	64-95 192-223	96-127 224-255
258	0-31 128-159 256	32-63 160-191 257	64-95 192-223	96-127 224-255
ROUND-ROBING SCHEDULING (schedule 1)				
8	0, 4	1, 5	2, 6	3, 7
13	0, 4, 8, 12	1, 5, 9	2, 6, 10	3, 7, 11
128	0, 4, 8, ..., 124	1, 5, 9, ..., 125	2, 6, 10, ..., 126	3, 7, 11, ..., 127
256	0, 4, 8, ..., 252	1, 5, 9, ..., 253	2, 6, 10, ..., 254	3, 7, 11, ..., 255
258	0, 4, 8, ..., 252, 256	1, 5, 9, ..., 253, 257	2, 6, 10, ..., 254	3, 7, 11, ..., 255

4.3.2.2 Unrolling

TFluxCpp provides automatic unrolling of the TFlux Loops [26, 92] which is achieved with the use of the ***unroll*** statement. For example the TFlux Loop depicted in Figure 23 is set to be unrolled 8 times. Unrolling the loop is often very helpful for TFlux as increasing the size of the L-DThreads helps to better amortize the parallelization overhead. TFluxCpp automatically handles all side-effects of unrolling, such as, replicating the code, adjusting the increase of the control variable and the corresponding change in value of the upper bound of the loop.

Kyriakos Stavrou

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

Figure 23: Example of a TFlux Loop with unrolling.

4.3.2.3 Reduction

A common operation performed by parallel loops is for all loop iterations to lead to a single value which is called “reduction”. TFluxCpp provides special support for such TFlux Loops, *i.e.* TFlux Loops that perform a reduction operation (Figure 24). This is achieved through the use of the ***reduction*** statement.

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

Figure 24: Example of a TFlux Loop with reduction.

Referring to the example depicted in Figure 24, each TFlux Kernel will calculate the sum of the iterations it executes on the *localSum* variable. To find the total sum, *i.e.* the sum 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 TFluxCpp.

In addition to the simple reduction operations (summation, subtraction and multiplication), TFluxCpp allows its user to *express* more complex reduction operations using custom functions. The only responsibility for the the user is to define this function; all other details are handled by TFluxCpp automatically. As an example, Figure 25 depicts a TFlux Loop that calculates the minimum and maximum value of an array of integer values.

Kyriakos Stavrou

```
#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 25: Example of a TFlux Loop with reduction with function.

4.3.2.4 TFlux Loop Dependencies

TFlux Loops can depend on other TFlux Loops and DThreads. As explained in Section 3.2.2.1 when a TFlux Loop depends on another TFlux Loop (Figure 26-(a)) no L-DThread of the second loop can start its execution unless *all* L-DThreads of the first loop have completed. Similarly, when a TFlux Loop depends on a DThread (Figure 26-(b)) no L-DThread of the loop can start its execution unless the DThread has completed. As for the situation where a DThread depends on a TFlux Loop, the DThread can start is execution only when *all* L-DThreads have completed. Notice that it is possible for a TFlux Loop to depend on multiple DThreads or TFlux Loops. All these dependencies can be expressed with very simple TFlux directives as depicted in Figure 26

4.3.2.5 Iteration Level Dependencies

TFluxCpp allows expressing dependencies at the iteration-level of loops, *i.e.* dependencies between L-DThreads (Section 3.2.2.1). As depicted in Figure 27, this is done by using the ***ilc***

Kyriakos Stavrou

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

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

(b)

Figure 26: Declaration of TFlux Loop dependencies. (a) Dependencies between TFlux Loops. (b) Dependency between a TFlux Loop and a DThread.

statement (Iteration Level Consumers). Each ***ilc*** statement is a six-tuple entity consisting of a type, the consumer TFlux 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 L-DThread based on the Iteration Id of the L-DThread that has completed according to the expressions which are detailed in Section B.1. An example of a program with Iteration level dependencies is depicted in Figure 27. Notice that for the second TFlux Loop of this program the Ready Count value has been set explicitly by the programmer to be equal to 2 as each L-DThread of this loop depends on two L-DThreads of the first TFlux 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 27: Example of a TFlux Loops with Iteration Level Dependencies.

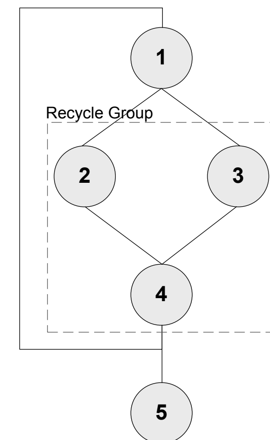
Kyriakos Stavrou

4.3.3 Thread Recycling

As explained in Section 3.2.3 *Thread Recycling* allows to a part of the application's Synchronization Graph to be executed multiple times. To express that a DThread is to be executed multiple times TFluxCpp provides the ***recycle*** statement. Figure 28 depicts an application that uses this feature. In this example DThreads 2, 3 and 4 belong to a *"recycle-group"* which is controlled by DThread 1. Based on a condition $(if((*x) > 8))$, DThread 1 "wakes-up" either its consumers inside the recycle-group (DThreads 2 and 3) or the Consumers of the DThreads of this group (DThread 5). In the first case, where DThread 1 "wakes-up" DThreads 2 and 3, upon completing their execution these DThreads will lead to DThread 4 becoming executable. When DThread 4 completes it will not "wake-up" DThread 5 but rather DThread 1. In the second case, where the condition evaluated by DThread 1 is not true any more, instead of "waking-up" DThreads 2 and 3, it will "wake-up" DThread 5.

```
#pragma ddm thread 1 kernel 1 recycle
  (*x)=(*x)+1;
  if((*x)>8)
  {
    #pragma ddm threadCompleted
  }
#pragma ddm recycle

#pragma ddm thread 2 kernel 1 depends (1) recycle
  (*y) +=(*x)*(*x);
#pragma ddm recycle

#pragma ddm thread 3 kernel 2 depends (1) recycle
  (*z) +=(*x)*(*x)*(*x);
#pragma ddm recycle

#pragma ddm thread 4 kernel 1 depends (2) recycle 1
  (*k)+=(*x)+(*y);
#pragma ddm recycle

#pragma ddm thread 5 kernel 2 depends (4)
  (*f)+=sin(k);
#pragma ddm endthread
```

(a)



(b)

Figure 28: Example of Thread Recycling. (a) The code of the application. (b) The Synchronization Graph of the application

Kyriakos Stavrou

4.4 TFlux directives Expressibility

After having presented the TFlux directives, this Section qualitatively examines their ability to express the parallelism inside an application (expressibility). Notice that the ability of the TFlux Platform to efficiently *execute* these parallel segments is not relevant for this discussion as what is evaluated here, is the ability to *express* and not the ability to *exploit* the concurrency between different code segments using the TFlux directives.

Using DThreads it is possible to express *all* concurrency that exists in an application due to the fact that TFlux directives allow *arbitrary* dependencies between DThreads. This feature is what allows to TFlux directives to have higher expressibility compared to widely used parallel *"traditional"* programming models, *i.e.* parallel programing models that have barriers and locks as synchronization primitives. Examples of such models are OpenMP [81] and MapReduce [30].

Figure 29 depicts an application for which the TFlux directives can express more parallelism compared to the "traditional" model. To parallelize this application using the "traditional" approach it is necessary to split the program into parallel execution phases and add barriers between them for synchronization. In particular, as can be seen from Figure 29, the first phase is composed of code segments 1, 2, 3 and 4, the second phase of code segments 5,6 and 7 and the last phase of segment 8 (the term *"code segment"* has the same conceptual meaning as the term *DThread*). Although this partitioning of code into phases guarantees correct execution as the data dependencies between the code segments (shown as the arcs of the Synchronization Graph) are satisfied, it introduces additional synchronization points. As can be seen from Figure 30-(a), that presents the application's code parallelized with OpenMP, two barriers have been introduced.

Kyriakos Stavrou

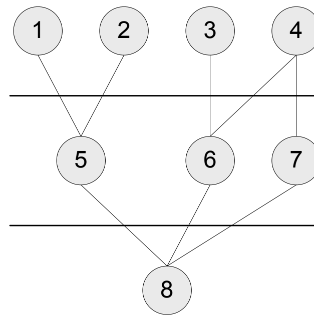


Figure 29: Example application. Nodes correspond to code segments, arcs to data dependencies and vertical lines to barriers.

This "traditional" parallelization approach provides a *partial* ordering of the application's Synchronization Graph. With TFlux directives however (Figure 30-(b)), it is possible to express the application with *minimum* ordering and consequently expose to the hardware the maximum amount of parallelism.

To better explain this we will refer to the execution condition of Code Segment 5. According to the partial ordering of the the "traditional" approach Code Segment 5 can be executed only after code segments 1, 2, 3 and 4 have completed. However, according to the minimum ordering expressed with the TFlux directives, Code Segment 5 can execute after code segments 1 and 2 complete without the need to also wait for the completion of Code Segments 3 and 4. If code segments 1 and 2 require 4 time-units each to execute whereas code segments 3 and 4 10 time-units, given enough computational resources, execution of code segment 5 could be overlapped by 6 time-units if parallelization was done using TFlux directives. However, this opportunity would be lost by parallelizing the code using the "traditional" model.

Another situation where TFlux directives allow expressing more parallelism, regards parallel loops with iteration level dependencies. An example of such a situation is presented by Figure 31. Notice that the data dependencies between the two loops are such that for an iteration of the second

Kyriakos Stavrou

```
#pragma omp parallel
{
  #pragma omp section
    code segment 1

  #pragma omp section
    code segment 2

  #pragma omp section
    code segment 3

  #pragma omp section
    code segment 4

  #pragma omp section
    code segment 5

  //BARRIER
  #pragma omp barrier

  #pragma omp section
    code segment 6

  #pragma omp section
    code segment 7

  #pragma omp section
    code segment 8

  //BARRIER
  #pragma omp barrier

  #pragma omp section
    code segment 9
}
```

(a)

```
#pragma ddm thread 1
  code segment 1
#pragma ddm endthread

#pragma ddm thread 2
  code segment 2
#pragma ddm endthread

#pragma ddm thread 3
  code segment 3
#pragma ddm endthread

#pragma ddm thread 4
  code segment 4
#pragma ddm endthread

#pragma ddm thread 5 depends(1,2)
    code segment 5
#pragma ddm endthread

#pragma ddm thread 6 depends(3,4)
    code segment 6
#pragma ddm endthread

#pragma ddm thread 7 depends(4)
    code segment 7
#pragma ddm endthread

#pragma ddm thread 8 depends(5,6,7)
    code segment 8
#pragma ddm endthread
```

(b)

Figure 30: (a) OpenMP code for the application depicted in Figure 29. (b) TFlux code for the application depicted in Figure 29.

loop to start its execution it is *not* necessary to wait for *all* iterations of the first loop to complete. For instance, iteration 0 of the second loop can start its execution after iterations 0 and 1 of the first loop have completed.

```
for(i=0;i<1024;i++)
{
  A[i]=i*i;
}

for(i=0;i<512;i++)
{
  B[i]=A[2*i+1]-A[2*i];
}
```

Figure 31: Example of a loops with dependencies at the iteration level.

Kyriakos Stavrou

The "traditional" parallelization approaches lack the expressibility to expose this type of parallelism to the hardware. As such, when such applications are to be parallelized using, for example OpenMP, a barrier point is implicitly imposed between the two loops (Figure 32-(a)). This leads to *no* iteration of the second loop getting executable unless *all* iterations of the first loop have completed and consequently to less exploitable parallelism.

As explained earlier in Section 4.3.2.5, the TFlux directives *are* able to expose this type of parallelism to the hardware. In particular, using the *ilc* statement in the directive describing the parallel loop it is possible to limit the dependencies of the iterations of the second loop to only those required by the true data dependencies (Figure 32-(b)).

```
#pragma omp for
  for(i=0;i<1024;i++)
  {
    A[i]=i*i;
  }

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

(a)

```
#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
```

(b)

Figure 32: (a) OpenMP code for the application depicted in Figure 31. (b) TFlux code for the application depicted in Figure 31.

4.5 Limitations

Although TFluxCpp provides to its user numerous features and options there is still room for improvement. In the paragraphs that follow we identify the two most important limitations of the tool.

The first limitation regards the fact that currently TFluxCpp supports directives only in the body of the *main()* function, *i.e.* the directives placed in other functions are not taken into account.

Kyriakos Stavrou

The consequence of this limitation is that to parallelize code in a function it is necessary to first inline its code into the body of *main()*.

The second limitation of TFluxCpp, is the lack of support for nested parallelism. This feature could be useful for nested loops. To provide nested parallelism it is necessary to first add the appropriate support in the Runtime Support in order to differentiate the level at which each code segment is executed. This support is also useful to allow execution of parallel sections into functions without the need to inline the code. Currently, to exploit the parallelism of nested parallel constructs the user needs to modify the code in order for all DThreads to be at the same level using techniques such as loop merging [92].

Kyriakos Stavrou

Chapter 5

TFluxHard

This Chapter focuses on the TFlux incarnation for which the Thread Scheduler as a hardware component. We call this design the ***TFluxHard*** system.

Section 5.1 presents a general description of TFluxHard followed by a detailed presentation of its Scheduler in Section 5.2 that focuses on the implementation of the Basic Operation. Section 5.3 presents the interface between the Scheduler and the TFlux Kernel whereas Section 5.4 discusses the several logic and memory units of the scheduler. Section 5.5 presents an analysis about the Scheduler's hardware budget and finally, Section 5.6 discusses implementation issues for TFluxHard.

5.1 The TFluxHard System

TFluxHard follows the layered design of TFlux and its major characteristic is the implementation of the Scheduler, which functionality is provided at the hardware level by a dedicated module.

Kyriakos Stavrou

As for the other TFlux layers, *i.e.* the programming layer, the Runtime Support and the TFlux Kernels, they are as previously presented for the TFlux Platform.

Although TFluxHard can not be directly applicable to an off-the-shelf multicore system, the fact that it uses only *commodity* components (unmodified OS, compiler, CPUs, caches) allows it to be easily adopted by future systems. In particular, for a multicore to become a TFluxHard machine the only requirement is the augmentation of the machine with a hardware module providing the functionality of the Scheduler. As will be explained in more detail in Section 5.2, this module can be attached to the system network without interfering with any other component of the system. It is relevant to stress that due to the design of TFlux, no changes are required to the original ISA in order for TFlux to schedule the Threads in a dataflow-like way. A general, abstract TFluxHard configuration equipped with 4 CPUs is depicted in Figure 33.

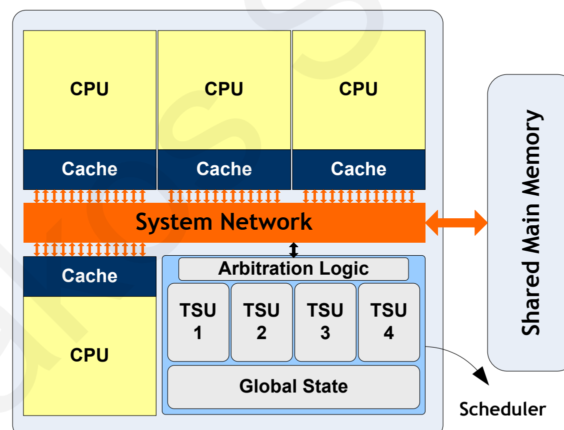


Figure 33: Abstract TFluxHard configuration with 4 cores.

5.2 TFluxHard Scheduler

This Section presents the *purpose* of the different logic and memory units (Section 5.2.1) of the TFluxHard Scheduler as well as the implementation of the Basic Operations (Section 5.2.2).

5.2.1 TFluxHard Scheduler Units

As can be seen from Figure 34 that depicts the TFluxHard Scheduler, similar to the generic TFlux Scheduler, it consist of as many TSUs as the number of TFlux Kernels it is able to serve and a number of shared units. The discussion that follows presents all memory and logic units of the TFluxHard Scheduler grouped according to their operation.

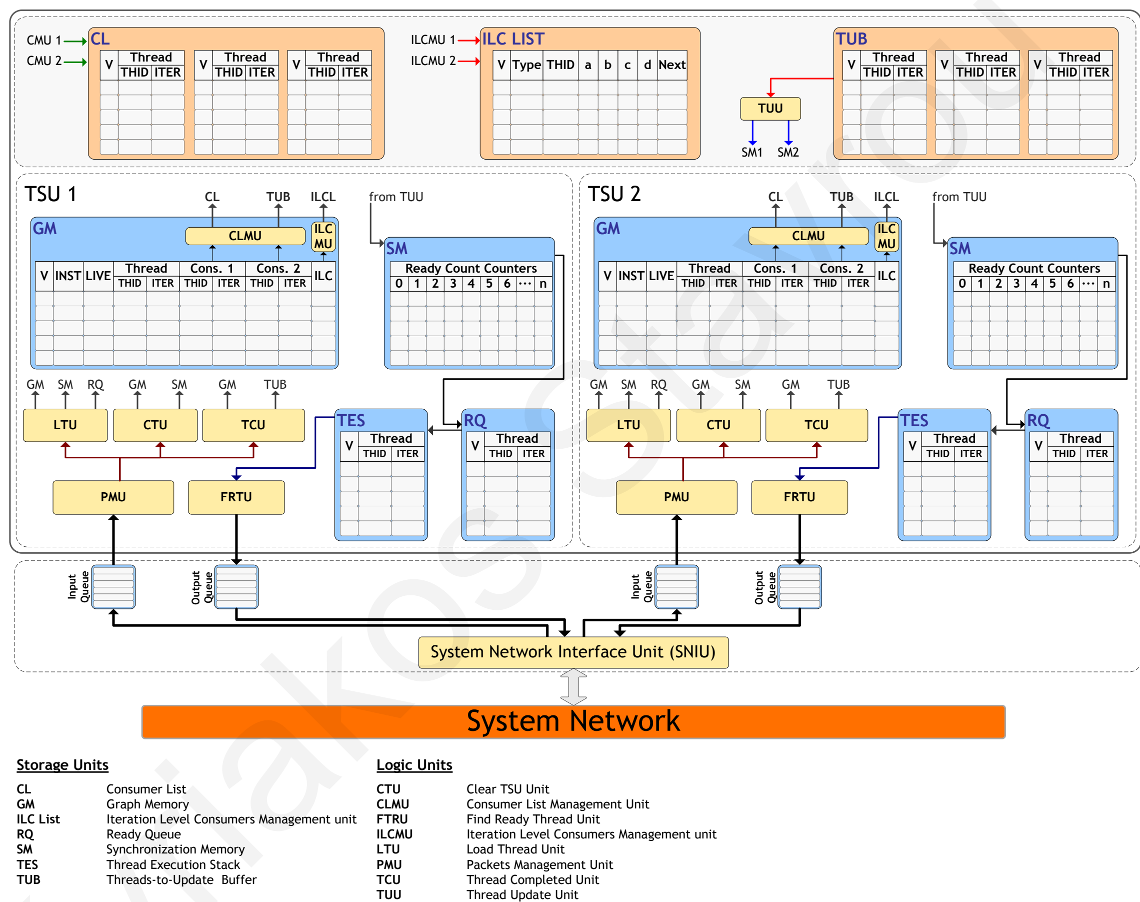


Figure 34: TFluxHard Scheduler.

Kyriakos Stavrou

5.2.1.1 Communication Units

The TFluxHard Scheduler is accessible as a memory mapped device. This simplifies the communication between the TFlux Kernel and the TFluxHard Scheduler as it can be achieved through

simple *load* and *store* operations. Notice that in high level languages, such as C, accessing devices attached to the memory mapped addresses is done through the *read()* and *write()* functions that operate at the user level, therefore incurring in a minimum overhead.

The purpose of the ***System Network Interface Unit (SNIU)*** is to serve as the interface between the Scheduler and the System's Network. In particular, this unit receives all data packets sent to the Scheduler and also handles all the data sent by the Scheduler to the CPUs. Notice that each TSU is mapped to a different address; it is a responsibility of the SNIU to receive the data packets for all the TSUs and distribute them appropriately.

As the rate by which data is produced and consumed by the Scheduler and the System's Network may be different, to avoid loss of data, the Scheduler is equipped with two queues per TSU. The ***Input Queue (InQ)*** buffers all data sent by the TSU to the System's Network until the later is able to transmit them whereas the ***Output Queue (OutQ)*** buffers all data received from the System's Network until the TSU is able to process them.

5.2.1.2 Units for the implementation of the Basic Operations

The interface between the Scheduler and the CPU, which will be presented in detail in Section 5.3, is through specially coded data packets that represent *Scheduler Instructions*. The purpose of the ***Packets Management Unit (PMU)*** is to decode these Scheduler Instructions and trigger the appropriate Basic Operation. The TSU is equipped with one logic unit for each such operation.

The ***Load Thread Unit (LTU)*** performs the *Load Thread Operation* (Section 3.3.1), *i.e.* it loads the metadata of the DThreads sent by the CPU to the TSU's internal structures. Whenever a DThread completes, the TFlux Kernel triggers the *Thread Completed Operation* (Section 3.3.2), which functionality is provided by the ***Thread Completed Unit (TCU)***. As for the *Clear TSU* operation (Section 3.3.5), which is executed at the end of each Block to deallocate the resources

used for the execution of the Block's DThreads, it is provided by the ***Clear TSU Unit (CTU)***. The ***Find Ready Thread Unit (RFTU)*** is responsible for returning to the CPU the metadata of a Ready DThread whenever the CPUs perform the *Find Ready Thread* operation (Section 3.3.4). Finally, the ***Thread Update Unit (TUU)*** is responsible for decreasing the Ready Count counters of the consumers of the completed DThreads by executing the *Thread Update* operation (Section 3.3.3).

5.2.1.3 Units for DThread Metadata Storage

The metadata of the application's DThreads is stored in the ***Graph Memory (GM)*** and ***Synchronization Memory (SM)*** unit. More specifically the static metadata, *i.e.* the one that does not change during the execution of the application, is stored in the *Graph Memory* whereas the *Ready Count*, that is changed dynamically, is stored in the *Synchronization Memory* (dynamic metadata).

As explained earlier the *Graph Memory* provides storing for only two Consumers per row, *i.e.* it supports only two Consumers per DThread. However, it is possible for a DThread to have more than two Consumers. To support such DThreads the Scheduler has a special unit named ***Consumer List (CL)***. The Thread Templates of the Consumers of the these DThreads are stored in the *Consumer List* which is indexed by the *Consumer 2* field of the *Graph Memory* (further details are given in Section 5.2.2.1). The management of the Consumers of a DThread is done with the help of the ***Consumer List Management Unit (CLMU)***. A similar approach is followed for the Iteration Level Consumers. In particular, their information is stored in the ***Iteration Level Consumers List (ILCL)*** which is indexed by the *ILC* field of the *Graph Memory*. For handling the Iteration Level Consumers the Scheduler uses the ***Iteration Level Consumers List Management Unit (ILCLMU)***.

Kyriakos Stavrou

5.2.1.4 Units for Handling Ready Threads

Whenever the Ready Count counter of a DThread reaches zero, representing the fact that the DThread is ready to execute, its Thread Template is copied to the ***Ready Queue (RQ)*** unit. As for the DThreads that have been scheduled for execution, their templates are moved from the *Ready Queue* to the ***Thread Execution Stack (TES)***.

5.2.1.5 Units for Thread Updating

For the Ready Count of Consumers of the completed DThreads to be decreased the Scheduler utilizes the ***Threads-to-Update Buffer (TUB)***. More specifically, whenever a DThread completes, the *Thread Completed Unit* copies its consumers to the *Threads-to-Update Buffer*. The *Thread Update Unit* reads the entries of this unit and decreases their Ready Count counters leading to new Threads being deemed executable.

5.2.2 Implementation of Basic Operations

This Section presents the details of the implementation of the Basic Operations. For each such operation, we discuss the interaction between the different memory and logic units involved.

5.2.2.1 Thread Load

When all the data for a DThread have been received by the Packet Management Unit (PMU), the Load Thread Unit (LTU) stores its metadata onto the Graph Memory (GM) and Synchronization Memory (SM) structures. The first step towards this operation is to find an *empty* entry into the Graph Memory where the Thread Templates of the DThread, of its Consumers and of its Iteration Level Consumers are to be stored. At the same time the corresponding Ready Count counters in the Synchronization Memory are initialized. Notice that each SM row corresponds to one GM

Kyriakos Stavrou

row; as such if an available GM entry has been identified it is not necessary to search for an available SM entry. If the DThread has more than one instance, *i.e.* the DThread executes a TFlux loop, then multiple Ready Count counters in the particular Synchronization Memory row are used.

As an example of Thread Loading refer to Figure 35 which depicts the state of the Graph and Synchronization Memories after loading DThreads 1/0 and 2/32. DThread 1/0 has 1 instance, 2 Consumers (2/0 and 2/32) and initial Ready Count value equal to 3 whereas DThread 2/32 has 4 instances with and initial Ready Count equal to 1 and 1 consumer (4/0). As can be seen from Figure 35 the number of Ready Count counters used in the SM is equal to the number on DThread instances, 1 for 1/0 and 4 for 2/32. Moreover, notice that for DThreads with multiple instances (*e.g.* 2/32), all Ready Count counters are initialized to the same value.

Graph Memory								
V	**INST**	**Thread Template**		**Consumer 1**		**Consumer 2**		**ILC**
		THID	**ITER**	**THID**	**ITER**	**THID**	**ITER**	
1	*1*	*1/0*		*2/0*		*2/32*		*0*
1	*4*	*2/32*		*4/0*		*0/0*		*0*

Synchronization Memory								
Ready Count Counters								
0	**1**	**2**	**3**	**4**	**5**	**6**	**...**	**31**
3								
1	*1*	*1*	*1*					

Figure 35: Example of Thread Loading

For DThreads with at most two Consumers, the Scheduler uses the the Graph Memory for storing their Thread Templates whereas for DThreads with more than two Consumers it uses the Consumer List. For this situation where there are more that 2 Consumers, the *Consumer 1* field of the Graph Memory row is set to 0/0 whereas the Consumer 2 field to x/0 where x is the position into the Consumer List where the *first* Consumer of this DThread is stored (if the DThread had only one Consumer then the *Consumer 1* field would hold its Thread Template whereas the *Consumer 2* field would be equal to 0/0).

Kyriakos Stavrou

To find the *last* consumer of each DThread and to avoid segmentation the Consumer List uses a field named *"next"*. For each Consumer List entry, this field is an index to the next consumer of the same DThread. For the last consumer however, the *next* field indexes itself.

To better explain the way consumers are handled assume that the DThreads shown in Table 4 are to be loaded onto the same TSU.

Table 4: Consumer List usage example

DThread	***Consumers***
1/0	-
1/32	2/0
2/0	4/0, 4/32
3/0	5/0, 6/0, 7/0
4/0	5/32, 6/32, 7/0, 7/32

The state of the Consumer List *before* loading these DThreads is depicted in Figure 36-(a) whereas the state of the Graph Memory and Consumer List *after* loading these DThreads by Figure 36-(b). DThread 1/0 has no consumers so in its Graph Memory both *Consumer 1* and *Consumer 2* fields are equal to 0/0. As for DThread 1/32, it has only one Consumer. Therefore, in the *Consumer 1* field we store the Thread Template of the single Consumer whereas the *Consumer 2* field is equal to 0/0. As for DThread 2/0, it has two Consumers. As such, the Thread Template of the first Consumer in the *Consumer 1* field and the Thread Template of the second Consumer is stored in the *Consumer 2* field. Regarding DThread 3/0 it has 3 consumers. In this case, the *Consumer 1* field is set to 0/0 and the *Consumer 2* field is set equal to the number of the first available entry in the Consumer List. Assume that the Consumer List at this point has the state depicted in Figure 36-(a) with the first available entry being the one with number 4. This means that for DThread 3/0 the *Consumer 1* field will be equal to 0/0 whereas field *Consumer 2* equal to 4/0. The first consumer of DThread 3/0 will be stored in entry 4 and its *next* field will index the next Consumer (6/0) which will be stored in entry 5. This, in turn, will index the next Consumer

Kyriakos Stavrou

(7/0) stored in the next available free entry which is the 9th one. As this is the last consumer of DThread 3/0 it will index itself, *i.e.* its *next* field will be equal to 9. With the same rationale the consumers of DThread 4/0 are stored as depicted in Figure 36-(b).

Consumer List *before* insertion — Consumer List (CL)

CL Entry	V	Thread Template THID/ITER	NEXT
1	0	***11/0***	***2***
2	1	***11/32***	***3***
3	1	***13/0***	***3***
4	0	-	-
5	0	-	-
6	1	***8/0***	***7***
7	1	***8/32***	***8***
8	0	***9/0***	***8***
9	0	-	-
10	0	-	-
11	0	-	-
12	0	-	-
13	0	-	-
14	0	-	-
15	0	-	-

(a)

Graph Memory and Consumer List after insertion — Graph Memory (GM)

V	INST	Thread Template THID/ITER	Consumer 1 THID/ITER	Consumer 2 THID/ITER	ILC
1	1	1/0	0/0	0/0	0
1	1	1/32	2/0	0/0	0
1	1	2/0	4/0	4/32	0
1	1	3/0	0/0	1/0	0
1	1	4/0	0/0	4/0	0
0	…	…	…	…	…

Consumer List (CL)

CL Entry	V	Thread Template THID/ITER	NEXT
1	1	***11/0***	***2***
2	1	***11/32***	***3***
3	1	***13/0***	***3***
4	1	5/0	5
5	1	6/0	9
6	1	***8/0***	***7***
7	1	***8/32***	***8***
8	1	***9/0***	***8***
9	1	7/0	9
10	1	5/32	11
11	1	6/32	12
12	1	7/0	13
13	1	7/32	13
14	0	-	-
15	0	-	-

(b)

Figure 36: Consumer List usage example.

For DThreads that have Iteration Level Consumers (ILC), their information is stored in the *Iteration Level Consumers List* (ILCL) with a rationale similar to that used for the *Consumers List*. However, as it is only a certain category of the L-DThreads that has iteration level consumers, for better TSU space utilization, the metadata of the ILC are always stored in the ILC list which is indexed by the *ILC* field of the Graph Memory. If a DThread does not have ILCs, then this field is equal to zero.

Finally, notice that the DThreads that are loaded with initial Ready Count equal to zero are also inserted into the Ready Queue by the *Thread Load* operation. Examples of such DThreads are the *Inlet* DThreads of the first Block.

Kyriakos Stavrou

5.2.2.2 Thread Completed Execution

When a DThread completes its execution the corresponding Kernel notifies its TSU which triggers the *Thread Completed Execution* operation. As explained in Section 3.3.2 this operation has three different versions: (1) *Execution Completion* which applies for DThreads that have completed their execution and are not to be re-instantiated in any way; (2) *L-DThread Recycle* which regards the L-DThreads that recycle themselves to execute a new loop iteration and (3) *Thread Recycle Execution* that is for DThreads that belong in a recycle-group and will be reinvoked in the future.

For *Execution Completion* the Scheduler performs two actions. The first is to remove the currently executed thread from the Thread Execution Stack (TES) (recall that the Thread Template of the currently executed thread is always the one at the top of the TES). The second step for this operation is to insert the Consumers of the completed DThread into the *Threads-to-Update Buffer* (TUB). For this step the *Thread Completed Unit (TCU)*, uses the Thread Template that was removed from the TES, to search into the Graph Memory until the entry corresponding to the completed DThread is found. Then, with the help of the Consumer List Management Unit (CLMU), it identifies the consumers of the completed DThread and inserts their Thread Templates into the TUB.

For the *L-DThread Recycle* operation, the TCU, in addition to the actions described for *Execution Completion*, it “recycles” the completed L-DThread to allow it to execute a new iteration of the TFlux Loop. For this it is necessary to re-insert the metadata of the new instance of this L-DThread into the Graph Memory (GM) and Synchronization Memory (SM) structures. As already explained in Section 3.2.2.2, there are two possibilities for this insertion. The first is that a *suitable* GM entry already exists (a detailed explanation of the suitability of GM entries has been

Kyriakos Stavrou

given in Section 3.2.2.3) whereas the second regards the allocation of a *new* GM entry. As such, the TCU first searches the GM for such a suitable entry and if such an entry is found uses it for the new instance of the completed DThread. In case an appropriate row does not exist, the TCU allocates a new GM row for the insertion of the new instance.

Finally, for the *Thread Recycle Execution* operation the steps followed are identical to those regarding the *L-DThread Recycle* operation with the only difference regarding the Thread Template of the new instance. In particular, in contrast to the *L-DThread Recycle* operation where the new Iteration Id of the new instance is calculated as an offset to the Iteration Id of the completed DThread, for *Thread Recycle Execution* the new Iteration Id is the one given by the TFlux Kernel as a parameter of the operation invocation.

5.2.2.3 Thread Update

The purpose *Thread Update* operation is to decrease the Ready Count counters of the DThreads the Thread Templates of which are in the Threads-to-Update Buffer (TUB). Moreover, when the Ready Count counter of a DThread becomes zero, the *Thread Update Unit (TUU)* copies its Thread Template to the Ready Queue unit of the corresponding TSU.

Whenever the TUU detects a valid entry in the TUB, *i.e.* a Thread update-request, it tries to serve it. To achieve this the *TUU* searches the different Graph Memory (GM) units in order to identify where the metadata of this DThread is stored. Finding this "host" GM entry leads also to identifying this DThread's Ready Count counter due to the one-to-one correlation between the GM and SM rows. The TUU then decreases this counter and if its value reaches zero, it copies the metadata of the completed DThread into that TSU's Ready Queue structure.

Notice that it is possible for an update-request to refer to a DThread that does not exist in any of the Scheduler's GM unit. This would be the case for an L-DThread that has as a Consumer an

Kyriakos Stavrou

L-DThread that is to be created in a later phase when some other L-DThread recycles itself (Section 3.2.2.3). For such situations, the TUU will not be able to find the GM row that corresponds to the particular DThread. When this situation applies, the update-request remains in the TUB and the TUU will try to serve it again during the next Thread Update cycle.

5.2.2.4 Clear TSU

The purpose of the *Clear TSU* operation is to deallocate all resources used for the execution of the Block's DThreads. To complete this operation the *Clear TSU Unit (CTU)* flushes all state of the Graph Memory (GM) and the Synchronization Memory (SM) units. Notice that whenever a GM entry is invalidated this leads to the invalidation of the corresponding Consumer List and Iteration Level Consumers list entries (if any). The Ready Queue (RQ) and Thread Execution Stack (TES) do not need to be flushed as their contents are cleared dynamically when a ready DThread is requested by the CPU. The same applies for the Threads-to-Update Buffer (TUB) as whenever an update-request is serviced the corresponding entry is invalidated from the TUB.

5.2.2.5 Find Ready Thread

To find the next DThread to execute, the CPU invokes the Find Ready Thread operation, which returns to the CPU the template of an executable DThread, *i.e.* of a DThread with Ready Count equal to 0. When this operation is invoked, the Find Ready Thread Unit (FRTU) accesses the Thread Execution Stack (TES) and if it *is not* empty, it returns to the CPU the Thread Template at the top of the TES. If the TES *is* empty, the FRTU first copies all the DThreads from the Ready Queue (RQ) to the TES. If both the TES and the RQ are empty this means that no ready DThread exists for this particular TSU and the operation completes by sending to the CPU the template 0/0 which forces the TFlux Kernel to request again for a new ready DThread.

Kyriakos Stavrou

5.3 TFluxHard Scheduler Interface

The CPU controls the TSU through specially coded data packets that encode Scheduler Instructions. In addition to defining the operation to be performed, these Instructions contain also the necessary parameters. In this Section, we describe the interface of the Scheduler for the different operations it supports. Notice that each *Instruction* consists of several packets while each packet is 32-bit long. A detailed example of the TFluxHard Scheduler interface is given in Appendix A.

5.3.1 Load TSU

With this operation a TSU is loaded with the metadata of *one or more* DThreads. As can be seen from Figure 37 that depicts the Scheduler's interface for the Load TSU operation, the first packet defines that a load operation is to be performed (Byte 3 is equal to 0) and also defines the number of threads that will be loaded (Byte 0).

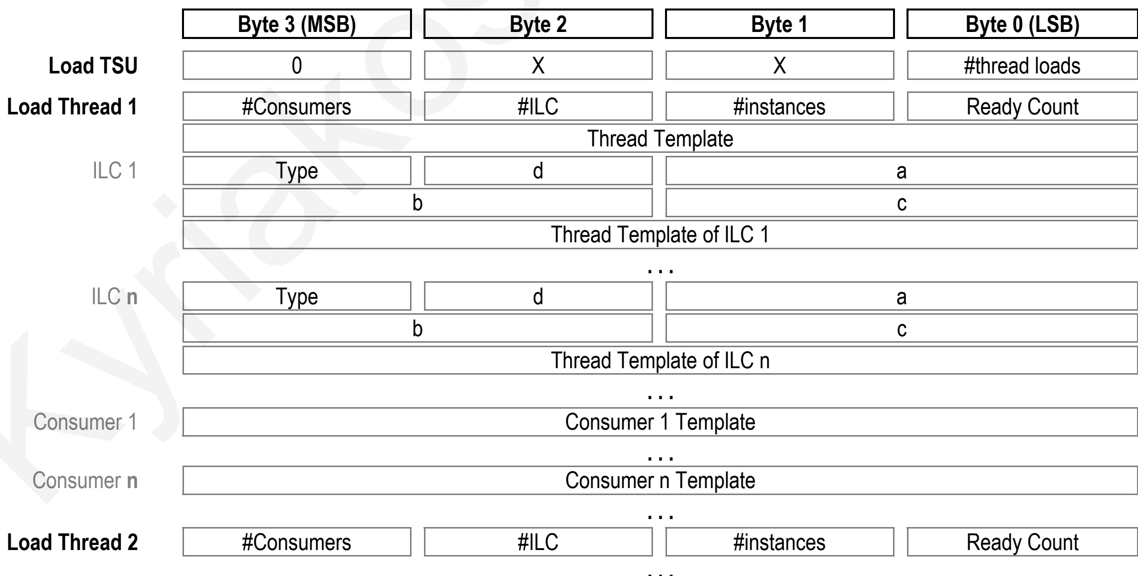


Figure 37: TFluxHard Scheduler Interface for *Load TSU*.

Kyriakos Stavrou

For each DThread that will be loaded onto the TSU, a sequence of packets must be sent. For the first packet, Byte 3 is equal to the number of consumers for this DThread, Byte 2 to the number of Iteration Level Consumers (ILC), Byte 1 to the number of Instances of the DThread being loaded and finally Byte 0 to its initial Ready Count value. The second packet keeps the Thread Template of the loaded DThread.

The set of packets that follow regard the Iteration Level Consumers and there are 3 packets per such consumer. The first of these 3 packets contains the type for this Iteration Level Consumer (Byte 3), its d parameter (Byte 2) and its a parameter (Byte 1 and Byte 0). As for the b parameter it is contained in Bytes 3 and 2 of the second packet whereas parameter c in Bytes 1 and 0. Finally, the third packet contains the Thread Template of the iteration level consumer.

After the packets regarding all Iteration Level Consumers have been sent, the packets that follow regard the Consumers of the DThread that is being loaded. For each consumer the TSU receives one packet which represents its Thread Template.

What follows are the packets describing the subsequent DThread, if such a DThread exist. Recall the first packet of the Thread Load operation defined the number of DThreads to be loaded.

5.3.2 Clear this TSU

In order to perform the Clear TSU operation, a packet with Byte 3 equal to 1 and Byte 0 equal to 0 must be sent to that TSU. The interface for this operation is depicted in Figure 38.

	Byte 3 (MSB)	Byte 2	Byte 1	Byte 0 (LSB)
Clear this TSU	1	X	X	0

Figure 38: TFluxHard Scheduler Interface for *Clear TSU*.

Kyriakos Stavrou

5.3.3 Flush Scheduler

When a program starts, all the state of the scheduler is flushed. To perform this operation, as depicted in Figure 39 that regards the interface for *Flush Scheduler*, a packet with Byte 1 and Byte 0 equal to 1 must be sent to *each* TSU.

	Byte 3 (MSB)	Byte 2	Byte 1	Byte 0 (LSB)
Flush Scheduler	1	X	X	1

Figure 39: TFluxHard Scheduler Interface for *Flush Scheduler*.

Kyriakos Stavrou

5.3.4 Thread Completed Execution

5.3.4.1 Execution Completion

The interface for the Execution Completion operation is depicted in Figure 40. This Scheduler Instruction consists of only one packet for which Byte 3 must be equal to 2. As for Byte 1 and Byte 0 they represent *C1 Iters* and *C2 Iters* which force the TSU to insert into the TUB additional update-requests. This feature is very helpful for the efficient execution of TFlux loops and allows loading less consumers for each DThread.

More specifically, if the thread template of Consumer 1 is equal to *T/C/I* then the TSU inserts into the TUB update-requests for the DThreads *T/C/I*, *T/C/I+1*, *T/C/I+2*, ..., *T/C/I+C1_Iters*. With a similar rational *C2 Iters* defines the update-requests to be inserted into the TUB for the second consumer.

	Byte 3 (MSB)	Byte 2	Byte 1	Byte 0 (LSB)
Execution Completion	2	X	C1 Iters	C2 Iters

Figure 40: TFluxHard Scheduler Interface for *Execution Completion*.

5.3.4.2 L-DThread Recycle

As can be seen from Figure 41, for the L-DThread recycle operation, the first packet is identical to that regarding the *Execution Completion* operation. As for the second packet, Byte 2 defines the Ready Count value for the new instance of the DThread whereas Bytes 1 and 0 the *offset* to the Iteration Id. More specifically, if the DThread for which this operation was performed had Thread Template *T/C/I*, the Thread Template for the new instance will be *T/C/I+offset*.

	Byte 3 (MSB)	Byte 2	Byte 1	Byte 0 (LSB)
L-DThread Recycle	3	CITD1 - CITD2	C1 Iters	C2 Iters
	X	New Ready Count	Iteration Offset	

Figure 41: TFluxHard Scheduler Interface for *L-DThread Recycle*.

5.3.4.3 Recycle Execution

The only difference between the interface of this operation and the interface of the *L-DThread Recycle* operation regards the fact that for *Recycle Execution* operation the new Iteration Id is given explicitly as a parameter of the Scheduler Instruction. As can be seen from Figure 42 that summarizes the interface of this operation, the new Iteration Id is given by Bytes 1 and 0 of the second packet.

	Byte 3 (MSB)	Byte 2	Byte 1	Byte 0 (LSB)
Recycle Execution	4	CITD1 - CITD2	C1 Iters	C2 Iters
	X	New Ready Count	New Iteration Id	

Figure 42: TFluxHard Scheduler Interface for *Recycle Execution*.

Kyriakos Stavrou

5.4 TFluxHard Scheduler Hardware

In this Section we discuss the hardware complexity of the Scheduler's memory and logic units and their criticality in terms of timing for the implementation of the TFlux Basic Operations.

5.4.1 Logic Units

5.4.1.1 Packet Management Unit (PMU)

The Packet Management Unit (PMU) decodes the packets sent from the CPU and triggers the appropriate operation. As most operations require multiple packets, the PMU first copies all the packets that have been sent for a particular operation in a buffer prior to triggering the circuit that will execute the operation. The logic of the PMU is very simple as its functionality can be implemented with a simple FSM.

5.4.1.2 Load Thread Unit

The *Load Thread Unit* (LTU) receives the data packets describing the DThread to be loaded from the *Packets Management Unit* (PMU), decodes them and inserts their metadata in the appropriate TSU's structures. As explained earlier in Section 5.3.1, it is very simple to extract the metadata of the DThread to be loaded onto the TSU from the corresponding data packets. As such, this decoding phase can be implemented by a simple FSM.

The next step is to find an available entry in the Graph Memory. As the number of entries for this unit is small (64 entries) this operation is also trivial. The same applies for writing the DThread's metadata in the GM and SM structures. As for the process of writing the Consumers and Iteration Level Consumers in the Consumer List and ILC List respectively, the LTU uses the Consumer List Management Unit (CLMU) Iteration Level Consumer List Management Unit (ILCLMU) which are presented later in this Section.

Kyriakos Stavrou

Finally, notice that the delay of the Thread Load operation has negligible effect on the overall performance of TFluxHard. This is due to the very few times this operation is executed (as many times as the number of Blocks).

5.4.1.3 Thread Completed Unit

The first step for the *Thread Completion Operation* is to remove the top entry of the Thread Execution Stack (TES) which is trivial. The same applies for identifying the Graph Memory (GM) entry that stores the completed DThread due to the small size of this unit and the efficiency of the hardware for such searches. The phase of this operation that is likely to have the longer delay is that of the identification and insertion of the consumers into the Threads-to-Update Buffer (TUB). This delay is likely to be longer for the Iteration Level Consumers as to define the update-request it is required to perform some calculations that for some of the ILC types may be complex.

This operation is not in the critical path. After sending the command for the Thread Completion operation the TFlux Kernel will request for a new ready DThread and will proceed to its execution. In parallel the Thread Completed Unit will continue its operation.

5.4.1.4 Find Ready Thread Unit

The operations performed by the *Find Ready Thread Unit* (FRTU) are very simple for both possible paths. For the first path, Find Ready Thread completes just by writing to the *System Network Interface Unit* the 32-bit value of the top entry of the Thread Execution Stack (TES). This is a trivial operation as it only involves accessing a specific memory location. For the second path, the FRTU also needs to copy all the data from the Ready Queue (RQ) to the TES prior to returning the data of the top TES entry. Given the small size of the RQ this operation is also simple.

Kyriakos Stavrou

The Find Ready Thread operation is critical for the performance of TFluxHard as its delay can not be hidden. This is due to the fact that the CPU remains idle until a new DThread is identified. However, the FRTU executes very simple operations and consequently, its delay is expected to be very small.

5.4.1.5 Thread Update Unit

The Thread Update operation traverses the Threads-to-Update Buffer (TUB) and for each update-request it finds and decreases the corresponding Ready Count counter. Identifying the Graph Memory (GM) row that holds the metadata of the DThread the Ready Count counter of which is to be decreased, involves searching the different GM units for a match. This search process might be lengthy for configurations with large number of such units. To avoid this problem the operations to search the Graph Memory units progress in parallel. An alternative technique would be to use indexing for easier identification of the host GM unit (Section 6.2.1). Given the host GM entry the identification of the corresponding Ready Count counter in the Synchronization Memory (SM) is trivial. In particular, this is the ith counter where i is equal to the difference of the Iteration Id of the DThread being serviced minus the Iteration Id of the host GM entry. As such the *Thread Update Unit*, although loaded with a lot of work, can be implemented efficiently.

Although the Thread Update operation is not in the critical path, the more efficient it is the faster new DThreads will be deemed executable. This is of significant importance for applications with a large number of DThreads that have complex dependencies between them. An example of such application is the LU benchmark (Section 7.2.8).

5.4.1.6 Clear TSU Unit

The *Clear TSU Unit* flushes the data of the Graph Memory (GM) and Synchronization Memory (SM) as well as the corresponding entries for the contained DThreads from the Consumer List

(CL) and ILC List (ILCL) units. The costly part of this operation is the identification of the corresponding CL and ILCL entries which involves traversing this structures. However, as the Clear TSU operation is executed very rarely in TFlux programs (only 1 time for most applications), even if the delay operation is not very small, the impact on the overall performance will be negligible.

5.4.1.7 Consumer Management Unit

The purpose of the Consumer Management Unit (CMU) is to manage the consumers stored in the *Consumer List* which regard the DThreads that have more than two Consumers. In particular, the CMU finds all the consumers of a DThread given the values of fields *Consumer 1* and *Consumer 2* of the Graph Memory (GM). As explained in Section 5.2.2.1, finding the Consumers belonging to the same DThread given the position of the first one in the Consumer List is a simple process.

Another parameter that has to be handled by the CMU is the fact that the Consumer List is shared among all TSUs of the system. As such, some arbitration logic will be necessary when writing to the Consumer List. However, as delay of the read/write operations performed for the Consumer List is small any congestion caused will have negligible effect.

5.4.1.8 Iteration Level Consumers Management Unit

The requirements of the Iteration Level Consumers Management Unit (ILCMU) are the same as for the Consumer Management Unit (CMU) as the operations performed by the two units are identical. As such, the hardware specifications for the two units are the same.

5.4.1.9 System Network Interface Unit (SNIU)

The purpose of the System Network Interface Unit (SNIU) is to transfer the data from the Scheduler to the System's Interconnection Network and vise versa. This unit monitors the address

Kyriakos Stavrou

bus of the network and whenever a value is sent to the addresses that correspond to the Scheduler it copies this value to the Input/Output Queue of the corresponding TSU. Similarly, when a TSU is to send a value, it sends it to the SNIU which in turn, upon granted permission from the arbiter, transmits it to the system network. The SNIU is a simple device that consists of a number of 3-state buffers to interact with the system network and some logic to identify the TSU to which it will forward the packets it receives from the network.

5.4.2 Memory Units

5.4.2.1 Thread Execution Stack (TES)

The Thread Execution Stack (TES) is accessed as a *stack* structure. In particular, when a DThread is scheduled for execution its Thread Template is written at the top of this stack. In addition, when a ready thread is returned to the CPU, the Thread Template that is returned is that of the top element. As such, the TES can be implemented as a direct mapped memory. Moreover, as it is not concurrently accessed by multiple units, it can have only one read and one write port.

5.4.2.2 Ready Queue (RQ)

The Ready Queue (RQ) structure holds the Thread Templates of the DThreads that have been deemed executable, *i.e.* the DThreads with Ready Count equal to zero. This structure is written by the Thread Update Unit (TUU) during the Thread Update operation. In particular, the TUU decreases the Ready Count of the Consumer DThreads and when it identifies that this value has reached zero it copies its template of that particular DThread into the RQ. As for reading, this is done during the Find Ready Thread operation. As no searching is required for this structure it can be implemented as a direct mapped memory. Moreover, as writing is done by the Thread Update Unit (TUU) only and reading only by the Find Ready Thread Unit (FRTU) it can have only one read and one write port.

Kyriakos Stavrou

5.4.2.3 Synchronization Memory (SM)

The Synchronization Memory (SM) holds the Ready Count counters of the DThreads loaded onto the TSU. Each row of this unit has multiple counters to allow efficient execution of TFlux loops. In the current version of the TFluxHard Scheduler, the Graph and Synchronization Memory units have the same number of rows and the later is indexed by the former. As for the Ready Count counter accessed in a particular row of the Synchronization Memory this is defined by the Iteration Identifier (ITER) of the DThread. As accessing the Synchronization Memory does not require searching this unit can be implemented as a direct mapped memory structure. SM is written by the Thread Update Unit during the Thread Update operation and by the Thread Completed Execution when a DThread is to "recycle" itself. As such, two write ports and one read port are enough.

5.4.2.4 Graph Memory (GM)

The Graph Memory (GM) unit holds the metadata of the application's DThreads. According to our previous description of the logic units, it is possible to see that it is often necessary to identify the GM entry in which a particular Thread Template is stored. As such, the part of the Graph Memory containing the Valid, Thread Id (THID) and Iteration Id (ITER) fields is implemented as a Content Addressable Memory (CAM). As for the other fields of the Graph Memory they can be implemented as a direct mapped memory indexed by the CAM part.

As at most one entity writes into the TSU at any time point, the Graph Memory needs to have only one write port. As for the number of read ports, it is necessary to have two as, in addition to the Thread Load or Thread Completion Units, it is possible that the Thread Update Unit is also reading from the GM. This however applies only for the CAM part as the direct mapped part is only accessible by at most one unit at any point in time.

Kyriakos Stavrou

5.4.2.5 Threads-to-Update Buffer (TUB)

The Threads-to-Update Buffer (TUB) is used by the TSUs to insert consumers of the completed DThreads during the Thread Completion operation. Each such TUB-write operation requires first finding an empty entry. This can be done with the help of a small CAM indexing unit that will have as many bits as the entries of the TUB. These write operations can take place concurrently which makes necessary for the TUB to have multiple write ports. As having one write port per TSU will be very costly it is better to have a smaller number of write ports and use some arbitration logic. This second solution is not expected to affect the performance as according to our previous analysis for the *Thread Completed Unit* (Section 5.4.1.3), this operation is not in the critical path.

Regarding the Thread Update Unit its operation involves reading the valid entries from the TUB in order to service the corresponding requests. Finding the valid entries can be done with the help of the indexing structure described above. As reading is performed by this unit only, the TUB can have a single read port.

5.4.2.6 Consumers List (CL)

The Consumer List (CL) is indexed by the Graph Memory directly. Moreover, each entry indexes the next entry, as such, this unit can be implemented as a direct mapped memory. According to the previous description of the Load Thread operation, only one TSU can write to this unit at any time point which leads to a requirement for only one write port. However, reading from the CL could occur concurrently by all the TSUs which can happen only during the Thread Completed operation. As explained earlier however, this operation is not in the critical path. As such, having a smaller number of read ports combined with some arbitration logic is not likely to affect the performance.

Kyriakos Stavrou

5.4.2.7 Iteration Level Consumers List (ILCL)

As the operation of the Iteration Level Consumers List (ILCL) is identical to that of the Consumer List this unit can also be implemented with the same configuration.

5.4.2.8 Input Queue (InQ)

The input Queue is used as a buffer for the data packets received from the System Network until they are processed by the TSU. This queue can be is being written by the System Network Interface Unit and read by the Packets Management Unit. As such,the Input Queue can be implemented as a direct mapped memory unit with one read and one write port.

5.4.2.9 Output Queue (OutQ)

The Output Queue has similar operation as the Input Queue and so it is possible to use the same configuration. As such, this unit is can also be implemented by a direct mapped memory with one read and one write port.

5.4.2.10 PMU Buffer

The buffer of the Packets Management Unit stores all the packets for the different operations until the corresponding unit services them. This FIFO buffer has a single write, single read port and can be implemented as a direct mapped memory unit.

5.5 Hardware Budget Estimation

In this Section we present an estimation of the hardware budget of the TFluxHard Scheduler. This analysis is based on CACTI [93], a well known tool for estimating the area of on-chip caches.

Kyriakos Stavrou

Notice that the estimation focuses on the Scheduler's memory units as they are the ones that consume the largest portion of the real-estate. In particular, according to [33] the real-estate consumed by the logic units is in the order of 10%.

The configuration of the Scheduler analyzed here is the one used for the performance evaluation presented in Chapter 9. This configuration, which is summarized in the first rows of Table 5, regards to a Scheduler that serves 27 TFlux Kernels. In addition to the number of rows for each memory units, this Table reports the size of each row and the type of each unit according to the analysis of the previous Section.

The second part of Table 5 shows how each of these memory units was modeled in CACTI. This modeling follows the configuration of each memory unit but is also "limited" by the restrictions of CACTI. As an example, notice that we have merged the *Input Queue* and *Output Queue* in one unit (Queues) as CACTI can not model units that have the size of one queue (128 Bytes).

Table 5: The configuration of the memory units of TFluxHard Scheduler.

Real Configuration	**GM Index**	**GM Data**	**SM**	**RQ**	**TES**	**Queues**	**PMUQ**	**CL**	**ILC List**	**TUB**
- Memory Type	CAM	DM	DM	FIFO	FIFO	FIFO	FIFO	DM	DM	DM
- # entries	64	64	64	32	32	8	32	512	128	2048
- Bits per entry	33	77	160	33	33	33	33	42	57	33
CACTI Configuration	**GM Index**	**GM Data**	**SM**	**RQ**	**TES**	**Queues**	**PMUQ**	**CL**	**ILC List**	**TUB**
- Associativity	FA	FA	DM	DM	DM	DM	DM	DM	DM	DM
- Cache Size (B)	266	617	1281	133	133	34	128	2689	913	8449
- Block Size	10	32	8	16	10	8	8	16	16	16
- Associativity	FA	1	1	1	1	1	1	1	1	1
- Read Ports	1	1	1	1	1	1	1	1	1	1
- Write Ports	1	1	1	1	1	1	1	1	1	1
- R/W Ports	0	0	0	0	0	0	0	0	0	0
- # sub-banks	1	1	1	1	1	1	1	4	4	4
#Transistors	37	39	74	25	25	14	15	314	136	776

As CACTI's results are in terms of area and not transistor count, we determined the #transistor/area ratio using a known cache example. We modeled in CACTI, the Data-Cache of a 180nm Pentium III processor and compared its area with the processor's floorplan and total number of

Kyriakos Stavrou

transistors. For the 180nm technology, this ratio was found to be 185K transistors per mm^2. Based on this analysis, the private memory units of the Scheduler require a total number of 229K transistors. As for the units that are shared, *i.e.* TUB, ILC List and CL, their total hardware budget is in the order of 1326K transistors. As such, a Scheduler able to serve 27 TFlux Kernels requires 7509K transistors. As for the hardware budget *per TFlux Kernel* this is equal to 259K transistors.

5.6 TFluxHard Implementation Issues

Although the TFluxHard system does not require any modifications to the CPU cores, it requires the system to be equipped with the Scheduler implemented as a hardware device. As such, it was not possible to apply TFluxHard directly to an existing machine. However, we are exploring several alternatives for a hardware prototype which are presented in Section 5.6.2.

5.6.1 Current Design

Currently TFluxHard exists at the simulation level. The simulated TFluxHard system has been built on top of the Simics full system simulator [72]. Simics models the major components of a system in such level of detail that allows to the simulated machine to boot an unmodified Operating System. These components include the CPU cores, their interconnection, the caches, the main memory and the motherboard (more details about Simics will be given in Section 8.1.1).

The virtualization offered by the TFlux Platform allowed us to simulate TFluxHard without any modification to any component, *i.e.* the CPUs, the memory hierarchy or the interconnection network. As for the scheduler, it was developed using the Device Modeling Language [98] (a language provided by Simics for describing hardware components) and was attached to the network as a memory mapped device. Although our model for the Scheduler is not cycle-accurate, it includes all logic and memory units described in the previous Sections.

Kyriakos Stavrou

The operation of TFluxHard was validated using different configurations. These included setups with the number of CPUs ranging from 2 to 28 CPUs (28 is a limitation of the Simics version we are using), setups with *x86* and *UltraSPARC II* CPUs and setups with different versions of Linux.

More details about these machines and the relevant experimentation are presented in Chapters 8 and Section 10.1 respectively.

5.6.2 Possible Future Implementations

For a "real" implementation of TFluxHard the only requirement is for the CPUs to have access to a hardware module providing the Scheduler functionality. Although there are multiple approaches for developing a TFluxHard machine in this Section we discuss only three.

The first approach regards the implementation of the Scheduler as a hardwired module inside a multicore chip such as the Intel Quad Core [51] or the AMD Opteron [2] CPUs. The Scheduler can be attached to the on-chip system network and made accessible as a memory mapped device. The only requirement for this configuration will be an additional connection on the system network for the Scheduler.

The second approach regards the implementation of TFluxHard using future multicores that will include on-chip programmable logic [27, 95, 96, 97]. Similar to other proposals that use programmable logic to speedup execution [89, 91, 123, 124], for TFluxHard this component will be used for the implementation of the Scheduler. For such a system, the TFluxHard applications could trigger mechanisms to load the programmable logic with the description of the Scheduler prior to their execution.

The third approach regards the implementation of the Scheduler on an off-chip device which will be fast to access from the CPUs. An example of such a device is the HTX board [36, 105]

Kyriakos Stavrou

that uses the AMD HyperTransport technology [49]. Given that the HTX board provides programmable logic it can be used for the implementation of the Scheduler leading to a configuration where TFluxHard platform will be applicable to existing off-the-shelf commodity system with an add-on card. However, for such a solution it might be necessary to optimize the runtime of TFlux to make it more tolerable to the Scheduler's delay. A technique towards this direction is to read multiple ready DThreads from the corresponding TSU instead of only one at a time.

Kyriakos Stavrou

Chapter 6

TFluxSoft

TFluxSoft is the other incarnation of the TFlux Platform presented in this work. The key contribution with this incarnation is the fact that TFluxSoft makes it possible for applications written for the TFlux platform, *i.e.* applications that exploit parallelism using an efficient parallel application execution model, to execute on *off-the-shelf* systems without significant penalty. Applications developed for TFluxSoft may execute on a multitude systems without *any* requirement for additions or modifications to the hardware or software of the host machine.

The major issue regarding the TFluxSoft is to provide the Scheduler's functionality at the software level. In order to achieve this, instead of having a hardware module providing the Scheduler's operations to the Runtime System, these operations are provided by software entities. In particular, in TFluxSoft four of the five Basic Operations of the Scheduler are provided by the processors/processes running the TFlux Kernels whereas one of the processors/processes is dedicated to providing the fifth operation, *Thread Update*. Due to the operation it performs, this processor/process is called the *"Updater"*. Notice that while TFluxSoft conceptually does not

Kyriakos Stavrou

require any multiprocessor or multicore system to execute, for performance sake, each Kernel and the Updater should execute on its own processing element (e.g. processor or core). As in this work we target the multicore hardware, we will assume each Kernel executes on its own core and the Updater executes on another core. Reserving a core for the execution of the Updater is the tradeoff made in order to offer the benefits of TFlux on off-the-shelf systems.

Without loss of generality Figure 43-(a) depicts a multicore with 8 cores where 7 of them are used to execute the TFlux Kernels and 1 is used for the execution of the *Updater's* code. Notice that the processors executing the TFlux Kernels also provide some of the TSU operations depicted as *Local TSU* in this Figure. For systems with large number of cores, in order to avoid contention at the Updater, TFluxSoft may operate with multiple Updaters. For illustration purposes only, Figure 43-(b) depicts a configuration with 2 *Updaters* and 6 cores running the TFlux Kernels.

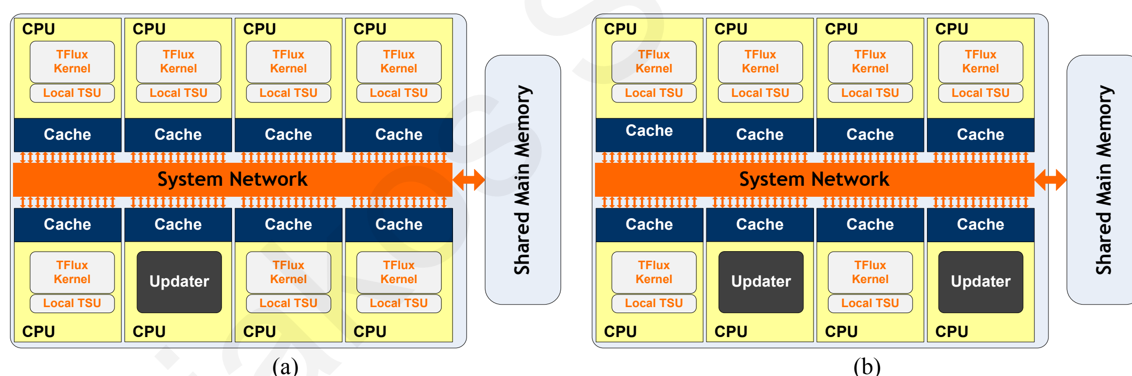


Figure 43: TFluxSoft multicore configured with (a) 1 *Updater* and 7 processors executing TFlux Kernels. (b) 2 *Updaters* and 6 processors executing TFlux Kernels.

This Chapter focuses on the implementation of the Scheduler at the software level which is named the *"SoftScheduler"*. Section 6.1 introduces *SoftScheduler* and discusses the implementation of the Basic Operations. Moreover, this Section studies the cost of these operations and identifies potential bottlenecks. Section 6.2 presents the design of *SoftScheduler* and discusses

Kyriakos Stavrou

how each performance bottleneck is targeted. Finally, Section 6.3 present possible extensions of TFluxSoft for better scalability.

6.1 SoftScheduler

Other than its implementation being at the software-level, *SoftScheduler* provides the same *Basic Operations* as the generic TFlux Scheduler. Nevertheless, instead of using a centralized approach, it distributes the execution of these operations to the processors executing the Kernels and to one processor dedicated to the execution of the *Thread Update* operation (*Updater*).

Figure 44 depicts the SoftScheduler configured to serve a TFluxSoft system with two Kernels. As can be seen from this Figure, SoftScheduler introduces some additional units compared to the generic Scheduler of TFlux (Section 3.1.3). The purpose and operation of these extra units (Local TUB, TKT, LIT, CAR and ILCAR) will be explained in Section 6.2 which presents the design issues of SoftScheduler.

Dynamically Managed Units		Static Units	
GM	Graph Memory	TKT	Thread to Kernel Table
SM	Synchronization Memory	LIT	Loop Information Table
TES	Thread Execution Stack	CAR	Consumers Array
TUB	Threads-to-Update Buffer	ILCAR	Iteration-level Consumers Array

Figure 44: SoftScheduler configured to serve 2 TFlux Kernels.

Kyriakos Stavrou

6.1.1 Implementation of Basic Operations

As explained earlier in Section 3.3, for a system to provide to its user the DDM model of execution, its scheduler is required to provide to the Runtime Support System five Basic Operations: (1) the *"Thread Load"* operation which enables the TFlux Kernels to load the Scheduler with the metadata of the DThreads to be executed; (2) the *"Find Next Thread"* operation which is used by the Kernel to find the next ready DThread to execute; (3) the *"Thread Completion"* operation by which the Scheduler is notified each time a DThread completes its execution; (4) the *"Thread Update"* operation by which the Ready Count values of the Consumers of the completed DThreads are decreased and (5) the *"Clear TSU operation"* which clears the resources allocated onto the several units of the Scheduler.

The sections that follow present the implementation of these operations for SoftScheduler.

6.1.1.1 Thread Load

The *Thread Load* operation is performed by the TFlux Kernel during the execution of the *Inlet DThread* and its purpose is to load the metadata corresponding Block's DThreads into the Graph Memory (GM) and Synchronization Memory (SM) structures. The only difference of this operation compared to its generic specification for TFlux (Section 3.3.1) is the way the Consumers are stored.

As can be seen from Figure 44, instead of a Consumer List, SoftScheduler has another unit named ***Consumers Arrays (CAR)***. This unit has as many arrays as the number of DThreads and each such array stores the Thread Templates of all Consumers of the corresponding DThread. The advantage of CAR compared to the Consumer List is that the former is *private* to each TSU and consequently eliminates any operations to be granted mutual exclusion that would be necessary if the Consumer List was used. Such a requirement for mutual exclusion would be necessary during

Kyriakos Stavrou

the load operation as it would be possible that two different Kernels were concurrently inserting their consumers into the Consumer List. Moreover, CAR guarantees that the space allocated to store the Consumers is the minimum possible. As for the Iteration-level Consumers, they are stored in the ILC ARRAY with a rational similar to what applies for the CAR.

6.1.1.2 Thread Completed

The *"Thread Completed"* operation is executed when a DThread completes its execution and has three different versions. The first version, *"Execution Completion"* (Section 3.3.2.1), applies for DThreads that do not belong to a recycle-group (Section 3.2.3). The second version is *"L-DThread Recycle"* (Section 3.3.2.2) and applies for Loop DThreads that do not belong to the last generation of L-DThreads (Section 3.2.2.2) and finally, the third version *"DThread Recycling"* (Section 3.3.2.3) applies for DThreads that belong to a recycle-group.

Execution Completion

When a DThread completes its execution its Kernel invokes the *Execution Completion* operation which is a two steps process. First the Kernel inserts the Thread Templates of the Consumers of the completed DThread in the "Threads-to-Update Buffer" (TUB) in order for their Ready Count values to be decreased. Notice that TUB is shared among all Kernels and consequently, before writing into this unit, the Kernel is required to lock it for mutual access. The second step is to remove the completed DThread from the *Thread Execution Stack* (TES) which is private to each Kernel.

Kyriakos Stavrou

L-DThread Recycle

As explained in Section 3.2.2.2, during the execution of TFlux Loops, all L-DThreads except those of the last generation, “recycle” themselves to execute a new iteration. For this purpose, instead of performing the *Execution Completion* operation, the TFlux Kernel invokes the *L-DThread Recycle* operation.

During *L-DThread Recycle* operation the TFlux Kernel removes the L-DThread from the Thread Execution Stack (TES) and inserts the *new* instance of the L-DThread in the Graph and Synchronization Memory units. In case the L-DThread that completed its execution has Iteration-level Consumers its Kernel inserts their Thread Templates into the TUB.

Thread Recycle Execution

In addition to the actions taken by the *Execution Completion* operation, *Thread Recycle Execution* reinserts the metadata of the completed DThread into the Graph and Synchronization Memory units. The only difference of *Thread Recycle Execution* compared to *L-DThread Recycle* is that whereas the former reinserts the DThread with exactly the same metadata, the latter modifies the Iteration Id field in order for the new instance of the L-DThread to execute a different iteration.

6.1.1.3 Thread Update

This operation, which is performed by the *Updater*, decreases the Ready Count value of the Consumers of the completed DThreads the identifiers of which have been inserted by the TFlux Kernels in the TUB during the *Thread Completion* operation. To achieve this, the *Updater* traverses the TUB and for each valid entry, it locates the Ready Count value of the corresponding DThread and decreases it.

Kyriakos Stavrou

6.1.1.4 Find Ready Thread

This operation is executed by the TFlux Kernel and its target is to find the next ready DThread. In the common case, where the *Thread Execution Stack* (TES) is not empty (recall that the Thread Execution Stack holds the Thread Templates of the DThreads that have been *scheduled* for execution), the operation completes by returning the DThread at the head of the TES. If the TES *is* empty, the TFlux Kernel first traverses its own Graph and Synchronization Memory units and copies the Thread Templates of all ready threads into the TES. If the TSU contains no ready DThreads this operation will return to the TFlux Kernels the Thread Template 0/0. This particular template forces the TFlux Kernel to request again for a ready DThread. As such, this process will complete when a ready DThread has finally been returned to the TFlux Kernel.

6.1.1.5 Clear TSU

This operation is performed by the TFlux Kernel during the execution of the *Outlet DThread* and its result is to release the resources allocated into the several Scheduler units for the execution of the particular Block.

6.1.2 Mutual Exclusion

During the execution of these Basic Operations the different parallel entities (the TFlux Kernels and the *Updater*) may access the same units concurrently. As such, to guarantee the validity of data, in some cases, mutual exclusion is required.

When performing the *Thread Load* operation the TFlux Kernel needs to lock the GM and SM structures as it is possible that at the same time the *Updater* is decreasing a Ready Count value while performing a *Thread Update* operation. The three variations of the *Thread Completion* operation, *i.e. Execution Completed*, *L-DThread Recycled* and *Thread Recycled*, write the consumers of the completed DThread into the TUB. Locking the TUB is necessary as at the same time it is

Kyriakos Stavrou

possible that another TFlux Kernel is also performing the *Thread Completion* operation or that the *Updater* is invalidating an entry while performing the *Thread Update* operation.

Table 6 summarizes the units read, written and locked during the execution of the Basic Operations. In addition, it reports the parallel entity that performs each Basic Operation.

Table 6: Summary of the Basic Operations.

Operation	***Performed By***	***Units***		
		Write	***Read***	***Locks***
Thread Load	Kernel	GM/SM	GM/SM	GM/SM
Thread Completed				
- Execution Completion	Kernel	GM/TUB/TES	GM/TUB/TES	TUB
- L-DThread Recycle	Kernel	GM/TUB/TES	GM/TUB/TES/SM	TUB
- Thread Recycle Execution	Kernel	GM/TUB/TES	GM/TUB/TES/SM	TUB
Find Ready Thread	Kernel	TES/GM/SM	TES/GM/SM	-
Clear TSU	Kernel	GM/SM	GM/SM	GM/SM
Thread Update	Updater	TUB/GM/SM	TUB/GM/SM	TUB/GM/SM

6.1.3 Upper Bound of Basic Operations Cost

The Scheduler's Basic Operations are critical for the performance of the TFluxSoft system as they are the main source of the parallelization overhead. As such, it is important to understand the factors that affect their cost and identify potential performance bottlenecks. In this Section we present a theoretical analysis of the upper bound of the cost of these operations[1] .

The most common operations, and consequently the ones that are most critical to the overall performance, are the three variations of *Thread Completed* (*Execution Completed*, *L-DThread Recycle* and *Thread Recycle Execution*), the *Find Ready Thread* and the *Thread Update*. This is due to the fact that the *Thread Completed* and *Find Ready Thread* operations are executed each time a DThread completes whereas the third, *Thread Update* is executed continually by the *Updater* in order for new DThreads to be deemed executable. Moreover, notice that the *Find*

[1]Part of the work presented in this Section was done in collaboration with Demos Pavlou who during his Final-Year Project [84] worked on the TFluxSoft system

Kyriakos Stavrou

Ready Thread operation is expected to be invoked more times compared to *Thread Completed* as it is possible that a ready DThread is not always present. A factor that makes the cost of *Thread Update* of major importance is the fact that it is this operation that deems DThreads ready for execution. If this operations is not efficient, there will be delay in enabling DThreads for execution with a consequent negative impact on performance.

Table 7 presents the theoretical upper bound of the cost of the Basic Operations as a function of the different parameters of the System. This cost is analyzed in the paragraphs that follow.

Table 7: Theoretical analysis of the cost of the SoftBasic Operations. This Table comes from [84]

Operation	*Order of the execution time*
Thread Load	$O\left(GM_{Size} + (SM_{size} + \#Instances)\right)$
Execution Completion	$O\left(t^{TUB}_{Lock} + TUB_{size} + \#Consumers\right)$
L-DThread Recycle	$O\left(t^{TUB}_{Lock} + TUB_{size} + \#Consumers + \left(t^{GM}_{Lock} + SM_{size}\right)\right)$
Thread Recycle Execution	$O\left(t^{TUB}_{Lock} + TUB_{size} + \#Consumers + \left(t^{GM}_{Lock} + SM_{size}\right)\right)$
Find Ready Thread	$O\left(GM_{size} \cdot \#entries\ per\ SM\ row\right)$
Clear TSU	$O\left(GM_{size} + t^{GM}_{Lock}\right)$
Thread Update	$O\left(\#TFlux\ Kernels \cdot GM_{size} + t^{TUB}_{Lock} + t^{GM}_{Lock}\right)$

Thread Load: The execution cost of the *Thread Load* operation can be split into two parts; loading the metadata of the DThread onto the Graph Memory (GM) and writing the Ready Count values of these DThreads onto the Synchronization Memory (SM). Loading the DThread's metadata onto the GM requires finding an empty entry. This is performed by sequentially checking the entries of this unit until an empty entry is found $\left(O\left(GM_{Size}\right)\right)$. If the DThread to be loaded is not an L-DThread, its Ready Count is stored in the RC field of the GM whereas for L-DThreads this field is an index to the SM. To insert the Ready Count values into the SM it is first necessary to find an empty SM entry $\left(O\left(SM_{size}\right)\right)$ which is done in the same way as in the case of the GM. Then the Ready Count counters are set to their initial value, which is related to the number of instances this L-DThread has $\left(O\left(\#Instances\right)\right)$.

Kyriakos Stavrou

Thread Completion - Execution Completion: The *Execution Completion* operation accesses the TES to remove the entry corresponding to the DThread that has completed its execution. As this entry is always at the top of the TES, the cost for this action is constant. The lengthy part of this operation is related to inserting the Consumers of the completed DThread into the TUB. To perform this, the Kernel needs to lock the TUB for mutual exclusion $\left(O\left(t_{Lock}^{TUB}\right)\right)$. Then for each Consumer it needs to find an empty entry in the TUB $(O\left(TUB_{size}\right))$ in order to write its identifier $(O\left(\#Consumers\right))$.

Thread Completion - L-DThread Recycle: In addition to the actions taken by *Execution Completion* operation, *L-DThread Recycle* needs to create a new instance for the L-DThread that completed its execution. For this additional action, *L-DThread Recycle* needs to lock the GM for mutual access in order to write the metadata of the *new* instance of the completed L-DThread $\left(O\left(t_{Lock}^{GM}\right)\right)$. The next step is to access the SM in order to set the Ready Count counter which may involve finding an empty SM entry $(O\left(SM_{size}\right))$.

Thread Completion - Thread Recycle Execution: The *Thread Recycle Execution* operation has the same cost as *L-DThread Recycle* as the actions it takes are of the same cost.

Find Ready Thread: The common case for the *Find Ready Thread* operation is when the Thread Execution Stack (TES) is *not* empty. If this is the case, this operation completes just by returning the Thread Template located at the head of the TES with a constant cost. However, if TES *is* empty *Find Ready Thread* needs to first traverse the GM and SM units and copy all ready DThreads into the TES. For this action, it accesses the GM and for each valid entry searches the SM to find a ready DThread $(O\left(GM_{size} \cdot \#entries\ per\ SM\ row\right))$.

Clear TSU: The purpose of the *Clear TSU* operation is to access the GM unit and lock it in order to invalidate its entries. As such, its cost is related to the size of the GM$\left(O\left(GM_{size} + t_{Lock}^{GM}\right)\right)$.

Kyriakos Stavrou

Thread Update: During the *Thread Update* operation the *Updater* traverses the Threads-to-Update Buffer (TUB) and for each valid entry it locates the corresponding Ready Count counter and decreases its value. To locate the corresponding GM entry the *Updater* needs to traverse the different GM units in order to find a match $(O\left(\#TFlux\ Kernels \cdot GM_{size}\right))$. Whenever such an update-request is to be served the TUB and GM structures need to be locked for mutual access $\left(O\left(t_{Lock}^{TUB} + t_{Lock}^{GM}\right)\right)$.

6.1.3.1 Potential Bottlenecks

The theoretical analysis of the upper bound cost of the Basic Operations reveals three issues that are likely to have an important negative impact on performance. These issues, which are detailed in this Section, are mostly related to the *Thread Completion* and *Thread Update* operations.

Potential Bottleneck 1: Mutual Exclusion Access to TUB

Whenever a DThread completes its execution its Kernel will invoke one of the three variations of the *Thread Completion* operation. Each of these variations requires the TFlux Kernel to pose an update-request for each Consumer of the completed DThread by inserting its Thread Templates into the TUB. As explained in Section 6.1.2, each such action requires the TFlux Kernel to lock the TUB for mutual exclusion. At the same time however, it is possible that another TFlux Kernel which is also performing a *Thread Completion* operation that is also trying to lock the TUB. Another possibility is that the *Updater* is trying to lock the TUB during the execution of the *Thread Update* operation.

As the number of TFlux Kernels increases the number of parallel execution entities compeeting to be granted mutual access to the TUB will also increase with a consequent increase of the time required for this lock operation. This fact has, in order, a negative effect on the scalability of the system.

Kyriakos Stavrou

Potential Bottleneck 2: Finding the Ready Count Counter

During the *Thread Update* operation the *Updater* reads entries from the TUB and then locates the Ready Count counter that corresponds to this update-request in order to decrease its value. For this task the *Updater* needs to traverse the different Graph Memory structures until a match is found with a cost linearly dependent to the number of TFlux Kernels (recall that there is one Graph Memory structure for each TFlux Kernel). This dependence is another factor that has a negative effect on the scalability of TFluxSoft.

Potential Bottleneck 3: Update-Requests for DThreads that do not exist

Applications exploiting the Iteration-level Dependencies often cause high contention to the TUB structure, which in turn, leads to non negligible performance degradation. The main origin of this behavior is that a large portion of the TUB entries may to regard to L-DThreads that have *not yet been created*.

To better explain this assume an application with two TFlux Loops that depend at the loop-iteration level. When an L-DThread of the producer TFlux Loop completes, an Update-Request for its iteration-level consumers L-DThreads will be inserted into the TUB. If this Consumer L-DThread is not yet present into the TSU (this is possible if this Consumer L-DThread does not belong to the first L-DThread generation (see Section 3.2.2.2)), the *Updater* will not be able to complete serving the update-request. This request will *not* be invalidated from the TUB and the *Updater* will retry to serve it the next time it finds it into the TUB. As the number of these unsuccessful serving attempts increase, the *Updater* will be less efficient in serving the update-requests which are necessary to allow the program to continue with a consequent negative effect on performance.

Kyriakos Stavrou

6.2 SoftScheduler Design Issues

This Section presents the design choices made for SoftScheduler to target the bottlenecks identified in the previous Section and to decrease as much as possible the cost of the Basic Operations.

6.2.1 Thread to Kernel Indexing (TKI)

As explained earlier, during the *Thread Update* operation, for each update-request the *Updater* is required to find the Ready Count counter corresponding to the Consumer being serviced in order to decrease its value. As depicted in Figure 45-(a), for this purpose the *Updater* needs to access sequentially the different Graph Memory structures until it finds the corresponding Ready Count counter. This lengthy process is a limitation to the performance and scalability of the system as its cost increases with the number of TFlux Kernels.

To overcome this limitation TFluxSoft includes a special table named the *"Thread to Kernel Table"* (TKT) (Figure 45-(b)) which allows the *Updater* to *directly* locate the Graph Memory unit corresponding to the update-request being served (*host GM*). TKT has as many entries as the number of different DThreads of the program and each entry of this table is a pointer to the DThread's *host GM*. To allow fast accessing this table is indexed with the Thread Id, *i.e.* TKT entry x corresponds to the DThread with Thread Id x.

Figure 45-(b) depicts the process of identifying the Ready Count counter corresponding to the update-request being served with the help of the TKT. In particular, the *Updater* accesses the TKT using the Thread Id which allows it to *directly* access the *host GM*. Then, by sequentially accessing this unit, it locates the entry that matches the request being served and therefore the corresponding Ready Count counter.

With this technique only one GM needs to be searched for a match decreasing significantly the execution time of *Thread Update* operation. Notice that the TKT is inserted into the application's

Kyriakos Stavrou

code statically by the TFlux Preprocessor and therefore its existence does not come with any disadvantage.

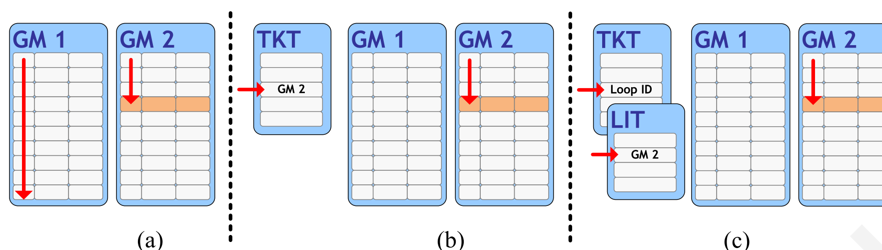


Figure 45: (a) Finding the host GM by searching all GMs sequentially (b) Finding the host GM using the Thread to Kernel Table (TKT) for DThreads (c) Finding the host GM using TKT and LIT for L-DThreads. *Vertical lines show the entries being searched.*

To avoid having a too large TKT, only one entry per DThreads exits for DThreads causing multiple instances, *i.e.* for DThreads executing loops (L-DThreads). For such cases, for the *Updater* to locate the *host GM* one additional step is required. In particular, for L-DThreads the entry in the TKT is not a pointer to the *host GM* but rather a pointer to another table, named the *Loop Information Table* (LIT) (Figure 45-(c)). Each entry of this table contains information for one TFlux Loop that allows the *Updater* to calculate with very few operations the TFlux Kernel that executes the particular L-DThread and therefore the *host GM*. The LIT is also added in the application's code statically by the TFlux Preprocessor.

The TKT and LIT units are accessed by the *Updater* only, as such, given their small size these units are likely to be resident in the cache of the corresponding CPU. The introduction of TKT and LIT leads to an overall performance benefit of more than $10\times$ on systems with large number of processors (> 16) even for programs with one TFlux Loop.

Kyriakos Stavrou

6.2.2 TUB Segmentation

The *Threads-to-Update Buffer* (TUB) is a shared unit which is modified by both the *Updater* and all TFlux Kernels. In particular, the TFlux Kernels write into the TUB the Consumers of the completed DThreads during the *Thread Completed* operation (Figure 46-(a)) whereas the *Updater* reads and invalidates these entries during the *Thread Update* operation (Figure 46-(b)). As these operations execute in parallel, the TUB needs to be locked in order to guarantee the validity of the data. This mutual exclusion requirement leads to an overhead which increases with the number of TFlux Kernels.

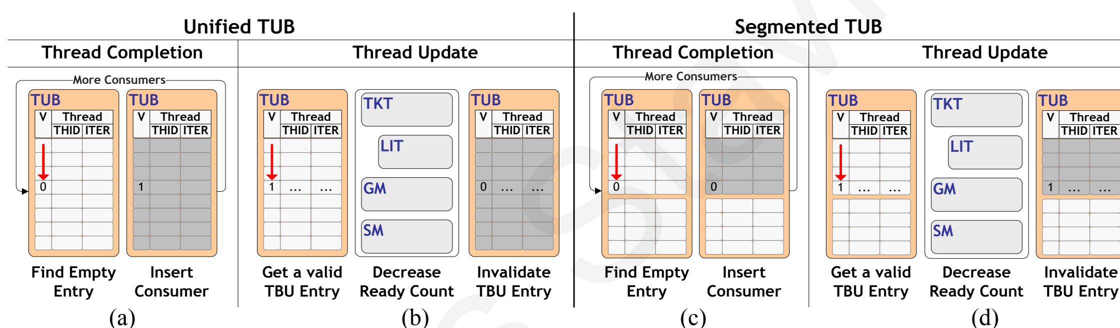


Figure 46: (a) Thread Completion operation using unified TUB (b) Thread Update operation using unified TUB (c) Thread Completion operation using TUB with two Segments (d) Thread Update operation using TUB with two Segments. *Locked units are shown shaded.*

To decrease the idle time during which the *Updater* or the TFlux Kernels wait for the TUB to become available, this unit has been partitioned into segments (Figure 46-(c)-(d)). When a TFlux Kernel needs to write into the TUB it applies a *try/lock* operation (*try/lock* is not blocking as it locks a unit only if this unit is free) on the segments in a round-robin fashion until it manages to lock one (Figure 46-(c)). Notice that only one segment is locked at a time, as such, other operations requiring mutual access to TUB can proceed with the rest of the segments. This allows to as many Kernels as the number of segments to perform the *Thread Completed* operation concurrently.

Kyriakos Stavrou

To perform the *Thread Update* operation the *Updater* follows exactly the same rationale. In particular, it traverses the segments in a round robin fashion and it locks a segment only if it is not used by another entity at the same time (Figure 46-(d)).

Having too many segments is likely to harm the performance due to the increased number of lock operations that will be required. This is due to the fact that as the number of segments increases the size of each segment decreases. For DThreads with multiple update-requests one segment may not be enough, as such, the TFlux Kernel may need to insert a number of these requests to other segments as well. This will cause additional lock operations which number may increase with the number of TUB segments.

The segmentation of the TUB leads to an important performance benefit for the system. In particular, even for very simple applications the performance improvement more than doubles.

6.2.3 Local TUB

To avoid the bottleneck presented in Section 6.1.3.1 as *"Potential Bottleneck 3: Update-requests for DThreads that do not exist"* we introduced the *Local TUB* unit which is private to each TSU (Figure 44). When the *Updater* serves an update-request that corresponds to a DThread which is not present in the structures of the corresponding TSU, it moves this request from the TUB to the *Local TUB* of this TSU. As such, the *Updater* will manage to serve this TUB entry the *first time* it finds it. As for the update-request inserted into the Local TUB, it will be used to decrease the Ready Count counter of the DThread it corresponds to during the *Find Read Thread* operation, which, for this purpose, has been slightly modified.

As explained in Section 6.1.1.4, whenever *Find Ready Thread* is invoked it accesses the Thread Execution Stack (TES) in order to return to the CPU the next ready DThread. If however, the TES is empty, *Find Ready Thread* first accesses the GM/SM structures in order to move all ready

Kyriakos Stavrou

DThreads to the TES. To serve the update-requests present in the Local TUB this second step of the *Find Ready Thread* operation is modified. In particular, whenever *Find Ready Thread* finds a *not* ready DThread into the GM it accesses the Local TUB to test if a corresponding update-request exits; and if this is the case, it decreases its Ready Count counter. To decrease as much as possible the overhead of this additional step, the Local TUB structure is implemented as a highly hashed, to the Iteration Id, two dimensional matrix.

The cost of this additional step is, in the worst case, in the order of $O\left(\frac{GM_{size} \cdot SM_{size} \cdot LocalTUB_{size}}{HashFactor}\right)$. However, the fact that in the common case only a very small number of *not* Ready DThreads will exist in the GM, combined with the hihgly hashed nature of Local TUB makes this step to have significantly lower cost.

The experimental results showed Local TUB to have an impact that may result in a performance improvement up to $8\times$. This happens for the cases of applications with large TFlux loops exploiting the iteration-levels dependencies feature.

6.2.4 TUB Buffers

The *TUB Buffers* are used to decrease the number of times a TFlux Kernel needs to lock the TUB segments in order to perform the *Thread Completion* operation. As can be seen from Figure 47-(a), which depicts the execution of the the *Thread Completion* operation without using TUB Buffers, each time a Consumer is to be inserted into the TUB the TFlux Kernel locks a segment for mutual exclusion. As such, the number of times a TFlux Kernel locks a segment is equal to the number of the Consumers of the completed DThread.

The idea behind the *TUB buffers* is that all Consumers of the completed DThread are first copied in a small *private* unit named *"TUB Buffer"*. Then a TUB segment is locked once and the Thread Templates of *all* consumers are copied from this buffer into the TUB (Figure 47-(b)).

Kyriakos Stavrou

For applications with DThreads that have multiple consumers TUB Buffers have been found to contribute more than 4% to the overall performance.

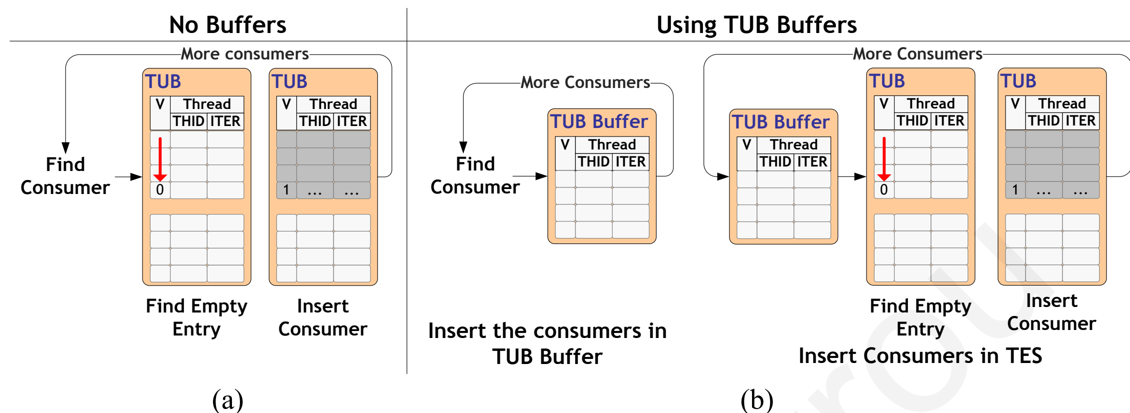


Figure 47: Inserting Consumers into the TUB (a) Without TUB Buffers (b) With TUB Buffers.

6.2.5 TUB Ranges

Execution of TFlux Loops often causes inserting into the TUB multiple consumers with the identical Thread Id and consecutive Iteration Ids. An example of such a situation is when a TFlux Loop depends on a DThread (Section 3.2.2.1). To avoid using multiple TUB entries for these update-requests, each TUB entry is able to represent multiple update-requests for DThreads with identical Thread Id and a consecutive range of Iteration Ids. In addition, to leading to a smaller TUB unit, the *TUB Ranges* decrease the time required for inserting the entries in the TUB. Moreover, as DThreads with the same Thread Id but consecutive Iteration Ids are likely to be stored in the same GM unit, the *TUB Ranges* also benefit the *Thread Update* operation.

For applications with multiple TFlux loops, using TUB Ranges was found to provide performance benefit reaching 8%.

Kyriakos Stavrou

6.2.6 Summary

After having presented the purpose and operation of the SoftScheduler units as well as the relevant design issues it is useful to present the size and number of entries for each of these units. Notice that we carefully selected the data types used for these units in order to decrease as much as possible their size and the consequent cache pollution caused by the operations that use them. The final configuration of these units which was used for the evaluation of the architecture (Chapter 9) is presented in Table 8.

Table 8: Size of the SoftScheduler units. The size of the TUB regards a configuration with 27 TFlux Kernels.

Unit	*Size (bytes)*			*Accessed By*
	Entry	*#Entries*	*Total*	
GM	18	64	1152	Updater / Local Kernel
SM	67	64	4288	Updater / Local Kernel
TES	6	512	3072	Local Kernel
Local TUB	32	16	512	Updater / All Kernels
TUB	7	512	3584	Updater / All Kernels

6.3 TFluxSoft Scalability Issues

Execution under TFluxSoft is performed by a number of TFlux Kernels served by an *Updater* which is executed on one of the on-chip cores. In the first TFluxSoft design there was only one *Updater* for all the Kernels in the system. However, as the number of TFlux Kernels increases, for applications with complex Synchronization Graphs it is expected that one *Updater* will not be able to serve them *efficiently* and consequently, will be the bottleneck to the system. To avoid this situation TFluxSoft was extended to be able to operate using multiple *Updaters*.

Using more than one *Updaters* leads to a tradeoff regarding the achievable performance. On one hand, given a specific number of cores, the *theoretical maximum* performance decreases as the

Kyriakos Stavrou

number of *Updaters* increases. This is due to the fact that the *Updaters* will "consume" more computation resources leading to less cores available for the execution of the application's DThreads. On the other hand however, having multiple *Updaters* lowers the cost of the *Thread Update* operation which, in turn, can lead to higher *overall* performance.

As explained in Section 6.2.2, for better performance the TUB is partitioned into segments. These segments are accessed by the TFlux Kernels in order to insert the identifiers of the Consumers of the completed DThreads and by the *Updater* to read these entries and perform the corresponding *Thread Update* operation (Section 6.1.1.3). When these parallel entities (TFlux Kernels and *Updater*) are to access the TUB, they perform a *try/lock* on the different segments in a round-robin fashion until they manage lock one which they release after completing the predefined task.

For the case of multiple *Updaters* the rationale remains the same. In particular, instead of having only one *Updater* trying to lock a segment and serve the included update-requests, there are multiple *Updaters* performing this operation. The main benefit of having multiple *Updaters* comes from the fact that in this situation a TUB entry will wait for a shorter period of time until it is served.

A study of the potential and the tradeoffs regarding the operation of TFluxSoft with multiple *Updaters* will be presented in Section 9.4.

Kyriakos Stavrou

Chapter 7

TFlux Evaluation Suite

This Chapter presents the set of applications used for the evaluation of the TFlux Platform. This set consists of 8 real-life applications and 10 synthetic applications. The targets set for the evaluation suite are discussed in Section 7.1. As for the details of the real-life and synthetic applications they are presented in Section 7.2 and Section 7.3 respectively. Notice that the TFlux Evaluation suite is general enough and may be used by any other architecture that supports a dataflow-like model of execution.

7.1 Introduction

In general, for an accurate performance evaluation of a system, it is necessary to use applications that have different characteristics in order to test the different components of the system. For the case of the TFlux platform, these characteristics should cover the size (in terms of number of dynamic instructions) and number of DThreads, the stress on the memory hierarchy, the programming constructs and most importantly, the complexity of the Synchronization Graph. Moreover,

Kyriakos Stavrou

the selected applications should correspond to compute-intensive workloads and have practical interest in real-life problems.

For selecting an application to be part of the TFlux Evaluation Suite we had two constrains. First, for an application to be eligible for this suite, we required for their code to be publicly available. This is due to the fact that in order execute an application in the TFlux platform the user needs to augment the original code with the TFlux compiler directives and then pass it through the TFlux Preprocessor. A second constrain comes from the fact that these applications are also executed on top of a simulator. As such, we were forced to exclude cases that required very long simulation time even with small input sizes.

To select applications for the TFlux Evaluation Suite we studied benchmarks from several well known suites such as the MiBench [44], MediaBench [64], PARSEC [16], SPLASH [129] and NAS [13] as well as algorithms commonly used in compute-intensive workloads. Given the constrains described above we selected 3 commonly used kernels namely the *Trapezoidal Rule for Integration*, *Matrix Multiply* and the *Runge Kutta* method for solving ordinary differential equations, 2 benchmarks from the MiBench suite, *Sort* and *Susan* and 3 benchmarks from the NAS Parallel Benchmarks suite, *FFT*, *CG* and *LU*.

Each of these applications will be presented in detail in Section 7.2. Notice that for these applications we used three different input sizes, *small*, *medium* and *large*. The *large* input size was defined such that the simulation time on our current machines was less than 24 hours. As for the *medium* and *small* input sizes they have been set to be the half and one quarter of the *large* input size respectively.

Although this set of “real-life applications”, *i.e.* applications that correspond to commonly used algorithms, is adequate to evaluate the performance and scalability of TFlux, in order to study specific features of the platform we also used a number of synthetic applications. The major

Kyriakos Stavrou

benefit of using these synthetic applications is that we were able to focus on specific characteristics of TFlux by excluding all other factors that could affect the results. These synthetic applications are presented in detail in Section 7.3.

In the description of the different applications we introduce the term *basic task* which corresponds to the smallest job that of the algorithm that usually produces a single result value. This *basic task* will be identified in order to study the parallelization of a particular algorithm. Ideally each *basic task* may be executed concurrently.

Finally, Table 9 presents the characteristics of the caches used to measure the miss rates, which are used in the dynamic behavior analysis for the different benchmarks.

Table 9: The characteristics of the caches used to measure the miss rates for the different benchmarks. This setup resembles the Quad-core AMD Opteron processor [2].

Parameter	*L1-Data*	*L2-Unified*
Size	32KB	2MB
# Lines	512	8192
Line Size	64B	256B
Associativity	4	8
Replacement Policy	LRU	LRU

7.2 Real-life Applications

This section presents the real-life applications. The order these applications are presented corresponds to the complexity of their Synchronization Graph, *i.e.* applications with simple graphs will be presented first while applications with complex graphs will be presented last. For applications that do not come from some benchmark suite but rather correspond to commonly used kernels, this section also presents the operation of the algorithm[1] .

[1]Part of the work presented in this Section was done in collaboration with Marios Nicolaides who during his Final-Year Project [83] worked on the TFlux Benchmark suite

Kyriakos Stavrou

7.2.1 Matrix Multiply (MMULT)

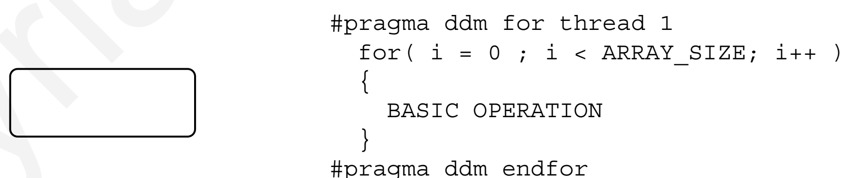
MMULT multiplies two 2-D matrices ($C = A \times B$). The *"basic task"* of this algorithm is the calculation of one element of the output matrix. To find the value of the element $C_{i,j}$ the *basic task* calculates the dot product of row i of matrix A and column j of matrix B. As such, each execution of the *basic task reads* a row from matrix A, a column from matrix B and only *modifies* the element $C_{i,j}$ which, for each *basic task* invocation, is a different memory location (Figure 48). Therefore, *all* invocations of the *basic task* can run in parallel.

$$\begin{bmatrix} C_{0,0} & C_{0,1} & C_{0,2} & \dots & C_{0,k} \\ C_{1,0} & C_{1,1} & C_{1,2} & \dots & C_{1,k} \\ C_{2,0} & C_{2,1} & C_{2,2} & \dots & C_{2,k} \\ \dots & & & & \\ C_{n,0} & C_{n,1} & C_{n,2} & \dots & C_{n,k} \end{bmatrix} = \begin{bmatrix} A_{0,0} & A_{0,1} & A_{0,2} & \dots & A_{0,n} \\ A_{1,0} & A_{1,1} & A_{1,2} & \dots & A_{1,n} \\ A_{2,0} & A_{2,1} & A_{2,2} & \dots & A_{2,n} \\ \dots & & & & \\ A_{m,0} & A_{m,1} & A_{m,2} & \dots & A_{m,n} \end{bmatrix} \times \begin{bmatrix} B_{0,0} & B_{0,1} & B_{0,2} & \dots & B_{0,k} \\ B_{1,0} & B_{1,1} & B_{1,2} & \dots & B_{1,k} \\ B_{2,0} & B_{2,1} & B_{2,2} & \dots & B_{2,k} \\ \dots & & & & \\ B_{n,0} & B_{n,1} & B_{n,2} & \dots & B_{n,k} \end{bmatrix}$$

Figure 48: Operation of *MMULT*.

7.2.1.1 Porting MMULT

The TFlux version of *MMULT* consists of a TFlux Loop (Figure 49-(a)). Each L-DThread of this loop calculates an element of the output matrix. As depicted in Figure 49-(b), the Synchronization Graph of *MMULT* can be expressed with a single *ddm loop* directive.



(a) (b)

Figure 49: (a) The Synchronization Graph of *MMULT*. (b) *MMULT* parallelized using TFlux directives.

Kyriakos Stavrou

7.2.1.2 Dynamic Behavior

As explained earlier, each L-DThread of the program's TFlux Loop calculates one element of the output matrix, as such, the number of DThreads for this program is equal to the number of elements of the output array. For the version of *MMULT* used in the TFlux Evaluation Suite the input size is the size of the *square* matrices being multiplied.

MMULT operates on a significant amount of data. The first matrix (A) is accessed by row whereas the second (B) is accessed by column. As such, the accesses to the second matrix (B) suffer from low spacial locality, therefore resulting in a significant number of data cache misses. In addition, if multiple instances of the *basic task* are executed on the same Kernel, if the matrices are large and the caches small, the temporal locality of the accesses will also not be captured. Although there are cache-efficient versions of this algorithm, the reason we did not select such a version is to include in our suite a benchmark that is representative of an application that causes a high stress on the memory hierarchy.

For *MMULT* the size of the *basic task*, in terms of number of dynamic instructions, depends on the input size of the application. As can be seen from Table 10 which summarizes the characteristics of *MMULT*, this size approximately doubles for an input array of double size. In particular, for the small input size (64×64) each execution of the *basic task* corresponds to 1196 instructions, for the medium size (128×128) to 2292 instructions and for the large size (256×256) to 4479 instructions. This effect is expected as the input matrices are square $n \times n$ resulting in the operations for a *basic task* to be $O(n)$. Unrolling the main loop also leads to an increase of this size. This increase is almost proportional to the unroll factor.

As for the L1-data cache miss rate, it is significantly affected by the input size. In particular, for the small input size the miss rate is in the order of $< 1\%$ wheareas for the medium and large

input sizes it reaches 49%. The miss rate however, is not affected by the unroll factor due the fact that consecutive elements of the output array use the same column of the second array which is the one that causes most misses due to its access pattern. As for the L2-data cache miss rate it is less than 1% for medium and large sizes. The L2-data cache miss rate for the small size appears to be in the order of 10% but this is due to the very small number of accesses (< 20).

Table 10: *MMULT* characteristics. Reported values are averaged over all executions of the *basic task*. Miss rates correspond to the cache configuration presented in Table 9.

DThread	***Unroll factor***	***Dynamic Instructions***	***Data Accesses***		***Miss Rate***	
			L1D	***L2D***	***L1D***	***L2D***
Small Size: 64×64						
1	1	1196	147	1	0.3%	10.6%
	8	9054	1077	3	0.2%	17.5%
	64	71521	8470	18	0.2%	17.4%
Medium Size: 128×128						
1	1	2292	276	130	47.2%	0.6%
	8	17823	2111	1039	49.2%	0.3%
	64	141697	16738	8303	49.6%	0.2%
Large Size: 256×256						
1	1	4479	533	258	48.4%	0.2%
	8	35307	4172	2062	49.4%	0.2%
	64	281902	33256	16485	49.6%	0.1%

7.2.2 Trapezoidal Rule for Integration (TRAPEZ)

TRAPEZ calculates the definite integral of a function in a given interval. To perform this calculation in parallel the integration interval is partitioned into subintervals to which the Trapezoidal rule is then applied (Figure 50). This operation is the *basic task* of the algorithm. The sum of all the partial results is the value of the integral, according to this numerical method.

Each invocation of the *basic task* can be executed in parallel given that each processor keeps a *private* variable for the sum of the partial results it calculates. The final step of the algorithm is to calculate to sum of these private variables (reduction operation) in order to get the total value of the integral.

Kyriakos Stavrou

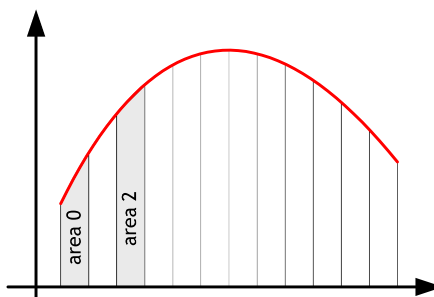


Figure 50: Operation of the parallel version of TRAPEZ.

7.2.2.1 Porting TRAPEZ

The TFlux version of TRAPEZ consists of a reduction TFlux Loop (Figure 51-(a)) where each DThread performs the *basic task* on a different subinterval. As depicted in Figure 51-(b), the Synchronization Graph of TRAPEZ can be expressed with a single *ddm loop reduction* directive.

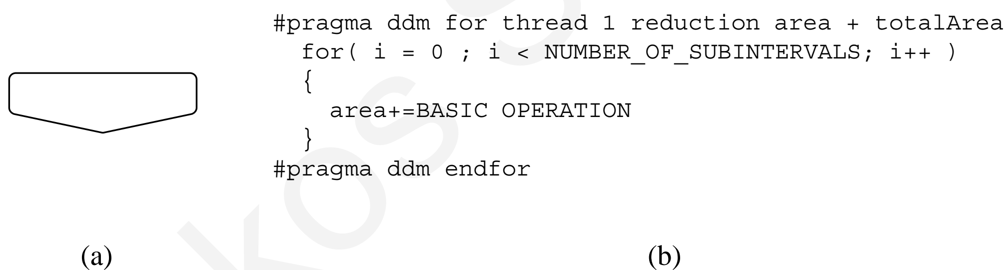


Figure 51: (a) The Synchronization Graph of TRAPEZ. (b) TRAPEZ parallelized using TFlux directives.

7.2.2.2 Dynamic Behavior

As each DThread of the program's TFlux Loop performs the *basic task* on a subinterval, the number of DThreads in the program will be equal to the number of subintervals plus the number of reduction DThreads (one per TFlux Kernel). The input size for *TRAPEZ* defines the number of these intervals.

Kyriakos Stavrou

As can be seen from Table 11 which summarizes the characteristics of *TRAPEZ*, the *basic task* is very small as it consists of only 461 dynamic instructions. To increase the granularity of the L-DThreads we unrolled the loops (unrolling leads to each L-DThread executing multiple times the *basic task*). According to the data of Table 11 unrolling the loop increases the number of dynamic instructions. However, this increase is slightly smaller compared to the product of the unroll factor and the size of the non-unrolled basic task due to optimizations performed by the compiler.

As for data accesses, TRAPEZ uses only scalar variables. This justifies the negligible miss rate (Table 11).

Finally, notice that the input size for *TRAPEZ* does not affect in any way the number of dynamic instructions or data accesses of the DThreads. This is due to the fact that the input size for this benchmark defines the number of invocations of the *basic task* and not the code it executes.

Table 11: *TRAPEZ* characteristics. Reported values are averaged over all invocations of the *basic task*. Miss rates correspond to the cache configuration presented in Table 9.

DThread	***Unroll factor***	***Dynamic Instructions***	***Data Accesses***		***Miss Rate***	
			L1D	***L2D***	***L1D***	***L2D***
1	1	461	102	< 1	$< 0.1\%$	$< 0.01\%$
	8	1849	431	< 1	$< 0.1\%$	$< 0.01\%$
	64	12877	3047	< 1	$< 0.1\%$	$< 0.01\%$

7.2.3 Susan Smoothing (SUSAN)

SUSAN is an image processing application and comes from the MiBench benchmark suite [44]. The operation of *SUSAN* that was used as a benchmark for the TFlux Evaluation Suite was the *smoothing()* function. This function applies a transformation to the picture given as input and operates in two steps. The first step regards the creation of a two dimensional mask that is used during the second step. The second step applies the smoothing transformation to the picture, which is also represented as a two dimensional array. The *basic task* of the first step is the creation of

Kyriakos Stavrou

an element of the mask. As for the second step, its *basic task* is the application of the smoothing filter to a pixel of the input picture.

The *basic task* of the first step works only on one element of the mask array, therefore each *basic task* can proceed in parallel. The same stands for the second step. However, as each operation of the second step requires *all* elements of the mask array, no iteration of the second loop may start its execution unless *all* iterations of the first loop have completed.

7.2.3.1 Porting SUSAN

The TFlux version of *SUSAN* consists of two parallel loops with a dependency between them (Figure 52-(a)). Each loop corresponds to each different step of the algorithm. The first loop corresponds to the creation of the mask and the second to the application of the smoothing filter to the picture. The Synchronization Graph of *SUSAN* can be expressed with two *ddm loop* directives as depicted in Figure 52-(b).

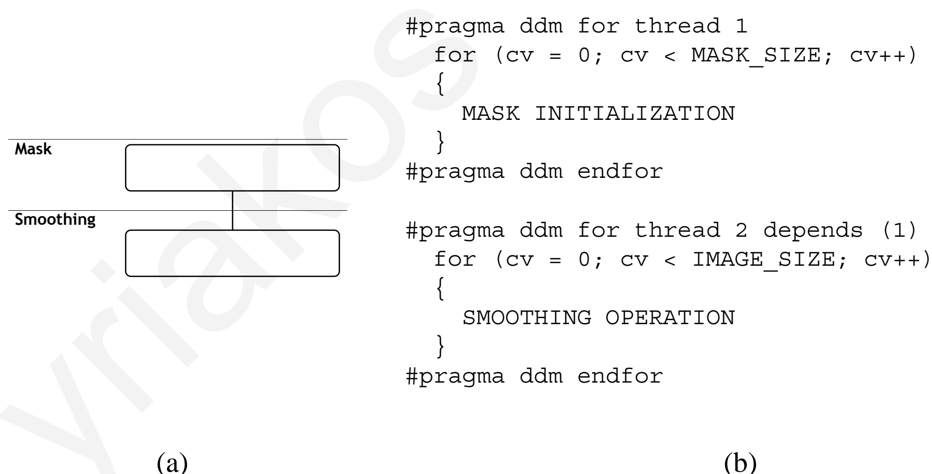


Figure 52: (a) The Synchronization Graph of SUSAN. (b) SUSAN benchmark parallelized using TFlux directives.

Kyriakos Stavrou

7.2.3.2 Dynamic Behavior

The number of DThreads for *SUSAN* is equal to the sum of the number of DThreads necessary for the execution of the two loops, *i.e.* it is equal to $MASK_SIZE + IMAGE_SIZE$. The first loop, the one that creates the mask, executes the same number of iterations regardless the size of the input picture. Regarding the loop that executes the smoothing operation, the number of its DThreads is equal to the number of elements of the output picture. As for the data accesses, although *SUSAN* operates on a large amount of data, the cache miss rate is low due to the exploitation of both temporal and spatial locality.

The size of the DThreads of the first loop (mask creation) is small but can be increased by unrolling. In particular, as can be seen by Table 12 that summarizes the characteristics of *SUSAN*, unrolling the loop 64 times increases the DThread's size from 459 to 12667 instructions. As for its data access pattern, it has low L1-data cache miss rate which is at the order of $< 5\%$ and is not affected by the unrolling.

Regarding the DThreads executing the smoothing operation (TFlux Loop 2), their size is approximately 10 times larger whereas its L1-data cache miss rate is less than 1%. Notice that the number of dynamic instructions executed by these DThreads does not increase with the input size. Instead, it is the number of the DThreads executing the smoothing operation that increases with the input size. Finally, althouth the miss rate for the L2-data cache appears to be high ($> 10\%$) this is due to the very small number of accesses.

7.2.4 Sorting using qSort (SORT)

The idea behind the *SORT* benchmark, which comes from the MiBench [44] suite, is to sort an array of numbers using the sorting function as a *"black box"*. The sequential version of this benchmark sorts the *whole* array using the system's *qsort()* function.

Kyriakos Stavrou

Table 12: *SUSAN* characteristics. Reported values are averaged over all executions of the *basic task*. Miss rates correspond to the cache configuration presented in Table 9.

DThread	*Unroll factor*	*Dynamic Instructions*	*Data Accesses L1D*	*Data Accesses L2D*	*Miss Rate L1D*	*Miss Rate L2D*
DThread 1: Mask creation operation						
1	1	459	98	4	3.7%	24.7%
	8	3426	742	35	4.6%	22.2%
	64	12667	2744	115	4.2%	22.2%
DThread 2: Smoothing operation						
2	1	7216	699	1	0.3%	12.7%
	8	57243	5506	2	0.3%	11.6%
	64	457110	43891	15	0.3%	11.8%

To perform this operation in parallel, the algorithm was modified to execute in two steps (Figure 53). First, the input array is partitioned in segments equal to the number of TFlux Kernels. Each Kernel sorts the subarray assigned to it using the system's *qsort()* function which has not been modified in any way. In the second step, the sorted subarrays are merged into the final output array using the merge-sort algorithm. Notice that the merge operation can be performed as a multilevel operation (Figure 53).

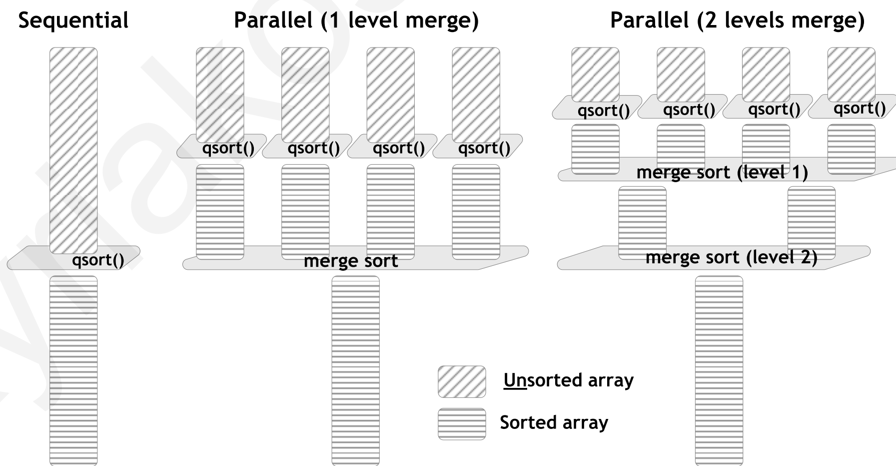


Figure 53: Operation of the parallel version of *SORT*.

Kyriakos Stavrou

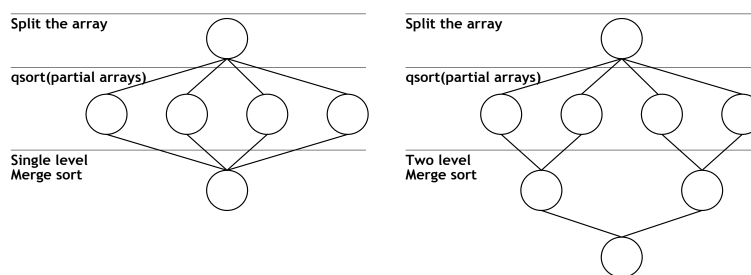


Figure 54: The Synchronization Graph of *SORT* for 4 TFlux Kernels.

7.2.4.1 Porting SORT

As can be seen from the Synchronization Graph of this benchmark (Figure 54) it consists only of dependent DThreads split in two groups. The DThreads of the first group execute the *qsort()* function on the subarray assigned to them whereas the second group's DThreads performs the *merge* operation. Figure 53 depicts the Synchronization Graphs of *SORT* when the merge operation is done with one (left part of the Figure) and two levels (right part of the Figure).

As all DThreads of the first group perform the same operation they can be expressed using the *ddm thread kernel all* directive which defines a DThread executed by all TFlux Kernels (Figure 55). As for the DThreads of the second group that performs the merge operation, they are expressed using the *#pragma ddm thread* directive with the dependencies being expressed with the *depends* statement.

As each such DThread operates on a different segment of the initial array each DThread first identifies the segments that correspond to it. The merge operation is performed by the DThreads of the second group. Each such DThread is defined to depend on the corresponding DThreads of the first group.

To parallelize this benchmark it was necessary to include operations that partition the initial array as well as code to perform the merge operation. In addition, a temporary array was needed

Kyriakos Stavrou

```
#pragma ddm thread 1 kernel all
  myL = findLowerBoundMyArray();
  myU = findUpperBoundMyArray();
  qSort(array, myL, myU-myL);
#pragma ddm thread

#pragma ddm thread 2 depends(1)
  outputArray = merge();
#pragma ddm endthread
```

(a)

```
#pragma ddm thread 1 kernel all
  myL = findLowerBoundMyArray();
  myU = findUpperBoundMyArray();
  qSort(array, myL, myU-myL);
#pragma ddm thread

#pragma ddm thread 2 depends(1/0,...)
  tempArray1 = merge(...);
#pragma ddm endthread

#pragma ddm thread 3 depends(..., 1/k)
  tempArray2 = merge(...);
#pragma ddm endthread

#pragma ddm thread 4 depends(2, 3)
  outputArray = merge();
#pragma ddm endthread
```

(b)

Figure 55: The code of *SORT* parallelized using TFlux directives. (a) Single level merging. (b) Two levels merging.

to store the sorted subarrays after the execution of the DThreads of the first group. This temporary array was the input to the DThreads of the second group, which write the final output to the memory that stored the input array.

7.2.4.2 Dynamic Behavior

The number of DThreads for this benchmark is small and independent of the input size. In particular, the sort phase has as many DThreads as the number of TFlux Kernels whereas for the merge operation the number of DThreads is smaller (1 DThread for single level merging and $1 + k/2$ for two levels merging where k is the number of TFlux Kernels). For *SORT* the input size describes the number of elements in the array to be sorted.

As can be seen from Table 13, that summarizes the characteristics of *SORT*, the size of the DThreads performing the partial sort operation is very large, in particular in the order of millions of instructions. Notice that this size increases with the input size (3M, 6M and 17M for the small, medium and large input sizes respectively). As for the DThreads that perform the merge operation,

Kyriakos Stavrou

although their size is approximately one order of magnitude smaller than the size of the DThreads that perform the partial sort, they are also very large (500K, 1.1M and 2.8M instructions for the small, medium and large input sizes respectively).

The data access pattern of the DThreads performing the sort operation depends by the implementation of the system's *qsort* function. According to our results the L1 data cache miss rate is less than 4%. As for the DThreads performing the merge operation, accesses to the different arrays are done on successive elements exploiting a high degree of spatial locality leading to an L1 data cache miss rate of 4.2%.

Table 13: *SORT* characteristics. Reported values are averaged over all executions of each DThread. Miss rates correspond to the cache configuration presented in Table 9.

DThread	***Dynamic Instructions***	***Data Accesses***		***Miss Rate***	
		L1D	***L2D***	***L1D***	***L2D***
Small Size: 10K					
qsort()	3045162	341825	12498	3.7%	3.9%
Merge Sort	578625	164801	6904	4.2%	11.2%
Medium Size: 20K					
qsort()	6480140	725623	27081	3.7%	2.9%
Merge Sort	1152938	329762	13890	4.2%	11.9%
Large Size: 50K					
qsort()	17694138	1989383	78345	3.9%	2.1%
Merge Sort	2877869	823733	34076	4.1%	13.1%

7.2.5 Runge-Kutta (RK)

Runge Kutta *(RK)* is a numeric method for solving initial value problems for ordinary differential equations. The implementation used as a benchmark is the *fourth-order Runge-Kutta formula* which is a four step process. Each step consists of a *parallel loop* which calculates a value necessary for *all* iterations of the subsequent loop (Figure 56-(a)). As such, for an iteration of a loop to start its execution *all* iterations of the previous loop must complete.

Kyriakos Stavrou

7.2.5.1 Porting RK

As shown in (Figure 56-(b)) that depicts the TFlux code of *RK*, parallelization was done using using *ddm for* directives. As explained earlier, for *RK*, all iterations of a loop need to complete before an iteration of the following loop can start its execution. Notice that this applies to all TFlux Loops of *RK*, therefore none of the synchronization barriers can be removed.

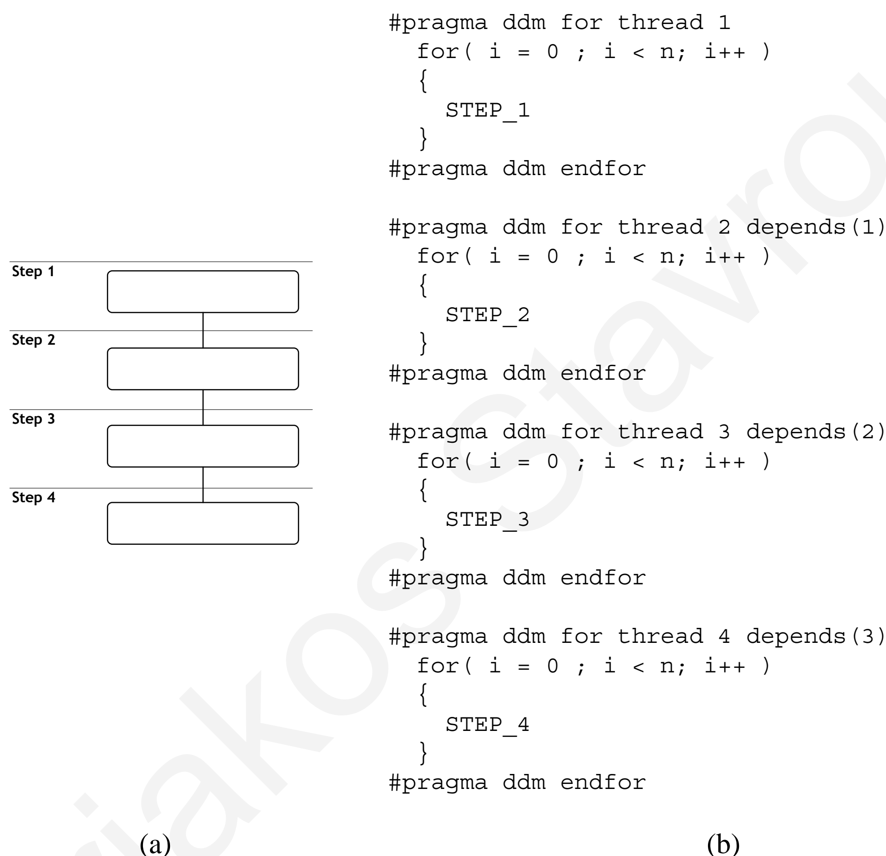


Figure 56: (a) The Synchronization Graph of RK. (b) RK parallelized using TFlux directives.

7.2.5.2 Dynamic Behavior

The algorithm operates on a set of regular data structures; a 2-D matrix is used to describe the input, one vector for the output and one vector for each step of the RK method. This leads to a medium to high pressure on the memory hierarchy.

Kyriakos Stavrou

The number of DThreads for *RK* depends on the input size, which is a function of the number of elements in each direction of the array describing the system. In particular, each of the four loops has one DThread per input element, leading to a total number of DThreads that is four times larger than the input size. The size of each DThread, in terms of the number of dynamic instructions, is also dependent on the input size, as each DThread contains a loop with as many iterations as the size of the input.

The DThreads of the last 3 TFlux Loops have identical dynamic characteristics as the code they execute is very similar. The size of these DThreads is rather large. In particular, for the small input size (1024) it is approximately 13K, for the medium size (2048) approximately 26K and for the large size (4096) approximately 52K instructions. As for the DThreads of the first TFlux Loop, the number of dynamic instructions and accesses to the data cache is somehow smaller (by approximately 15%). As shown in Table 14, unrolling these loops leads to an increase in the size of the DThreads by a factor that is slightly smaller compared to the number of times the loop is unrolled. This is due to optimizations performed by the compiler.

As for the L1 data cache miss rate for RK it is different for the two TFlux Loop groups. In particular, for TFlux Loop 1 it ranges from 4.4% for the small input size to 8.4% for the large input size. As for TFlux Loops 2-4, the L1 data cache miss rate ranges from 3.5% to 9.5%. Notice that the miss rate, is not affected by the loop unroll factor due to the fact that consecutive iterations of the loop reuse the same data as such exploiting high temporal locality.

7.2.6 Fast Fourier Transformation (FFT)

FFT computes the Discrete Fourier Transformation (DFT) which is widely used in several fields such as digital signal processing and electromagnetics. The implementation of *FFT* used for evaluating TFlux contains the computational kernel of a 3-D FFT-based spectral method. This

Kyriakos Stavrou

Table 14: *RK* characteristics. Reported values are averaged over all executions of each DThread. Miss rates correspond to the cache configuration presented in Table 9.

DThread	***Unroll factor***	***Dynamic Instructions***	***Data Accesses***		***Miss Rate***	
			L1D	***L2D***	***L1D***	***L2D***
Small Size: 1024						
1	1	13096	3182	140	4.4%	23.6%
	8	89693	25343	1125	4.4%	23.2%
	64	704998	202643	8983	4.4%	23.3%
2-4	1	15592	4132	145	3.5%	22.5%
	8	109924	32990	1106	3.4%	23.4%
	64	865362	263701	8830	3.3%	23.4%
Medium Size: 2048						
1	1	26022	6337	350	5.5%	18.6%
	8	178846	50579	2710	5.4%	19.2%
	64	1399760	404287	21619	5.3%	19.3%
2-4	1	34471	8926	823	9.2%	8.1%
	8	219534	65879	6255	9.5%	8.2%
	64	1725610	526830	49987	9.5%	8.2%
Large Size: 4096						
1	1	51886	12654	1064	8.4%	12.2%
	8	356925	101088	8484	8.4%	12.3%
	64	2797790	808505	67777	8.4%	12.3%
2-4	1	62076	16480	1565	9.5%	8.3%
	8	438438	131702	12479	9.5%	8.3%
	64	3448890	1053330	99744	9.5%	8.3%

benchmark comes from the NAS Parallel Benchmarks suite [13] and operates on a 3-D matrix of complex numbers. The version on which the TFlux FFT was based on is the OpenMP implementation of the NAS FFT [46].

Although the whole benchmark has been ported to TFlux, the analysis focuses on the *fft()* function which is the component that performs the useful computation. This function calls three other routines (*cffts1*, *cffts2* and *cffts3*) which apply the 1D FFT transformation to each of the three dimensions of the 3-D input matrix. Each such function calls the *cfftz* routine which performs one variant of the Stockham FFT.

Kyriakos Stavrou

7.2.6.1 Porting FFT

To port *FFT*, the *cffts1*, *cffts2* and *cffts3* routines have been inlined in the body of the calling function. The largest portion of the execution time for these routines is spent calling *cfftz*. Notice that the different invocations of the *cfftz* by *cfftz1*, *cfftz2* and *cfftz3* can proceed in parallel. This routine is invoked multiple times during the execution of *cffts1*, *cffts2* and *cffts3*. As such, parallelization has been done at this level, *i.e.* at the invocation of *cfftz*.

Figure 57-(a) depicts the Synchronization Graph for the whole *FFT* benchmark with the *fft* function shown shaded. *DThreads* 4, 5, 6 and 7 perform initializations and precalculations necessary for the execution of the TFlux Loops that follow. As can be seen from the Synchronization Graph, the calls to *cfftz* corresponding to *cffts1* and *cffts2*, which have been included in the same loop body (TFlux Loop 8), can be executed in parallel whereas *cfftz* calls for *cffts3* (TFlux Loop 9) are dependent on those for *cffts1* and *cffts2*. As such, it was not possible to remove the synchronization barrier between these loops. The TFlux code corresponding to the *fft* function is depicted in Figure 57-(b).

7.2.6.2 Dynamic Behavior

FFT operates on 3-D complex number matrices which leads to a large number of data accesses resulting in a medium to high miss rate. The benchmark has a medium number of DThreads that depends on the input size. In particular, it executes 11 simple DThreads and 8 TFlux Loops with the number of iterations for these loops being at the order of $O(n^2)$, where n is the size of the input. As for the *fft* function, it consists of 4 DThreads and 2 TFlux Loops leading to a very limited number of DThreads. In particular, the *FFT* function consist of 32 DThreads for the small input size, 64 DThreads for the medium input size and 128 DThreads for the large input size.

Kyriakos Stavrou

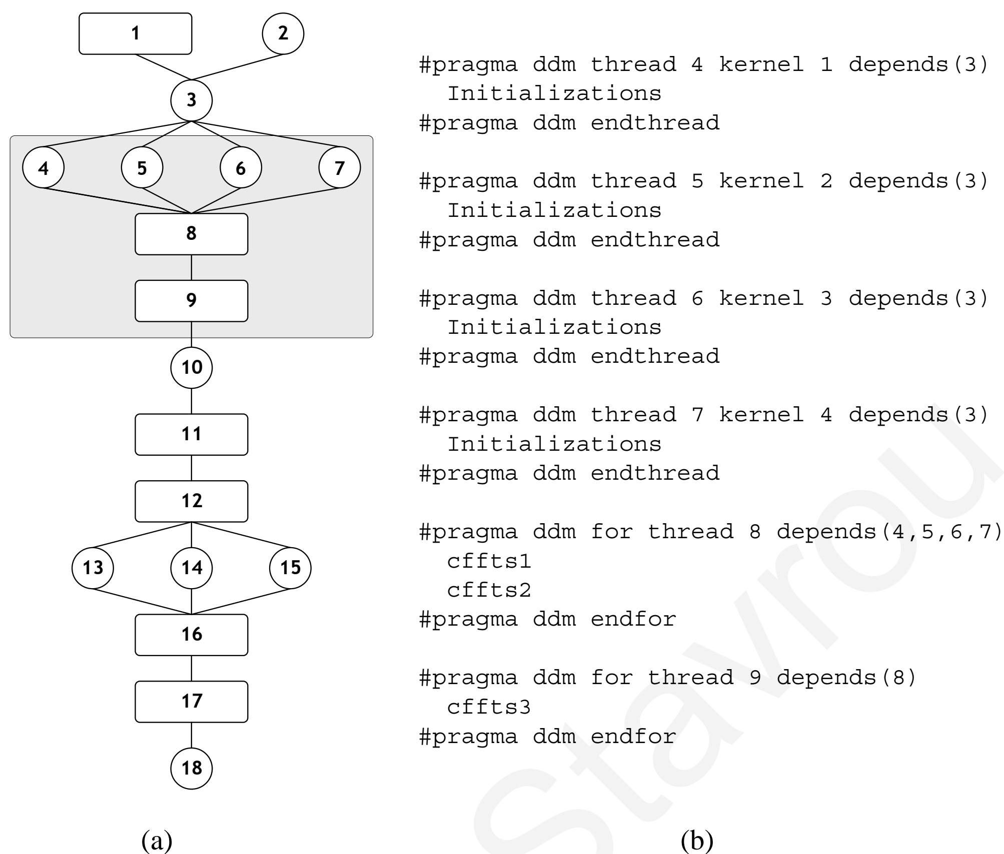


Figure 57: (a) The Synchronization Graph of the FFT benchmark. The *fft()* function is shown as shaded. (b) The *fft()* function parallelized using TFlux directives.

As can be seen from Table 15, the L-DThreads of this application consist of coarse-grained threads in the order of tenths of thousands of instructions. This is why unrolling the loop body for *FFT* was not necessary. As for the L1 data cache miss rate it is in the order of 10-15%.

7.2.7 Conjugate Gradient Method (CG)

CG comes from the NAS Parallel Benchmarks Suite [13] and computes an approximation of the smallest eigenvalue of a large, sparse, unstructured matrix using a Conjugate Gradient method. The code used for the porting *CG* to TFlux was based on the OpenMP implementation of *CG* [46].

The core of *CG* consists of a section of code in the *conj_grad* routine that is repeated multiple times to increase the accuracy of the approximation. This code mainly consists of a number loops

Kyriakos Stavrou

Table 15: *FFT* characteristics. Reported values are averaged over all executions of the basic task. Miss rates correspond to the cache configuration presented in Table 9.

DThread	*Dynamic Instructions*	*Data Accesses*		*Miss Rate*	
		L1D	*L2D*	*L1D*	*L2D*
Do not depend on input size					
4	43373	6825	524	7.7%	25.0%
5	6713	351	81	23.7%	0.0%
6	7740	490	106	21.6%	22.6%
7	6668	348	32	9.2%	28.1%
13	898	100	40	40.0%	15.0%
14	820	101	37	36.7%	13.6%
15	177	15	0	0.0%	0.0%
17	190	34	1	2.0%	15.7%
18	7285	461	67	14.5%	19.4%
Depend on input size					
- SMALL: 32x32x32					
1	48051	7455	113	1.5%	5.9%
2	20934	2997	289	9.7%	14.9%
3	2652882	91815	13808	15.3%	1.3%
8	231996	34694	4699	13.5%	0.2%
9	132859	18403	2548	13.8%	0.0%
10	2239316	72463	9121	12.6%	0.5%
11	19662	5605	510	9.9%	11.7%
12	116738	17410	2390	13.7%	0.5%
16	246422	35507	4875	13.7%	0.3%
- MEDIUM: 64x64x64					
1	258633	34123	4855	14.2%	0.51%
2	61984	8698	836	9.6%	10.9%
3	21239651	734095	110160	15.0%	0.4%
8	957652	138223	19241	13.9%	1.4%
9	549655	73635	10944	14.9%	0.0%
10	17779590	558788	71353	12.8%	0.5%
11	77099	21858	2857	13.6%	11.6%
12	493075	70058	10582	15.1%	2.5%
16	1017020	141585	20092	14.2%	1.4%
- LARGE: 128x128x128					
1	1044410	136409	27464	20.1%	1.48%
2	239734	36277	3292	9.7%	8.6%
3	169905122	5893026	872357	14.8%	0.5%
8	4682070	686179	105682	15.4%	1.4%
9	2618790	360681	52969	14.7%	0.0%
10	141598044	4359210	550108	12.6%	0.4%
11	305788	86408	15080	17.5%	8.8%
12	2402180	348239	52007	14.9%	2.1%
16	4908200	698518	109369	15.7%	1.1%

Kyriakos Stavrou

and reduction operations. After the desired accuracy has been reached, *CG* calculates a number of other metrics and terminates its execution.

The parallel loops of the section that is repeated have two important characteristics. First, these loops depend on each other. However, this dependency is not at the level of the whole loop however but rather at the level of the *loop iterations*. This means that an iteration of the Consumer loop does *not* need to wait for *all* iterations of the producer loop to complete before it initiates its execution. Second, the different iterations of these loops do not require the same execution time, which may lead to load imbalance problems.

7.2.7.1 Porting CG

As depicted in Figure 58 the Synchronization Graph *CG* mainly consists of loops. Notice that the execution of a part of this graph (DThreads 3-9) is repeated multiple times.

The first step for porting *CG* was to inline code of the *conj_grad* routine. Most loops in this function, as well as most loops of the rest of the program, were parallelized by expressing them as TFlux Loops using the *ddm for* directive (Figure 59). As for the five loops that perform reduction operations (TFlux Loops 2, 5, 7, 11 and 12) they were expressed using a *ddm for reduction* directive. Regarding the portion of the Synchronization Graph executed multiple times (DThreads 3-9), the directives expressing the *recycle operation* (Section 3.2.3) were used. Finally, for the iteration-level dependencies that exist between loops 4-5 and 6-7 they have been expressed through *ilc* statements (Section 4.3.2.5).

7.2.7.2 Dynamic Behavior

The *CG* benchmark operate on vectors, which are accessed sequentially. Although the size of these vectors is large, the locality of these accesses lead to a rather small miss rate (approximately 5% for the L1 data cache). This benchmark executes a large number of DThreads and this is

Kyriakos Stavrou

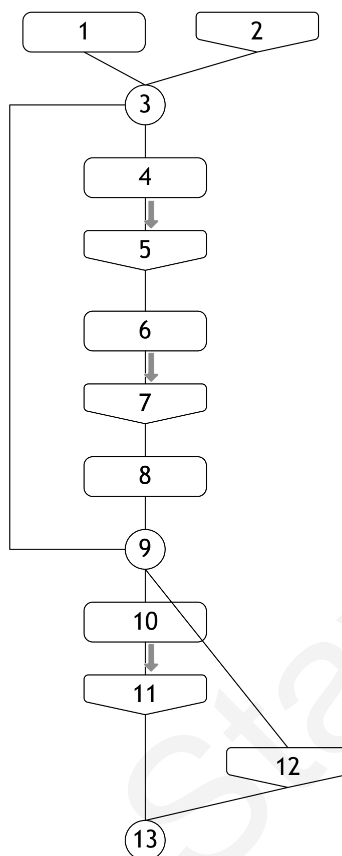


Figure 58: The Synchronization Graph of CG.

mainly due to the fact that part of the Synchronization Graph that is repeated consists mainly of TFlux Loops.

As can be seen from Table 16, these DThreads are very fine grained as they include only a small number of instructions. As such, in order to amortize the parallelization overheads it is necessary for these loops to be unrolled. However, due to the simplicity of the operations inside the DThreads, for all cases except DThreads 4 and 10, unrolling does not increase significantly their size. As an example, the size of DThreads increases only by a factor of 4.6 (from 81 to 369 instructions) after unrolling the body of the loop 64 times. This is due to optimizations performed by the compiler.

Kyriakos Stavrou

```
#pragma ddm for thread 1
  initializations
#pragma ddm endfor

#pragma ddm for thread 2
  rho=||r||
#pragma ddm endfor

#pragma ddm thread 3 kernel 1 depends(1,2) recycle
  if(tempCgit>cgitmax)
    #pragma ddm threadCompleted
  else
    iteration initializations
#pragma ddm recycle

#pragma ddm for thread 4 depends(3) recycle
  q = A.p
#pragma ddm endfor

#pragma ddm for thread 5 reduction var01 + double d depends(4) recycle
  Obtain p.q
#pragma ddm endfor

#pragma ddm for thread 6  depends(5) recycle
  Calculate: z = z + alpha*p
  Calculate: r = r - alpha*q
#pragma ddm endfor

#pragma ddm for thread 7 reduction var01 + double rho depends(6) recycle
  rho = r.r; beta = rho/rho0;
#pragma ddm endfor

#pragma ddm for thread 8 depends(7) recycle
  Calculate: p = r + beta*p
#pragma ddm endfor

#pragma ddm thread 9 kernel 1 depends(8) recycle 3
#pragma ddm recycle

#pragma ddm for thread 10 depends(9)
  ||r|| = ||x - A.z||
#pragma ddm endfor

#pragma ddm for thread 11 reduction reductionVar01 + double sum depends(10)
  Calculate: sum += (x[j] - r[j]).(x[j] - r[j]);
#pragma ddm endfor

#pragma ddm for thread 12 reduction reductionFunction(...) depends(9)
  Calculate: norm11 += x[j]*z[j];
  Calculate: norm12 += z[j]*z[j];
#pragma ddm endfor

#pragma ddm thread 13 kernel 1 depends(11, 12)
  Calculate final metrics
#pragma ddm endthread
```

Figure 59: *CG* parallelized with TFlux directives *without* exploiting iteration-level dependencies.

7.2.7.3 Removing the barriers

As can be seen from the Synchronization Graph of *CG* (Figure 58) the TFlux Loops have barriers between them. However, using TFlux Iteration-Level Dependencies (Section 4.3.2.5) it

Kyriakos Stavrou

Table 16: *CG* characteristics. For DThreads executed multiple times, reported values are averaged over all invocations. The configuration of caches is presented in Table 9.

DThread	***Unroll factor***	***Dynamic Instructions***	***Data Accesses***		***Miss Rate***	
			L1D	***L2D***	***L1D***	***L2D***
1	1	203	33	1	2.4%	7.9%
	8	918	109	6	5.5%	12.4%
	64	3211	287	20	6.7%	8.4%
2	1	79	16	1	0.2%	0.0%
	8	103	23	1	1.6%	3.2%
	64	588	92	2	1.4%	4.8%
3	-	815	72	5	5.9%	11.0%
4	1	730	193	12	6.2%	0.2%
	8	5326	1448	94	6.4%	0.9%
	64	41965	11453	733	6.4%	0.5%
5	1	81	18	1	1.4%	0.3%
	8	113	32	2	3.7%	0.0%
	64	369	150	9	5.7%	0.0%
6	1	94	25	1	3.8%	0.0%
	8	183	66	5	6.9%	0.0%
	64	909	412	20	4.8%	0.0%
7	1	79	16	1	0.6%	0.1%
	8	102	23	1	3.4%	0.0%
	64	300	85	1	0.2%	6.0%
8	1	83	19	1	1.1%	0.3%
	8	128	40	2	2.9%	0.0%
	64	500	215	2	0.9%	0.0%
9	-	62	11	0	0.0%	0.0%
10	1	723	191	12	6.2%	0.1%
	8	9688	2321	136	5.8%	3.8%
	64	42144	11428	736	6.4%	0.3%
11	1	84	18	1	2.2%	0.3%
	8	123	32	3	6.6%	0.0%
	64	413	147	16	10.7%	0.0%
12	1	86	19	1	1.1%	0.0%
	8	132	33	3	6.3%	0.0%
	64	482	148	3	2.0%	0.0%
13	-	25973	4525	247	5.5%	26.3%

was possible to remove three out of the five barriers. As for the other two barriers, *i.e.* between TFlux Loops 5 and 6 and between 7 and 8, this technique could not be applied as the Producer loops (TFlux Loops 5 and 7) perform a reduction operation, which is necessary for each iteration of the Consumer loops (TFlux Loops 6 and 8). As the reduction result will be ready only after *all*

Kyriakos Stavrou

iterations of the loop have completed the dependencies between these loops can not be expressed at the iteration level.

Figure 60 presents the operation of TFlux Loops 6 and 7 and will be used to better explain how the barrier dependence has been expressed in terms of iteration level dependencies. As can be seen from this Figure, each iteration of TFlux Loop 6 calculates the value of an element of vectors w and q whereas each iteration of TFlux Loop 7 uses the value of an element of vector q. As such, an iteration of TFlux Loop 7 can start its execution whenever the corresponding TFlux Loop 6 iteration it depends on has completed. Another observation that can be made from the code depicted in Figure 60 is that the load of each iteration loop 6 depends on the value of the control variable.

```
//Code of TFlux Loop 6
for (j = 1; j < inputSize; j++)
{
  tempDouble = 0.0;
  for (k = rowstr[j]; k < rowstr[j+1]; k++)
  {
    tempDouble = tempDouble + a[k]*p[colidx[k]];
  }
  w[j] = tempDouble;
  q[j] = w[j];
  w[j] = 0.0;
  tempDouble=0.0;
}

//Code of TFlux Loop 7
for (j = 1; j < inputSize; j++)
{
  reductionVar01 = reductionVar01 + p[j]*q[j];
}
```

Figure 60: The code of TFlux Loops 6 and 7 which have dependencies at the iteration level.

The TFlux code of *CG* that exploits the iteration-level dependencies is depicted in Figure 61.

7.2.8 LU

As the previous benchmark, LU was also selected from the NAS Parallel Benchmarks Suite [13]. LU is a simulated CFD application that uses symmetric successive over-relaxation (SSOR) method

Kyriakos Stavrou

```
#pragma ddm for thread 1
  initializations
#pragma ddm endfor

#pragma ddm for thread 2
  rho=||r||
#pragma ddm endfor

#pragma ddm thread 3 kernel 1 depends(1,2) recycle
  if(tempCgit>cgitmax)
    #pragma ddm threadCompleted
  else
    iteration initializations
#pragma ddm recycle

#pragma ddm for thread 4 depends(3) ilc [1 5 1 0] recycle
  q = A.p
#pragma ddm endfor

#pragma ddm for thread 5 reduction var01 + double d readyCount 1 recycle
  Obtain p.q
#pragma ddm endfor

#pragma ddm for thread 6  depends(5) ilc [1 7 1 0] recycle
  Calculate: z = z + alpha*p
  Calculate: r = r - alpha*q
#pragma ddm endfor

#pragma ddm for thread 7 reduction var01 + double rho readyCount 1 recycle
  rho = r.r; beta = rho/rho0;
#pragma ddm endfor

#pragma ddm for thread 8 depends(7) recycle
  Calculate: p = r + beta*p
#pragma ddm endfor

#pragma ddm thread 9 kernel 1 depends(8) recycle 3
#pragma ddm recycle

#pragma ddm for thread 10 depends(9) ilc [1 11 1 0]
  ||r|| = ||x - A.z||
#pragma ddm endfor

#pragma ddm for thread 11 reduction reductionVar01 + double sum readyCount 1
  Calculate: sum += (x[j] - r[j]).(x[j] - r[j]);
#pragma ddm endfor

#pragma ddm for thread 12 reduction reductionFunction(...) depends(9)
  Calculate: norm11 += x[j]*z[j];
  Calculate: norm12 += z[j]*z[j];
#pragma ddm endfor

#pragma ddm thread 13 kernel 1 depends(11, 12)
  Calculate final metrics
#pragma ddm endthread
```

Figure 61: *CG* parallelized with TFlux directives exploiting iteration-level dependencies.

to solve a seven-block-diagonal system resulting from finite-difference discretization of the Navier-Stokes equations in 3-D by splitting it into block Lower and Upper triangular systems [46].

Kyriakos Stavrou

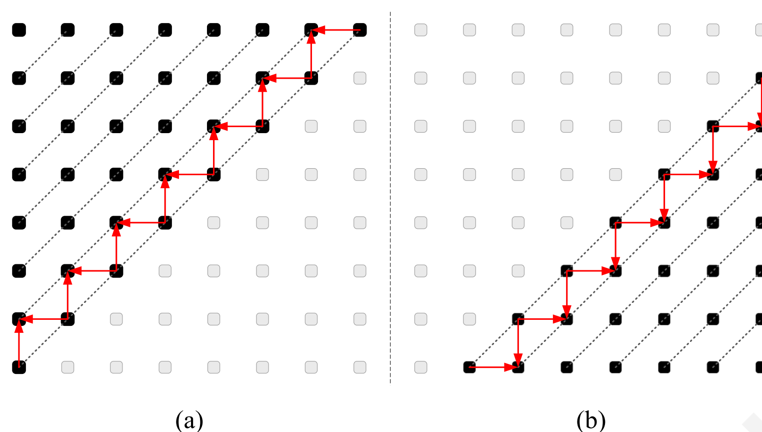


Figure 62: The dependencies of the iterations of loops 4 and 8.

The core of LU is split into two phases. The first forms and solves the lower-triangular and diagonal systems, which are represented by two-dimensional arrays, whereas the second phase forms and solves the upper-triangular system. Referring to the Synchronization Graph of the LU benchmark (Figure 63), the first phase is executed by Loops 3 and 4 whereas the second phase by Loops 7 and 8. Notice that each pair of these loops (3-4 and 7-8) needs to be executed multiple times until the desired accuracy is reached.

The loops solving the two systems (4 and 8), present a particularity regarding their parallelization process. More specifically, each iteration of these loops processes only one element of the system. The data-dependencies between these operations are such that processing an element can start only if processing the element in the previous row and the previous column has completed (Figure 62-(a) and 62-(b)). The result of these dependencies is that the operations that can be executed concurrently are those lying in the diagonals of the array.

Kyriakos Stavrou

7.2.8.1 Porting LU

As can be seen from Figure 63, which depicts the Synchronization Graph of LU, this benchmark consists only of loops. As for DThreads 2, 5, 6 and 9, they are used to control the recycling operation, which is necessary for repeating the execution of TFlux Loops 3, 4, 7 and 8. As for the functions contained in these loops, in order to port the application to TFlux, it was necessary to inline them in the body of the *main()* function.

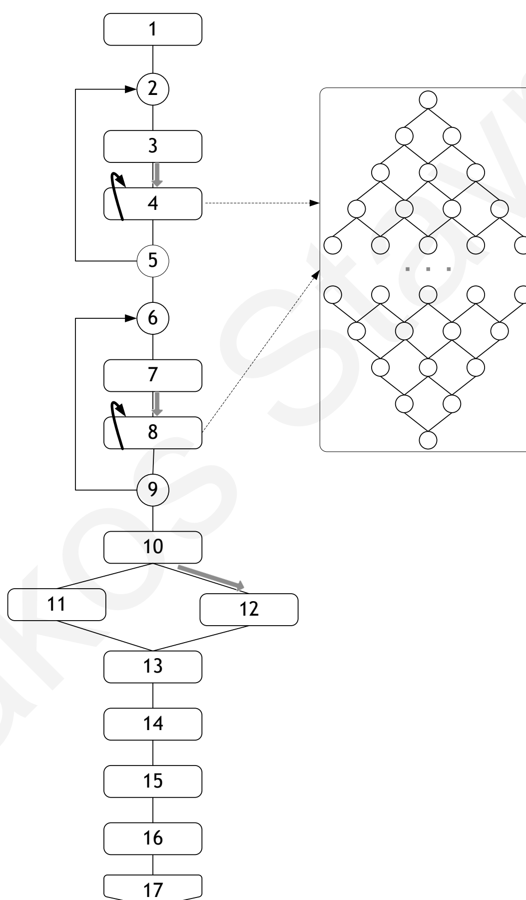


Figure 63: The Synchronization Graph of LU.

Kyriakos Stavrou

What deserves further explanation is the way Loops 4 and 8 were parallelized using TFlux directives. As explained earlier (Figure 62) the iterations of these loops are not independent.

More specifically, for an iteration to start its execution the iteration of the previous row and that of the previous column are required to have completed.

As depicted in Figure 64-(a) that depicts the OpenMP version of the application [46] on which we based the TFlux version a *#pragma omp for* is used in order to express that iterations are to be executed concurrently whereas proper code is used at the start and end of the loop body to enforce the necessary synchronization. The first instructions of the loop body examine if the producer iterations (previous row and previous column) have completed. This is done by checking a flags array. If not, the processor executing this iteration spins until the iterations on which it depends complete. When an iteration finally executes it sets the corresponding elements of the flags array so that its consumer iterations can proceed.

In TFlux, however, these dependencies can be enforced implicitly without any additional code for synchronization. As can be seen from Figure 64-(b) that depicts the TFlux version of the code of these loops, these dependencies can be expressed using the appropriate *ilc* statement. What deserves further explanation, however, is the scheduding model selected for these loops.

As explained in Section 4.3.2.1, TFluxCpp provides two types of scheduling for the iterations of a TFlux Loop. The first, *chunk scheduling*, assigns a number of consecutive iterations to each processor whereas the second, *Round-Robin scheduling*, assigns consecutive iterations to different processors. Figure 65-(a) depicts the execution of Loops 4 and 8 on a 2 CPUs system with *chunk scheduling* (chunk size is equal to 32) whereas Figure 65-(b) with Round-Robin scheduling. As can be seen from this Figure, chunk scheduling decreases significantly the exploitable parallelism as it splits execution into phases assigned to different processors. As such, for Loops 4 and 8 we selected the Round-Robin scheduling (*schedule 1*).

Kyriakos Stavrou

```
#pragma omp for
for( i = 0 ; i < NO_ITERATIONS; i++ )
{
//
  //Wait for producer iterations to complete.
  //
  // - Wait for producer 1
  while(producer[i/ITERS_PER_DIM]==0)
   ;
  // - Wait for producer 2
  while(producer[i%ITERS_PER_DIM]==0)
   ;

  //
  // Do the ''useful'' computation
  //

  //
  //Set the flags to ''wakeup'' the consumers
  //
  producer[i/ITERS_PER_DIM]=1;
  producer[i%ITERS_PER_DIM]=1;
}
```

(a)

```
#pragma ddm for thread 4 ilc[...][...]

    USEFULL CODE

#pragma ddm endfor
```

(b)

Figure 64: Parallelization of loops 4 and 8 using (a) OpenMP and (b) TFlux directives.

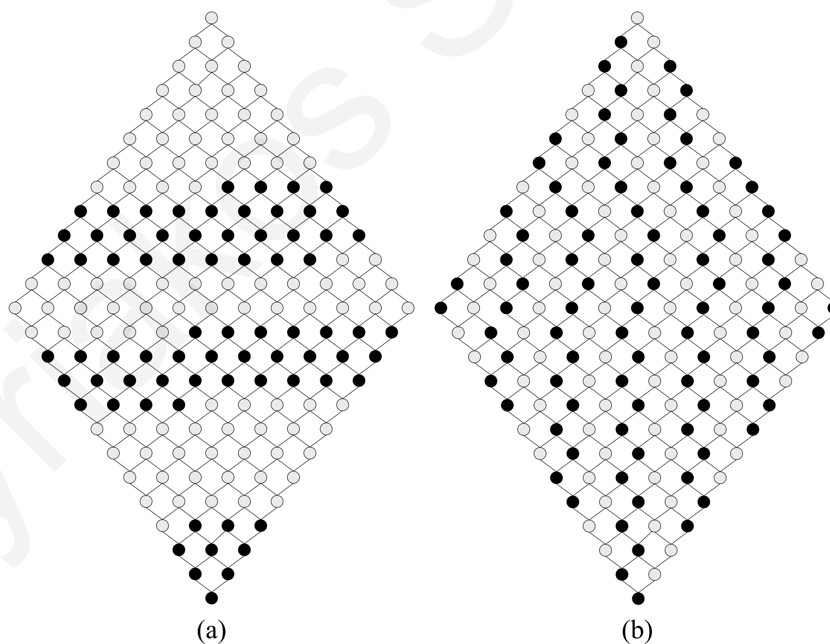


Figure 65: Scheduling types for Loops 4 and 8 of LU: (a) Chunk and (b) Round-Robin. Gray nodes are executed by CPU 0 and black nodes executed by CPU 1.

Kyriakos Stavrou

7.2.8.2 Dynamic Behavior

The input size of *LU* refers to the size of the system it solves. For our experimentation the small input size uses a 16×16 system , the medium size a system of 32×32 elements and finally the large size a system of 64×64 elements.

LU executes a large number of L-DThreads: approximately 13K for the small input size, 217K for the medium and 996K for the large input size. As can be seen from Table 17, the number of dynamic instructions executed by these DThreads is very small (approximately 830 instructions for DThreads 3 and 7 and approximately 470 instructions for DThreads 4 and 8) which leads to only partial amortization of the parallelization overheads. The number of data accesses, similarly to the number of dynamic instructions, is very small. This is why the L1 data-cache miss rate appears to be large. The only L-DThreads that have a large number of data-accesses and at the same time large miss rate (17% for the large input size) are those of TFlux Loop 13 which traverse a large number of arrays.

7.2.9 Summary

The 8 real-life applications of the TFlux Evaluation Suite that have been presented in this Section meet the criteria presented at the beginning of this chapter. As depicted in Figure 66 that summarizes the Synchronization Graphs of these applications, the complexity of these graphs ranges from a single parallel loop for *MMULT* to multiple loops with complex iteration level dependencies for *LU*.

As for the other characteristics of these applications, they are summarized in Table 18. The *static* number of TFlux Loops and DThreads ranges from only 1 for *MMULT* and *TRAPEZ* to 17 for *LU*. As for the number of dynamically executed DThreads, this ranges from as many as the number of Kernels for *SORT* to 2^{21} for *TRAPEZ*. These applications also differ significantly

Kyriakos Stavrou

Table 17: *LU* characteristics. Reported values are averaged over all executions of the basic task. Miss rates correspond to the cache configuration presented in Table 9.

DThread	***Dynamic Instructions***	***Data Accesses***		***Miss Rate***	
		L1D	***L2D***	***L1D***	***L2D***
Do not depend on input size					
2	187	30	5	13.93%	27.41%
3	836	353	34	5.7%	6.28%
4	462	221	22	6.58%	0.53%
5	61	11	0	0.0%	0.0%
6	191	29	5	14.92%	7.46%
7	825	355	30	4.48%	1.7%
8	482	234	24	7.1%	0.49%
9	61	11	0	0.0%	0.0%
Depend on input size					
- SMALL: 16x16					
1	1392	289	14	4.84%	27.12%
10	1358	347	25	6.94%	0.0%
11	1292	252	18	6.94%	20.41%
12	1107	332	17	4.97%	6.6%
13	7426	2748	136	4.94%	2.28%
14	869	313	13	3.89%	0.64%
15	6840	2452	96	3.88%	1.1%
16	7170	2551	42	1.64%	0.0%
17	660	187	11	5.79%	0.0%
- MEDIUM: 32x32					
1	2590	562	26	4.54%	24.21%
10	2680	702	47	6.64%	9.33%
11	2309	453	29	6.39%	21.49%
12	1985	646	29	4.47%	20.41%
13	22910	6341	539	8.48%	3.36%
14	1752	630	26	3.98%	11.20%
15	13873	5194	221	4.25%	6.65%
16	14939	5322	81	1.52%	2.25%
17	1279	381	23	5.89%	0.0%
- LARGE: 64x64					
1	5029	1115	52	4.59%	22.58%
10	5304	1420	94	6.56%	24.32%
11	4400	866	57	6.46%	22.73%
12	3787	1283	57	4.40%	23.32%
13	50553	13209	2306	17.45%	2.1%
14	3317	1271	50	3.92%	24.84%
15	31469	10871	746	6.85%	6.46%
16	30556	10875	168	1.53%	14.29%
17	2505	766	45	5.87%	25.22%

in terms of the stress they cause on the memory hierarchy. In particular, the suite includes applications with minimum (*TRAPEZ*), medium (*SORT*, *SUSAN*, *RK* and *LU*) and large data usage

Kyriakos Stavrou

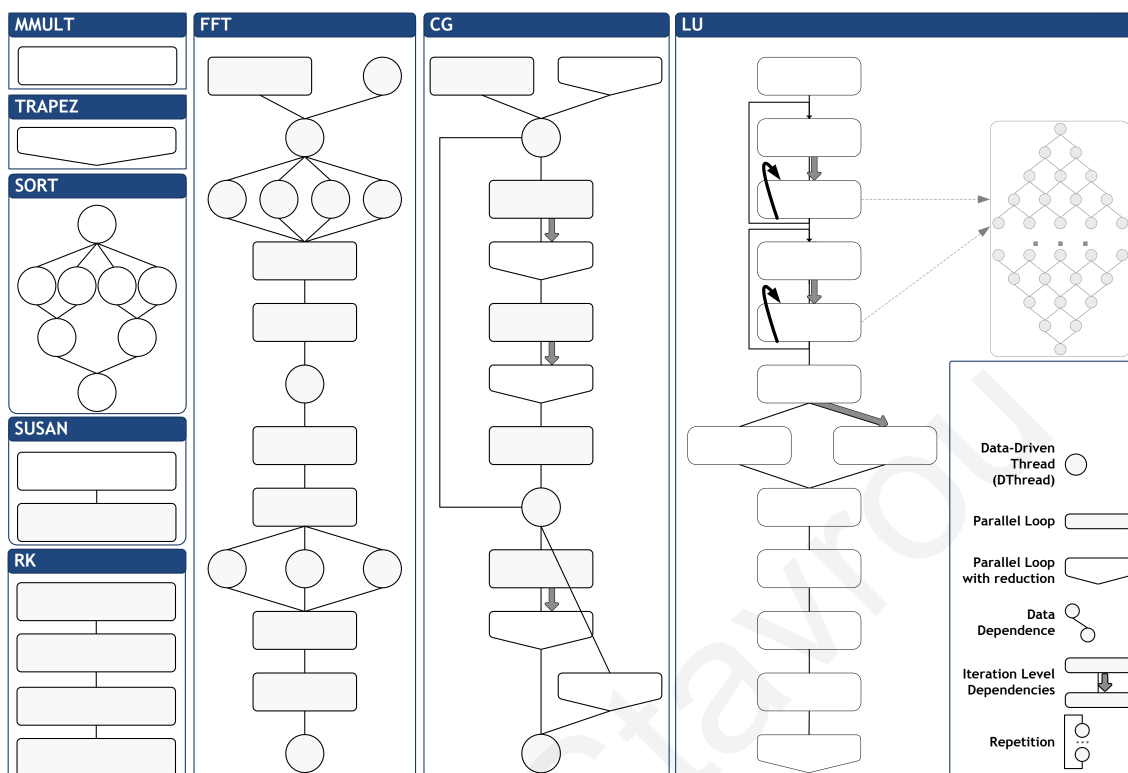


Figure 66: Synchronization Graphs of the real-life applications.

(*MMULT*, *RK*, *FFT*). The L1-data cache miss rate for these applications ranges from almost 0% for *TRAPEZ* to approximately 50% for *MMULT*. Regarding the DThreads granularity, *CG* and *LU* applications have fine-, *MMULT*, *TRAPEZ*, *SUSAN* and *RK* medium- and *SORT* and *FFT* coarse-grained DThreads.

7.3 Synthetic Applications

The synthetic applications presented in this Section allow the analysis of *specific characteristics* of TFluxHard and TFluxSoft isolating all other factors that may affect the performance. Such factors include the cache behavior and the communication delay. Isolating these factors is achieved by having the synthetic applications executing a parameterizable computational load that

Kyriakos Stavrou

Table 18: Summary of the characteristics of the real-life application of the TFlux Evaluation Suite. (N: number of Kernels)

Characteristic	***MULT***	***TRAPEZ***	***SORT***	***SUSAN***	***RK***	***FFT***	***CG***	***LU***
Synchronization Graph								
TFlux Loops	1	1	-	2	4	2	10	13
Static # DThreads	-	N	N+2	-	-	4	3	4
Porting to TFlux								
# Directives	1	1	2	-	4	7	13	17
Origination of the benchmark								
Kernel	$\surd$	$\surd$			$\surd$			
MiBench			$\surd$	$\surd$				
NAS						$\surd$	$\surd$	$\surd$
Input Sizes								
Small	64x64	2^{17}	10K	256x288	2048	32	2048	16x16
Medium	128x128	2^{19}	20K	512x576	4096	64	4096	32x32
Large	256x256	2^{21}	50K	1024x576	8192	128	8192	64x64
Dynamic DThreads								
Small	2^{12}	2^{17}+N	N+1	$\sim 2^{16}$	2^{13}	$\sim 2^{7}$	$\sim 2^{18}$	$\sim 2^{15}$
Medium	2^{14}	2^{19}+N	N+1	$\sim 2^{18}$	2^{14}	$\sim 2^{8}$	$\sim 2^{19}$	$\sim 2^{17}$
Large	2^{16}	2^{21}+N	N+1	$\sim 2^{19}$	2^{15}	$\sim 2^{9}$	$\sim 2^{20}$	$\sim 2^{19}$
Dynamic Data Behavior - Data Usage								
Low		$\surd$						
Medium			$\surd$	$\surd$	$\surd$			$\surd$
High	$\surd$					$\surd$	$\surd$	
Dynamic Data Behavior - L1 Data Cache Miss Rate								
Low		$\surd$	$\surd$	$\surd$				
Medium					$\surd$		$\surd$	$\surd$
High	$\surd$					$\surd$		
DThread Granulariry								
Fine Grained							$\surd$	$\surd$
Medium Grained	$\surd$	$\surd$		$\surd$	$\surd$			
Coarse Grained			$\surd$			$\surd$		

is composed of instructions that operate on *private scalar* variables only. This leads to negligible data cache miss rate and zero data transfers between the execution nodes.

To avoid situations where the compiler optimizes the instructions that comprise the computational load, these instructions have been included in a function called *load()* which has been compiled with all optimizations turned-off, *i.e.* with the *-O0* compilation flag. The resulting object file is then linked to the rest of the application. As for the size of the computational load, in terms of dynamic instructions, this is controlled by the number of times the *load()* function is

Kyriakos Stavrou

called (*effect*). As can be seen from Figure 67 that depicts the code executed by each DThreads. This code consinst of a loop that calls the *load()* function *effort* times.

```
for(cv=0;cv<effort;cv++)
{
  load();
}
```

Figure 67: Computational load executed by each DThread of the synthetic applications.

At the code level the *load()* function, as can be seen from Figure 68 that depicts its code in ANSI C, consist of multiple *i*++ instructions. Given that compilation is done without any optimization, each *i*++ instruction translates to three assembly instructions, a *load* operation to load variable *i*, an *increase* operation to increase the value of the variable and finally a *store* operation to save the result. Notice that the *load()* function operates on this scalar variable only (*i*) and therefore, with the exception of a pathological case, all memory accesses, except maybe for the first, will lead to a memory hit.

```
void load()
{
  i++; i++; i++; i++; i++; i++; i++; i++;
  i++; i++; i++; i++; i++; i++; i++; i++;
  i++; i++; i++; i++; i++; i++; i++; i++;
  ...
}
```

Figure 68: The code of the *load()* function.

Table 19 reports how many assembly instructions are executed for different *effort* values. As can be seen from the Table, the number of executed assembly instructions do not increase proportionally to the number of calls to the *load()* function. For example, calling the *load()* function once leads to 162 instructions whereas calling it 16 times leads to 1797 instead of 2592. This is due to compiler optimizations *on the code that calls* (and not the code *of*) the *load()* function.

Kyriakos Stavrou

Table 19: Number of dynamic assembly instructions in synthetic applications' computational load as a function of the value of the *effort* variable.

Calls to the load function (*effort*)	**Executed assembly instructions**
1	162
2	271
4	489
8	925
16	1797
32	3541
64	7029
128	15196
256	27957
512	57134
1024	112909
2048	225753
4096	451451

The Sections that follow present the different synthetic applications and explain their purpose. These synthetic applications are categorized in different groups according to the way they are used for the evaluation of TFlux.

7.3.1 Parallel Threads

The purpose of the *Parallel Threads* synthetic application is to quantify the parallelization overhead for TFluxHard and TFluxSoft. More specifically, by using this application it is possible to find the minimum size, in terms of dynamic instructions, a DThread needs to have in order to overcome the overheads of creating, synchronizing and handing it.

As can be seen from Figure 69-(a) that depicts the Synchronization Graph of this application, there is one DThread per TFlux Kernel and these DThreads do not have any dependencies between them. As for the *total computational load* executed by this application (*numFunctionCalls* calls to the *load()* function) it is divided between the parallel DThreads, *i.e.* each DThread executes $\frac{numFunctionCalls}{numKernels}$ calls to the *load()* function (care *has* been taken so that *numFunctionCalls* is multiple of *numKernels* which stands for the number of Kernels).

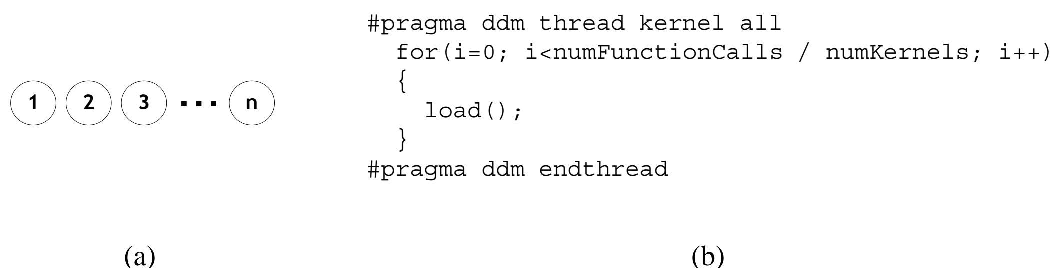


Figure 69: (a) The Synchronization Graph of the “Parallel Threads” synthetic application. (b) The DDM code of the “Parallel Threads” synthetic application.

The baseline for this application regards the execution of the same computational load, but sequentially. Figure 70 depicts the code of the baseline program.

```
for(i=0; i<numFunctionCalls; i++)
{
  load();
}
```

Figure 70: The baseline for the *Parallel Threads* synthetic application.

7.3.2 Basic Loops

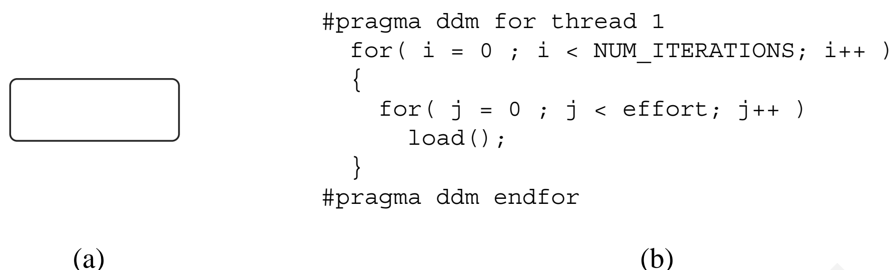
This set of synthetic applications focuses on the execution of parallel loops. In particular, it is used to evaluate the performance of TFluxHard and TFluxSoft on applications with parallel loop constructs. Each L-DThread of these loops executes a constant computational load and this load is the same for all the L-DThreads of the application. Each parallel loop executes 2048 iterations (*NUM_ITERATIONS*) as this number was found to be adequate for giving representative results for all experiments. The baseline for these applications is always the sequential execution of the same computational load.

7.3.2.1 Single loop: L1

L1 consists of a single parallel TFlux Loop. The Synchronization Graph of *L1*, which is depicted in Figure 71-(a), is a frequently-used component for a large number of applications such

Kyriakos Stavrou

as the *MMULT* (Section 7.2.1) and *TRAPEZ* (Section 7.2.2) benchmarks. The TFlux code for this application is depicted in Figure 71-(b).



```
#pragma ddm for thread 1
  for( i = 0 ; i < NUM_ITERATIONS; i++ )
  {
    for( j = 0 ; j < effort; j++ )
      load();
  }
#pragma ddm endfor
```

(a) (b)

Figure 71: (a) The Synchronization Graph of L1 (b) The TFlux code of L1.

The code of the baseline for *L1* is depicted in Figure 72. As can be seen from this Figure, the baseline application executes the same computational load sequentially.

```
for(i=0; i<NUM_ITERATIONS; i++)
  for(j=0; j<effort; j++)
    load();
```

Figure 72: The baseline for the *L1* synthetic application.

7.3.2.2 Two dependent loops: L2

L2 has two TFlux Loops which depend on each other not at the iteration level but rather at the level of the whole TFlux Loops. This Synchronization Graph (Figure 73-(a)) is identical to that of the *SUSAN* benchmark (Section 7.2.3). The TFlux code for this application is shown in Figure 73-(b).

The code of the baseline for *L2*, which executes the same computational load sequentially, is depicted in Figure 74.

Kyriakos Stavrou

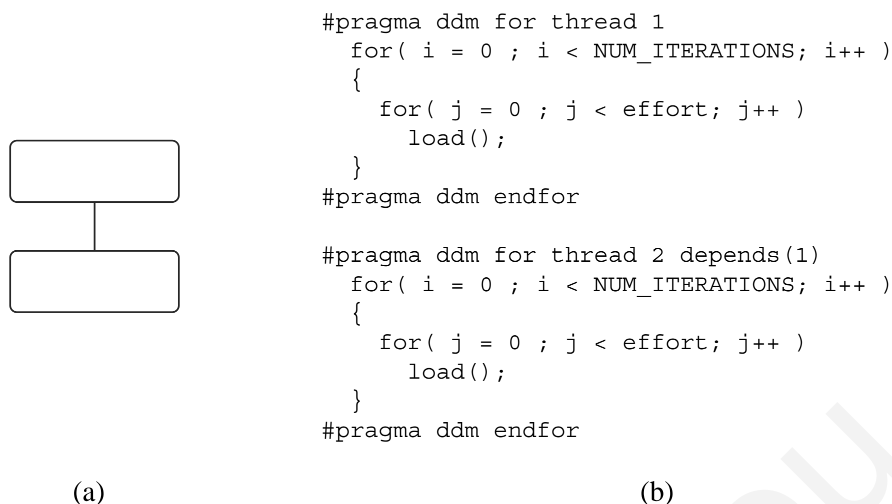


Figure 73: (a) The Synchronization Graph of L2. (b) The TFlux code of L2

```
for(i=0; i<NUM_ITERATIONS; i++)
  for(j=0; j<effort; j++)
    load();

for(i=0; i<NUM_ITERATIONS; i++)
  for(j=0; j<effort; j++)
    load();
```

Figure 74: The baseline for the *L2* synthetic application.

7.3.2.3 Four dependent loops: L4

L4 executes 4 dependent TFlux Loops and has a Synchronization Graph (Figure 75-(a)) identical to the *RK* benchmark (Section 7.2.5). As for the TFlux Code of this application, it is depicted in Figure 75-(b).

The code of the baseline for *L4*, which executes the same computational load sequentially, is depicted in Figure 76.

7.3.2.4 Dependent loops and recycle:L2R

L2R consists of two dependent TFlux Loops which are included inside a recycle-group. This code structure, *i.e.* a number of dependent TFlux Loops which are executed multiple times, can be found in the Synchronization Graphs of both the *CG* (Section 7.2.7) and the *LU* benchmarks

Kyriakos Stavrou

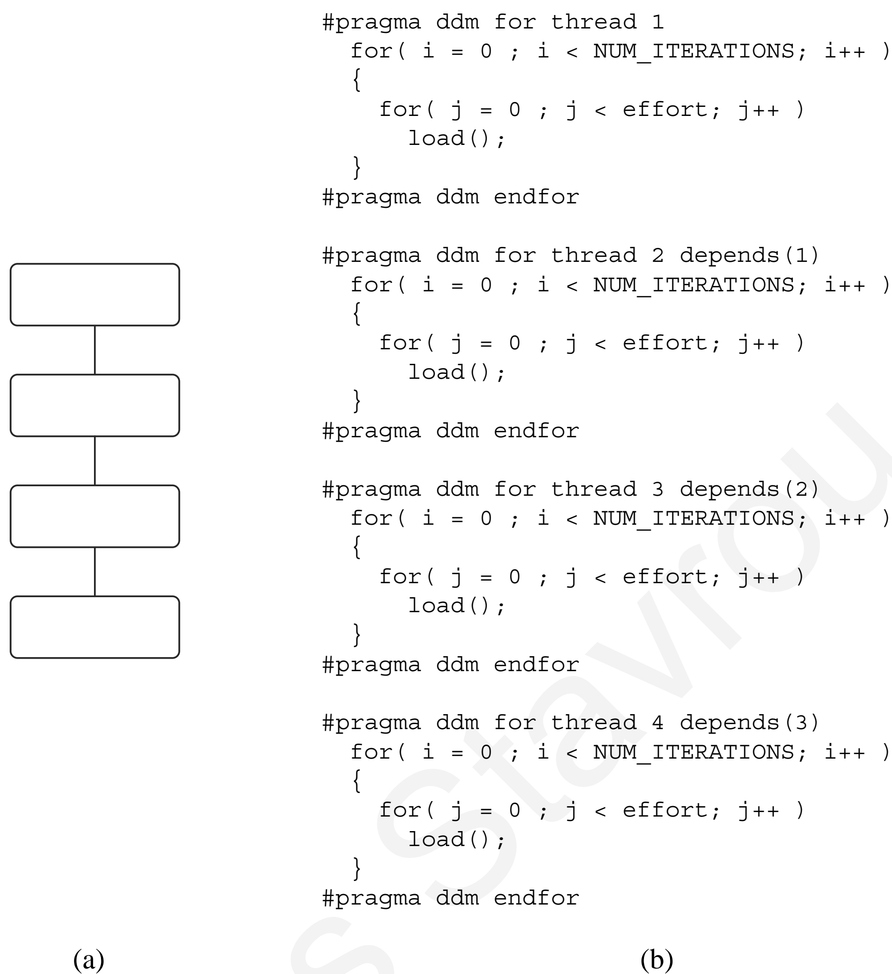


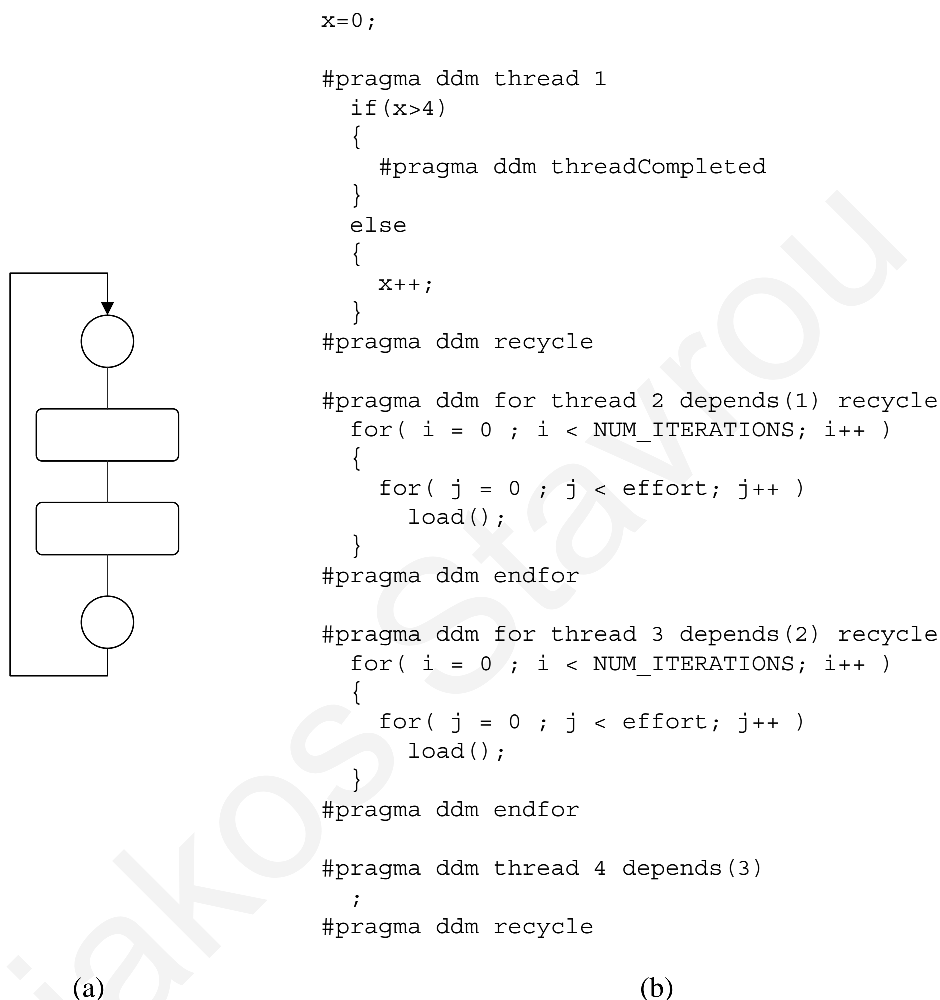
Figure 75: (a) The Synchronization Graph of L4 (b) The TFlux code of L4.

```
for(i=0; i<NUM_ITERATIONS; i++)
  for(j=0; j<effort; j++)
    load();

for(i=0; i<NUM_ITERATIONS; i++)
  for(j=0; j<effort; j++)
    load();

for(i=0; i<NUM_ITERATIONS; i++)
  for(j=0; j<effort; j++)
    load();

for(i=0; i<NUM_ITERATIONS; i++)
  for(j=0; j<effort; j++)
    load();
```

Figure 76: The baseline for the *L4* synthetic application.

Kyriakos Stavrou

(Section 7.2.8). For this application the execution of the two loops is repeated four times which has been found adequate for our experimentation. Figure 77-(a) depicts the Synchronization Graph of *L2R* whereas Figure 77-(b) its TFlux code.



```
x=0;

#pragma ddm thread 1
  if(x>4)
  {
    #pragma ddm threadCompleted
  }
  else
  {
    x++;
  }
#pragma ddm recycle

#pragma ddm for thread 2 depends(1) recycle
  for( i = 0 ; i < NUM_ITERATIONS; i++ )
  {
    for( j = 0 ; j < effort; j++ )
      load();
  }
#pragma ddm endfor

#pragma ddm for thread 3 depends(2) recycle
  for( i = 0 ; i < NUM_ITERATIONS; i++ )
  {
    for( j = 0 ; j < effort; j++ )
      load();
  }
#pragma ddm endfor

#pragma ddm thread 4 depends(3)
  ;
#pragma ddm recycle
```

(a) (b)

Figure 77: (a) The Synchronization Graph of L2R (b) The TFlux code of L2R.

The code of the corresponding baseline program for this application is depicted in Figure 78. Notice that the two applications execute the same computational load.

7.3.3 TFlux Advantage of Dataflow Scheduling

The purpose of this set of synthetic applications is to quantify the advantage of TFluxSoft resulting from its dataflow scheduling policy. As the objective of using synthetic applications is

Kyriakos Stavrou

```
for(i=0; i<4; i++)
{
      for(i=0; i<NUM_ITERATIONS; i++)
        for(j=0; j<effort; j++)
          load();
}
```

Figure 78: The baseline for the *L2R* synthetic application.

to analyze the differences between the dataflow and *"traditional"* parallel execution, we use the latter as the baseline. By *"traditional" parallel execution*, we refer to the parallel execution that uses parallel threads, loops and lock and barrier synchronization. An example of this would be the execution using OpenMP. Our baseline programs are programmed in TFluxSoft but using the OpenMP-like model. This is done in order to avoid the results being affected by the overheads or optimizations of a particular OpenMP implementation.

7.3.3.1 Iteration Level Dependencies between 2 loops

ILD2$_x$ consists of 2 TFlux Loops which do not depend at the level of whole TFlux Loops but rather at the loop-iteration level *i.e.* at the level of L-DThreads. To study the potential of Iteration Level Dependencies we experimented with several alternatives of this application. These different versions, for which we use the notion $ILD2_x$, describe the imbalance between different iterations of the loops (this imbalance regards *both* TFlux Loops). For these applications, the load of consecutive loop iterations increases by a constant factor (which is the load executed by loop iteration 0) whereas x consecutive loop iterations have the same load. Figure 79 depicts the number of calls to the *load()* function each iteration of these loops performs as a function of x, *i..e* as a function of the imbalance factor. Notice that $ILD2_0$ refers to the case where all loop iterations have the same load. This type of load increase between the loop iteration exists in the CG benchmark (Section 7.2.7).

Kyriakos Stavrou

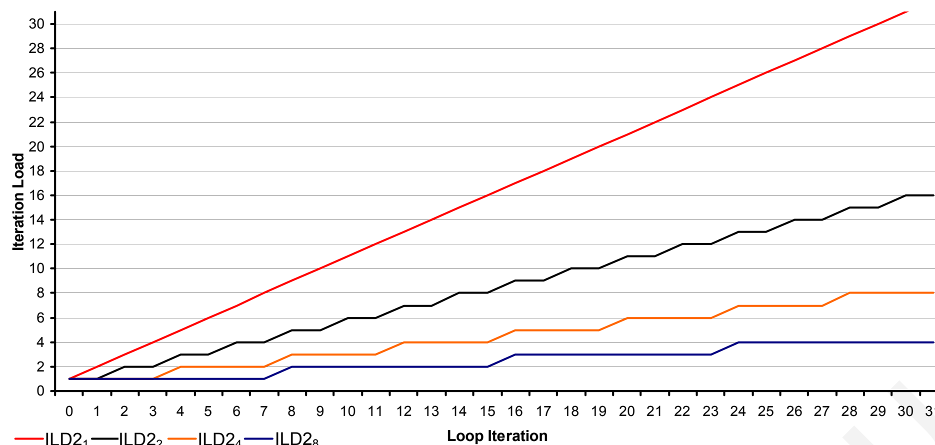


Figure 79: The load executed by each loop iteration for synthetic applications $ILD2_x$.

Figure 80-(a) depicts the Synchronization Graph of the TFlux version of *ILD2*$_x$ whereas Figure 80-(b) the Synchronization Graph of its baseline. The baseline application executes the same computational load as the TFlux version of *ILD2*$_x$ but instead of applying the synchronization between the two loops at the iteration level it applies it at the level of the whole loops, *i.e.* this execution follows what would apply for the "traditional" parallel processing model.

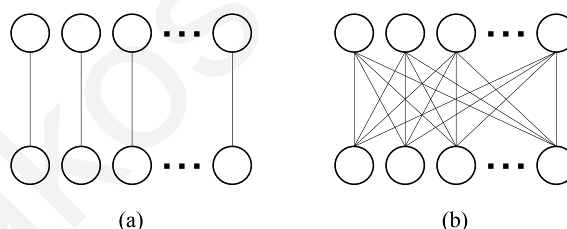


Figure 80: The Synchronization Graph of the (a) TFlux version and the (b) baseline of the $ILD2_x$ synthetic application.

For the comparative study of the TFlux and the baseline version of *ILD2*$_x$, each TFlux Loop executes for 1024 iterations as it results in a simulated execution time within the criteria (less than 24 hours simulation time). The code for the TFlux version of *ILD2*$_x$ is depicted in Figure 81-(a) whereas the code of the baseline version by Figure 81-(b).

Kyriakos Stavrou

```
#pragma ddm for thread 1 ilc[1 2 1 0 0 0]
  for(i=0;i<NUM_ITERATIONS;i++)
  {
   effort=i/X;
    for(j=0;j<effort;j++)
      load();
  }
#pragma ddm endfor

#pragma ddm for thread 2 readyCount 1
  for(i=0;i<NUM_ITERATIONS;i++)
  {
   effort=i/X;
    for(j=0;j<effort;j++)
      load();
  }
#pragma ddm endfor
```

(a)

```
#pragma ddm for thread 1
  for(i=0;i<NUM_ITERATIONS;i++)
  {
   effort=i/X;
    for(j=0;j<effort;j++)
      load();
  }
#pragma ddm endfor

#pragma ddm for thread 2 depends 1
  for(i=0;i<NUM_ITERATIONS;i++)
  {
   effort=i/X;
    for(j=0;j<effort;j++)
      load();
  }
#pragma ddm endfor
```

(b)

Figure 81: (a) The TFlux code of the TFlux version of ILD2_x. (b) The TFlux code of the baseline version of ILD2_x.

7.3.3.2 Binary Tree

The second synthetic application used for this study is named *BINARY TREE* as its Synchronization Graph is identical to that of a complete binary tree (Figure 82-(a)). This graph is very common in compute-intensive real-life workloads and is used for “reduction-like” operations [17, 22, 23, 104]. Each node of the graph of *BINARY TREE* represents the calculations performed in order to produce the results for the next phase until the final result is reached. An application from the TFlux Evaluation Suite that uses this technique is the *SORT* benchmark (Section 7.2.4).

For the baseline of this application, which is depicted in Figures 82-(b) and 82-(c), synchronization is achieved through barriers between the different phases as this would be the case for “traditional” parallel execution according to our previous definition.

The code of the TFlux and baseline version of the *BINARY TREE* application with 3 levels are depicted in Figures 83-(a) and 83-(b) respectively. For *BINARY TREE* experiments have been made for 4, 5 and 6 levels as well as for different values of the computational load. Notice however,

Kyriakos Stavrou

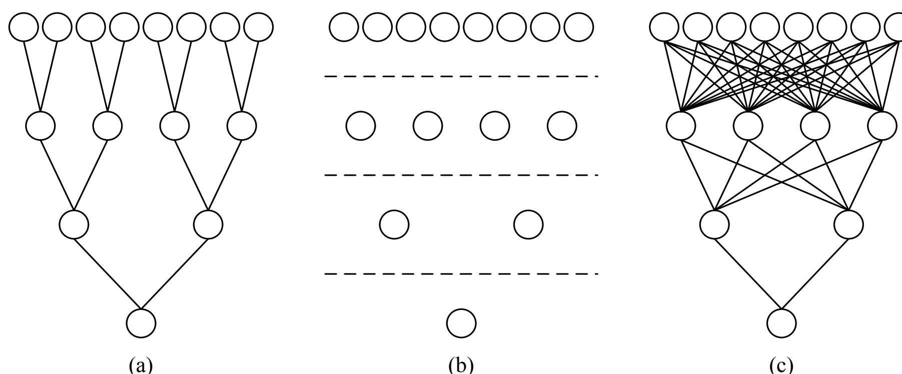


Figure 82: The Synchronization Graph of the (a) TFlux version and the (b) baseline (vertical lines represent barriers) of the *BINARY TREE* synthetic application. (c) Detail of the synchronization necessary for the baseline version of *BINARY TREE*.

that the computational load executed by the different DThreads within each experiment is the same.

7.3.3.3 Diagonal

DIAGONAL consists of a loop the iteration of which have a specific data-dependency pattern. In particular, each iteration of this loop operates on one element of a square matrix and the iterations that can proceed in parallel at a given point in time are those that operate on the elements of the same diagonal (see Section 7.2.8). Examples of applications that include this component are the *LU* (Section 7.2.8), *OCEAN* [129] and *H.264* [52] benchmarks. As can be seen from Figure 85-(b) that presents the TFlux code for this application, its Synchronization Graph (Figure 85-(a)) can be expressed with a single *#pragma ddm for* directive using the *ilc* statement (*i.e.* by using iteration-level dependencies).

The code of the baseline for this application is depicted in Figure 85. This code is based on the approach followed by the OpenMP implementation of the NAS parallel benchmarks [46], for this particular graph as explained in Section 7.2.8.

Kyriakos Stavrou

```
#pragma ddm for thread 1 kernel 1
  for(j=0;j<effort;j++)
    load();
#pragma ddm endthread

#pragma ddm for thread 2 kernel 2
  for(j=0;j<effort;j++)
    load();
#pragma ddm endthread

#pragma ddm for thread 3 kernel 3
  for(j=0;j<effort;j++)
    load();
#pragma ddm endthread

#pragma ddm for thread 4 kernel 4
  for(j=0;j<effort;j++)
    load();
#pragma ddm endthread

#pragma ddm for thread 5 kernel 1\
depends(1,2)
  for(j=0;j<effort;j++)
    load();
#pragma ddm endthread

#pragma ddm for thread 6 kernel 2\
depends(3,4)
  for(j=0;j<effort;j++)
    load();
#pragma ddm endthread

#pragma ddm for thread 7 kernel 1\
depends(5,6)
  for(j=0;j<effort;j++)
    load();
#pragma ddm endthread
```

(a)

```
#pragma ddm for thread 1 kernel 1
  for(j=0;j<effort;j++)
    load();
#pragma ddm endthread

#pragma ddm for thread 2 kernel 2
  for(j=0;j<effort;j++)
    load();
#pragma ddm endthread

#pragma ddm for thread 3 kernel 3
  for(j=0;j<effort;j++)
    load();
#pragma ddm endthread

#pragma ddm for thread 4 kernel 4
  for(j=0;j<effort;j++)
    load();
#pragma ddm endthread

#pragma ddm for thread 5 kernel 1\
depends(1,2,3,4)
  for(j=0;j<effort;j++)
    load();
#pragma ddm endthread

#pragma ddm for thread 6 kernel 2\
depends(1,2,3,4)
  for(j=0;j<effort;j++)
    load();
#pragma ddm endthread

#pragma ddm for thread 7 kernel 1\
depends(5,6)
  for(j=0;j<effort;j++)
    load();
#pragma ddm endthread
```

(b)

Figure 83: (a) The TFlux code of the TFlux version of *BINARY TREE*. (b) The TFlux code of the baseline version of *BINARY TREE*.

For *DIAGONAL* experiments have been performed using 2025 iterations of the main loop (this number was found to be adequate for the experimentation. As for the computational load executed by each iteration varied from 16 to 4096 calls to the *load()* function.

7.3.4 TFluxSoft Scalability Study

The target of the synthetic applications presented in this Section is to allow studying the scalability of TFluxSoft by using multiple *Updaters*. To enable this study these applications should

Kyriakos Stavrou

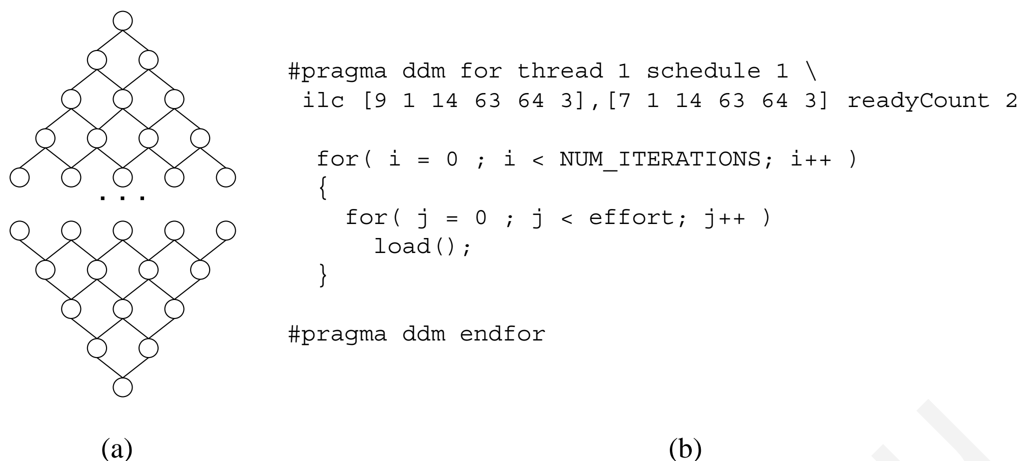


Figure 84: (a) The Synchronization Graph of DIAGONAL (b) The TFlux code of DIAGONAL.

```
#pragma ddm for thread 1
  for( i = 0 ; i < NUM_ITERATIONS; i++ )
  {
    //
    //Wait until the producer iterations have completed.
    //
    // - Wait for producer 1
    while(producer[i/ITERS_PER_DIM]==0)
      ;
    // - Wait for producer 2
    while(producer[i%ITERS_PER_DIM]==0)
      ;

     //
     // Do the ''useful'' computation
     //

     //
     //Set the flags to ''wakeup'' the consumers
     //
     producer[i/ITERS_PER_DIM]=1;
     producer[i%ITERS_PER_DIM]=1;
  }
#pragma ddm endfor
```

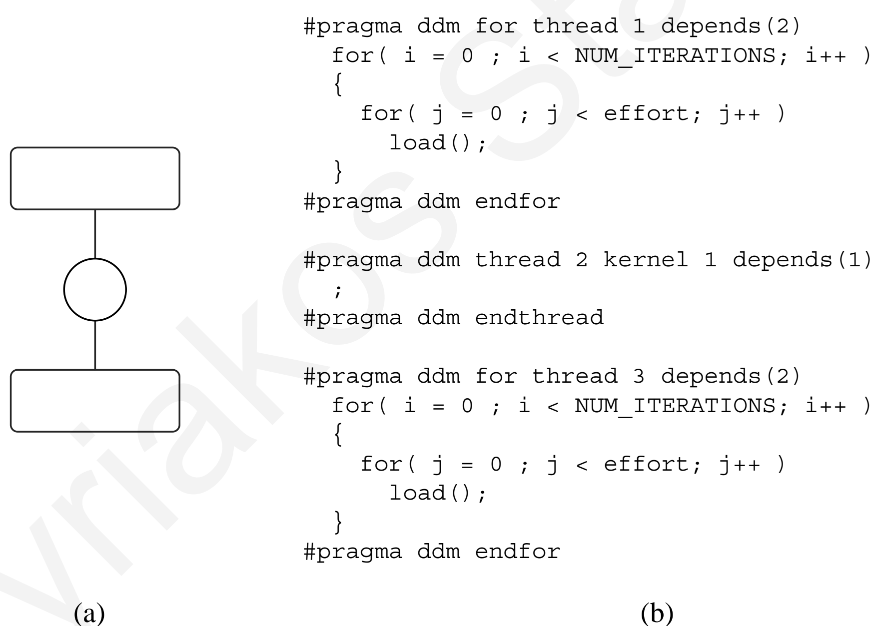
Figure 85: The baseline for the *DIAGONAL* synthetic application. Notice that $ITERS_PER_DIM^2 = NO_ITERATIONS$.

Kyriakos Stavrou

differ in the number of update-requests and the number of *bursts*. With the term *"burst"* we refer to the situation where in a small period of time a large number of update-requests are inserted into the TUB. An example of such a situation is when a TFlux Loop that has as consumer another TFlux Loop completes (Section 3.2.2).

In addition to *L1* (Section 7.3.2.1), *L2* (Section 7.3.2.2) and *L4* (Section 7.3.2.3) this set includes 4 more applications, *L2-T1*, *L4-T3*, *ILD2* and *ILD4*.

L2-T1: *L2-T1* is a variation of *L2* where the synchronization between the two TFlux Loops is achieved through an intermediate DThread. Notice that this extra DThread does not execute any computational load and serves for synchronization purposes only. The practical difference between *L2* and *L2-T1* is the that the former poses significantly more update-requests than the later ($32 \cdot 32 \cdot numKernels$ vs. $2 \cdot 32 \cdot numKernels$) for synchronization between the loops (see Section 3.2.2.1). The Synchronization Graph for this application is depicted in Figure 86-(a) whereas the TFlux code in Figure 86-(b). The baseline for *L2-T1* is identical to the baseline for *L2* (Figure 74)



```
#pragma ddm for thread 1 depends(2)
  for( i = 0 ; i < NUM_ITERATIONS; i++ )
  {
    for( j = 0 ; j < effort; j++ )
      load();
  }
#pragma ddm endfor

#pragma ddm thread 2 kernel 1 depends(1)
  ;
#pragma ddm endthread

#pragma ddm for thread 3 depends(2)
  for( i = 0 ; i < NUM_ITERATIONS; i++ )
  {
    for( j = 0 ; j < effort; j++ )
      load();
  }
#pragma ddm endfor
```

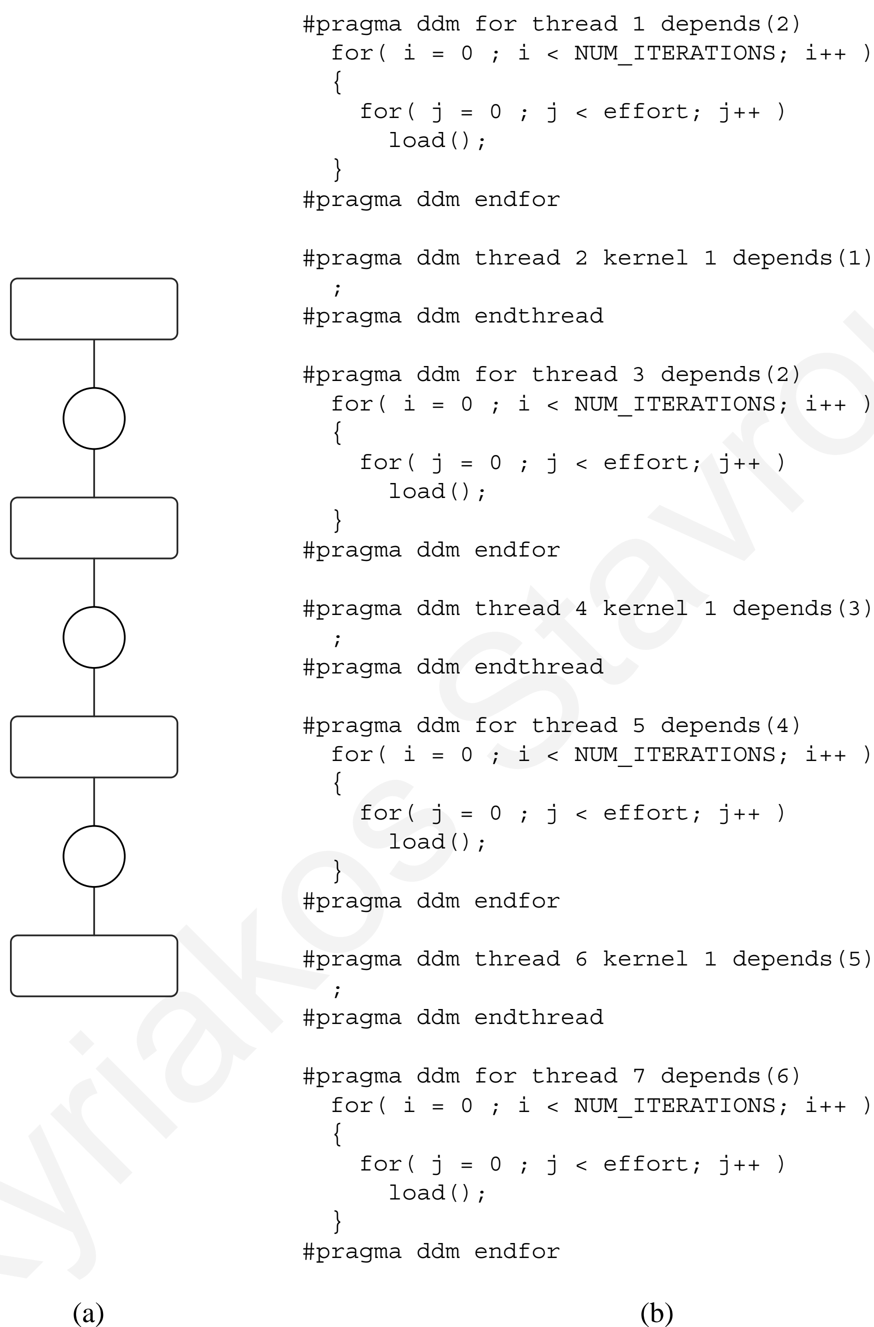
(a) (b)

Figure 86: (a) The Synchronization Graph of *L2-T1* (b) The TFlux code of *L2-T1*.

Kyriakos Stavrou

L4-T3: Similar to *L2-T1*, application *L4-T3* applies the same technique of introducing an intermediate DThread for the synchronization between TFlux Loops and regards the application

L4. Figure 87-(a) depicts the Synchronization Graph of this application and 87-(b) the corresponding TFlux code. As for the baseline of *L4-T3* it is identical to that of *L4* (Figure 76).



```
#pragma ddm for thread 1 depends(2)
  for( i = 0 ; i < NUM_ITERATIONS; i++ )
  {
    for( j = 0 ; j < effort; j++ )
      load();
  }
#pragma ddm endfor

#pragma ddm thread 2 kernel 1 depends(1)
  ;
#pragma ddm endthread

#pragma ddm for thread 3 depends(2)
  for( i = 0 ; i < NUM_ITERATIONS; i++ )
  {
    for( j = 0 ; j < effort; j++ )
      load();
  }
#pragma ddm endfor

#pragma ddm thread 4 kernel 1 depends(3)
  ;
#pragma ddm endthread

#pragma ddm for thread 5 depends(4)
  for( i = 0 ; i < NUM_ITERATIONS; i++ )
  {
    for( j = 0 ; j < effort; j++ )
      load();
  }
#pragma ddm endfor

#pragma ddm thread 6 kernel 1 depends(5)
  ;
#pragma ddm endthread

#pragma ddm for thread 7 depends(6)
  for( i = 0 ; i < NUM_ITERATIONS; i++ )
  {
    for( j = 0 ; j < effort; j++ )
      load();
  }
#pragma ddm endfor
```

(a) (b)

Figure 87: (a) The Synchronization Graph of *L4-T3* (b) The TFlux code of *L4-T3*.

Kyriakos Stavrou

ILD2: Similar to *L2*, *ILD2* consists of two TFlux Loops. The key difference between these two applications is that whereas for *L2* the two loops have a full dependency between them

(*all* iterations of the Consumer loop depend on *all* iterations of the Producer loop - Section 3.2.2.1) for *ILD2* the dependencies are at the iteration-level. In particular, these two TFlux Loops execute the same number of iterations and iteration x of the Producer TFlux Loop has been set to “wake-up” iteration x of the Consumer TFlux Loop. The TFlux code for *ILD2* is depicted in Figure 88-(a) and its Synchronization Graph by Figure 88-(b).

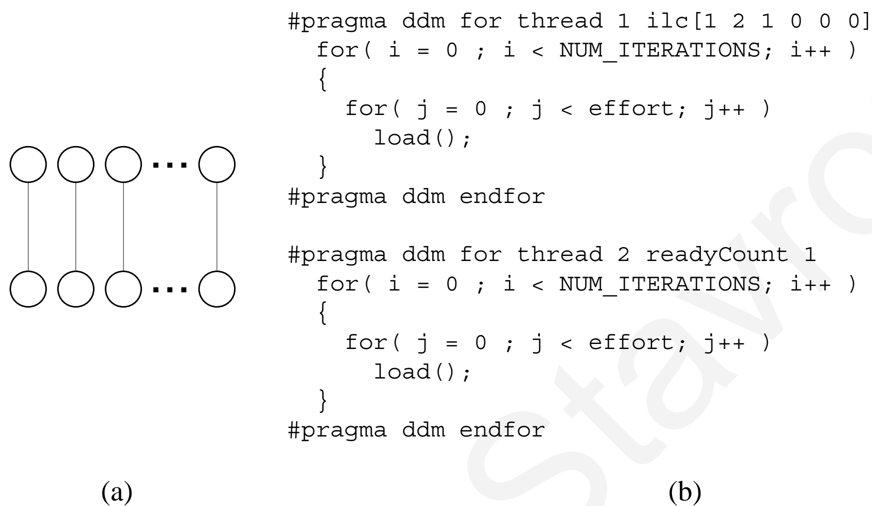


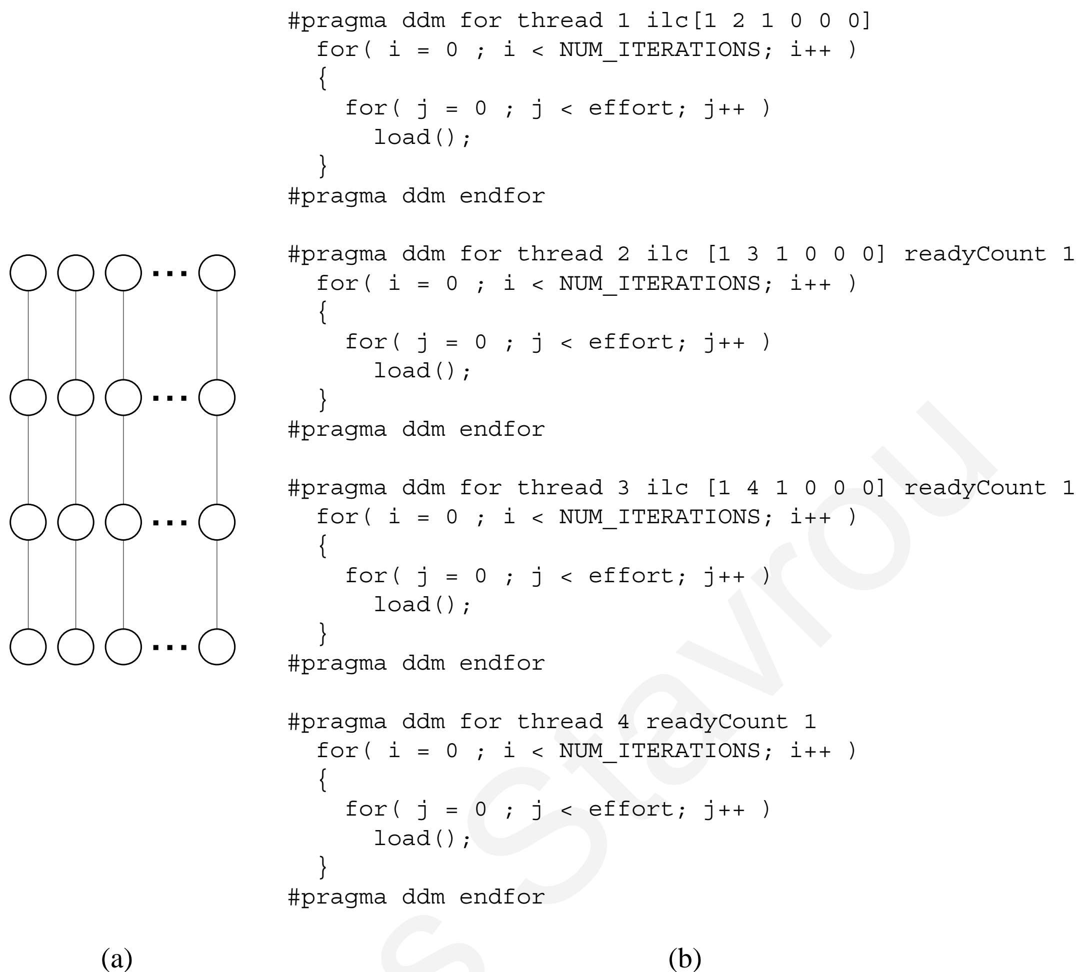
Figure 88: (a) The Synchronization Graph of ILD2. (b) The TFlux code of ILD2

ILD4: *ILD4* follows the same rationale as *ILD2* but instead of 2, it has 4 TFlux loops. The purpose of *ILD2* and *ILD4* is to study the behavior of the system when update-requests are inserted into the TUB with a continuous rate. The TFlux code for *ILD4* is depicted in Figure 89-(a) whereas its Synchronization Graph by Figure 89-(b).

The Synchronization Graphs of the set of synthetic applications used to study the potential of using multiple *Updaters* are summarized in Figure 90. As for Table 20, it summarizes the characteristics of these applications. In particular, for each synthetic application this Table reports the total number of update-requests and the number of *bursts*.

Regarding the number of update-requests notice that the term $2 \times n$ is common for all applications with n being the number of TFlux Kernels. This term corresponds to the requests inserted

Kyriakos Stavrou



(a) (b)

Figure 89: (a) The Synchronization Graph of ILD4 (b) The TFlux code of ILD4.

L1	L2	L2-T1	L4	L4-T3	ILD 2	ILD 4

Figure 90: The synthetic applications used to study the potential of using multiple *Updaters*.

Kyriakos Stavrou

into the TUB by the Inlet DThreads in order to "wake-up" the first TFlux Loop and those the last TFlux Loop inserts in order to deem executable the Outlet DThreads. The term $32 \times 32 \times n$, which is present in applications *L2* and *L4* corresponds to the update-requests necessary when two TFlux Loops depend direcly on each other (Section 3.2.2-Figure 8). As for the term $32 \times n$, which is present in applications *L2-T1* and *L4-T3*, it describes the udpate-request regarding the dependency between a DThread and a TFlux Loop (Section 3.2.2-Figure 7). Finally, the term $NumIters$ is for the update-requests related to iteration-level dependencies (Section 3.2.2-Figure 9) as each L-DThread of the Producer TFlux Loop has as a consumer one L-DThread of the Consumer TFlux Loop whereas both loops execute $NumIters$ iterations.

As for the number of *bursts*, it is possible to observe that all applications have at least two, one related to the Inlet DThread "waking-up" the first TFlux Loop and one related to the last TFlux Loop waking up the Outlet DThread. The other situations that lead to a *burst* are the direct dependencies between TFlux Loops and the dependence between a TFlux Loop and a DThread.

All these synthetic applications have been executed for 3 different sizes of the computational load (S: small, M: Medium and L: large). For this experimental setup, for the Small computational load each loop iteration executes 925 *simple* assembly instructions, for the Medium size 1800 and for the Large size 3500 (8, 16 and 32 calls to the *load()* function respectively). As for the number of iterations per TFlux Loop, for all benchmarks this is equal to 4096 as this number has been found to be adequate for simulation purposes.

Kyriakos Stavrou

Table 20: Number of update-requests and number of *bursts* for the synthetic applications used for studying the potential of using multiple *Updaters*. n is the number of TFlux Kernels in the system and $NumIters$ the number of iterations executed by each TFlux Loop.

Benchmark	***Number of update-requests***	***Number of bursts***
L1	$2 \cdot 32 \cdot n$	2
L2	$2 \cdot 32 \cdot n + 32 \cdot 32 \cdot n$	3
L2-T1	$2 \cdot 32 \cdot n + 2 \cdot 32 \cdot n$	4
L4	$2 \cdot 32 \cdot n + 3 \cdot 32 \cdot 32 \cdot n$	5
L4-T3	$2 \cdot 32 \cdot n + 3 \cdot 2 \cdot 32 \cdot n$	8
ILD 2	$2 \cdot 32 \cdot n + NumIters$	2
ILD 4	$2 \cdot 32 \cdot n + 3 \cdot NumIters$	2

Kyriakos Stavrou

Chapter 8

Experimental Setup

This Chapter presents the details of the experimental methodology followed for the quantitative evaluation of the TFlux platform. More specifically, it presents the infrastructure used for the evaluation of TFlux and discusses several practical issues regarding the experimentation process.

8.1 Experimentation Infrastructure

This Section presents details of the machines used to evaluate TFluxHard and TFluxSoft. Two of the systems presented in this Section where used to collect *performance* statistics while the other systems were used to study the virtualization of TFlux.

As all the *simulated* systems were based on the Virtutech Simics[72] *full-system* simulator, this Section will first introduce this tool and explain the way it has been used for the experimentation. Moreover, this Section also explains how Simics was used to model the Scheduler as a hardware unit in order to simulate the TFluxHard system.

Kyriakos Stavrou

8.1.1 Virtutech Simics Full System Simulator

Simics is a *full-system* simulator that models the major components of the system in such a level of detail that allow it to boot an *unmodified* Operating System. Simics is able to simulate a large number of *different* systems. The main parameters that can be configured for these systems are the number of processors and the cache hierarchy. However, the number of processors can not be larger than 28 for the currently publicly available version of Simics [102].

8.1.1.1 Generic simulated machine

All simulated systems used in this work follow the scheme depicted in Figure 91 which is similar to that of recent, commercial multicore chips. This configuration consists of a number of interconnected CPUs operating under a *shared memory* environment. Each CPU has its own, private L1 data and instruction cache, as well as a private unified L2 cache. All L2 caches are connected to the system's network which in turn, communicates with the main memory. Notice that the cache coherency is enforced at the level of the L2 caches.

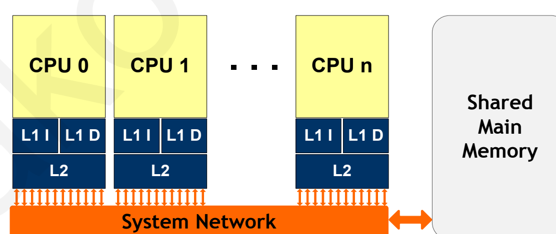


Figure 91: Conceptual view of the generic simulated machine.

Kyriakos Stavrou

8.1.1.2 Simics Timing

Regarding timing, Simics provides three operation modes. The first, *"Normal"* provides only functional simulation, *i.e.* Simics just executes the instructions of the application without taking

into consideration the time each instruction or memory operation takes. The second operation mode is named *"Stall"* and provides accurate timing for the memory hierarchy. As for the instructions execution, different type of assembly instructions do take different number of cycles, but the microarchitectural model is that of a in-order processor. The third mode, *"Micro-architecture"* simulates an out-of-order pipelined CPUs but is not as accurate in terms of the simulation of the cache hierarchy [100, 102].

In this work all experiments have been executed using the *"Stall"* mode, *i.e.* the mode that provides accurate memory hierarchy simulation combined with a simple in-order processor model. This choice is justified by the fact that the experimental evaluation is based on the *"speedup"* metric, *i.e.* how many times execution under TFlux is faster compared to sequential execution with both the sequential and the TFlux binaries executing on identical processor cores. As speedup is calculated as the ratio of these execution times, the details of the CPU on this metric are not likely to have a significant effect on the results.

8.1.2 TFluxHard Simulation: Modeling the Scheduler

In order to simulate TFluxHard we needed to model the Scheduler as a hardware module and attach it to the system network of the simulated machine as a memory mapped device. Figure 92 presents the scheme of the simulated machine with the addition of the Scheduler. Notice that the addition of the Scheduler is the only difference between this configuration and that of the generic simulated machine (Figure 91).

Kyriakos Stavrou

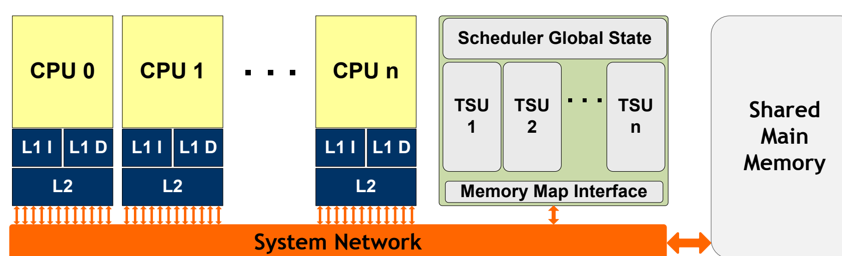


Figure 92: The conceptual view of the TFluxHard simulator.

8.1.2.1 Scheduler Model

The Scheduler was modeled using the Device Modeling Language (DML) [98] provided by Virtutech, for describing hardware components. DML, in addition to providing the means for describing the operation of the module at the functional level, it also provides the necessary interfaces and API that allow this module to interact with the rest of the system components.

To model the Scheduler we followed a bottom-up approach; each major component was modeled individually and then these components were combined to form the final design. In particular, this model comprises of several TSUs connected through the shared units.

8.1.2.2 Scheduler Timing

As explained in Section 5.3, in TFluxHard the several Scheduler operations are invoked by the CPU through the Scheduler Instructions. Figure 93 depicts the timing of these operation as these are handled by the simulated system. In particular, at time instance t_s the CPU sends an Instruction which is received by the Scheduler at time instance t_r. Then at time instance t_{os} the operation starts and completes at time instance t_{oc}. As such, the time required for the command to be sent from the CPU to the Scheduler is $d_s = t_r - t_s$. The time for the Scheduler to initiate the operation is $d_a = t_r - t_{os}$ whereas the time needed for the operation to complete is $d_o = t_{oc} - t_{os}$.

Kyriakos Stavrou

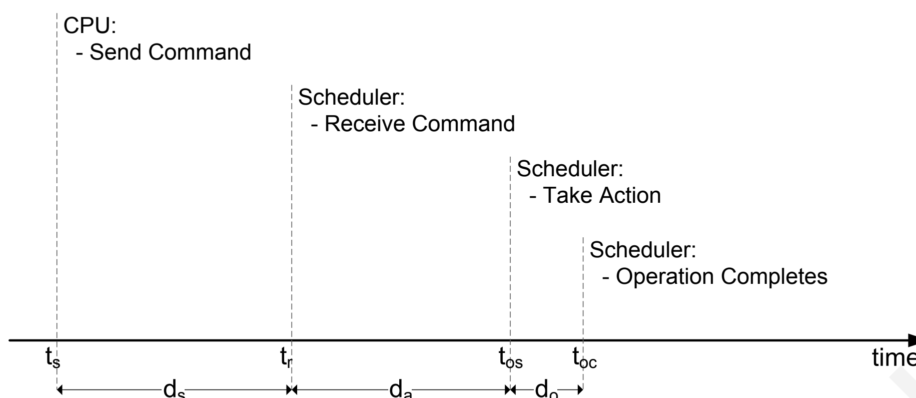


Figure 93: Timing of executing Scheduler operation for TFluxHard.

Similar to what would be the case in a real system, in the simulated system the time needed for the command sent by the CPU to reach the TSU (d_s) is defined implicitly by the memory hierarchy. As for the time elapsed between the time instances that the TSU receives a command until it invokes the appropriate operation (d_a), this depends on the particular implementation of the Scheduler. In the simulated system, this delay (d_a) can be defined by the user through the *delay* parameter. Finally, whereas the time needed to perform an operation (d_o) could take multiple cycles in a real implementation, in the simulated system it always takes only one cycle. However, using the *delay* parameter it is possible to simulate the impact of the Scheduler's timing on the performance of TFluxHard.

8.1.3 Systems used for Performance Evaluation

To evaluate the performance of TFluxHard and TFluxSoft we used 2 systems. The first is a simulated machine with 28 CPUs and the second an *off-the-shelf* IBM x3650 server with 2 Intel Xeon E5320 Quad Core processors. TFluxSoft was evaluated on both systems whereas TFluxHard, due its requirement for hardware extensions, was evaluated only on the simulated machine.

Kyriakos Stavrou

8.1.3.1 Simulated System: TFluxSim

The main system used for the evaluation of the TFlux platform is based on a Simics configuration with the code name *"Cashew"*. Cashew is a Sun Enterprise 6500 server with UltraSPARC II processors [99]. It has one Ethernet adapter, one SCSI disk and one SCSI CD-ROM. Cashew is configured to run an unmodified Aurora 2.0 Linux (Fedora Cora 3) disk. The Linux' kernel version is 2.6.13.

The simulated machine, which we named *"TFluxSim"*, follows the design of the generic machine depicted in Figures 91 and 92 and was configured with 28 CPUs. *TFluxSim* operates as a shared memory system. Each CPU has its own, private L1 Data and Instruction Cache, as well as private Unified L2 Cache. Notice that the MESI cache-coherency protocol is supported at the L2 cache level and the corresponding coherency overheads are correctly simulated. The details of the memory hierarchy of *TFluxSim* are summarized in Table 21. Notice however, no detailed model exists for the delay of the communication infrastructure.

The compiler of this system is gcc version 3.4.5 with glibc version 2.3-5.

Table 21: Memory hierarchy configuration for TFluxSim

	L1 I	***L1 D***	***L2***	***Main Memory***
Private/Shared	Private	Private	Private	Shared
Size	32KB	32KB	2MB	256MB
# lines	512	512	8192	N/A
Line Size	64	64	256B	N/A
Associativity	4	4	8	N/A
Write Back	No	No	Yes	N/A
Write Allocate	No	No	Yes	N/A
Replacement Policy	LRU	LRU	LRU	N/A
Read Delay	2	2	20	200
Write Delay	0	0	20	200

Kyriakos Stavrou

8.1.3.2 Off-the-shelf system: IBM3650

IBM3650, which configuration is depicted in Figure 94, is an IBM x3650 server equipped with 2 Intel Xeon E5320 Core2 QuadCore processors clocked at 1.86GHz. As such, *IBM3650* provides to its user an 8-core shared-memory system. *IBM3650* runs Fedora Core 6 with Linux Kernel 2.6.22.

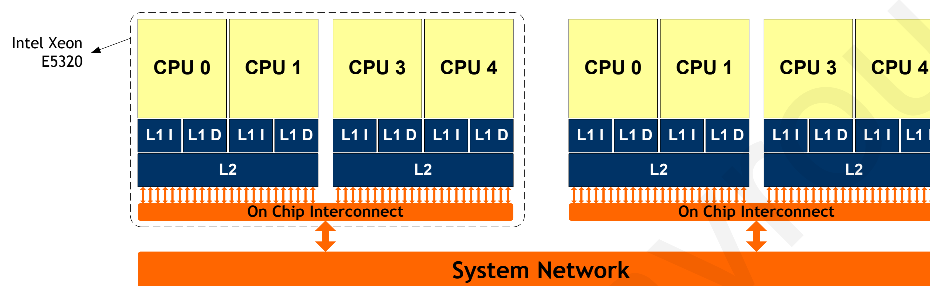


Figure 94: The conceptual view of the IBM3650 system.

Each core has access to a private 64KB L1 Instruction Cache, a 64KB L1 Data Cache. As for the L2 cache its size is 4MB and is shared between 2 cores. The system is equipped with 18GB of DDR2 main memory and the FSB is clocked at 1066MHz. More details about the memory hierarchy of the Intel Xeon E5320 processor are presented in Table 22.

The compiler of this system is gcc version 4.1.2 with glibc 2.5-3.

Table 22: Memory hierarchy of Intel Xeon E5320

	L1 I	*L1 D*	*L2*	*Main Memory*
Private/Shared	Private	Private	Private	Shared
Size	64KB	64KB	4MB	18GB
Line Size	64	64	64	-
Associativity	8	8	16	-

Kyriakos Stavrou

8.1.4 Systems used for Studying Virtualization

In order to study the virtualization capabilities of TFlux we used three more systems in addition to *TFluxSim* and *IBM3650* which are presented in the sections that follow.

8.1.4.1 Bagle

This system is based on the Simics machine code named *"Bagle"*. Bagle [99] shares the same hardware components as the *Cashew* machine but has a different Operating System. In particular, Bagle runs SuSE 7.3 Linux with Kernel version 2.4.14. Similar to the *Cashew* machine, Bagle can also be configured with up to 28 CPUs.

8.1.4.2 Tango

The *Tango* machine is an x86-based system equipped with Pentium 4 processors. *Tango* [101] runs Fedora Core 5 with Kernel version 2.6.15 and can be configured with up to 15 CPUs.

8.1.4.3 Enterprise

Lastly, the *Enterprise* system [101] shares the same hardware configuration as *Tango* but runs a different Operating System, namely Red Hat Linux 7.3 with Linux Kernel 2.4.18. In contrast to *Tango*, *Enterprise* may be configured with up to 8 CPUs.

8.1.5 Summary

As presented in this Section, the systems used for the evaluation of TFlux consist of both real, *off-the-shelf* multiprocessors and simulated machines. These systems differ in the number and type of CPUs, the ISA of these CPUs, the Operating System and the Linux Kernel. The characteristics are summarized in Table 23. This set of machines includes systems with significant differences which allows us to perform a qualitative study of the virtualization capabilities of TFlux.

Kyriakos Stavrou

Table 23: Summary of the machines of the experimentation infrastructure

	TFluxSim	**Bagle**	**Tango**	**Enterprise**	**IBM3650**
Nature:	Simulated	Simulated	Simulated	Simulated	Off-the-shelf
Used to study:	Performance	Virtualization	Virtualization	Virtualization	Performance
ISA	SPARC	SPARC	x86	x86	x86
Endian	Big	Big	Little	Little	Little
#Cores	28	28	15	8	8
CPU Type	UltraSPARC II	UltraSPARC II	Pentium 4	Pentium 4	Xeon E5320
OS	Fedora Core 3	SuSe 7.3	Fedora Core 5	Red Hat .3	Fedora Core 6
Linux Kernel	2.6.13	2.4.14	2.6.15	2.4.18	2.6.22
Compiler	gcc V 3.4.5	gcc V 3.2.8	gcc V 3.4.5	gcc V 3.2.9	gcc V 4.1.2
glibc	2.3.5	2.2.3	2.3.5	2.2.3	2.5.3

8.2 Compilation

All benchmarks used for the evaluation of TFlux have been compiled using the *-O3* compilation flag in order to exploit all available compiler optimizations. Notice that this applies to both the sequential and the TFlux versions. For the off-the-shelf system, *IBM3650*, we compiled the applications natively on the machine whereas for the simulated system, *TFluxSim*, we used a cross-compiler identical to the one installed on that system.

8.3 Scheduling Policy

In Linux-based multiprocessor systems the scheduler tries to keep all CPUs equally utilized. To achieve this, the Linux scheduler usually does not leave a process to the same CPU for all its lifetime, even if the number of compute-intensive processes in the system is smaller than the number of CPUs. This scheduling policy causes multiple process migrations with a negative effect on the data locality and consequently may result in performance degradation.

To avoid this situation and have more fair comparisons we used the affinity scheduling functionality provided by recent Linux Kernels (2.6.x). In particular, for both the sequential and TFlux

Kyriakos Stavrou

versions of the benchmarks all execution entities were pinned to a different CPU. In the case of the TFluxSoft experiments, the pinning also applies to the *Updater*.

8.4 Unrolling

For some of the benchmarks of the TFlux evaluation suite the DThread size, in terms of number of dynamic instructions, was very small leading to a situation where the parallelization overheads were not amortized. A commonly used technique to avoid having too fine-grained tasks is to *unroll* the body of loops. Although this leads to fewer parallel tasks each such task becomes larger and therefore it amortizes better the overheads. Notice that unrolling often benefits the sequential version of applications as it leads to fewer dynamic instructions through decreasing the number of conditional branches executed by the loops.

To have fair comparisons, unrolling was applied both to the sequential and the TFlux versions of the benchmarks. Nevertheless, as the unroll factor that gave the minimum execution time for the sequential and the TFlux versions of the benchmarks was different, for each situation we used the unroll factor that lead to minimum execution time. More specifically, let the execution time of a benchmark with unroll factor x be t_x. If for a certain benchmark, experiments were performed using unroll factors of 1, 2, 4, ..., 64, we consider as the benchmark's execution time (t) the minimum execution time among all these experiments, *i.e.* $t = min\{t_1, t_2, t_4, ..., t_{64}\}$. This applies for both the sequential and the TFlux versions of the benchmark.

8.5 Metrics

The metric used to evaluate the TFlux platform is *"speedup"*, *i.e.* how many times TFlux execution is faster compared to sequential execution. Formally, if t^{Seq} is the time required to sequentially execute a benchmark and t^{TFlux} is the time required to execute this benchmark on

a TFlux system, the speedup of TFlux versus sequential execution is defined as shown by Equation 1.

$$Speedup = \frac{t^{Seq}}{t^{TFlux}} \tag{1}$$

Notice that the processors used for TFlux and sequential execution are the same; one core is used for sequential execution and multiple such cores for TFlux execution.

For benchmarks where unrolling was applied, speedup is calculated as the *best* sequential time over the *best* TFlux execution time as shown by Equation 2.

$$Speedup = \frac{t_{Seq}}{t_{TFlux}} = \frac{min\left\{t_1^{Seq}, t_2^{Seq}, t_4^{Seq}, ..., t_{64}^{Seq},\right\}}{min\left\{t_1^{TFlux}, t_2^{TFlux}, t_4^{TFlux}, ..., t_{64}^{TFlux}\right\}} \tag{2}$$

In some situations we also report the *usage* of a TFlux Kernel. With this term we refer to the percentage of time a TFlux Kernel was executing DThreads. Formally, if t_{Kernel} is the time period a TFlux Kernel is alive and t_{Exec} is the sum of the time periods during which this TFlux Kernel has been executing DThreads, *usage* is defined as shown by Equation 3.

$$Usage = \frac{t_{Exec}}{t_{Kernel}} \tag{3}$$

As such, *Usage* is a metric that measures the percentage of time a Kernel was doing "useful" work. Similarly, $1 - usage$ is the percentage of time a Kernel was waiting for the Scheduler to assign to it a ready DThread.

8.6 Collecting Statistics

For both metrics, *Speedup* and *Usage*, it was necessary to measure the time that elapsed between certain points in time, such as the time point regarding the start and completion of a TFlux

Kyriakos Stavrou

Kernel. For the simulated system, these time intervals were measured using the internal Simics timers through an interface we developed. As for the execution on the real system, *IBM3650*, we used the *gettimeofday()* system call.

8.7 Statistical Significance

For the native execution experiments to have statistical significance we executed all experiments multiple times. In particular, we took the average of 60 executions after removing the 5 smaller and 5 larger values. For all cases this resulted in a variance of less than 2%. Moreover, to isolate any interference from the Operating System, for native execution experiments, one CPU was left for the execution of OS's processes.

As for the simulated environment executing the same experiment multiple times lead to variance less than 0.5% for the benchmarks of the TFlux Evaluation Suite. Given this small variance, for the simulated system each experiment was executed only once.

Kyriakos Stavrou

Chapter 9

Performance Evaluation

Kyriakos Stavrou

In this Chapter we evaluate the performance of TFlux. The performance evaluation study starts with Section 9.1 that quantifies the minimum size a DThread needs to have in order to amortize the parallelization overheads for both TFluxHard and TFluxSoft. Section 9.2 presents the performance of TFlux for the real-life applications whereas Section 9.3 focuses on the performance for the synthetic workload. Finally, in the last two Sections we study implementation-specific parameters. In particular, in Section 9.4 we study the potential of using TFluxSoft configurations with multiple *Updaters* whereas in Section 9.5 the effect of the Scheduler's delay for TFluxHard.

9.1 Minimum DThread Size

The term *"Minimum DThread Size"* refers to the minimum number of dynamic instructions a DThread needs to execute in order for it to amortize the related parallelization overhead, *i.e.* the cost for its management and handling. This overhead includes the associated *DThread Load*, *DThread Completion* and *Find Ready DThread* operations.

The *Minimum DThread Size* is found using the *"Parallel Threads"* application which was presented in Section 7.3.1. Recall that for this application each TFlux Kernel executes only one DThread and that these DThreads do not have dependencies between themselves. For this experiment we test *Parallel Threads* of different *size*, and the case where the TFlux execution is faster than the execution of the same computational load determines the minimum DThread size. For example, for a configuration with N Kernels each of which executing a computational load of x instructions, the computational load that is to be executed by the sequential application is $N \times x$ instructions.

While for the sequential application, the execution time includes only the time required to execute the computational load, for the TFlux execution it also includes the time required for managing the DThreads that execute this load. In particular, this time includes all related Scheduler operations, *i.e.* the Thread Load, the Find Ready Thread and the Thread Completed operations.

The experimental results are presented in Figures 95 and 96 for TFluxHard and TFluxSoft respectively. The reason for which the results do not cover the 27 TFlux Kernels configuration is related to the increase of the computational load executed by the DThreads, which at each step, is doubled (this would therefore require a 32 node system which we are not able to simulate).

The experimental results present a clear trend which is the increase of the speedup with the computational load. However, it is possible to observe that TFluxHard is able to deliver the same speedup as TFluxSoft for smaller computational load, *i.e.* it is able to deliver its benefits for finer-grained DThreads.

With a closer look at the results it is possible to observe that TFluxHard outperforms the corresponding sequential execution when the computational load executed per DThread is at least equal to 2 calls to the *load()* function, *i.e.* 271 assembly instructions (Table 19 - Section 7.3). As for TFluxSoft, this minimum thread size is 8 calls to the *load()* function and correspond to

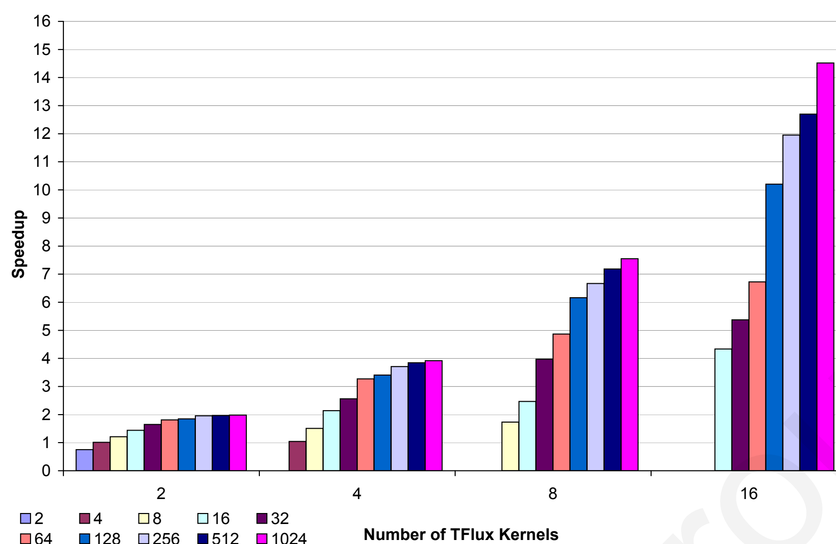


Figure 95: Quantification of the minimum DThread size for ***TFluxHard***.

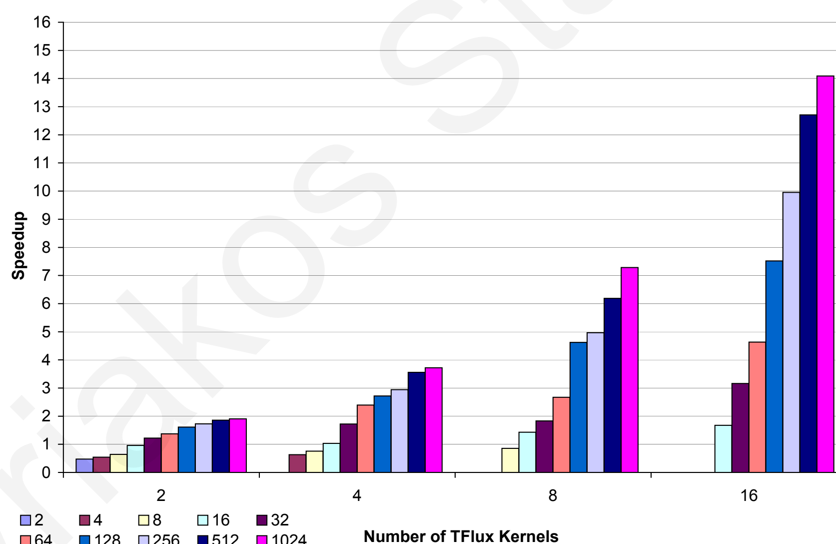


Figure 96: Quantification of the minimum DThread size for ***TFluxSoft***.

925 instructions. This performance advantage of TFluxHard over TFluxSoft is justified by the overhead that results from implementing the Scheduler's functionality at the software level for TFluxSoft.

Kyriakos Stavrou

9.2 Real Life Applications

This Section analyzes the performance of the TFlux platform for the real-life applications presented Section 7.2. All results are in the form of *Speedup*, *i.e.* how many times execution under TFlux is faster compared to the sequential execution (for more details on the definition of *Speedup* refer to Section 8.5).

In order to provide further depth to the experimental results presented later, in Figures 97, 98 and 99 we summarize the most important characteristics of the real-life applications.

Figure 97 depicts the number of DThreads executed by each application for the three input sizes. Notice that the Y-axis of this chart is in ***logarithmic*** scale.

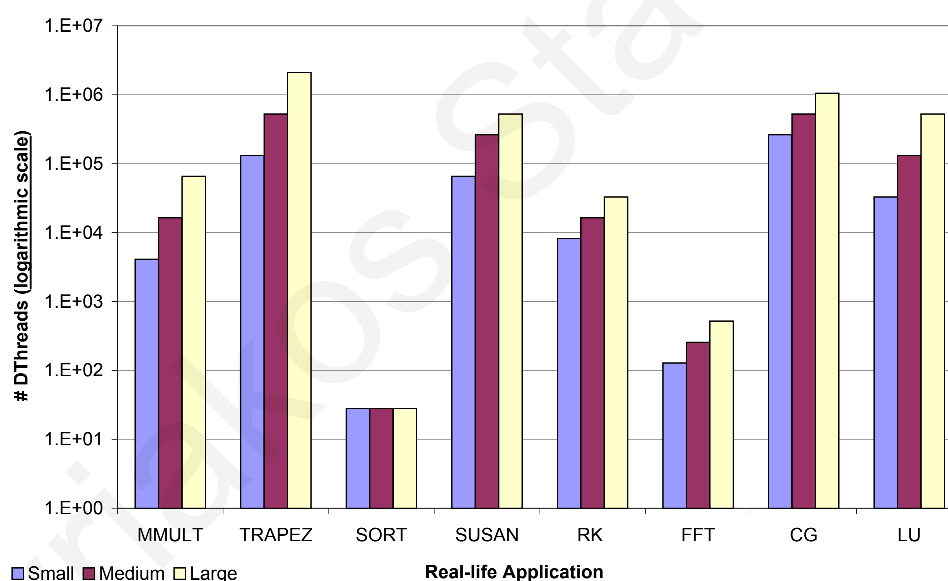


Figure 97: Number of dynamically executed DThread for the real-life applications. Notice that Y axis is in ***logarithmic*** scale.

As for Figure 98 it depicts the size of the critical DThreads for the different applications in terms of dynamic instructions. The term *"critical DThread"* refers to the DThread that consumes

Kyriakos Stavrou

the largest portion of the execution time. Notice that for this chart, the *Y* axis is also in ***logarithmic*** scale.

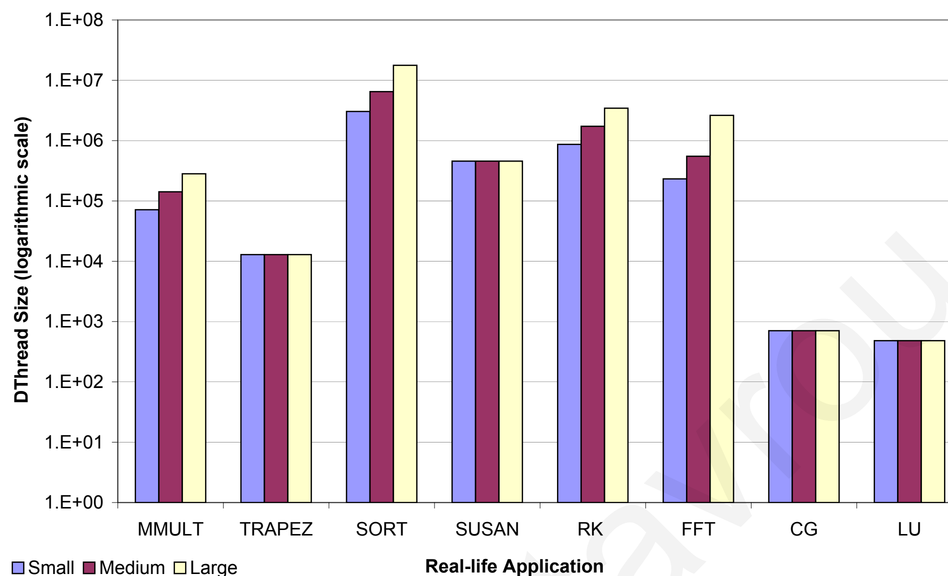


Figure 98: DThread sizes for the real-life applications. *Notice that Y axis is in **logarithmic** scale.*

Kyriakos Stavrou

Finally, Figure 99 depicts the L1-Data Cache miss rate of the *"critical DThread"* of each real-life application. The configuration of this cache is summarized in Table 9 (Section 7.2).

9.2.1 TFluxHard

The performance results for TFluxHard are presented in Figure 100. These experimental results have been collected through execution of the real-life applications on the *TFluxSim* system which, as was described in Section 8.1.3.1, corresponds to a simulated multicore machine with 28 UltraSPARC II processors. However, experiments have been conducted with only up to 27 CPUs to allow comparison with the TFluxSoft system. Recall that for performance reasons, for TFluxSoft it is necessary to dedicate one CPU for the *Updater*.

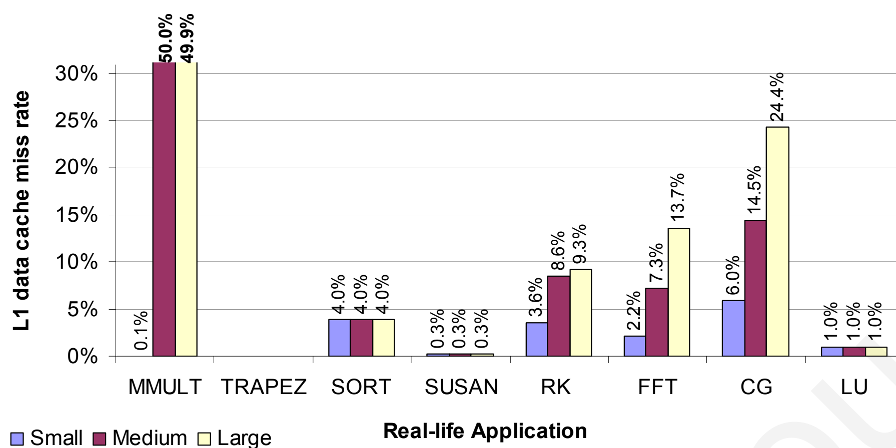


Figure 99: L1-data cache miss rate for the real-life applications.

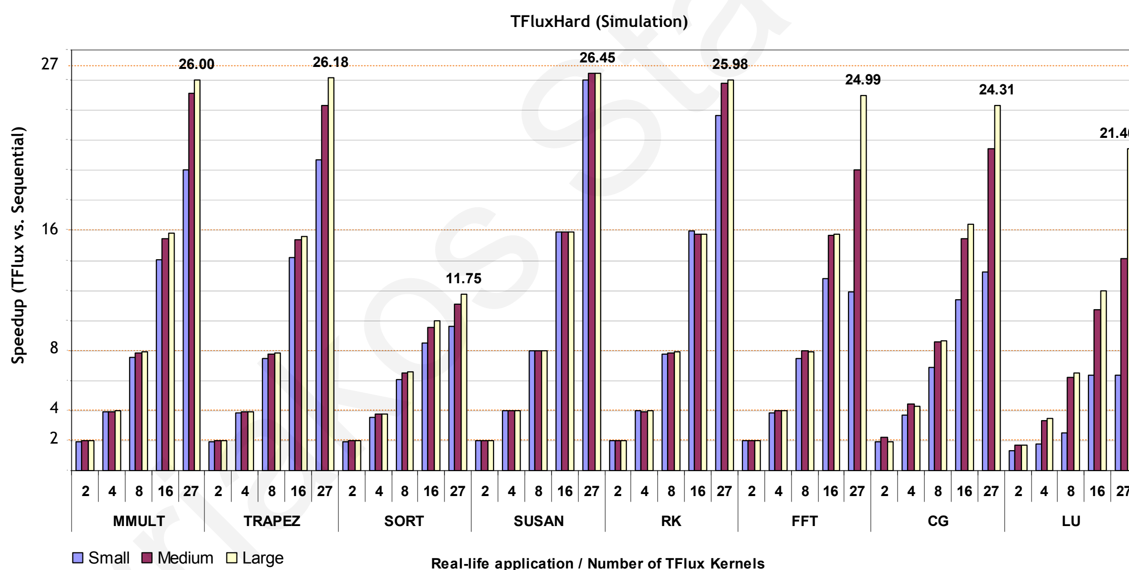


Figure 100: The performance of TFluxHard for the real-life applications.

From the experimental results it is possible to draw two conclusions which are common for all applications. First, as the input size increases, TFluxHard achieves better performance as it better amortizes the parallelization overheads. The second common observation is related to the increase

Kyriakos Stavrou

of the speedup with the number of processors which is justified by the ability of TFluxHard to achieve good scalability.

As for the per-application performance results these are as follows.

MMULT

As explained in Section 7.2.1 *MMULT* is an embarrassingly parallel application. As can be seen from Figure 100, TFluxHard achieves almost linear speedup for this application which also scales well as the number of TFlux Kernels increases.

Referring to Figure 101 that depicts the average usage of the TFlux Kernels for this application, it is possible to see that the Kernels spend more than 95% of their time executing "useful" work. Also notice that the average usage improves with the input size, which is due to the decrease of the *relative* time spent for DThread handling as the input size increases. Moreover, it is possible to observe a small decrease of the usage, especially for the small input sizes, as the number of TFlux Kernels increases. This is justified by the fact that for configurations with more Kernels it is more likely that their completion time will be different leading to load imbalance among the different Kernels.

TRAPEZ

TRAPEZ has a similar behavior to *MMULT* as they have a similar Synchronization Graph. The main difference between the two applications is that *MMULT* causes much higher stress to the memory hierarchy (the large input size leads to an L1 Data Cache miss rate in the order of 50%). As can be seen by the experimental data, this does not affect the performance *MMULT* is able to achieve with TFluxHard. Consequently we may conclude that the performance delivered by TFluxHard scales even in the presence of high memory pressure.

Comparing the usage of these two applications (*TRAPEZ* and *MMULT*) it is possible to observe that the usage of TRAPEZ is more sensitive to the input size (Figure 102). This is related to

Kyriakos Stavrou

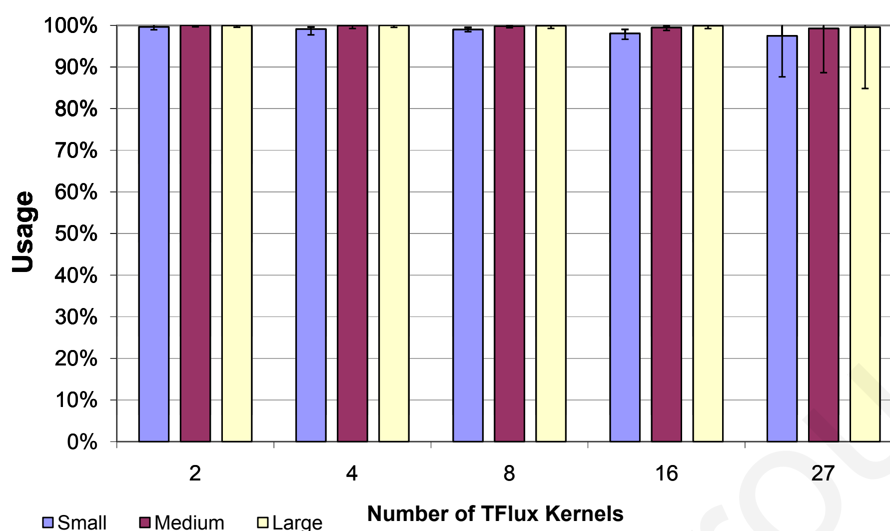


Figure 101: TFlux Kernels usage for *MMULT*. Solid-bars depict the *average* usage among all Kernels whereas the error-bars the minimum and maximum value.

the fact that the DThreads of *MMULT* are significantly more coarse grained compared to *TRAPEZ* (Figure 98).

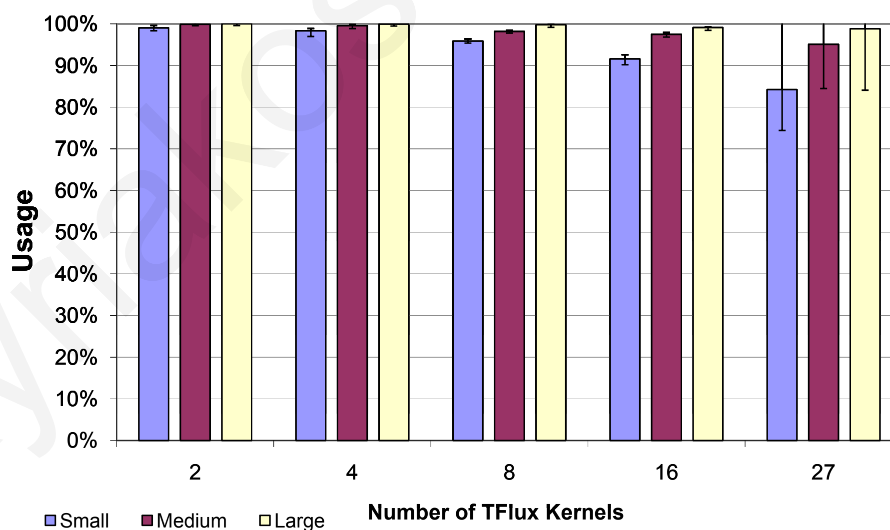


Figure 102: TFlux Kernels usage for *TRAPEZ*. Solid-bars depict the *average* usage among all Kernels whereas the error-bars the minimum and maximum value.

Kyriakos Stavrou

SORT

The third application is *SORT*. Its performance differs significantly compared to the other benchmarks. This behavior is justified by the particular characteristics of this application. As explained in Section 7.2.4 *SORT* consists of two phases; a first phase that sorts the partial arrays and a second phase which merges these sorted partial arrays. Although the first step scales very well, the second step does not.

The usage of Kernels for *SORT* is depicted in Figure 103. From the experimental data it is possible to observe that the average usage for *SORT* is significantly smaller compared to *MMULT* and *TRAPEZ*. This is due to the fact that during the second phase only a subset of the Kernels are executing DThreads. This can be seen from the minimum and maximum values of the usage. In particular, whereas the maximum value is never less than 97%, the minimum and average values decrease with the number of Kernels.

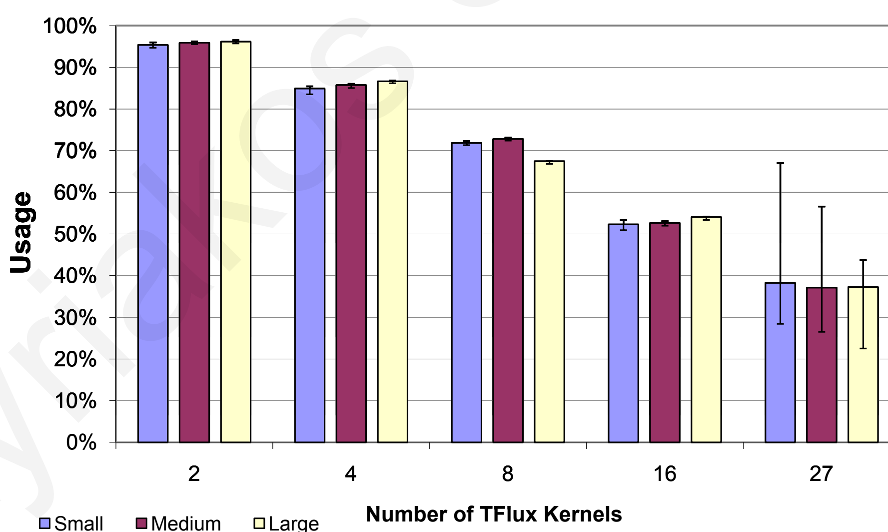


Figure 103: TFlux Kernels usage for *QSORT*. Solid-bars depict the *average* usage among all Kernels whereas the error-bars the minimum and maximum value.

Kyriakos Stavrou

SUSAN

The performance of *SUSAN* is similar to that of *MMULT* and *TRAPEZ* despite the barrier that exists between the two parallel loops of this application (Section 7.2.3). This is justified by two facts. The first is that TFluxHard is able to efficiently enforce such barriers without significant performance loss whereas the second is that the DThreads for *SUSAN* are coarser leading to a smaller relative effect for that barrier.

The barrier that exists between the two loops of *SUSAN* is expected to create some imbalance to the Kernels as it is unlikely that all will reach this synchronization point at the same time. As such, the Kernel usage for this application is expected to decrease. However, as can be seen by Figure 104 that depicts the usage of *SUSAN*, this application has higher usage compared to *TRAPEZ* and *MMULT*. According to the experimental results, the usage for *SUSAN* is never lower than 99% which is justified by the large granularity of its DThreads. In particular, they are larger than *MMULT* DThread's by a factor of 2 and *TRAPEZ* DThread's by a factor of 35 (Figure 98).

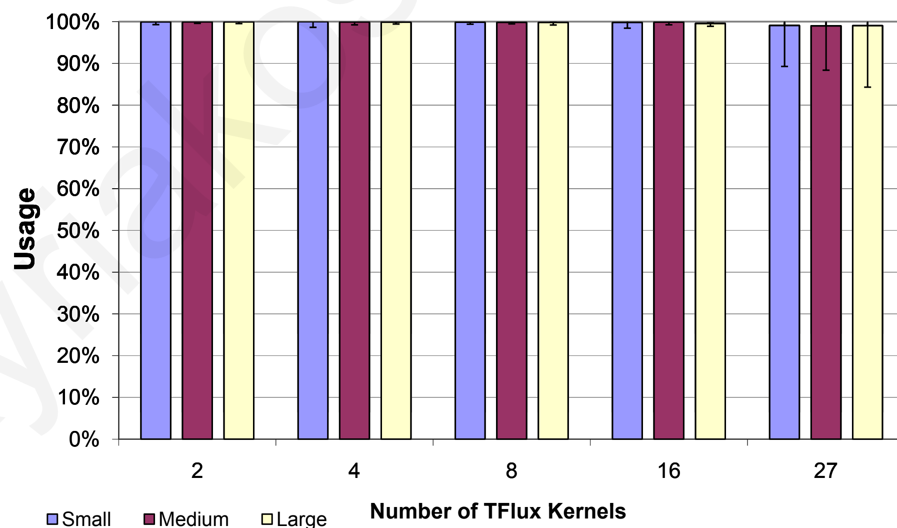


Figure 104: TFlux Kernels usage for *SUSAN*. Solid-bars depict the *average* usage among all Kernels whereas the error-bars the minimum and maximum value.

Kyriakos Stavrou

RK

Regarding *RK*, its performance is similar to *MMULT*, *TRAPEZ* and *SUSAN*. For *RK* TFluxHard also achieves almost linear speedup despite the fact that it has barriers between all its parallel loops (Section 7.2.5). This is justified by the same reasons as the ones mentioned for *SUSAN*, *i.e.* that TFluxHard is able to efficiently enforce the barrier synchronizations and the fact that the application's DThreads are coarse-grained.

The usage of the TFlux Kernels for *RK*, which is depicted in Figure 105, is slightly lower than the usage of *SUSAN* despite the fact that the DThreads of *RK* are larger, by a factor of 7, than those of *SUSAN* (Figure 98). However, as can be seen from Figure 99 that depicts the L1 data cache miss rate for these applciations, *RK* has a miss rate of approximately 3.6% for the small input size, 8.6% for the medium and 9.3% for the large input size. For *SUSAN* however, the miss rate is in the order of 0.3% regardless the input size. The larger number of data cache misses for *RK* create an imbalance on the execution time of the DThreads. Consequently, this results in a negative effect on the usage. The Kernels with DThreads that complete faster their execution have to wait at the synchronization barrier until the DThreads of the other Kernels also complete.

FFT

FFT also achieves very good performance with a speedup of 24$\times$ for 27 TFlux Kernels. This speedup however, is slightly lower compared to the previously analyzed benchmarks (with the exception of *SORT*). The reasons for this behavior are twofold. The first is that the Synchronization Graph for this application is significantly more complex while the second is the smaller number of DThreads which limits the exploitable parallelism. Another observation that can be made for *FFT* is related to the effect of the input size on the achievable performance which is larger compared to the other applications. This behavior is justified by the fact that for *FFT* it is not only the number of DThreads that increases but also their dynamic instructions.

Kyriakos Stavrou

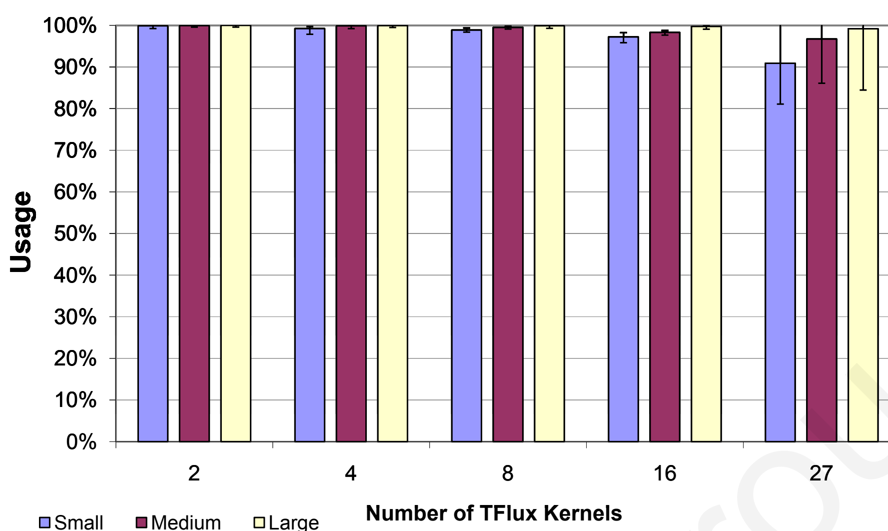


Figure 105: TFlux Kernels usage for *RK*. Solid-bars depict the *average* usage among all Kernels whereas the error-bars the minimum and maximum value.

Figure 106 depicts the usage of the TFlux Kernels for *FFT*. From the experimental results it is possible to observe that the usage is always larger than 94% for the configurations with 2, 4, 8 and 16 Kernels. As for the configuration with 27 Kernels the average usage decreases significantly especially for the small input size (68% for the small, 78% for the medium and 92% for the large input sizes respectively). This behavior is justified by the small number of DThreads of *FFT* which has 128 DThreads for the small input size, 256 for the medium and 512 for the large size. Whereas for the configuration for which the number of Kernels is a power of 2 the distribution of DThreads to the Kernels can be even, this is not the case for the 27-Kernels configuration. The fact that the DThreads of FFT are coarse grained (in the order of 230K, 549K and 2.6M instructions for the small, medium and large input sizes) increases the effect of this imbalance on usage.

CG

The next application we analyze is *CG*. Although *CG* has a highly complex Synchronization Graph with fine grained DThreads, it is possible to observe that TFluxHard is able to achieve a

Kyriakos Stavrou

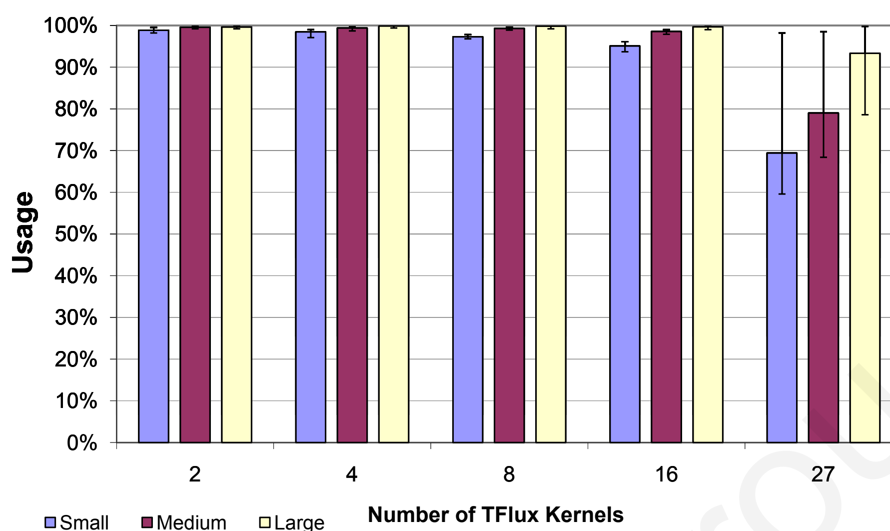


Figure 106: TFlux Kernels usage for *FFT*. Solid-bars depict the *average* usage among all Kernels whereas the error-bars the minimum and maximum value.

speedup that exceeds 24× for 27 Kernels. An important factor towards this good performance is the exploitation of the Iteration Level Dependencies which leads to a performance benefit ranging from 10% to 15%.

As can be seen from Figure 107 that depicts the usage of the TFlux Kernels for *CG*, usage is sensitive to both the number of TFlux Kernels and the input size. This is justified by the fine-grained DThreads of CG and the complexity of its Synchronization Graph. Another factor that significantly affects the usage is the large number of cache misses (Figure 99) that increases the imbalance between the execution time of the DThreads.

LU

Finally, the *LU* benchmark also shows good performance, although the speedup achieved is the lowest compared to the other applications with the exception of *SORT*. The main reason that affects the performance of *LU* is the complexity of its Synchronization Graph which leads to decreased parallelism. As such, for a non negligible fraction of the execution time a number of

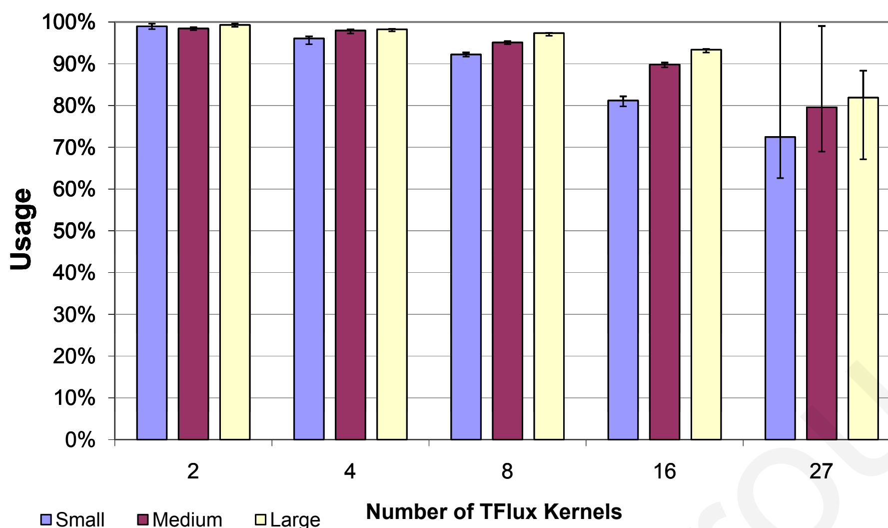


Figure 107: TFlux Kernels usage for *CG*. Solid-bars depict the *average* usage among all Kernels whereas the error-bars the minimum and maximum value.

Kernels remain underutilized. This benchmark, however, is the one that benefits the most from the exploitation of dataflow scheduling. In particular, scheduling with Iteration Level Dependencies give to this application a performance benefit in the order of 20%.

As for the usage of *LU* (Figure 108) it is possible to observe that it does not show a clear trend. On one hand, as the number of DThreads and their size increases with the input size, usage improves due to better amortization of the parallelization overheads. On the other hand, as the number of DThreads increases, the number of dependencies and synchronization points also increase.

9.2.2 TFluxSoft

The performance results for TFluxSoft for the real-life applications, when executed on the simulated system (*TFluxSim* - see Section 8.1.3.1), are depicted in Figure 109. Notice that for TFluxSoft, collecting usage statistics was found to have a non-negligeble impact on the execution

Kyriakos Stavrou

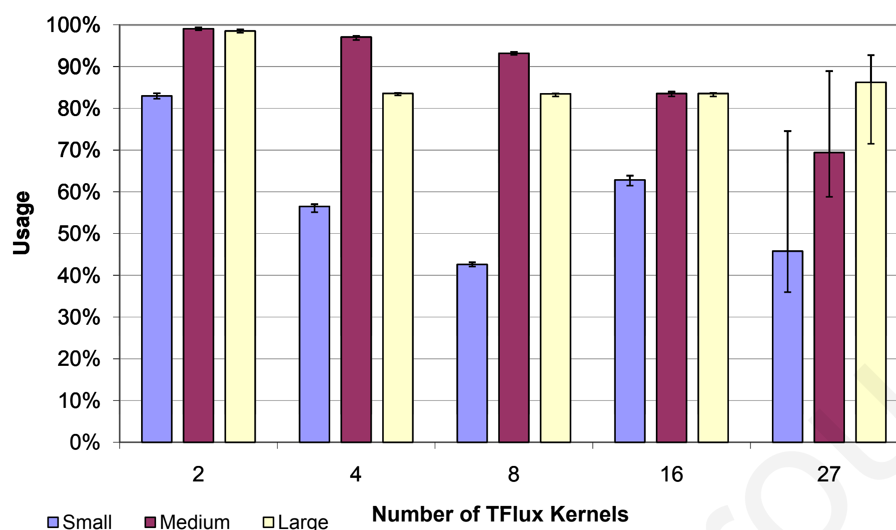


Figure 108: TFlux Kernels usage for *LU*. Solid-bars depict the *average* usage among all Kernels whereas the error-bars the minimum and maximum value.

of the application. As such, usage is not reported here. The reason for this is related to the need to add to the code multiple extra functions that are invoked very often thus affecting the dynamic execution. This effect was not observed for TFluxHard as all the management of the statistics was done by the hardware module without interfering with the execution of the application.

According to the analysis presented in Section 9.1, that quantified the minimum DThread size for TFluxHard and TFluxSoft, the former does not require as coarse grained DThreads as the latter. In particular, TFluxHard is able to achieve speedup with DThreads larger than 271 instructions whereas TFluxSoft with DThreads larger than 925. This fact indicates that the parallelization overheads of TFluxSoft are larger than those for TFluxHard. The granularity of the DThread is the main factor that defines this performance difference.

The performance results for TFluxSoft, which are presented in Figure 109, verify this. In particular, comparing the performance of TFluxHard and TFluxSoft it is possible to observe that the former always outperforms latter.

Kyriakos Stavrou

The two benchmarks for which the performance of TFluxHard and TFluxSoft differs significantly are the *CG* and *LU* applications. *CG* contains 5 TFlux loops which are executed multiple times and consequently consume the largest portion of the execution time. As can be seen from Table 16 that reports the size of all the DThreads of *CG*, only one of the loops has DThreads that are large enough to amortize the parallelization overhead of TFluxSoft. Nevertheless, given that the overheads are smaller for TFluxHard and can therefore be amortized with smaller sized DThreads, all five loops have DThreads that are large enough for efficient TFluxHard execution. These facts justify the good speedup achieved by *CG* for TFluxHard and the smaller speedup values observed for TFluxSoft.

This same reason is what justifies the smaller speedup values observed for *LU* for TFluxSoft. Comparing the performance of *CG* and *LU* it is possible to observe that TFluxSoft delivers higher speedup for the former than for the latter even though the DThreads of *LU* are slightly larger compared to the DThreads of *CG*. What justifies this behavior is that one of the mostly executed TFlux Loops of *CG* (TFlux Loop 4) has a very large number of instructions (in the order of 42K) whereas this does not apply for *LU*.

Finally, Figure 110 depicts the performance results for TFluxSoft which were collected through *native execution* on the IBM3650 machine (Section 8.1.3.2). As can be seen from these results, native execution validates the simulation data.

9.3 Synthetic Applications

In this Section we present the performance of TFluxHard and TFluxSoft for the synthetic applications presented in Section 7.3. The main target of this analysis is to quantify the performance of TFlux for a number of basic execution constructs. Section 9.3.1 focuses on TFlux Loops whereas Section 9.3.2 on applications with more complex dependencies.

Kyriakos Stavrou

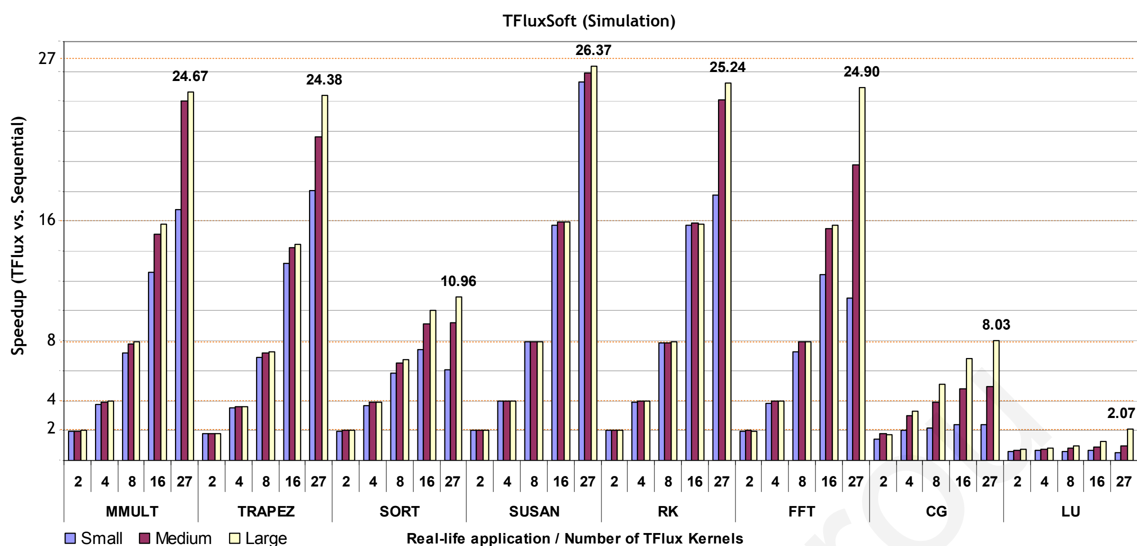


Figure 109: TFluxSoft performance for the real-life applications (simulation results).

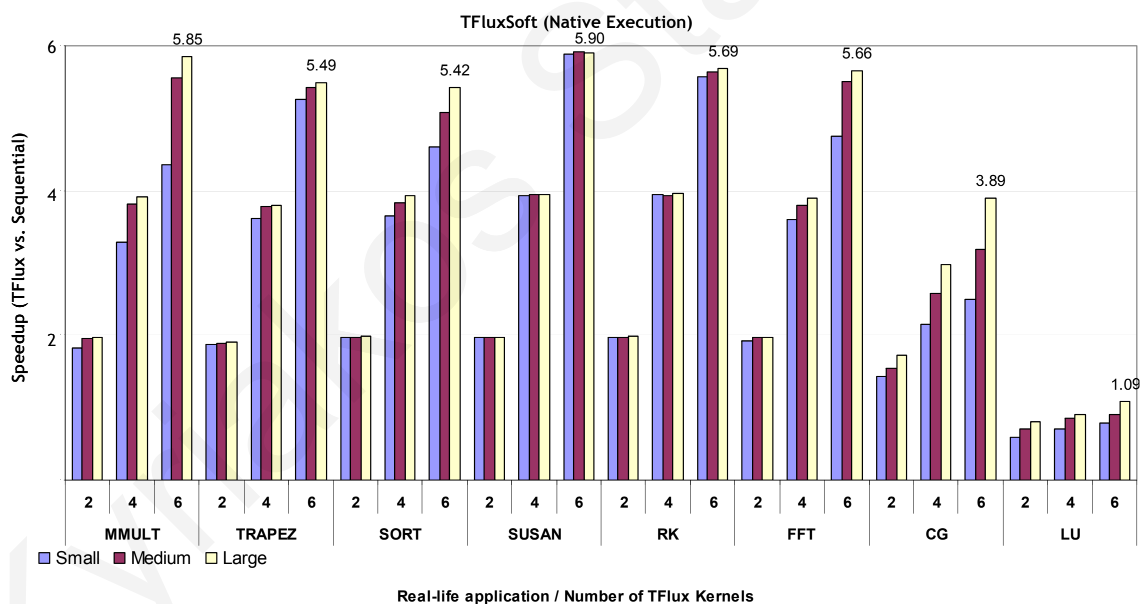


Figure 110: TFluxSoft performance for the real-life applications (native execution results).

9.3.1 TFlux Loops Dependencies

The applications analyzed in this Section have dependencies at the level of TFlux Loops. This set consists of the synthetic applications *L1*, *L2*, *L2R* and *L4* that have been presented in

Section 7.3.2. For the performance analysis we first present the experimental data for TFluxHard and then for TFluxSoft.

9.3.1.1 TFluxHard

Figure 111 depicts the performance of TFluxHard for the aforementioned synthetic applications. For each application, experiments have been conducted for different number of TFlux Kernels as well as for different computational load executed by each DThread of the TFlux Loops. The chart presents the computational load in terms of calls to the load function (Section 7.3). The number of instructions that correspond to each such computational load has been presented in Table 19.

From the experimental results it is possible to draw two conclusions that are common for all applications. The conclusions are that the performance increases with the computational load and the number of TFlux Kernels. These trends are the same as those that have been observed for the real-life application results analyzed in Section 9.2. What justifies these trends is the better amortization of the parallelization overheads as the computational load increases and the ability of TFlux to scale leading to a consequent performance increase with the number of TFlux Kernels.

Figures 112-(a) to 112-(d) depict the speedup achieved by TFluxHard for these synthetic applications as a percentage of the theoretical maximum, *i.e.* the *"efficiency"* of TFluxHard defined as $speedup/NumKernels$. From these data it is possible to observe that for all cases TFluxHard achieves a speedup of at least $0.9\times$ of the theoretical maximum when the computational load executed by each DThread includes 16 calls to the load function (1797 instructions - see Table 19/page161). Most importantly, this number is constant among all applications leading to the conclusion that TFluxHard is able to handle efficiently the dependencies between the parallel loops. Another observation that can be made is that TFluxHard scales very well to configurations

Kyriakos Stavrou

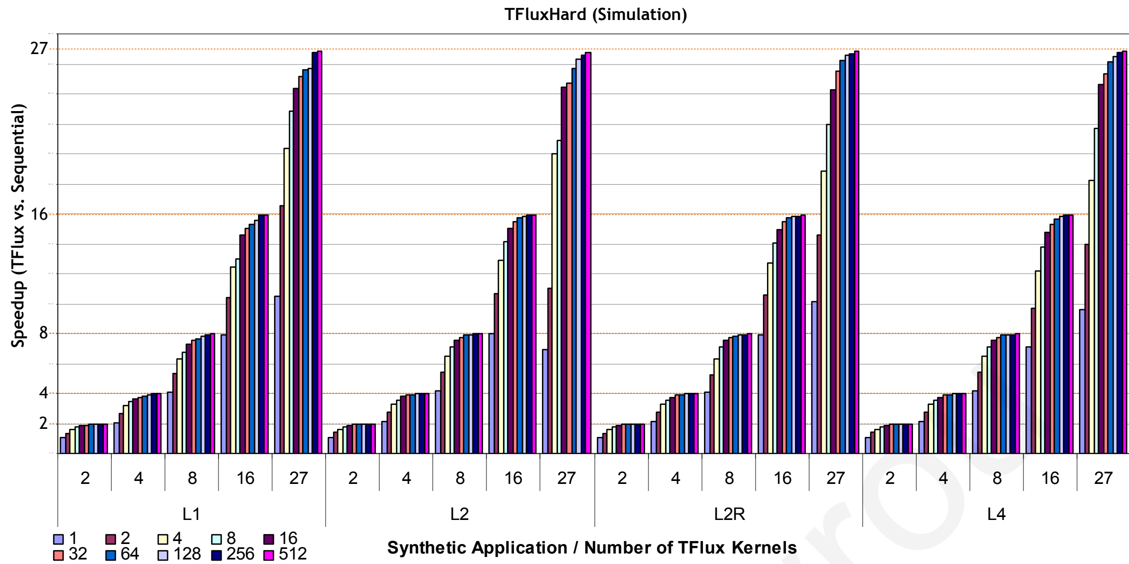


Figure 111: TFluxHard performance for the synthetic applications with dependencies at the level of TFlux Loops.

with larger number of TFlux Kernels. This is justified by the experimental results that show the number of Kernels to have a very small impact on the speedup as a percentage of the theoretical maximum.

9.3.1.2 TFluxSoft

The experimental results of TFluxSoft for these applications are depicted in Figure 113. Comparing the performance of TFluxSoft to the performance of TFluxHard it is possible to see that the trends are the same. In particular, the speedup achieved increases with the number of TFlux Kernels and the computational load executed by each DThread. However, for TFluxSoft to achieve a specific speedup value it needs coarser grained DThreads compared to TFluxHard due to the fact that it suffers from higher parallelization overheads.

Kyriakos Stavrou

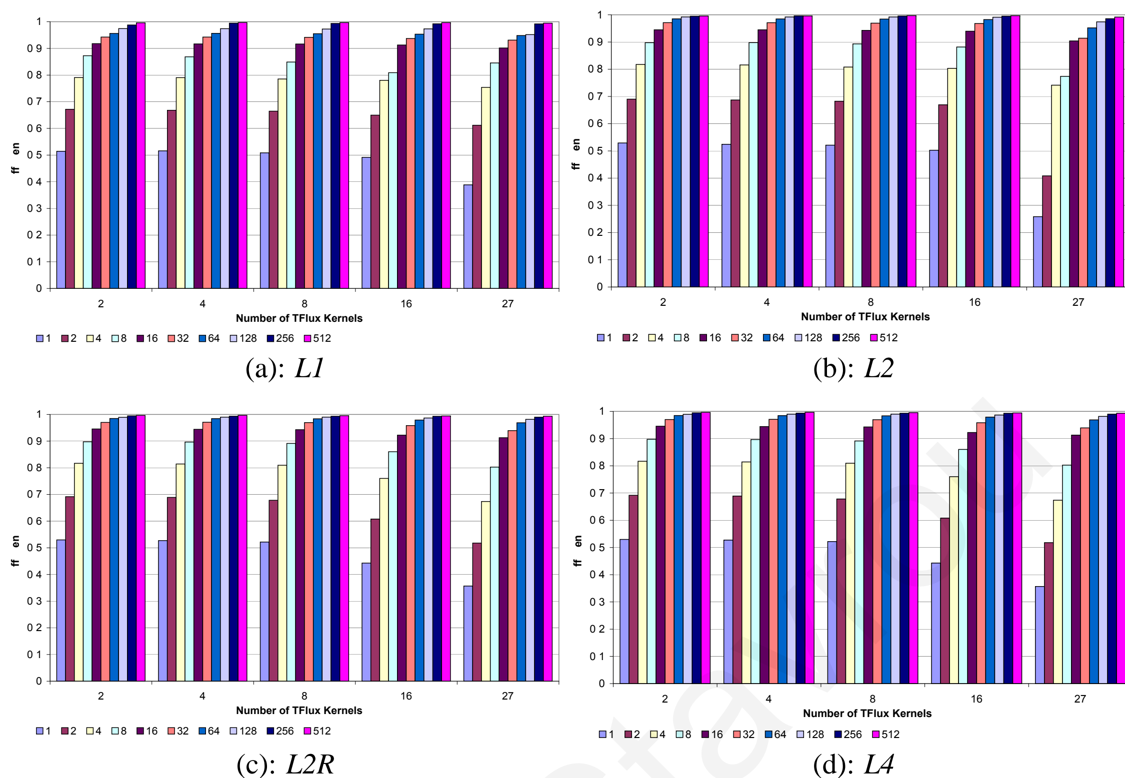


Figure 112: The *efficiency* of TFluxHard for the synthetic applications with dependencies at the level of TFlux Loops.

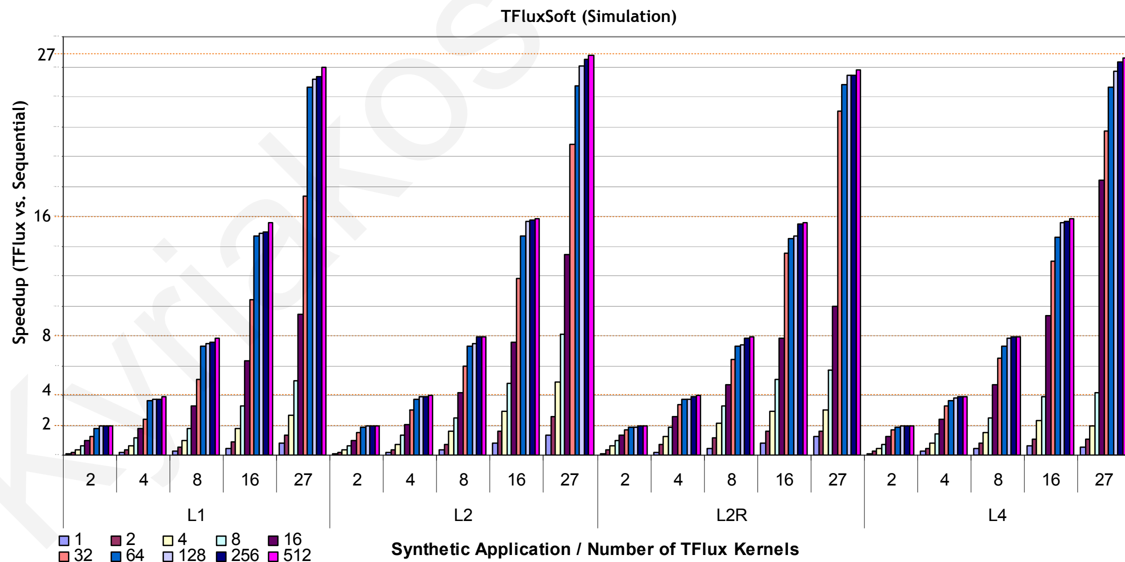
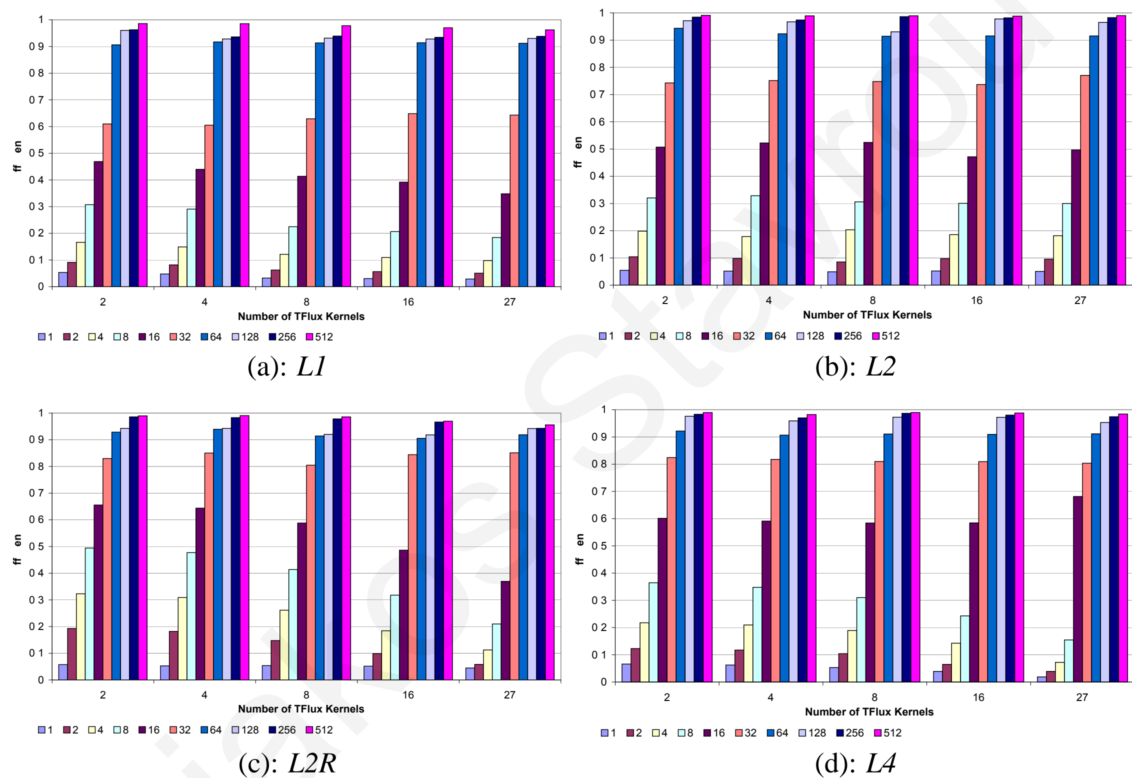


Figure 113: TFluxSoft performance for the synthetic applications with dependencies at the level of TFlux Loops (simulation results).

Kyriakos Stavrou

As can be seen from Figures 114-(a) to 114-(d) that depict the speedup as percentage of the theoretical maximum (*efficiency*), TFluxSoft achieves a speedup of 90% of the theoretical maximum when DThreads execute a computational load consisting of 64 calls to the load function (7029 instructions *i.e.* 64 calls to the *load* function). However, as has been shown in the previous Section, TFluxHard achieves the same performance for finer grained DThreads (1797 instructions *i.e.* 16 calls to the *load* function).



(a): *L1* (b): *L2*

(c): *L2R* (d): *L4*

Figure 114: The *efficiency* of TFluxSoft for the synthetic applications with dependencies at the level of TFlux Loops.

These simulation results are in full agreement with the native execution results which are presented by Figure 115. Most importantly, they also prove TFluxSoft is able to achieve performance of 90% of the theoretical maximum for DThread sizes of 7029 instructions (64 calls to the *load* function).

Kyriakos Stavrou

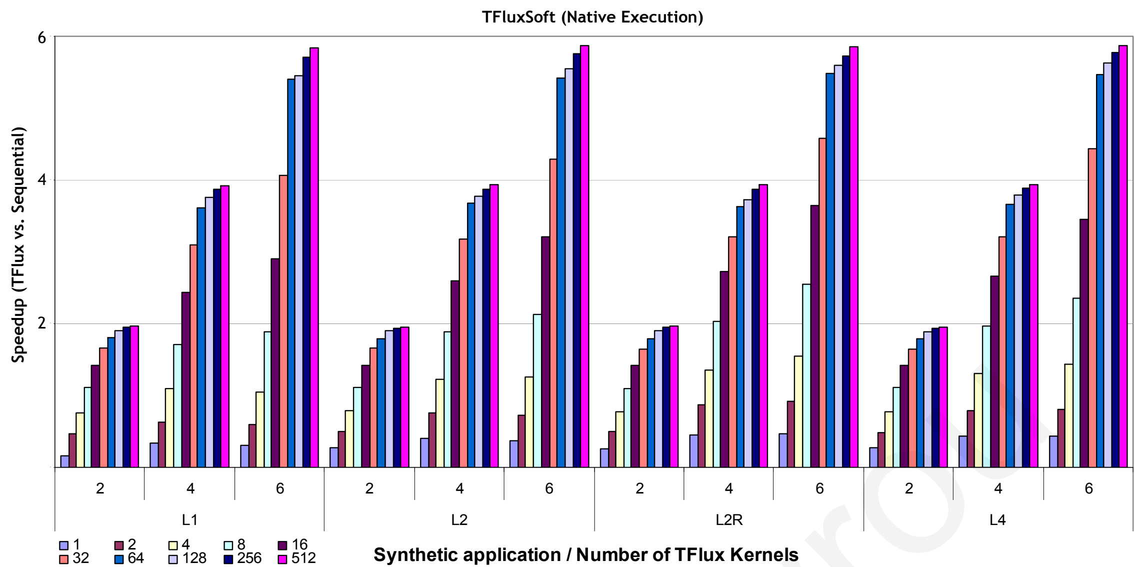


Figure 115: TFluxSoft performance for the synthetic applications with dependencies at the level of TFlux Loops (native execution results).

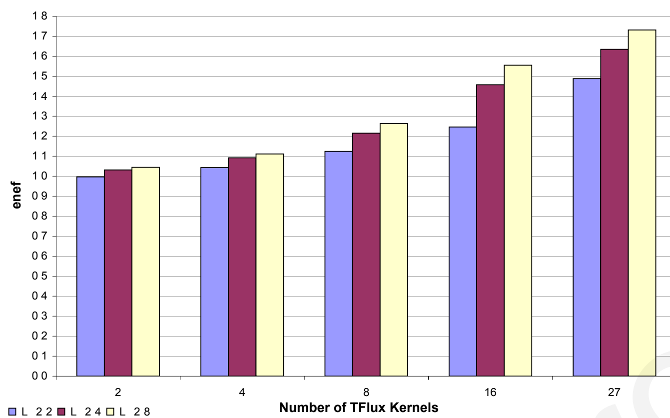
9.3.2 Applications with Complex Dataflow Dependencies

This Section presents the performance of the TFluxHard and TFluxSoft for the synthetic applications with complex dataflow dependencies, *i.e.* applications *ILD2*$_x$, *BINARY TREE* and *DIAGONAL* which have been presented in Section 7.3.3. Recall that for these applications the metric we use is the benefit, *i.e.* how many times TFlux execution is faster compared to a "traditional" parallel execution model as this has been defined in Section 7.3.3.

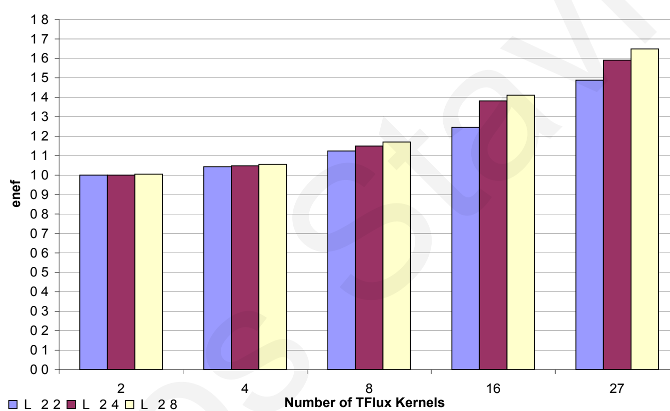
9.3.2.1 ILD2x

The TFlux benefit for the ILD2$_x$ application is depicted in Figures 116-(a), 116-(b) and 116-(c) for the TFluxHard simulation, TFluxSoft simulation and TFluxSoft native execution results respectively. The experiments have been performed for different number of TFlux Kernels and different imbalance factor (*i.e.* the value of parameter x as it has been explained in Section 7.3.3.1).

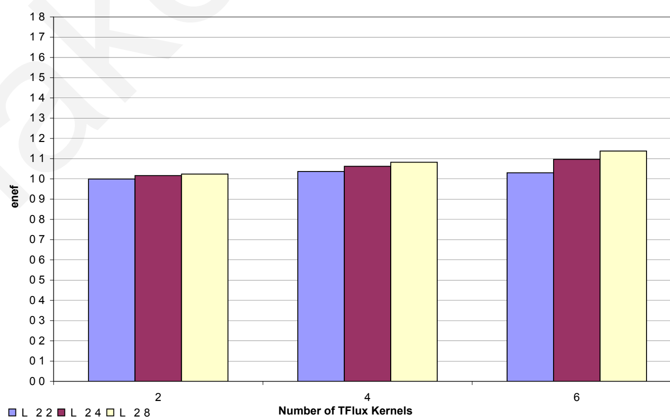
Recall that the larger the value of x the larger the imbalance between the DThreads of the application.



(a): TFluxHard Simulation



(b): TFluxSoft Simulation



(c): TFluxSoft Native Execution

Figure 116: TFlux performance for ILD2_x

Kyriakos Stavrou

As can be seen from the experimental data, the benefit of TFlux over the “traditional” parallel execution model increases with the imbalance (*i.e.* with the value of x). This is due to the fact that the increase of the imbalance leads to an increase of the time the CPUs need to wait to the barrier which is the case for execution under the “traditional” model. However, TFlux is able to convert this barrier to dependencies at the iteration-level of loops allowing the CPUs that have completed the execution of the DThreads of the first loop to proceed to the execution of the DThreads of the second loop. This leads to better usage of the CPUs and therefore to better performance.

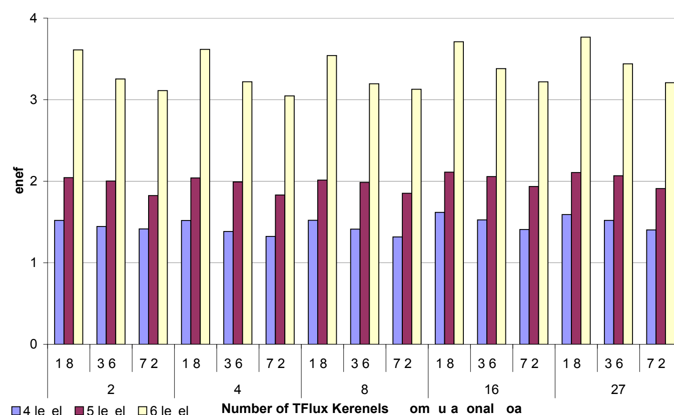
According to the experimental results, the benefit of TFlux also increases with the number of TFlux Kernels. This behavior is due to the fact that the workload executed by each CPU is not the same and as such, the more the imbalance the longer the time spent at the barrier and therefore the larger the potential benefit in overcoming this synchronization point.

Finally, as can be seen from Figure 116, the performance benefit of TFluxHard reaches $1.75\times$ for the execution of $ILD2_8$ on a configuration with 27 TFlux Kernels configuration. As for TFluxSoft, the performance benefit reaches $1.65\times$. Finally, from the native execution results, it is possible to observe that the benefit of TFluxSoft reaches $1.15\times$ for $ILD2_8$ on a configuration with 6 Kernels.

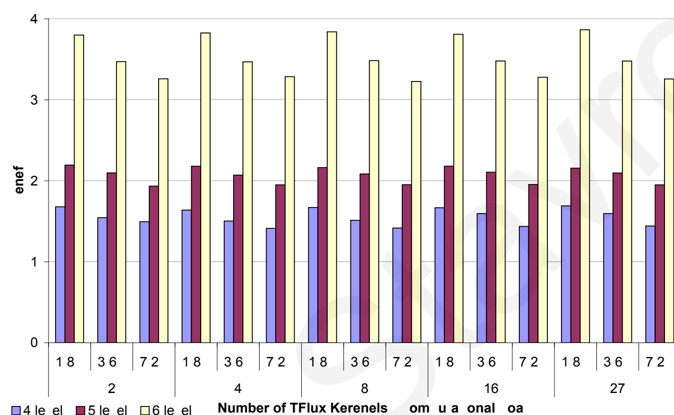
9.3.2.2 BINARY TREE

The TFlux benefit for the *BINARY TREE* application is depicted in Figures 117-(a) for TFlux-Hard, 117-(b) for TFluxSoft simulation and finally by 117-(c) for TFluxSoft native execution results. These experiments were conducted for different number of TFlux Kernels and different computational load executed by the application’s DThreads. Notice however that, for each particular experiment, *all* DThreads execute the same computational load; *i.e.* the load varies between different experiments.

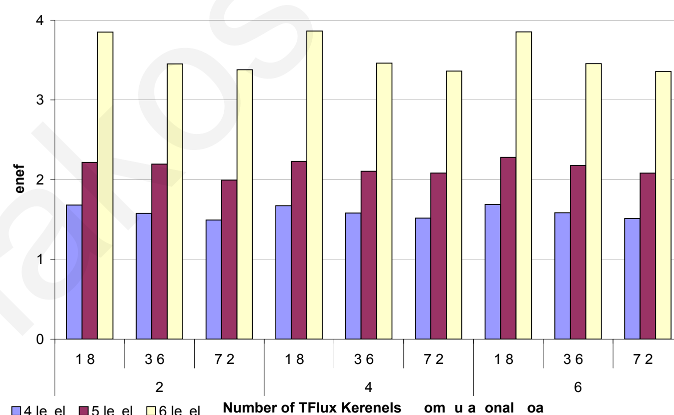
Kyriakos Stavrou



(a) TFluxHard Simulation



(b) TFluxSoft Simulation



(c) TFluxSoft Native Execution

Figure 117: TFlux performance for BINARY TREE

The first observation that can be made from the experimental results is that both TFluxHard and TFluxSoft always outperform the "traditional" parallel execution. This is justified by the fact

Kyriakos Stavrou

that the "traditional" execution introduces unnecessary synchronization points in order to guarantee program correctness. TFlux, however, is able to enforce this consistency with the minimum number of synchronization points. This same reason also explains the increase of TFlux performance benefit with the number of the levels of the *BINART TREE*. In particular, as the number levels of the tree increases the number of unnecessary synchronization points removed by TFlux also increases. This leads to better resource utilization as the CPUs need to spend less time waiting for synchronization and consequently achieve better performance.

The other trend that can be observed from the experimental results is that the performance benefit of TFlux is larger for smaller computational loads. This is due to the decrease of the relative time spent for synchronization and consequently of the potential for improvement, as the computational load increases.

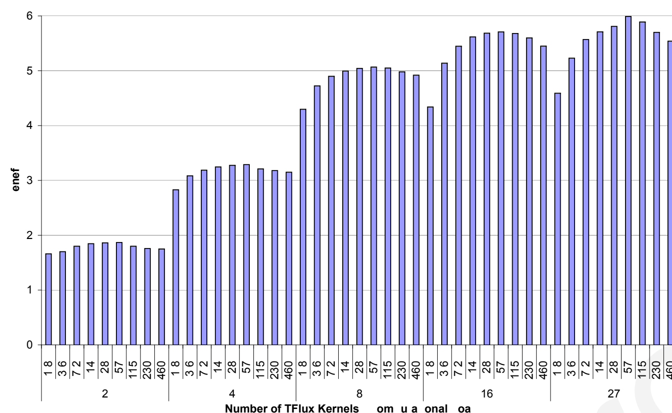
Overall, the experimental results show that both TFluxHard and TFluxSoft outperform the "traditional" parallel processing model for all the experiments performed. More specifically, the performance benefit of TFluxHard ranges from approximately $1.5\times$ to $3.8\times$ for the experiments performed. As for TFluxSoft, this benefit ranges from $1.4\times$ to $3.7\times$. The same applies to the native TFluxSoft execution.

9.3.2.3 DIAGONAL

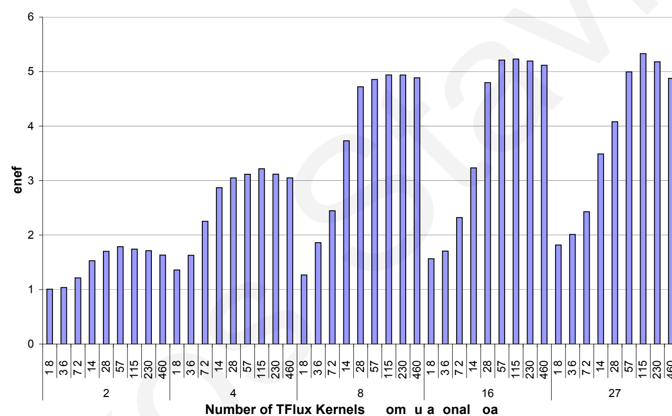
The performance of TFlux for the *DIAGONAL* synthetic application, in terms of additional benefit, is presented in Figure 118-(a) for the TFluxHard system and in Figure 118-(b) and Figure 118-(c) for the TFluxSoft system, simulation and native execution, respectively. The experiments cover configurations with different number of TFlux Kernels as well as different computational load executed by each of the application's DThreads. Similar to the experiments performed

Kyriakos Stavrou

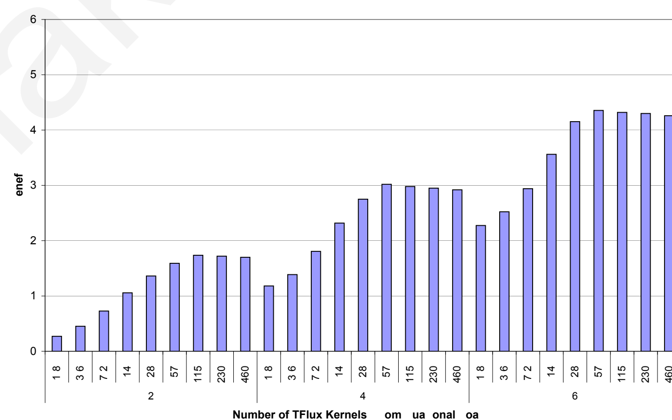
for *BINARY TREE*, for each of the experiments presented in this Section the size of the DThreads varies among the experiments whereas it is constant for the same experiment.



(a): TFluxHard Simulation



(b): TFluxSoft Simulation



(c): TFluxSoft Native Execution

Figure 118: TFlux performance for DIAGONAL

Kyriakos Stavrou

From the experimental data it is possible to conclude that the additional benefit of TFlux depends on both the computational load executed by the DThreads and the number of TFlux Kernels. The performance advantage increases with the computational load and after a certain point ($\sim$ 115K instructions) it starts to decrease. This effect is due to two competing factors. The first, which leads to a decrease of the TFlux advantage, is due to the fact that as the computational load increases, the relative effect of the time needed for synchronization decreases. The other factor, which leads to the increase of the advantage with the computational load, is related to the fact that the larger this load is, the more possible it is for two different CPUs not to complete at the same time with a consequent imbalance which, in order, delays the dependent nodes from initiating their execution. The key characteristic of TFlux that allows it to significantly outperform the baseline is the way scheduling of nodes is done. In particular, whereas for the baseline each CPU takes a node and waits until its consumers complete (see Figure 85), in TFlux, a CPU is always assigned a node ready to execute. This allows TFlux to achieve better resource utilization by avoiding idle cycles while waiting for the producer nodes to complete.

According to the experimental results, for TFluxHard this performance benefit reaches a value of $6\times$ for TFluxHard and $5\times$ for TFluxSoft. Moreover, it is possible to observe that TFluxHard achieves higher speedup values for smaller computational loads compared to TFluxSoft due to the fact that it introduces smaller overheads.

9.4 TFluxSoft Scalability - Operation with Multiple Updaters

As explained in Section 6.3, TFluxSoft is able to operate with more than one *Updater*. This however leads to a tradeoff as on the one hand, more *Updaters* give better performance through more efficient execution of the Thread Update operation whereas on the other hand they consume valuable resources. This Section first analyzes the potential of using multiple *Updaters* and then

Kyriakos Stavrou

studies the aforementioned tradeoff. For this study we use the synthetic applications presented in Section 7.3.4 (*L1*, *L2*, *L2-T1*, *L4*, *L4-T3*, *ILD2* and *ILD4*).

9.4.1 Potential of using Multiple Updaters

The first set of experimental results compares the execution time of the synthetic applications using 1 and 2 *Updaters* for TFluxSoft systems with different number of TFlux Kernels. These results, which are depicted in Figure 119, show the *speedup*, *i.e.* how many times execution with the 2 *Updaters* is faster compared to execution using 1 Updater.

The first observation that can be made from the experimental data is that the potential of using multiple *Updaters* increases with the number of TFlux Kernels. This is due to the fact that the number of *update-requests* that must be served for an application to complete increases with the number of TFlux Kernels. Moreover, the more the TFlux Kernels, the larger the number of requests to be served *during a burst* (see Section 7.3.4 and Table 20).

The second conclusion is related to the effect of the computational load executed by each DThread on the potential of using multiple *Updaters*. As can be seen from the experimental results, the general trend is that the larger the computational load the smaller this benefit is. This behavior is justified by the fact that as the load of each iteration increases the *relative* time spent to serve the dependencies decreases.

The third observation is that the complexity of the program is the dominant factor on the benefit of using multiple *Updaters*. In particular, the longer the time during which the program poses update-requests the larger the benefit of using multiple *Updaters*. This is justified by the more efficient execution of the *Thread Update* operation when multiple *Updaters* exist.

Comparing the experimental results for programs *L1* and *L2*, it is clear that the potential of multiple *Updaters* increases for L2 regardless the number of TFlux Kernels. This is due to the

Kyriakos Stavrou

demanding step, in terms of update-requests, included in *L2* for the synchronization between the two TFlux Loops which is not required for *L1*. The smaller the delay for serving this burst the higher the performance. *L4* has similar behavior to *L2* as the relative time for synchronization is the same for these two applications. With the same rationale it is possible to explain the reason from which multiple *Updaters* deliver less benefit for applications *L2-T1* and *L4-T3* as opposed to *L2* and *L4* respectively. This is justified by the smaller potential that exists, because of the smaller number of *Updater* requests $(2 \cdot 32 \cdot n\ \ vs.\ \ 32 \cdot 32 \cdot n)$.

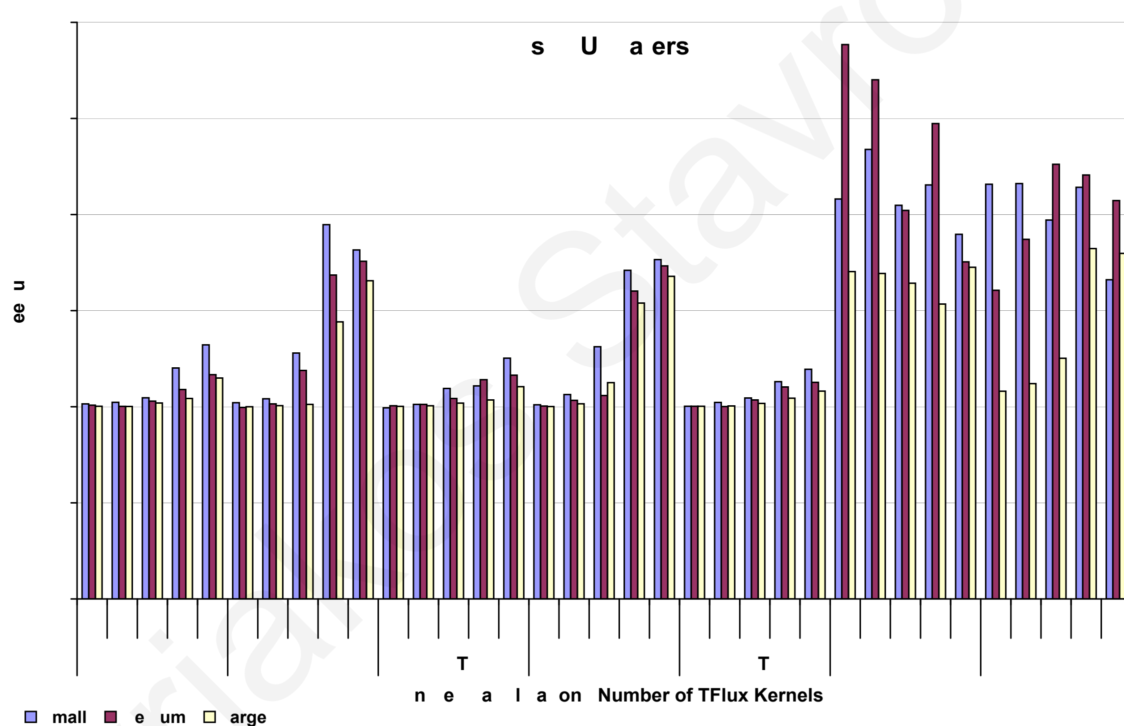


Figure 119: Speedup of executing the synthetic applications using 2 *Updaters* vs. using 1 *Updater*.

Kyriakos Stavrou

To better understand the effect of using multiple *Updaters*, Figure 120 presents the number of update-requests per time interval for the execution of program *L4* on a system with 16 TFlux Kernels and 1, 2 and 4 *Updaters*. Notice that the number of update-requests does not depend on the number of *Updaters* as it is a characteristic of the program. As such, the time required to

serve the burst can not be seen directly from this Figure but rather indirectly given the following observation. For this program, a TFlux Kernel operates without posing update-requests during the period when it is executing the application's DThreads or when no ready DThread exists. This second factor, *i.e.* not having ready DThreads to execute, is the reason for which the period during which a Kernel does not pose update-requests is prolonged when fewer *Updaters* are used. As can be seen from this Figure, having more *Updaters* significantly decreases the time during which a Kernel is idle due to the fact that its waiting DThreads have not been yet notified by the *Updater(s)*.

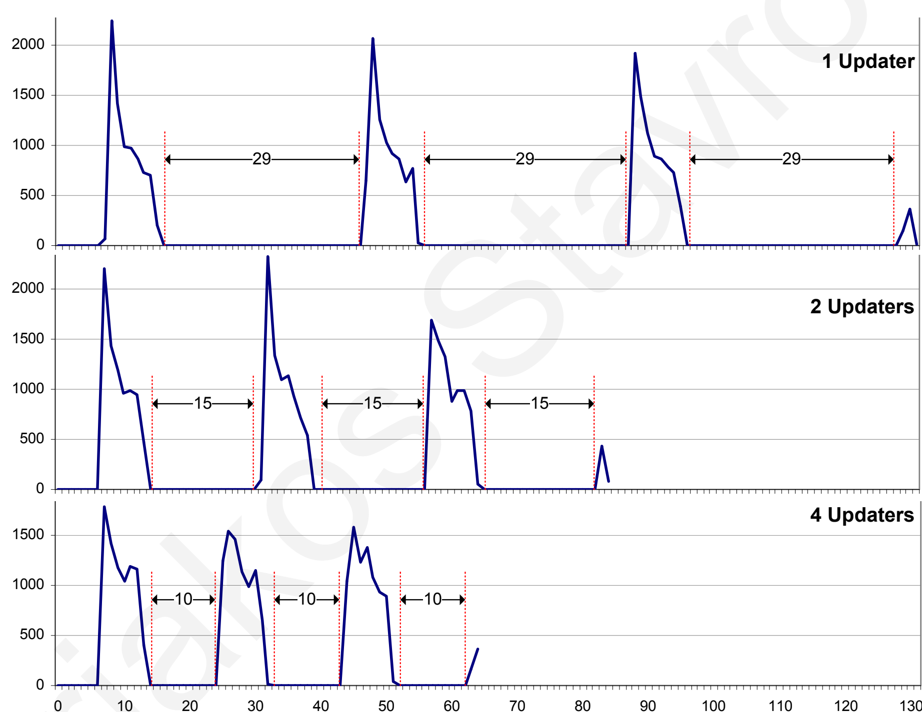


Figure 120: Update-requests per interval for the execution of program L4 with 16 TFlux Kernels for 1, 2 and 4 *Updaters*. Numbers on the horizontal arrows show the length of the corresponding period in terms of time intervals.

The programs for which using multiple *Updaters* deliver the most benefits are *ILD2* and *ILD4*, *i.e.* the programs with iteration-level dependencies. The reason for this is related to the pattern by which the update-requests are inserted into the TUB which, in contrast to the other programs,

is not bursty. In particular, for *ILD2* and *ILD4*, each Kernel poses an update-request each time an L-DThread completes and not only when an L-DThread of the last generation completed (as is the case for the other applications). As such, the update-requests for applications with iteration level dependencies are more evenly distributed in time leading to better potential for improvement.

Whereas Figure 119 depicts the speedup of execution with 2 *Updaters* versus execution with only 1 *Updater*, Figure 121 goes one step further and presents the *additional* speedup that can be achieved when instead of 2 the system has 4 *Updaters*. As can be seen from the experimental results increasing the number of *Updaters* can provide additional performance benefits. This is especially true for the applications that have a large number of update-requests to be served, *i.e.* *ILD2* and *ILD4*.

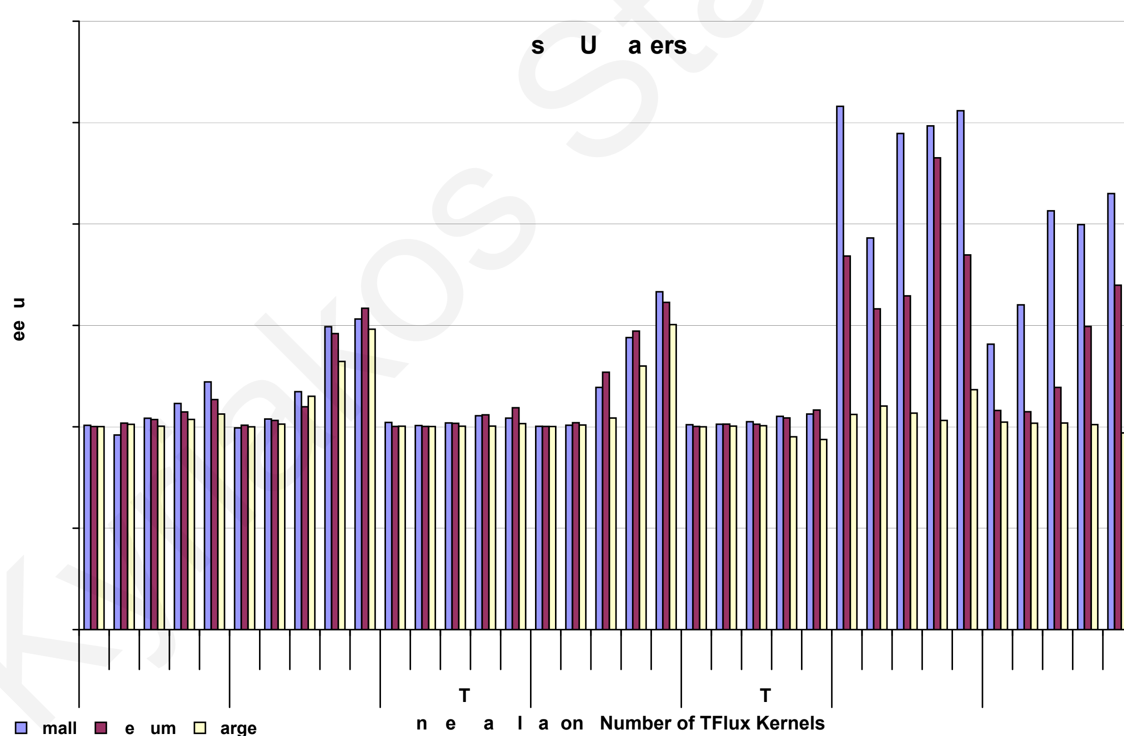


Figure 121: Speedup of executing the synthetic applications using 2 *Updaters* vs. using 1 *Updater*.

Kyriakos Stavrou

9.4.2 Performance Evaluation with Multiple Updaters

According to the results presented in the previous Section, configurations with multiple *Updaters* have the potential for better performance. However, given a system with a *specific* number of processors, increasing the number of *Updaters* means that fewer processors will be available for the execution of application's DThreads leading to a tradeoff. As such, the critical question regards the configuration that leads to the best *overall* performance.

For the experiments presented in this Section, the number of CPUs of the system is kept constant, *i.e.* having more *Updaters* decreases the number of Kernels. The experimental results are presented in Figures 122 and 123. Figure 122 presents the speedup of execution with 2 *Updaters* compared to execution with 1 *Updater* whereas Figure 123 the speedup of a system with 4 *Updaters* compared to a system with 1 *Updater*.

The first observation that can be made from the experimental results is related to the load executed by each iteration; in particular, as this load increases the potential of using multiple *Updaters* decreases. This is justified by the fact that the increase of the computational load executed by each loop iteration leads to a decrease of the relative time spent for synchronization. The second observation regards the total number of system nodes. As the number of CPUs increases using multiple *Updaters* is beneficial as the application's requirements for synchronization increase. However, for systems with small number of CPUs using multiple *Updaters* often leads to performance degradation.

Finally, notice that the potential of using multiple *Updaters* is, in a major degree, defined by the application's Synchronization Graph. Although the synchronization to computation ratio does is not depicted in these Figures, the real trend is that the benefit from using multiple *Updaters*

Kyriakos Stavrou

increases with this ratio. As expected, the applications for which it is better to use multiple *Updaters* even though less execution cores are available for application DThreads, are those exploiting iteration-level dependencies. This is due to the large amount of synchronization requirements for applications with iteration-level Dependencies.

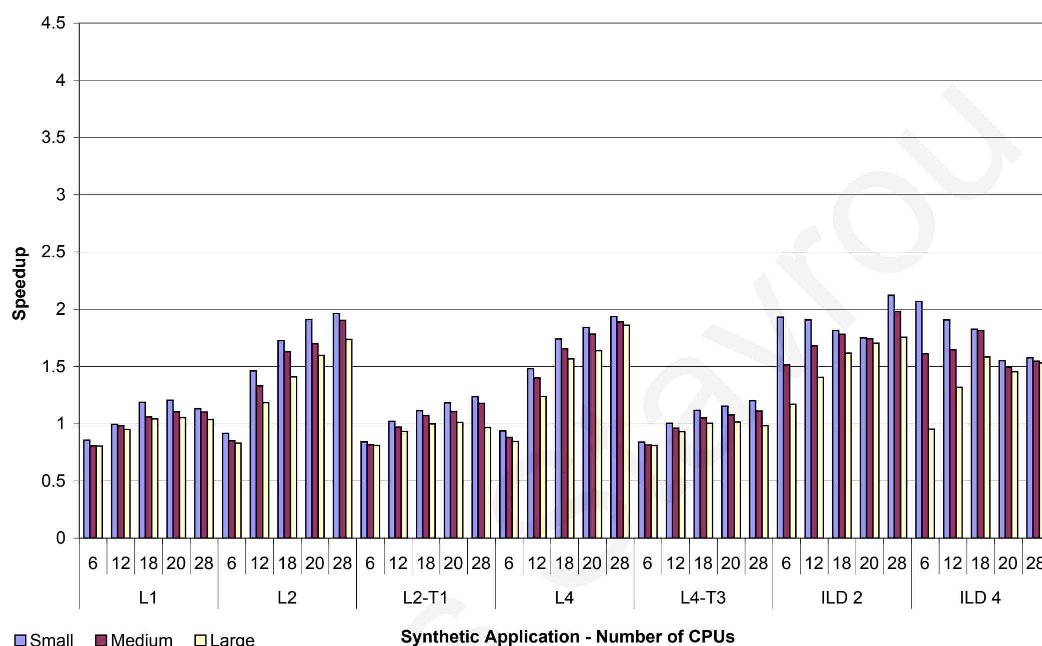


Figure 122: Performance comparison of a system with 2 *Updaters* versus a system with 1 *Updater*. For these results the total number of the system's CPUs is constant, *i.e.*. using more *Updaters* decreases the number of TFlux Kernels.

Kyriakos Stavrou

Figure 123 goes one step further and evaluates configurations with 4 *Updaters*. As can be seen from the experimental results using even more *Updaters* is better in some situations especially when the load executed by each DThred is small. However, as the executed computational load increases this benefit decreases.

As a concluding remark it is possible to state that using multiple *Updaters* can deliver better performance compared to configurations with only one *Updater* when two conditions stand: the first is for the Synchronization Graph of the application to be complex whereas the second for

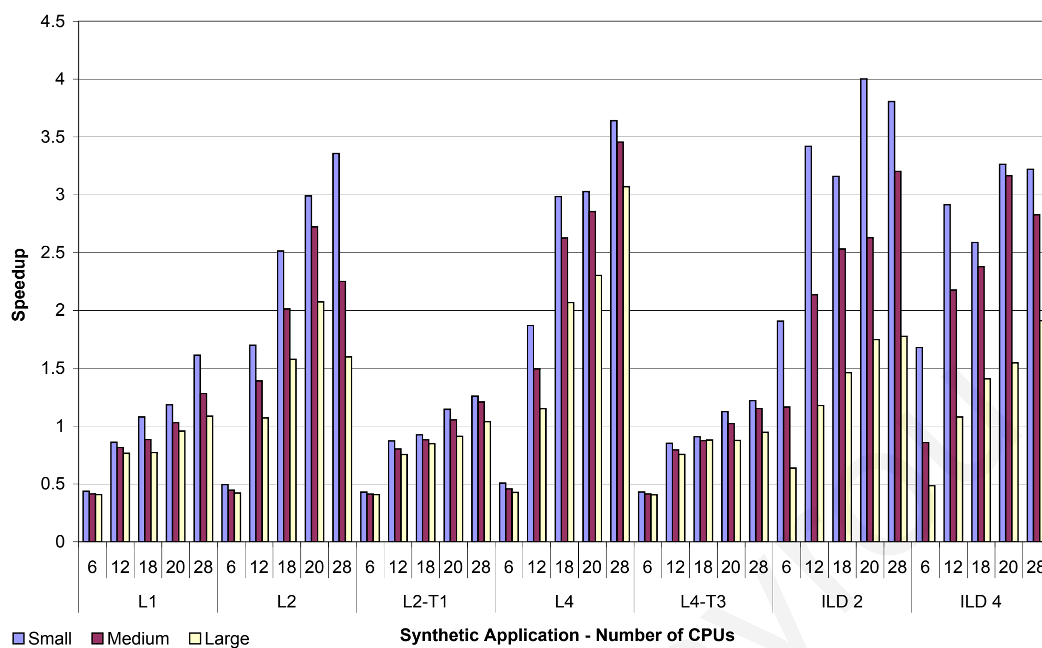


Figure 123: Performance comparison of a system with 4 *Updaters* versus a system with 1 *Updater*. For these results the total number of the system's CPUs is constant, *i.e.*. using more *Updaters* decreases the number of TFlux Kernels.

the DThreads to execute rather small computational load. As this load increases however, the synchronization to computation ratio will also decrease and finally will lead to a situation where it will be better to use as many resources as possible to execute the application's computational load. The same applies for applications with simple Synchronization Graphs.

9.5 TFluxHard Scheduler Delay

TFluxHard requires the machine to be equipped with a hardware module that provides the scheduler's functionality. The target of this Section is to quantify the effect the delay of this unit has on the overall performance. For this study we used the synthetic applications presented in Section 9.3 ((*L1*, *L2*, *L4*, *L2R*, *ILD2*, *BINARY TREE* and *DIAGONAL*)). For the experimental results presented in this Section the DThreads of these applications execute only 1 call to the

Kyriakos Stavrou

load() function in order to increase as much as possible the realtive effect of synchronization on performance which, in part, is dependent on the delay of the scheduler. The metric we used is the *relative execution time*, *i.e.* the execution time with the scheduler having a delay of a specific number of cycles versus execution with the scheduler having a delay of only one cycle.

The experimental results which are presented by Figure 124 cover the range of 1 to 128 cycles for the delay of the TFluxHard Scheduler with the baseline being the configuration where the scheduler has a delay of 1 cycle. Moreover, notice that the experiments have been conducted for a configuration with 27 Kernels as the larger the number of Kernels the most sensitive the performance is to the delay of the scheduler. This is due to the fact that given an application the number of operations performed by the scheduler increase with the number of Kernels.

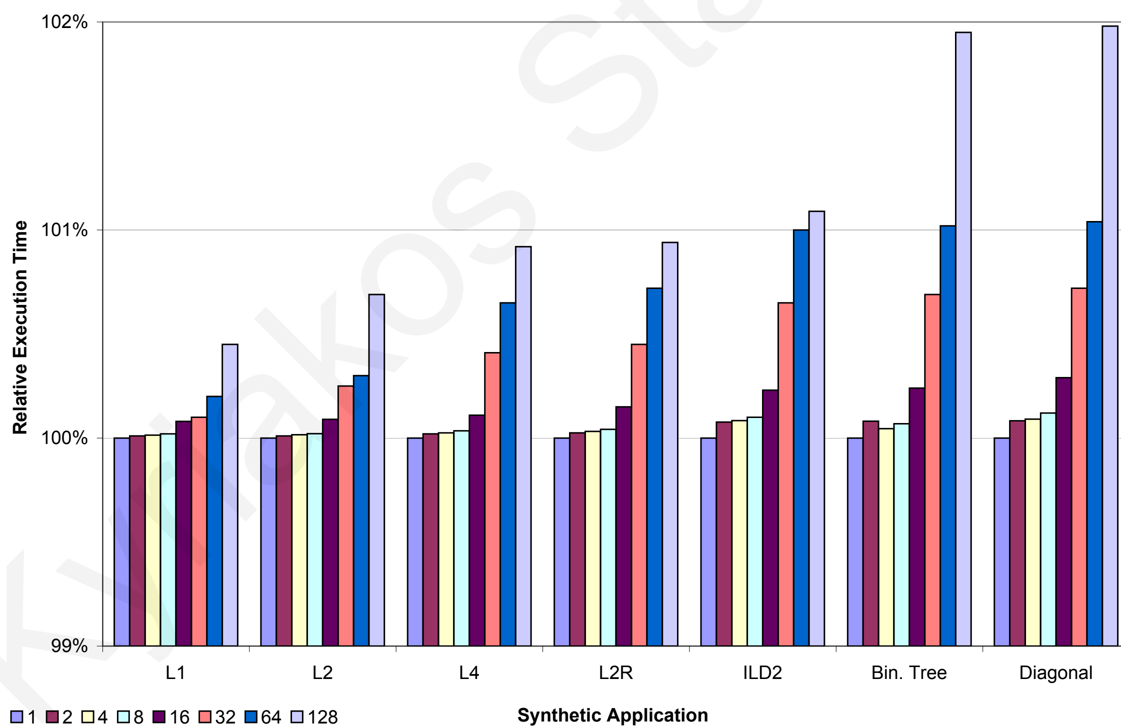


Figure 124: The impact of the Scheduler delay in the performance of TFluxHard for the synthetic applications.

Kyriakos Stavrou

As can be seen from the experimental data, the penalty of increasing the delay of the scheduler from 1 to 128 cycles is never larger than 2%. This effect however is strongly related to the application being executed. The applications that show the largest sensitivity to the delay of the scheduler are the *Binary Tree* and the *Diagonal* ones. This is justified by the fact that for these applications there are many dependencies among DThreads executed by different Kernels. The more complex the dependencies are, the longer the delay in the Scheduler to perform its operations. Consequently the CPU needs to wait longer until a ready DThread is identified, which results in an increased overhead and consequently lower performance.

Kyriakos Stavrou

Chapter 10

Virtualization and Portability

In this Chapter we explore the *Virtualization* and *Portability* of the TFlux Parallel Processing Platform, *i.e.* its ability to hide the details of the underlying system and to be ported to new architectures with the minimum possible effort.

10.1 TFlux Virtualization

With the term *"Virtualization"* we refer to the ability of a system to provide to its user a programming and operation layer that does not depend on the details of the underlying machine. Ideally, the programmer should be able to develop applications for TFlux without being aware of the details of the particular machine. The most important factor, is that this unawareness should not affect the performance. Notice however, that our study about Virtualization does not include any quantitative results but only qualitative arguments.

Providing to its user an abstraction layer that hides all details of the underlying machine is one of the key characteristics of TFlux. The main components that allow TFlux to achieve this

Kyriakos Stavrou

virtualization is its layered design. As explained already in Section 3.1 the user of TFlux is only required to understand the programming layer which hides all implementation- and machine- specific details.

To achieve this goal, during the design of TFlux we avoided any modifications to the existing interfaces. More specifically, all entities of TFlux operate at the user-level in order to allow execution of TFlux applications to all systems that provide the standard functionality of the Linux-based Operating Systems. However, the TFlux Runtime is able to take advantage of functionalities offered by the system for better performance. Such an example is the *pin* of the TFlux Kernels to the different CPUs of the system when this is offered by the Linux distribution that. Attaching the TFlux Kernels to different CPUs has been found to give better and more stable results across different executions.

The implementation of the Scheduler is another component for which special care was taken in order to avoid modifying existing interfaces. For TFluxSoft, the Scheduler could have been integrated with the scheduler of Linux. In addition to the fact that this would require the installation of software patches to udpate the system's scheduler, it would require the Operating System to be aware of the TFlux execution leading to a violation of the interface between the processes and the OS. Similar to that, the Scheduler of TFluxHard is attached to the system's network as a memory mapped device allowing it to be accessed by the TFlux Kernels through simple *load* and *store* instructions.

The TFlux layer that is exposed to the user is the set of TFlux directives which allow defining the boundaries and the dependencies of the DThreads at the high-level code. The abstraction layers of TFlux are responsible to convert this code into a binary that is able to execute on all shared memory Linux-based multiprocessor systems. Most importantly, this code is the same regardless if the target system is TFluxSoft or a TFluxHard.

Kyriakos Stavrou

To prove the ability of TFlux to meet the virtualization goal we executed all applications that have been presented in the previous Sections on a variety of computer systems. The applications were coded using the TFlux directives and through the TFlux compilation toolchain leaded into both a TFluxSoft and a TFluxHard binary. The systems which we used for this study are summarized in Table 24 (details about these systems have been presented in Sections 8.1.3 and 8.1.4). These systems differ in three key components: (1) the number of TFlux Kernels ranging from 1 to 28; (2) the Operating System including machines with different Linux distributions and different versions of the Linux kernel and (3) the ISA of the CPU including systems with x86 and SPARC processors. Whereas for all these systems we were able to try both TFlux incarnations, for TFluxSoft we went one step further and executed the applications on a number of different off-the-shelf systems.

Table 24: Machines used for virtualization study

Name	Nature	ISA	#Cores	CPU	OS	Kernel
TFluxSim	Simulated	SPARC	1-28	UltraSparcII	Fedora Core 3	2.6.13
Bagle	Simulated	SPARC	1-28	UltraSparcII	SuSe 7.3	2.4.14
Tango	Simulated	x86	1-15	Pentium 4	Fedora Core 5	2.6.15
Enterprise	Simulated	x86	1-8	Pentium 4	Red Hat 3	2.4.18
IBM3650	Off-the-shelf	x86	8	Xeon E5320	Fedora Core 6	2.6.22
Thales	Off-the-shelf	x86	4	AMD Opteron	Fedora Core 5	2.6.20
QUAD	Off-the-shelf	x86	4	Core2 QuadCore	Fedora Core 7	2.6.22

Both TFluxHard and TFluxSoft managed to execute all these applications without any modification to the host systems. However, for some of these machines we performed system-specific optimizations or configurations. The key optimizations were to utilize the affinity scheduling for the Linux Kernels that supported it, *i.e.* for versions newer than 2.6.x and to set the TFlux Kernels to execute in the highest possible priority (Linux Kernel 2.2.x). As for the configuration parameters, these mainly regard the address the Scheduler of TFluxHard was mapped to as different systems provide different addresses for memory mapped devices.

Kyriakos Stavrou

10.2 TFlux Portability

Whereas virtualization refers to the ability of a TFlux incarnation to be applied to similar systems *as is*, *i.e.* without any modification, *portability* refers to the effort needed to port this incarnation to system with important differences. In this Section we will present three situations that a specific TFlux incarnation was ported to another system and comment on the effort that was required.

10.2.1 From TFluxHard to TFluxSoft

The first incarnation of the TFlux platform was the TFluxHard system; and based on it we designed TFluxSoft. Although these two systems have a completely different Scheduler implementation, they share all other TFlux layers. In particular, the TFlux directives, the TFlux Preprocessor and the TFlux Runtime are common for the two systems.

What needed to change in order to go from TFluxHard to TFluxSoft was the interface between the TFlux Kernel and the scheduler layer. For TFluxHard this interface is the implementation of the Scheduler Instructions protocol used by the TFlux Kernels to interact with the TFluxHard Scheduler (Section 5.3). As for TFluxSoft, this interface consists of function calls to the different routines provided by Scheduler. This change also affected the TFlux preprocessor as the code it produces needed to include the correct interface.

10.2.2 TFluxCell

This second case-study regards porting the TFlux Platform to the Cell/BE [4]. The target set for TFluxCell was to complete the implementation with the minimum possible effort which, similar to TFluxSoft, needed to be a software-only solution.

Kyriakos Stavrou

It was clear from the first stages of the design of TFluxCell that the TFlux Kernels were to be executed on the SPEs whereas the Scheduler's functionality by the PPE as depicted in Figure 125. The main particularity for this implementation was the lack of the shared-memory environment which is the component that allows the different execution entities of TFluxSoft to communicate between them. As such, direct execution of TFluxSoft was not possible.

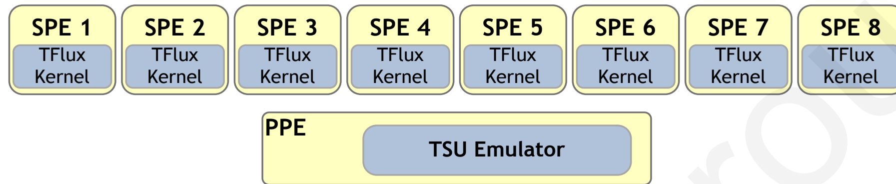


Figure 125: Performance of TFluxCell

This approach allowed us to have a successful implementation of TFlux on Cell/BE with minimum effort as no new component needed to be designed. As explained earlier this implementation uses identical layers as TFluxSoft with the only exception being the interface between the TFlux Kernels and the Scheduler which is that of TFluxHard.

Figure 126 presents some experimental performance evaluation data for the TFluxCell implementation. The reason for which some bars are missing is related to the fact that the application's data sets did not fit into the local-store memory of the SPEs. However, from these data it is possible to observe that TFlux is not only able to be ported to a new architecture with small effort but also to keep its good performance.

10.2.3 Execution on Distributed Memory Environments

The layered design of the TFlux platform allowed us to expand its design for execution to distributed memory environments by extending only the Scheduler. As such, TFlux is able to enforce the dependencies between the application DThreads's in such an environment without any

Kyriakos Stavrou

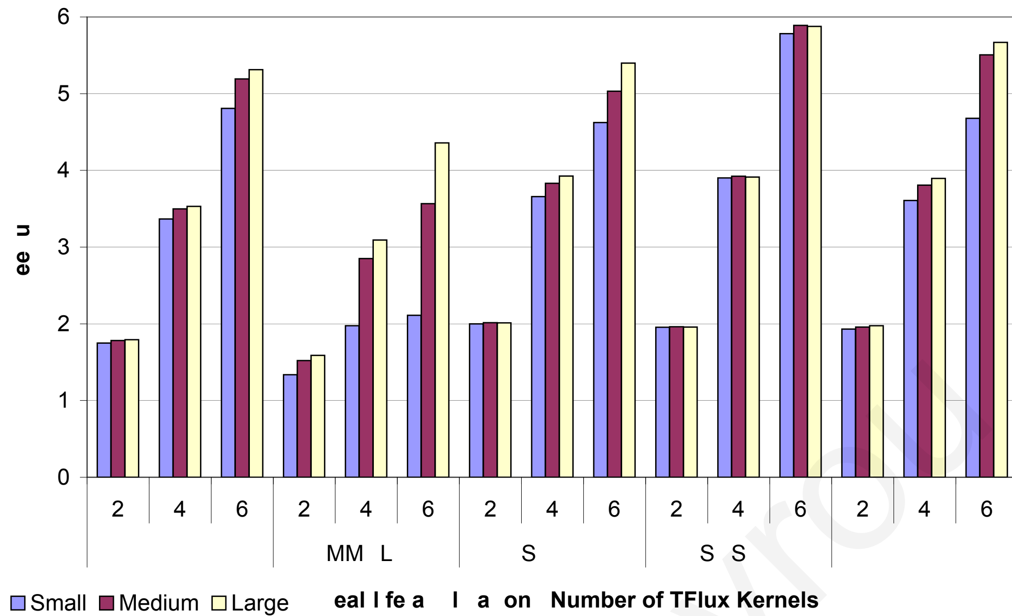


Figure 126: Performance of TFluxCell

Kyriakos Stavrou

of the other layers been unaware of it. Notice however, that it remains the responsibility of the user to perform the necessary *send* and *receive* operations for the *application's* data.

The major issue regarding TFlux execution on multiple nodes regards the *Thread Update* operation. When a DThread completes its execution its Kernel copies into the TUB the Thread Template of its consumers. It is the responsibility of the Scheduler to read these entries from the TUB and update the Ready Count values of the corresponding DThreads. If the Scheduler is to update the Ready Count of a DThread which Graph Memory entry is in one of the local TSUs of the same TFlux Node, the *Thread Update* is exactly the same to what was described in the previous Chapters. The case that requires further explanation is when the DThread entry is in a Graph Memory of a *remote* TFlux Node.

To identify this *Remote Thread Update* situation, the Scheduler is required to know the TFlux Node each DThread will be executed by. This information (Thread to Kernel Table (Section 6.2.1))

is statically inserted into the binary by the TFlux Preprocessor during the application's compilation phase and is loaded into the Scheduler when the program starts. During the *Thread Update* operation, the Scheduler reads an entry from the TUB and then consults this table. If the corresponding GM for this DThread is in a remote TFlux Node, the Scheduler will not perform the *Thread Update* operation but rather the *Remote Thread Update* operation which consists of sending the Identifier of the DThread to the remote Scheduler.

The *Remote Thread Update* operation will be completed by the target Scheduler, *i.e.* the TSU that serves the GM in which the DThread to be updated is located. To update the Ready Count counter of this DThread, the target TSU needs to be aware of the existence of the corresponding request. For this purpose, all Schedulers of the system periodically check their connection to the other TSU of the system. As such, every time these is such a *Remote DThread Update* request they will be able to process it.

Kyriakos Stavrou

Chapter 11

Conclusions and Future Work

11.1 Conclusions

This work presented the ***TFlux*** (*Thread Flux*) *Parallel Processing Platform*, a complete system from the hardware to the programming tools, that offers dataflow-like thread scheduling to off-the-shelf systems. ***TFlux*** was shown able to deliver *high-performance* by using data-driven scheduling, *virtualization* and *portability* from its layered design and *easy programmability* due to its dedicated programming toolchain.

A key component of ***TFlux*** is that it offers all its benefits to the user requiring only commodity components, *i.e.* unmodified Operating System, unmodified compiler and unmodified ISA hardware making it applicable to *off-the-shelf* systems. The abstraction layer ***TFlux*** provides to its user hides all the details of the underlying machine allowing different hardware configurations to support its model of execution transparently to the programmer.

In this work we present two incarnations of ***TFlux***: ***TFluxHard*** and ***TFluxSoft***. For ***TFluxHard*** the Thread Scheduler is a hardware unit whereas for ***TFluxSoft***, the Thread Scheduler's

Kyriakos Stavrou

functionality is provided at the software level. As such, ***TFluxHard*** is applicable to systems that offer the ability to augment the machine with a hardware module while ***TFluxSoft*** is directly applicable to *off-the-shelf* systems. A distribution of *TFluxSoft* was made available to the public upon request.

An evaluation suite which consists of applications with different characteristics in terms of both their dynamic behavior and the complexity of their dataflow graph was created to evaluate TFlux. Both TFluxHard and TFluxSoft were shown to achieve very good performance and scalability on the applications of the evaluation suite. Although for most applications the performance of the two incarnations is close, ***TFluxHard*** has an advantage over ***TFluxSoft*** arising from offloading the Scheduler's functionality to the hardware module. Nevertheless, the penalty for TFluxSoft is small compared to the benefit it offers to the off-the-shelf systems. Comparison of TFlux with a traditional parallel programming model, *i.e.* parallel execution that uses parallel threads, loops and lock and barrier synchronization, shows that TFlux is able to deliver significantly better performance especially for applications with complex Synchronization Graphs.

11.2 Future Work

The analysis presented in the previous Chapters proved that the TFlux Parallel Processing Platform is able to efficiently deliver a dataflow model of execution to commodity multicore systems. In this Chapter we present directions to further improve the performance, the programmability and the applicability of the platform. This Chapter consists of four Sections which present the future directions for the TFlux Platform, the TFluxHard system, the TFluxSoft system and the TFlux Preprocessor.

Kyriakos Stavrou

11.2.1 TFlux Platform

The specification of the TFlux components can be expanded to include mechanisms that will increase the performance and the applicability of the system. Such mechanisms could provide load balancing, fault tolerance and prefetching.

A load balancing scheme for TFlux could be implemented by merging the Graph Memory structures to a common unit. This would allow to the TFlux Kernels to execute DThreads from a common pool avoiding situations where a TFlux Kernel is idle whereas others still have not-executed ready DThreads. For this mechanism to be effective it is important to have finer-grained DThreads. On the other hand however, this will have a negative effect on performance due to larger parallelization overheads. As such, a careful study of the tradeoffs is required.

Fault tolerance could be applied to TFlux as an extension of the load balancing scheme. TFlux should be equipped with mechanisms able to detect failures at run-time and re-schedule the application's load according to the new configuration of the system. However, special care is required in order to avoid performance degradation due to the application of this mechanism. Moreover, these mechanisms should focus on the failures that apply for large-scale multicore systems.

Prefetching is expected to play a key role to increasing the performance of TFlux given that the DDM model of execution has already been shown able to significantly benefit from such a mechanism [62]. The design of these mechanisms should be applicable without requiring modifications to the system's component. This will allow the updated specification of TFlux to also be applicable to commodity machines.

11.2.2 TFluxHard

Maybe the most important future direction for TFluxHard is the development of a hardware prototype. This would allow deeper analysis of the system, fine-tuning and better performance.

Kyriakos Stavrou

Alternatives towards this direction include implementing the Scheduler on add-on card utilizing the Hyper Transport protocol as explained in Section 5.6.2. Another alternative is the implementation of the whole system on an FPGA using softcores. As for the mechanisms described earlier for the TFlux Platform it is important to be included into this prototype.

11.2.3 TFluxSoft

As explained earlier the most important drawbacks of TFluxSoft compared to TFluxHard is the fact that it requires coarser grained DThreads. To fully utilize the potential of TFluxSoft the focus of the future work should be to decrease the runtime overheads. For this it might be necessary to include additional structures to the Soft-Scheduler or preform some operations statically with the help of the TFlux Preprocessor. Moreover, operation with a distributed execution of the *Thread Update* operation should be explored.

11.2.4 TFlux Preprocessor

A common pitfall for TFlux users is the definition of too small DThreads, *i.e.* DThreads the size of which is smaller than what is required in order to amortize the parallelization overheads. Based on heuristics the TFlux Preprocessor could estimate the size of each DThread and warn the user appropriately. Notice that this minimum DThread size is different for each TFlux incarnation. With the same rational, the TFlux Preprocessor would automatically set the unroll factor for TFlux Loops to what is estimated to give the maximum performance.

Numerous research projects target automatic code parallelization but the complexity of the problem brought an intermediate solution, *i.e.* using compiler directives, in wide use in both industry and academia. As such, the ultimate goal which is to have a tool that automatically defines the boundaries and dependencies between the DThreads of an application seems to be too

Kyriakos Stavrou

far-fetched. However, the TFlux Preprocessor could be expanded to target easy-to-find DThreads and give hints to the programmer.

Kyriakos Stavrou

Appendix A

Example of using the interface of TFluxHard-Scheduler

Kyriakos Stavrou

To better understand the operation of the interface of the TFluxHard Scheduler we present the Scheduler Instructions for the execution of the example program depicted by Figure 127 on a TFluxHard system with 2 nodes. For this application DThread 8 is the Inlet DThread for the first Kernel and DThread 10 the Inlet DThread for the second TFlux Kernel. As for DThreads 9 and 11 they are the Outlet DThreads of the first and second TFlux Kernel respectively. The DThreads shown at the left of this Figure (8, 1, 3, 5 and 9) are executed by TFlux Kernel 1 whereas the DThreads on the right (10, 2, 4, 6, 7 and 11) by TFlux Kernel 2.

The Table that follows shows all the data packets exchanged between the TFlux Kernels and the Scheduler for the execution of this application as well as each packet's meaning. Notice that for this application the Thread Id (THID) field is represented by 7 bits and the Iteration Id (ITER) by 25 bits (recall that the Thread Template that is composed of the THID and the ITER must be a

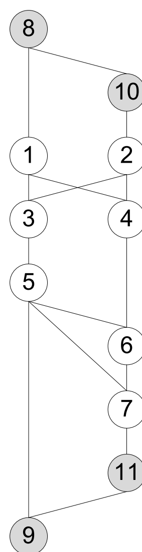


Figure 127: Example program for the better explanation of the TFluxHard Scheduler interface.

32-bits value). As an example, for this particular partitioning, the thread template 10000000_{HEX} corresponds to the DThread 8/16. To better explain this consider that:

$$10000010_{HEX} = 00010000000000000000000000010000_{BIN} \quad .$$

As the first 7 bits are for the THID, the THID value is equal to $0001000_{BIN} = 8_{DEC}$ and ITER $0000000000000000000010000_{BIN} = 16_{DEC}$.

Operation	Value	TSU	Description
Write	0-0-0-1	0	1 Thread Load will follow for TSU 0
Write	0-2-0-1	0	This thread has 2 consumers, 0 ILC and 1 instance
Write	0-0-0-1	0	Ready Count = 1
Write	10-0-0-0	0	Thread Template: 8/0
Write	2-0-0-0	0	Thread template of consumer 1 (1/0)
Write	14-0-0-0	0	Thread template of consumer 2 (10/0)
Write	0-0-0-1	1	1 Thread Load will follow for TSU 1

to be continued on next page

Kyriakos Stavrou

Operation	Value	TSU	Description
Write	0-1-0-1	1	This thread has 1 consumer, 0 ILC and 1 instance
Write	0-0-0-1	1	Ready Count = 1
Write	14-0-0-0	1	Thread Template: 10/0
Write	4-0-0-0	1	Thread Template of consumer 1: 2
Write	7-1-0-1	1	Explicitly decrease the ready count of the following 1 DThreads
Write	10-0-0-0	1	Thread Template: 8/0
Read	8/0	0	Execution of thread 8/0 started
Write	0-0-0-4	0	4 Thread loads will follow
Write	0-2-0-1	0	This thread has 2 consumers, 0 ILC and 1 instance
Write	0-0-0-1	0	Ready Count=1
Write	2-0-0-0	0	Thread Template: 1/0
Write	6-0-0-0	0	Consumer 1: 3
Write	8-0-0-0	0	Consumer 2: 4
Write	0-1-0-1	0	This thread has 1 consumer, 0 ILC and 1 instance
Write	0-0-0-2	0	Ready Count=2
Write	6-0-0-0	0	Thread Template: 3/0
Write	a-0-0-0	0	Consumer 1: 5
Write	0-3-0-1	0	This thread has 3 consumers, 0 ILC and 1 instance
Write	0-0-0-1	0	Ready Count=1
Write	a-0-0-0	0	Thread Template: 5/0
Write	c-0-0-0	0	Consumer 1: 6/0
Write	e-0-0-0	0	Consumer 2: 7/0
Write	12-0-0-0	0	Consumer 3: 9/0
Write	0-0-0-1	0	This thread has 0 consumers, 0 ILC and 1 instance

to be continued on next page

Kyriakos Stavrou

Operation	Value	TSU	Description
Write	0-0-0-2	0	Ready Count=2
Write	12-0-0-0	0	Thread Template: 9/0
Write	2-0-0-0	0	The currently executed thread completed its execution (8/0)
Read	1/0	0	Execution of thread 1/0 started
Write	2-0-0-0	0	The currently executed thread completed its execution (8/0)
Read	10/0	1	Execution of thread 10/0 started
Write	0-0-0-5	1	5 Thread loads will follow
Write	0-2-0-1	1	This thread has 2 consumers, 0 ILC and 1 instance
Write	0-0-0-1	1	Ready Count=1
Write	4-0-0-0	1	Thread Template: 2/0
Write	6-0-0-0	1	Consumer 1: 3/0
Write	8-0-0-0	1	Consumer 2: 4/0
Write	0-1-0-1	1	This thread has 1 consumer, 0 ILC and 1 instance
Write	0-0-0-2	1	Ready Count=2
Write	8-0-0-0	1	Thread Template: 4/0
Write	c-0-0-0	1	Consumer 1: 6/0
Write	0-1-0-1	1	This thread has 1 consumer, 0 ILC and 1 instance
Write	0-0-0-2	1	Ready Count=2
Write	c-0-0-0	1	Thread Template: 6/0
Write	e-0-0-0	1	Consumer 1: 7/0
Write	0-1-0-1	1	This thread has 1 consumer, 0 ILC and 1 instance
Write	0-0-0-2	1	Ready Count=2
Write	e-0-0-0	1	Thread Template: 7/0
Write	16-0-0-0	1	Consumer 1: 11/0

to be continued on next page

Kyriakos Stavrou

Operation	Value	TSU	Description
Write	0-1-0-1	1	This thread has 1 consumer, 0 ILC and 1 instance
Write	0-0-0-1	1	Ready Count=1
Write	16-0-0-0	1	Thread Template: 11/0
Write	12-0-0-0	1	Consumer 1: 9/0
Write	2-0-0-0	1	The currently executed thread completed its execution (10/0)
Read	2/0	1	Execution of thread 2/0 started
Write	2-0-0-0	1	The currently executed thread completed its execution (2/0)
Read	4/0	1	Execution of thread 4/0 started for CPU 1
Read	3/0	0	Execution of thread 4/0 started for CPU 0
Write	2-0-0-0	1	The currently executed thread completed its execution (4/0)
Write	2-0-0-0	0	The currently executed thread completed its execution (3/0)
Read	5/0	0	Execution of thread 5/0 started for CPU 0
Write	2-0-0-0	0	The currently executed thread completed its execution (5/0)
Read	6/0	1	Execution of thread 6/0 started for CPU 0
Write	2-0-0-0	1	The currently executed thread completed its execution (6/0)
Read	7/0	1	Execution of thread 7/0 started for CPU 0
Write	2-0-0-0	1	The currently executed thread completed its execution (6/0)
Read	11/0	1	Execution of thread 11/0 started for CPU 0
Write	1-0-0-0	1	Clear TSU 1
Write	2-0-0-0	1	The currently executed thread completed its execution (11/0)
Write	6-0-0-0	1	TFlux Kernel 1 completed its execution
Read	9/0	0	Execution of thread 9/0 started for CPU 0
Write	1-0-0-0	0	Clear TSU 0
Write	6-0-0-0	0	TFlux Kernel 0 completed its execution

to be continued on next page

Kyriakos Stavrou

Operation	Value	TSU	Description
Write	2-0-0-0	0	The currently executed thread completed its execution (9/0)

Kyriakos Stavrou

Appendix B

TFlux Directives

This appendix describes the TFlux Directives supported by the current version of the TFlux Preprocessor. Before the presentation of the directives, Section B.1 will introduce the necessary notations.

B.1 Notations

Depends-on-list

The ***dependList*** list defines the producers of DThread, *i.e.* the list of DThreads a DThread depends-on. This list is composed of multiple numbers each defining the Thread Id of one of the producers.

$dependList = (THID_1, ..., THID_n)$

Name Thread List

The ***nameThreadList*** list describes a list of variables and is used by the *import* and *export* statements (Section 4.3.1.1). For each variable, in addition to the variable's name it includes the Thread Id of the DThread that produces this value.

$nameThreadList = (VarName_1 : THID_1, ..., VarName_n : THID_n)$

Kyriakos Stavrou

Name Thread List

The ***typeNameList*** list describes a list of variables and is used by the *import* and *export* statements. For each variable, in addition to the variable's name it includes the type of the variable.

$typeNameList = (VarType_1\ \ VarName_1, ...,\ \ VarType_n\ \ VarName_n$

Iteration Level Consumers List

The ***ilcList*** list describes the iteration-level consumers of an L-DThread. The *ilcList* has the following format:

$ilcList = [Type, THID, a, b, c, sched], ..., [Type, THID, a, b, c, sched]$

Each tuple (Type, THID, a, b, c, d) defines the information of one of the iteration-level Consumers. The different fields are used to define the *Iteration Id* of the iteration-level Consumer. As for the Thread Id of this Consumer this is equal to the ***THID*** field of the tuple. The ***SCHED*** describes the scheduling algorithm followed for the Consumer and the Producer TFlux Loops. The possible values for the ***SCHED*** field are 0, 1, 2 and 3. The meaning of each different value is as follows:

SCHED	***Producer TFlux Loop***	***Consumer TFlux Loop***
0	Chunk Scheduling	Chunk Scheduling
1	Chunk Scheduling	Round-Robin Scheduling
2	Round-Robin Scheduling	Chunk Scheduling
3	Round-Robin Scheduling	Round-Robin Scheduling

The calculation of the Iteration Id of the Consumer L-DThread (ITER) is based on the ***Type*** field. In particular, the value of this field defines the algorithm that is used to identify the Iteration Id of the Consumer. As for the ***a***, ***b*** and ***c*** fields they are used as parameters to this calculation. Moreover, notice that the Iteration Id of the Consumer L-DThread (*CITER*) is calculated as a function of the Iteration Id of the Producer L-DThread (*PITER*). The algorithm that is applied according to the different values of the ***Type*** field is as follows:

Kyriakos Stavrou

Type	*Algorithm*
1	`CITER = PITER*a+b;`
2	`CITER = PITER/a+b;`
3	`CITER = PITER*a-b;`
4	`CITER = (PITER*a-b>0 ?  PITER*a-b :  0)`
5	`CITER = (PITER>=b ?  PITER : -1)`
6	`CITER = (PITER==a ?  b :  -1)`
7	`CITER = ((PITER+a)<=b ?  PITER+a :  -1)`
8	`CITER = ((PITER%a)==0 ?  PITER+b :  -1)`
9	`CITER = ((PITER%a)!=a-1 ?  PITER+1 :  -1)`
10	`CITER = (PITER>=b?PITER : -1)`
11	`CITER = (PITER<a ?  PITER-b :  -1)`
12	`CITER = (PITER>=a ?  PITER+b :  -1)`

B.2 Program Control

Program Start

Defines the start of the program's code and must be placed in the *main()* function. All variables declared after the *startprogram* directive are private to the application's TFlux Kernels.

```
#pragma ddm startprogram
```

Program End

Defines the end of the program's code and must be placed in the *main()* function. This directives defines the last instruction of *main()*.

```
#pragma ddm endprogram
```

Number of TFlux Kernels

Defines the number of TFlux Kernels executing the application to be equal to *noKernels*.

```
#pragma ddm kernel noKernels
```

Kyriakos Stavrou

B.3 Variables Declaration

Declaration of private variable

Declares and allocates memory for variable *varName* which is of type *varType*. This variable is private to the TFlux Kernels.

```
#pragma ddm private ddm varType varName
```

Declaration of private array

Declares and allocates memory for array *varName* which is of type *varType* and has *noElements* elements. This array is private to the TFlux Kernels.

```
#pragma ddm private ddm varType varName noElements
```

Declaration of shared variable

Declares and allocates memory for variable *varName* which is of type *varType*. This variable is shared among the TFlux Kernels.

```
#pragma ddm global ddm varType varName
```

Declaration of private array

Declares and allocates memory for array *varName* which is of type *varType* and has *noElements* elements. This variable is shared among the TFlux Kernels.

```
#pragma ddm global ddm varType varName noElements
```

B.4 Block Declaration

Declaration of DDM Block

Declaration of a DDM Block with Block Id equal to ***BID***. The block can optionally be set to import or export variables. Importing will make the particular variables accessible by the

Kyriakos Stavrou

DThreads of this Block. Similarly, exporting a variable, makes it accessible to the blocks that import it.

```
#pragma ddm block BID                                    \
        <import typeNameList> <export nameThreadList>

    //DECLARATION OF DTHREADs

#pragma ddm enblock
```

`import typeNameList:` If used the Block *imports* the list of variables defined in the *typeNameList* that follows.

`export nameThreadList:` If used the Block *exports* the list of variables defined in the *nameThreadList* that follows.

B.5 DThread Declaration

DThread with *implicit* dependencies

Declaration of DThread with Thread Id equal to ***TID*** that is executed by TFlux Kernel ***KID***. Optionally, this DThread can be set to *import* or *export* variables through using the *import* and *export* statements respectively.

```
#pragma ddm thread TID kernel KID                            \
        <import nameThreadList> <export typeNameList>

    //DThread ANSI C Code

#pragma ddm endthread
```

`import nameThreadList:` If used the DThread *imports* the list of variables defined in the *nameThreadList* that follows.

`export typeNameList:` If used the DThread *exports* the list of variables defined in the *typeNameList* that follows.

DThread with *implicit* dependencies that recycles

Declaration of DThread with Thread Id equal to ***TID*** that is executed by TFlux Kernel ***KID*** and belongs to a recycle group, *i.e.* it can re-invoke it self to execute another instance of the same static code. Optionally, this DThread can be set to import or export variables through using the *import* and *export* statements respectively.

Kyriakos Stavrou

```
#pragma ddm thread TID kernel KID                            \
        <import nameThreadList> <export typeNameList>       \
        recycle

    //DThread ANSI C Code

#pragma ddm recycle
```

`import nameThreadList:` If used the DThread *imports* the list of variables defined in the *nameThreadList* that follows.

`export typeNameList:` If used the DThread *exports* the list of variables defined in the *typeNameList* that follows.

DThread with *implicit* dependencies executed by all Kernels

Declaration of DThread with Thread Id equal to ***TID*** that is executed by *all* TFlux Kernels. Optionally, this DThread can be set to import or export variables through using the *import* and *export* statements respectively.

```
#pragma ddm thread TID kernel all                            \
        <import nameThreadList> <export typeNameList>

    //DThread ANSI C Code

#pragma ddm endthread
```

`import nameThreadList:` If used the DThread *imports* the list of variables defined in the *nameThreadList* that follows.

`export typeNameList:` If used the DThread *exports* the list of variables defined in the *typeNameList* that follows.

DThread with *implicit* dependencies that recycles and is executed by all Kernels

Declaration of DThread with Thread Id equal to ***TID*** that is executed by *all* TFlux Kernels and belongs to a recycle group, *i.e.* it can re-invoke it self to execute another instance of the same static code. Optionally, this DThread can be set to import or export variables through using the *import* and *export* statements respectively.

```
#pragma ddm thread TID kernel all                            \
        <import nameThreadList> <export typeNameList>       \
        recycle
```

Kyriakos Stavrou

```
    //DThread ANSI C Code

#pragma ddm recycle
```

`import nameThreadList:` If used the DThread *imports* the list of variables defined in the *nameThreadList* that follows.

`export typeNameList:` If used the DThread *exports* the list of variables defined in the *typeNameList* that follows.

DThread with *explicit* dependencies

Declaration of DThread with Thread Id equal to ***TID*** that is executed by TFlux Kernel ***KID***. Optionally, this DThread can be set to depend on other DThreads or TFlux Loops.

```
#pragma ddm thread TID kernel KID <depends (dependList)>

    //DThread ANSI C Code

#pragma ddm endthread
```

`depends dependList:` The list of DThreads and TFlux Loops this DThread depends on.

DThread with *explicit* dependencies that recycles

Declaration of DThread with Thread Id equal to ***TID*** that is executed by TFlux Kernel ***KID*** and belongs to a recycle group, *i.e.* it can re-invoke it self to execute another instance of the same static code. Optionally, this DThread can be set to depend on other DThreads or TFlux Loops.

```
#pragma ddm thread TID kernel KID <depends (dependList)> recycle

    //DThread ANSI C Code

#pragma ddm recycle
```

`depends dependList:` The list of DThreads and TFlux Loops this DThread depends on.

DThread with *explicit* dependencies executed by all Kernels

Declaration of DThread with Thread Id equal to ***TID*** that is executed by *all* TFlux Kernels. Optionally, this DThread can be set to depend on other DThreads or TFlux Loops.

Kyriakos Stavrou

```
#pragma ddm thread TID kernel all <depends (dependList)>

    //DThread ANSI C Code

#pragma ddm endthread
```

depends dependList: The list of DThreads and TFlux Loops this DThread depends on.

DThread with implicit dependencies that controls the execution of a recycle group

Declaration of the DThread that controls the execution of a recycle group, *i.e.* that defines if the DThreads of the recycle group will be executed again. The DThread has Thread Id equal to ***TID*** and is executed by TFlux Kernel ***KID***. The dependencies of this DThread are defined implicitly through the *import* and *export* statements.

```
#pragma ddm thread TID kernel KID                          \
        <import nameThreadList> <export typeNameList>     \
        recycle

    //DThread ANSI C Code

    if(recycle group termination condition)
    {
      #pragma ddm threadCompleted
    }

#pragma ddm recycle
```

import nameThreadList: If used the DThread *imports* the list of variables defined in the *nameThreadList* that follows.

export typeNameList: If used the DThread *exports* the list of variables defined in the *typeNameList* that follows.

DThread with explicit dependencies that controls the execution of a recycle group

Declaration of the DThread that controls the execution of a recycle group, *i.e.* that defines if the DThreads of the recycle group will be executed again. The DThread has Thread Id equal to ***TID*** and is executed by TFlux Kernel ***KID***. This DThread depends on the DThreads and TFlux Loops included in the *dependsList*

```
#pragma ddm thread TID kernel KID depends dependList recycle
```

Kyriakos Stavrou

```
    //DThread ANSI C Code

    if(recycle group termination condition)
    {
      #pragma ddm threadCompleted
    }

#pragma ddm recycle
```

depends dependList: The list of DThreads and TFlux Loops this DThread depends on.

B.6 TFlux Loop

Defines a TFlux Loop with Thread Id equal to *TID*.

```
#pragma ddm for thread TID                                          \
        <schedule 1>                                                \
        <unroll N>                                                  \
        <reduction localVar op globalVarType globalVarName>         \
        <depends (dependList)>                                      \
        <ilc ilcList>                                               \
        <readyCount rcValue>                                        \
        <recycle>

   //TFlux Loop Code

#pragma ddm endfor
```

schedule 1: If it is used the loop iterations will be assigned to the TFlux Kernels according to the Round Robin scheduling scheme. If it is not used, the Chunk Scheduling scheme is applied.

unroll N: If this option is used, the loop is unrolled ***N*** times

reduction localVar op globalVarType globalVarName: The loop performs a reduction operation. The operation to be performed is defined by the ***op*** and can be subtraction (-), summation (+) or product (*). The ***localVar*** variable holds the partial result for each Kernel. The total result will be stored to the ***globalVarName*** variable which type is ***globalVarType***.

depends (dependList): If this option is used the TFlux Loop is set to depend on the DThreads and TFlux Loops of the ***dependList***.

ilc ilcList: If this option is used the TFlux Loop is set to have the DThreads described in the *ilcList* as Iteration Level Consumers.

readyCount rcValue: Defines that the Ready Count value for the L-DThreads of this loops will be initialized to ***rcValue***.

recycle: Sets the TFlux Loop to recycle (the TFlux Loop belongs in a recycle group)

Kyriakos Stavrou

Index

TFlux Preprocessor, 58

Bagle, 186
Basic Execution Components, 39
Basic Operations, 53
Basic TSU Operations, 53
Block, 30, 52
Blocks, 35, 62
bursts, 172

CAR, 109
Cashew, 184, 186
Chip Multiprocessor, 11
CL, 37
Clear TSU Operation, 56
Clear TSU Unit, 82
CLMU, 82
Consumer DThreads, 40
Consumer List, 37, 82
Consumer List Management Unit, 82
Consumers Arrays, 109
CPU-Memory gap, 10
CTU, 82

Data-Driven Multithreading, 27
Data-Driven Multithreading Chip Multiprocessor, 31
Data-Driven Network of Workstations, 30
Data-Driven Threads, 27
DDM Block, 30, 52
DDM C Preprocessor, 38
DDM thread, 39
ddmCpp, 38
dependList, 246
depends statement, 65
DThread, 39
DThread List, 61
DThreads, 27
Dynamic dataflow architectures, 20
dynamic metadata, 82

effect, 160
Enterprise, 186
Execution Completion Operation, 54
export statement, 65
expressibility, 73

Find Ready Thread loop, 35
Find Ready Thread Operation, 56
first generation L-DThreads, 47
first generation of L-DThreads, 49
First Kernel, 62
Fully Parallel Loops, 41

GM, 37
Graph Memory, 37

host GM, 118

IBM3650, 185
ILC ARRAY, 110
ilc statement, 71
ILCL, 38, 82
ilcList, 247
ILCLMU, 82
ILD, 44
import statement, 65
Inlet DThread, 30, 35, 52, 62
Input Queue, 81
InQ, 81
Inter-block Sequential Code, 52
Iteration Id, 40
Iteration Level Consumers List, 82
Iteration Level Consumers List Management Unit, 82
Iteration Level Dependencies, 41, 44
Iteration-level Consumers List, 38
Iteration-level dependencies, 41
iterations-level dependency, 44

kernel all statement, 66
kernel statement, 66

Kyriakos Stavrou

L-DThread, 42
L-DThread Recycle Operation, 55
L-DThread Recycling, 47
Large-grain dataflow, 22
last generation of L-DThreads, 49
last-generation L-DThreads, 48
LIT, 119
load function, 159
Load Thread Unit, 81
Local TUB, 121
Loop DThread, 42
Loop Information Table, 119
loop-level dependencies, 41
LTU, 81

macro-actor Threaded dataflow, 22
Memory Wall, 10
metadata, 29
metadata of DThread, 39
Minimum Thread Size, 161

nameThreadList, 246

Outlet DThread, 30, 52, 62
OutletDThread, 35
Output Queue, 81
OutQ, 81

Parallel Threads, 161
Post-Processing Phase, 30, 35
Power Wall, 11
pragma ddm endfor, 66
pragma ddm endthread, 64
pragma ddm for, 66
pragma ddm thread, 64
Producers, 40

Ready Count, 28, 40
Ready Queue, 83
recycle statement, 72
Recycle-Group, 72
Reduction, 69
Reduction Loops, 49
Reduction Operation, 49
reduction statement, 69
Reduction TFlux Loops, 69
RQ, 83
Runtime Support, 34

schedule 1 statement, 67
Scheduler Instructions, 81
Simics, 180
Simultaneous Multithreading, 10
single static assignment language, 19
single-token-per-arc dataflow machines, 18
SM, 37
SNIU, 81
SoftScheduler, 107
speedup, 188
static dataflow architectures, 18
static metadata, 82
Synchronization Graph, 27, 39
Synchronization Memory, 37
System Network Interface Unit, 81

Tango, 186
TES, 37
TFlux, 32
TFlux C Preprocessor, 38
TFlux directives, 33, 38, 58, 64
TFlux Kernel, 34, 35
TFlux Layers, 33
TFlux Loop, 41
TFlux Loop - DThread dependency, 43
TFlux Loop - TFlux Loop dependency, 43
TFlux Loop unrolling, 68
TFlux Preprocessor, 33
TFlux Runtime Support, 33
TFlux Scheduler, 33
TFluxCpp, 38, 58
TFluxHard, 78
TFluxSim, 184
THID, 39
Thread Execution Stack, 37
Thread Footer, 60
Thread Id, 39
Thread Label, 60
Thread Level Speculation, 12
Thread Load Operation, 53
Thread Recycle Operation, 55
Thread Recycling, 50
Thread Select Loop, 59
Thread Template, 29, 39
Thread to Kernel Table, 118
Thread Update Operation, 56
Thread Update phase, 38

Kyriakos Stavrou

Thread Update Unit, 82
Thread-Flux, 32
Threads-to-Update Buffer, 38, 83
TKT, 118
Traditional Parallel Execution Model, 167
traditional programming models, 73
TUB, 38, 83
TUB Buffers, 122
TUB Ranges, 123
TUU, 82
typeNameList, 247

unroll, 188
unroll statement, 68
Unrolling, 68
update-request, 38
update-requests, 172
Updater, 106
usage, 189

Kyriakos Stavrou

Bibliography

[1] Vikas Agarwal, M. S. Hrishikesh, Stephen W. Keckler, and Doug Burger. Clock rate versus IPC: the end of the road for conventional microarchitectures. In *Proceedings of the 27th annual International Symposium on Computer Architecture (ISCA 27)*, pages 248–259, New York, NY, USA, 2000. ACM Press.

[2] AMD. Product Brief: Quad-Core AMD Opteron Processor. www.amd.com/, 2008.

[3] Boon Ang, Arvind, and Derek Chiou. StarT the Next Generation: Integrating Global Caches and Dataflow Architecture. In *Proceedings of the International Conference on Computer Systems and Education, IISc*, 1994.

[4] Samer Arandi, Kyriakos Stavrou, Pedro Trancoso, and Paraskevas Evripidou. DDM-Cell: Data-Driven Multithreading on the Cell Processor. In *Proceedings of the 3rd International Summer School on Advanced Computer Architecture and Compilation for Embedded Systems (ACACES 2007)*, pages 275–278, July 2007.

[5] Joseph M. Arul and KrishnaM. Kavi. Scalability of scheduled dataflow architecture (sdf) with register contexts. In *Proceedings of the Fifth International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP 02)*, pages 214–221, 2002.

[6] Arvind, K. Gostelow, and M. Plouffe. An asynchronous programming language and computing machine. Technical Report Techical Report TR 114a, University of California, Irvine, Irvine, CA, 1978.

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

[8] K. Arvind and Rishiyur S. Nikhil. Executing a program on the mit tagged-token dataflow architecture. *IEEE Transactions on Computing*, 39(3):300–318, 1990.

[9] Arvind and R. E. Thomas. I-structures: An efficient data type for functinal languages. Technical Report LCS/TM-178, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1980.

[10] Siamak Arya, Howard Sachs, and Sreemam Duvvuru. An architecture for high instruction level parallelism. In *HICSS '95: Proceedings of the 28th Hawaii International Conference on System Sciences (HICSS'95)*, page 153, Washington, DC, USA, 1995. IEEE Computer Society.

Kyriakos Stavrou

[11] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[12] Edward Ashcroft and William Wadge. Lucid, a nonprocedural language with iteration. *Commun. ACM*, 20(7):519–526, 1977.

[13] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, 1991.

[14] Saisanthosh Balakrishnan and Gurindar S. Sohi. Program Demultiplexing: Data-flow Execution of Methods in Sequential Programs. In *Proceedings of the 33rd annual International Symposium on Computer Architecture (ISCA 33)*, pages 302–313, 2006.

[15] Pieter Bellens, Josep M. Perez, Rosa M. Badia, and Jesus Labarta. CellSs: a programming model for the cell BE architecture. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 86, New York, NY, USA, 2006. ACM.

[16] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 72–81, October 2008.

[17] Guy E. Blelloch. Prefix sums and their applications. Technical report, Synthesis of Parallel Algorithms, 1990.

[18] Blume, W. and Eigenmann, R. and Faigin, K. and Grout, J. and Hoeflinger, J. and Padua, D. and Petersen, P. and Pottenger, W. and Rauchwerger, L. and Tu, P. and Weatherford, S. Polaris: The Next Generation in Parallelizing Compilers. Technical Report 1375.

[19] Koen De Bosschere, Wayne Luk, Xavier Martorell, Nacho Navarro, Mike OBoyle, Dionisios Pnevmatikatos, Alex Ramirez, Pascal Sainrat, Andre Seznec, Per Stenstrom, , and Olivier Temam. High-Performance Embedded Architecture and Compilation Roadmap. *Transactions on High-Performance Embedded Architectures and Compilers*, 1:5–29, 2007.

[20] Doug Burger, Stephen W. Keckler, Kathryn S. McKinley, Mike Dahlin, Lizy K. John, Calvin Lin, Charles R. Moore, James Burrill, Robert G. McDonald, William Yoder, and the TRIPS Team. Scaling to the End of Silicon with EDGE Architectures. *IEEE Computer*, 37(7):44–55, 2004.

[21] David R. Butenhof. *Programming with POSIX threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[22] Siddhartha Chatterjee, Guy E. Blelloch, and Marco Zagha. Scan primitives for vector computers. In *Supercomputing '90: Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*, pages 666–675, Washington, DC, USA, 1990. IEEE Computer Society.

Kyriakos Stavrou

[23] Murray Cole. *Algorithmic skeletons: structured management of parallel computation*. MIT Press, Cambridge, MA, USA, 1991.

[24] Dominique Comte, Guy Durrieu, O. Gelly, A. Plas, and Jean claude Syre. Parallelism, control and synchronization expression in a single assignment language. *SIGPLAN Not.*, 13(1):25–33, 1978.

[25] M. Cornish, D. W. Hogan, and J. C. Jensen. The Texas Instruments distributed data processor. In *Proceedings of the Louisiana Computer Exposition*, pages 189–193, 1979.

[26] Wayne R. Cowell and Christopher P. Thompson. Transforming fortran do loops to improve performance on vector architectures. *ACM Trans. Math. Softw.*, 12(4):324–353, 1986.

[27] Cray Inc. Cray XD1 supercomputer for reconfigurable computing. http://www.cray.com/downloads/FPGADatasheet.pdf, 2005.

[28] David E. Culler, Seth Copen Goldstein, Klaus Erik Schauser, and Thorsten von Eicken. Tam - a compiler controlled threaded abstract machine. *Journal of Parallel and Distributed Computing*, 18(3):347–370, 1993.

[29] David E. Culler, Klaus E. Schauser, and Thorsten von Eicken. Two Fundamental Limits on Dataflow Multiprocessing. In *Proceedings of the IFIP WG10.3. Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism (PACT 93)*, pages 153–164, 1993.

[30] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, pages 107–113, 2004.

[31] Monty Denneau and Henry S. Warren. 64-bit cyclops principles of operation part i. Technical report, IBM Thomas J. Watson Research Center, April 2005.

[32] Jack B. Dennis. First Version of a Dataflow Procedure Language. In *Programming Symposium, Proceedings Colloque sur la Programmation*, pages 362–376, 1974.

[33] Paraskevas Evripidou. Thread Synchronization Unit (TSU): A building block for High Performance Computers. In *Proceedings of the International Symposium on High Perfomance Computing, Fukuoka, Japan*, pages 107–118, November 1997.

[34] Paraskevas Evripidou and J. Gaudiot. A decoupled graph/computation data-driven architecture with variable resolution actors. In *Proceedings of ICPP 1990*, pages 405–414, 1990.

[35] Paraskevas Evripidou and Costas Kyriacou. Data driven network of workstations (D2NOW). *Journal of Universal Computer Science*, 6(10):1015–1033, October 2000.

[36] Holger Froning, Mondrian Nussle, David Slogsnat, Heiner Litz, and Ulrich Brüning. The HTX-Board: A Rapid Prototyping Station. In *3rd Annual FPGAworld Conference*, November 2006.

[37] Daniel Gajski, David A. Padua, David J. Kuck, and Robert H. Kuhn. A second opinion on data flow machines and languages. *IEEE Computer*, 15(2):58–69, 1982.

[38] GPGPU. General-Purpose computation on GPUs (GPGPU). www.gpgpu.org/, 2008.

Kyriakos Stavrou

[39] Gregory M. Papadopoulos and David E. Culler. Monsoon: An explicid token store architecture. In *In Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 82 – 91, May 1990.

[40] Gregory. M. Papadopoulos and Kenneth. R. Traub. Multithreading: A revisionist view of dataflow architectures. In *In Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 342 – 351, May 1991.

[41] The Parafrase Group. Parafrase 2. http://www.csrd.uiuc.edu/parafrase2/, 2008.

[42] J. R. Gurd and W. Bohm. Implicit parallel processing: SISAL on the Manchester dataflow computer. In *Proceedings of the IBM-Europe Institute on Parallel Processing*, 1987.

[43] John R. Gurd, Chris C. Kirkham, and Ian Watson. The manchester prototype dataflow computer. *Communications of the ACM*, 28(1):34–52, 1985.

[44] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the 4th IEEE International Workshop on Workload Characterization (WWC-4)*, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society.

[45] Chris L Hankin and Hugh Glaser. The data flow programming language CAJOLE - an informal introduction. *SIGPLAN Not.*, 16(7):35–44, 1981.

[46] Michael Frumkin Haoqiang Jin and Jerry Yan. The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance. Technical Report NAS Technical Report NAS-99-011, NASA Ames Research Center - NAS System Division, October 1999.

[47] Olivier Maquelin *et al.* Herbert H. J. Hum. A design study of the earth multiprocessor. In *PACT '95: Proceedings of the IFIP WG10.3 working conference on Parallel architectures and compilation techniques*, pages 59–68, Manchester, UK, UK, 1995. IFIP Working Group on Algol.

[48] Mark D. Hill and Alan Jay Smith. Experimental evaluation of on-chip microprocessor cache memories. In *ISCA '84: Proceedings of the 11th annual international symposium on Computer architecture*, pages 158–166, New York, NY, USA, 1984. ACM.

[49] Hypertransport Consortium. HyperTransport Technology I/O Link. www.hypertransport.org.

[50] Intel. Intel's Teraflops Research Chip, 2006.

[51] Intel. Intel Quad-Core technology. . http://www.intel.com/technology/quad-core/, 2008.

[52] International Telecommunication Union (ITU). H.264 : Advanced video coding for generic audiovisual services - Recommendation. http://www.itu.int/rec/T-REC-H.264/e, 2008.

[53] Jack B. Dennis. Dataflow Supercomputers. *Computers*, pages 48 – 56, November 1980.

[54] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in dataflow programming languages. *ACM Computer Surveys*, 36(1):1–34, 2004.

Kyriakos Stavrou

[55] Kaeli David and Pen-Chung Yew. *Speculative Execution In High Performance Computer Architectures*. Chapman and Hall/CRC, 2005.

[56] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM J. Res. Dev.*, 49(4/5):589–604, 2005.

[57] Krishna Kavi and Behrooz Shirazi. Dataflow architecture: Are dataflow computers commercially viable. *IEEE Potentials*, pages 27–30, 1992.

[58] Krishna M. Kavi, Roberto Giorgi, and Joseph Arul. Scheduled Dataflow: Execution Paradigm, Architecure, and Performance Evaluation. *IEEE Transactions on Computers*, 50(8):834–846, August 2001.

[59] Youngsoo Kim and Suleyman Sair. Designing real-time H.264 decoders with dataflow architectures. In *Proceedings of the 3rd IEEE/ACM/IFIP International Conference on Hardware/software codesign and system synthesis (CODES+ISSS 05)*, pages 291–296, New York, NY, USA, 2005. ACM Press.

[60] Sanjeev Kumar, Christopher J. Hughes, and Anthony Nguyen. Carbon: architectural support for fine-grained parallelism on chip multiprocessors. pages 162–173, 2007.

[61] Costas Kyriacou. *Data Driven Multithreading using Conventional Control Flow Microprocessors*. PhD dissertation, University of Cyprus, 2005.

[62] Costas Kyriacou, Paraskevas Evripidou, and Pedro Trancoso. CacheFlow: A Short-Term Optimal Cache Management Policy for Data Driven Multithreading. In *Proceedings of the 2004 EuroPar (EuroPar 04)*, pages 561–570, August 2004.

[63] Costas Kyriacou, Paraskevas Evripidou, and Pedro Trancoso. Data-Driven Multithreading Using Conventional Microprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 17(10):1176–1188, 2006.

[64] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *International Symposium on Microarchitecture*, pages 330–335, 1997.

[65] Johnny K. F. Lee and Alan J Smith. Analysis of branch prediction strategies and branch target buffer. Technical report, Berkeley, CA, USA, 1983.

[66] Johnny K. F. Lee and Alan Jay Smith. Branch prediction strategies and branch target buffer design. *Computer*, 17(1):6–22, 1984.

[67] Yingmin Li, David Brooks, Zhigang Hu, and Kevin Skadron. Performance, Energy, and Thermal Considerations for SMT and CMP Architectures. In *Proceedings of the 11th IEEE International Symposium on High Performance Computer Architecture (HPCA 11)*, pages 71–82, February 2005.

[68] David Lilja. The impact of parallel loop scheduling strategies on prefetching in a shared memory multiprocessor. *IEEE Trans. Parallel Distrib. Syst.*, 5(6):573–584, 1994.

[69] Zhijian Lu, Wei Huang, Shougata Ghosh, John Lach, Mircea Stan, and Kevin Skadron. Analysis of Temporal and Spatial Temperature Gradients for IC Reliability. Technical Report CS-2004-08, University of Virginia, March 2004.

Kyriakos Stavrou

[70] Zhijian Lu, John Lach, Mircea Stan, and Kevin Skadron. Banking Chip Lifetime: Opportunities and Implementation. In *Proceedings of the 1st Workshop on High Performance Computing Reliability Issues (HPCRI 1)*, 2005.

[71] Bing Luo and Chris Jesshope. Performance of a micro-threaded pipeline. In *CRPIT '02: Proceedings of the seventh Asia-Pacific conference on Computer systems architecture*, pages 83–90. Australian Computer Society, Inc., 2002.

[72] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, 2002.

[73] James R. McGraw. The VAL Language: Description and Analysis. *ACM Transactions on Programming Languages and Systems*, 4(1):44–82, 1982.

[74] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface, November 2003.

[75] Soo-Mook Moon. Increasing instruction-level parallelism through multi-way branching. In *ICPP '93: Proceedings of the 1993 International Conference on Parallel Processing*, pages 241–245, Washington, DC, USA, 1993. IEEE Computer Society.

[76] Gordon Moore. Cramming more components onto integrated. *Electronics Magazine*, pages 114–117, April 1965.

[77] NCI. National Compiler Infrastructure NCI. http://www.cs.virginia.edu/nci/, 2008.

[78] NVIDIA. CUDA Zone. www.nvidia.com/cuda, 2008.

[79] Kunle Olukotun, Lance Hammond, and Mark Willey. Improving the performance of speculatively parallel applications on the hydra CMP. In *Proceedings of ICS*, pages 21–30, 1999.

[80] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The case for a single-chip multiprocessor. In *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems (ASPLOS-VII)*, pages 2–11, New York, NY, USA, 1996. ACM Press.

[81] OpenMP Architecture Review Board. OpenMP Application Program Interface. Version 2.5, May 2005.

[82] David A. Patterson and John L. Hennessy. *"Computer Architecture: A Quantitative Approach"*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2003.

[83] Demos Pavlou. Preprocessor and benchmarks for the TFluxSoft platform. Diploma Thesis. University of Cyprus, Computer Science Department, 2008.

[84] Demos Pavlou. TFluxSoft: a Portable Software Runtime System for Parallel Execution Based on the Dataflow Principles for Commodity Multiprocessors. Diploma Thesis. University of Cyprus, Computer Science Department, 2008.

Kyriakos Stavrou

[85] A. Plas, D.Comte, O.Gelly, and J.C.Syre. LAU System Architecture: A Parallel Data-driven Processor Based on Single Assignment. In *Proceedings of the International Conference on Parallel Processing*, pages 293–302, 1976.

[86] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes in C: The Art of Scientific Computing 2nd edition*. Cambridge University Press, 1992.

[87] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 13–24, Washington, DC, USA, 2007. IEEE Computer Society.

[88] Robert Iannucci *et al. Multithreaded Computer Architecture a Summary of the State of the Art*. Kluwer Academic Publishers, 1994.

[89] Manuel Saldana, Daniel Nunes, Emanuel Ramalho, and Paul Chow. Configuration and Programming of Heterogeneous Multiprocessors on a Multi-FPGA System Using TMD-MPI. In *Reconfigurable Computing and FPGA's, 2006. ReConFig 2006. IEEE International Conference on*, pages 1–10, 2006.

[90] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Doug Burger, Stephen W. Keckler, and Charles R. Moore. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. *SIGARCH Computer Architecture News*, 31(2):422–433, 2003.

[91] Ron Sass, William V. Kritikos, Andrew G. Schmidt, Srinivas Beeravolu, and Parag Beeraka. Reconfigurable Computing Cluster (RCC) Project: Investigating the Feasibility of FPGA-Based Petascale Computing. In *Proceedings of the 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '07)*, pages 127–140, 2007.

[92] Paul B. Schneck. A survey of compiler optimization techniques. In *ACM'73: Proceedings of the annual conference*, pages 106–113, New York, NY, USA, 1973. ACM.

[93] Premkishore Shivakumar and Norman P. Jouppi. CACTI 3.0: An integrated cache timing, power and area model. Technical report, Compaq Computer Corporation Western Research Laboratory, August 2001.

[94] J. Sile, B. Robic, and T. Ungerer. Asynchrony in Parallel Computing: From Dataflow to Multithreading. *Parallel and Distributed Computing Practices*, 1(1), March 1998.

[95] Silicon Graphics Inc. Extraordinary Acceleration of Workflows with Reconfigurable Application-specific Computing from SGI. http://www.sgi.com/pdfs/3721.pdf, 2004.

[96] Silicon Graphics Inc. General Purpose Reconfigurable Computing Systems. http://www.sgi.com/pdfs/3721.pdf, 2004.

[97] Silicon Graphics Inc. Reconfigurable Application Specific Computing: Accelerating Production Workflows. http://www.sgi.com/pdfs/3984.pdf, 2006.

[98] Virtutech Simics. *DML 1.0 Reference Manual, Revision 1403*. Virtutech, 2007.

Kyriakos Stavrou

[99] Virtutech Simics. *Simics SunFire Target Guide, Revision 1403*. Virtutech, 2007.

[100] Virtutech Simics. *Simics user guide for Windows, Revision 1403*. Virtutech, 2007.

[101] Virtutech Simics. *Simics x86-440BX Target Guide, Revision 1403*. Virtutech, 2007.

[102] Virtutech Simics. Simics Forum. www.simics.net, 2008.

[103] Kevin Skadron, Mircea R. Stan, Wei Huang, Sivakumar Velusamy, Karthik Sankaranarayanan, and David Tarjan. Temperature-Aware Microarchitecture: Extended Discussion and Results. Technical Report TR-CS-2003-08, University of Virginia, April 2003.

[104] David B. Skillicorn and Domenico Talia. Models and languages for parallel computation. *ACM Comput. Surv.*, 30(2):123–169, 1998.

[105] David Slogsnat, Alexander Giese, and Ulrich Brüning. A versatile, low latency hypertransport core. In *FPGA '07: Proceedings of the 2007 ACM/SIGDA 15th international symposium on Field programmable gate arrays*, pages 45–52, New York, NY, USA, 2007. ACM.

[106] Alan J. Smith. Sequential program prefetching in memory hierarchies. *IEEE Computer*, 11(12):7–21, 1978.

[107] James E. Smith. A study of branch prediction strategies. In *ISCA '81: Proceedings of the 8th annual symposium on Computer Architecture*, pages 135–148, Los Alamitos, CA, USA, 1981. IEEE Computer Society Press.

[108] James E. Smith and Gurindar S. Sohi. The Microarchitecture of Superscalar Processors. In *Proceedings of the IEEE*, volume 83, pages 1609–1624, December 1995.

[109] Michael D Smith. Support for speculative execution in high-performance processors. Technical report, Stanford, CA, USA, 1992.

[110] Michael D. Smith, Mark Horowitz, and Monica S. Lam. Efficient superscalar performance through boosting. In *ASPLOS-V: Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*, pages 248–259, New York, NY, USA, 1992. ACM.

[111] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA 22)*, pages 414–425, 1995.

[112] Kyriakos Stavrou, Paraskevas Evripidou, and Pedro Trancoso. DDM-CMP: Data-Driven Multithreading on a Chip Multiprocessor. In *Proceedings of the 5th International Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, volume 3553, pages 364–373. Lecture Notes in Computer Science, July 2005.

[113] Kyriakos Stavrou, Paraskevas Evripidou, and Pedro Trancoso. Fitting More Data-Driven Multithreading Cores into the Chip. In *Proceedings of the 1st International Summer School on Advanced Computer Architecture and Compilation for Embedded Systems (ACACES 2005)*, pages 83–86, July 2005.

Kyriakos Stavrou

[114] Kyriakos Stavrou, Paraskevas Evripidou, and Pedro Trancoso. Hardware Budget and Runtime System for Data-Driven Multithreaded Chip Multiprocessor. In *Proceedings of the 11th Asia-Pacific Computer Systems Architecture Conference (ACSAC 11)*, pages 244–259, 2006.

[115] Kyriakos Stavrou and Pedro Trancoso. Thermal-Aware Scheduling: A solution for Future Chip Multiprocessors Thermal Problems. In *Proceedings of the 9th Euromicro Conference on Digital System Design (DSD 2006)*, pages 123–126, 2006.

[116] Kyriakos Stavrou, Pedro Trancoso, and Paraskevas Evripidou. Parallel Execution with Data-Driven Multithreading. In *Proceedings of the 3nd International Summer School on Advanced Computer Architecture and Compilation for Embedded Systems (ACACES 2007)*, pages 115–118, July 2007.

[117] Steve Swanson, Ken Michelson, Andrew Schwerin, and Mark Oskin. WaveScalar. In *Proceedings of the 36th Annual International Symposium on Microarchitecture (MICRO-36)*, pages 291–302, 2003.

[118] Steven Swanson, Andrew Putnam, Martha Mercaldi, Ken Michelson, Andrew Petersen, Andrew Schwerin, Mark Oskin, and Susan Eggers. Area-Performance Trade-offs in Tiled Dataflow Architectures. In *Proceedings of the 33rd annual International Symposium on Computer Architecture (ISCA 33)*, pages 314–326, 2006.

[119] Robert Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development*, 11(1):25–33, January 1967.

[120] Pedro Trancoso, Paraskevas Evripidou, Kyriakos Stavrou, and Costas Kyriacou. A case for chip multiprocessors based on the data-driven multithreading model. *International Journal of Parallel Programming*, 34(3):213–235, June 2006.

[121] Pedro Trancoso, Kyriakos Stavrou, and Paraskevas Evripidou. DDMCPP: The Data-Driven Multithreading C Pre-Processor. In *Proceedings of the 11th Annual Workshop on the Interaction between Compilers and Computer Architecture (Interact-11), held in conjunction with the 13th International Symposium on High-Performance Computer Architecture (HPCA-13)*, pages 32–39, 2007.

[122] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *In 22nd Annual International Symposium on Computer Architecture*, pages 392–403, 1995.

[123] Stamatis Vassiliadis, Stephan Wong, Georgi Gaydadjiev, Koen Bertels, Georgi Kuzmanov, and Elena Moscu Panainte. The molen polymorphic processor. *IEEE Trans. Comput.*, 53(11):1363–1375, 2004.

[124] Xiaofang Wang and Sotirios G. Ziavras. Parallel LU factorization of sparse matrices on FPGA-based configurable computing engines: Research Articles. *Concurr. Comput. : Pract. Exper.*, 16(4):319–343, 2004.

[125] K. S. Weng. Stream oriented computation in recursive data-flow schemas. Technical Report Techical Report 68, Laboratory for Computer Science, MIT, Cambridge, MA, 1975.

Kyriakos Stavrou

[126] Tien-Hsiung Weng and Barbara Chapman. Implementing OpenMP Using Dataflow Execution Model for Data Locality and Efficient Parallel Execution. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS 2002)*, pages 01–07, 2002.

[127] Paul G. Whiting and Robert S. V. Pascoe. A history of data-flow languages. *Annals of the History of Computing, IEEE*, 16(4):38–59, 1994.

[128] Robert Wilson, Robert French, Christopher Wilson, Saman Amarasinghe, Jennifer Anderson, Steve Tjiang, Shih Liao, Chau Tseng, Mary Hall, Monica Lam, and John Hennessy. The SUIF Compiler System: a Parallelizing and Optimizing Research Compiler. Technical report, Stanford, CA, USA, 1994.

[129] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd annual International Symposium on Computer Architecture (ISCA 22)*, pages 24–36, 1995.

[130] Kelvin K. Yue and David J. Lilja. Parallel loop scheduling for high performance computers. In *High Performance Computing: Technology, Methods, and Applications*, pages 243–264. Elsevier Publishing Company, 1995.

[131] Weirong Zhu, Vugranam C Sreedhar, Ziang Hu, and Guang R. Gao. Synchronization State Buffer: Supporting Efficient Fine-Grain Synchronization on Many-Core Architectures. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 35–45, New York, NY, USA, 2007. ACM.

Kyriakos Stavrou