University of Cyprus

DEPARTMENT OF COMPUTER SCIENCE

ARCHITECTURAL AND SOFTWARE SUPPORT FOR DATA-DRIVEN EXECUTION ON MULTI-CORE PROCESSORS

DOCTOR OF PHILOSOPHY DISSERTATION

GEORGE MATHEOU

2017

GEORGE MATHEOU

University of Cyprus

DEPARTMENT OF COMPUTER SCIENCE

ARCHITECTURAL AND SOFTWARE SUPPORT FOR DATA-DRIVEN EXECUTION ON MULTI-CORE PROCESSORS

GEORGE MATHEOU

A Dissertation Submitted to the University of Cyprus in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

November, 2017

GEORGE MATHEOU

GEORGE MATHEOU

@George Matheou, 2017

VALIDATION PAGE

Doctoral Candidate: George Matheou

Doctoral Dissertation Title: Architectural and software support for data-driven execution on multi-core processors

The present Doctoral Dissertation was submitted in partial fulfillment of the requirements for the Degree of Doctor of Philosophy at the ***Department of Computer Science*** *and was approved on November 27, 2017 by the members of the* ***Examination Committee****.*

Examination Committee:

Research Supervisor: Professor Paraskevas Evripidou

Committee Member: Professor Constantinos S. Pattichis

Committee Member: Assistant Professor Theocharis Theocharides

Committee Member: Professor Ian Watson

Committee Member: Dr. Albert Cohen

GEORGE MATHEOU

DECLARATION OF DOCTORAL CANDIDATE

The present doctoral dissertation was submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy of the University of Cyprus. It is a product of original work of my own, unless otherwise mentioned through references, notes, or any other statements.

George Matheou

GEORGE MATHEOU

ABSTRACT

Το τέλος της εκθετικής ανάπτυξης των σειριακών επεξεργαστών έχει διευκολύνει την ανάπτυξη των πολυπύρηνων συστημάτων. Έτσι, οποιαδήποτε αύξηση της απόδοσης πρέπει να προέρχεται από τον παραλληλισμό. Για να επιτευχθεί αυτό, πρέπει να αναπτυχθούν αποτελεσματικά μοντέλα παράλληλου προγραμματισμού/εκτέλεσης. Προτείνουμε την ανάπτυξη τέτοιων συστημάτων χρησιμοποιώντας το μοντέλο εκτέλεσης Data-Driven Multithreading (DDM). Το DDM είναι ένα πολυνηματικό μοντέλο που συνδυάζει ταυτοχρονισμό, βασισμένο στο δυναμικό μοντέλο ροής δεδομένων, και αποδοτική διαδοχική εκτέλεση σε συμβατικούς επεξεργαστές. Το DDM χρησιμοποιεί το Thread Scheduling Unit (TSU) για τη χρονοδρομολόγηση των νημάτων κατά τη διάρκεια εκτέλεσης, βάσει της διαθεσιμότητας των δεδομένων. Σε αυτό το έργο, παρέχουμε αρχιτεκτονική και λογισμική υποστήριξη για την αποτελεσματική εκτέλεση σε πολυπύρηνες αρχιτεκτονικές, μέσω δύο διαφορετικών υλοποιήσεων που βασίζονται στο μοντέλο DDM.

Η πρώτη υλοποίηση πραγματοποιεί το μοντέλο DDM στο υλικό, χρησιμοποιώντας Field Programmable Gate Arrays (FPGAs). Η υλοποίηση αυτή στοχεύει να βοηθήσει στην ανάπτυξη μελλοντικών πολυπύρηνων συστημάτων υψηλής απόδοσης και χαμηλής ισχύος. Το TSU υλοποιήθηκε σε υλικό χρησιμοποιώντας τη γλώσσα προγραμματισμού Verilog και ενσωματώθηκε σε ένα πολυπύρηνο επεξεργαστή με μη-συνεκτικούς και χαμηλής πολυπλοκότητας πυρήνες. Ο επεξεργαστής αυτός ονομάζεται MiDAS (Multi-core with Data-Driven Architectural Support) και έχει παραχθεί σε πρωτότυπο χρησιμοποιώντας ένα Xilinx Virtex-6 FPGA. Ο MiDAS επεξεργαστής έχει αξιολογηθεί χρησιμοποιώντας εφαρμογές με διαφορετικά χαρακτηριστικά, οι οποίες αναπτύχθηκαν σε C/C++ χρησιμοποιώντας μια διεπαφή προγραμματισμού εφαρμογών (API). Η αξιολόγηση της απόδοσης του MiDAS έδειξε ότι η αρχιτεκτονική υποστήριξη για την εκτέλεση ροής δεδομένων μπορεί να επιτύχει πολύ καλά αποτελέσματα, ακόμη και σε εφαρμογές με πολύ μικρά μεγέθη προβλημάτων.

Στο πλαίσιο αυτής της εργασίας, παρέχουμε αρκετά αποτελέσματα για τον MiDAS επεξεργαστή, όπως για παράδειγμα τη χρήση των πόρων του FPGA, εκτιμήσεις για την κατανάλωση ενέργειας και καθυστερήσεις (σε κύκλους) των διαφόρων λειτουργιών του TSU. Τα αποτελέσματα δείχνουν ότι το TSU μπορεί να υλοποιηθεί με μικρό προϋπολογισμό υλικού. Το TSU συγκρίνεται με το Task Superscalar, μια αρχιτεκτονική που υλοποιεί το μοντέλο StarSs σε υλικό, χρησιμοποιώντας απαιτήσεις σε πόρους και μακρο-στατιστικές. Τα αποτελέσματα δείχνουν ότι η υλοποίηση ενός μοντέλου ροής δεδομένων στο υλικό, που ανιχνεύει δυναμικά εξαρτήσεις

GEORGE MATHEOU

μεταξύ εργασιών και κατασκευάζει το γράφημα εξαρτήσεων κατά τη διάρκεια εκτέλεσης, όπως το Task Superscalar, αυξάνει σημαντικά τη χρήση των πόρων (και κατά συνέπεια την κατανάλωση ενέργειας).

Η δεύτερη υλοποίηση, ονομαζόμενη FREDDO (efficient Framework for Runtime Execution of Data-Driven Objects), είναι μια αποδοτική και φορητή αντικειμενοστρεφής υλοποίηση του μοντέλου DDM που επιτρέπει χρονοδρομολόγηση βασισμένη στο μοντέλο ροής δεδομένων σε κατανεμημένα συστήματα με συμβατικούς πολυπύρηνους επεξεργαστές. Το FREDDO στοχεύει στην αποδοτική DDM εκτέλεση σε κατανεμημένα συστήματα υπολογισμού υψηλών επιδόσεων. Παρέχει επίσης νέες δυνατότητες στο μοντέλο DDM, όπως υποστήριξη αναδρομής και επεκτείνει τη διεπαφή προγραμματισμού του DDM με αντικειμενοστρεφή προγραμματισμό. Το FREDDO αξιολογήθηκε σε δύο διαφορετικά συστήματα: ένα σύστημα 4-κόμβων AMD με συνολικά 128 πυρήνες και ένα σύστημα 64-κόμβων Intel με συνολικά 768 πυρήνες. Η αξιολόγηση της απόδοσης δείχνει ότι το προτεινόμενο σύστημα κλιμακώνεται καλά και ανέχεται αποτελεσματικά το κόστος χρονοδρομολόγησης και τις καθυστερήσεις μνήμης. Επίσης, συγκρίνουμε το FREDDO με τα συστήματα OpenMP, MPI, DDM-VM και OmpSs. Τα αποτελέσματα σύγκρισης δείχνουν ότι το προτεινόμενο σύστημα επιτυγχάνει συγκρίσιμες ή καλύτερες επιδόσεις.

GEORGE MATHEOU

ABSTRACT

The end of the exponential growth of the sequential processors has facilitated the development of multi-core systems. Thus, any growth in performance must come from parallelism. To achieve that, efficient parallel programming/execution models must be developed. We propose to develop such systems using the Data-Driven Multithreading (DDM) model of execution. DDM is a non-blocking multithreading model that combines dynamic data-flow concurrency with efficient sequential execution on conventional processors. DDM utilizes the Thread Scheduling Unit (TSU) for scheduling threads at runtime, based on data availability. In this work, we provide architectural and software support for efficient data-driven execution on multi-core architectures, through two different DDM-based implementations.

The first implementation realizes the DDM model in hardware, using Field Programmable Gate Arrays (FPGAs). The hardware DDM implementation aims to help in the development of future high-performance and low-power multi-core systems. DDM's TSU was implemented in hardware using Verilog. The hardware TSU implementation was integrated into a shared-memory multi-core processor with non-coherent in-order cores, called MiDAS (Multi-core with Data-Driven Architectural Support). MiDAS was prototyped and evaluated on a Xilinx Virtex-6 FPGA using benchmarks with different characteristics. The benchmarks were developed in C/C++ using a software API. The performance evaluation of MiDAS has shown that the architectural support for data-driven execution can achieve very good results, even on benchmarks with very small problem sizes.

We provide several results for the hardware TSU and MiDAS, including FPGA resource requirements, power consumption estimations and latencies (in cycles) of various TSU operations. The results show that TSU can be implemented in hardware with a small hardware budget. The proposed TSU is compared with Task Superscalar, an architecture that implements the StarSs programming framework in hardware. The results show that implementing a data-driven model in hardware, that dynamically detects inter-task dependencies and constructs the dependency graph at runtime, like Task Superscalar, significantly increases the resource requirements and power consumption.

GEORGE MATHEOU

The second implementation, called FREDDO (efficient Framework for Runtime Execution of Data-Driven Objects), is an efficient and portable object-oriented implementation of DDM that enables data-driven scheduling on conventional single-node and distributed multi-core systems. The FREDDO implementation aims to allow efficient DDM execution on distributed High Performance Computing (HPC) systems. It also provides new features to the DDM model like recursion support and it extends DDM's programming interface with the object-oriented programming paradigm. FREDDO was evaluated on two different systems: a 4-node AMD system with a total of 128 cores and a 64-node Intel HPC system with a total of 768 cores. The performance evaluation shows that the proposed framework scales well and tolerates scheduling overheads and memory latencies effectively. We also compare our framework to OpenMP, MPI, DDM-VM and OmpSs. The comparison results show that the proposed framework obtains comparable or better performance.

GEORGE MATHEOU

ACKNOWLEDGEMENTS

I would like to thank my advisor, Professor Paraskevas Evripidou, for his guidance and support during the completion of this thesis. I am forever indebted to him for accepting me as his student and for his unconditional support, especially during difficult times.

For parts of this work I need to acknowledge and thank other researchers. I would like to thank, Dr. Pedro Trancoso, Dr. Costas Kyriacou, Dr. Samer Arandi, George Michael and Andreas Diavastos for their invaluable help and support. Thanks for the inspiring opinions and insightful discussions, and for helping me to understand the different implementations of the Data-Driven Multithreading (DDM) model.

In addition, I would like to thank my committee members, Professor Constantinos S. Pattichis, Assistant Professor Theocharis Theocharides, Professor Ian Watson and Dr. Albert Cohen, for their valuable comments.

I truly thank my friend and best man, Diomidis Papadiomidous for being supportive throughout my time here and for helping me with proofreading my publications. I also thank my friends for providing support and friendship that I needed . Thanks to Constantinos Costa, George Nikolaides, George Larkou, Panagiwta Nikolaou, Panagiwtis Loizias, Loizos Ioakim, Paraskevas Koutras, Xrisos Vasiliou and Andreas Dimitriou. If I have forgotten anyone, I apologize.

I am also grateful to the funding sources that made my Ph.D. work possible. This work was partially funded by the University of Cyprus, by the Cyprus State Scholarship Foundation (IKYK), and by the EU TERAFLUX project.

Last but not least, I would like to thank my family; my father Adamos, my mother Eleni, my brothers Theodosis, Petros and Andreas, and my wife's parents and sisters. Their constant care and support have helped me through tough times, and I cannot thank them enough for it. I would like to thank a very special person, my wife, Margarita for her patience, unwavering support and partnership in my life. My son, Adamos-Panagiwtis, who bring so much fun, excitement, and entertainment to my life. Thank you Adamos-Panagiwtis!

GEORGE MATHEOU

TABLE OF CONTENTS

Chapter 1: Introduction **1**

1.1 Motivation . . . 1

1.2 Thesis Statement . . . 2

1.3 Approach . . . 3

1.4 Thesis Contributions . . . 3

1.5 Thesis Outline . . . 5

Chapter 2: Related Work **6**

2.1 Introduction . . . 6

2.2 The shift to the multi-core era . . . 6

2.3 Field Programmable Gate Arrays (FPGAs) . . . 9

2.4 The Data-flow model of execution . . . 10

2.4.1 Pure Data-flow Architectures . . . 11

2.4.2 Hybrid Data-flow Architectures . . . 14

2.5 Recent Data-flow Developments . . . 16

2.5.1 Software Implementations . . . 16

2.5.2 Hardware Implementations . . . 19

2.6 Data-Driven Multithreading (DDM) . . . 24

2.6.1 Context and Nesting attributes . . . 24

2.6.2 Thread Template . . . 26

2.6.3 DDM Dependency Graph . . . 26

2.6.4 DDM Implementations . . . 27

2.7 Concluding Remarks . . . 38

Chapter 3: MiDAS: a Multi-core system with Data-Driven Architectural Support **42**

3.1 Introduction . . . 42

3.2 TSU: Hardware Support for DDM . . . 42

3.2.1 Thread Template . . . 43

3.2.2 TSU Micro-architecture . . . 43

3.2.3 TSU's RTL schematics . . . 52

3.3 MiDAS System Architecture . . . 59

3.3.1 Memory Model . . . 60

Chapter 4: FREDDO: an efficient Framework for Runtime Execution of Data-Driven Objects **61**

GEORGE MATHEOU

4.1 Introduction . . . 61
4.2 Single-node Implementation . . . 62
4.2.1 New features . . . 62
4.2.2 Architecture . . . 66
4.3 Distributed Implementation . . . 71
4.3.1 Architecture . . . 71
4.3.2 Memory Model . . . 72
4.3.3 Distribution Scheme and Scheduling Mechanisms for DThread instances . . . 74
4.3.4 Network Manager . . . 75
4.3.5 Distributed Execution Termination . . . 76
4.3.6 Reducing Network Traffic . . . 77

Chapter 5: Programming Methodology 79
5.1 Introduction . . . 79
5.2 FREDDO API . . . 80
5.2.1 Basic Runtime Functions . . . 80
5.2.2 DFunctions . . . 81
5.2.3 DThread Classes . . . 82
5.2.4 UML Diagram of DThread Classes . . . 85
5.3 Programming examples using FREDDO . . . 85
5.3.1 Simple application . . . 85
5.3.2 Synthetic application . . . 87
5.3.3 Tile LU Decomposition: single-node and distributed implementations . . . 89
5.4 MiDAS API . . . 93
5.4.1 Common functions for the C and C++ APIs . . . 95
5.4.2 C API . . . 95
5.4.3 C++ API . . . 96
5.5 Implementing Matrix Multiplication for MiDAS . . . 97
5.5.1 Implementation using the C API . . . 98
5.5.2 Implementation using the C++ API . . . 99
5.5.3 Implementation using TFlux directives . . . 100

Chapter 6: Recursion Support for the DDM model 102
6.1 Introduction . . . 102
6.2 The U-Interpreter and PBGR models . . . 102
6.2.1 The U-Interpreter model . . . 102
6.2.2 The PBGR model . . . 104

GEORGE MATHEOU

6.2.3 Differences between DDM and PBGR . . . 105
6.3 Basic functionalities for supporting recursion in DDM . . . 105
6.4 DThread Classes for Recursion Support in FREDDO . . . 109
6.4.1 RecursiveDThreadWithContinuation Class . . . 110
6.4.2 RecursiveDThread and ContinuationDThread Classes . . . 111
6.4.3 Distributed Recursion Support . . . 113
6.5 Implementing the recursive Fibonacci algorithm in FREDDO . . . 114
6.5.1 Implementation using the RecursiveDThreadWithContinuation Class . . . 114
6.5.2 Implementation using the RecursiveDThread and ContinuationDThread Classes 114
6.5.3 Distributed Implementation . . . 115

Chapter 7: Evaluation 118

7.1 Introduction . . . 118
7.2 Benchmark Suite . . . 118
7.2.1 Benchmarks with simple dependency graphs . . . 118
7.2.2 Benchmarks with complex dependency graphs . . . 119
7.2.3 Recursive algorithms . . . 120
7.3 Experimentation Infrastructure . . . 121
7.4 The evaluation of the hardware TSU and MiDAS . . . 123
7.4.1 TSU Resource Requirements . . . 123
7.4.2 Latencies (in cycles) of various TSU Operations . . . 127
7.4.3 Performance Evaluation of MiDAS architecture . . . 127
7.4.4 FPGA Resource Requirements and Power Consumption Estimations for MiDAS . . . 132
7.4.5 DDM Architectural Support in MiDAS vs. Task Superscalar Architecture . . . 134
7.4.6 Hardware vs. Software TSU - Preliminary Results . . . 135
7.5 Single-node FREDDO Evaluation . . . 138
7.5.1 Experimental Setup . . . 138
7.5.2 Performance Evaluation . . . 139
7.5.3 Comparisons . . . 141
7.6 Distributed FREDDO Evaluation . . . 142
7.6.1 Experimental Setup . . . 142
7.6.2 Performance Evaluation . . . 143
7.6.3 FREDDO: CNI vs MPI . . . 146
7.6.4 Performance comparisons with other systems . . . 147
7.6.5 Network Traffic Analysis . . . 151

GEORGE MATHEOU

7.6.6 Execution Times 152

Chapter 8: Conclusions and Future Work **155**

8.1 Conclusions 155

8.2 Future Work 156

8.2.1 MiDAS 157

8.2.2 FREDDO 159

8.2.3 Extending the functionalities of DDM 160

Bibliography **161**

Appendices **173**

Appendix A: Publications **174**

GEORGE MATHEOU

LIST OF TABLES

1 Early representative hardware data-flow prototypes (from 1974 to 1992). 39
2 Recent hardware data-flow developments (real and simulated implementations). . . . 40
3 Recent software data-flow developments. 41
4 Context encoding according to the Nesting attribute for each Context size. 64
5 Systems used for the benchmark evaluation of FREDDO. 122
6 Latencies (in cycles) of various TSU Operations. 126
7 Benchmark suite characteristics used in evaluating MiDAS's performance. 128
8 Sequential execution time of the benchmarks running on MiDAS. 128
9 Average speedup and efficiency for each problem size and number of enabled cores. 130
10 Characteristics of DThreads for each benchmark running on MiDAS. 131
11 Virtex-6 FPGA resource requirements and power consumption estimations in implementing MiDAS incorporating either the PO-TSU or the AO-TSU. 132
12 Resource requirements and macro statistics of the proposed TSU vs. Task Superscalar. 135
13 Versions of *Synth* benchmark used for comparing software and hardware TSUs. . . . 136
14 The benchmark suite characteristics for the FREDDO evaluation. 139
15 Average sequential execution time (in seconds) of the sequential version of the benchmarks. 145
16 Thresholds used for the execution of the recursive algorithms. 145
17 Speedup results along with the utilization percentage of the available cores in each case. 146
18 Best average execution time (in seconds) for FREDDO+CNI on AMD. 153
19 FREDDO+MPI vs. MPI: best average execution time (in seconds) on AMD. 153
20 FREDDO+CNI vs. DDM-VM: best average execution time (in seconds) on AMD. . 153
21 Best average execution time (in seconds) for FREDDO+MPI on CyTera. 154
22 FREDDO+MPI vs. MPI: best average execution time (in seconds) on CyTera. . . . 154
23 FREDDO+MPI vs. OmpSs@Cluster: best average execution time (in seconds) on CyTera. 154

GEORGE MATHEOU

LIST OF FIGURES

1 Growth in processor performance since the mid-1980s [1]. 7
2 MPPA many-core architecture (C = Compute Cluster) [2]. 8
3 The architecture of the MPPA Compute Cluster [2]. 8
4 A high-level view of a platform FPGA [3]. 9
5 An example of the direct matching approach. 13
6 High-level view of Task Superscalar (retrieved from [4]). 23
7 Computing with Maxeler's implementation of streaming data-flow cores [5]. 23
8 Example of using multiple instances of the same DThread. 25
9 Example of a DThread that parallelizes a two-level nested loop. 25
10 Example of a DThread that parallelizes a three-level nested loop. 25
11 Example of a DDM Dependency Graph. 27
12 The D^2NOW architecture [6]. 28
13 A DDM Node [6]. 29
14 The TSU's internal structure [6]. 29
15 The TIU with the basic prefetch CacheFlow policy [6]. 30
16 Several alternatives of the DDM-CMP architecture [7]. 31
17 The layered design of the TFlux Platform [8]. 32
18 A TFluxHard chip with 4 cores [8]. 33
19 TFluxSoft system on a system with n CPUs [9]. 33
20 The TFluxCell system [8]. 34
21 The architecture of the DDM-VMc [10]. 35
22 The architecture of the DDM-VMs [11]. 36
23 The architecture of the Distributed DDM-VM [11]. 37
24 Block diagram of the TSU micro-architecture supporting an arbitrary number of cores. 45
25 Block diagram of the Template Memory. 46
26 Block diagram of the hardware Dynamic Synchronization Memory. 49
27 High-level RTL schematic of the TSU. 54
28 High-level RTL schematic of the Fetch Unit and the Update Queue. 55
29 High-level RTL schematic of the Update Unit connected with Template Memory, Update Queue and Ready Queue. 56
30 High-level RTL schematic of the Update Unit. 57
31 High-level RTL schematic of the TSU's output side (Ready Queue, Scheduling Unit, Waiting Queues and Transfer Units). 58
32 MiDAS architecture supporting an arbitrary number of cores. 59

GEORGE MATHEOU

33 LU algorithm: DThreads and Context values. 63
34 Architecture of the single-node FREDDO implementation. 66
35 Block diagram of the FREDDO's TSU. 67
36 The TSU's basic data structures. 68
37 Example of computing the RC values of Pending Thread Templates (PTT=Pending Thread Template, TT=Thread Template). 70
38 The FREDDO's Distributed Architecture. 72
39 Example of reducing the network traffic generated by a Multiple Update. T1(X,Y) denotes a Multiple Update for DThread T1. 78
40 The UML diagram of all DThread classes. 85
41 The DDM dependency graph of a simple application. 86
42 Example of a synthetic DDM application. 87
43 LU Decomposition: dependencies between operations for the first iteration. 90
44 The LU's DDM dependency graph for the first two iterations of a 3×3 tile matrix (N=3). 91
45 FREDDO code of the tile LU algorithm (the highlighted code is required for the distributed execution). 92
46 Programming methodology of the MiDAS system. 94
47 Matrix Multiplication: dynamic instantiations of *thread_1*. 98
48 A program that computes the square of a number. 103
49 U-Interpreter graph of the square function call. 103
50 U-Interpreter graph of Fibonacci. 104
51 PBGR evaluation of fib(2). 104
52 RData implemented as a fixed-size array. 106
53 RData implemented as a hash-map. 106
54 The Fibonacci's DDM Dependency Graph. 108
55 Task graphs of high complexity algorithms. 120
56 A solution to the 4Queens problem [12]. 121
57 Knight's graph showing all possible paths for a knight's tour on a standard 8×8 chessboard [13]. 121
58 The Xilinx ML605 Evaluation Board. 122
59 Effect of TID on TSU resource requirements. 124
60 Effect of RC size on TSU resource requirements. 125
61 Effect of CSE number on TSU resource requirements. 125
62 Effect of the number of cores on TSU resource requirements. 126

GEORGE MATHEOU

63 MiDAS's performance using the PO-TSU implementation under various numbers of enabled cores and problem sizes. 129
64 Performances of MiDAS using the PO-TSU vs. MiDAS using the AO-TSU under the Cholesky and LU benchmarks. 131
65 Comparing the DynamicSM's latencies of PO-TSU and AO-TSU. 132
66 Per-component resource utilization and power consumption of MiDAS using PO-TSU. 133
67 Per-component resource utilization and power consumption of MiDAS using AO-TSU. 133
68 Per-component power consumption of the PO-TSU and AO-TSU. 134
69 Hardware vs. Software TSU on *Synth*'s versions that do not use an SM implementation. 137
70 Hardware vs. Software TSU on *Synth*'s versions that use an SM implementation. . . 137
71 Performance scalability of FREDDO for different number of computation cores (Kernels) and problem sizes. 140
72 FREDDO vs. OmpSs on an AMD node using 32 cores. 141
73 FREDDO vs. OpenMP on an AMD node using 32 cores. 142
74 Strong scalability and problem size effect on the AMD system using FREDDO+CNI (MS=Matrix Size, SP=Single-Precision, DP=Double-Precision, $K = 2^{10}$, $M = 10^6$). 144
75 Strong scalability and problem size effect on the CyTera system using FREDDO+MPI (MS=Matrix Size, SP=Single-Precision, $K = 2^{10}$). 145
76 FREDDO+CNI vs. FREDDO+MPI on AMD for the 4-node configuration. P1, P2 and P3 indicate the smaller, medium and largest problem sizes, respectively. 147
77 FREDDO+MPI vs. MPI on CyTera. 148
78 FREDDO+MPI vs. MPI on AMD. 148
79 FREDDO+CNI vs. DDM-VM on AMD (MS: $32K \times 32K$). 149
80 FREDDO+MPI vs. OmpSs@Cluster on CyTera (MS: $60K \times 60K$). 150
81 Network traffic analysis: FREDDO against DDM-VM on the AMD system, for the 4-node configuration and the largest problem size ($32K \times 32K$). 151
82 Tile size effect on the AMD and CyTera systems using FREDDO. 152
83 Future distributed data-driven many-core implementation. 157

GEORGE MATHEOU

Chapter 1

Introduction

The sequential model of execution has dominated digital computing since the early 1940's. Chip designers were using the exponentially increasing number of transistors (predicated by Moore's Law [14]) to improve the performance of single-chip microprocessors, by increasing the clock frequency and designing more complex processors with larger cache sizes and sophisticated hardware mechanisms such as *pipelining* and *out-of-order* execution. Nevertheless, the encountered memory, power and Instruction Level Parallelism (ILP) Walls have slowed down uniprocessor performance [1]. As a result, the entire industry has shifted from single-core-based to multi-core-based systems to expand performance envelopes [1, 15]. Multi-core processors utilize multiple cores on the same die in order to achieve higher performance. These cores usually have lower frequencies and are simpler than the traditional monolithic designs leading to power efficient systems. Ongoing technology trends dictate that multi-core chips will continue to accommodate an increasing number of cores in the years to come as a means of achieving constantly increasing performance through parallel execution, a trend that escalates toward the many-core paradigm. Currently, multi-core architectures have dominated the High Performance Computing (HPC) field, from shared memory systems to large-scale distributed memory clusters (e.g., supercomputers) [16].

1.1 Motivation

The switch to multi-core architectures has elevated *concurrency/parallelism* as the main source for achieving high performance [17]. Programming of such systems is mainly done through parallel extensions of the sequential model like MPI [18] and OpenMP [19]. These extensions do facilitate high productivity parallel programming but also suffer from the inability to tolerate long memory latencies and waits due to synchronization events [20, 21, 22]. As a result, the computational resources now available in multi-core-based systems, such as single-chip shared-memory processors and distributed multi-core clusters, are not efficiently utilized [16].

GEORGE MATHEOU

Indeed, high performance linear algebra software libraries, like LAPACK [23], have shown limitations on multi-core architectures since the parallelism is based on the expensive fork-join paradigm [24, 25, 26]. Such libraries are basic components of the traditional software stack, thus should be redesigned to take advantage of the available on-chip resources. Furthermore, realistic applications running on current supercomputers typically use only 5%-10% of the machine's peak processing power at any given time [21]. Even worse, as the number of cores will inevitably be increased in the coming years, the fraction that can be kept busy at any given time can be expected to plummet [21]. *As such, new programming/execution models need to be developed in order to efficiently utilize the resources of multi-core architectures*. We propose the implementation of such programming/execution models based on *data-flow* [27, 28, 29, 30].

Systems based on data-flow/data-driven execution have several advantages over the sequential model of execution: *(i)* allow asynchronous data-driven execution of fine-grained tasks/threads; fine-grain programming models have a great potential to efficiently use the underlying hardware [31, 32, 22, 33], *(ii)* can expose the maximum degree of parallelism in a program since the data-flow model only enforces true data-dependencies [34], *(iii)* can handle concurrency and tolerate memory and synchronization latencies efficiently [20] and *(iv)* can exclude power hungry modules like out-of-order execution (since we have only true data-dependencies) and utilize non-coherent memory hierarchies [10]; this can lead to simpler and more power efficient designs. Thus, data-flow-based systems can be used to efficiently exploit the computing power of current and future multi-core architectures.

Data-flow systems can be realized in both software and hardware or can be provided as simulated hardware implementations. Software data-flow systems are provided mostly in the form of virtual machines and runtime libraries [35, 36, 37, 38, 39, 40]. Real hardware data-flow implementations are provided as Application-Specific Integrated Circuits (ASICs) [29, 41, 42, 43, 44, 45] or are implemented in Field Programmable Gate Arrays (FPGAs) using hardware description languages [46, 47, 48, 49, 50, 51]. Finally, simulated data-flow hardware systems are implemented and evaluated using software simulators [52, 53, 54, 55, 56, 9, 4]. Software data-flow implementations can be used to allow data-flow/data-driven concurrency on conventional/commodity multi-core systems. On the other hand, a hardware implementation can deliver the ultimate performance compared to a software implementation with the same functionalities [4, 57].

1.2 Thesis Statement

In this thesis, we try to solve the problem of finding a suitable execution model that utilizes the resources of multi-core-based systems by software and architectural techniques.

GEORGE MATHEOU

1.3 Approach

We propose a paradigm shift to a hybrid control-flow/data-flow model, the Data-Driven Multithreading (DDM) model [55], as the basis for an execution model that efficiently exploits the resources of multi-core architectures. DDM is a non-blocking multithreading model which combines dynamic data-flow concurrency with efficient sequential execution on conventional processors. The core of the DDM model is the *Thread Scheduling Unit (TSU)* [58] which is responsible for scheduling threads at runtime, based on data availability. The goal of this thesis is twofold:

1. Design, implement and evaluate a real hardware data-flow/data-driven system using commodity processors. We choose to implement a real hardware implementation instead of a simulated one since the former allows a deeper and more accurate analysis of the system as well as better performance. The proposed hardware system aims to help in the development of future high-performance and low-power multi-core systems. Early hardware data-flow prototypes [29, 41, 42] as well as recent ones [45, 59, 54, 60, 53, 56], failed to convince the electronics industry in terms of being feasible and sustainable systems, as they could not utilize, at the time, current processor technology; instead they used customized designs which could not easily become mainstream. However, they inspired several data-flow projects including our work. We are proposing a DDM-based hardware system which can be implemented on unmodified commodity microprocessors, thus allowing utilizing the state-of-the-art in processor design.
2. Design, implement and evaluate an efficient and portable programming framework, based on the DDM model, that enables dynamic data-flow/data-driven concurrency on commodity multi-core architectures (single-chip shared-memory processors and distributed multi-core clusters).

1.4 Thesis Contributions

The contributions of this thesis can be summarized as follows:

- **Contribution 1:** the design, implementation and evaluation of DDM's Thread Scheduling Unit (TSU), in hardware, using the Verilog HDL. TSU was implemented as a fully parameterizable hardware Intellectual Property (IP) core. The hardware TSU was synthesized with different configurations and several results are provided, including FPGA resource requirements, power consumption estimations and latencies (in cycles) of various TSU operations. Finally, the TSU was compared with the Task Superscalar architecture [46] using FPGA resource requirements and macro statistics. The results show that implementing a data-driven model in hardware, that dynamically detects inter-task dependencies and constructs the dependency graph at runtime, like Task Superscalar, significantly increases the resource requirements and power consumption.

GEORGE MATHEOU

- **Contribution 2:** the design, implementation and evaluation of a shared-memory multi-core processor paired with the hardware TSU implementation, called MiDAS. The processor consists of in-order non-coherent processing elements implemented using the Xilinx MicroBlaze soft-core [61]. MiDAS was prototyped and evaluated on a Xilinx ML605 Evaluation Board, that is equipped with a Xilinx Virtex-6 FPGA [62], using benchmarks with different characteristics. The benchmarks were developed in C/C++ using a software Application Programming Interface (API). The API allows users to manage the processor's hardware peripherals (timers, interrupt controllers, memory controller, etc.) and to communicate with the TSU. Finally, FPGA resource requirements and power consumption estimations of the MiDAS system are provided.
- **Contribution 3:** the design, implementation and evaluation of FREDDO. It is an efficient and portable object-oriented implementation of the DDM model [55]. FREDDO is a C++ framework that supports efficient data-driven execution on conventional multi-core clusters.
    1. FREDDO is a high-performance implementation of DDM that achieves better performance than similar state-of-the-art systems.
    2. It extends the programming interface of DDM with the object-oriented programming (OOP) paradigm. This allows DDM applications to benefit from OOP concepts such as Data Abstraction, Encapsulation and Inheritance.
    3. It evaluates the DDM model on HPC systems. Particularly, FREDDO was evaluated on an open-access 64-node Intel HPC system with a total of 768 cores. The DDM model was previously evaluated only on very small distributed multi-core systems with up to 24 cores using the DDM-VM implementation [63].
    4. It provides simple mechanisms/optimizations to reduce the network traffic of distributed DDM applications.
    5. It utilizes a connectivity layer with two different network interfaces: a Custom Network Interface (CNI) and MPI [18]. The CNI support allows to have a direct and fair comparison with frameworks that also utilize a custom network interface (e.g., DDM-VM [63]) where the MPI support provides portability and flexibility to the FREDDO framework.
    6. FREDDO is the first open-source implementation of the DDM model. It is publicly available for download in *https://github.com/george-matheou/freddo-project*, under GNU General Public License v3.0.
- **Contribution 4:** provide recursion support for the DDM model. This functionality is based on two different data-flow models, U-Interpreter [64] and Packet Based Graph Reduction (PBGR) [65, 66, 67]. The recursion support was implemented under the FREDDO framework which allows single-node and distributed execution of recursive algorithms.

GEORGE MATHEOU

1.5 Thesis Outline

This thesis is organized as follows. Chapter 2 presents the related work of this thesis. It introduces the historical background of data-flow/data-driven computing and presents recent research and projects that are relevant. Also, the Data-Driven Multithreading (DDM) model [55] and its implementations are presented. Chapter 3 presents the MiDAS system. Chapter 4 describes the single-node and distributed FREDDO implementation. Chapter 5 describes the programming methodology with an emphasis on the description of the APIs used in FREDDO and MiDAS. The recursion support of the DDM model is presented in Chapter 6. It also presents the new functionalities implemented in FREDDO's API in order to allow parallel/distributed execution of recursive algorithms, under the DDM model. Chapter 7 presents the evaluation results for the FREDDO framework and MiDAS. The benchmark suite used in this thesis as well as the hardware environment used in our experiments are also presented. Finally, Chapter 8 concludes this thesis and provides directions for future work.

GEORGE MATHEOU

Chapter 2

Related Work

2.1 Introduction

In this chapter we present a representative subset of the related work relevant to the concepts of this thesis. Section 2.2 discusses the factors that led to the development of multi-core architectures whereas Section 2.3 briefly describes the Field Programmable Gate Array (FPGA) technology. Following that, we introduce the data-flow model of computation (Section 2.4) which was proposed by Jack Dennis [27, 68] in 1974, as an alternative to the control-flow model of execution. The evolution of data-flow architectures is described, from *pure data-flow* (Static, Dynamic and Explicit Token Store) to *hybrid data-flow/control-flow* architectures. Data-flow principles are currently used in modern processor architectures (e.g., out-of-order execution [1] and non-blocking threads) and compiler technologies (e.g., register renaming). Several research projects use today's mature hardware technology and apply the data-flow paradigm to a coarser-grained level in order to minimize the hardware requirements and make their implementation feasible. In Section 2.5 we give an overview of recent software and hardware data-flow projects. Section 2.6 presents the Data-Driven Multithreading (DDM) model of execution [55]. Finally, the concluding remarks are presented in Section 2.7.

2.2 The shift to the multi-core era

The previous decades the performance of the microprocessors was increased due to the advances in integrated circuit (IC) technology [15]. By increasing the integration density (more and faster transistors into the same chip) it is possible to achieve higher clock rates, use large sizes of cache and extracting more instruction-level parallelism (ILP) by implementing more sophisticated hardware mechanisms [69]. Such mechanisms include: *pipelining*, *out-of-order* execution (or dynamic scheduling), branch prediction and *superscalar* execution.

Although the aforementioned techniques improved significantly the performance of single-chip microprocessors have lead to the memory, power and ILP walls. Advances in speed of commodity

GEORGE MATHEOU

CPUs have far outpaced advances in memory (DRAM) latency. Thus, main-memory access become a performance bottleneck for many computer applications; a phenomenon that is widely known as the memory wall [70]. Several techniques were proposed to reduce the negative effect of the memory wall, like software and hardware prefetching and on-chip caches. The power wall is the trend of consuming exponentially increasing power due to the complexity of the designs. This state have lead to the generation of more heat inside the chips. The ILP wall refers to the increasing difficulty of finding enough parallelism in a single instructions stream to keep a high-performance single-core processor busy.

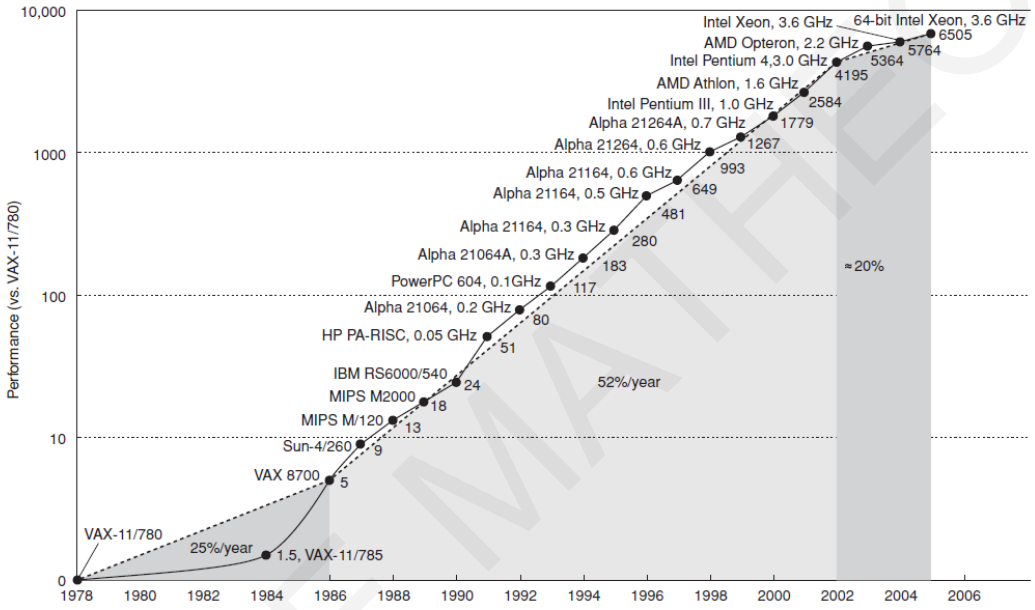


Figure 1: Growth in processor performance since the mid-1980s [1].

Figure 1 depicts the growth in processor performance since the mid-1980s as measured by the SPECint benchmarks. Prior to the mid-1980s, processor performance was averaged about 25% per year. From 1986 to 2001 the performance growth to about 52% due to more advanced architectural and organizational ideas. By 2002, the limits (walls) we mentioned above have slowed down the uniprocessor performance to about 20% per year. A lot of techniques were proposed to exploit better performance (e.g., larger caches, wider pipelines, multilevel speculation, etc.) but this started to result in *diminishing returns*, i.e., little improvement was obtained for the additional complexity in the design. As such, the entire industry shift to the Chip-Multiprocessor (CMP) or Multi-core Processor paradigm, i.e., the use of multiple processors per chip, rather than via faster uniprocessors, in order to achieve higher performance [1].

GEORGE MATHEOU

Multi-core Architectures

Multi-core architectures can be classified as homogeneous and heterogeneous. Examples of homogeneous multi-cores include: AMD's Phenom II X4, Intel's i5 and i7, AMD's Phenom II X6 and Intel's Xeon E7-2820. These processors equipped with four to eight cores per chip. Two representatives heterogeneous multi-core systems are the Cell microprocessor [71] and the Parallella system [72]. The Cell microprocessor consists of one PPE (Power Processing Element) core and eight SPEs (Synergistic Processing Elements) cores. The Parallella system is a high performance computing platform which consists of a dual-core ARM-A9 Zynq System-On-Chip and a 16-core Epiphany multi-core coprocessor.

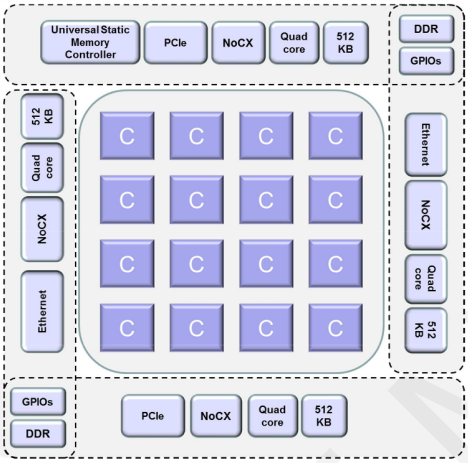


Figure 2: MPPA many-core architecture (C = Compute Cluster) [2].

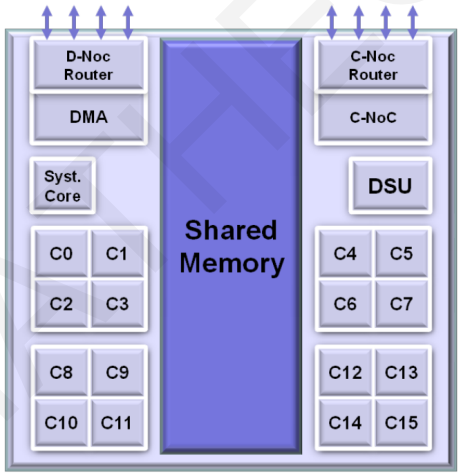


Figure 3: The architecture of the MPPA Compute Cluster [2].

Many-core Architectures

Multi-core systems improve power efficiency and performance by exploiting more parallelism at lower clock rates [73]. Many-core architectures extend this trend by combining many simple cores on a single processor chip. Usually, in many-core architectures, the cores are connected via an on-chip network that enables the communication and data exchange between the cores. Examples of many-core architectures are: the Intel's Single-chip Cloud Computer (SCC) experimental 48-core processor [74], the Kalray's MPPA-256 processor [75] which consists of 256 user cores and 32 system cores, and the Intel's Xeon Phi coprocessor [76] which consists of 61 cores. Figure 2 depicts the architecture of the MPPA-256 processor which contains 16 compute clusters. Each cluster contains 16 processing engine (PE) cores and a system core (Figure 3).

GEORGE MASON UNIVERSITY

2.3 Field Programmable Gate Arrays (FPGAs)

Field Programmable Gate Arrays (FPGAs) are semiconductor devices designed to be configured by a customer/designer after manufacturing [77]. FPGAs can be used to implement any logical function that an Application-Specific Integrated Circuit (ASIC) could perform. An ASIC is an integrated circuit (IC) customized for a particular use. An FPGA is based around a matrix of configurable logic blocks (CLBs) connected via programmable interconnects. The FPGA configuration is generally specified using a hardware description language (HDL), such as VHDL [78] or Verilog [79]. Compared to ASICs, FPGAs offer many design advantages including: rapid prototyping, shorter time to market, the ability to re-program and lower Non-recurring Engineering (NRE) costs.

The block diagram of a platform FPGA is depicted in Figure 4. A typical layout of the FPGA is an array of CLBs which implement combinational and sequential logic. The CLBs are sitting in a "sea" of interconnect wires. Interconnects between wires are programmed by turning on/off transistors at the wire junctions. Input/output from the FPGA is handled via I/O Blocks which themselves also contain sequential logic circuitry. Additionally, FPGAs contain Block RAMs, i.e., dedicated dual-port memories which contain several kilobits of RAM.

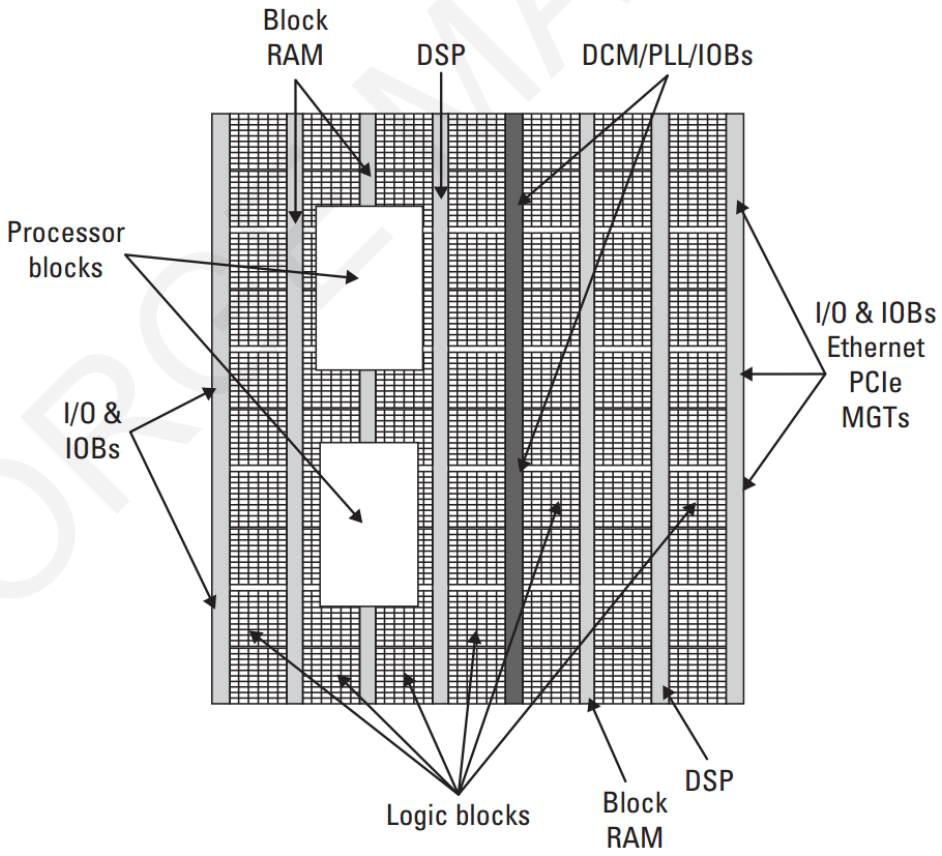


Figure 4: A high-level view of a platform FPGA [3].

GEORGE MATHEOU

The most basic element of an FPGA is the logic cell (LC) which contains a small lookup table (LUT), a D flip-flop [80] and a 2-to-1 mux. A K-input LUT [81] is a digital memory that can implement any boolean function of K variables. The K inputs are used to address a 2^K-by-1 memory that stores the truth table of the boolean function. A K-input LUT can also be configured as a 2^K-by-1 static RAM (SRAM) or as a 2^K-bit shift register. Several LCs along with special-purpose circuity (e.g., adder/subtractor carry chain) form a *slice*. Two or more slices are grouped to form a CLB. For example, in 7-series Xilinx FPGAs, 6-input LUTs are used where four LCs form a slice and two slices form a CLB. Additionally, FPGAs include special-purpose function blocks, such as: Digital Signal Processing (DSP) Blocks, embedded processors (e.g., IBM PowerPC) and Digital Clock Managers (DCMs) [3].

To conclude, FPGA is an interesting technology that is used in a lot of areas such as: Aerospace and Defence, ASIC Prototyping, Automotive, Consumer Electronics, High Performance Computing and Data Storage, Medical, Security and Image Processing. Recently, processor vendors, like Intel, are interested on integrating conventional processors along with FPGA devices onto the same chip [82]. It is believed that such hybrid CPU-FPGA architectures will help customers to drive performance while holding down power consumption.

2.4 The Data-flow model of execution

The data-flow model of execution was proposed by Jack Dennis [27, 68] in 1974 as an alternative to the control-flow model of execution. In this model instructions are scheduled dynamically at runtime based on data availability. An instruction becomes executable as soon as all of its input operands are available to it. A data-flow program is a directed graph consisting of nodes and arcs. The nodes represent the instructions of the program, while the arcs represent the data dependencies among instructions. During the execution of a data-flow program, data/values propagate along the arcs in the graph, in data packets, called tokens. This flow of tokens, enables some of the nodes/instructions and fires them.

Imperative programs rely on a program counter (PC) to control sequential execution without regard for data (i.e., instructions are executed in sequence). Contrary, in the data-flow paradigm data is the central controller of execution. In the execution model of a data-flow language [83, 84, 85, 86] each node within the graph is capable of executing whenever data is available on its input arcs. If several instructions become fireable at the same time, they can be executed in parallel. This principle provides the potential for massive parallel execution at the instruction level [87, 34]. Thus, data-flow computers allow fine-grain concurrency. Data-flow architectures can be classified as *pure* data-flow architectures (static, dynamic and explicit token store) and *hybrid* data-flow architectures [88].

GEORGE MATHEOU

2.4.1 Pure Data-flow Architectures

2.4.1.1 Static Data-flow Architectures

Static data-flow architectures [68] (also known as Single-Token-Per-Arc) allow at most one instance of a node/instruction to be enabled for firing. In particular, a data-flow node can be executed only when all of its input tokens are available and no tokens exist on any of its output arcs. This rule was applied in order to avoid difficulties and malfunctions when a graph is re-entrant (like a loop body) [89]. Static data-flow architectures implement this rule through Acknowledge Signals which are shown on the data-flow graph with additional arcs. A token on an Acknowledge arc indicates that the corresponding data arc is empty. A node is enabled when a token is present on each input arc and on each Acknowledge arc. When an instruction is fired it sends an Acknowledge signal indicating that it's ready to accept a new token (the Acknowledge Signals travel from the consuming to producing nodes).

Several static data-flow architectures have been proposed. Most notable examples are: the MIT Static Data-flow machine [68, 90], the Data-Driven Machine #1 (DDM1) [91], the Language Assignation Unique (LAU) system architecture [92] and the TI Distributed Data Processor [93]. The major advantage of static data-flow is its simplified mechanism for detecting enabled nodes. However, static data-flow has three main drawbacks:

- It is inefficient when dealing with iterative constructs and re-entrancy [89, 28]. Consecutive iterations of a loop can only be pipelined, thus only a limited amount of parallelism is exploited.
- The traffic is doubled due to the Acknowledge tokens.
- Essential programming constructs such as procedure calls and recursion are not supported.

2.4.1.2 Dynamic Data-flow Architectures

Static data-flow limits the performance because iterations are executed one at a time. As an alternative solution, the dynamic data-flow model was proposed. It allows simultaneous activation of several instances of a node at runtime. For instance, a loop body in a dynamic data-flow program can be represented as a single node. Multiple instances of the node representing the loop body can be created and executed concurrently at runtime. In dynamic data-flow architectures the arcs can be viewed as buffers containing multiple data-items. The different instances of a node are distinguished using tags. A tag is associated with each token that identifies the context in which a particular token was generated. An actor can be executed when all of its input tokens with identical tags are available. Dynamic data-flow architectures are also called *tagged-token* data-flow architectures. Three representative dynamic data-flow architectures are:

GEORGE MATHEOU

- The *Manchester Data-flow machine* [94, 29]. This project was concentrated on constructing a powerful processing element based on dynamic tagging. It has demonstrated reasonable performance, i.e., up to 1.2 MIPS (Million Instructions per Second).
- The *MIT Tagged-token Data-flow architecture* [95, 30], also known as the Tagged-Token Data-flow Architecture (TTDA). TTDA is an evolution of the U-Interpreter [64] in the direction of a realizable architecture. It consists of a number of identical processing elements (PEs) and storage units (I-Structure elements [96, 97]) interconnected by an n-cube packet network. The storage units are addressed uniformly in a global address space. A single PE and a single I-Structure element constitute a complete data-flow computer.
- *SIGMA-1* [98, 41]. It was designed to show the feasibility of a fine-grain data-flow computer to achieve highly parallel computation. SIGMA-1 is a large-scale computer consisting of 128 processing elements and 128 structure elements. Data-flow programs were developed in DFC (Data-flow C), a subset of the C language. SIGMA-1 demonstrated a performance of 170 MFLOPS which was about 39% of the theoretical peak performance of 427 MFLOPS.

Advantages and Limitations of Dynamic Data-flow Architectures

The major advantage of the dynamic data-flow model is its ability to provide better performance, compared with static, since it allows multiple tokens on each arc thereby unfolding more parallelism. However, the dynamic data-flow model has a number of shortcomings [89, 28, 88]:

- *Matching tokens incurs a lot of overheads:* Performance depends directly on the rate at which matching mechanism processes tokens. To facilitate matching while considering the cost and availability of a large associative memory, a pseudo-associative matching mechanism was proposed by Arvind. This mechanism requires several memory accesses which degrades the performance and the efficiency of the dynamic data-flow machines.
- *Complex resource allocation:* A failure in finding a match implicitly allocates memory within the matching unit. Mapping a code-block to a processor places an unspecified commitment on the processor's matching unit. This can result in a deadlock if this resource becomes overcommitted.
- *Inefficient data-flow instruction cycle:* A typical data-flow instruction cycle involves detecting enabled nodes, determining the operation to be performed, computing the results, generating result tokens and sending the result tokens to destination nodes. Also, the procedure of matching tokens is more complex than simply incrementing a program counter. As such, this model incurs more overheads compared to the control-flow model.

GEORGE MATHEOU

- *Handling data-structures is a complicated procedure:* Several schemes were proposed for handling data structures efficiently [97, 99, 100]. However, the problem of efficiently representing and manipulating data structures was a difficult challenge.

2.4.1.3 Explicit Token-Store Architectures

To overcome the inefficient matching of the dynamic data-flow model, explicit token store (ETS) proposed a direct matching [89]. ETS executes dynamic data-flow graphs directly and it was developed within the Monsoon [101, 43] data-flow processor. Later, the ETS principle was also applied in other data-flow machines like the Epsilon-2 multiprocessor [44].

Monsoon [101, 43] is a large scale data-flow multiprocessor which was based on the MIT Tagged-Token Data-flow architecture [95, 30]. It consists of several pipelined (8-stage) processing elements (PEs) and I-Structure Memory Interleaves (IS) which are connected via an Interprocessor Network. The ETS architecture eliminates the need for the associative memory by allocating a separate memory frame (called activation frame) for each activation of a loop or subprogram invocation. The idea is basically to allocate a frame of wait-match storage on each code-block invocation. The activation frame holds the synchronization information of instructions within the code block. This approach allows the wait-match store to be a fast and directly addressable memory. Access to locations within the activation frame is performed through offsets relative to a pointer to that frame, thus, there is no need for associative memory searching.

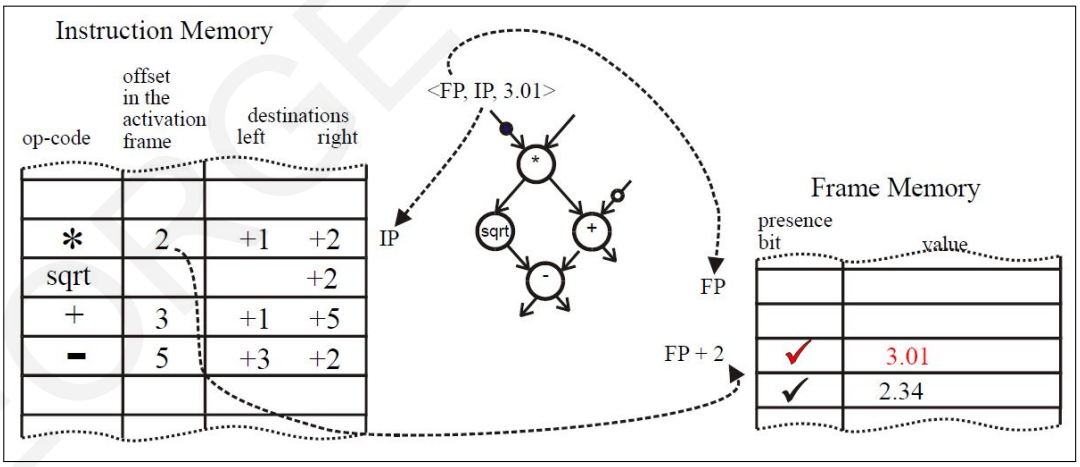


Figure 5: An example of the direct matching approach.

A token consists of a value, a pointer to the instruction to execute (IP) [destination instruction], a pointer to an activation frame (FP) and a destination port number. IP and FP attributes form the *tag*. The instruction fetched from location IP specifies an opcode (e.g. ADD), the offset in the activation frame where the match will take place, i.e., where its input tokens wait to rendezvous, (e.g. FP+3), and one or more destination instructions that will receive the result of the operation (e.g. instructions IP+1 and IP+2). When a token arrives, IP is used first to fetch the instruction from the Instruction

Memory. The offset r encoded in the instruction, together with FP, is used to interrogate exactly one location, FP + r, in the wait-match memory. If the slot is empty, this token is deposited there to wait. If the slot is full, the partner token is extracted, and the instruction is dispatched. Figure 5 illustrates an example of the direct matching approach, where the token <FP, IP, 3.01> is received in the root node of the graph. As a result, the value 3.01 will be stored in the address FP+2 of the Frame Memory.

2.4.1.4 Pure Data-flow Limitations

Pure data-flow architectures, despite the proposed Explicit Token-Store (ETS) architectures, do not perform very well with sequential code, compared to conventional control-flow architectures. The reasons are the following:

- Inability to efficiently handle complex data structures (e.g., arrays) [102]. Despite the fact that data-flow enables simple arithmetic values to move between instructions easily, problems arise when structured data was to be passed [103]. According to the data-flow semantics, when dealing with structured data types, the entire data structure must be carried in a token. This is because, modification to even one element of an array results in a new data value. Also, if a token contains a pointer to the structure, then the elements of the structure should not be modified (due to *single assignment semantics*).
- Excessive overheads due to the per-instruction token matching and fine-grained context switching at the level of each instruction.
- Inefficient use of the pipeline. An instruction of the same thread can only be issued to the data-flow pipeline after the completion of its predecessor instruction.

2.4.2 Hybrid Data-flow Architectures

The hybrid architectures were proposed to address the limitations of the pure data-flow architectures by combining data-flow with control-flow mechanisms/techniques [28, 104, 105]. In hybrid architectures, a node in a data-flow 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. Hybrid architectures can be classified as *threaded data-flow*, *coarse-grain data-flow* or *RISC data-flow* [106, 107]. Several hybrid architectures were proposed. Most notable examples are: the MIT Hybrid machine [108], the EM-4 architecture [109, 110], the Epsilon-2 multiprocessor [44], the USC Decoupled Graph/Computation (DGC) architecture [111, 112, 113], the McGill Dataflow Architecture (MDFA) architecture [114] and the P-RISC (Parallel-Reduced Instruction Set Computer) [115].

GEORGE MATHEOU

2.4.2.1 Threaded Data-flow Architectures

Threaded data-flow architectures [108, 109, 110, 44] used fine-grained data-flow graphs where each node is a single machine-level instruction. Graphs were analyzed in order to identify sub-graphs that exhibit low levels of parallelism that should always execute in sequence. Each sub-graph is identified and transformed into a sequential thread of instructions. Such a thread is issued consecutively by the matching unit without matching further tokens, except for the first instruction of the thread. Data passed between instructions in the same thread is stored in registers instead of being written back to memory. This approach improves single-thread performance since the total number of tokens needed to schedule program instructions is reduced. As a result, this approach, saves time and resources.

2.4.2.2 Coarse-Grain Data-flow Architectures

In coarse-grain data-flow architectures [111, 112, 113, 114], data-flow graphs are analyzed and divided into sub-graphs, similarly to the threaded approach. Sub-graphs are compiled into sequential von Neumann processes which they are referred as coarse-grained nodes or macroactors [28]. A data-flow graph is executed using the traditional data-flow rules where each macroactor contains an entire function, or part of a function. The macroactors can be programmed in an imperative language, such as C/C++ or Java. Since macroactors are executed according to the data-flow rules this approach retains the advantages of data-flow while the overheads of the fine-grained data-flow are eliminated.

Coarse-grain data-flow architectures [111, 112, 114] decouple the token matching stage from the execution stage using FIFO buffers. Thus, pipeline bubbles can be avoided by the decoupling. Also, off-the-shelf microprocessors can be used to support the execution stage. One of the earliest architectures that utilize the decoupling principle is the USC Decoupled Graph/Computation (DGC) architecture [111, 112, 113]. The decoupled architecture consists of two basic units, the *graph unit* and the *execution unit*, that operate in an asynchronous manner. The graph unit executed all graph operations, i.e., it was responsible for updating the data-flow graph and determining whether a graph node can be scheduled for execution. The computation unit was responsible for the execution of the instructions of the graph's nodes.

2.4.2.3 RISC Data-flow Architectures

RISC data-flow architectures (e.g. P-RISC [115]) support the execution of existing software written for conventional processors [107, 105]. A RISC data-flow architecture uses a RISC-like instruction set, supports multithreaded computation, provides fork and join instructions in order to manage multiple threads, implements all global storage as I-structure storage and implements the

GEORGE MATHEOU

load/store instructions to execute in a split-phase mode. RISC-like hybrid multithreaded processors represent the closest architecture type to the von Neumann machines.

2.4.2.4 Non-blocking Multithreaded Architectures

Architectures that have evolved from the coarse-grain data-flow 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. Three representative examples of non-blocking multithreaded architectures are: StarT (or *T) [116], EARTH (Efficient Architecture of Running Threads) [52, 117, 118] and TAM (Threaded Abstract Machine) [119, 120].

2.5 Recent Data-flow Developments

2.5.1 Software Implementations

2.5.1.1 Cilk and Cilk Plus

Cilk is a parallel programming extension to the C language that provides keywords (*cilk*, *spawn*, *sync*, etc.) for facilitating parallelism [121]. When the Cilk keywords are removed from source code, the result is a valid C program, called the *serial elision*. Cilk uses a fork-join paradigm on top of the existing threading model. The keywords are used to spawn functions as asynchronous parallel tasks and synchronize amongst the tasks using a barrier-like *join* method. Cilk programs are preprocessed to C and then compiled and linked to a runtime library. The runtime manages the scheduling of tasks using a work-stealing scheduling policy. The fork-join approach adopted by Cilk is well-suited for expressing recursive algorithms (e.g., divide-and-conquer algorithms). At runtime, a Cilk program can be viewed as a directed acyclic graph (DAG) that unfolds dynamically as the program executes. A Cilk program consists of a collection of *Cilk procedures*, each of which is broken into a sequence of *threads*, which form the vertices of the DAG. Each thread is a non-blocking C function. A thread from the Cilk procedure can spawn a child thread which begins a new child procedure. Return values and other values sent from one thread to another, induce data dependencies among the threads, where a thread receiving a value cannot begin until another thread sends the value.

Cilk Plus [122, 123] is a language extension that provides both task and data parallelism constructs. It's a commercial implementation of Cilk provided by Intel and supports both C and C++. Users can use three simple keywords to express task parallelism: *cilk_for*, *cilk_spawn* and *cilk_sync*. Cilk Plus also provides array notations for expressing data parallelism, reducers for eliminating contention for shared data and SIMD-enabled functions.

GEORGE MATHEOU

2.5.1.2 Gupta and Sohi's Data-flow Approach

Gupta and Sohi [37] have also proposed a software system that implements data-flow/data-driven execution of sequential imperative programs on multi-core systems. In particular, they have implemented a C++ runtime library that exploits *Functional-Level Parallelism* (FPL) by executing functions on the cores in a data-flow fashion. The system employs multithreading to implement mechanisms of parallel execution (using PThreads) where a randomized task-stealing policy is used, similar to Cilk-5, to balance the load in the system's cores. The runtime determines data dependences between computations/functions dynamically and executes them concurrently in a data-flow fashion. Each function has a *data set* which includes its input (read set) and output (write set) operands. An operand is actually an *object*. Since objects in the data set may be unknown statically, the data set is evaluated dynamically, at runtime, before the function is invoked. As such, an *object-based* data-flow graph is used. The identity of the objects in the data set is used to establish the data dependence between functions. The runtime determines if the function currently being processed is dependent on any prior function(s) that are still executing. If not, it is submitted (or *delegated*) to a core for execution. If so, it is *shelved* until its dependences are resolved. In either case, the runtime then proceed to process the next function in the sequential program.

2.5.1.3 SWift Adaptive Runtime Machine (SWARM)

SWARM [38] is a software runtime that uses an execution model based on codelets [22]. The codelets model was based on the EARTH project [52]. Codelet is a collection of instructions that can be scheduled "atomically" as a unit of computation which is run until completion. SWARM divides a program into tasks with runtime dependencies and constraints that can be executed when all runtime dependencies and constraints are met. The runtime schedules the tasks for execution based on resource availability. SWARM utilizes a work-stealing approach for on-demand load-balancing. Furthermore, it allows data-flow execution on single-node and multi-node (clusters) systems. In the latter case, data and work must be directed to specific nodes, i.e., they cannot be distributed automatically under runtime control. There are two options for writing SWARM programs, using a C API or SCALE (SWARM Codelet Association Language Extensions). SCALE is a set of extensions to C that provide a simpler means of declaring and interacting with codelets.

2.5.1.4 StarSs

Star Superscalar (StarSs) is a parallel programming platform that targets a variety of architectures: the Cell processor (CellSs [124, 125]), multi-cores and Symmetric Multiprocessors (SMPSs [39]) and GPUs (GPUSs [126]). StarSs targets automatic function level parallelism. Parallelism is achieved through hints given by programmer in the form of *pragmas* that identify atomic parts of the code that

GEORGE MATHEOU

operate over a set of parameters. These parts of the code are encapsulated in the form of functions (called tasks). With these hints StarSs builds a parallel application that detects the task calls and their inter-dependencies. A task-graph is dynamically generated, scheduled and run in parallel using a different number of threads.

StarSs has two major components, a source-to-source compiler and a runtime system. The compiler translates C code with pragmas into standard C99 code with calls to the supporting runtime library. The runtime takes the memory address, size and directionality (*input*, *output* or *inout*) of each parameter of each task invocation and uses them to analyze the dependencies between them. The runtime schedules the tasks to the different cores when their input dependencies are satisfied. Additionally, the runtime is capable of *renaming* the data, leaving only the true dependencies (this technique is also used in superscalar processors and compilers). StarSs exploits data locality by scheduling dependent tasks sequentially to the same core so that output data is reused immediately. Another feature of StarSs is the task hierarchy, i.e., the instantiation of tasks within tasks [127]. In this case, a given task waits for the end of its children tasks before finishing. This feature can be used to execute parent and children tasks on different architectures (e.g., parent-tasks are executed with SMPSs and children-tasks with CellSs).

2.5.1.5 OmpSs

OmpSs [40, 128] is a programming model that provides the features of the StarSs using OpenMP directives. This framework allows to express data-dependencies between tasks using the *in, out* and *inout* clauses. The Nanos++ runtime system is used to support task parallelism using synchronizations based on data-dependencies. Also, the Mercurium source-to-source compiler is used which recognizes constructs and transforms them to calls to the runtime system. Additionally, OmpSs includes CUDA and OpenCL kernels in order to provide a single programming environment that covers the different homogeneous and heterogeneous architectures. OmpSs was evaluated in different architectures: SMPS, GPUs and hybrid SMP/GPU environments.

While OpenMP has a fork-join model, OmpSs defines a thread-pool model where all the threads exist from the beginning of the execution. One of these threads, the master thread, starts executing the user code while the other threads remain ready to execute work when available. OmpSs provides an extended set of constructs that allow users to specify data dependencies and target devices. Finally, OmpSs was extended to support asynchronous task parallelism on clusters of heterogeneous architectures [129, 130]. The distributed OmpSs implementation (called OmpSs@Cluster) uses the annotation-based programming model to move data across a disjoint address space.

GEORGE MATHEOU

2.5.1.6 Thread Building Blocks (TBB)

TBB is an API developed by Intel that relies on C++ templates to facilitate parallel programming [131]. It provides a set of data structures and algorithmic skeletons that supports the execution of tasks (*parallel_for*, *parallel_reduce*, etc.). TBB provides also support for dependency and data-flow graphs. Moreover, a set of concurrent containers (hash-maps, queues, etc.) and synchronization constructs (mutex constructs, atomic operations, etc.) is provided. The TBB runtime implements a tasks-stealing scheduling policy and adopts a fork-join approach for the creation and management of tasks, similarly to the Cilk approach [121].

2.5.2 Hardware Implementations

2.5.2.1 Scheduled Data-flow (SDF)

SDF [53, 132] is a multithreaded architecture that decouples the synchronization from the computation of non-blocking threads. It uses data-flow-like synchronization at the thread-level and control-flow semantics within a thread. A thread is enabled for execution when it has received all its inputs. SDF allocates a register set and a *frame* for each enabled thread. Data is pre-loaded into the register set prior the scheduling of the thread on the execution pipeline. All results are post-stored after the completion of the thread's execution (from the thread's registers into memory). All data needed for the thread, including a Synchronization Count (SC), is stored in the frame. SC indicates the number of inputs needed for a thread before it can be scheduled for execution. When data is stored for a thread, the SC is decremented, and once it reaches zero, the thread is ready to execute. In that case, the pre-load code of the thread moves the data from the frame into the register set of the thread. The post-store code of a thread stores data from the thread's registers into the frames of its awaiting consumers.

A processor in SDF consists of two pipelines: the execution pipeline and the synchronization pipeline. The execution pipeline (4-stages) is responsible for executing threads. It behaves more like a conventional pipeline (e.g., MIPS) while retaining the primary data-flow properties (single assignment and flow of data from instruction to instruction). This eliminates the need for complex hardware for: (i) detecting write-after-read (WAR) and write-after-write (WAW) dependencies and (ii) register renaming. Also, unnecessary thread context switches on cache misses are eliminated. The synchronization pipeline (6-stages) handles the pre-load and post-load instructions, i.e., it performs all memory accesses. A separate Synchronization Unit (SU) is provided which is responsible for scheduling the non-blocking threads and for allocating memory frames, synchronization counters and register sets.

SDF faces a major issue that limits the usability and scalability of the architecture [56, 133]. Each pipeline must be able to communicate with local memory, register file and control logic in one cycle.

GEORGE MATHEOU

This is a reasonable assumption as long as the architecture has few pipelines in a processor. However, the growth of the number of pipelines is a limiting factor of the SDF scalability since it is difficult to achieve the communication between components in one cycle.

2.5.2.2 Decoupled Threaded Architecture - Clustered (DTA-C)

DTA-C (Decoupled Threaded Architecture - Clustered) [56] is an SDF architecture which tries to improve the scalability by clusterizing resources and balancing workload among the processing cores. All clusters in the architecture share the same structure and can be considered high-level tiles. This cluster property requires the use of a fast interconnection network inside the cluster (intra-cluster network) and the use of a slower but more complex network for connecting all clusters (inter-cluster network). Internally, each cluster consists of one or more processing elements (PEs) and a Distributed Scheduler Element (DSE). The set of all DSEs constitutes the Distributed Scheduler (DS) which is responsible for assigning threads at runtime. Each PE contains pipelines, a frame memory (small on-chip memory), a register file and a Local Scheduler (LS). LS is responsible for communicating with other processors and clusters (serves the requests for new resources and for data communication). DTA-C implements a two-level scheduling, handled by the LS inside each PE and by the DSE inside each cluster. The scheduling mechanism is responsible for assigning frames to the threads and for balancing the load in the system.

2.5.2.3 Explicit Data Graph Execution (EDGE)

Explicit Data Graph Execution (EDGE) [134] proposes a new instruction set architecture (ISA) that supports *direct instruction communication*, i.e., the hardware is responsible for delivering a producer instruction's output directly as an input to a consumer instruction, rather than writing it back to a shared namespace, such as a register file. Using this direct communication from producers to consumers, instructions are executed in a data-flow order (each instruction is fired when its inputs are available). An EDGE ISA provides a richer interface between the compiler and the micro-architecture. Specifically, the ISA directly expresses the data-flow graph that the compiler generates internally. This approach exposes a higher degree of concurrency and achieves a more power-efficient execution since complex hardware is not required to discover data-dependencies dynamically at runtime. An EDGE program is partitioned by the compiler into *hyperblocks* comprising a large number of instructions. A major difference between EDGE and other RISC/CISC architectures is that in EDGE, the instructions specify only their targets or consumers (instead of specifying their source operands). Each hyperblock is executed atomically in parallel on an array of functional units (ALUs).

GEORGE MATHEOU

TRIPS: An EDGE Architecture

TRIPS (Tera-op, Reliable, Intelligently adaptive Processing System) [45, 134, 59] is an instance of the EDGE architecture. It combines control-flow execution across hyperblocks of code consisting of up to 128 instructions with a data-flow execution inside the blocks. A hyperblock is equivalent to the block granularity. TRIPS uses the *block-atomic execution*, i.e., each hyperblock is fetched, executed, and committed atomically, similar to the conventional notion of transactions. This approach allows TRIPS to support conventional languages such as C, C++, or Fortran. The TRIPS micro-architecture behaves like a conventional processor with sequential semantics at the block level (each block behaves as a "megainstruction"). Inside the executing blocks, a fine-grained data-flow model is used, based on the direct instruction communication, to execute the instructions quickly and efficiently.

The TRIPS prototype contains two processing cores, each of which is a 16-wide out-of-order issue processor that can support up to 1024 instructions in flight. Each processor core consists of 16 execution nodes (a 4x4 array) connected by a lightweight network. An execution node contains a fully functional ALU and 64 instruction buffers. The compiler builds 128-instruction blocks, organized into groups of eight instructions per node at each of the 16 execution nodes. The locations of all instructions are statically determined by the compiler. However, the processor fires each instruction dynamically. When a block is fetched and mapped, the processor fetches the instructions in parallel and loads them into the instruction buffers at each ALU in the array.

TRIPS exposes more instruction-level parallelism by allowing up to eight blocks executing concurrently. A scheduler is responsible for: (i) placing independent instructions on different ALUs to increase concurrency, thereby reducing the probability of two instructions competing to issue on the same ALU in the same cycle and (ii) placing instructions near one another to minimize routing distances and thus communication delays. To conclude, TRIPS can achieve power-efficient out-of-order execution across an extremely large instruction window (1024 instructions - 8x128-instruction blocks) because it eliminates many of the power-hungry structures found in traditional RISC implementations. However, the overall performance depends on the compiler.

2.5.2.4 WaveScalar

WaveScalar [54, 60] is a tiled architecture, similar to EDGE [134], which provides a data-flow ISA and execution model targeting scalable low-complexity/high-performance processors. It consists of a large number of processing elements (PEs) surrounded by intelligent cache banks that hold the current working set of instructions. Instructions are executed in-place and explicitly communicate with its dependent instructions in a data-flow fashion (i.e., send their results to their dependent

GEORGE MATHEOU

instructions). WaveScalar can execute programs written with conventional von Neumann-style memory semantics (like C/C++) by using the *wave-ordered memory* technique which guarantees correct memory ordering.

The WaveScalar compiler breaks the control-flow graph of a program into single-entrance directed acyclic blocks of instructions, called *waves*. An example of a wave is a loop iteration. Each wave is tagged via a distributed tagging mechanism using special instructions to distinguish between different dynamic instances of a wave. WaveScalar loads instructions from memory and dynamically binds them to PEs as an application executes, swapping them in and out on demand. Instructions remain in the cache over many invocations. This enables dynamic optimization of an instruction's physical placement in relation to its dependents. Additionally, a highly tuned placement algorithm was implemented that uses depth-first traversal of the data-flow graph to build chains of dependent instructions that execute sequentially at one PE. It then assigns those chains to PEs on demand as the program executes.

The authors in [135] investigate the area/performance trade-offs of a tiled data-flow architecture, that of the WaveScalar processor. A synthesizable RTL model and a cycle-level simulator were used. The WaveScalar's area efficiency is compared to that of an aggressive out-of-order superscalar processor and to that of a Sun's Niagara chip multiprocessor. The main conclusion of this work is that the data-flow nature of WaveScalar provides substantially more performance per unit area and better area scaling compared to the other two systems.

2.5.2.5 Task Superscalar

Task Superscalar [4] is a hybrid data-flow/von-Neumann architecture that implements the StarSs programming model. It combines data-flow execution of tasks with control-flow execution within the tasks. Particularly, the Task Superscalar pipeline is an abstraction of an out-of-order superscalar pipeline that operates at the task-level. As ILP pipelines uncover parallelism in a sequential instruction stream, similarly, the Task Superscalar uncovers task level parallelism among tasks. The StarSs programming model enables programmers to explicitly expose task side-effects by annotating the operands of a task as *input*, *output* or *inout*. With these annotations, the Task Superscalar pipeline dynamically detects inter-task dependencies, constructs the data dependency graph at runtime and dynamically schedules tasks for execution (in out-of-order manner).

The high-level operational flow of Task Superscalar is illustrated in Figure 6. A task-generating thread sends tasks (non-speculative) to the pipeline *frontend* for dependency decoding. The pipeline frontend maintains a window of recently generated tasks, for which it generates the data dependency graph (with tasks as nodes and dependencies between tasks as arcs), and uncovers task-level parallelism. The task window may consist of tens of thousands of tasks which enables it to uncover large amounts of parallelism. Furthermore, the pipeline increases the available parallelism by renaming

GEORGE MATHEOU

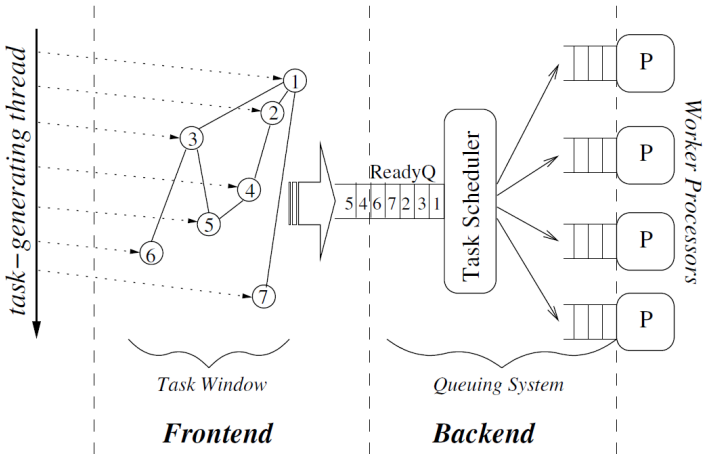


Figure 6: High-level view of Task Superscalar (retrieved from [4]).

memory objects, thus breaking anti- and output dependencies. Ready tasks are sent to the execution *backend*, which consists of a task scheduler, a queuing system, and a many-core fabric. The backend can also function as a regular chip multiprocessor (CMP). Finally, Task Superscalar was evaluated using TaskSim, a trace-driven cycle-accurate CMP simulator. Recently, it was simulated (using the ModelSim simulator) and synthesized in a FPGA device [46].

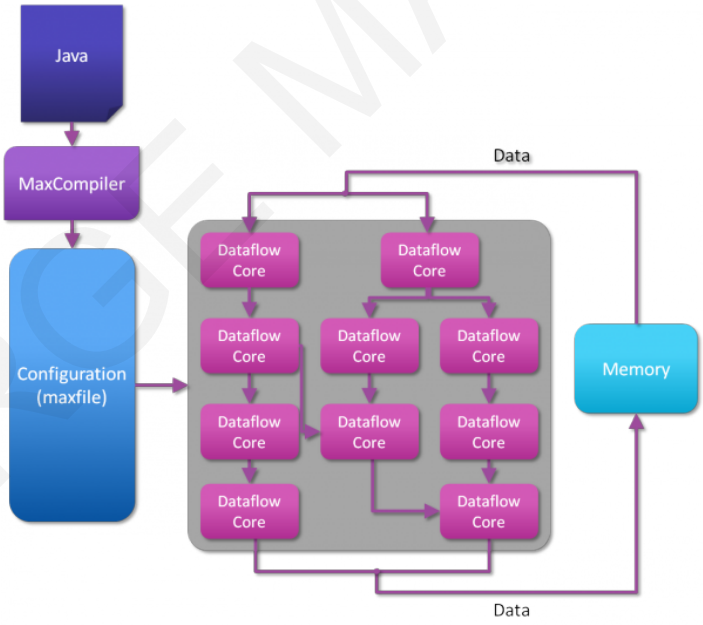


Figure 7: Computing with Maxeler's implementation of streaming data-flow cores [5].

2.5.2.6 Maxeler Streaming Data-flow Machines

Maxeler Technologies [5, 47] is a company specializing in data-flow solutions on FPGAs. A Maxeler streaming Data-flow machine exploits the data-flow computing. Maxeler's Data-flow computers focus on optimizing the movement of data in an application and utilizing massive parallelism between thousands of tiny "Data-flow cores" to provide order of magnitude benefits in performance, space and

GEORGE MATHEOU

power consumption. Maxeler Streaming Data-Flow machines consist of Data-flow engines which handle the bulk part of computation (as a coprocessor) and traditional control-flow CPUs which are responsible to run OS, main application code, etc. In a data-flow application, the program source is transformed into a data-flow engine configuration file as shown in Figure 7. The configuration file describes the operations, layout and connections of a streaming data-flow engine. Data can be streamed from memory into the chip where operations are performed. Inside the data-flow engine/chip, data is forwarded directly from one computational unit to another, without being written to the off-chip memory.

2.6 Data-Driven Multithreading (DDM)

The Data-Driven Multithreading (DDM) [6, 55, 136] model of execution was inspired by the Decoupled Data Driven (D^3) model [111, 137]. It is a non-blocking multithreading model that schedules threads based on data availability on sequential processors. Scheduling based on data availability can effectively tolerate synchronization and memory latencies [55, 11]. A DDM program consists of several threads of instructions (called *DThreads*) that have producer-consumer relationships. The instructions within the DThreads are fetched and executed by the CPU sequentially in a control-flow manner. This allows the exploitation of a plethora of control-flow optimizations, either by the CPU at runtime or statically by the compiler (pipelining, branch prediction, out-of-order execution, etc.).

The core of the DDM model is the *Thread Scheduling Unit (TSU)* [58] which is responsible for scheduling DThread instances at runtime, based on data availability. For each DThread, the TSU collects meta-data (called *Thread Templates*) that enable the management of the dependencies among DThreads and determine when a DThread instance can be scheduled for execution. In particular, TSU schedules a DThread instance for execution when all its producer-instances have completed their execution. This ensures that all the data that this DThread instance needs are available.

The DDM model has three basic implementations: the Data-Driven Network of Workstations (D^2NOW) [138, 139, 55, 6, 136], the Thread Flux platform (TFlux) [9, 8] and the Data-Driven Multithreading Virtual Machine (DDM-VM) [10, 11, 35, 63].

2.6.1 Context and Nesting attributes

2.6.1.1 Context attribute

DDM allows multiple *instances* of the same DThread to co-exist in the system, using the Context attribute. The Context attribute is a 32-bit value that enables multiple instances of the same DThread to co-exist in the system and run in parallel. This is essential for programming constructs such as loops and recursion. This idea was based on the U-Interpreter's tagging system [64] which provides a formal distributed mechanism for the generation and management of the tags at execution time.

GEORGE MATHEOU

U-Interpreter was used in dynamic data-flow architectures to allow loop iterations and sub-program invocations to proceed in parallel via the tagging of data tokens [140]. Figure 8 depicts a simple example of using multiple instances of the same DThread through the Context attribute. The for-loop shown on the top of the figure is fully parallel thus it can be executed by only one DThread. Each instance of the DThread is identified by the Context and it executes the inner command of the for-loop. The for-loop is executed 64 times, thus 64 instances are created with Contexts from 0 to 63.

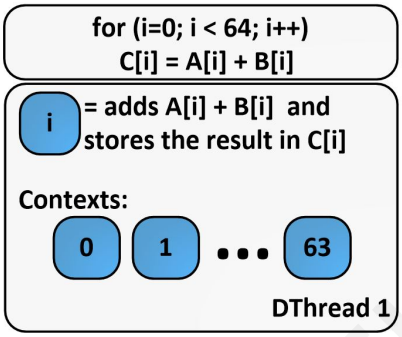


Figure 8: Example of using multiple instances of the same DThread.

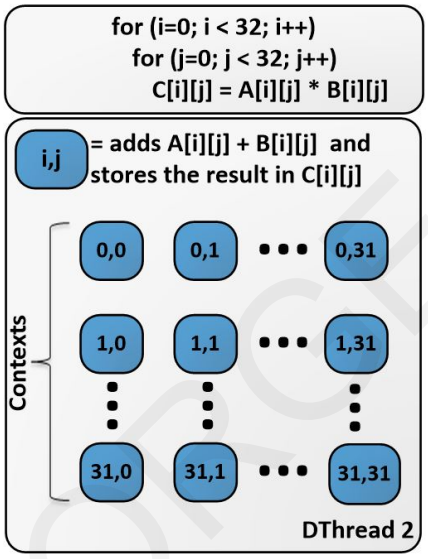


Figure 9: Example of a DThread that parallelizes a two-level nested loop.

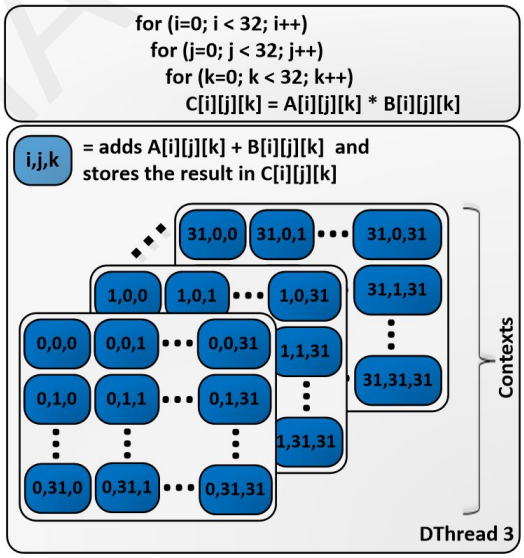


Figure 10: Example of a DThread that parallelizes a three-level nested loop.

2.6.1.2 Nesting attribute

The DDM model allows the parallelization of nested loops that can be mapped into a single DThread by using the Nesting attribute. This attribute is a small number that indicates the loop nesting level for the DThreads that implement loops. In the latest DDM implementations [35, 10, 141, 142] three nesting levels are supported, i.e., the DThreads are able to implement one-level (Nesting-1), two-level (Nesting-2) or three-level (Nesting-3) nested loops. If a DThread does not implement a

loop, its Nesting attribute is set to zero (Nesting-0). The Nesting attribute is used in combination with the Context. The indexes of the loops are encoded into a 32-bit Context value. The TSU uses the Nesting attribute to manage the Context value properly. An example of an one-level loop is depicted in Figure 8 where a Context value holds an index of the loop. An example of a two-level nested loop is shown in Figure 9. Each instance of the DThread will execute the basic block of the nested loops. A Context value in this case will include the indexes of the inner (in the lower 16-bits) and the outer (in the upper 16-bits) loops. Similarly, an example of a three-level nested loop is shown in Figure 10. An index of the inner loop is stored in the lower 12-bits of a Context value, an index of the middle loop is stored in the bits 12-21 and an index of the outer loop is stored in the upper 10-bits.

2.6.2 Thread Template

A Thread Template contains the following information:

- **Thread ID (TID):** an identification number that identifies uniquely each DThread.
- **Instruction Frame Pointer (IFP):** a pointer to the address of the DThread's first instruction.
- **Ready Count (RC):** a value that is equal to the number of the producer-threads of a DThread.
- **Nesting:** the Nesting attribute.
- **Scheduling Policy:** the method that is used by the TSU to map the ready DThread instances to the cores.
- **Consumer Threads:** a list of consumers that is used to determine which RC values will be decreased, after a DThread instance completes its execution.
- **Data Frame Pointer (DFP):** a pointer to the data frame assigned for a DThread.

2.6.3 DDM Dependency Graph

The DDM dependency graph is a directed graph in which the nodes represent the DThread instances and the arcs represent the data-dependencies amongst them. Each instance of a DThread is paired with a special value called *Ready Count (RC)* that represents the number of its producers. An example of a dependency graph is shown in Figure 11 which consists of four DThreads (T1-T4). Notice that the number inside each node indicates its Context value. T1 is a SimpleDThread, i.e., it has only one instance. T2 and T3 are MultipleDThreads. A MultipleDThread manages a DThread with multiple instances and it can be used to parallelize one-level loops or similar constructs. T4 is a MultipleDThread2D, a special type of MultipleDThread, where the Context values consist of two parts, outer and inner. A MultipleDThread2D can be used to parallelize a two-level nested loop where each Context value will hold an index of the outer and the inner loop. In this example T4 consists of 64 instances (with Contexts from <0,0> to <7,7>). Similarly, a user can parallelize a three-level

GEORGE MATHEOU

nested loop by using a MultipleDThread3D where its Context values consist of three parts: outer, middle and inner.

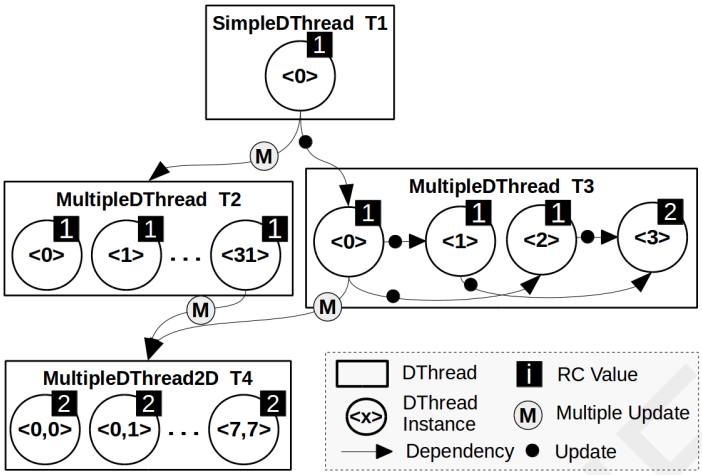


Figure 11: Example of a DDM Dependency Graph.

The RC values are depicted as shaded values next to the nodes. For example, the instance <3> of T3 has RC=2 because it has two producers, the instances <1> and <2> of T3. All instances of T2 have RC=1 because they are waiting for only one producer, the instance <0> of T1. The RC value is initiated statically and is dynamically decremented by the TSU each time a producer completes its execution. A DThread's instance is deemed executable when its RC value reaches zero. In DDM, the operation used for decreasing the RC value is called *Update*. Update operations can be considered as tokens that are moving from the producer to consumer instances through the arcs of the graph. *Multiple Updates* [11] are introduced in order to decrease multiple RC values of a DThread at the same time. This reduces the number of tokens in a DDM graph. For instance, DThread T1 sends a Multiple Update command to DThread T2 in order to spawn all its instances, instead of sending 32 single Updates. A DThread instance can send single and multiple Updates to any other instance of any type, including itself (the only requirement is to have data dependencies). Thus, it is possible to build very complex dependency graphs.

2.6.4 DDM Implementations

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

The first implementation of DDM was the Data-Driven Network of Workstations (D^2NOW) [138, 139, 55, 6, 136]. D^2NOW is a simulated cluster of distributed machines/workstations augmented with a hardware Thread Scheduling Unit (TSU). TSU was responsible for handling all the synchronization and communication operations. D^2NOW was evaluated using an execution driven simulator based on native execution. The simulator was running directly on the target processor. The execution of application threads was interleaved with the simulation of the TSU and the interconnection network.

GEORGE MATHEOU

All simulations were carried out on an 800-MHz Intel Pentium III workstation with 256-MB RAM. Finally, the D^2NOW implementation was simulated with up to 32 nodes.

D^2NOW Architecture

The architecture of D^2NOW is depicted in Figure 12. Each node/workstation has a separate TSU. The TSU is an add-on card which is attached to the workstation's motherboard using a CPU slot. For the communication of the TSUs, D^2NOW uses a dedicated communication network.

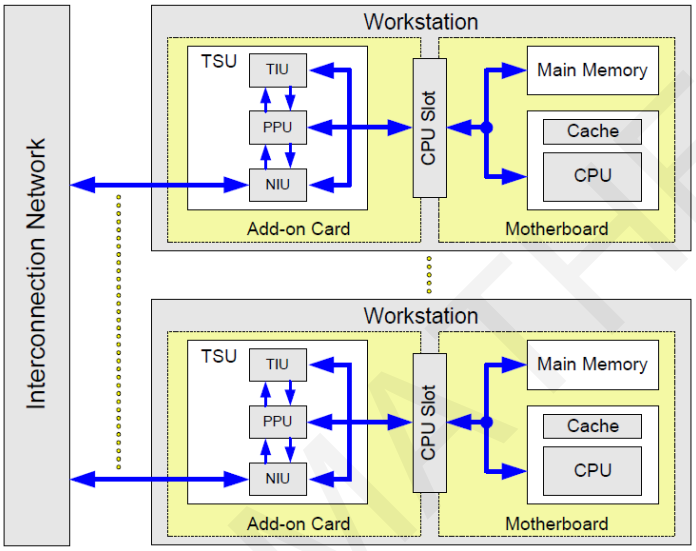


Figure 12: The D^2NOW architecture [6].

Figure 13 illustrates the block diagram of a D^2NOW's node where the TSU is connected with a processing element. The processing element communicates with the TSU via the Ready Queue (RQ) and the Acknowledgment Queue (AQ). The RQ contains information of the DThread instances that are ready for execution (e.g., TID and Context). The AQ contains information about executed DThread instances. The processing element reads the information of the next ready DThread instance from the RQ and executes it. After the processing element completes the execution of a DThread instance, it stores its information into the AQ. The Graph Memory (GM) holds the Thread Templates while the Synchronization Memory (SM) holds the Ready Count values for each DThread instance/invocation. The main functionality of the TSU's control unit is as follows:

1. Fetches the completed DThread instances from AQ.
2. Finds the consumers of the completed DThread instances from GM.
3. Updates the Ready Count of the corresponding consumer-instances in SM.
4. Stores the consumer-instances that are ready for execution into RQ.

GEORGE MATHEOU

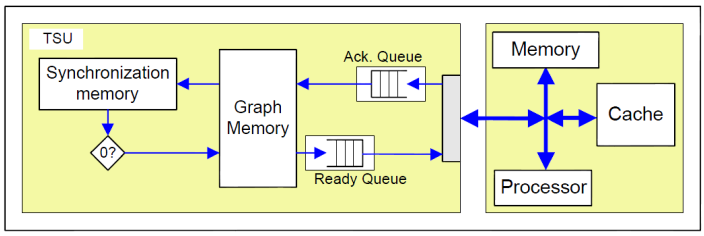


Figure 13: A DDM Node [6].

TSU Architecture

The purpose of the TSU is to provide hardware support for data-driven thread synchronization on conventional microprocessors. The TSU is a memory mapped device and it consists of three units: the Network Interface Unit (NIU), the Post Processing Unit (PPU) and the Thread Issue Unit (TIU). The NIU is responsible for the communication between the TSU and the Interconnection Network. The PPU is responsible for updating the Ready Count (RC) of the consumer threads. If a thread is ready for execution, the PPU forwards it to the TIU which schedules the threads for execution. Figure 14 illustrates the TSU's internal structure as well as the interface to the motherboard.

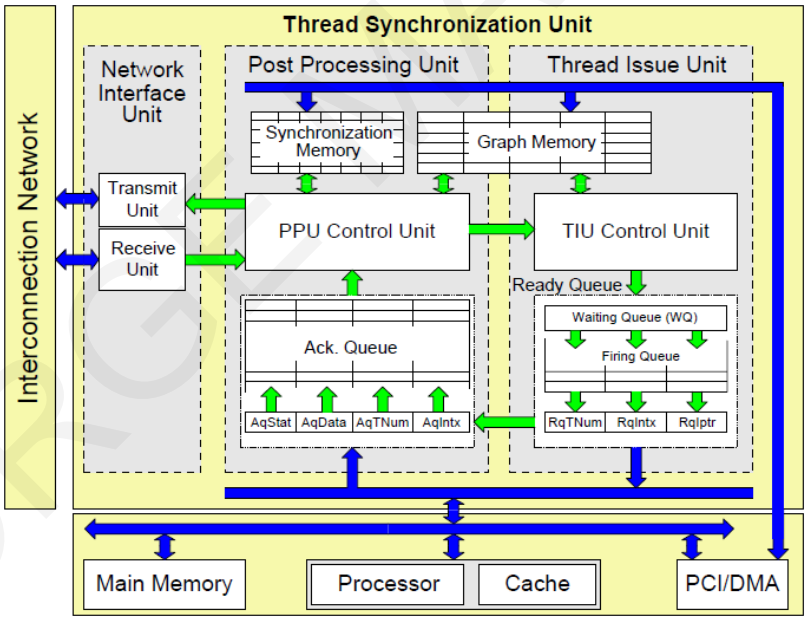


Figure 14: The TSU's internal structure [6].

CacheFlow Policy

Although DDM exploits the benefits of the non-blocking multithreading model, scheduling based on data availability may have a negative effect on locality. The data-driven scheduling can lead to irregular memory access patterns which can affect negatively the cache effectiveness. This is because temporal and spatial locality are not taking into account. To address these issues, the CacheFlow

policy [143] was proposed, which implements data prefetching using data-driven caching policies. The CacheFlow policy ensures that the data of a thread are preloaded in the cache before the thread is fired for execution. Also, it ensures that data preloaded in the cache are not evicted before the corresponding thread is executed, thus, reducing possible cache conflicts. Three implementations of the CacheFlow policy were provided:

1. **Basic Prefetch CacheFlow:** the data of the threads, that will be scheduled for execution in the near future, are prefetched into the cache. When the prefetching for a thread finishes, it is placed in the Firing Queue where it waits its turn to be executed. The structure of the TIU that supports the Basic Prefetch policy is shown in Figure 15.
2. **CacheFlow with Conflict Avoidance:** the prefetched data belonging to threads waiting in the Firing Queue are protected from eviction until the corresponding threads are executed.
3. **CacheFlow with Thread Reordering:** the sequence of executable threads is re-ordered before they enter the Firing Queue, to take advantage of locality.

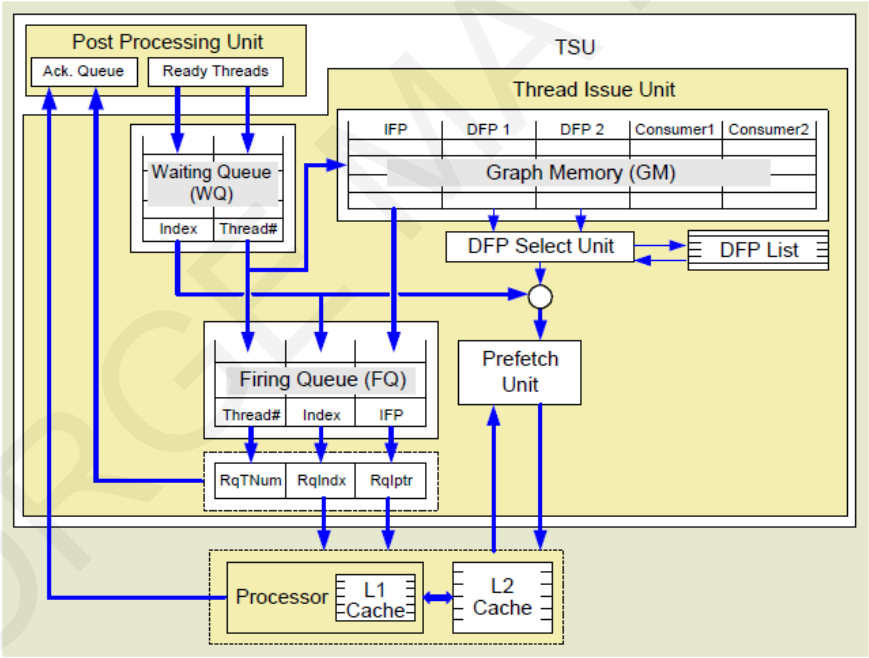


Figure 15: The TIU with the basic prefetch CacheFlow policy [6].

Simulation results, on a 32-node system with CacheFlow, for eight scientific applications, have shown a significant reduction in the cache miss ratio. This results in a speedup improvement ranging from 10% to 25% (average 18%) when the Basic Prefetch CacheFlow policy was used. A larger increase (14% to 34% with a 26% average) was observed when the CacheFlow with Conflict Avoidance policy was used. Finally, a further improvement (18% to 39% with a 31% average) was observed when the CacheFlow with Thread Reordering policy was utilized.

GEORGE MATHEOU

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

The main target of DDM-CMP (Data-Driven Multithreading Chip Multiprocessor) [7, 144] architecture was to explore the potential of applying the DDM model to the Chip Multiprocessor architectures. In addition to the potential performance, the proposed design studied the power consumption, the hardware cost and several ways to benefit from the particular characteristics of CMP architectures. The DDM-CMP design proved able to deliver not only high performance but also high power efficiency. DDM-CMP examined four alternative implementations of the TSU for space savings: (a) each core has its own TSU, (b) one TSU is shared among two cores and the number of system's cores increases, (c) one TSU serves all the cores of the chip and the number of system's cores increases, and (d) one TSU is shared among two cores and the saved space is used to implement on-chip shared cache. Figure 16 illustrates the four alternatives of the DDM-CMP architecture. For evaluation purposes, a DDM-CMP implementation that utilizes the same hardware budget as the Pentium 4 processor was presented, to implement four Pentium III processors together with the necessary TSUs and the interconnection network. The performance results were derived from the results obtained for the D^2NOW implementation [55].

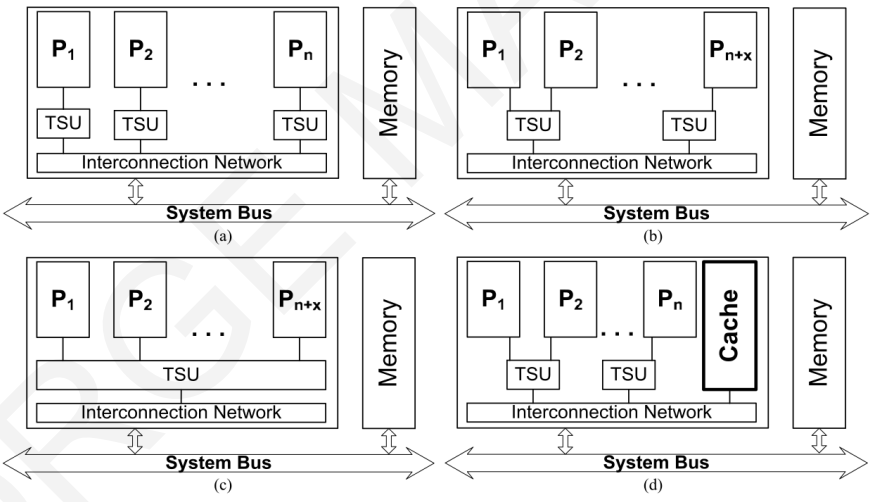


Figure 16: Several alternatives of the DDM-CMP architecture [7].

2.6.4.3 TFlux Parallel Processing Platform

Thread Flux (TFlux) [9, 8] is a parallel processing platform that supports the DDM model on commodity multiprocessor systems. TFlux provides virtualization for the parallel execution of TFlux programs on a variety of computer systems, independently of the underlying architecture. TFlux is composed of a collection of entities in a layered design (Figure 17) that abstracts the details of the underlying machine. The code of a TFlux program consists of ANSI-C and TFlux directives that express the code of the Data-Driven Threads (DThreads) and the dependencies among them. The user's code passes through the TFlux Preprocessor [145] which generates a C program with calls to

GEORGE MATHEOU

the TFlux Runtime Support. The generated code can be compiled using a commodity C compiler. This allows the users to generate binaries for any ISA.

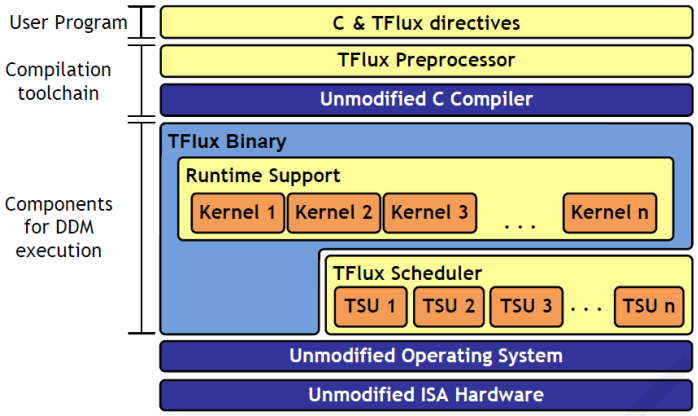


Figure 17: The layered design of the TFlux Platform [8].

The Runtime Support runs on top of an unmodified Unix-based Operating System (OS) and hides the details of the TSU implementation. The functionality of the runtime is supported by simple user-level processes called *TFlux Kernels*. The main target of the TFlux Kernel is to communicate with the TFlux Scheduler in order to schedule DThreads according to the DDM model. The TFlux Scheduler consists of a group of TSUs (called TSU Group). The TSU Group is a single unit that it is responsible for the scheduling of threads based on data availability. It consists of a global part shared by all Kernels and private TSU units that serve each Kernel individually.

TFlux has three basic implementations: *TFluxHard*, *TFluxSoft* and *TFluxCell*. Recently, the *TFluxSCC* system [146] was proposed, which allows DDM execution on many-core devices, like the 48-core Intel Single-chip Cloud Computing (SCC) processor. TFluxSCC achieves scalable performance using a global address space without the need of cache-coherency support. One major difference between TFluxSCC and previous TFlux implementations is that it has a non-centralized runtime system, i.e., TSU functionalities are distributed to the cores of the system.

Data-Driven Multithreading C Pre-Processor (DDMCPP)

TFlux provides the Data-Driven Multithreading C Pre-Processor (DDMCPP) tool [145]. DDMCPP takes as input a regular C code program with DDM directives and automatically generates TFlux code which includes all the necessary calls to the TFlux Runtime System. DDMCPP is divided into two modules, the front-end and the back-end. The front-end module is a parser tool which parses the DDM directives and then it passes the information to the back-end module. The back-end module generates the code required for the TFlux runtime support, such as, the Kernel code and the load operations to the TSU.

GEORGE MATHEOU

TFluxHard

TFluxHard [9] is a shared-memory Chip Multiprocessor augmented with the TSU Group. TFluxHard was evaluated using the Simics full-system simulator [147] which runs unchanged production binaries of the target hardware at high-performance speeds. The TSU Group is a hardware module that is attached to the system's network as a memory-mapped device. The CPU communicates with the TSU via the Memory-Mapped Interface (MMI). MMI snoops the network and transfers all the memory requests directly to the TSU. Also, it sends information to the network when the TSU wants to communicate with the CPU. TFluxHard groups multiple TSUs (one for each core) into one single unit in order to decrease the additional interconnection cost. Figure 18 depicts a TFluxHard chip configured with 4 cores.

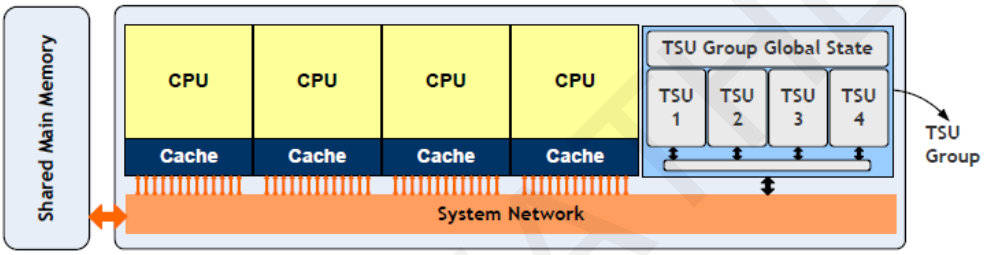


Figure 18: A TFluxHard chip with 4 cores [8].

TFluxSoft

The TFluxSoft [9] implementation targets commodity multi-core processors with a single address space and hardware cache coherency. Figure 19 depicts the execution of TFluxSoft on a multi-core system with *n* CPUs. The TSU (or *TSU Emulator*) is implemented as a software module which emulates the functionalities of the hardware TSU. TSU is executed on one of the cores of the multi-core processor while the DThreads are executed on the other cores. In TFluxSoft, both the TSU Emulator and the execution of application's DThreads use POSIX threads. TFluxSoft avoids to overload the core that provides the TSU functionality by splitting the TSU's operations. Specifically, some operations are executed by the Local TSUs while others by the TSU Emulator.

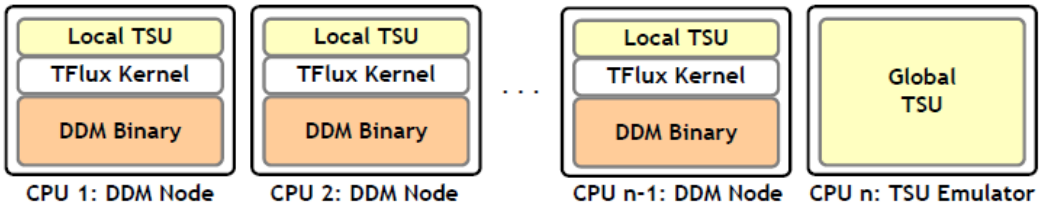


Figure 19: TFluxSoft system on a system with n CPUs [9].

TFluxCell

The TFluxCell [9] implementation targets the Cell/BE processor. Cell/BE is a heterogeneous multi-core chip processor which consists of nine cores. The one of them is a general purpose processor called PPE (Power Processor Element) whereas the other eight cores are fully functional SIMD co-processors, each called SPE (Synergistic Processor Element). TFluxCell executes the DThreads on the SPE cores while has the TSU Emulator running on the PPE core (like in TFluxSoft implementation). The TFlux Kernels communicate with the TSU and vice versa using DMA calls and mailboxes. Furthermore, the threads exchange data among them using a shared buffer in the main memory. When a thread finishes its execution it stores the produced data into a shared buffer in main memory. After that, the data from the buffer is transferred into the Local Store (LS) of the SPEs of the consumer-threads before start their execution. Figure 20 depicts the TFluxCell system.

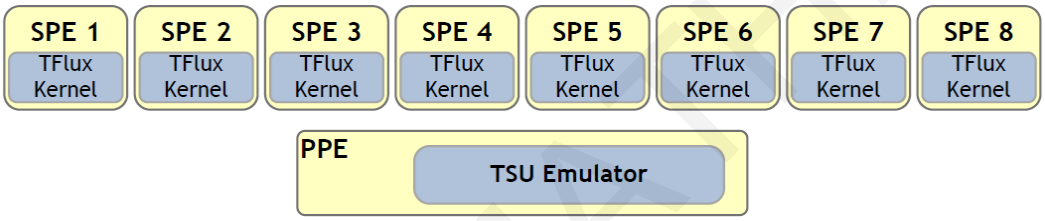


Figure 20: The TFluxCell system [8].

Transactional Memory Support

Diavastos *et al.*, in [148, 149], integrated Transactional Memory (TM) support in the TFlux platform. TM support provides simplified sharing of mutable data in those circumstances where it is important to the expression of the program. For this purpose the TinySTM software library [150] was used. Additional TFlux directives were introduced for providing TM functionalities at the user level. These directives allow users to define transactional threads and variables. The TFlux runtime system (the TSU) runs on top of the TM runtime library and it is responsible only for scheduling data-flow threads while the TM runtime is only responsible for aborting/committing transactions. The scheduling of a data-flow thread may impose the start of a transaction but the TSU will not interfere with the monitoring of the transactions. When a transaction starts executing, the TM runtime will take over and monitor for conflicts. If such a conflict occurs, the TM runtime will reschedule the transaction without the TSU noticing any changes. As soon as a transaction commits and the data-flow thread is finished, the TSU will take over again and schedule the next ready thread.

2.6.4.4 Data-Driven Multithreading Virtual Machine (DDM-VM)

The Data-Driven Multithreading Virtual Machine (DDM-VM) is a virtual machine that supports DDM execution on homogeneous and heterogeneous multi-core systems. The Thread Scheduling

GEORGE MANIOU

Unit (TSU) is implemented as a software module and it is responsible for scheduling threads dynamically at runtime, based on data-availability. A DDM-VM program consists of ANSI-C with a set of C macros that expand into calls to the DDM-VM runtime. DDM-VM programs consist of two parts: (1) the code of the DThreads and (2) the dependency graph which describes the consumer-producer dependencies among the DThreads. DDM-VM has three different implementations: the Data-Driven Multithreading Virtual Machine for the Cell processor (DDM-VMc) [11, 10], the Data-Driven Multithreading Virtual Machine for Symmetric Multi-cores (DDM-VMs) [11, 35] and the Distributed Data-Driven Multithreading Virtual Machine (Distributed DDM-VM) [11, 63].

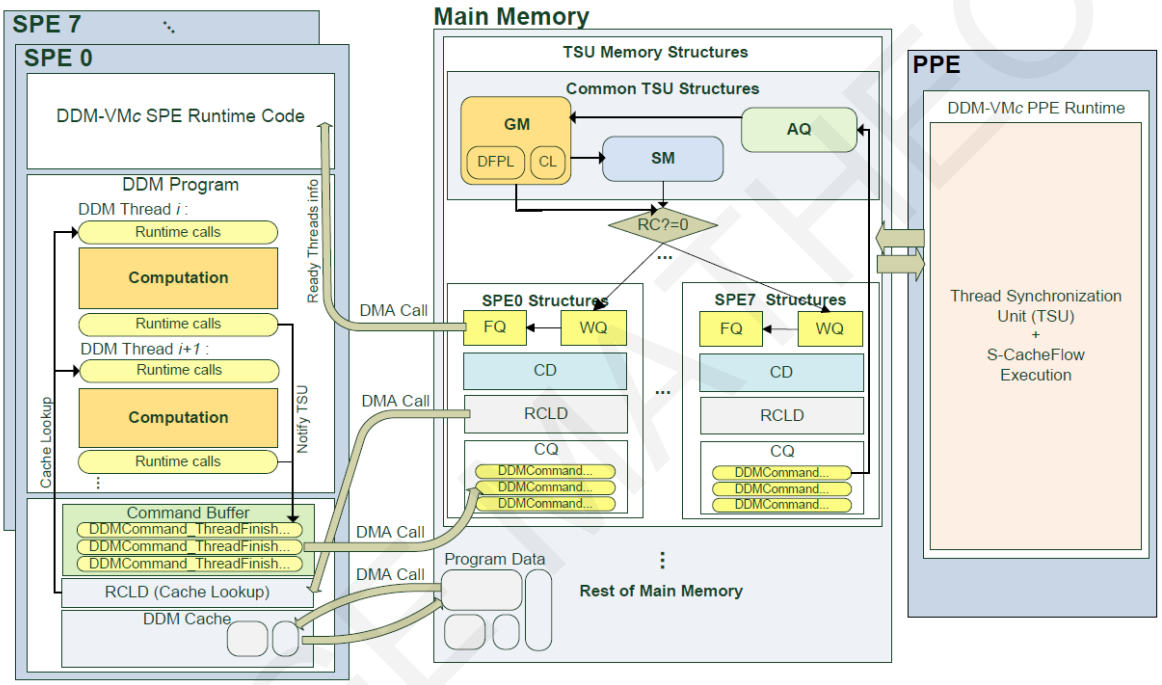


Figure 21: The architecture of the DDM-VMc [10].

DDM-VMc

The architecture of DDM-VMc [10] is depicted in Figure 21. The DDM-VMc targets heterogeneous multi-cores with software-managed memory. It was evaluated on the Cell Broadband Engine (BE) processor. The TSU was implemented as a software module running on the PPE core where the execution of the threads takes place on the SPE cores. The TSU memory structures, i.e., the structures holding the synchronization information and the state of the TSU, are allocated in main memory. DDM-VMc implements the CacheFlow policy [143] in software using the Software CacheFlow (S-CacheFlow) module. S-CacheFlow is a software prefetching cache module that manages data transfers between the main memory and the LS memory transparently to the programmer. Also, S-CacheFlow module is responsible for managing prefetching automatically. The LS memory is used for storing the DDM threads linked with the runtime library, some of the S-CacheFlow structures and

the data of the threads. The Direct Memory Access (DMA) technology is used for the communication between the runtime on the SPEs and the TSU module that runs on the PPE.

DDM-VMs

DDM-VMs targets homogeneous multi-cores architectures. The TSU runs as a software module on one of the cores while the threads' execution takes place on the other cores, as shown in Figure 22. The DDM threads communicate with the TSU via the main memory. This is because the state of the TSU structures are allocated in main memory which is shared between the threads and the TSU module. The DDM-VMs implementation was accomplished by using the basic functionalities of the DDM-VMc implementation [11] and more specifically the functionalities of the TSU module. The main differences between the two implementations are the following: the TSU in DDM-VMc runs on the PPE core which has a separate address space from the SPE cores executing the threads. Contrary, in DDM-VMs, all cores share the same address space. Additionally, in DDM-VMs, the memory hierarchy is managed by hardware while, in DDM-VMc, the memory hierarchy of the Cell processor is managed by the S-CacheFlow module.

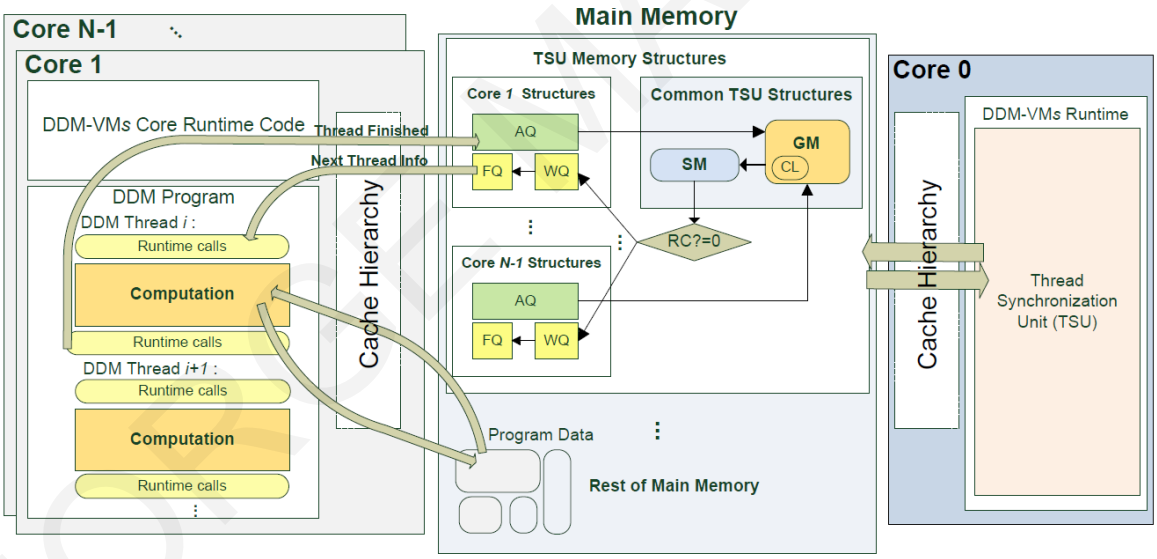


Figure 22: The architecture of the DDM-VMs [11].

Distributed DDM-VM

The Distributed DDM-VM [11, 63] implementation supports DDM execution across a number of multi-core nodes (a cluster) connected over an off-chip network. Each node is an independent multi-core machine running an operating system and capable of executing multiple DDM threads concurrently [11]. A Shared Global Address Space (GAS) was implemented across all the nodes in order to transfer the data produced by a thread to its consumer threads that is possible to run in

different nodes. Figure 23 illustrates the architecture of the Distributed DDM-VM. Each node has its own TSU module where the multiple TSUs communicating across the network to coordinate the overall DDM execution. For the communication between multiple TSUs, DDM-VM implements the Network Interface Unit (NIU) [55] in software. NIU is responsible for handling the low-level communication operations.

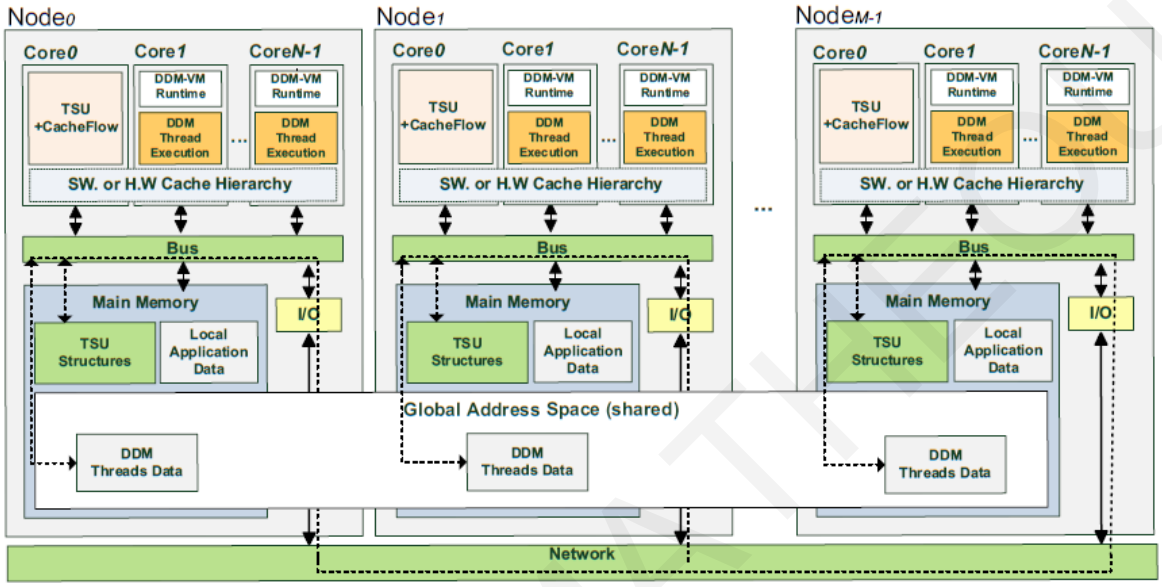


Figure 23: The architecture of the Distributed DDM-VM [11].

Runtime dependency resolution with I-Structures

The previous DDM implementations utilize the compile-time dependency resolution, i.e., the programmer was responsible for constructing the dependency graph. However, some dependencies cannot be discovered at compile-time and thus the code has to be executed serially. To this end, DDM-VM introduced a run-time dependency resolution protocol [36], based on the I-Structure model [96, 97]. The basic mechanism of the run-time dependency resolution is as follows: a task/thread exposes its input and output data to a scheduler, which as the execution of tasks proceeds, it examines the output data of a completed task and checks if it satisfies any of the pending dependencies. When all the dependencies of a task are satisfied the task is ready for execution. This approach can handle all programs but incurs extra overheads at runtime even when part of the dependencies can be determined statically at compile time.

DDM-VM programming tool-chain

The DDM-VM programming tool-chain provides the following methods for developing DDM applications:

- **C macros:** a set of macros is used to develop a DDM-VM program in C language. The macros identify the boundaries of the threads and the producer-consumer relationships amongst the threads. The macros expand into calls to the TSU in order to manage the execution of the program according to the DDM model.
- **TFlux Directives:** the DDMCPP [145] tool was extended to generate code that targets the DDM-VM system. The tool takes as input a regular C code program with DDM directives and automatically generates code which consists of C code augmented with the DDM-VM macros.

2.7 Concluding Remarks

Unsustainable power consumption and ever-increasing design and verification complexity have driven the microprocessor industry to pack multiple cores on a single chip as an architectural solution for sustaining the Moore's law [14]. As a result, the major challenge today is to find programming/execution models that will be able to efficiently keep all the available resources busy while keeping energy efficiency in high levels. Such a model is the data-flow model of execution [27, 68, 28, 29, 30].

Several research projects have adopted the data-flow principles but only few were implemented in real hardware. This is because the development of a real hardware system is an expensive and complicated procedure. On the other hand, a hardware system can deliver the ultimate performance compared to a software implementation or simulation with the same functionalities. Early hardware data-flow prototypes (Table 1) as well as recent hardware data-flow systems like TRIPS [45, 134, 59], WaveScalar [54, 60] and DTA-C [53, 132, 56], propose new processor architectures. Thus, they cannot utilize the current processor technology. Table 2 compares several recent hardware data-flow developments (real and simulated implementations). On the other hand, several software data-flow projects were implemented (Table 3) in order to allow data-driven/data-flow execution on conventional/commodity multi-core and many-core systems.

This thesis proposes the implementation of software and hardware data-flow/data-driven systems based on the DDM model of execution for the following reasons:

- DDM allows efficient dynamic data-flow execution, in a coarser-grained level, on conventional processors. Thus, the maximum available parallelism can be exploited as well as communication and synchronization latencies can be tolerated.
- The fact that DDM can be implemented on unmodified commodity microprocessors allows utilizing the state-of-the-art in processor design.
- Data-flow-based systems, like DDM, do not require the support from an underlying coherent shared-memory system. Data shared among different threads may be shipped to and from the threads before and after their execution. The only requirement is to provide a single address space for all computing cores [151].

GEORGE MATHEOU

Name	Architecture	Evaluation Details
Manchester Prototype	Pure Data-flow: Dynamic	• 14 benchmark programs were used with 29 input data sets • A software data-flow simulator was used for making assumptions about the system architecture • Programs were implemented in the MACRO, MAD and SISAL languages
SIGMA-1		• Architecture with 128 PEs and 128 structure elements • Evaluation with 11 programs • Programs were developed in DFC, a single assignment language whose syntax is similar to the C language.
Monsoon	Pure Data-flow: Explicit Token Store	• Evaluation with large programs (GAMTEB, TSP, etc.) • The Id language was used
EM-4	Hybrid Data-flow	• Architecture with 80 PEs • 4 programs were developed • Every program was written in assembly language
EPSILON-2		• A sample program was described but results are not provided • Programs are compiled in Id
StarT(*T)		• Hardware prototype using the 2-issue superscalar RISC microprocessor Motorola 88110

Table 1: Early representative hardware data-flow prototypes (from 1974 to 1992).

- DDM implements the CacheFlow policy which improves locality, thus, reducing cache misses even on caches of small sizes.
- The dependency graph of DDM programs is built at compile time (Static Dependency Resolution), thus, DDM exploits all the available parallelism at any given time with the minimum overheads. Data-flow systems based on Dynamic Dependency Resolution (built their dependency graph at runtime) expose only a part of the dependency graph and consequently a fraction of the concurrency opportunities is visible at any given time. Also, building the dependency graph at runtime incurs extra overheads (extra cycles and power). Systems that support Dynamic Dependency Resolution are the StarSs [124, 125, 39] and OmpSs [40, 128], and the C++ framework of Gupta and Sohi [37]. Finally, implementing a data-flow system with Dynamic Dependency Resolution in hardware, like Task Superscalar [4], requires complex modules and a lot of resources (Flip Flops, LUTs, etc.) [46].

GEORGE MATHEOU

	EARTH	D²NOW	TRIPS	WaveCache	DTA-C	TFluxHard	Task Superscalar	OpenSPlySER
Year Started	1995	1999	2003	2003	2007	2008	2010	2012
Execution Model/Architecture	EARTH	DDM	EDGE	WaveScalar	SDF	DDM	Task Superscalar	DySER
Main Features	• Decoupled Non-blocking Multithreading • Dynamic Load balancing via Work Stealing	• Decoupled Non-blocking Multithreading • Simulated cluster of distributed workstations • CacheFlow	• Polymorphous Parallelism (multiple parallelism modes) • Statically assigning instructions of a block to ALUs in order to reduce inter-instruction communication distances and exposing parallelism • Loop Unrolling • Branch Prediction	• Tagged-Token dynamic data-flow architecture • Fully distributed data-flow tag management scheme that is under compiler control • Wave-ordered memory technique for guaranteeing correct memory ordering • Loop Unrolling • Cache coherency	• Decoupled Non-blocking Multithreading • Overlapping the three execution phases of the threads (pre-load, execute, post-load) • Transactional Memory • Clusterizing resources	• Decoupled Non-blocking Multithreading • A shared memory Chip Multiprocessor augmented with a hardware TSU Group • TFlux Pre-processor	• Out-of-order task execution (implements the StarSs model in hardware) • Dynamically detects inter-task dependencies and constructs the data dependency graph at runtime • Renaming memory objects to break anti- and output dependencies	• Accelerator design integrated into the OpenSPARC processor pipeline • Profiling-based detection of ultra-wide instructions • Compiler support to execute instructions on DySER blocks. A DySER block consists of a circuit-switched network of heterogeneous functional units (FUs)
Implementation Details	Emulated using the MANNA processor	An execution driven simulator was used, based on native execution	Hardware prototype (170M transistor, 130nm ASIC chip)	• A synthesizable RTL model was implemented using Verilog which targets a 90nm ASIC implementation • Cycle-level functional simulation	Simulated with a cycle-accurate C++ simulator	• Evaluated using the Simics simulator • The CACTI tool was used for hardware budget estimation	• Evaluated using TaskSim, a trace-driven cycle-accurate CMP simulator • Simulated in the ModelSim simulator • Synthesized in Xilinx FPGA devices (Virtex-7 Family)	• FPGA emulation (Xilinx Virtex-5 FPGA). Applications executed under the Ubuntu 7.10 OS. • Simulator on top of the M5 SPARC
Distributed Execution?	Yes	Yes	Yes	No	Yes	No	No	No
Parallelism Level	TLP	TLP	ILP, TLP, DLP	ILP, TLP	TLP	TLP	TLP	ILP, DLP, TLP (dual threaded)
Block Granularity	Thread (Any Size)	Thread (Any Size)	HyperBlock (128 instructions)	Wave (any number of instructions)	Thread (Fixed Size)	Thread (Any Size)	Thread/Task (Any Size)	Equivalent to few 100s of ISA instructions
Inter-block Scheduling	Dynamic Data-flow	Dynamic Data-flow	Control-Flow	Dynamic Data-flow	Dynamic Data-flow	Dynamic Data-flow	Dynamic Data-flow	Control-Flow
Intra-block Scheduling	Control-Flow	Control-Flow	Static Data-flow	Dynamic Data-flow	Control-Flow	Control-Flow	Control-Flow	Static Data-flow
Programmability	Threaded-C	C. The Microsoft Visual C++ V6 compiler was used.	C and Fortran. The Scale compiler was used.	C. The Compaq cc (v5.9) compiler was used.	• Sisal and MIDC as intermediate language • Haskell	• C augmented with macros • Source-to-source compiler	C with the StarSs source-to-source compiler	The LLVM compiler framework was used
Dependency Resolution	Static (by programmer)	Static (by programmer)	Static (by the compiler, the ISA helps because in each instruction only its consumers are specified and no its source operands)	Dynamic (compiler breaks the program into waves and dependencies among instructions are detected by the hardware at runtime)	Static (by programmer)	Static (by programmer)	Dynamic (annotating the operands of the tasks and the hardware uses these annotations to dynamically detect inter-task dependencies)	Static (by compiler)

Table 2: Recent hardware data-flow developments (real and simulated implementations).

	StarSs	Intel TBB	TFlux	Cilk Plus	DDM-VM	Gupta and Sohi's Execution Model	SWARM	OmpSs
Year Started	2006	2007	2008	2010	2011	2011	2011	2011
Execution Model/Architecture	StarSs	Intel TBB	DDM	Cilk	DDM	Gupta and Sohi's Execution Model	Codelets/EARTH	StarSs

Main Features	• Parallel programming platform that targets a variety of architectures:
	  ○ Cell processor (CellSs)
	  ○ Multi-cores and symmetric multiprocessors (SMPSs)
	  ○ GPUs (GPUSs)
	• Automatic function level parallelism
	• Data renaming (leaving only the true dependencies)
	• Exploits data locality by scheduling dependent tasks sequentially to the same core
	• Enables task hierarchy, meaning it allows the instantiation of tasks within tasks
		• Relies on C++ templates to facilitate parallel programming
		• It provides a set of data structures and algorithmic skeletons that supports the execution of tasks
		• Supports data-flow graphs
		• Provides concurrent containers (hash-maps, queues, etc.) and synchronization constructs (mutex, atomic operations, etc.)
		• Fork-join approach for the creation and management of tasks
			• Decoupled Non-blocking Multithreading
			• TFlux Pre-processor
			• Parallel processing platform that targets:
			  ○ Commodity multi-core processors (*TFluxSoft*)
			  ○ Cell/BE processor (*TFluxCell*)
			  ○ Many-core devices (*TFluxSCC*)
			• Transactional Memory
				• Fork-join Execution Model
				• Task parallelism constructs
				• Array notations for expressing data parallelism
				• Reducers for eliminating contention for shared data
				• SIMD-enabled functions
					• Decoupled Non-blocking Multithreading
					• Virtual machine that supports DDM execution on homogeneous (*DDM-VMs*) and heterogeneous (*DDM-VMc*) multi-core systems
					• Supports systems with software-managed memory hierarchies using Software CacheFlow (S-CacheFlow)
					• Distributed DDM execution
					• Runtime dependency resolution with I-Structures
						• Data-flow execution of sequential imperative programs on multicore architectures
						• C++ runtime library
						• Functional-Level Parallelism (FPL)
						• Object-based data-flow graphs
						• Identify and track data-dependencies at runtime
							• Decoupled Non-blocking Multithreading
							• Heterogeneous Computing:
							  ○ x86
							  ○ Multicores
							  ○ Servers
							  ○ GPGPUs
							  ○ Intel's Xeon Phi
								• A programming model that provides the features of the StarSs using OpenMP directives
								• Includes CUDA and OpenCL kernels in order to provide a single programming environment that covers the different homogeneous and heterogeneous architectures
								• Utilizes a thread-pool model
								• FPGA support (*OmpSs@FPGA*)
								• Allows the exploitation of shared memory parallelism on a cluster of nodes with accelerators (*OmpSs@cluster*)

Scheduling Policy	Locality-aware + Work-Stealing	Work-Stealing	Static	Work-Stealing	• Dynamic • Round-Robin • Modular	Work-Stealing	Work-Stealing	Locality-aware + Work-Stealing
Distributed Execution?	No	No	No	No	Yes	No	Yes	Yes
Programmability	C augmented with *pragmas* (a source-to-source compiler is used)	C++	• C augmented with macros • Source-to-source compiler	C and C++	• C augmented with macros • TFlux Source-to-source compiler	C++	• C with the SWARM API • SCALE (extension of C)	C/C++ augmented with *pragmas* (the Mercurium source-to-source compiler is used)
Dependency Resolution	Dynamic (through hints given by the programmer in form of *pragmas*)	Static (by the programmer through C++ templates)	Static (by the programmer)	Static (by the programmer, using keywords)	• Static (by the programmer) • Dynamic (with I-Structures)	Dynamic (by the runtime using the *data sets* of the functions)	Static (by the programmer)	Dynamic (*in*, *out* and *inout* clauses are provided by the programmer)

Table 3: Recent software data-flow developments.

GEORGE MASON U

Chapter 3

MiDAS: a Multi-core system with Data-Driven Architectural Support

3.1 Introduction

The DDM model was evaluated by three basic software implementations (DDM-VM [35, 10, 11], TFluxSoft [9] and FREDDO [142]) and two simulated hardware systems (D^2NOW [139, 55, 6] and TFluxHard [9, 8]). In this thesis we are moving to the next step by developing a real hardware system that supports data-driven execution under the DDM model. TSU was implemented in hardware using Verilog [152]. In order to demonstrate the efficiency and functionality of the hardware TSU implementation, a shared-memory multi-core system was developed which is managed by the TSU. The developed processor is called *MiDAS* (Multi-core with Data-Driven Architectural Support). A software API (supports both C and C++) is provided for developing DDM applications. For evaluation purposes, a Xilinx ML605 Evaluation Board is used, that is equipped with a Xilinx Virtex-6 FPGA [62]. In this chapter we present the hardware TSU implementation of the DDM model (Section 3.2) and MiDAS's architecture (Section 3.3). Chapter 5 describes the programming methodology and the software API for developing DDM applications targeting the MiDAS system. Finally, a comprehensive evaluation of the hardware TSU and MiDAS is provided in Chapter 7.

3.2 TSU: Hardware Support for DDM

The Thread Scheduling Unit (TSU) is a fully parameterizable hardware Intellectual Property (IP) core. It uses Thread Templates for the data-driven scheduling of DThreads. A DThread is identified by the Thread ID (TID) and Context and is paired with a Thread Template. Currently, the hardware TSU implementation utilizes only 32-bit Context values. Furthermore, it allows the parallelization of nested loops using the Nesting attribute (like in DDM-VM and FREDDO). In our implementation we allow up to three nesting levels, i.e., the DThreads are able to implement one-level (Nesting-1), two-level (Nesting-2) or three-level (Nesting-3) nested loops. If a DThread does not implement a loop, its Nesting attribute is set to zero (Nesting-0). The Nesting attribute is used in combination with

GEORGE MATHEOU

the Context attribute. The TSU uses the Nesting attribute to manage the Context value properly. More details about the Nesting attribute can be found in Section 2.6.1.2.

3.2.1 Thread Template

The Thread Template is a collection of the following attributes: Thread ID (TID), Instruction Frame Pointer (IFP), Ready Count (RC), Nesting and Scheduling Policy. TID identifies uniquely a DThread, while IFP is a pointer to the address of the DThread's first instruction. RC specifies the number of producers of a DThread. Finally, the Scheduling Policy determines the method that is used by the TSU to map the ready instances of a DThread to the processing elements. It consists of two distinct parts, the Scheduling Method and the Scheduling Value.

3.2.2 TSU Micro-architecture

The block diagram of the TSU's micro-architecture, supporting an arbitrary number of cores, is shown in Figure 24. Each block of this diagram comprises a Verilog module that consists of several internal hardware modules. Following, we elaborate on the functionality of each block comprising our proposed hardware TSU.

3.2.2.1 TSU-Cores Communication

During DDM operation there are five basic transactions that are carried out when the TSU communicates with the interconnected cores: *(1)* transmission of Update commands from the cores to the TSU, *(2)* transmission of ready DThread instances from the TSU to the cores to be executed, *(3)* storage and removal of Thread Templates to/from the TSU, *(4)* transmission of control signals from the cores to the TSU, and *(5)* receipt of acknowledgement (acks) and status signals at the cores from the TSU. Control signals are used to inform the TSU about the number of cores that will be used during DDM operation. This is necessary in cases such as when fewer cores are chosen to be utilized than the total number of cores available in the architecture required when evaluating system scalability. Control signals are also used to invalid RC entries of a specific TID from the Synchronization Memory of the TSU, such as when a user deletes a Thread Template. Acknowledgement signals are sent from the TSU to the cores after the completion of data management operations such as when storing/removing Thread Templates and invaliding RC entries. Finally, status signals are sent from the TSU to the cores in order to inform the user about the status of the TSU's data-structures, such as the number of Thread Templates stored in TSU or when a data-structure is full.

Operations *(1)* and *(2)* are the most frequently carried out operations when the DDM is active. Thus, a very fast communication system is required to achieve high performance. We endorse the use of low-complexity high-performance FIFO-based buses to carry out such operations. For the

GEORGE MATHEOU

remaining operations we favor the use of a lightweight bus with memory-mapped communication functionalities, such as read/write from/to control and status registers. In this work we use a Xilinx FPGA and associated software tools to prototype and evaluate our design. As such, we used bus architectures provided by Xilinx to handle the communication between the TSU and the interconnected cores. The FIFO-based buses were implemented using the Fast Simplex Link (FSL) interface [153], while the memory-mapped bus was constructed using the AXI4-Lite interface [154]. FSL is a very fast 32-bit wide interface that provides unidirectional FIFO-based communication between any two design elements and allows single-cycle read/write operations. AXI4-Lite is a simple low-throughput memory-mapped bus, and comprises an implementation of the Advanced eXtensible Interface (AXI) protocol based on the AMBA interface specification from ARM. The AXI4-Lite read/write latency is about 4-8 clock cycles long, depending on the chosen configuration [154].

For each core, the TSU utilizes two different FSL buses, the *Input FSL Bus* and the *Output FSL Bus*. The Output FSL Bus holds the ready DThread instances that will be executed by the associated core. The Input FSL Bus holds Update commands that are received from the associated core. Three different Update commands are supported:

1. **Single Update** which decreases the RC value of a specific DThread.
2. **Multiple Update** which decreases the RC value of multiple instances of a specific DThread.
3. **Simple Update** which decreases the RC value of a DThread with Nesting=0. It does not require a Context value since a DThread with Nesting=0 has only one instance with Context=0.

Although our first hardware TSU implementation (presented in [141]) supported additional Update commands that decrement the RC value(s) of all the consumer-DThreads of a DThread, such commands are not supported in the current implementation. This is because we observed that these commands, called *Consumer Update commands*, are rarely used in real-life applications. The reason is that such commands are used when all the Consumer DThread instances *(i)* should be updated with the same Context value, *and (ii)* have the same Nesting value. As such, complex and large hardware modules required to support Consumer Update commands such as *Graph Memory (GM)* [55, 141] are excluded from our current design. Notice that our first hardware TSU implementation (presented in [141]) was utilizing a GM module consisted of two internal fully associative data-structures where the consumers' TIDs were stored as Linked Lists. GM operations (read, write and invalid) incurred a lot of overheads, especially when the number of consumers is large, due to the complex organization of the GM's data structures. A faster GM can be implemented as a direct-mapped data-structure where each entry will hold all the consumers of a DThread. This requires to define a maximum number of consumers that each DThread will have (e.g., up to 16).

GEORGE MATHEOU

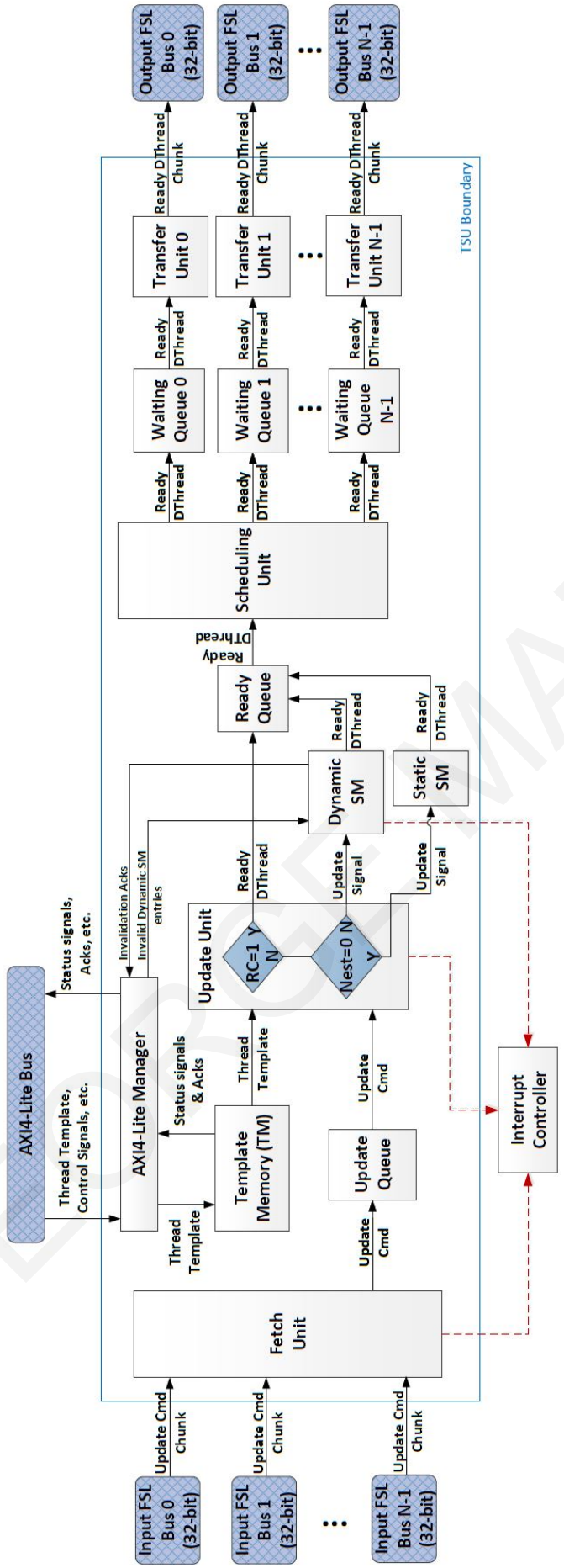


Figure 24: Block diagram of the TSU micro-architecture supporting an arbitrary number of cores.

GEORGE MATHEOU

3.2.2.2 Template Memory (TM)

The Template Memory is used by the TSU to access the Thread Templates of DThreads. The block diagram of TM is illustrated in Figure 25. It is a direct-mapped data-structure where the TID of DThreads is used to index Thread Templates. Each TM entry consists of the attributes of the Thread Template, as well as the Valid field which indicates whether the entry contains valid data. We note that the TID is not required to be stored in the TM's entries since it is used as the memory address. This configuration allows the TM to store $2^{TID\ Size}$ Thread Templates, where $TID\ Size$ is the size of the TID attribute in bits; for example, with an 8-bit TID, TM can store up to 256 Thread Templates. The TM utilizes a dual-port RAM which allows writing/invalidating and reading operations to be executed simultaneously. Writing/invalidating operations are used from the TSU's AXI4-Lite Manager to store/remove Thread Templates while read operations are used by the Update Unit (see Section 3.2.2.6) to execute Update commands. The interconnection between the TM, AXI4-Lite Manager and Update Unit modules is shown in Figure 24.

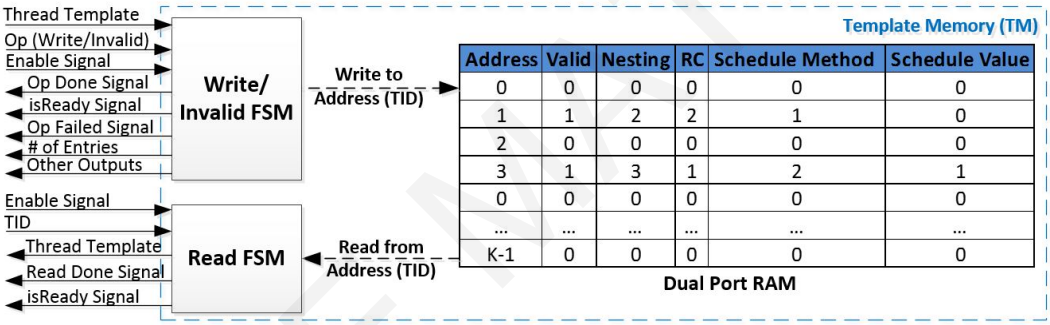


Figure 25: Block diagram of the Template Memory.

3.2.2.3 Handling Ready Count (RC) values

In DDM, the Synchronization Memory (SM) was introduced in order to manage the RC values for each DThread. A DThread that implements a loop has multiple instances, one for each iteration. The TSU holds a separate entry for each instance of a DThread in the SM. Real-life benchmarks can have complex dependency graphs and several loop constructs. As such, an SM can hold thousands or even millions of RC values. This is an issue for a hardware SM implementation since the hardware resources are limited. Notice that a software SM implementation allocates the RC values in the Main Memory. Additionally, the SM unit is the most critical component of a DDM architecture since its performance affects the execution of Update operations. In this work, the hardware TSU implementation handles the allocation of RC values through three basic techniques outlined next.

GEORGE MATHEOU

1) Managing the RC Values of DThreads with $\mathbf{RC = 1}$

The RC values of DThreads with $RC = 1$ are not allocated in the SM. The instances of such DThreads are scheduled for execution when Update operations are received by the TSU. This enables the reduction of SM memory allocations and the acceleration of the Update operations. This technique has also been adopted by the FREDDO framework.

2) Managing the RC Values of DThreads with $\mathbf{RC > 1}$ using DynamicSM

The RC values of DThreads with $RC > 1$ are allocated and deallocated dynamically in the form of blocks, called *RC Blocks*, based on application needs. This enables applications to reuse the RC Blocks, thus saving memory resources. Each RC Block is associated with a counter which indicates the number of valid RC entries. When an RC Block is allocated the counter is initialized, and every time the value of an entry reaches zero the counter is decremented by one. When the counter of an RC Block reaches zero the RC Block is deallocated, i.e., marked as free. This technique is implemented within the hardware TSU with a special SM, called *DynamicSM* (see Figure 24), which is based on the DDM-VM's Hybrid SM [35]. To locate the RC Blocks, DynamicSM uses intermediate entries, called SMI entries. Each SMI entry is associated with an RC Block and it holds information about the DThread that owns the RC Block such as TID and Context ranges, the RC Block's counter (called *Valid Cells #*), etc. The DynamicSM stores the SMI entries in the *SM Indexer (SMI)* and the RC Blocks in the *Ready Count Memory (RCM)*. The RCM holds *M* RC Blocks of *N* entries, allowing the DynamicSM to hold $M * N$ RC values. It is organized as a data-structure with *N* Single Port RAM modules of *M* entries such that each entry of an RC Block is stored in a separate Single Port RAM. This configuration allows the storage of the entries of an RC Block simultaneously, thus accelerating write operations. SMI is divided into *L Context Search Engines (CSEs)* where each CSE is responsible for managing M/L RC Blocks. CSEs are introduced in order to accelerate the SMI's search operations. The architecture and operations of DynamicSM are presented in detail in Section 3.2.2.4.

3) Managing the RC Values of DThreads with $\mathbf{RC > 1}$ and $\mathbf{Nesting = 0}$ using StaticSM

The RC values of DThreads with $Nesting = 0$ and $RC > 1$ are not allocated in the DynamicSM. Such DThreads have one instance only (with $Context = 0$) thus allocating them in the DynamicSM increases complexity as more states should be added to the DynamicSM's Finite State Machine (FSM). Also, RC Blocks can be wasted since a DThread with $Nesting = 0$ will use only one entry of an RC Block. The RC values of such DThreads are stored in a separate SM implementation, called *StaticSM* (see Figure 24). The StaticSM is a simple direct-mapped data structure which holds one RC value for each DThread (Thread Template). To support this functionality, the size of the StaticSM

GEORGE MATHEOU

is set equal to the size of the TM. StaticSM consists of a Single Port RAM and an FSM which is responsible for reading and updating the RC values. To access the Single Port RAM we make use of direct accessing where the address is equal to the DThread's TID. If the RC value of a DThread becomes zero, the StaticSM stores the ready DThread along with its scheduling information in the Ready Queue.

3.2.2.4 DynamicSM Implementation: architecture and operations

Accessing an RC value in the DynamicSM is an associative operation that uses part of the Context to locate the RC Block (which owns the RC value), followed by a direct operation using the remaining part of the Context to locate the exact entry in the RC Block. The architecture of the Dynamic Synchronization Memory (DynamicSM) is shown in Figure 26. It consists of a Finite State Machine (FSM) managing the *SM Indexer (SMI)* and the *Ready Count Memory (RCM)*. The architecture of the SMI and RCM modules and the description of the basic operations of DynamicSM's FSM are outlined next.

Ready Count Memory (RCM)

The RCM holds the RC Blocks where each RC Block has *N* entries. The architecture of RCM is shown in the upper-right side of Figure 26. It consists of *N* Single Port RAM (SP RAM) modules of *M* entries. This supports parallel writing into the SP RAMs which accelerates the initialization of the RC Blocks. The RCM implementation allows the storage of *M* RC Blocks of *N* entries, i.e., $M * N$ RC values. The RCM module supports the following operations:

- **Allocate:** allocates an RC Block where each entry is equal to the RC that is stored in the Template Memory (TM). For example, in Figure 26, the entries of the first RC Block are set to 2. The RC of the TM is written to the SP RAMs in parallel.
- **Invalid:** invalidates an RC Block.
- **Write:** writes to a specific entry of an RC Block.
- **Read:** returns the value of an entry of a specific RC Block.

GEORGE MATHEOU

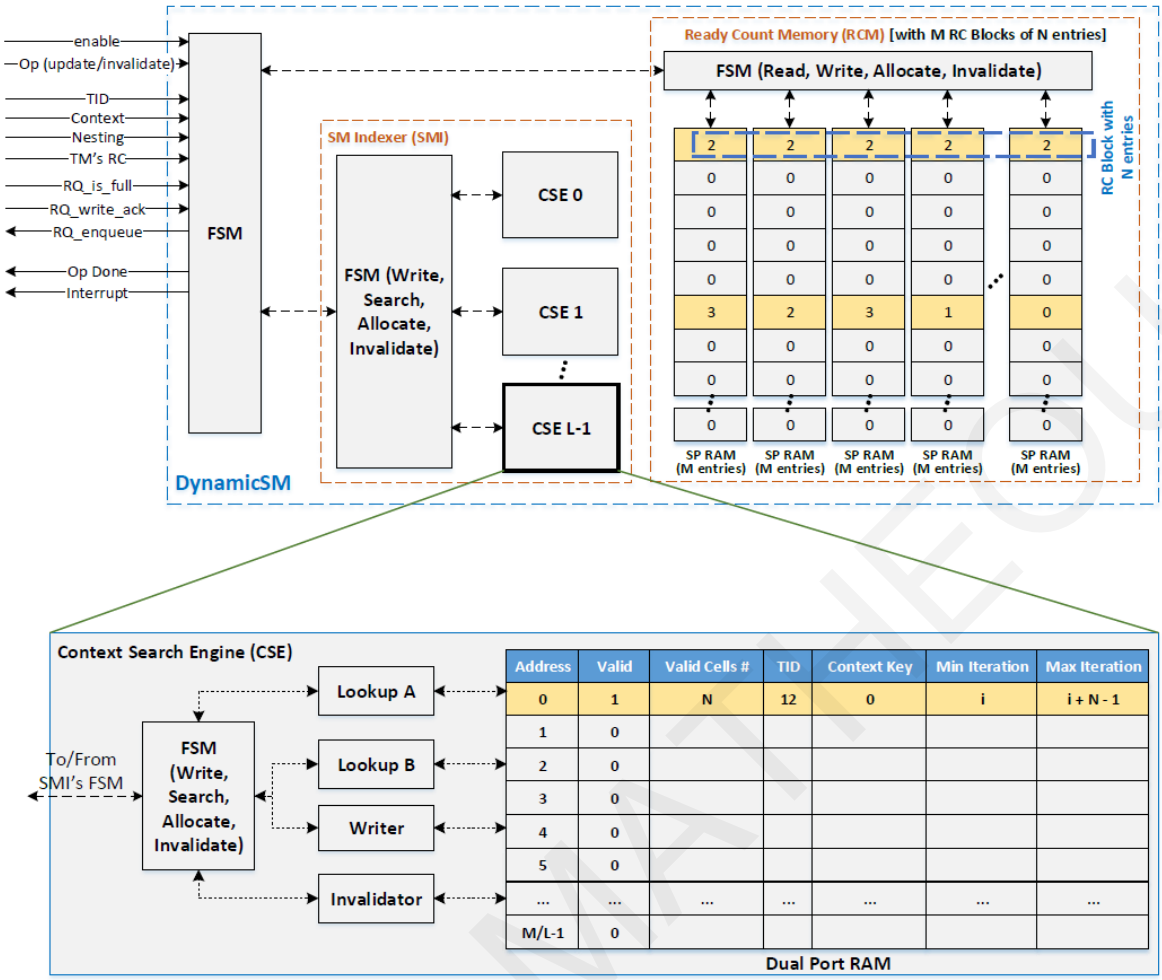


Figure 26: Block diagram of the hardware Dynamic Synchronization Memory.

SM Indexer (SMI)

The architecture of SMI is shown in the top-middle part of Figure 26. It is responsible for holding the SMI entries where each SMI entry corresponds to a specific RC Block. The attributes of an SMI entry are described below:

- **Valid:** indicates if the SMI entry is valid.
- **Valid Cells #:** indicates the number of the valid entries, i.e., non-zero RC values, of the associated RC Block. Initially, it is equal to *N*. If its value becomes zero, then the SMI entry and its RC Block will be deallocated.
- **TID**: indicates the TID of the DThread that owns the RC Block.
- **Context Key**: keeps a part of the Context based on the Nesting attribute. In the case of Nesting-1, it is equal to 0 indicating that it is not used. In the case of Nesting-2, it holds the outer part of the Context. Finally, in the case of Nesting-3, it holds the outer and middle parts of the Context. The *Context Key* allows the RC value of consecutive inner iterations to map in the same RC Block. This technique enables the efficient deallocation of RC Blocks in the case inner loops

with a large number of iterations exist. Since the inner loops are executed first, their RC Blocks will be deallocated before we proceed to the next outer iteration.

- **Min Iteration**: indicates the minimum Context for which the RC Block holds its RC value. It is equal to $(RContext/N) * N$, where *RContext* is the remaining part of the Context which is not included in the *Context Key*. For example, if the RC Block holds the RC values of the Contexts spanning from 0 to 31 (we assume that $N = 32$ and that Nesting=1), then the *Min Iteration* equals to 0.
- **Max Iteration**: indicates the maximum Context for which the RC Block holds its RC value. It is equal to $Min\ Iteration + N - 1$.

The SMI module is divided into *L* Context Search Engines (CSEs), where each CSE is responsible for managing M/L RC Blocks. This enables the searching operations to perform faster since each CSE operates on a subset of the SMI entries (and consequently RC Blocks). The architecture of a CSE unit is depicted in the bottom side of Figure 26. A CSE utilizes a Dual Port RAM for storing the SMI entries. This allows simultaneous access to two SMI entries during the searching operation, and thus the processing time is halved. The operations of the CSE module are described below:

- **Search:** searches for a specific SMI entry, i.e., an entry that is mapped to the input TID and the input Context which is between its *Min Iteration* and *Max Iteration* fields. The searching operation is implemented by two independent modules (the *Lookup A* and *Lookup B* modules) which use the two ports of the Dual Port RAM simultaneously. If the SMI entry is not found, the CSE module informs the FSM of the SMI. If the SMI entry is found, the address of the SMI entry is returned to the FSM of the SMI.
- **Write/Invalid/Allocate:** modifies/invalidates/allocates an SMI entry.
- **Invalidate All:** invalidates all the SMI entries of a specific TID. This is used when the user removes a Thread Template.

DynamicSM's FSM operations

- **Invalidate:** removes the SMI and RCM entries that correspond to the input TID signal. The basic input signals of the FSM are depicted in the upper-left side of Figure 26. The user sends the invalidation signals, i.e., TID, enable signal, etc., to the DynamicSM through the AXI4-Lite bus.
- **Update:** decreases the RC value of a specific DThread with a specific Context. The Update operation consists of three algorithmic steps:
  - **STEP 1 - Searching:** search the SMI module for the SMI entry with $TID = input\ TID$, $Context\ Key = the\ Context\ Key\ of\ the\ input\ Context$, $Min\ Iteration <=$

GEORGE MATHEOU

$RContext$ and $Max\ Iteration >= RContext$. The *RContext* is based on the *input Context*, and the *Context Key*, as we mentioned earlier. For this operation the CSEs of the SMI are used in parallel. For the searching operation we use a hashing technique based on the TID and Context attributes. If the entry is found, go to STEP 2, otherwise go to STEP 3.

- **STEP 2 - Updating:** the RC value of the RC Block that corresponds to the SMI entry found in *STEP 1* is decreased. In particular, the RC value is fetched from the RCM's SP RAM with $ID = RContext - Min\ Iteration$. If the RC value becomes zero, the ready DThread along with its scheduling information are stored in the *Ready Queue* of the TSU (see Figure 24) and the *Valid Cells #* attribute is decreased by one. If the *Valid Cells #* value becomes zero, the SMI entry and its RC block are invalidated.
- **STEP 3 - Allocating:** This step is responsible for allocating a new SMI entry and its associated RC Block. To select a new SMI entry among all the CSEs, we use hashing operations. After the allocation, the RC value will be updated, i.e., go to *STEP 2*.

3.2.2.5 Fetch Unit

The Fetch Unit is responsible for reading the Update commands from the Input FSL Buses and for storing them in the *Update Queue*, as shown in Figure 24. To support this functionality the Fetch Unit utilizes a custom *FSL Reader* for each Input FSL Bus. The FSL Reader fetches Update commands (in chunks of 32 bits) from its associated Input FSL Bus and stores them in an intermediate buffer, called *CMD_Buffer*. The Fetch Unit utilizes a separate CMD_Buffer for each FSL Reader. Each CMD_Buffer holds entire Update commands which come in three categories: single, simple and multiple. A generic round-robin arbiter is used to transfer the Update commands from the CMD_Buffers to the Update Queue. In case an Update command is not valid, an interrupt is sent to an interrupt controller in order to inform the user about the error.

3.2.2.6 Update Unit

The Update Unit fetches the Update commands from the Update Queue and executes them. Each Update command is made up of the following fields: *Type*, *TID*, *Context*, and *Max Context*. For each Update command the Update Unit locates the corresponding Thread Template from the Template Memory. If the RC of the Thread Template is equal to 1, the DThread instance (TID + Context) along with its scheduling information is stored in the Ready Queue for execution. If a Multiple Update is processed, a separate ready DThread entry is stored in the Ready Queue, one for each Context value.

In the scenario where Update commands target DThreads with $RC > 1$, the Update Unit decrements the RC values in the TSU's Synchronization Memories (StaticSM or DynamicSM). For Multiple Updates, a separate Update signal is sent to the DynamicSM, one for each Context value. The

GEORGE MATHEOU

Update Unit manages these Multiple Updates through a special unit, called *Mult_Update_Generator*, where depending on the Nesting value of the DThread, it generates different Context values, from *Context* to *Max Context*. The *Mult_Update_Generator* is omitted from Figure 24 for simplicity. When the RC value of a DThread becomes zero, the SM unit (StaticSM or DynamicSM) stores the ready DThread along with its scheduling information in the Ready Queue. Finally, the Update Unit informs the user about errors through interrupts (e.g., an Update decrements the RC of a DThread which does not exist, or the DynamicSM is full hence an SMI entry cannot be allocated, etc.).

3.2.2.7 Scheduling Unit

The Scheduling Unit dequeues the ready DThread instances along with their scheduling information from the Ready Queue of the TSU (see Figure 24). It enforces the Scheduling Policy by assigning a ready DThread, by inserting its *TID* and *Context*, to the corresponding Waiting Queue. The Waiting Queues hold the ready DThread instances that are waiting to be transferred to the Output FSL Buses. The Scheduling Policy consists of two fields: (1) the scheduling method and (2) the scheduling value. Three scheduling methods have been implemented: dynamic, round-robin, and static. The dynamic method distributes the thread invocations to the cores in order to achieve load-balancing, where the Waiting Queue with the least amount of work is selected. Next, the round-robin method distributes the thread invocations to the cores in a round-robin fashion. We note that the scheduling value is not used in the dynamic and round-robin methods. Last, under the static method, all the instances of a DThread are assigned to a specific core. As such, the scheduling value is used to hold the identity of the specific core. For instance, if a user wants to execute a DThread only on the core with ID=1, then a DThread with Method=Static and Value=1 is created.

3.2.2.8 Transfer Unit

The TSU utilizes the Transfer Units to transfer the ready DThread instances from the Waiting Queues to the Output FSL Buses, as shown in Figure 24. A different Transfer Unit is used for each Waiting Queue-Output FSL pair. The Transfer Unit splits the information of the ready DThread instances into 32-bit chunks since the FSL bus is a 32-bit wide interface. Each ready DThread instance contains a TID and a Context value. We note that the Instruction Frame Pointer (IFP) of each ready DThread is stored in the DDM application, as the hardware TSU will be attached to an SPMD (single program, multiple data) architecture. As such, the IFP of each DThread has a different value in each core.

3.2.3 TSU's RTL schematics

In this subsection we present Register Transfer Level (RTL) schematics of the hardware TSU implementation. In order to generate high-quality schematics we have ported the TSU's Verilog code

GEORGE MATHEOU

in the Xilinx Vivado Design Suite (version 2017.1) [155]. Additionally, the TSU was configured to support eight cores. Figure 27 depicts the high-level RTL schematic of the TSU architecture. Several hardware components (signals, gates, registers, multiplexers, etc.) were omitted from the schematics for simplicity.

The Fetch Unit consists of two different modules, the *CMD_BUF_SET* and the *CMD_Mng_Unit* (Figure 28). It also includes the FSL Readers (one for each Input FSL Bus) which are implemented in the Xilinx Platform Studio (XPS) tool, thus are not visible in the Vivado's RTL schematics. CMD_BUF_SET is a generic Verilog module which holds the CMD_Buffers (one for each core). The CMD_Mng_Unit is responsible for dequeuing the Update commands from the CMD_Buffers (in a round-robin fashion) and for storing them in the Update Queue. The basic modules of the CMD_Mng_Unit are: a generic *round_robin_arbiter*, a General Priority Encoder (*GeneralPriorityEncoder*) and a general Multiplexer (*GeneralMUX*). The GeneralPriorityEncoder and GeneralMUX modules are used to select the data of the selected CMD_Buffer among the input data of all CMD_Buffers (the input data of all CMD_Buffers is transferred via the same input signal, called CMD_BUF_SET_outputs).

Figure 29 depicts the connection interface between the Template Memory, the Update Queue, the Update Unit and the Ready Queue. The Update Unit (Figure 30) includes a controller of the Dynamic Synchronization Memory (SM) and a multiple Update generator (MULT_UPD_GEN_RC1) which is used for generating Update signals in the case of Multiple Updates targeting DThreads with RC=1. For simplifying the TSU's top Verilog module, the Dynamic and Static SMs were placed inside the Update Unit.

Finally, Figure 31 depicts the connection interface between the Ready Queue, the Scheduling Unit, the Waiting Queues and the Transfer Units. The Waiting Queues are included in a generic Verilog module, called *WQ_BUF_SET*. Similarly, the *TRANSFER_UNIT* includes the Transfer Units of all cores.

GEORGE MATHEOU

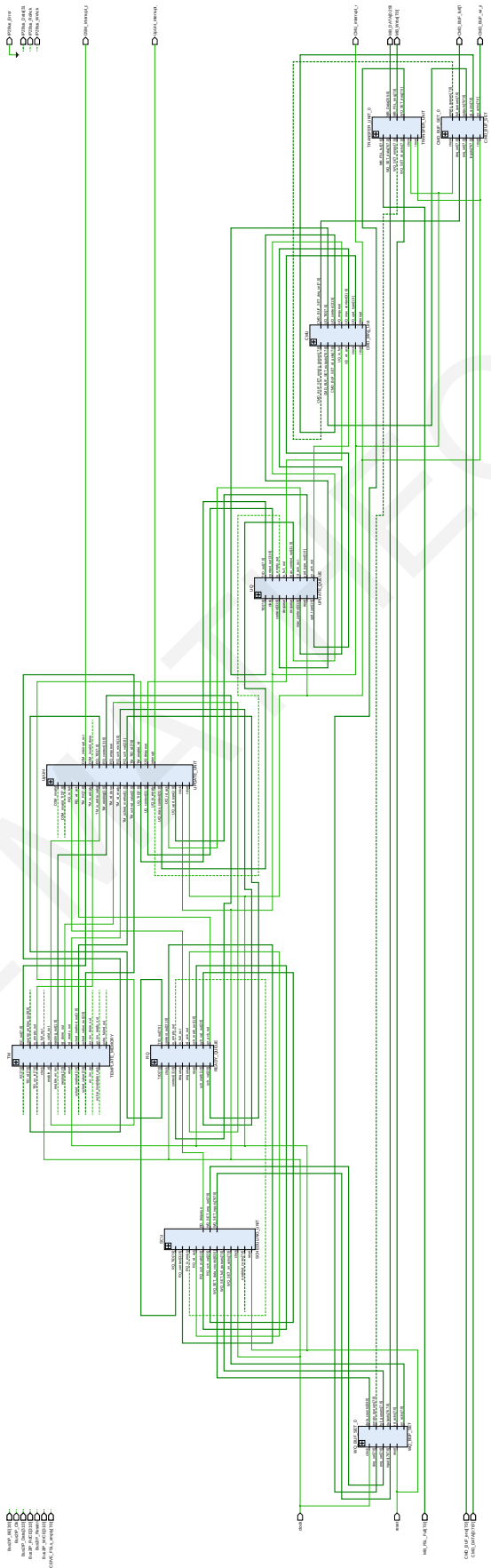


Figure 27: High-level RTL schematic of the TSU.

GEORGE MASON

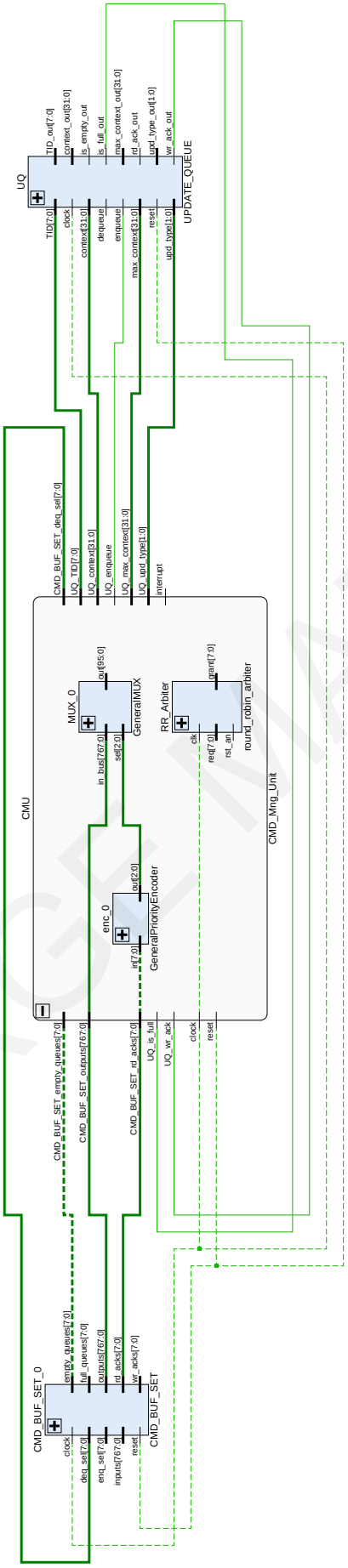


Figure 28: High-level RTL schematic of the Fetch Unit and the Update Queue.

GEORGE MATHEOU

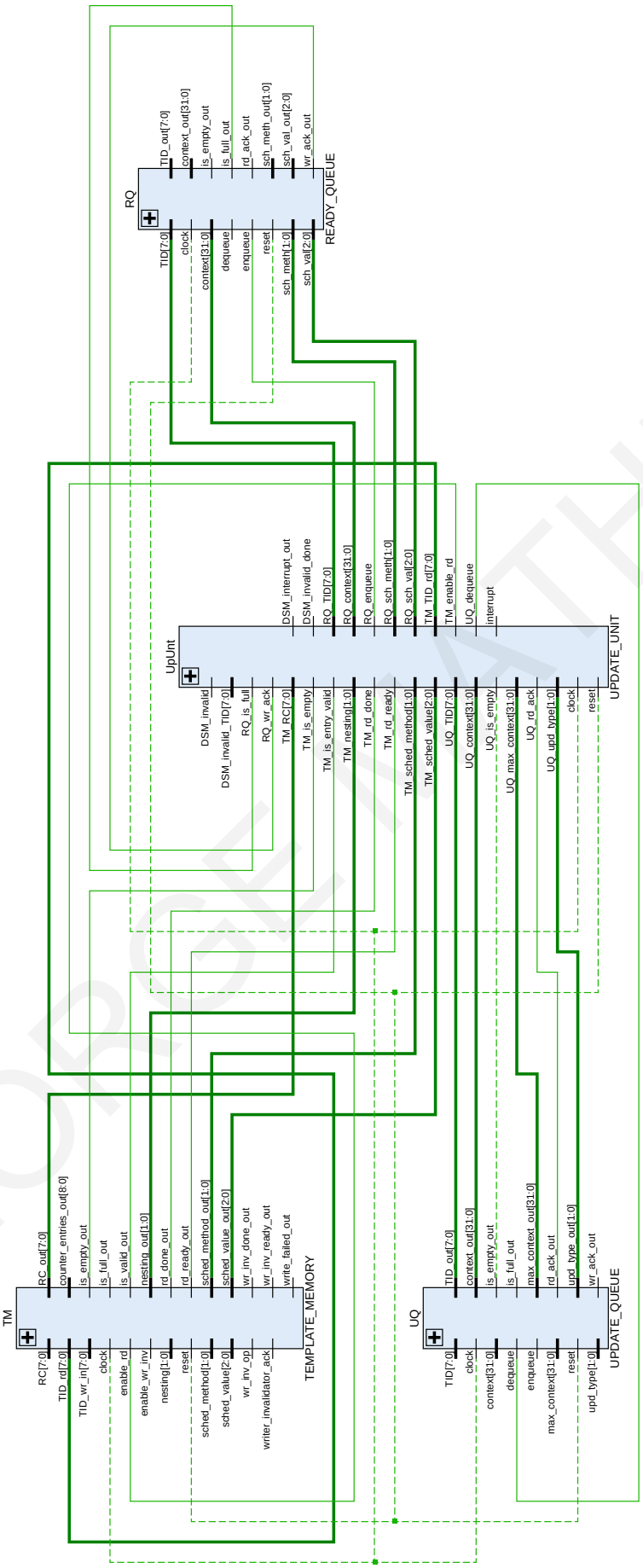


Figure 29: High-level RTL schematic of the Update Unit connected with Template Memory, Update Queue and Ready Queue.

GEORGE ATHEOU

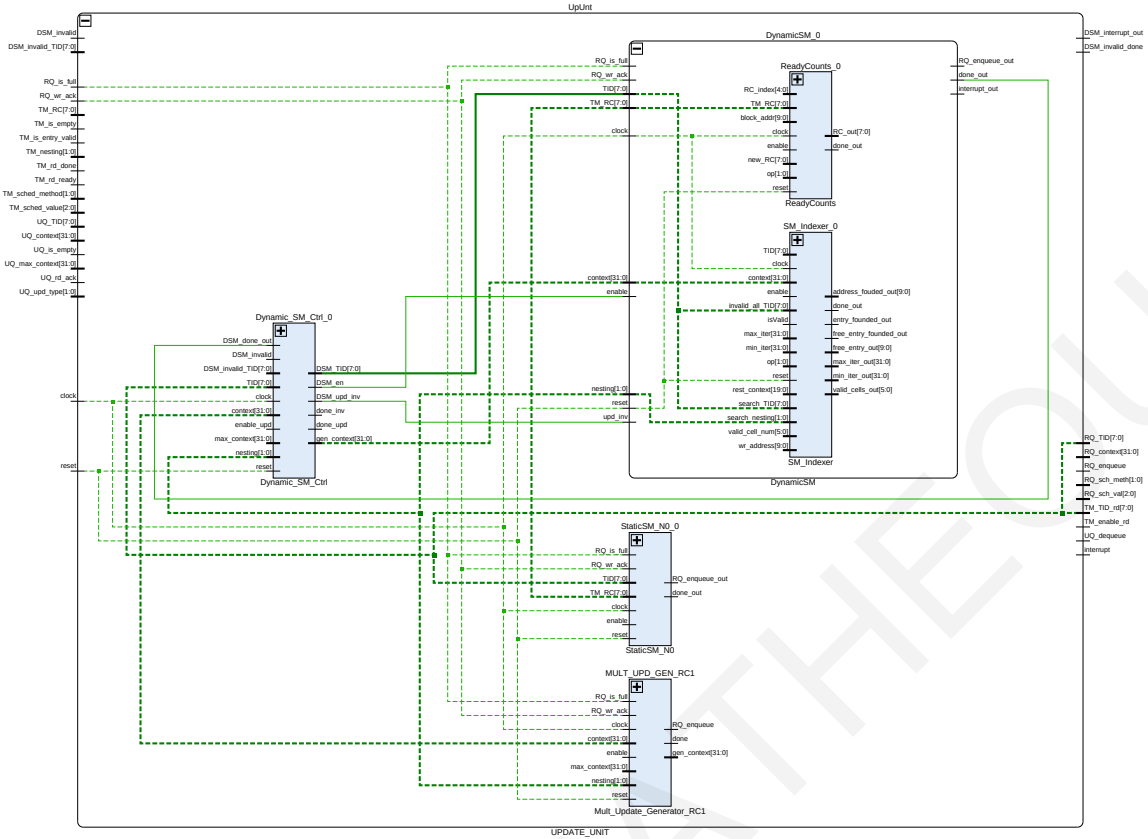


Figure 30: High-level RTL schematic of the Update Unit.

GEORGE MATHEOU

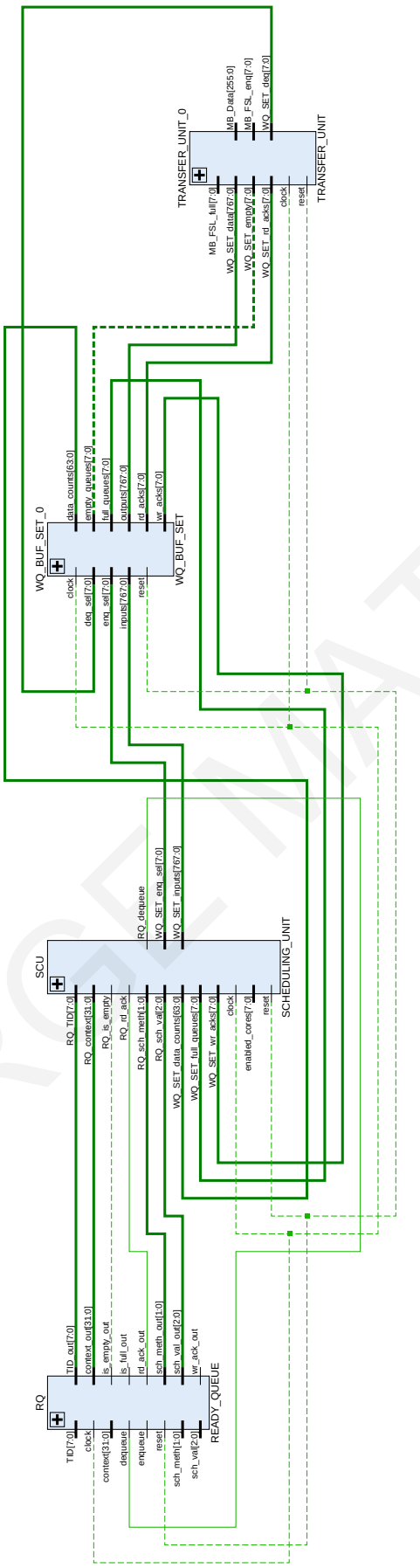


Figure 31: High-level RTL schematic of the TSU's output side (Ready Queue, Scheduling Unit, Waiting Queues and Transfer Units).

GEORGE MATHEOU

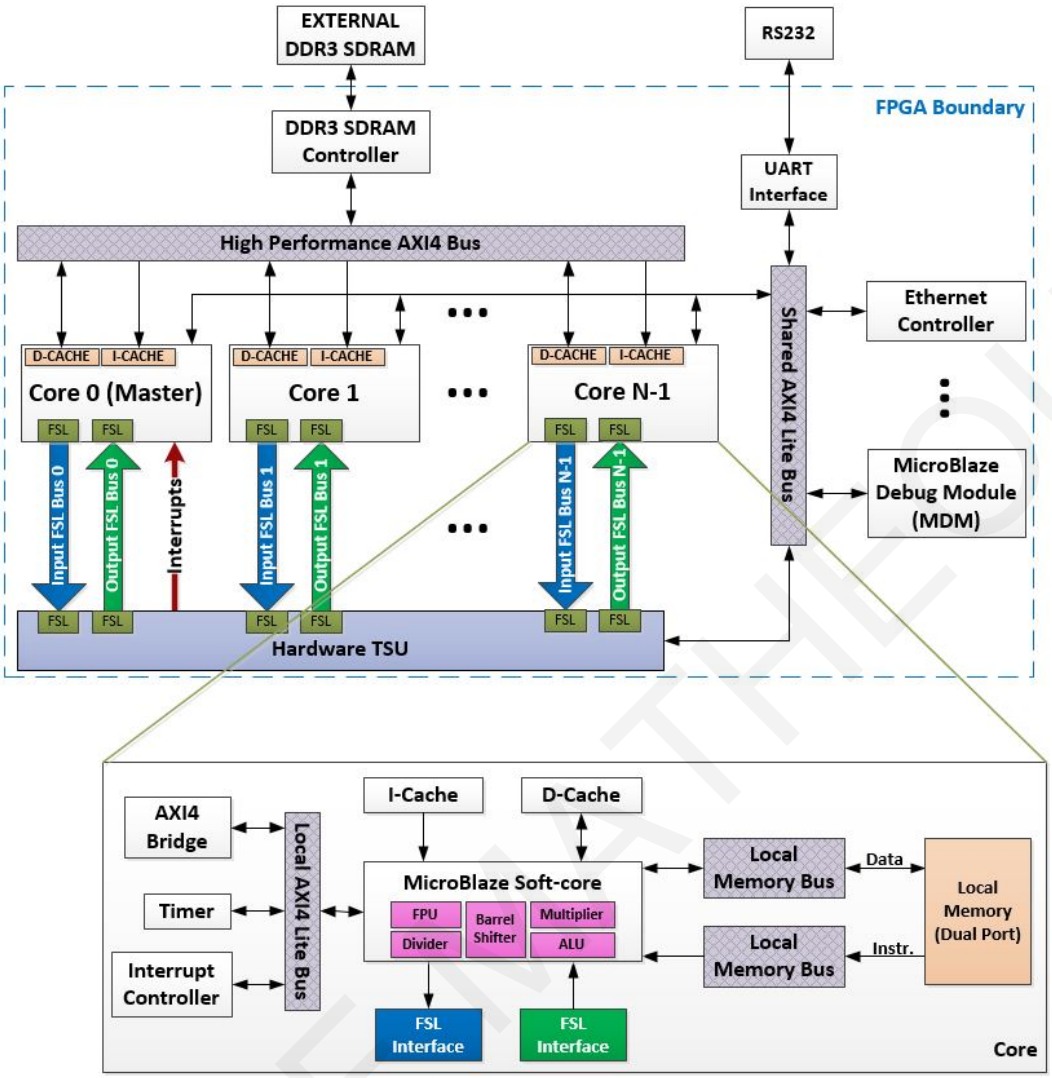


Figure 32: MiDAS architecture supporting an arbitrary number of cores.

3.3 MiDAS System Architecture

MiDAS is a multi-core processor consisting of non-coherent in-order cores and an optimized hardware TSU implementation that we proposed in Section 3.2. As a proof of concept, we have prototyped MiDAS on FPGA devices using the Xilinx Platform Studio (XPS) 14.7 tool. XPS allows hardware designers to develop embedded processor-based systems. The MiDAS architecture that supports an arbitrary number of cores is shown in Figure 32. In this work we implemented MiDAS's cores using Xilinx MicroBlaze [62], a 32-bit RISC Harvard soft-core. In its basic configuration each MicroBlaze core is configured with 32-KB L1 Data Cache (D-CACHE), 32-KB L1 Instruction Cache (I-CACHE), and 4-KB Local Memory. The caches and Local Memory are implemented using Block RAM (BRAM) [62]. It is important to note that MiDAS can be implemented with any non-coherent processing element that supports memory-mapped communication buses such as AXI or AMBA [154], and/or FIFO-based communication buses such as FSL [153]. In this work we choose

to incorporate MicroBlaze cores since they are readily provided directly from Xilinx, while being highly configurable soft-cores. Also, MicroBlaze cores can be programmed in both C and C++ , an important feature which proves handy in the implementation of our programming interface.

The cores share a DDR3 SDRAM Controller via a shared high-performance AXI4 bus that provides access to a DDR3 SDRAM chip (see top side of Figure 32). The cores also share a set of peripherals through a shared AXI4-Lite bus. These peripherals comprise, among others, a UART interface to access an RS-232 port, a MicroBlaze Debug Module (MDM) [62] which enables JTAG-based debugging to one or more MicroBlaze cores, and an Ethernet controller (currently not utilized). The shared AXI4-Lite bus is also used for establishing communication between the TSU and the cores. Each core has a local AXI4-Lite bus which enables the usage of these three peripherals: an interrupt controller, a timer and an AXI4 bridge. The AXI4 bridge provides access to the shared peripherals, i.e., MDM, UART, etc. One of the cores is selected to act as the master core (usually Core 0) which receives interrupts initiated by the TSU. The interrupt controller in each core is used to receive interrupts from local devices such as the timer and the FSL buses. Finally, each core has two FSL buses that function as the TSU's Input and Output FSL Buses.

3.3.1 Memory Model

In our memory model, concurrent DThread instances cannot modify the same data since this would result in a data dependence violation. This allows MiDAS to implement an efficient *single-writer/multiple-readers* model based on data-flow [27], where synchronization constructs (e.g., locking) and cache coherency protocols are not required. Correctness of the application can be assured by updating cached data to main memory on completion of a DThread instance. This can be achieved by flushing updated values (output data) to memory and then activating the consumer DThread instances. The DDM semantics, which are implemented in the hardware TSU, guarantee that the consumer DThreads instances will be activated only after the producer DThread instances terminate (using Update commands). The fact that MiDAS can be implemented without cache-coherency allows for increasing performance scalability, reducing hardware costs and improving energy-efficiency.

GEORGE MATHEOU

Chapter 4

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

4.1 Introduction

FREDDO (efficient Framework for Runtime Execution of Data-Driven Objects) [142, 156] is an efficient and portable object-oriented implementation of the Data-Driven Multithreading (DDM) model [55]. It is a C++ framework that supports efficient data-driven execution on conventional single-node and distributed multi-core systems. The main contributions of the FREDDO framework can be summarized as follows:

1. Provide an efficient and portable distributed implementation of the DDM model. The FREDDO's distributed implementation is based on the DDM-VM system [11, 63].
2. Provide recursion support for the DDM model. Our mechanisms/techniques are presented in Chapter 6.
3. Extend the programming interface of DDM with new features (object oriented programming, larger Context sizes, DFunctions, automatic computation of RC values, etc.).
4. The evaluation of the DDM model on HPC systems. Particularly, FREDDO was evaluated on an open-access 64-node Intel HPC system with a total of 768 cores. The DDM model was previously evaluated only on very small distributed multi-core systems with up to 24 cores, using the DDM-VM implementation [63].
5. The comparison of the results obtained from FREDDO and four parallel software platforms: OpenMP [19], OmpSs [157, 40, 129, 130], MPI [18] and DDM-VM [63]. The comparison results show that FREDDO achieves similar or better performance.
6. Provide simple mechanisms/optimizations to reduce the network traffic of distributed DDM applications.

GEORGE MATHEOU

7. The implementation of a connectivity layer with two different network interfaces: a Custom Network Interface (CNI) and MPI [18]. The CNI support allows to have a direct and fair comparison with frameworks that also utilize a custom network interface (e.g., DDM-VM [63]) where the MPI support provides portability and flexibility to the FREDDO framework. We also provide comparison results between CNI and MPI for several benchmarks.

In this chapter we present the single-node and distributed FREDDO implementations, in Sections 4.2 and 4.3, respectively. Chapter 5 describes the FREDDO's programming methodology. Finally, a comprehensive evaluation of FREDDO is provided in Chapter 7.

4.2 Single-node Implementation

FREDDO is a software runtime system that supports efficient data-driven execution on conventional multi-core systems. The scheduling of DThreads is managed by a software TSU implementation which runs on one of the cores of the system. Like the previous software DDM implementations [9, 35, 10], FREDDO abstracts the details of the underlying machine and it handles DThread execution and data management implicitly, by providing two additional components: the Kernels and the Runtime system. The Runtime system runs on top of any commodity Unix-based Operating System (OS) and hides all the details of a DDM implementation (e.g., the TSU module).

FREDDO applications are developed using C++11 [158] and FREDDO's API. The API includes a set of runtime functions and classes which are grouped together in a C++ namespace called *ddm*. A user is able to create and manage DThreads by creating and accessing objects of special C++ classes. FREDDO's front-end and back-end are implemented in C++11 in order to take advantage of its new features, such as, range-based loops, initializer lists, Lambda expressions and atomic operations. A FREDDO program is compiled by a commodity C++ compiler, thus an executable binary for any ISA can be generated. Furthermore, FREDDO applications are composed of DThread objects that have producer-consumer relationships. A DThread object holds the Thread Template (e.g., Ready Count, Consumers, Nesting attribute, IFP, etc.) of a specific DThread and provides methods for decrementing the RC value of the DThread's consumers. The TSU uses the Thread Templates to schedule DThread instances based on data-availability. This is achieved by scheduling a DThread instance for execution when all its producer-instances finish their execution.

4.2.1 New features

In this subsection we describe the new features provided by FREDDO in order to extend the programming interface of the DDM model.

GEORGE MATHEOU

4.2.1.1 Extending the size of the Context Attribute

The Context size is important when we have loops with large indexes since the indexes are stored in the Context values. The Context attribute of DDM was encoding up to three nesting levels in a 32-bit long word. Thus, it was difficult to have nested loops with large indexes. For example, consider the Tile LU Decomposition algorithm (Figure 33) which can be parallelized in DDM using five DThreads and 32-bit Context values. *T1* implements the outermost loop of the algorithm while the rest DThreads are responsible for executing four basic operations/kernels (*diag*, *front*, *down* and *comb*). Details about the algorithm can be found in Section 5.3.3. The DThreads have different Nesting values which are specified by the loop nesting level. For instance, *T1* and *T2* have Nesting=1 while *T5* has Nesting=3.

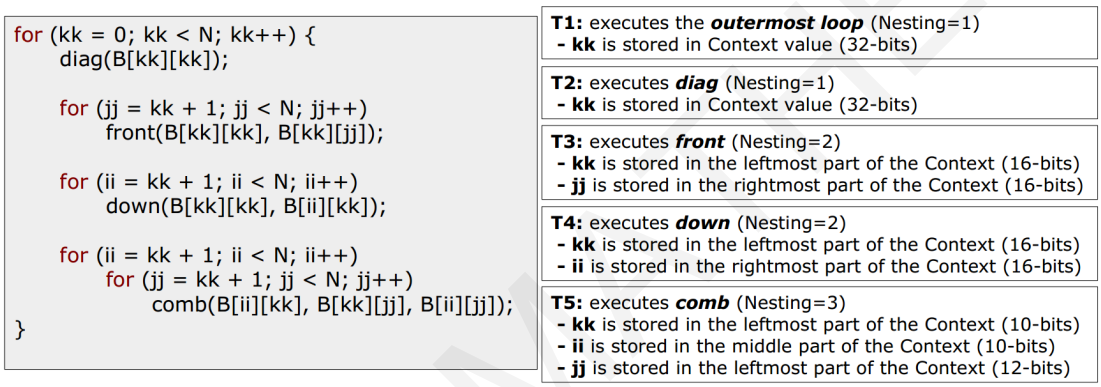


Figure 33: LU algorithm: DThreads and Context values.

The Context value of *T5* consists of three parts: left (10-bits), middle (10-bits) and right (12-bits). Thus, the left and middle parts can store indexes < 1024 and the right part can store indexes < 4096. This prohibits the programmer to execute the algorithm with large matrix sizes and small or medium tile sizes. For example, for a $32K \times 32K$ matrix size and 16×16 tile size or for a $64K \times 64K$ matrix size and 32×32 tile size, N is equal to 2048. Since, the upper limit of the indexes is the N value, a DDM system with 32-bit Context values is not able to support the LU benchmark with these problem sizes. Although these problem sizes are very large for a single-node multi-core system, are normal for a cluster with a large number of nodes/cores. Notice that, the smaller the tile size (i.e., fine-grained threads), the larger the number of DThread instances are spawned during the execution. In HPC systems fined-grained threads (e.g., with small tile sizes) are usually used in order to utilize the large number of computation cores in a better way.

FREDDO solves the aforementioned issue by supporting four different Context sizes: 32-bit, 64-bit, 96-bit and 192-bit. Table 4 describes how the indexes of the loops are encoded into the Context value for all possible combinations, for each Context size. Notice that for Nesting-0 the Context value is always zero and its width is equal to the Context size.

Context Size	Nesting Attributes					
32-bit	Nesting-1	Nesting-2		Nesting-3		
	Value	Outer Value	Inner Value	Outer Value	Middle Value	Inner Value
	32-bits	16-bits	16-bits	10-bits	10-bits	12-bits
64-bit	Nesting-1	Nesting-2		Nesting-3		
	Value	Outer Value	Inner Value	Outer Value	Middle Value	Inner Value
	64-bits	32-bits	32-bits	16-bits	16-bits	32-bits
96-bit	Nesting-1	Nesting-2		Nesting-3		
	Value	Outer Value	Inner Value	Outer Value	Middle Value	Inner Value
	32-bits	32-bits	32-bits	32-bits	32-bits	32-bits
192-bit	Nesting-1	Nesting-2		Nesting-3		
	Value	Outer Value	Inner Value	Outer Value	Middle Value	Inner Value
	64-bits	64-bits	64-bits	64-bits	64-bits	64-bits

Table 4: Context encoding according to the Nesting attribute for each Context size.

4.2.1.2 Introducing dynamically allocated data-structures in TSU

The data-structures of the previous TSU modules were implemented using static memory allocation. This requires the programmer to recompile the TSU code when: (1) the number of Kernels has to be changed, (2) the queues that hold information about the executed DThread instances are full and (3) the Synchronization Memory (SM) is full. Recall that SM is responsible for holding the RC values of DThread instances. This increases the development time and affects usability. FREDDO addresses this issue by implementing the majority of the TSU's data-structures using dynamic memory allocation.

4.2.1.3 Reducing the memory allocated by the SM module

In the previous DDM implementations, an RC value is allocated for each instance of each DThread. This results in allocating thousands or even millions of RC values in the SM unit. We observed that it is not necessary to allocate RC values for DThreads that have RC value equal to one. The instances of such DThreads can be scheduled immediately for execution when Update operations are received for them. This approach reduces the memory usage of DDM applications as well as it accelerates the Update operations. As an example, consider a for-loop with 10^6 iterations that is mapped to a DThread with RC=1. In this case our framework will avoid allocating 10^6 RC values. Furthermore, this approach is vital for hardware DDM implementations [55, 9, 159, 141] where the size of the SM is limited.

4.2.1.4 Object-Oriented Approach

DDM applications are developed using C macros [35, 10] or TFlux directives [9]. FREDDO extends the DDM's programming interface by allowing the development of DDM applications through the object-oriented programming (OOP) paradigm. We are providing four basic C++ classes for creating, updating and removing DThreads: *SimpleDThread*, *MultipleDThread*, *MultipleDThread2D* and

GEORGE MATHEOU

MultipleDThread3D. These classes correspond to DThreads with Nesting-0, Nesting-1, Nesting-2 and Nesting-3, respectively. OOP provides the following benefits to the DDM programs:

- *Data Encapsulation*, i.e., the binding of data and functions that manipulate the data, which keeps both safe from outside interference and misuse [160, 161]. FREDDO supports the properties of encapsulation and *information hiding* through the DThread classes.
- *Data Abstraction*, i.e., providing only essential information to the outside world and hiding their background details [161]. For example, the users can send only Update operations to the DThread objects and their consumers while the details of the communication between the DThread objects and the TSU are hidden.
- *Inheritance*, i.e., the objects can acquire the properties of objects of other classes. This provides re-usability and reduces the implementation time. FREDDO uses inheritance in order to organize the DThread objects into a hierarchy. The FREDDO's DThread classes are derived-classes of the *DThread* class.

The OOP is also used for developing the FREDDO's back-end, i.e., the TSU and the Runtime system, in order to improve productivity and maintainability as well as to reduce the development time. This is because OOP provides modularity, extensibility and re-usability.

4.2.1.5 Introducing DFunctions

In the previous software DDM systems the code of all DThreads had to be placed in the same function/place. This is because the runtime support of these systems uses *label* and *goto* statements for executing the code of the DThreads. Thus, programmers were restricted from having parallel code in different files as well as in different functions. In FREDDO, the code of DThreads can be embodied in any callable target (called *DFunction*) like: (i) standard C/C++ functions, (ii) Lambda expressions and (iii) functors. This allows the DThreads' code to be placed anywhere in a DDM program. Each DFunction has one input argument, the Context value. Different Context structures (ContextArg, Context2DArg and Context3DArg) are provided based on the type of the DThread class. Notice that the DFunctions of DThreads with Nesting-0 (i.e., SimpleDThreads) do not have input arguments since the Context value is always zero. DFunctions with different Context structures allows a more clear interface.

4.2.1.6 Supporting Recursion

FREDDO is the first DDM implementation that supports recursion. It provides special C++ classes in order to allow programmers to parallelize recursive algorithms (linear, tail, binary, multiple, etc.). The recursion support for the DDM model is presented in Chapter 6.

GEORGE MATHEOU

4.2.1.7 Automatic computation of the RC values

In the previous DDM systems the RC value of each DThread was required to be specified by the programmer. In this work we allow two different approaches for specifying the RC values of the DThreads:

1. Like the previous DDM systems, the RC values are given by the programmers. For this purpose the basic DThread classes have to be used (e.g., SimpleDThread, MultipleDThread, etc.).
2. The RC values will be computed at runtime based on the producer-consumer relationships of the DThreads. This feature is supported by introducing a special data-structure in TSU, the Pending Template Memory (PTM). Additionally, special DThread classes are provided, called FutureDThread classes: *FutureSimpleDThread*, *FutureMultipleDThread*, *FutureMultipleDThread2D* and *FutureMultipleDThread3D*. These classes are derived-classes of the basic DThread class and do not require an RC value to be specified in their constructors. Initially, the Thread Templates of the FutureDThreads will be stored in PTM and their RC values will be computed at runtime by the TSU module.

4.2.2 Architecture

FREDDO allows efficient DDM execution by utilizing three different components: the TSU, the Kernels and the Runtime support. The overall architecture of the single-node FREDDO implementation is depicted in Figure 34. The FREDDO's components are described below.

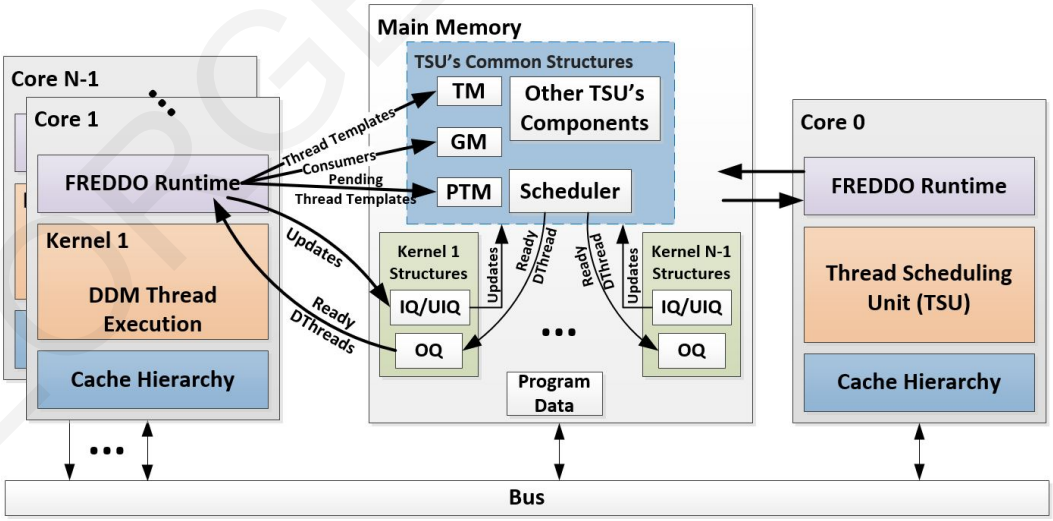


Figure 34: Architecture of the single-node FREDDO implementation.

4.2.2.1 Thread Scheduling Unit (TSU)

The block diagram of TSU is shown in Figure 35. The TSU is connected to a processor with an arbitrary number of cores, where core 0 is used to execute the TSU code while the other cores are

used for executing DThread instances. Each block of the diagram is a C++ object that may consist of several internal objects.

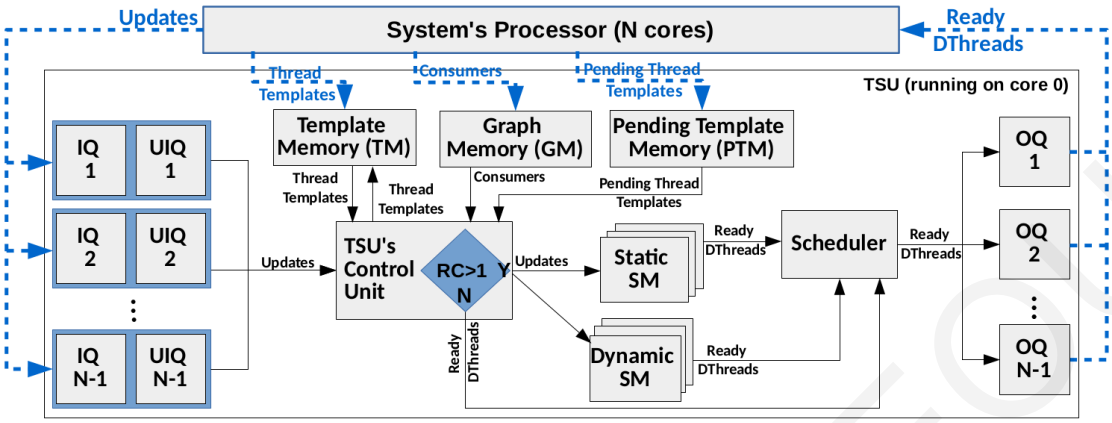


Figure 35: Block diagram of the FREDDO's TSU.

The TSU's storage units

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

- **StaticSM:** it is used when the number of instances of a DThread is known at compile time. Each instance allocates a unique entry in the StaticSM. The allocation of all RCs is performed at the time of creating the Thread Template. Accessing an RC entry at runtime is a direct operation that uses the Context of the DThread's instance.
- **DynamicSM**: it is used when there is no information about the number of instances of a DThread. A hash-map is used to allocate the SM entries where the *keys* are the Contexts and the *values* are the RCs. The allocation of RCs is performed as the execution proceeds. The hash-map is allocated at the time of creating the Thread Template. Moreover, accessing an RC entry is an associative operation.

Instead of having one global SM for each type (static/dynamic), we allocate a separate SM instance for each DThread, for two main reasons. Firstly, in the case of the StaticSM, we allocate exactly the amount of RCs that is required for each DThread. This is an improvement over the DDM-VM implementation which allocates redundant RCs if the loop indices don't start from 0 or if the

GEORGE MATHEOU

upper bound of the loop is not a power of 2 [11]. Secondly, in the case of the DynamicSM, we are using simpler/smaller key-value pairs. For instance, if we had a global DynamicSM, a possible *key* would be the tuple <Thread ID, Context> instead of <Context>. Furthermore, the rehashing operation of a DThread's DynamicSM will not cause the rehashing of the other DThreads' RCs.

Figure 36 depicts the basic structures of the TSU along with their fields and their types. Each entry of TM holds the IFP, RC and Nesting attributes as well as a pointer to the SM allocated for the associated DThread. In FREDDO, the IFP is implemented using the C++11's *std::function* in order to support the functionalities of DFunctions. Each PTM entry holds additional information that will help the TSU to allocate an SM structure after the calculation of the entry's RC value. This information includes the ranges of the loops (up to three-level nested loops) and the type of the SM (static or dynamic). Notice that the ranges (InnerRange, MiddleRange and OuterRange) are used when a StaticSM is required, in order to allocate the RCs of all instances at once.

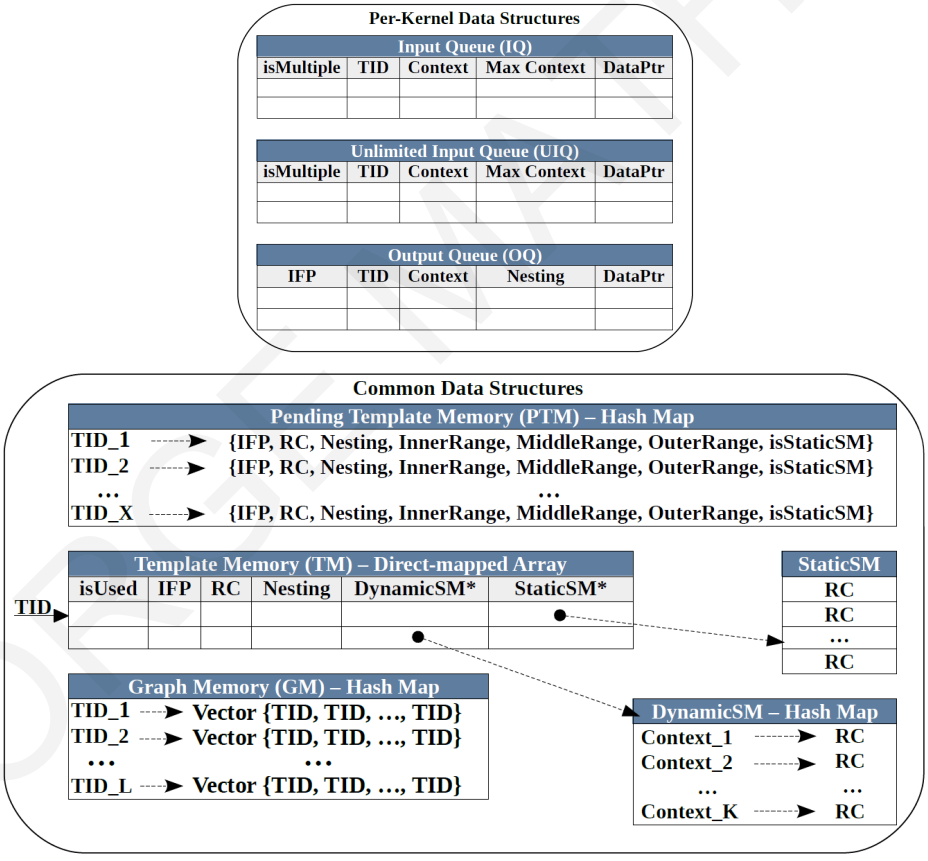


Figure 36: The TSU's basic data structures.

TSU-Cores Communication

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

the OQs. After a core completes the execution of a DThread instance, it sends Update commands (Single or Multiple) to the consumers of the completed DThread instance. A Single Update (isMultiple=0) consists of the Thread ID (TID) and the Context of the DThread instance that is going to be updated. Particularly, a Single Update operation indicates that the RC value that corresponds to the TID and Context attributes will be decreased by one. A Multiple Update includes an additional attribute, the Max Context, which allows to decrease multiple RC values of a DThread (from Context to Max Context). The Updates are stored in the core's IQ. If the IQ is full, then the Updates are stored in the associated UIQ. The UIQ is an efficient variable-length queue. Finally, the entries of IQs, OQs and UIQs include the DataPtr field which is used for the recursion support (for the RecursiveDThread and ContinuationDThread classes; see Section 6.4.2). DataPtr points to data (called *RData* object) of a specific recursive instance.

The TSU's Control Unit

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

Prior the DDM scheduling, the TSU fetches the Pending Thread Templates (PTTs) from PTM and computes their RC values based on the producer-consumer relationships of the DThreads. The algorithm performed by the TSU for computing the RC values is shown in Algorithm 1. Figure 37 illustrates an example of computing the RC values of four Future DThreads (with TIDs 1 to 4) where each Future DThread is associated with a Pending Thread Template (PTT). The example shows the procedure partitioned into three algorithmic steps as well as the final dependency graph with the calculated RC values.

GEORGE MATHEOU

Algorithm 1: Compute RC values at runtime

foreach *Pending Thread Template* ∈ *PTM* **do**
  Create an RC value, *R*
  R = 0
ConsDThreads ← holds the consumers of each DThread from GM
foreach *D* ∈ *DThreads* **do**
  foreach *Cons* ∈ *Consumers of ConsDThreads[D]* **do**
    if *Cons* ∈ *PTM* **then**
      Cons's *R* = Cons's *R* + 1
foreach *Pending Thread Template (PTT)* ∈ *PTM* **do**
  if *PTT's R* = *0* **then**
    PTT's *R* = 1
  Remove PTT from PTM and store it in TM
  if *PTT's R* > *1* **then**
    Allocate a new SM (StaticSM or DynamicSM)

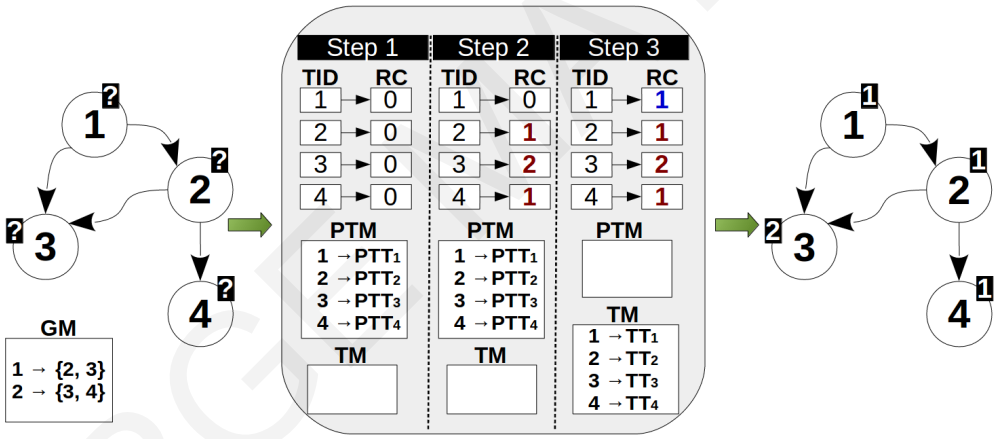


Figure 37: Example of computing the RC values of Pending Thread Templates (PTT=Pending Thread Template, TT=Thread Template).

Scheduling Ready DThreads

The Scheduler is responsible for assigning the ready DThread instances to the Output Queues (OQs). When it receives a ready DThread instance it locates the OQ with the least amount of work. In the case that all OQs have the same amount of work, the OQ of Kernel 0 will be selected. After that, the Scheduler will enqueue the information of the ready DThread instance into the selected OQ.

GEORGE MATHEOU

Memory allocation of the TSU's structures

The IQs and OQs are implemented as fixed-size circular buffers that are allocated statically at compile time in order to accelerate the enqueue and dequeue operations. The third fixed-size data-structure is the TM which is implemented as a direct-mapped array in order to provide efficient search operations. The other TSU's data structures are allocated dynamically at runtime. This avoids the need of recompiling the TSU code in the cases we mentioned in Section 4.2.1.2.

4.2.2.2 FREDDO Kernels

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

4.2.2.3 Runtime Support

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

4.3 Distributed Implementation

In this section we describe the FREDDO's distributed architecture and its memory model, scheduling and termination mechanisms, network support, and techniques that are used for reducing the network traffic in the system.

4.3.1 Architecture

The distributed architecture of FREDDO is depicted in Figure 38. It is composed by multi-core nodes connected by a global network interconnect (e.g., Ethernet, InfiniBand, etc.). A *Network Manager* is implemented in each node, that abstracts the details of the network interconnect and allows the inter-node communication. FREDDO's runtime system was extended to: 1) handle the communication and data management across the nodes, 2) manage the applications' dependency graphs in a distributed environment and 3) schedule/execute the ready DThread instances on the

GEORGE MATHEOU

cores of the entire distributed system. The same application binary is executed on all nodes where one of them is selected as the *RootNode*. The *RootNode* is responsible for detecting the termination of the distributed FREDDO applications and for gathering the results for validation purposes (this is optional).

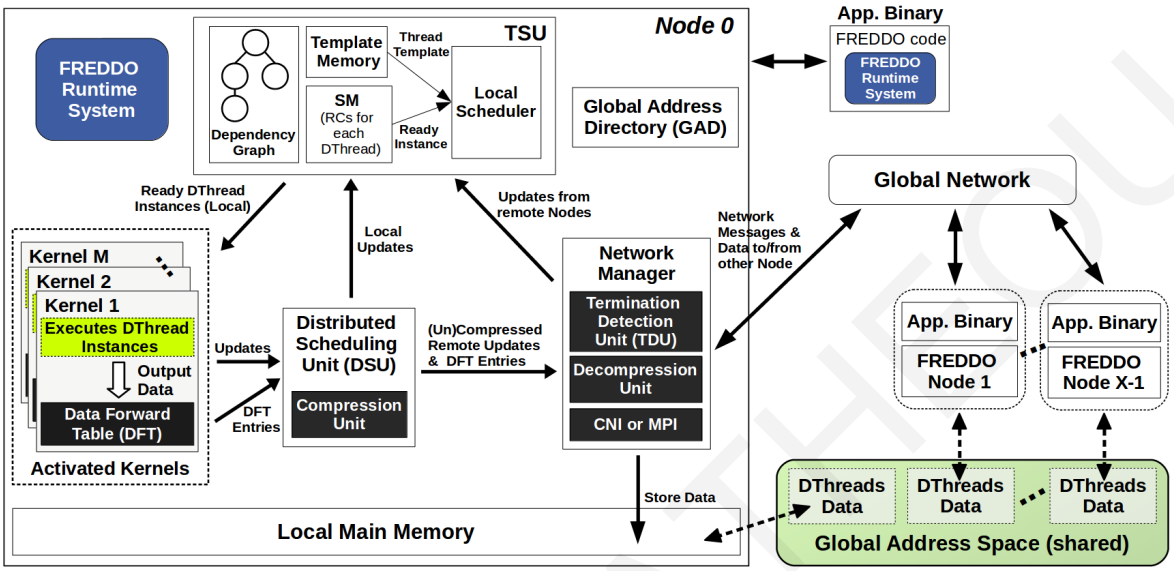


Figure 38: The FREDDO's Distributed Architecture.

4.3.2 Memory Model

FREDDO implements a software Distributed Shared Memory (DSM) system [162] with a shared Global Address Space (GAS) support. Part, or all of the main memory space on each node, is mapped to the DSM's GAS. This approach creates an identical address space on each node, which gives the view of a single distributed address space. The conventional main memory addresses of shared objects (scalar values, vectors, etc.) are registered in the GAS by storing them in the *Global Address Directory (GAD)* of each node. For each such address, the runtime assigns to it a unique identifier (called *GAS_ID*) which is identical in each node. This allows the runtime system to transfer data between nodes (from one local main memory to another) using *GAS_IDs*.

FREDDO uses the DSM implementation to employ implicit *data forwarding* [163]. The produced/output data of a DThread instance is *forwarded* to its consumers, running on remote nodes, before the latter start their execution. This is guaranteed by sending Update operations after data transfers are completed. To support this functionality, each Kernel is associated with a Data Forward Table (DFT). A DFT keeps track of output data segments of the currently executed DThread instance. The DFT allocates a separate entry for each output data segment. When a DThread instance finishes its execution, the DFT entries related to the DThread instance are removed from the associated DFT.

A DFT entry consists of the following attributes: *GAS_ID*, *addrOffset* (the offset in bytes from the conventional address which maps to the *GAS_ID*), *segmentSize* (the size in bytes of the data segment)

GEORGE MATHEOU

and *sentToTable* (marks the nodes that have already received the data segment). *sentToTable* is used to send a data segment to a node only once. The runtime uses DFT entries to transfer produced data to remote nodes, implicitly. Algorithm 2 depicts the basic algorithm of sending Updates and output data of a DThread instance to remote nodes. Expensive coherence operations implemented in typical DSM systems [164] are not required. The remote read operations are eliminated, thus, the total communication cost can be reduced. Coherence operations are applied only within each node's memory hierarchy, by hardware, since each node is a conventional multi-core processor. It is important to notice that applications, where tasks write to the same data simultaneously, without specifying dependencies, result in an undefined behaviour.

Algorithm 2: Sending Updates and output data of a DThread instance *DI* which is executed on Kernel *k*

```
foreach Upd ∈ Updates of DI code do
    i ← the id of the node that will execute Upd
    if i = local node id then
        Send Upd to local TSU
    else
        foreach dftEntry ∈ DFT of Kernel k do
            if dftEntry.sentToTable[i] = false then
                sendData(i, dftEntry.GAS_ID, dftEntry.addrOffset, dftEntry.segmentSize)
                dftEntry.sentToTable[i] ← true
        Send Upd to the node with id=i
```

DSM eases the development of distributed FREDDO/DDM applications that use shared objects/data-structures (e.g., scalar values, arrays, etc.). Programmers only need to register shared objects of single-node FREDDO applications in GAS and specify the output data of each DThread, using special runtime functions. For algorithms without shared objects, FREDDO allows data forwarding through *data objects* which are exchanged between the nodes. This approach has been used for FREDDO's distributed recursion support, where objects (called *DistRData*) are used to transfer the arguments and return values of recursive function calls. In this case, the DSM implementation and the DFTs are not used. FREDDO sends Update operations to remote nodes after transfers of the data objects are completed.

Currently, the FREDDO's DSM implementation requires the shared objects to have the same memory size in each node (e.g., the tile matrix in a tile algorithm like Cholesky [26]). This simplifies the implementation of the proposed programming model but it limits the total amount of memory used by a DDM program. Particularly, a program can use only as much memory as is available in the *RootNode* since the output results are gathered in that node. This approach is also adopted by DDM-VM [63] and OmpSs@Cluster [129, 130]. Future work will focus on mechanisms that will

GEORGE MATHEOU

overcome this limitation in order to allow FREDDO to execute real life applications with enormous input size (e.g., big data applications) on large-scale supercomputers.

4.3.3 Distribution Scheme and Scheduling Mechanisms for DThread instances

FREDDO provides a lightweight *distribution scheme*, based on the DDM's tagging system [55], to distribute DThread instances on the system's nodes. In particular, FREDDO implements a *static scheme* in which the *mapping* of DThread instances to the nodes is determined at compile time, based on their Context values (tags). The node in which a DThread instance will be executed is defined by the following formula: $node\ id = f_{cn}(Cntx\ \%\ totNumCores)$, where $Cntx$ is the Context value of the DThread instance and $totNumCores$ is the total number of cores of the entire system. f_{cn} returns the node id of a core (e.g., in a 4-node system with 4-cores per node, $f_{cn}(0) = 0$ and $f_{cn}(15) = 3$). The *static scheme* only specifies *where* a DThread instance will be scheduled for execution. However, DThread instances are scheduled at runtime, based on data-availability (the DThread instances are dynamically created). This approach simplifies the scheduling and data management operations. Also, it reduces runtime overheads.

FREDDO utilizes two different scheduling mechanisms for executing DThread instances, *intra-node* and *inter-node*, outlined next.

4.3.3.1 Inter-node Scheduling Mechanism

The *inter-node* mechanism is handled by the *Distributed Scheduling Unit (DSU)* which decides, based on the FREDDO's *distribution scheme*, if Update operations and the output data of a Producer-DThread instance will be forwarded to a remote node (or nodes) or in the local node. In the former case, the runtime will send Updates operations and the output data as network messages, via the FREDDO's Network Manager, to the corresponding remote node(s). In the latter case, Update operations are sent to the local TSU through the IQs/UIQs of the Kernels.

4.3.3.2 Intra-node Scheduling Mechanism

The *intra-node* mechanism is handled by the TSU (Section 4.2.2.1) in each node. The TSU fetches local Updates from the Kernels, through the DSU, or remote Updates from the Network Manager. For the distributed FREDDO implementation an additional IQ/UIQ pair was introduced in the TSU architecture in order to store the remote Updates coming from the Network Manager. TSU executes the Update commands and determines the ready DThread instances (their RC=0). The TSU's Scheduler (or Local Scheduler) distributes the ready DThread instances to the Kernels (through the OQs) for execution. Like in the single-node execution, Local Scheduler selects the OQ with the least amount of work.

GEORGE MATHEOU

4.3.4 Network Manager

The Network Manager is responsible for handling the inter-node communication. It is implemented as a software module that relies on the underlying network hardware interface. Network Manager has the following responsibilities:

1. Establishes connections between the system's nodes.
2. Exchanges network messages between the nodes. The network messages carry Update Operations, Termination Tokens (used for the Distributed Termination), Data Descriptors (hold information about data segments that will be forwarded to consumer nodes), Shutdown Acknowledgements (used for a graceful system termination), etc.
3. Processes incoming network messages appropriately (e.g., it sends Updates to the local TSU).
4. Supports data forwarding across the global address space.

4.3.4.1 Connectivity Layer

The Network Manager handles the low-level connectivity by utilizing two different network interfaces: a Custom Network Interface (CNI) which is an optimized implementation with TCP sockets, and the widely used MPI library [18]. CNI implements a fully connected mesh of inter-node connections (i.e., each node maintains a connection to all other nodes). Currently, CNI supports only Ethernet-based interconnects. As a first step, we have implemented the CNI in order to identify all the required functionalities and mechanisms that are needed for the inter-node communication. Based on these functionalities/mechanisms we have implemented the Network Manager on top of MPI. The major benefits of providing MPI support are portability and flexibility. On the other hand, the CNI implementation allows to have a direct and fair comparison with similar frameworks that utilize a custom network interface.

4.3.4.2 Sending/Receiving Functionalities

The Network Manager tolerates network communication latencies by overlapping its sending/receiving functionalities with the DThread instances' execution and the TSU's functionalities. Sending functionalities, i.e., operations for sending commands and produced data, are handled by the Kernels. This removes the cost of such operations from the TSU's critical path. However, this can lead to race conditions since multiple DThread instances can send messages to the same destination at the same time. To avoid this situation while keeping the performance in high levels, we are using *atomic variables* as much as possible and synchronization constructs (lock/unlock) to the bare minimum. Notice that a sending operation returns when the message was stored in the network layer buffers of the OS. For receiving functionalities, an *auxiliary thread* is used, which continuously retrieves incoming network messages from the other nodes. For this purpose, the *pselect* routine is used

GEORGE MATHEOU

for the CNI implementation. For the MPI implementation we are using the *MPI_Recv* routine with *source=MPI_ANY_SOURCE*.

4.3.5 Distributed Execution Termination

Detecting termination of data-driven programs in distributed execution environments it's not a straightforward procedure, since the availability of data governs the order of execution. In this work we have implemented an implicit distributed termination algorithm based on the Dijkstra and Scholten's parental responsibility algorithm [165, 166], which requires minimal message exchange. The algorithm assumes termination when: *the state of all the nodes is passive (idle) and no messages are on their way in the system*. In our implementation, the state *passive* refers to the state when the TSU has no pending Update operations and pending ready DThread instances that are waiting for execution.

When the parent node (*RootNode*) detects termination, it broadcasts a termination message to the other nodes and waits for their acknowledgements in order to have a graceful system termination. The distributed termination algorithm is implemented by the *Termination Detection Unit (TDU)* which keeps track of the incoming and outgoing network messages in each node. We choose an implicit distributed termination detection algorithm in order to reduce the programming effort. The same algorithm was adopted by DDM-VM [11, 63]. The main difference between the two implementations is that FREDDO uses *atomic variables* to count the number of outgoing and incoming messages in each node. In DDM-VM, the same functionality is implemented using lock/unlock operations which incur more overheads. The algorithm is described below:

- Every node maintains a message counter (*MC*) which is incremented when a network message is sent and decremented when a network message is received. The sum of *MCs* on all the nodes represents the number of pending messages in the network.
- The algorithm uses a *Termination Token* for detecting termination, which is exchanged between the nodes. A color is assigned to each node and *Termination Token* where initially all are white. When a node receives a network message, the node's color=black. When a node forwards the *Termination Token*, the node's color=white.
- When RootNode is idle, it initiates a termination probing, i.e., it sends the *Termination Token* with value=0 and color=white to node N-1. The RootNode's color=white.
- Every node *i* keeps the *Termination Token* until it becomes idle; it then sends the *Termination Token* to node *i-1* increasing the *Termination Token*'s value by *MC*. Additionally, if the color of node *i* is black, the *Termination Token*'s color=black, otherwise the *Termination Token* keeps its color. Finally, the color of node *i* becomes white.

GEORGE MATHEOU

- When RootNode receives the *Termination Token* the termination probing is finished for one ring-round. In that case, the algorithm detects termination if: (1) RootNode is idle, (2) the *Termination Token*'s color=white, (3) the RootNode's color=white and (4) the *Termination Token*'s value + *MC* = 0. Otherwise, RootNode initiates a new termination probing (when it will become idle).

4.3.6 Reducing Network Traffic

Reducing the network traffic in HPC systems is critical since it can help to avoid network saturation and reduce the power consumption. The US Department of Energy (DOE) [33, 167, 168] clearly states that the biggest energy cost in future massively parallel HPC systems will be in data movement, especially moving data on and off chip. To this end, we are recommending four simple and efficient techniques for reducing the network traffic in distributed DDM applications running on HPC systems. These techniques are mostly applied to the Update operations which are the most frequent commands executed in a DDM application.

4.3.6.1 Use General Network Packets

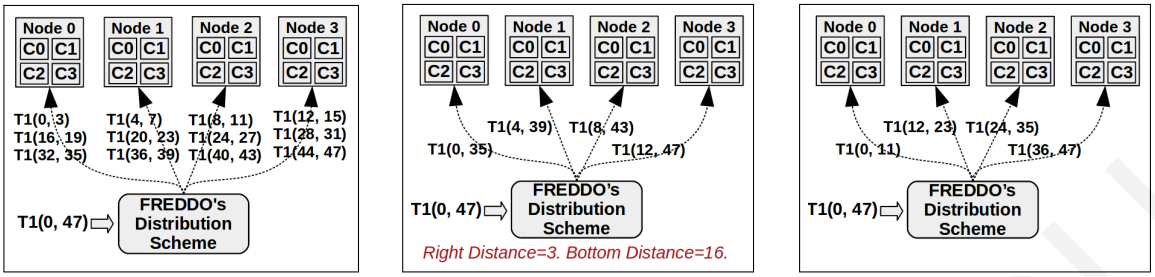
A network message can carry any type of command (Update, Multiple Update, Data Descriptor, etc.). The most common practice to send such a message is to use a header (as a separate message) that describes the message's content, like in [63]. In this work we are introducing a *general packet* with four fields (Type=1-byte, Value_1=4-bytes, Value_2=*sizeof*(Context Value) and Value_3=*sizeof*(Context Value)) that can carry all the basic types of commands. As a result, the number of sending network messages can be halved.

4.3.6.2 Compressing Multiple Updates for DThreads with RC >1

Multiple Updates decrement the RC values of several instances of a DThread. Since the mapping of instances to the nodes is based on their Context values, a Multiple Update should be unrolled and each of its Updates should be sent to the appropriate node. Figure 39a shows an example where a Multiple Update, with Contexts from <0> to <47>, is distributed to a 4-node system (each node has 4 cores). We are compressing consecutive Multiple Updates that are sent to the same node based on a simple pattern recognition algorithm. The algorithm takes into account the difference between the minContext and maxContext of each Multiple Update command (called *Right Distance*) and the difference between the minContexts of two consecutive Multiple Updates (called *Bottom Distance*). In this example, the algorithm compresses the Multiple Updates that are sent to each node using $RightDistance = 3$ and $BottomDistance = 16$ (see Figure 39b). As a result, the number of messages will be reduced by 75% for this specific Multiple Update command. The proposed algorithm is

GEORGE MATHEOU

implemented by the DSU's *Compression Unit*. When a node receives a compressed Multiple Update, the Network Manager decompresses it by its *Decompression Unit*.



(a) No compression.

(b) Compression for DThreads with $RC > 1$.

(c) Reducing Update messages for DThreads with $RC = 1$.

Figure 39: Example of reducing the network traffic generated by a Multiple Update. T1(X,Y) denotes a Multiple Update for DThread T1.

4.3.6.3 Reducing the number of messages in the case of Multiple Updates for DThreads with RC=1

In FREDDO, the DThreads with RC=1 are treated differently, compared to other DDM implementations. The TSU does not allocate RC values for their instances in order to reduce the memory allocation. Instances of such DThreads are scheduled immediately when Updates are received for them. This approach allows to schedule a DThread instance for execution to any node. We can benefit from this, by dividing the range of Context values of a Multiple Update, into equal parts where the number of parts is equal to the number of nodes. As an example, consider the Multiple Update of Figure 39a and that the DThread T1 has RC=1. In this case, four different Multiple Updates will be distributed to the nodes as described in Figure 39c. This methodology does not require compression and it is possible to reduce the number of messages by 75% for this specific example.

4.3.6.4 Packing correlated Updates together

Our final technique reduces the messages that carry Update commands for the same destination node. In particular, when a producer-instance sends several Update commands (Single Updates, (Un)compressed Multiple Updates) to a remote node, for the same DThread, the FREDDO's runtime performs two steps. First, it sends the DThread's TID along with the number of the Updates that will be sent, through a *general packet*. After that, the Context values of the Updates are sent as a single data packet to the remote node.

GEORGE MATHEOU

Chapter 5

Programming Methodology

5.1 Introduction

In this chapter we present the programming methodology used in FREDDO and MiDAS. Both implementations are based on the DDM semantics for programmability. In this thesis we provide software APIs which permit programmers to have a total control of DDM applications. The APIs enable programmers to manage the DDM execution environment and the dependency graph (create and remove DThreads) as well as to perform Update operations.

The APIs provided in this work can be used by high-level tools for better programmability. Such tools include source-to-source compilers (e.g., the TFlux source-to-source compiler [145]) and declarative parallel programming languages like Concurrent Collections (CnC) [169]. As a proof of concept, an extension of the TFlux source-to-source compiler [145] was implemented in order to ease the development of DDM applications targeting the MiDAS system. The primary target of the compiler is to hide the details of the API from programmers. To develop a DDM application, the programmer only needs to describe the parallel sections of the application using TFlux directives, in a similar manner as those of OpenMP. Functionalities such as loading/unloading the TSU and managing Context values are handled automatically. Notice that the TFlux directives of MiDAS were developed by Dr. Pedro Trancoso and Andreas Diavastos. Details about the TFlux source-to-source compiler and its directives can be found in [9, 145, 170], and are omitted here for brevity.

Section 5.2 presents the API provided by FREDDO for implementing DDM applications. Programming examples implemented using FREDDO's API are given in Section 5.3. Section 5.4 presents the API provided for developing DDM applications targeting the MiDAS architecture. Finally, Section 5.5 presents the Matrix Multiplication application implemented using the MiDAS API and TFlux directives.

GEORGE MATHEOU

5.2 FREDDO API

FREDDO provides an API that enables programmers to develop DDM applications. The API is a C++ library that includes a set of runtime functions and classes which are grouped together in a C++ namespace called *ddm*. The API's functions and classes are described below.

5.2.1 Basic Runtime Functions

- **void init(string peerfile, PortNumber port, freddo_config* conf = nullptr)**: initializes FREDDO for distributed execution using the Custom Network Interface (CNI) support. The user should provide the peer-file and the port number that will be used for the inter-node communication. The number of Kernels of each node should be included in the peer-file. *conf* is a C++ object which allows users to configure the FREDDO runtime. Currently, it provides functionalities for configuring the pinning of TSU, the Network Manager's *receiving thread* and Kernels, on the system's cores. If *conf* is not provided, TSU will run on core 0, the Network Manager's *receiving thread* will run on core 1 and Kernels will run on the rest cores of the system.
- **void init(int *argc, char ***argv, unsigned int numOfKernels, freddo_config* conf = nullptr)**: initializes FREDDO for distributed execution using the MPI support. The argc and argv arguments are used for the MPI initialization. The function will start numOfKernels Kernels and it will use the *conf* object to configure the pinning, as described above. The peer-file and other parameters needed for the distributed execution will be provided through the *mpirun* command.
- **void init(unsigned int numKernels, freddo_config* conf = nullptr)**: initializes FREDDO for single-node execution and it starts numKernels Kernels. The *conf* object can be used for configuring the pinning of the TSU and Kernels.
- **void run()**: computes the RC values of Pending Thread Templates and starts the scheduling of DThreads. The scheduling will finish if the TSU has no Updates to execute (the Input Queues and Unlimited Input Queues are empty) and there are no pending ready DThread instances (the Output Queues are empty). If the distributed mode is enabled, the function returns when the FREDDO's runtime detects distributed execution termination (see Section 4.3.5). We note that all initial Updates have to be sent to the TSU prior the execution of this command. If this does not happen, the *run* function will return immediately since the TSU's queues will be empty. Thus, every DDM program should have Update commands before the *run* function.
- **void finalize()**: releases all the resources allocated by FREDDO (used for both single-node and distributed environments).

GEORGE MATHEOU

- **void buildDistributedSystem()**: builds the distributed system. Particularly, the following steps are performed:
  - The Network Manager starts.
  - The nodes of the distributed system communicate and exchange their number of Kernels.
  - The Distributed Scheduling Unit (DSU) and the Data Forward Tables (DFTs) are created.

  Notice that, the *init* runtime function, the DThreads' declarations and the registration of the variables in the shared Global Address Space should precede this function.
- **PeerID getPeerID()**: returns the ID (or rank) of a peer/node.
- **bool isRoot()**: indicates if a node is the RootNode.
- **unsigned int getNumberOfPeers()**: returns the number of nodes of the distributed system.
- **AddrID addInGAS(void* address)**: registers a shared object in the Global Address Space (GAS). It also returns a unique identifier for the shared object (called *GAS_ID*). The type of *GAS_ID* is the AddrID.
- **void addModifiedSegmentInGAS(AddrID addrID, void* address, size_t size)**: registers an output data segment/object of a DThread instance. It requires the data segment's *GAS_ID* (addrID), the conventional main memory address of that segment and its size in bytes. This function informs the runtime which data segment(s) will be forwarded to the DThread instance's consumer(s) running on remote node(s).
- **void sendDataToRoot(AddrID addrID, void* address, size_t size)**: sends a data segment to the RootNode. This function is used for gathering the results on the RootNode and it requires the same arguments as the *addModifiedSegmentInGAS* function.

5.2.2 DFunctions

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

1. **SimpleDFunction**: for DThreads with Nesting-0. It has no arguments since the Context value is always zero.
2. **MultipleDFunction**: for DThreads with Nesting-1. It has as input argument the *ContextArg* data type which is a single value that contains an index of a one-level loop.

GEORGE MATHEOU

3. **MultipleDFunction2D**: for DThreads with Nesting-2. It has as input argument the *Context2DArg* data type which contains indexes of a two-level loop (outer and inner parts).
4. **MultipleDFunction3D**: for DThreads with Nesting-3. It has as input argument the *Context3DArg* data type which contains indexes of a three-level loop (outer, middle and inner parts).

5.2.3 DThread Classes

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

5.2.3.1 DThread Class

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

- **void updateAllCons()**: decrements the RC of the DThread's consumers, by one. All consumer-DThreads should have Nesting-0.
- **void updateAllCons(Context context)**: decrements the RC that corresponds to the input Context of the DThread's consumers, by one, with Nesting>=1. Examples:
  - *updateAllCons(3)*: decreases the RC value that corresponds to the Context=3, for all consumers of this DThread.
  - *updateAllCons(*$\{2, 3\}$*)*: decreases the RC value that corresponds to the Context with *outer index*=2 and *inner index*=3, for all consumers of this DThread.
  - *updateAllCons(*$\{2, 5, 3\}$*)*: decreases the RC value that corresponds to the Context with *outer index*=2, *middle index*=5 and *inner index*=3, for all consumers of this DThread.
- **void updateAllCons(Context context, Context maxContext)**: decrements the RC of multiple instances, by one, of the DThread's consumers with Nesting>=1. For each Nesting we are providing a separate example:
  - *updateAllCons(0,13)*: decrements the RCs that correspond to Contexts 0 to 13, of all consumers of this DThread.

GEORGE MATHEOU

  - *updateAllCons(*$\{0,0\}$, $\{0,4\}$*)*: decrements the RCs that correspond to Contexts $\{0,0\}$ to $\{0,4\}$, of all consumers of this DThread. More specifically, the instances with Contexts $\{0,0\}$, $\{0,1\}$, $\{0,2\}$, $\{0,3\}$ and $\{0,4\}$ will be updated, for each consumer.
  - *updateAllCons(*$\{0,0,1\}$, $\{0,0,3\}$*)*: decrements the RCs that correspond to Contexts $\{0,0,1\}$ to $\{0,0,3\}$, of all consumers of this DThread. More specifically, the instances with Contexts $\{0,0,1\}$, $\{0,0,2\}$ and $\{0,0,3\}$ will be updated, for each consumer.

- **unsigned int getTID()**: returns the Thread Identifier (TID) of the DThread. The TID of each DThread is created at runtime by the framework. In the previous DDM implementations, users have to specify the TID manually.
- **void setConsumers(Consumers consList)**: sets the list of consumers. The *consList* variable is a C++ vector that contains pointers to DThread objects (DThread*).

5.2.3.2 SimpleDThread Class

SimpleDThread is a DThread with Nesting-0 and it has only one instance with Context=0. Constructors/methods:

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

5.2.3.3 MultipleDThread Class

MultipleDThread is a DThread with Nesting-1 and has multiple instances. Constructors/methods:

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

GEORGE MATHEOU

5.2.3.4 MultipleDThread2D Class

Similar with MultipleDThread, but it has Nesting-2. Constructors/methods:

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

5.2.3.5 MultipleDThread3D Class

Similar with MultipleDThread, but it has Nesting-3. Constructors/methods:

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

GEORGE MATHEOU

5.2.3.6 FutureDThread Classes

FutureDThread classes are derived-classes of the "Regular" DThreads (SimpleDThread, MultipleDThread, MultipleDThread2D and MultipleDThread3D). FutureDThread classes have the same constructors and methods with the "Regular" DThreads, however the RC value is not required in their constructors. As a result, their RC value will be evaluated at runtime (initially they will be stored in the Pending Template Memory) using the producer-consumer relationships of the program. The following FutureDThread classes are provided: FutureSimpleDThread, FutureMultipleDThread, FutureMultipleDThread2D and FutureMultipleDThread3D.

5.2.4 UML Diagram of DThread Classes

The UML diagram of all DThread classes is depicted in Figure 40. All classes are derived-classes of the DThread class.

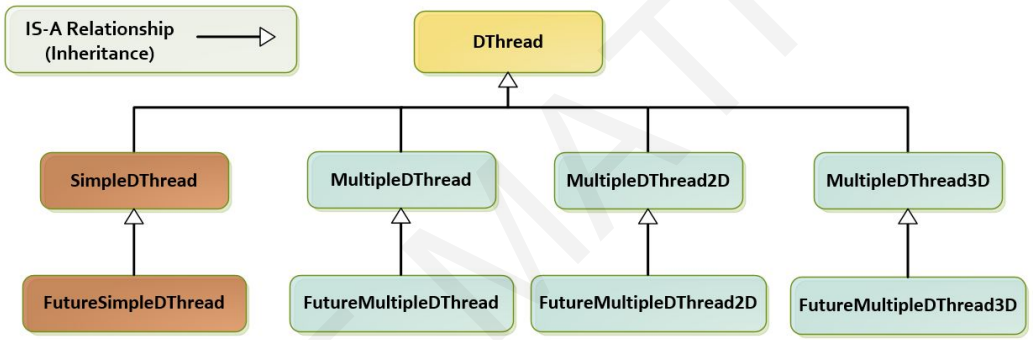


Figure 40: The UML diagram of all DThread classes.

5.3 Programming examples using FREDDO

5.3.1 Simple application

In this section we present the mapping of a very simple application to a DDM program. The dependency graph of the application is shown in Figure 41 which is composed of three DThreads. The RC values are depicted as shaded values next to the nodes. The rounded rectangles illustrate the functionalities of the DThreads. The program will print the string "Hello World from the FREDDO framework!".

Listing 5.1 depicts the FREDDO code of the example implemented using *Future DThreads*. The code of DThreads t1 and t2 are embodied in Lambda expressions where the code of DThread t3 is embodied in a standard C++ function. The type of all DThreads is *FutureSimpleDThread* since they don't implement loops or recursion. In line 30, an initial Update is sent to DThread t1 since is the root of the DDM dependency graph (it has no producers).

GEORGE MATHEOU

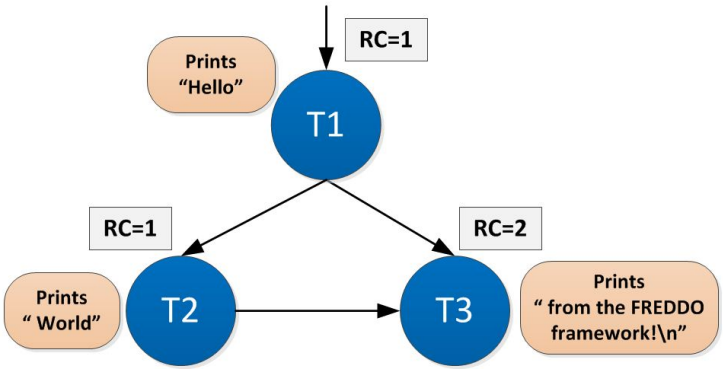


Figure 41: The DDM dependency graph of a simple application.

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

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

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

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

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

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

 t3 = new FutureSimpleDThread(t3_code);

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

 t1->update();  // Initial Update
 ddm::run(); // Start the DDM scheduling

 delete t1; delete t2; delete t3; // Remove the DThreads
 ddm::finalize(); // Stops the Kernels and releases the resources
}
```

Listing 5.1: A simple DDM example: solution with FutureSimpleDThreads.

An alternative solution is to use *SimpleDThreads*. In this case, we have to specify the RC value of each DThread manually. Listing 5.2 depicts the DDM code using *SimpleDThreads* where the code of each DThread is embodied in a standard C++ function.

GEORGE MATHEOU

```
1  #include <iostream>
2  #include <dthreads.h>
3  using namespace ddm;

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

7  void t1_code(){ // The t1's code
8   cout << "Hello";
9   t2->update(); // Update t2 DThread
10  t3->update(); // Update t3 DThread
11 }

13 void t2_code(){ // The t2's code
14  cout << " World";
15  t3->update(); // Update t3 DThread
16 }

18 void t3_code(){ // The t3's code
19  cout << " from the FREDDO framework!\n";
20 }

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

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

29  t1->update(); // Initial Update
30  ddm::run(); // Start the DDM scheduling

32  delete t1; delete t2; delete t3; // Remove the DThreads
33  ddm::finalize(); // Stops the Kernels and releases the resources
34 }
```

Listing 5.2: A simple DDM example: solution with SimpleDThreads.

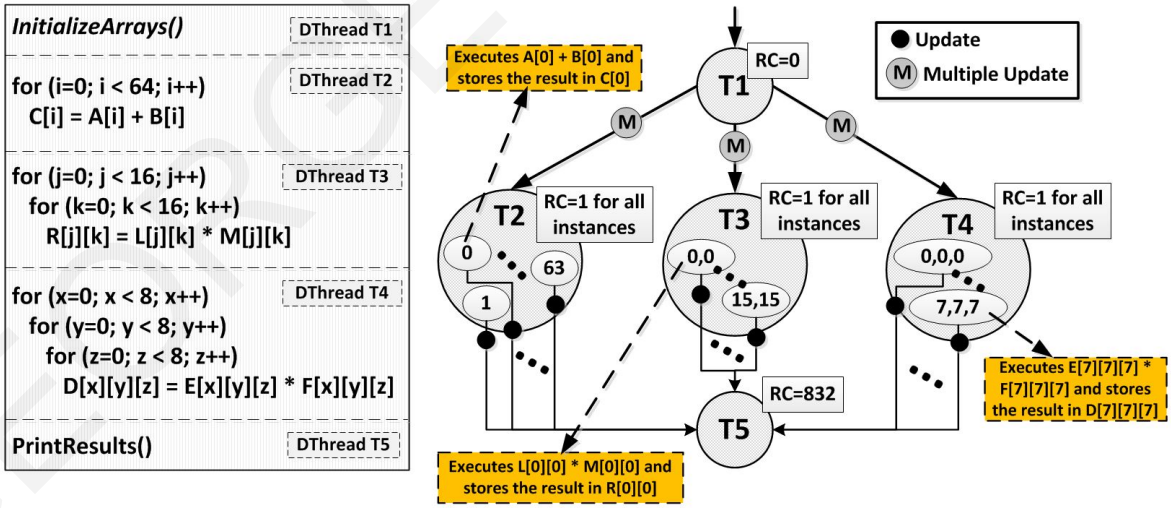


Figure 42: Example of a synthetic DDM application.

5.3.2 Synthetic application

An example of a synthetic DDM program is shown in Figure 42. On the left side of the figure, the pseudo-code of the application and its partitioning into five DThreads are depicted. From this code

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

6  // Declate DThread objects
7  FutureSimpleDThread *t1;
8  FutureMultipleDThread *t2;
9  MultipleDThread2D *t3;
10 FutureMultipleDThread3D *t4;
11 SimpleDThread *t5;

13 // Declare Global Variables (Arrays, etc.)

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

18  // Update the instances of consumers
19  t2->update(0, 63); // Multiple Update
20  t3->update({0,0}, {15,15}); // Multiple Update
21  t4->update({0,0,0}, {7,7,7}); // Multiple Update
22 }

24 void t4_code(Context3DArg c){ // The t4's code
25  auto x = c.Outer, y = c.Middle, z = c.Inner;
26  D[x][y][z] = E[x][y][z] * F[x][y][z];
27  t4->updateAllCons();
28 }

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

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

38  // DTheads declarations using Lambda expressions
39  t2 = new FutureMultipleDThread([&](ContextArg  cntx){ // The t2's code
40   C[cntx] = A[cntx] + B[cntx];
41   t5->update();
42  });

44  t3 = new MultipleDThread2D([&](Context2DArg  cntx){ // The t3's code
45   auto j = cntx.Outer, k = cntx.Inner;
46   R[j][k] = L[j][k] * M[j][k];
47   t5->update();
48  }, 1); // 1 at this point is the RC value

50  t5 = new SimpleDThread([&](){ // The t5's code
51   // Print Results ...
52  }, 832); // 832 at this point is the RC value

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

60  t1->update(); // Decrease the RC of T1
61  ddm::run(); // Start the DDM scheduling
62  delete t1; ...; delete t5;
63  ddm::finalize(); // Deallocate Resources
64 }
```

Listing 5.3: DDM code of a synthetic application.

it is possible to observe a number of dependencies. DThreads T2, T3 and T4 depend on T1 which is responsible for initializing the data. Also, T5 depends on T2, T3 and T4 since T5 prints the output

GEORGE MATHEOU

results generated by them. These dependencies form the DDM dependency graph of the application which is presented on the right side of Figure 42. The RC values are depicted as shaded values next to the nodes. The three for-loop blocks are fully parallel and are mapped into three different DThreads. Each instance of a DThread is identified by the Context and it executes the inner command of the block. T2 has 64 instances (with Contexts from 0 to 63), T3 has 256 instances (with Contexts from 0,0 to 15,15) and T4 has 512 instances (with Contexts from 0,0,0 to 7,7,7). DThread T5 depends on all instances of DThreads T2-T4, thus its RC is equal to 832. Moreover, the instances of DThreads T2-T4 have RC=1 because they have only one producer (T1). When T1 finishes its execution a *Multiple Update* will be sent in each consumer-thread. As a result, all the instances of DThreads T2-T4 will be executed concurrently.

Listing 5.3 depicts one possible implementation of the application. In this example, we place the DThreads' code in standard C++ functions for T1 and T4, and in Lambda expressions for T2, T3 and T5. Furthermore, we are using a combination of *Regular DThreads* (T3 and T5) and *Future DThreads* (T1, T2 and T4).

```
double AOrig[n*n]; // The original matrix
double *A[N][N];   // Each entry of A is a pointer to a tile

for (kk = 0; kk < N; kk++) {        // Loop 1
  // A[kk][kk]:inout
  diag(A[kk][kk]);

 for (jj = kk + 1; jj < N; jj++)  // Loop 2
  // A[kk][kk]:input, A[kk][jj]:output
  front(A[kk][kk], A[kk][jj]);

 for (ii = kk + 1; ii < N; ii++)  // Loop 3
  // A[kk][kk]:input, A[ii][kk]:output
  down(A[kk][kk], A[ii][kk]);

 for (ii = kk + 1; ii < N; ii++)  // Loop 4
  for (jj = kk + 1; jj < N; jj++) // Loop 5
   // A[ii][kk]:input, A[kk][jj]:input, A[ii][jj]:ouput
   comb(A[ii][kk], A[kk][jj], A[ii][jj]);
}
```

Listing 5.4: Tile LU Decomposition (Original Code).

5.3.3 Tile LU Decomposition: single-node and distributed implementations

In this section we are providing the FREDDO implementation of the Tile LU Decomposition which has a complex dependency graph. LU decomposition (also called LU factorization) is an important algorithm used for solving systems of linear equations efficiently [171]. The LU kernel factors a dense matrix into the product of a lower triangular *L* and an upper triangular *U* matrix [172]. The dense $n \times n$ matrix A is divided into an $N \times N$ array of $B \times B$ tiles ($n = NB$). This enables to exploit temporal locality on sub-matrix elements. The code of the original tile LU Decomposition is shown in Listing 5.4. The code is composed of five nested loops that perform four basic operations

GEORGE MATHEOU

on a tiled matrix. For demonstration purposes we choose the following indicative names for the operations: *diag*, *front*, *down* and *comb*. The algorithm is based on an earlier version developed in StarSs [127].

Benchmark Analysis

In every iteration of the outermost loop, the *diag* operation takes as input the diagonal tile that corresponds to the iteration number to produce its new value. The *front* operation produces the remaining tiles on the same row as the diagonal tile. For each one of those tiles, it takes as input the result of the *diag* in addition to the current tile to produce its new value. Similarly, the *down* operation produces the remaining tiles on the same column as the diagonal tile. The *comb* operation produces the rest of the tiles for that LU iteration. For every tile it produces, it takes as input three tiles: the current tile, the tile produced by the front operation and the tile produced by the down operation. It multiplies the second and third tiles and adds the result to the first tile to produce the final resulting tile. This computational pattern is repeated in the next LU iteration on a subset of the resulting matrix that excludes the first row and column and continues for as much iterations as the diagonal tiles of the matrix. Figure 43 depicts the tiles produced by the four operations for the first iteration of LU decomposition on a 4×4 tile matrix. Each tile is labeled with the first letter of its operation. The tile produced by the *diag* operation is labeled as diag. The arrows in the figure indicate the input tiles needed by each operation to produce its result.

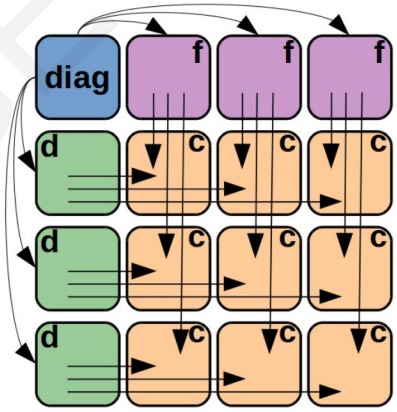


Figure 43: LU Decomposition: dependencies between operations for the first iteration.

Dependency Graph

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

GEORGE MATHEOU

- The DThreads that execute the operations depend on the *loop_1_thread* since the index of the outermost loop is used in the four operations.
- The *front_thread* and *down_thread* DThreads depend on the *diag_thread*.
- The *comb_thread* depends on the *front_thread* and *down_thread* DThreads.
- The next LU iteration depends on the results of the previous iteration. Particularly, the results produced by the *comb_thread* invocations, in the current iteration, are consumed by the invocations of the *diag_thread*, *front_thread*, *down_thread* and *comb_thread*, of the next LU iteration.

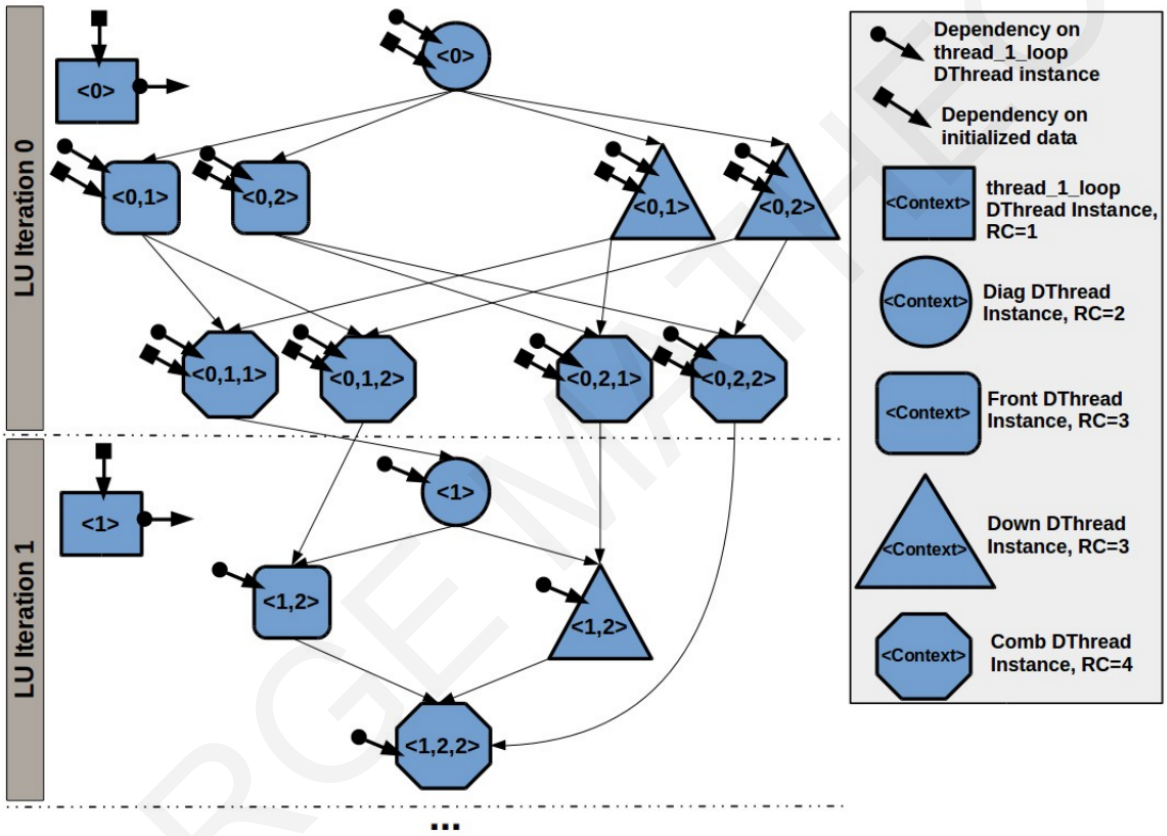


Figure 44: The LU's DDM dependency graph for the first two iterations of a 3×3 tile matrix (N=3).

The dependency graph shown in Figure 44 illustrates the dependencies among the instances of the DThreads for the first two iterations of the tile LU algorithm. For simplicity, a 3×3 tile matrix (N=3) was selected. Each DThread's instance is labeled with the value of its Context.

FREDDO Code

Figure 45 depicts the FREDDO code for the tile LU Decomposition algorithm. We have used the DSM/GAS features of FREDDO since the algorithm uses a shared object, the tile matrix *A*. The code of the DThreads is placed in standard C/C++ functions. Each call of an Update command in the DThreads' code corresponds to one dependency arrow in Figure 44. The Update operations at the

```
#include <freddo/dthreads.h>
using namespace ddm; // Use the freddo namespace

// DThread Objects
MultipleDThread   *loop_1DT, *diagDT;
MultipleDThread2D *frontDT,  *downDT;
MultipleDThread3D *combDT;
AddrID gasA; // The GAS_ID of matrix A
TYPE ***A;   // The tile matrix
TYPE *Aorig; // The original matrix
int tS = B*B*sizeof(TYPE); // size of tile in bytes

// The code of the thread_1_loop DThread
void loop_1_code(ContextArg kk) {
 diagDT->update(kk);

 if (kk < N-1) {
  frontDT->update({kk, kk+1}, {kk, N-1});
  downDT->update({kk, kk+1}, {kk, N-1});
  combDT->update({kk, kk+1, kk+1}, {kk, N-1, N-1});
 }
}

// The code of diag_thread DThread
void diag_code(ContextArg kk) {
 addModifiedSegmentInGAS(gasA, A[kk][kk], tS);
 diag(A[kk][kk]); // diag operation
 sendDataToRoot(gasA, A[kk][kk], tS);

 if (kk < N-1) {
  frontDT->update({kk, kk+1}, {kk, N-1});
  downDT->update({kk, kk+1}, {kk, N-1});
 }
}

// The code of the front_thread DThread
void front_code(Context2DArg context) {
 int kk = context.Outer, jj = context.Inner;
 addModifiedSegmentInGAS(gasA, A[kk][jj], tS);
 front(A[kk][kk], A[kk][jj]); // front operation
 sendDataToRoot(gasA, A[kk][jj], tS);

 combDT->update({kk, kk+1, jj}, {kk, N-1, jj});
}

// The code of the down_thread DThread
void down_code(Context2DArg context) {
 int kk = context.Outer, jj = context.Inner;
 addModifiedSegmentInGAS(gasA, A[jj][kk], tS);
 down(A[kk][kk], A[jj][kk]); // down operation
 sendDataToRoot(gasA, A[jj][kk], tS);

 combDT->update({kk, jj, kk+1}, {kk, jj, N-1});
}
```

```
// The code of the comb_thread DThread
void comb_code(Context3DArg context) {
 int kk = context.Outer, ii = context.Middle,
     jj = context.Inner;
 addModifiedSegmentInGAS(gasA, A[ii][jj], tS);

 // comb operation
 comb(A[ii][kk], A[kk][jj], A[ii][jj]);

 // Updates for the next LU iteration
 if (ii == kk+1 && jj == kk+1) {
       diagDT->update(kk+1);
 } else if (ii == kk+1) {
       frontDT->update({ii, jj});
 } else if (jj == kk+1) {
       downDT->update({jj, ii});
 } else {
       combDT->update({kk+1, ii, jj});
 }
}

// The main program
void main(int argc, char* argv[]) {
 // Initialize data (matrices, etc.)
 initializeData();

 // Register A in GAS
 gasA = addInGAS(A[0][0]);

 // Initializes the FREDDO execution environment
 init(&argc, &argv, NUM_OF_KERNELS);

 // Allocation of the DThread Objects
 loop_1DT = new MultipleDThread(loop_1_code, 1);
 diagDT   = new MultipleDThread(diag_code, 2);
 frontDT  = new MultipleDThread2D(front_code, 3);
 downDT   = new MultipleDThread2D(down_code, 3);
 combDT   = new MultipleDThread3D(comb_code, 4);

 // Updates resulting from data initialization
 if (ddm::isRoot()) {
  loop_1DT->update(0, N-1);
  diagDT->update(0);
  frontDT->update({0, 1}, {0, N-1});
  downDT->update({0, 1}, {0, N-1});
  combDT->update({0, 1, 1}, {0, N-1, N-1});
 }

 // Starts the DDM scheduling in each node
 run();

 // Releases the resources of distributed FREDDO
 finalize();
}
```

Figure 45: FREDDO code of the tile LU algorithm (the highlighted code is required for the distributed execution).

end of the comb_code DFunction implement a switch actor, which depending on the Context of the DThread's instance, a different consumer-instance is updated. In the *main function* of the program the matrices are allocated and initialized. After that, the tile matrix *A* is registered in the GAS using the *addInGAS* runtime function. At this point, the FREDDO's runtime registers the address of the tile matrix *A* in the *Global Address Directory* (GAD) of each node. The runtime also creates a GAS_ID for the matrix *A* which is stored in the *gasA* variable. The *init* runtime function initializes the FREDDO's execution environment and activates NUM_OF_KERNELS Kernels in each node. The constructor of each DThread object takes two arguments, the DFunction and the RC value (e.g., the diagDT object has DFunction=diag_code and RC=2).

GEORGE MARIOU

After the creation of the DThread objects, the initial Updates are sent to the TSUs for execution. These Updates correspond to the arrows of Figure 44 that describe dependencies on initialized data. The initial Updates have to be executed one time only. In this example the *RootNode* was selected to execute these updates which are distributed across the nodes through its DSU module. Notice that the initial Updates or any other Updates can be executed by any node of the system. The *run* function starts the DDM scheduling and waits until the FREDDO's runtime detects the distributed execution termination (see Section 4.3.5). When the *run* function returns, all the resources allocated by the FREDDO framework are deallocated using the *finalize* function.

The FREDDO's memory model in combination with its distribution scheme and the implicit distributed termination approach allows the distributed FREDDO programs to be fundamentally the same as the single-node ones. For the distributed data-driven execution users have to: (i) provide a peer file that contains the ip addresses or the host-names of the system's nodes, (ii) register the shared objects in the GAS using the *addInGAS* function and (iii) specify the output data of each DThread using the *addModifiedSegmentInGAS* runtime function. Additionally, for gathering the results in the *RootNode*, users have to use the *sendDataToRoot* runtime function. Both the *addModifiedSegmentInGAS* and *sendDataToRoot* functions require the GAS_ID of a shared object, the conventional main memory address of that object and its size in bytes. For instance, in the diag_code DFunction, the tile *A[kk][kk]* is declared as a modified segment since is computed by the *diag* routine. The size of this tile is equal to *tS* and its GAS_ID is equal to *gasA* since it's a part of the tile matrix *A*. In Figure 45, the required code for the distributed execution is highlighted.

5.4 MiDAS API

DDM applications targeting the MiDAS architecture are developed using two different programming interfaces (Figure 46). The first programming interface consists of ANSI-C augmented with a C API. The API is a C library that includes a set of functions which allow programmers to: (1) initialize/reset the multi-core processor, (2) send Update commands to the TSU via the Input FSL Buses, (3) fetch the ready DThread instances from the Output FSL Buses and execute their codes, (4) create Thread Templates and (5) manage the processor's hardware peripherals (timers, interrupt controllers, DDR3 RAM, etc.). The second programming interface implements the same functionalities as the C API and it consists of C++ augmented with a C++ API. In particular, it implements a subset of FREDDO's programming interface. The C++ programming interface does not support all the functionalities present in FREDDO since the MicroBlaze GCC compiler does not support C++11. Both programming interfaces have common functions/routines which are presented in Section 5.4.1.

GEORGE MATHEOU

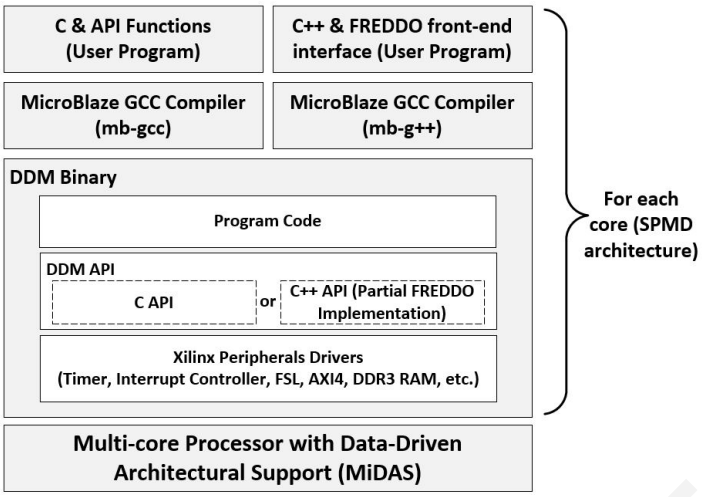


Figure 46: Programming methodology of the MiDAS system.

We developed two different programming interfaces in order to give the flexibility to the programmers to choose between two different programming paradigms: procedural/structured and object-oriented. The C API can be used in embedded multi-core data-driven processors where C++ compilers may not be available or may not be free of charge. Further, the C API can be used in systems where the memory size is limited since the C binaries are usually smaller than the C++ binaries. However, the C++ API can be used by programmers who need special C++ features like templates, exceptions, overloading, and access to the Standard Template Library (STL). Finally, the C++ API can provide a lesser error-prone programming interface, compared to the C API, since DThreads are implemented through special C++ classes which expose only the proper commands as public methods to users. For example, a user cannot perform a Multiple Update on a SimpleDThread since it only has one instance.

The DDM binary is produced by the MicroBlaze GCC compiler with the following compilation setup: mb-gcc for the C API and mb-g++ for the C++ API. It includes the user's application, the DDM API (in C or C++) and the drivers of the Xilinx peripherals, i.e., timers, buses, memories, interrupt controllers, etc. Users are able to write their programs using the Xilinx Software Development Kit (SDK). Xilinx SDK is an enhanced Eclipse platform that helps users to create software applications for all Xilinx embedded microprocessors.

To execute the DDM code on the multi-core processor, an instance of the DDM binary has to be loaded in each MicroBlaze, i.e., an SPMD architecture is provided. The flow of each program instance is managed by the TSU dynamically at runtime through the DDM API. The Xilinx Microprocessor Debugger (XMD) [62] is used to download the DDM binary into the Main Memory, as well as for executing the code. For each core we reserve a 2-MB private memory segment in the shared main memory which is used for storing the DDM binary. The private memory segment includes the code

GEORGE MATHEOU

and data sections of an application as well as the heap and stack sections (32-KB for heap and 32-KB for stack). For this configuration we used the *Generate Linker Script* tool of Xilinx Software Development Kit (SDK).

5.4.1 Common functions for the C and C++ APIs

- **void init_processor(unsigned int numOfEnabledCores)**: initializes the multi-core processor and enables *numOfEnabledCores* cores. This function should be called from each core.
- **bool is_master()**: indicates whether a core executed this command is the master core.
- **void finalize()**: deallocates the resources allocated by the API and disables the interrupts of the MicroBlaze and TSU.
- **void run_explicit()**: starts the scheduling of the DThreads. It also fetches the information of the ready DThread instances from the associated Output FSL Bus and executes their code (DFunctions). Prior the execution of the DFunctions, it constructs the Context arguments based on the Nesting attribute of each DThread. Notice that the DFunctions support only standard C/C++ functions. The function returns when the *notify_termination* function is called.
- **void notify_termination()**: indicates that the DDM program should be terminated. This function should be called by the last executed DThread instance.
- **CREATE_N2(OUTER,INNER)**: encodes the outer and inner fields in a 32-bit integer value. It is used to create the Context values of Update operations targeting DThreads with Nesting-2.
- **CREATE_N3(OUTER,MIDDLE,INNER)**: encodes the outer, middle and inner fields in a 32-bit integer value. It is used to create the Context values of Update operations targeting DThreads with Nesting-3.

5.4.2 C API

The C API provides functions for allocating, deallocating and updating DThreads. In the case of functions that create DThreads (*dthread_create_**), the user should specify the DFunction, the RC value and the Scheduling Policy of each DThread. When a DThread is created, the API returns the TID of the created DThread. The following functions are provided:

- **TID dthread_create_simple(SimpleDFunction sDFunction, RC readyCount, SchMethod sch_meth, SchValue sch_value)**: creates a Thread Template with Nesting=0.
- **TID dthread_create_multiple(MultipleDFunction mDFunction, RC readyCount, SchMethod sch_meth, SchValue sch_value)**: creates a Thread Template with Nesting=1.
- **TID dthread_create_multiple2D(MultipleDFunction2D mDFunction2D, RC readyCount, SchMethod sch_meth, SchValue sch_value)**: creates a Thread Template with Nesting=2.

GEORGE MATHEOU

- **TID dthread_create_multiple3D(MultipleDFunction3D mDFunction3D, RC readyCount, SchMethod sch_meth, SchValue sch_value)**: creates a Thread Template with Nesting=3.
- **void remove_thread_template(TID thread_id)**: removes a Thread Template from the TSU.
- **void simple_update(TID thread_id)**: decrements the RC value of the first instance (Context=0) of the DThread with TID=thread_id. This is usually used for DThreads with Nesting=0.
- **void single_update(TID thread_id, Context context)**: decrements the RC value that corresponds to the Context value of the DThread with TID=thread_id.
- **void multiple_update(TID thread_id, Context context, Context max_context)**: decrements the RC value of multiple instances (from context to max_context) of the DThread with TID=thread_id.

5.4.3 C++ API

The C++ API includes a set of functions (Section 5.4.1) and classes which are grouped together in a C++ namespace called *ddm*. The API enables programmers to manage the DDM execution environment and the dependency graph, i.e., to create and remove DThreads, as well as to perform Update operations. We provide four special C++ classes that enable the creation, update and removal of DThreads with different characteristics, i.e., Nesting values. These classes are derived-classes of the *DThread* base-class. A programmer is able to create and load Thread Templates in the TSU by using the constructors of the special classes. In each constructor, the user has to specify the DFunction, the RC value and the Scheduling Policy of the DThread. The DThreads are removed from the TSU using the *delete* operator of C++, where for each case the appropriate *destructor* is called.

5.4.3.1 SimpleDThread Class

Implements a DThread with Nesting-0. Constructors/methods:

- **SimpleDThread(SimpleDFunction sDFunction, RC readyCount, SchMethod sch_meth, SchValue sch_value)**: inserts a SimpleDThread in the TSU.
- **void update()**: decrements the RC value of this DThread.

5.4.3.2 MultipleDThread Class

Implements a DThread with Nesting-1. Constructors/methods:

- **MultipleDThread(MultipleDFunction mDFunction, RC readyCount, SchMethod sch_meth, SchValue sch_value)**: inserts a MultipleDThread in the TSU.
- **void update(Context context)**: decrements an RC value of this DThread.

GEORGE MATHEOU

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

5.4.3.3 MultipleDThread2D Class

Implements a DThread with Nesting-2. Constructors/methods:

- **MultipleDThread2D(MultipleDFunction2D mDFunction2D, RC readyCount, SchMethod sch_meth, SchValue sch_value)**: inserts a MultipleDThread2D in the TSU.
- **void update(Context context)**: decrements a RC value of this DThread. In order to create the Context value, the *CREATE_N2* macro should be used.
- **void update(Context context, Context maxContext)**: decrements the RC value of multiple instances of this DThread. In order to create the Context values, the *CREATE_N2* macro should be used.

5.4.3.4 MultipleDThread3D Class

Implements a DThread with Nesting-3. Constructors/methods:

- **MultipleDThread3D(MultipleDFunction3D mDFunction3D, RC readyCount, SchMethod sch_meth, SchValue sch_value)**: inserts a MultipleDThread3D in the TSU.
- **void update(Context context)**: decrements a RC value of this DThread. In order to create the Context value, the *CREATE_N3* macro should be used.
- **void update(Context context, Context maxContext)**: decrements the RC value of multiple instances of this DThread. In order to create the Context values, the *CREATE_N3* macro should be used.

5.5 Implementing Matrix Multiplication for MiDAS

In this section we show how a programmer is able to develop the Matrix Multiplication application (Listing 5.5) in DDM using the C and C++ APIs. In this simple example, the outer for-loop of the algorithm is parallelized using one MultipleDThread, called *thread_1*. Each instantiation of *thread_1* calculates one row of the matrix, i.e., it executes the two nested for-loops of the algorithm (see Figure 47). Each instantiation is labelled with its Context as $<Context>$. Initially, the N instantiations of *thread_1* are spawned in parallel due to the fact that the *thread_1*'s instances are independent. Finally, when N instantiations (spanning from 0 to N-1) are executed, the program is then completed.

GEORGE MATHEOU

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

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

 // Data initialization goes here ...

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

 // Post execution code goes here ...
}
```

Listing 5.5: The original Matrix Multiplication.

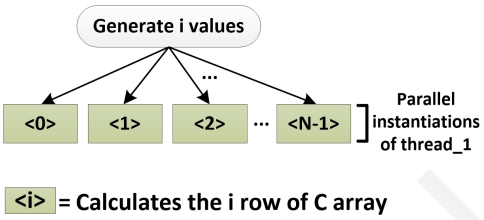


Figure 47: Matrix Multiplication: dynamic instantiations of *thread_1*.

5.5.1 Implementation using the C API

Listing 5.6 depicts the code of the DThreads. We declare two DThreads, *thread_1* and *thread_2*. *thread_1* is responsible for executing the two nested for-loops of the algorithm (lines 13-15) using the *Context* value as the *i* index of the outer loop. After that, each instantiation of *thread_1* updates *thread_2* (line 18). *thread_2* is used to print the results, to release the resources that are allocated by the two DThreads (lines 26-27), and to notify the API that it is the last DThread instance of the dependency graph (line 29).

```
1  // Includes ...
2
3  // Global Variables
4  TID t1_TID, t2_TID;
5
6  // Declare the code of thread_1
7  void thread_1(ContextArg cntx){
8   // Local Variables
9   int j, k;
10  float res;
11
12  // Executes the algorithm using the context as i
13  for (j = 0; j < N; j++)
14   for (k = 0; k < N; k++)
15    C[cntx][j] += A[cntx][j] * B[k][j];
16
17  // Update the thread_1's consumer, i.e thread_2
18  simple_update(t2_TID);
19 }
20
21 // Declare the code of thread_2
22 void thread_2(){
23  // Print results goes here ...
24
25  // Remove the DThreads
26  remove_thread_template(t1_TID);
27  remove_thread_template(t2_TID);
28
29  notify_termination();  // Notify that I am the last DThread instance
30 }
```

Listing 5.6: Matrix Multiplication implemented using the MiDAS C API: DThreads code.

GEORGE MATHEOU

```
1  // Global Variables
2  float *A, *B, *C;

4  // The main program
5  int main(){
6   // Initialize the TSU
7   init_processor(NUM_OF_CORES);

9   // Memory Allocations
10  A = (float*) malloc(N*N * sizeof(float));
11  B = (float*) malloc(N*N * sizeof(float));
12  C = (float*) malloc(N*N * sizeof(float));

14  // Initializations goes here ...

16  // Create Thread Template for thread_1
17  t1_TID = dthread_create_multiple(thread_1, 1, sch_dynamic, 0);

19  // Create Thread Template for thread_2
20  t2_TID = dthread_create_simple(thread_2, N, sch_static, core_0);

22  // Multiple Update (from 0 to N-1)
23  if(is_master())
24   multiple_update(t1_TID, 0, N-1);

26  run_explicit();  // Start the DDM scheduling

28  finalize(); // Deallocates the resources

30  return 0;
31 }
```

Listing 5.7: Matrix Multiplication implemented using the MiDAS C API: main program.

Listing 5.7 illustrates the main function of the DDM program. The MiDAS processor is initialized in line 7 where NUM_OF_CORES are enabled. After the arrays (A, B and C) are allocated and initialized, the Thread Templates are created and loaded in the TSU. *thread_1* is a Multiple DThread with RC=1 and its scheduling policy is set to dynamic. *thread_2* is a Simple DThread with RC=N because it needs to wait the N instantiations of *thread_1* to finish their execution. *thread_2* is scheduled to be executed on the core with ID=0 (scheduling method=static and scheduling value=0). Once the DThreads are loaded, the N instantiations of *thread_1* are released by using the *multiple_update* command (line 24). The Multiple Update command is executed only once. To achieve this, we use the *is_master* function. The *run_explicit* function is used to execute the ready DThread instances. This function will be returned when the *thread_2* DThread completes executing the *notify_termination* function. Finally, the API resources are deallocated by the *finalize* function.

5.5.2 Implementation using the C++ API

Listing 5.8 depicts the Matrix Multiplication application using the C++ API. The main difference between the C++ and C implementations is that in the latter case the DThreads are created using C++ objects and updated using Update methods. Also, the DThreads are removed from the TSU using the C++ *delete* operator.

GEORGE MATHEOU

```
1  // Includes ...
2  using namespace ddm;

4  // Global Variables
5  float *A, *B, *C;
6  MultipleDThread *t1;
7  SimpleDThread *t2;

9  // Declare the code of thread_1
10 void thread_1(ContextArg cntx){
11  // Local Variables
12  int j, k;
13  float res;

15  // Executes the algorithm using the context as i
16  for (j = 0; j < N; j++)
17   for (k = 0; k < N; k++)
18    C[cntx][j] += A[cntx][j] * B[k][j];

20  t2->update(); // Update the thread_1's consumer, i.e. thread_2
21 }

23 // Declare the code of thread_2
24 void thread_2(){
25  // Print results goes here ...

27  // Remove the DThreads
28  delete t1;
29  delete t2;

31  notify_termination();  // Notify that I am the last DThread instance
32 }

34 int main(){ // The main program
35  ddm::init_processor(NUM_OF_CORES);  // Initialize the TSU

37  // Memory Allocations (A, B and C) using malloc or C++'s new keyword

39  // Create Thread Template for thread_1
40  t1 = new MultipleDThread(thread_1, 1, sch_dynamic, 0);
41  // Create Thread Template for thread_2
42  t2 = new SimpleDThread(thread_2, N, sch_static, core_0);

44  // Multiple Update (from 0 to N-1)
45  if(is_master())
46   t1->update(0, N-1);

48  ddm::run_explicit(); // Start the DDM scheduling
49  ddm::finalize(); // Deallocates the resources

51  return 0;
52 }
```

Listing 5.8: Matrix Multiplication implemented using the MiDAS C++ API.

5.5.3 Implementation using TFlux directives

Listing 5.9 depicts the Matrix Multiplication application in DDM format, using TFlux directives. In this case, the TFlux source-to-source compiler takes as input the C application and produces the code that targets the MiDAS system, i.e., the same code we shown in Listings 5.6 and 5.7. Notice that currently the source-to-source compiler does not generate C++ code. Additional details about the TFlux source-to-source compiler can be found in [170].

GEORGE MATHEOU

The *#pragma ddm for thread* directive in line 15 creates DThread 1 (*thread_1*) that will parallelize the outer for-loop of the algorithm. The *start* keyword releases the N instantiations of DThread 1 (spanning from 0 to N-1). The *nesting* keyword defines the loop nesting level of the DThread. When this keyword is omitted, the TFlux compiler sets the DThread's Nesting to zero. The *#pragma ddm endfor* directive in line 20 closes the *#pragma ddm for thread* directive, and also it updates the DThread 2 (*thread_2*) in each for-loop iteration. DThread 2 is declared in line 22 by the *#pragma ddm thread* directive. The *kernel* keyword enforces the scheduling policy. The static method is used in core with ID=0 for DThread 2. When the *kernel* keyword is omitted, the dynamic scheduling policy will be enforced. Furthermore, the *readycount* keyword defines the RC value of the DThread. Finally, the *end* keyword indicates the last DThread of the program. It notifies the compiler to add the appropriate functions to remove the DThreads, to deallocate the resources occupied by the application, and to invoke the notify termination function.

```
1  // Global Variables
2  float*  A;
3  float*  B;
4  float*  C;

6  // The main program
7  int main(){
8   // Memory Allocations
9   A = (float*) malloc(N*N * sizeof(float));
10  B = (float*) malloc(N*N * sizeof(float));
11  C = (float*) malloc(N*N * sizeof(float));

13  // Initializations goes here ...

15  #pragma ddm for thread 1 start (0 :  N-1) nesting 1
16  for (i = 0; i < N; i++)
17   for (j = 0; j < N; j++)
18    for (k = 0; k < N; k++)
19     C[i][j] += A[i][k] * B[k][j];
20  #pragma ddm endfor update (2)

22   #pragma ddm thread 2 kernel (static, core_0) readycount N end
23   // Print results goes here ...
24   #pragma ddm endthread

26  return 0;
27 }
```

Listing 5.9: Matrix Multiplication implemented using TFlux directives.

GEORGE MATHEOU

Chapter 6

Recursion Support for the DDM model

6.1 Introduction

In this section we explored mechanisms needed for supporting recursion in DDM by studying two different data-flow models, the U-Interpreter [64] and the Packet Based Graph Reduction (PBGR) [65, 66, 67]. The two models are presented in Section 6.2. The proposed mechanisms for supporting recursion in DDM are presented in Section 6.3. As a proof of concept, the proposed mechanisms were implemented in FREDDO. Section 6.4 introduces the DThread classes that were developed under FREDDO's API in order to provide single-node and distributed recursion support. Finally, Section 6.5 presents the implementation of the recursive Fibonacci algorithm using FREDDO.

6.2 The U-Interpreter and PBGR models[1]

6.2.1 The U-Interpreter model

The DDM model is based on the U-Interpreter [64] and it uses token tagging to distinguish between different instantiations of a static code template. In the U-Interpreter model, each instance of the execution of an operator is called *activity* which has a unique name. Each token/value is combined with the name of its destination activity into a packet which is called *tagged token*. An activity name consists of four fields <u.c.s.i> where *u* is the context field, *c* is the code block name, *s* is the instruction number and *i* is the initiation/iteration number. A procedure activation is implemented using four basic operators: *A*, *BEGIN*, *END* and A^{-1}. *A* creates a new context u'. *BEGIN* receives the *A*'s output and it replicates tokens for each fork. *END* sends the result to A^{-1}. Finally, A^{-1} replicates its output for its successors. Figure 48 depicts a simple program which outputs the square of a number. Although this example does not have any parallelism or recursion, it helps with the explanation of more general mechanisms. There are three features that should be provided by a general mechanism:

[1] Based on a document prepared by Professor Ian Watson for the TERAFLUX project [151].

GEORGE MATHEOU

(1) create a separate thread for a function call, (2) return the result from a particular function call to different call sites and (3) distinguish different instantiations of parallel function calls.

```
int square(int n){
 int result;
 result = n*n;
 return result;
}

void main(){
 int s = square(4);
 printf("result:%d\n",s);
}
```

Figure 48: A program that computes the square of a number.

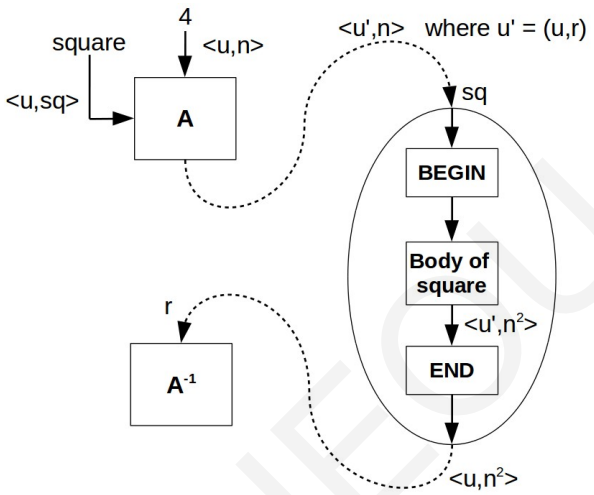


Figure 49: U-Interpreter graph of the square function call.

Figure 49 depicts the U-Interpreter graph of the square function of Figure 48. The *A* node is a representation of the action at the call site. The function *square* and the argument 4 are shown as two tokens <u.sq> and <u.n> both with a context *u*. The *A* operation constructs a new token <u',n> which is directed to the input of the square function (sq). The U-interpreter description uses the (u,r) pair as a new context. The call site has the information of the location r (the address of the node to which the result will be directed) and can clearly construct the new context from the information which it has locally. The reason it does this is threefold. Firstly, the new context must be unique and contexts constructed from this data must satisfy this. Secondly, it is necessary to pass the return link to the function so that it knows where to return its result. Thirdly, it is necessary to pass the old context to the function so that it can restore this old context to the return value.

Having executed the body of the function in the new context, the *END* operator takes the result, replaces the old context and directs the resulting token to the return link address. In order to explore further the mechanisms that are needed for supporting recursion in DDM, it is worth looking at the Fibonacci algorithm:

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

Although this function it's not an efficient implementation of the Fibonacci algorithm, it has the complexity of double recursion. The basic U-interpreter representation of the Fibonacci algorithm is shown in Figure 50. The recursive calls to the Fibonacci function are depicted as dotted instantiations.

GEORGE MATHEW

If we want to generate new threads to execute the recursive calls in parallel, it will be necessary to split the final add operation from the rest of the body as a continuation since we cannot execute it until the recursive calls have returned. The graph shows how the computation could be split into two threads: T1 and T2. The execution of T1 will cause the creation of the continuation T2 and two new frames will be created for the recursive calls of T1.

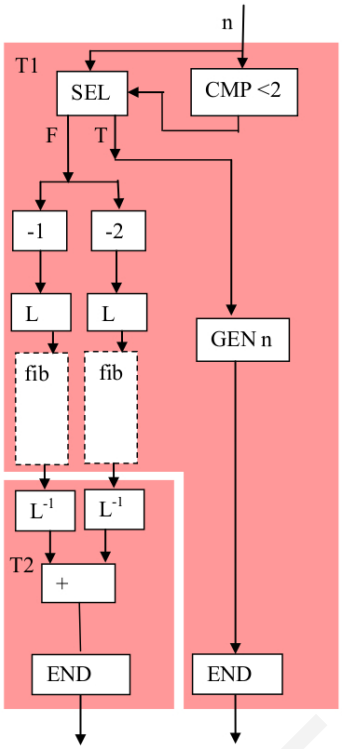


Figure 50: U-Interpreter graph of Fibonacci.

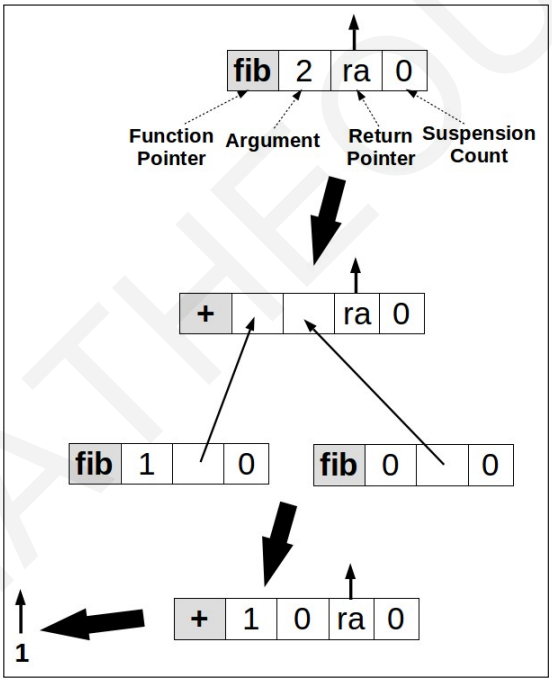


Figure 51: PBGR evaluation of fib(2).

6.2.2 The PBGR model

The Packet Based Graph Reduction (PBGR) approach was used in the ALICE [65] and Flagship [66, 67] projects back in the 1980s. PBGR generates new sections of data-flow graph dynamically to instantiate the body of a function. In PBGR, a *packet* (or thread descriptor) contains the following information: a function pointer (f), arguments (a1, a2, ...), a return pointer (ra) and a suspension count (sc). The computation is in the form of a graph of packets held in memory which is shared between cores that perform computation. A packet is *active* if its suspension count (sc) is zero. A scheduling queue contains the addresses of all active packets. The process of computation (or graph reduction) involves cores taking addresses of active packets from the scheduling queue and operating on their contents as determined by the code referenced by the function pointer. This may involve simply performing computation on the arguments and returning a value (to the place pointed to by ra) or it may involve the construction of a new piece of graph.

GEORGE MATHEOU

Figure 51 illustrates the PBGR evaluation of the Fibonacci algorithm for n=2. The evaluation starts with a single active packet. We assume that this call will return a value to an outer level computation (e.g., a print function). The fib(2) example involves only one packet reduction with the creation of two calls which will in turn be executed but return a value and generate no further packets. The first *fib* packet generates three new packets: two *fib* packets and one *add* packet. The *add* packet is suspended on two values which will be provided via the ra fields of the two *fib* packets which are immediately created as active (i.e., with sc=0 and an entry placed on the scheduling queue). Assume that the two *fib* packets execute in parallel. Because of their argument values, they immediately return a value (one packet will return 1 and the other packet will return 0) and decrement the sc of the *add* packet. When the *add* packet becomes active, it will be executed to produce the value 1 ($1 + 0 = 1$) which is then returned to the outer level computation.

6.2.3 Differences between DDM and PBGR

DDM and PBGR have two main differences. Firstly, PBGR holds the synchronization (or suspension) count in the packet/frame allocated to hold the arguments and context of the particular function instantiation. In contrast, DDM uses a separate synchronization memory area, called Synchronization Memory (SM). Secondly, DDM has a Thread Template which stores the pointer of the executable code (called Instruction Frame Pointer or IFP) where PBGR stores a pointer to code directly. In PBGR, all knowledge about how to construct any new graph associated with a function call is embedded in the code.

6.3 Basic functionalities for supporting recursion in DDM

In this section we will describe the basic functionalities and data-structures that are needed for providing recursion support in DDM, using the Fibonacci algorithm as an example. According to the U-Interpreter graph of Figure 50, two different threads (T1 and T2) are required for supporting the parallel execution of the Fibonacci algorithm. T1 will be responsible for spawning recursive calls while T2 will be responsible for summing/reducing the return values of children-calls and return the results to parent-calls. More specifically, in the case of $n > 2$, an instance of T1 (parent-call) will spawn two additional instances of T1 (children-calls). When the children-calls finish their execution, an instance of T2 will be responsible for summing the return values of the children-calls and sending the result ($fib(n-1) + fib(n-2)$) to the parent-call.

Implementing thread T1 in DDM

For implementing the thread T1 in DDM the following functionalities are required:

GEORGE MATHEOU

1. ***Allowing multiple instances of the same thread:*** in DDM this functionality can be provided by utilizing the Context attribute which is based on the U-Interpreter model.

2. ***A mechanism that will allow the spawning of recursive function calls:*** this can be done by using a DThread where its instances will be responsible for executing the recursive function calls. An instance can spawn additional recursive calls using an Update command.

3. ***A mechanism that will allow T1 to behave as a regular function:*** currently, in the DDM model, each DThread has a block of instructions. The Instruction Frame Pointer (IFP) is used to point to the address of the first instruction of the block. When an instance of the DThread is ready for execution, the TSU uses the IFP to execute its code. The code takes as input data only the Context of the instance. Each recursive instance of T1 needs to behave as a regular function, i.e, to have an argument list (AL) and a return value (RV). These attributes are also used in the packets of the PBGR approach. A special data-structure (called *RData*) is needed to hold the AL and RV of each recursive instance. An array can be used for recursive functions where their number of instances is known at compile-time (Figure 52). Each element of the array corresponds to a different instance. An instance is able to manage (read/write) its RData entry by using its Context value. Notice that in case of the Fibonacci algorithm, the AL consists of a single integer value (n). The RV is also an integer value.

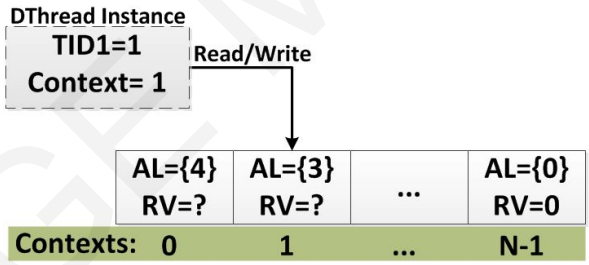


Figure 52: RData implemented as a fixed-size array.

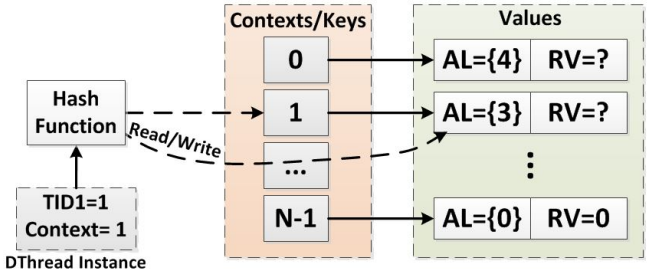


Figure 53: RData implemented as a hash-map.

For recursive functions where their number of instances is not known at compile-time, a hash-map data-structure can be used (Figure 53). Accessing an RData entry is an associative operation based on the Context value. The allocation/deallocation of RData entries can be

GEORGE MATHEOU

performed as the execution proceeds by the instances, in parallel. However, this requires an efficient hash-map implementation which allows concurrent insert and delete operations.

4. ***A mechanism that will guarantee that all recursive instances of T1 will have unique Context values, at runtime:*** the instances of T1 run concurrently and each instance can access its RData entry using its Context value. As such, a mechanism is needed to assign unique Context values to the instances. For this functionality we are proposing two different methods:

   - ***Method 1:*** assign Context values to the children-calls based on the Context of the parent-call. In the case of Fibonacci, the recursive calls construct a binary tree of executions. We can simply borrow an idea from the binary heap which is implemented in an array, where the root is at index zero. Following this idea, the children-calls will have the following Context values:

     (a) Child_1's Context = $2 * Parent's\ Context + 1$

     (b) Child_2's Context = $2 * Parent's\ Context + 2$

     However, this method cannot be used as a general solution for assigning unique Context values to the instances since it depends exclusively on the algorithm.

   - ***Method 2:*** use a global atomic variable/counter that holds the next available Context value. When a parent-instance spawns a child-instance, the Context value of the child-instance will be equal to the value of the counter. After that, the counter's value will be increased by one.

5. ***Create a Thread Template for T1:*** a new Thread Template has to be created for T1 with Nesting=1 and RC=1. This is because T1 has multiple instances where each instance has only one producer, its parent-instance. Finally, T1 will have two consumers, T1 and T2. T1 is a consumer because a parent-instance can send an Update to a child-instance of the same DThread. T2 is also a consumer because an instance of T1 can send an Update to an instance of T2. This is needed in order to spawn an instance of T2 to process the results of the children-calls when all of them finished their execution.

Implementing thread T2 in DDM

Each instance of T2 corresponds to one parent-call of T1. As such, a T2's instance and its associated T1's instance can have the same Context value. This simplifies the procedure of assigning unique Context values to T2's instances. A T2's instance is responsible for processing the return values of the children-calls of a parent-call. The result of the sum operation is actually the return value of the parent-call. For this functionality T2 thread needs to access the RData data-structure of T1 in order to write the return value of the parent-call to its RData entry. Finally, a new Thread Template has to

GEORGE MATHEOU

be created for T2 with Nesting=1 and RC=2. This is because T2 has multiple instances where each instance has two producers which are children-instances of T1.

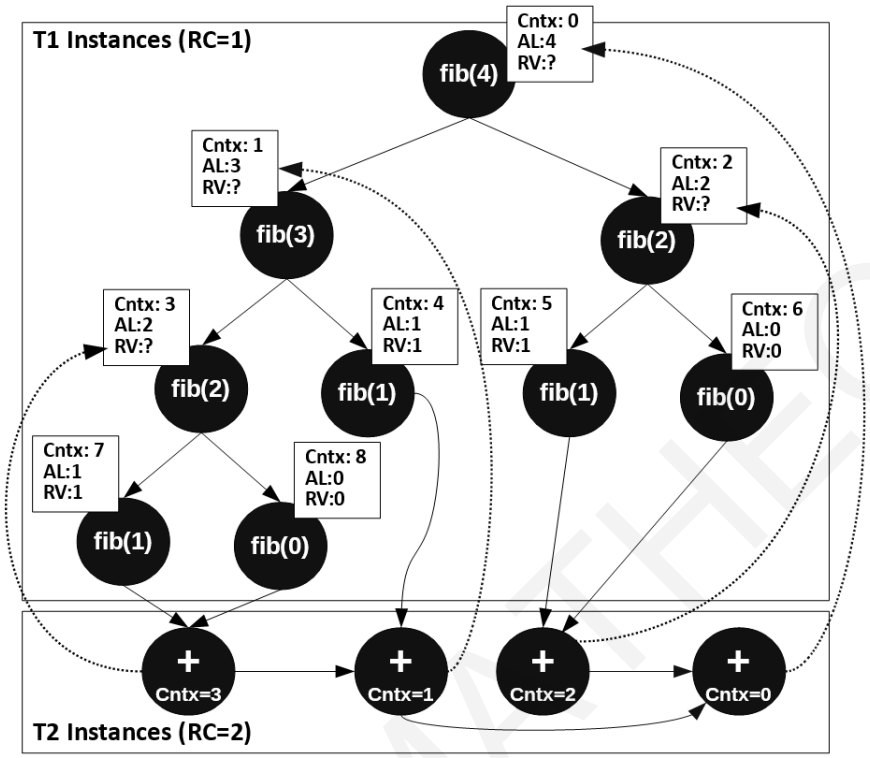


Figure 54: The Fibonacci's DDM Dependency Graph.

The DDM Dependency Graph of the Fibonacci algorithm

Figure 54 depicts the dependency graph of the Fibonacci algorithm with $n = 4$, based on the aforementioned functionalities/mechanisms. The graph consists of two DThreads, T1 and T2. Solid arrows indicate Update operations and dotted arrows indicate write operations to the *RV* attribute of T1's instances.

DDM pseudo-code of the Fibonacci algorithm

The DDM pseudo-code for the Fibonacci algorithm is depicted in Listing 6.1. It includes all the necessary functions and data-structures that are needed for supporting recursion in the DDM model. In this example, we are using an RData data-structure implemented as a fixed-size array. The size of the RData is equal to 2^n. This is because the total number of nodes of the binary tree of Fibonacci(n) is about 2^n. For generating unique Context values for T1's instances we are using the Context-based approach (Method 1). The *GET_PARENT* macro is used to calculate the Context of a parent-instance based on the Context value of a child-instance. This formula is also borrowed by the binary heap implementation. This special macro is required for sending Update commands to the parents of the

GEORGE MATHEOU

currently executed instances. Another solution to this requirement is to store the parent's Context of a child-instance in its RData entry. However, this solution will increase the memory consumption of the program.

```
#define GET_PARENT(C) floor((c-1) / 2)
RData rdata; // The RData (implemented as an array)

// The Code of the T1 DThread (ThreadID=2)
T1_code(Context cntx) {
 // Get the AL of the current instance, i.e. n
 n = GET_ARG_LIST(rdata, cntx);

 // Return n to the parent of the current instance
 if (n == 0 || n == 1) {
  SET_RETURN_VALUE(rdata, cntx, RV=n); // Set the RV of the current instance
  UPDATE(TID=3, Context=GET_PARENT(cntx)); // Update T2 instance of the parent
  return; // Do not call any children
 }

 // Call fib (n-1)
 SET_ARG_LIST(rdata, Context=2*cntx + 1, n-1);
 UPDATE(TID=2, Context=2*cntx + 1);

 // Call fib (n-2)
 SET_ARG_LIST(rdata, Context=2*cntx + 2, n-2);
 UPDATE(TID=2, Context=2*cntx + 2);
}

// The Code of the T2 DThread (ThreadID=3)
T2_code(Context cntx) {
 c1_RV = GET_RETURN_VALUE(rdata, 2*cntx + 1); // Get the RV of the first child
 c2_RV = GET_RETURN_VALUE(rdata, 2*cntx + 2); // Get the RV of the second child
 SET_RETURN_VALUE(rdata, cntx, RV=c1_RV + c2_RV); // Set the parent's RV
 UPDATE(TID=3, Context=GET_PARENT(cntx)); // Update the parent instance
}

// The main function
main(){
 size = power(2, n);      // The maximum number of instances
 ALLOCATE(rdata, size);  // Allocate elements for the RData

 // Load T1 & T2 DThreads in TSU
 ADD_T1(TID=2, IFP=T1_code's address, RC=1, Nesting=1, Consumers={T1, T2});
 ADD_T2(TID=3, IFP=T2_code's address, RC=2, Nesting=1, Consumers={T2});

 // Call the intance with Context=0 (Root instance)
 SET_ARG_LIST(rdata, Context=0, n);
 UPDATE(TID=2, Context=0);

 START_DDM_SCHEDULING();

 // The RV of Context 0 (the root) holds the result
 result = GET_RETURN_VALUE(rdata, 0);
 PRINT result;
}
```

Listing 6.1: DDM pseudo-code for the Fibonacci algorithm.

6.4 DThread Classes for Recursion Support in FREDDO

The recursion support for the DDM model is implemented under the FREDDO framework. However, we expect that our techniques/mechanisms can be implemented in other DDM implementations such as DDM-VM and MiDAS. FREDDO's API was extended with four additional classes for

GEORGE MATHEOU

recursion support: *RecursiveDThreadWithContinuation*, *RecursiveDThread*, *ContinuationDThread* and *DistRecursiveDThread*. The functionalities of these classes are based on the functionalities presented in Section 6.3. *RecursiveDThreadWithContinuation* and *RecursiveDThread* can be used only for single-node execution where *ContinuationDThread* and *DistRecursiveDThread* can be used for both single-node and distributed execution.

6.4.1 RecursiveDThreadWithContinuation Class

This special template class provides functionalities for algorithms with multiple recursion. It has two template parameters, *T_ARGS* and *T_RETURN*. *T_ARGS* indicates the type of the argument(s) of each recursive call. If a recursive function has more than one argument in its argument list, a struct can be used for holding them. *T_RETURN* indicates the type of the return value. *RecursiveDThreadWithContinuation* is responsible for creating and managing a Recursive-DThread and a Continuation-DThread, i.e., MultipleDThreads with additional functionalities.

The provided class holds the Argument List (*AL*) and the Return Value (*RV*) attributes of each recursive instance and it assigns unique Contexts at runtime. The *AL* and *RV* attributes of all instances are stored in an RData data-structure. RData is a dynamically allocated array (in the heap section of the program). Thus, *RecursiveDThreadWithContinuation* class is used when the number of instances of the recursive function is known at compile-time. Each recursive instance is associated with an RData entry which holds the following attributes: *RV*, *AL*, *PContext* and *CVector*. *PContext* is the Context value of the parent. CVector is a C++ vector data-structure which holds the Context values of children of a specific instance. The following constructors/methods are provided:

- **RecursiveDThreadWithContinuation(MultipleDFunction dFunction, UInt maxNumInstances, MultipleDFunction rFunction, UInt numOfChildren)**: inserts a DThread in the TSU that implements multiple recursion. The user has to specify the DFunctions of the recursive code (*dFunction*) and the continuation code (*rFunction*). Also, the maximum number of instances (*maxNumInstances*) of the DThreads has to be specified. For example, in Fibonacci, this number is equal to 2^n. Finally, the user has to specify the maximum number of children (*numOfChildren*) of a parent. For example, in a double recursion algorithm, like Fibonacci, the maximum number of children is 2.
- **T_ARGS* getArguments(RInstance rinst)**: returns the AL of a specific recursive instance. The *RInstance* indicates the Context of a recursive instance.
- **T_RETURN getReturnValue(RInstance rinst)**: returns the RV of a specific instance.
- **T_RETURN getRootReturnValue()**: returns the RV of the root instance.
- **void callChild(RInstance parentInstance, T_ARGS& args)**: spawns a child-instance, where *parentInstance* is the child's parent-instance and *args* is the child's AL. When this function is

GEORGE MATHEOU

called it creates a unique Context value for the child-instance, using C++'s atomic operations (the *fetch_add* routine is used). The Context value of the child-instance will be stored in the CVector of its parent. After that, a new RData entry will be created and an Update command will be sent to the TSU for spawning the child-instance.

- **void callRoot(T_ARGS& args)**: spawns the root recursive call with its arguments.
- **vector<RInstance>& getMyChilds(RInstance rinst)**: returns the Context values of the children of *rinst*.
- **void returnValueToParent(RInstance rinst, T_RETURN value)**: returns the value of a child-instance with Context=*rinst* to its parent. In particular, the return value will be stored in the RV attribute of the RData entry of the child-instance (*rinst*). Finally, an Update will be sent to the continuation instance of the parent-call.
- **void updateContinuationInstance(RInstance rinst)**: updates an instance of the Continuation-DThread directly.

6.4.2 RecursiveDThread and ContinuationDThread Classes

For recursive functions that their number of instances is not known at compile time, we are providing two special classes: *RecursiveDThread* and *ContinuationDThread*. The programmer is responsible for allocating/deallocating the arguments and the return values of the instances at runtime. A DDM user can use a *RecursiveDThread* along with a *ContinuationDThread* to implement an algorithm with multiple recursion (or any similar algorithm). Also, *RecursiveDThread* can be used as a standalone class for other types of recursion, such as linear, tail, and so on. RecursiveDThread and ContinuationDThread classes utilize a special template class, called *RData*, which correlates parent-instances with children-instances. Each recursive call is associated with an RData object. RData holds the arguments of a recursive call, pointers to the return values of its children (if any) and a pointer to the RData of its parent.

6.4.2.1 Constructors/methods of RData Class

- **RData(T_ARGS arg, RInstance parentInstance, RData* parentRData, unsigned int numChilds)**: constructs a new RData object. *T_ARGS* indicates the type of the argument(s) of each recursive call.
- **T_ARGS getArgs()**: returns the arguments of the recursive call.
- **RInstance getParentInstance()**: returns the Context value of the parent recursive call.
- **RData* getParentRData()**: returns a pointer to the parent's RData.
- **void addReturnValue(T_RETURN value)**: sends the return value of the child to the parent.

GEORGE MATHEOU

- **T_RETURN sum_reduction()**: applies sum reduction to the return values of the children of this instance/call and it returns the result.
- **T_RETURN* getChildrenReturnValues()**: returns the return values of the children of this instance/call.
- **void returnValueToParent(T_RETURN value, ContinuationDThread* contDThread)**: sends the return value to the parent of this instance/call. Also, it Updates the RC value that corresponds to the parent's Continuation instance. This is required in order to notify a parent instance that all its children have returned.
- **bool hasParent()**: indicates that this instance has a parent.

6.4.2.2 ContinuationDThread Class

A ContinuationDThread object accesses an RData object through its DFunction, called ContinuationDFunction. ContinuationDFunction has two input arguments, the Context value (called RInstance) and a pointer (*void**) to the RData object. Constructors/methods:

- **ContinuationDThread(ContinuationDFunction cDFunction, ReadyCount readyCount, UInt numOfInstances)**: inserts a ContinuationDThread in the TSU, where a StaticSM will be used.
- **ContinuationDThread(ContinuationDFunction cDFunction, ReadyCount readyCount)**: inserts a ContinuationDThread in the TSU, where a DynamicSM will be used.
- **void update(RInstance rinst, void* rdata)**: decrements the RC value of this DThread that corresponds to the *rinst*. *rdata* is a pointer to the RData object of the instance that is going to be updated. *rdata* is used by the TSU when a continuation-instance is ready for execution. In particular, *rdata* will be used as an input argument in the ContinuationDFunction call of the ready continuation-instance.

6.4.2.3 RecursiveDThread Class

FREDDO provides a separate DFunction for the RecursiveDThread class, called RecursiveDFunction. RecursiveDFunction has the same interface as the ContinuationDFunction. The following constructors/methods are provided:

- **RecursiveDThread(RecursiveDFunction rDFunction)**: inserts a RecursiveDThread in the TSU, where a DynamicSM will be used.
- **RInstance callChild(RData<T_ARGS, T_RETURN>* rdata)**: spawns a recursive child where *rdata* is a pointer to the child's RData object. This task includes the creation of a unique Context for the child, using C++'s atomic operations (the *fetch_add* is used) and sending an

GEORGE MATHEOU

Update command to the TSU targeting the child-instance. The Update command consists of the Context value and the *rdata* of the child.

6.4.3 Distributed Recursion Support

For supporting distributed execution of recursive algorithms, in a data-driven manner, we have extended the functionalities of RecursiveDThread, through the *DistRecursiveDThread* class, and we provide an enhanced RData class, called *DistRData*. Each recursive call of the DistRecursiveDThread is associated with a DistRData object. DistRData holds the arguments of a recursive call, pointers to the return values of its children (if any) and a pointer to the DistRData of its parent. Thus, the DistRData objects correlate children recursive instances with their parents.

When a parent-instance calls one or more children-instances, the Distributed Scheduling Unit (DSU) decides which of them will be executed on remote nodes, based on their Context values. In this case, the Network Manager will send the DistRData objects and the children-instances' Context values to the remote nodes in order to be scheduled for execution. When a child-instance returns a value to its parent, the runtime system acknowledges if the parent-instance is mapped on the local node or on a remote node. In the latter case, the return value is sent via a network message to the remote node and finally, it is stored in the parent's DistRData object.

6.4.3.1 Constructors/methods of DistRData Class

- **DistRData(void* arg, RInstance parentInstance, DistRData* parentData, unsigned int numChilds)**: creates a new DistRData object.
- **void* getArgs()**: returns the argument(s) of the recursive instance.
- **RInstance getParentInstance()**: returns the Context value of the parent instance.
- **DistRData* getParentRData()**: returns a pointer to the DistRData object of the parent instance.
- **void addReturnValue(void* value)**: adds the return value of a child instance (the owner of this DistRData object is the parent).
- **T_RETURN sum_reduction()**: applies sum reduction to the return values of the children of this instance/call and it returns the result.
- **T_RETURN** getChildrenRVs()**: returns the return values of the children instances.
- **bool hasParent()**: indicates if the recursive instance has a parent.
- **void makeParentRemote()**: informs the DistRData object that its parent resides on a remote node. This method is called by the Network Manager when it constructs a new DistRData object for a child that resides on a different node from its parent.

GEORGE MATHEOU

- **bool isMyParentRemote()**: indicates if the DistRData object's parent resides on a remote node.
- **unsigned int getNumberOfChildrenRVs()**: returns the number of children of this DistRData object.

6.4.3.2 Constructors/methods of the DistRecursiveDThread Class

- **DistRecursiveDThread(RecursiveDFunction rDFunction)**: constructs a new DistRecursiveDThread object.
- **DistRecRes callChild(void* args, size_t argsSize, RInstance parentInstance, DistRData* parentRData, unsigned int numChilds)**: calls an instance of the recursive function. *numChilds* is the number of children of the instance that is going to be created. It returns a *DistRecRes* value which holds the Context and a pointer to the DistRData object of the new child. FREDDO assigns unique Context values to the children instances using two techniques:
  1. It uses an atomic variable/counter in order to create unique Context values locally.
  2. The peer/node ID is stored in the leftmost 12 bits of each Context value. This creates unique Context values across the entire distributed system.
- **void returnValueToParent(void* value, size_t valueSize, ContinuationDThread* contDThread, DistRData* rdata)**: returns a value to the parent call which may reside on a remote node. *rdata* is a pointer to the DistRData object of the child.

6.5 Implementing the recursive Fibonacci algorithm in FREDDO

In this section we present the recursive Fibonacci algorithm, implemented in FREDDO, targeting single-node and distributed systems.

6.5.1 Implementation using the RecursiveDThreadWithContinuation Class

The Fibonacci algorithm implemented using the *RecursiveDThreadWithContinuation* class, is shown in Listing 6.2. The presented code is equivalent to the pseudo-code of Listing 6.1. In the pseudo-code we used a Context-based approach to generate unique Context values whereas in the presented example an atomic counter is used. In the latter case, the atomic counter is managed implicitly by the *RecursiveDThreadWithContinuation* class.

6.5.2 Implementation using the RecursiveDThread and ContinuationDThread Classes

Listing 6.3 depicts the Fibonacci implementation using the RecursiveDThread and ContinuationDThread classes. Notice that the programmer is responsible for creating the RData objects of the

GEORGE MATHEOU

child-instances (lines 22 and 24). After the *run* function is executed, a sum reduction is performed to the return values of the children of the rootRData in order to calculate the final result (line 51).

```
1  #include <recursive_dthreads.h>
2  using namespace ddm;
3  RecursiveDThreadWithContinuation<int, int> *fib_dt;

5  void fib_code(RInstance context) { // The Fibonacci Code
6   int n = *(fib_dt->getArguments(context)); // Get the arguments (n)

8   if (n == 0 || n == 1) {
9    fib_dt->returnValueToParent(context, n); // Return n
10   return;
11  }

13  int n1 = n-1, n2 = n-2;
14  fib_dt->callChild(context, n1); // Call fib (n-1)
15  fib_dt->callChild(context, n2); // Call fib (n-2)
16 }

18 void continuation_code(RInstance context) {  // The Continuation Code
19  // result holds the sum of the Return Values of the children-instances
20  int result = 0;

22  // Get the Return Value of each child and add it to the result variable
23  for (auto i : fib_dt->getMyChilds(context))
24   result += fib_dt->getReturnValue(i);

26  // fib(n) = fib(n-1) + fib(n-2)
27  fib_dt->returnValueToParent(context, result);
28 }

30 // The main program
31 void main(int argc, char* argv[]){  // Initializiations goes here ...
32  ddm::init(NUM_KERNELS); // Initializes the DDM execution environment
33  int maxInstances = pow(2, n);

35  // Create the DThread for recursion support
36  fib_dt = new RecursiveDThreadWithContinuation<int, int>(fib_code,
        maxInstances, continuation_code, 2);

38   // Call the root instance, i.e. the instance with Context=0
39  fib_dt->callRoot(n);
40  ddm::run(); // Start DDM scheduling
41  cout << fib_dt->getRootReturnValue() << endl; // Print the result
42  delete fib_dt;
43 }
```

Listing 6.2: The recursive Fibonacci algorithm implemented in FREDDO using the RecursiveDThreadWithContinuation class.

6.5.3 Distributed Implementation

Listing 6.4 depicts the distributed implementation of the Fibonacci algorithm. In the distributed implementation, the *callChild* routine has a different interface compared to the one presented in the previous example. In the former case, the user should provide the argument list and its size (in bytes), the parent's Context value, a pointer to the parent's DistRData object and the number of children. The *DistRecursiveDThread* will then create a DistRData object for the child, implicitly. Also notice that the two basic functions of *DistRecursiveDThread* (*callChild* and *returnValueToParent*) require

GEORGE MATHEOU

the size of input data (arguments or return value) in bytes. This information is needed by FREDDO when data will be sent to a remote node.

```
#include <recursive_dthreads.h>
using namespace ddm;

using T = long;

// Declare DThread Objects
RecursiveDThread* rDThread;
ContinuationDThread* cDThread;

// The DThread that executes the recursive function calls
void fib_code(RInstance context, void* data) {
 auto rd = (RData<T,T>*) data;
 auto n = rd->getArgs();

 // A leaf node, so we should return the value to parent for summing
 if (n == 0 || n == 1) {
  rd->returnValueToParent(n, cDThread); // Send the RV to my parent
  return;
 }

 // Call fib (n-1)
 rDThread->callChild<T,T>(new RData<T,T>(n-1, context, rd, 2));
 // Call fib (n-2)
 rDThread->callChild<T,T>(new RData<T,T>(n-2, context, rd, 2));
}

// The Continuation DThread
void continuation_code(RInstance context, void* data) {
 auto rData = (RData<T,T>*) data;

 // Sum the results of my children
 auto sum = rData->sum_reduction();
 rData->returnValueToParent(sum, cDThread);
}

void main(int argc, char* argv[]) {
 RData<T,T>* rootRData = nullptr;
 ddm::init(kernels); // Initializes the DDM execution environment

 // Create the DThread Objects
 rDThread = new RecursiveDThread(fib_code);
 cDThread = new ContinuationDThread(continuation_code, 2);

 // Call the root instance
 rootRData = new RData<T,T>(n, 0, nullptr, 2);
 rDThread->callChild<T,T>(rootRData);

 ddm::run();  // Start the DDM scheduling

 // Print the result
 T res = rootRData->sum_reduction();
 cout << "Result: " << res << endl;

 // Releases the resources of the DDM environment
 delete rDThread;
 delete cDThread;
 ddm::finalize();
}
```

Listing 6.3: The recursive Fibonacci algorithm implemented in FREDDO using the RecursiveDThread and ContinuationDThread classes.

GEORGE MATHEOU

```cpp
#include <recursive_dthreads.h>
using namespace ddm;

using T = long;

// Declare DThread Objects
DistRecursiveDThread* rDThread;
ContinuationDThread* cDThread;

// The DThread that executes the recursive function calls
void fib_code(RInstance context, void* data) {
 auto rd = (DistRData*) data;
 T n = *((T*) rd->getArgs());

 // A leaf node, so we should return the value to parent for summing
 if (n == 0 || n == 1) {
  rDThread->returnValueToParent(new T {n}, sizeof(T), cDThread, rd);
  return;
 }

 // Call fib (n-1)
 rDThread->callChild(new T {n-1}, sizeof(T), context, rd, 2);
 // Call fib (n-2)
 rDThread->callChild(new T {n-2}, sizeof(T), context, rd, 2);
}

// The Continuation DThread
void continuation_code(RInstance context, void* data) {
 auto rData = (DistRData*) data;

 // Sum the results of my children
 T sum = rData->sum_reduction<T>();
 rDThread->returnValueToParent(new T {sum}, sizeof(T), cDThread, rData);
}

void main(int argc, char* argv[]) {
 // Local Variables
 DistRecRes res = { };

 // Initialize Distributed FREDDO with CNI support
 freddo_config* conf = new freddo_config();
 conf->enableTsuPinning();
 conf->disableNetManagerPinning();
 conf->enableKernelsPinning();
 ddm::init("peers.txt", 1234, conf); // port=1234

 // Create the DThread objects
 rDThread = new DistRecursiveDThread(fib_code);
 cDThread = new ContinuationDThread(continuation_code, 2);

 // Build the distributed system
 ddm::buildDistributedSystem();

 if (ddm::isRoot()) {
  res = rDThread->callChild(new T {n}, sizeof(T), 0, nullptr, 2);
 }

 ddm::run(); // Start the DDM scheduling

 if (ddm::isRoot()) {
  T result = res.data->sum_reduction<T>(); // res.data is a DistRData* var
  cout << "Result: " << result << endl;
 }
}
```

Listing 6.4: Distributed Fibonacci implementation.

GEORGE MATHEOU

Chapter 7

Evaluation

7.1 Introduction

In this chapter we present the evaluation results for the DDM's architectural support (TSU), MiDAS and FREDDO. We start with the description of the benchmark suite used in this thesis (Section 7.2), followed by an overview of the hardware environment used in our experiments (Section 7.3). The evaluation of the hardware TSU and MiDAS is presented in Section 7.4. Section 7.5 presents the evaluation results of the single-node FREDDO implementation as well as performance comparisons with the OpenMP [19] and OmpSs [157, 40] frameworks. Finally, Section 7.6 presents the performance evaluation of the distributed FREDDO implementation, including comparisons with other systems as well as network traffic analysis results.

7.2 Benchmark Suite

The benchmark suite used in this thesis contains applications with different characteristics: benchmarks with simple dependency graphs, recursive algorithms and benchmarks with high complexity dependency graphs.

7.2.1 Benchmarks with simple dependency graphs

1. **Blocked Matrix Multiplication (BMMULT)**: it multiplies two block/partitioned matrices, *A* and *B*, and stores the result in matrix *C*, i.e., $C = A \times B$. The algorithm multiplies the blocks (square matrices) similarly to the original matrix multiplication.

2. **Swaptions**: it uses the Heath-Jarrow-Morton (HJM) framework to price a portfolio of swaptions. The HJM framework describes how interest rates evolve for risk management and asset liability management for a class of model. Swaptions employs a Monte Carlo (MC) simulation to compute the prices. The simulation number variable is set to 20,000. This benchmark

GEORGE MATHEOU

belongs to the PARSEC Benchmark Suite [173] and has the following characteristics: data-parallel application with coarse-grain granularity, medium working set, low data sharing and low data exchange.

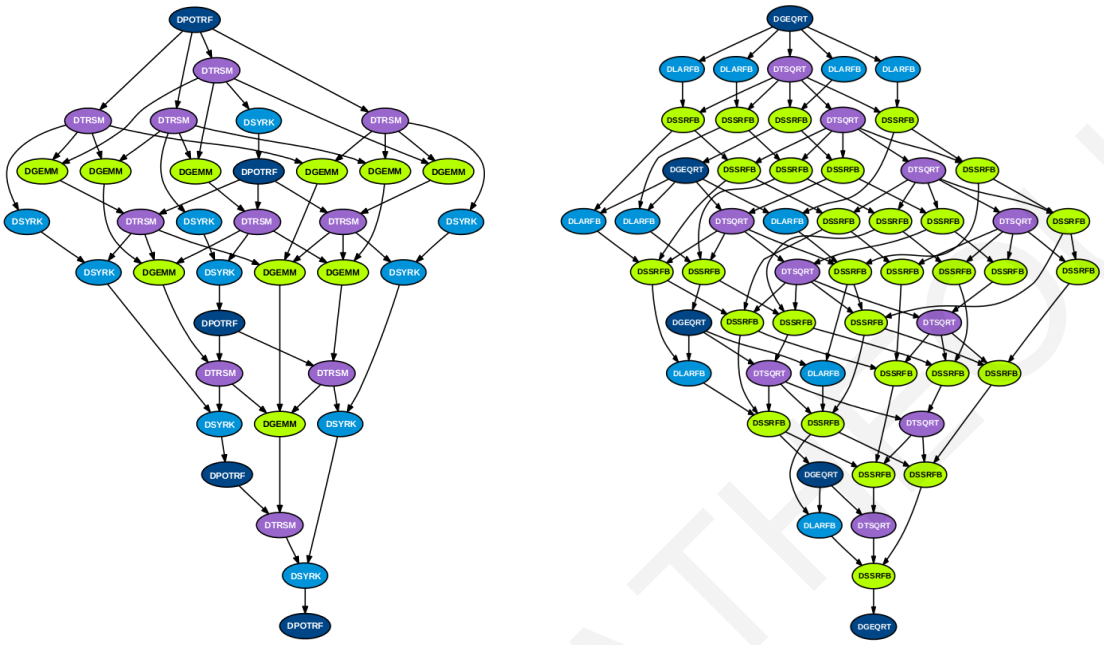
3. **Blackscholes**: it calculates the prices for a portfolio of European options analytically with the Black-Scholes partial differential equation (PDE). This benchmark belongs to the PARSEC Benchmark Suite [173]. The characteristics of the Blackscholes benchmark are: data-parallel application with coarse-grain granularity, small working set, low data sharing and low data exchange.
4. **Conv2D**: 9x9 convolution filter.
5. **Trapez**: trapezoidal rule for integration. It calculates the definite integral of a function in a given interval.
6. **Mandelbrot**: it implements the Mandelbrot set algorithm [174]. The Mandelbrot set is the set of complex numbers c for which the function $f_c(z) = z^2 + c$ does not diverge when iterated from $z = 0$. Mandelbrot set images can be created by sampling the complex numbers and determining, for each sample point c, whether the result of iterating the function f approaches infinity. The real and imaginary parts of c are treated as image coordinates. This allows the images' pixels to be colored according to how rapidly the sequence $z_n^2 + c$ diverges, with the black color used for points where the sequence does not diverge.

7.2.2 Benchmarks with complex dependency graphs

7. **Tile LU Decomposition (LU)**: it factors a dense matrix into the product of a lower triangular *L* and an upper triangular *U* matrix [172]. The dense $n \times n$ matrix A is divided into an $N \times N$ array of $B \times B$ tiles (*n* = *NB*). It has been based on an earlier version written in StarSs [127].
8. **Tile Cholesky Factorization (Cholesky)**: it is used for the numerical solution of linear equations $Ax = b$, where *A* is symmetric and positive definite. The Cholesky factorization of an $n \times n$ real symmetric positive definite matrix *A* has the form: $A = LL^T$, where *L* is an $n \times n$ real lower triangular matrix with positive diagonal elements. Operations on the tiles are performed using LAPACK [175] (V3.6.1) and BLAS [176] routines. Figure 55a depicts the task graph of the algorithm for a matrix of 5×5 tiles.
9. **Tile QR Factorization (QR)**: it offers a numerically stable way of solving underdetermined and overdetermined systems of linear equations (least squares problems). It is also the basis for the QR *algorithm* for solving the eigenvalue problem [26]. The QR factorization of an $m \times n$ real matrix *A* has the form $A = QR$, where *Q* is an $m \times m$ real orthogonal matrix and *R* is an $m \times n$ real upper triangular matrix. The QR version used in this thesis implements the right-looking tile QR factorization as described in [177]. The algorithm uses LAPACK

GEORGE MATHEOU

[175] (V3.6.1) and PLASMA [178] (V2.8.0) routines. Figure 55b shows the task graph of the algorithm for a matrix of 5×5 tiles.



(a) Cholesky factorization

(b) QR factorization

Figure 55: Task graphs of high complexity algorithms.

7.2.3 Recursive algorithms

10. **Fibonacci**: it calculates the Fibonacci numbers using double recursion. In mathematical terms, the sequence F_n of Fibonacci numbers is defined by the recurrence relation: $F_n = F_{n-1} + F_{n-2}$, with seed values $F_0 = 0$ and $F_1 = 1$.

11. **NQueens**: recursive implementation of the NQueens puzzle. It solves the problem of placing N queens on an $N \times N$ chessboard so that no two queens threaten each other [179, 12]. This application is implemented using a Branch and Bound Algorithm. The source code of this application was retrieved from the BSC Application Repository [180]. Figure 56 illustrates a solution to the 4Queens problem (no two queens are on the same row, column, or diagonal).

GEORGE MATHEOU

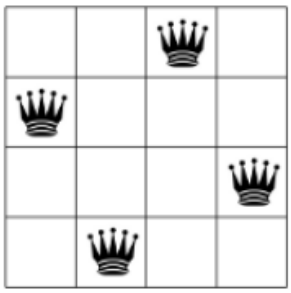


Figure 56: A solution to the 4Queens problem [12].

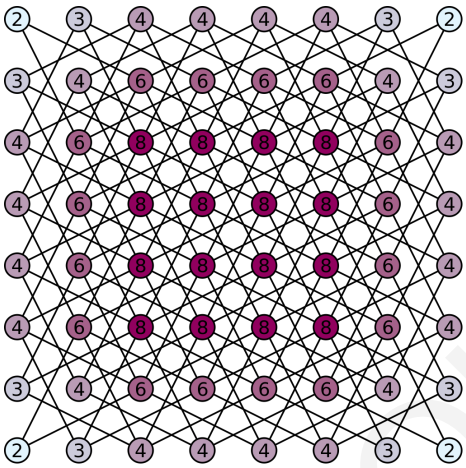


Figure 57: Knight's graph showing all possible paths for a knight's tour on a standard 8×8 chessboard [13].

12. **Knights-Tour**: recursive implementation of the knight's tour problem. A knight's tour is a sequence of moves of a knight on a chessboard such that the knight visits every square only once [13]. If the knight ends on a square that is one knight's move from the beginning square (so that it could tour the board again immediately, following the same path), the tour is closed, otherwise it is open. The source code of this application was retrieved from the BSC Application Repository [180]. Figure 57 shows all possible paths for a knight's tour on a standard 8×8 chessboard. The numbers on each node indicate the number of possible moves that can be made from that position.

13. **PowerSet**: it calculates the number of all subsets of a set with N elements, using a multiple recursion algorithm. The original algorithm was retrieved from the BSC Application Repository [180].

7.3 Experimentation Infrastructure

This section presents details of the systems used to evaluate MiDAS and FREDDO. For the evaluation of the DDM's hardware support the Xilinx ML605 Evaluation Board [62] is used. Figure 58 depicts the board which includes a Virtex-6 FPGA device (XC6VLX240T) and additional peripherals such as: VGA, RS-232, DDR3 Memory, Flash memory, Ethernet, etc. The board possesses 512 MB DDR3 SO-DIMM and 2 GB external Flash memory. The XC6VLX240T Virtex-6 FPGA has the following features:

- *Logic Cells:* 241,152. An LUT can be configured as either one 6-input LUT (64-bit ROMs) with one output, or as two 5-input LUTs (32-bit ROMs) with separate outputs but common addresses or logic inputs. Each LUT output can optionally be registered in a flip-flop.

GEORGE MATHEOU

- *Slices:* 37,680. Each slice contains four LUTs, eight flip-flops, multiplexers and arithmetic carry logic (only some slices can use their LUTs as distributed RAM or SRLs). Two slices form a configurable logic block (CLB).
- *Max Distributed RAM (Kb):* 3,650
- *DSP Slices:* 768. Each DSP slice contains a 25×18 multiplier, an adder, and an accumulator.
- *Block RAM Blocks:* 832 of 18 Kb or 416 of 36 Kb (Max: 14,976 Kb). Block RAMs are fundamentally 36 Kbits in size. Each block can also be used as two independent 18 Kb blocks.
- *Max User I/O:* 720

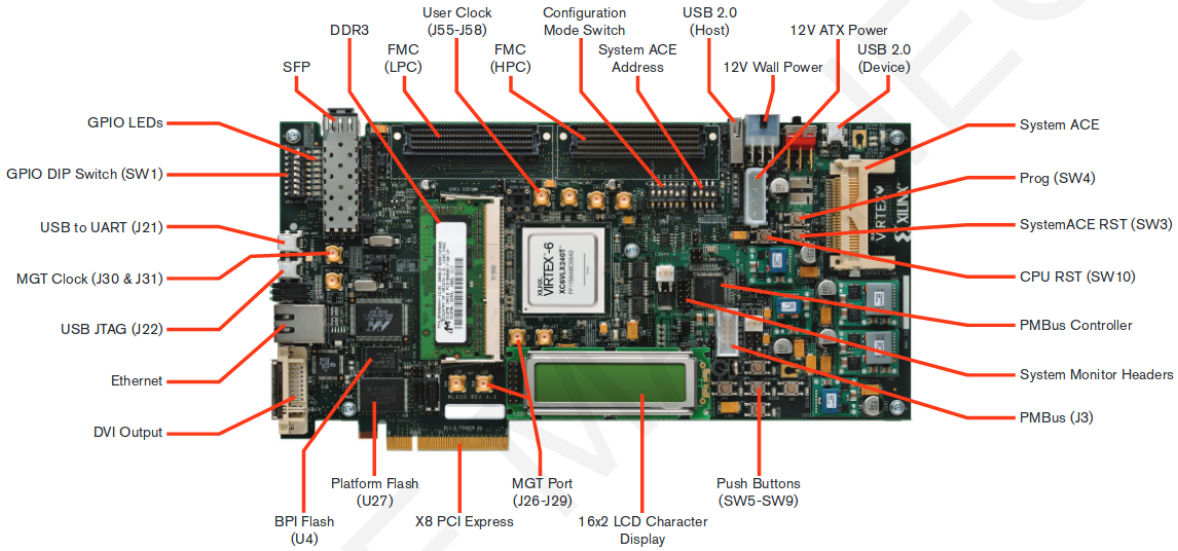


Figure 58: The Xilinx ML605 Evaluation Board.

To evaluate FREDDO we have used two different systems, AMD and CyTera. AMD is a 4-node local system. CyTera [181] is an open-access HPC system which provides up to 64 nodes per user. The specifications of the systems are shown in Table 5. Each AMD node runs Ubuntu 14.04 OS (server edition) where each Intel node runs CentOS 6.6.

System Specs	AMD	CyTera
Processor Type	AMD Opteron 6276	Intel Xeon X5650
Number of Nodes	4	64
Per Node Specs	(Total RAM for both systems: 48 GB)	
- Clock Frequency	1.4 GHz	2.67 GHz
- Cores	16 (2 sockets, 8 cores/socket)	12 (1 socket)
- Hardware Threads/Core	2	1
- Total Hardware Threads	32	12
- L1-I$, L1-D$, L2$/Core	32 KB, 16 KB, 1 MB	32 KB, 32 KB, 256 KB
- L3$	24 MB	12 MB
Interconnect	Gigabit Ethernet	Infiniband QDR (40Gb/s)
Linux Kernel	3.13.0	2.6.32
Compilers	gcc/g++ 4.8.4	gcc/g++ 4.9.3

Table 5: Systems used for the benchmark evaluation of FREDDO.

7.4 The evaluation of the hardware TSU and MiDAS

In this section we determine the performance and estimate the various relevant overheads of the proposed hardware-implemented TSU and of the overall MiDAS system. Firstly, we assess the resource requirements needed to implement the TSU in hardware under different configurations as well as the latencies of various TSU operations. We also evaluate MiDAS's performance using benchmarks with different characteristics in terms of code size, granularity, and inter-thread dependency complexities. MiDAS was implemented with two different TSU configurations: (1) a TSU with a large and fast DynamicSM (it utilizes 32 Context Search Engines (CSEs)) and (2) a TSU with smaller and slower DynamicSM (it utilizes 2 CSEs). The former configuration is called Performance Optimized TSU (*PO-TSU*) and the latter, Area Optimized TSU (*AO-TSU*). Finally, we present FPGA resource utilization results and power consumption estimations of the MiDAS system and compare them with performance-power metrics obtained by simulating other systems.

7.4.1 TSU Resource Requirements

We provide FPGA hardware overhead results needed for the implementation of the hardware TSU synthesized with various parameter values such as TID size, RC size, number of cores supported by the TSU and the number of the Context Search Engines (CSEs) of the DynamicSM. The following TSU parameters are kept constant in all experiments carried out in Section 7.4.1:

- **Context Size:** 32-bits
- **Template Memory:** $2^{TID\ Size}$ entries, $entry\ size\ (bits) = 5 + RC\ Size + \log_2(Number\ of\ cores)$
- **StaticSM:** $2^{TID\ Size}$ entries, $entry\ size\ (bits) = RCSize$
- **DynamicSM:** 1024 SMI entries, 1024 RC Blocks, 32 RCs per RC Block, $SMI\ entry\ size\ (bits) = 22 + \log_2(RCs\ per\ RC\ Block) + TID\ Size + 2 * Context\ Size$
- **CMD_Buffer:** 16 entries, 96-bit entry size, **Waiting Queue:** 256 entries, 96-bit entry size
- **Update Queue:** 256 entries, $entry\ size\ (bits) = 2 + TID\ Size + 2 * Context\ Size$
- **Ready Queue:** 256 entries, $entry\ size\ (bits) = TID\ Size + Context\ Size + 2 + \log_2(Number\ of\ cores)$
- **Number of supported cores:** 8 (except for Section 7.4.1.4)

7.4.1.1 Effect of TID on TSU resource requirements

The Thread ID (TID) attribute is used by the majority of the TSU components. It also determines the sizes of the Template Memory (TM) and the StaticSM. In order to study the effect of the TID

GEORGE MATHEOU

size on the TSU resource requirements we synthesized the TSU with five different TID sizes at 4-bits in increments of 2-bits up to 12-bits. In our experimental evaluation we used an 8-bit RC and a DynamicSM with 8 CSEs. Figure 59 depicts the effect of the TID size on the TSU resource requirements. The results are normalized to the 4-bit configuration. Slice logic requirements include the number of slice registers/flip-flops, i.e., the number of one-bit registers used in the entire FPGA, and the slice look-up tables (LUTs) that are used as logic or distributed RAM, or even as shift registers. Experimental results show that the TID size does not significantly affect overhead requirements. We chose an 8-bit TID as our default configuration since it causes the number of slice LUTs to increase by merely 2%. Also, such configuration allows the TSU to hold up to 256 Thread Templates, an adequate number for the benchmarks used in our tests as well as for larger benchmarks exhibiting greater complexity. We note that our high-complexity benchmarks utilize up to 6 Thread Templates.

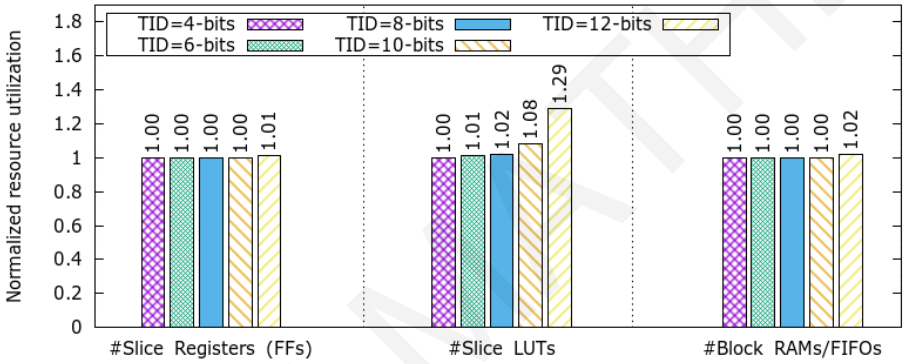


Figure 59: Effect of TID on TSU resource requirements.

7.4.1.2 Effect of RC size on TSU resource requirements

Another important parameter of the proposed TSU is the RC size as it determines the number of producers of each DThread instance. Each DThread instance can have up to $2^{RC\ Size} - 1$ consumers. Figure 60 depicts the effect of the RC size on the TSU resource requirements for five different configurations at 2, 4, 8, 16, and 32 bits, where the demonstrated results are normalized to the 2-bit configuration. In our experiments we used an 8-bit TID and a DynamicSM with 8 CSEs. Results show that the RC size does not significantly affect resource requirements except for the 32-bit configuration where the number of LUTs is increased by 9% and the number of BRAM by 29%. We opt for the 8-bit RC to act as the default configuration since with this configuration the number of slice LUTs required is increased by merely 2%, while it can also handle a reasonable number of consumers for each DThread instance, i.e., 255 consumers.

GEORGE MITHEOU

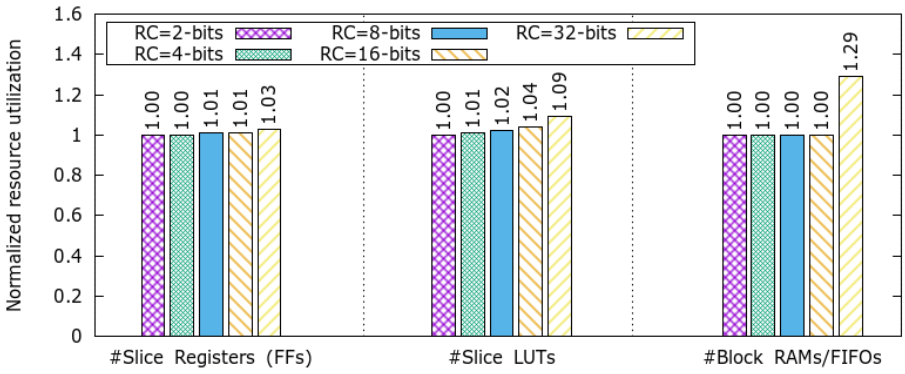


Figure 60: Effect of RC size on TSU resource requirements.

7.4.1.3 Effect of Context Search Engine (CSE) number on TSU resource requirements

The SM Indexer (SMI) of the DynamicSM is divided into several CSEs in order to accelerate search operations. The larger the number of CSEs, the better the performance that can be achieved. In order to evaluate the effect of the CSE number on resource requirements we synthesized the TSU at five different configurations of 2, 4, 8, 16, and 32 CSEs. Figure 61 depicts the effect of the CSE number on the TSU resource requirements, with results normalized to the #CSEs=2 configuration. These results show that the number of CSEs affects significantly the resource requirements. For example, when 32 CSEs are used, the slice register count is increased by 87%, the slice LUT count by 164%, and the Block RAM count by 243%. Choosing the proper number of CSEs depends on two basic parameters: (i) the performance of the TSU demanded by applications, where the larger the number of CSEs the better the attained performance, and (ii) the size of the target FPGA device.

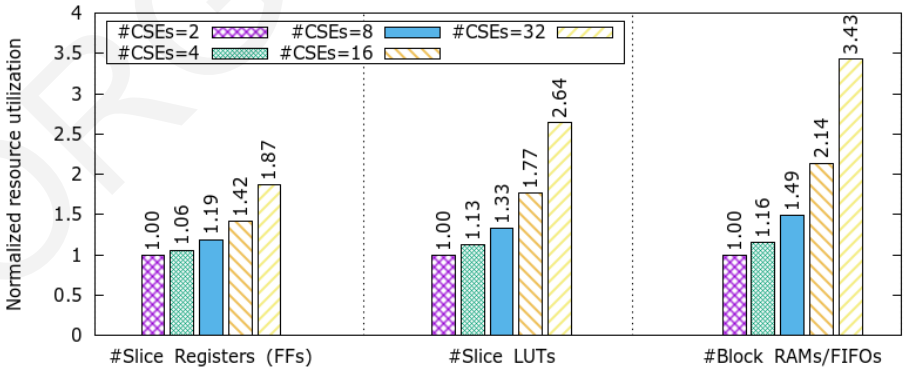


Figure 61: Effect of CSE number on TSU resource requirements.

7.4.1.4 Effect of the number of cores on TSU resource requirements

Figure 62 depicts TSU resource requirements with respect to the number of supported cores. The TSU was configured with 2, 4, 8, 16, 32, and 64 cores, and then synthesized. All experimental results are normalized to the #Cores=2 configuration. We used an 8-bit TID width, an 8-bit RC width, and

GEORGE MATHEOU

a DynamicSM with 4 CSEs. As our results show, resource requirements grow with increasing core numbers, as expected. The reason for this is that the number of CMD_Buffers, Waiting Queues, and Transfer Units is increased. Furthermore, the round-robin arbiters of the Fetch Unit and the Scheduling Unit need more resources since they now support more cores.

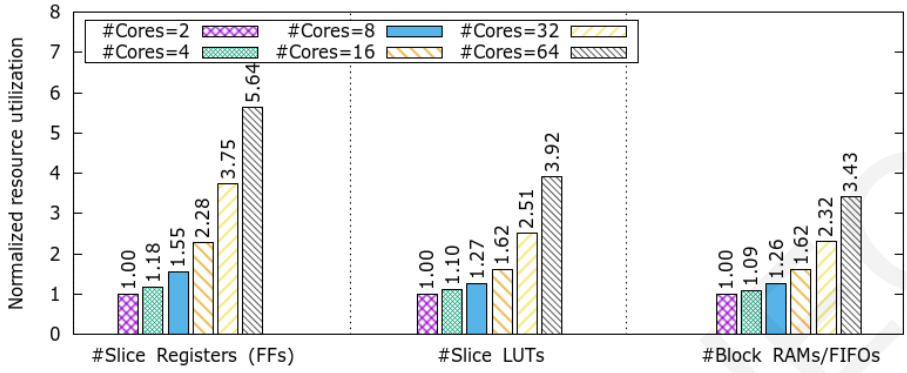


Figure 62: Effect of the number of cores on TSU resource requirements.

TSU Components and Operations	Cycles	Number of Entries	Entry Size (bits)
Input & Ouput FSL		128	32
- Enqueue/Dequeue	1		
Template Memory		256	16
- Read	3		
- Write	4		
- Invalid	4		
StaticSM		256	8
- Update (includes reading the RC and storing the new value, i.e. RC = RC-1)	6		
DynamicSM (1024 SMI entries, 32 CSEs, 32 RCs per block)		**SMI:** 1024 (32 CSEs x 32)	99
		RCM: 32 x 1024	8
- Update a DThread instance which its RC Block does not exist in DynamicSM	103		
- Update a DThread instance which its RC Block exist in DynamicSM	22		
- Invalid all the entries of a specific DThread	168		
CMD_Buffer		16	96
- Enqueue/Dequeue	1		
Update Queue		256	74
- Enqueue/Dequeue	1		
Ready Queue		256	45
- Enqueue/Dequeue	1		
Waiting Queue		256	96
- Enqueue/Dequeue	1		
Fetch Unit			
- Move an Update command from a CMD_Buffer to the Update Queue	4		
- Move a Simple Update from an Input FSL Bus to a CMD_Buffer	4		
- Move a Single Update from an Input FSL Bus to a CMD_Buffer	5		
- Move a Multiple Update from an Input FSL Bus to a CMD_Buffer	6		
Scheduling Unit			
- Store a DThread instance from the Ready Queue to a Waiting Queue using the Static Scheduling Policy	3		
- Store a DThread instance from the Ready Queue to a Waiting Queue using the Round-Robin Scheduling Policy	5		
- Store a DThread instance from the Ready Queue to a Waiting Queue using the Dynamic Scheduling Policy	6		
Transfer Unit			
- Move a ready DThread instance from a Waiting Queue to an Ouput FSL Bus	4		

Table 6: Latencies (in cycles) of various TSU Operations.

7.4.2 Latencies (in cycles) of various TSU Operations

Table 6 presents the latencies of various TSU operations, where the majority of these operations are executed in parallel. For instance, the Scheduling Unit moves data from the Ready Queue to the Waiting Queues, according to the scheduling policy, while the Update Unit sends Update signals to the Synchronization Memories, i.e., the StaticSM and the DynamicSM, while the Transfer Units move data from the Waiting Queues to the Output FSL buses. Additionally, the Fetch Unit moves data from the CMD_Buffers to the Update Queue while it fetches Update Commands from the Input FSL Buses and stores them in the CMD_Buffers. The presented results were obtained using waveform outputs from the Xilinx ISim simulator [182], where the TSU was synthesized with an 8-bit TID width and an RC of 8-bit width. Finally, Table 6 presents the number of entries of each data structure along with the entry sizes in bits. Experimental results show that TSU operations demand reasonable latencies to complete. However, the latencies of the DynamicSM are quite high due to the overheads of the SM Indexer (SMI) module. The performance of SMI can be improved by using more CSEs or by implementing the SMI using Content-Addressable Memories (CAMs) [183]. However, in the latter case more resources and power will be required, as we show in Section 8.2.1.2.

7.4.3 Performance Evaluation of MiDAS architecture

MiDAS was configured with eight MicroBlaze soft-cores. Although our TSU implementation supports a larger number of cores, as shown in Figure 62, Xilinx tools place restrictions upon the utilization of a greater number of cores in the target FPGA device. The reason is that the AXI4 bus used to connect the MicroBlaze soft-cores with the DDR3 SDRAM controller supports up to 16 master links; each MicroBlaze utilizes 2 master links for its Data and Instruction caches, thus only eight MicroBlaze cores can be supported. This issue can be solved in newer Xilinx FPGA devices, such as Virtex-7 and Virtex-UltraScale FPGAs, using the Xilinx Vivado Design Suite [155].

All benchmarks were developed using our C++ API, where Table 7 illustrates the characteristics of the benchmarks used in our experimental evaluation. Pertaining to the benchmarks operating on matrices, these matrices contain dense single-precision floating-point values. The problem sizes are separated into six categories as follows: *Tiny*, *XXSmall*, *XSmall*, *Small*, *Medium*, and *Large*. Pertaining to the block/tile algorithms, we choose two different granularities for each problem size. Table 8 depicts the sequential execution time for each problem size of all benchmarks. The best sequential and parallel execution time among the two selected granularities for each problem size of each block/tile algorithm was selected. The execution time measurements were collected using the hardware timer of the master core of the system.

The implementation details of the multi-core processor were presented in Section 3.3. Each MicroBlaze of the multi-core processor has 32-KB non-coherent L1 Data and Instruction caches. The

GEORGE MATHEOU

Benchmark Acronym	Problem Size					
	Tiny	XXSmall	XSmall	Small	Medium	Large
BMMULT	16x16 Matrix Blocks: 2x2, 4x4	32x32 Matrix Blocks: 4x4, 8x8	64x64 Matrix Blocks: 4x4, 8x8	128x128 Matrix Blocks: 8x8, 16x16	256x256 Matrix Blocks: 16x16, 32x32	512x512 Matrix Blocks: 16x16, 32x32
Trapez	2^8 steps	2^{10} steps	2^{12} steps	2^{14} steps	2^{16} steps	2^{18} steps
Blacksholes	8 options Num Runs: 8	16 options Num Runs: 8	32 options Num Runs: 8	32 options Num Runs: 16	64 options Num Runs: 16	128 options Num Runs: 16
Conv2D	16x16 Matrix Blocks: 2x2, 4x4	32x32 Matrix Blocks: 4x4, 8x8	64x64 Matrix Blocks: 4x4, 8x8	128x128 Matrix Blocks: 8x8, 16x16	256x256 Matrix Blocks: 16x16, 32x32	512x512 Matrix Blocks: 16x16, 32x32
Cholesky	16x16 Matrix Tile: 2x2, 4x4	32x32 Matrix Tile: 4x4, 8x8	64x64 Matrix Tile: 4x4, 8x8	128x128 Matrix Tile: 8x8, 16x16	256x256 Matrix Tile: 16x16, 32x32	512x512 Matrix Tile: 16x16, 32x32
LU	16x16 Matrix Tile: 2x2, 4x4	32x32 Matrix Tile: 4x4, 8x8	64x64 Matrix Tile: 4x4, 8x8	128x128 Matrix Tile: 8x8, 16x16	256x256 Matrix Tile: 16x16, 32x32	512x512 Matrix Tile: 16x16, 32x32

Table 7: Benchmark suite characteristics used in evaluating MiDAS's performance.

Benchmark Acronym	Sequential Time (in seconds)					
	Tiny	XXSmall	XSmall	Small	Medium	Large
BMMULT	0.09	0.64	5.34	40.82	320.26	2565.24
Trapez	0.40	1.61	6.46	25.82	103.17	415.26
Blacksholes	1.79	3.91	7.99	16.28	37.54	72.42
Conv2D	0.20	0.80	3.16	12.63	50.41	202.21
Cholesky	0.03	0.12	0.87	6.18	48.07	379.81
LU	0.03	0.22	1.79	13.68	106.99	858.76

Table 8: Sequential execution time of the benchmarks running on MiDAS.

caches were implemented using the write-through policy since our experiments showed that using caches with the write-back policy incur more overheads. The reason is that each time a DThread instance finishes its execution a flushing operation has to be called (in order to transfer the output data from the cache to the Main Memory before the consumer-DThreads start their execution) which decreases the performance. A better approach is to use software-controlled scratchpad memories [184] where the DThreads' output data will be transferred to the Main Memory using DMA operations. A similar approach was implemented in software, by DDM-VM [10], for the Cell processor.

MiDAS was evaluated under two different TSU implementations, first using the Performance Optimized TSU (PO-TSU), and second, using the Area Optimized TSU (AO-TSU). The PO-TSU implementation has a large DynamicSM with 1024 SMI entries, 1024 RC Blocks, 32 CSEs and 32 RCs per block. This enables the TSU to hold 32768 RC values at the same time, for DThreads with $RC > 1$. The 32 CSEs utilize 64 ports simultaneously in order to search the SMI module. The AO-TSU implementation has a smaller and slower DynamicSM with 1024 SMI entries, 1024 RC Blocks, 2 CSEs and 8 RCs per block. This configuration enables the TSU to hold 8192 RC values for DThreads with $RC > 1$. In both TSU implementations we used an 8-bit TID width and an 8-bit RC width. The number of entries of the TSU components, except for DynamicSM, are depicted in Table 6 (e.g., each Input FSL Bus has 128 entries).

7.4.3.1 Performance Evaluation using the Performance Optimized TSU (PO-TSU)

The performance evaluation of MiDAS using the PO-TSU implementation is illustrated in Figure 63. The results are presented in the form of speedups where speedup is defined as S/P, where S is the execution time of the sequential version of the benchmark (without any DDM overheads), and where P is the execution time of the DDM implementation. The benchmarks were executed using the following numbers of enabled cores: 1, 2, 4, and 8. Although the current MiDAS system utilizes eight cores, our API supports a higher number of cores through an API routine, called *init_processor*, which informs the TSU about the cores that will be used in DDM applications. We also evaluated the ability of the system to handle different problem sizes and thread granularities, i.e., fine-grained and coarse-grained. Our experimental evaluation shows that MiDAS's performance scales very well across the range of tested benchmarks and achieves very good speedups, especially under the larger problem sizes. This is justified by the fact that as the execution time of a benchmark increases, the parallelization overhead is amortized.

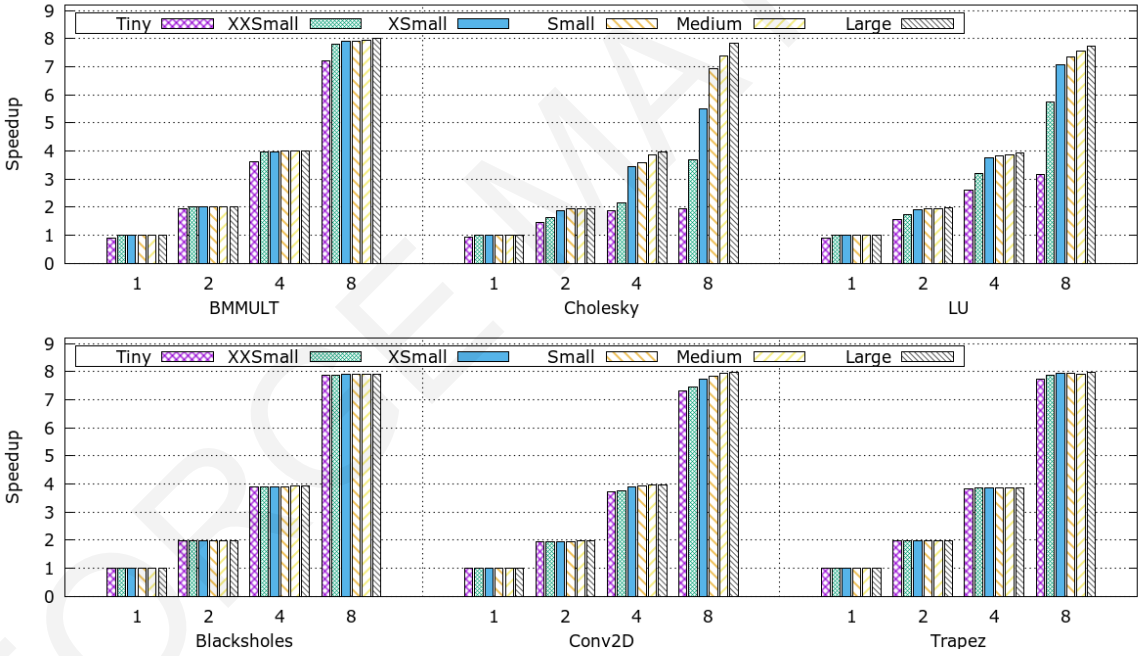


Figure 63: MiDAS's performance using the PO-TSU implementation under various numbers of enabled cores and problem sizes.

The average speedup and efficiency for each problem size and number of enabled cores is depicted in Table 9. For the smallest problem size (*Tiny*), the system achieves an average speedup from 0.95 to 5.88 while for the *Small* problem size it achieves an average speedup from 1 to 7.64. For the 8-core configuration and the largest problem size (*Large*) MiDAS achieves an average speedup of 7.91, exhibiting 98.8% efficiency with respect to the theoretical maximum speedup of 8. These results demonstrate that MiDAS is a performance-scalable architecture, achieving speedups close to linear.

GEORGE MATHEOU

MiDAS achieves such results since it provides efficient data-driven scheduling through an optimized TSU implementation with low overheads. There are three factors that contribute to this. First, MiDAS utilizes light-weight and very fast buses to transfer Update commands and ready DThread instances between the TSU and the processing elements. Second, the TSU was implemented as an optimized asynchronous hardware module where its data structures (such as TM, StaticSM, etc.) are implemented using an on-chip BRAM memory. On-chip memory is a much faster memory than the system's main memory (i.e., DDR3 SDRAM) which is used for the data structures of software TSU implementations [35, 10, 142]. Third, searching operations on the TSU's DynamicSM are accelerated using CSEs and dual-port RAMs, while writing operations are accelerated using multiple Single Port RAMs, used for storing RC Blocks in the DynamicSM's RCM.

Problem Size	Average Speedup				Average Efficiency			
	1	2	4	8	1	2	4	8
Tiny	0.95	1.81	3.25	5.88	95.2%	90.3%	81.3%	73.4%
XXSmall	0.99	1.87	3.47	6.73	99.1%	93.7%	86.8%	84.2%
XSmall	0.99	1.95	3.80	7.34	99.2%	97.5%	95.0%	91.7%
Small	1.00	1.96	3.84	7.64	99.9%	98.0%	96.1%	95.5%
Medium	1.00	1.97	3.92	7.78	99.9%	98.4%	97.9%	97.2%
Large	1.00	1.97	3.94	7.91	99.9%	98.7%	98.6%	98.8%

Table 9: Average speedup and efficiency for each problem size and number of enabled cores.

7.4.3.2 Performance Evaluation using the Area Optimized TSU (AO-TSU)

To evaluate MiDAS's performance when utilizing the AO-TSU implementation we executed the Cholesky and LU benchmarks using all available cores. The remaining benchmarks were excluded from our experiments since they do not use the DynamicSM module; MiDAS is expected to achieve the same performance levels either when utilizing the AO-TSU implementation or the PO-TSU implementation when executing such benchmarks. This is due to the fact that the main difference between the PO-TSU and the AO-TSU is the configuration of the DynamicSM, where the PO-TSU has a larger and faster DynamicSM compared to the AO-TSU. The characteristics of the DThreads of each benchmark are depicted in Table 10. It is evident that only Cholesky and LU utilize the DynamicSM since they have DThreads with $RC > 1$. The rest of the benchmarks have DThreads with $Nesting = 0$ (the StaticSM is used) and DThreads with $RC = 1$ (the SM is not used).

Figure 64 depicts MiDAS's attained performance when using the PO-TSU compared to that when MiDAS utilizes the AO-TSU. Results show that speedups are very comparable under both configurations. In particular, the PO-TSU shows a 0.24% performance increase, on average, for the Cholesky benchmark, and an equivalent 0.22% for the LU benchmark. The additional delays incurred in AO-TSU's DynamicSM are hidden from the user level since the TSU and the MicroBlaze cores operate

GEORGE MANIATIS

Benchmark Acronym	DThreads	Nesting	RC Value	Consumers	Benchmark Acronym	DThreads	Nesting	RC Value	Consumers
Blacksholes	**T1**	1	1	T1, T2	**BMMULT**	**T1**	3	1	T1, T2
	T2	0	NUM CORES	None		**T2**	0	1	None
Conv2D	**T1**	1	1	T2	**Trapez**	**T1**	1	1	T2
	T2	2	1	T1, T3		**T2**	0	NUM CORES	None
	T3	0	NUM BLOCKS	None	**LU**	**T1**	1	1	T2-T5
Cholesky	**T1**	1	1	T3, T5		**T2**	1	2	T3, T4, T6
	T2	2	2	T1, T2		**T3**	2	3	T5
	T3	2	2	T2, T4		**T4**	2	3	T5
	T4	3	3	T3, T4		**T5**	3	4	T2-T5
	T5	0	1	None		**T6**	0	1	None

Table 10: Characteristics of DThreads for each benchmark running on MiDAS.

asynchronously. One advantage of using the AO-TSU is that it demands fewer overheads and requires less power to operate compared to the PO-TSU (see Section 7.4.4). However, the AO-TSU holds fewer RC values, 8192 in specific, a fact which restricts some benchmarks in being executed with fine-grained threads, such as in the case of the LU benchmark with 512×512 matrices and 16×16 tiles.

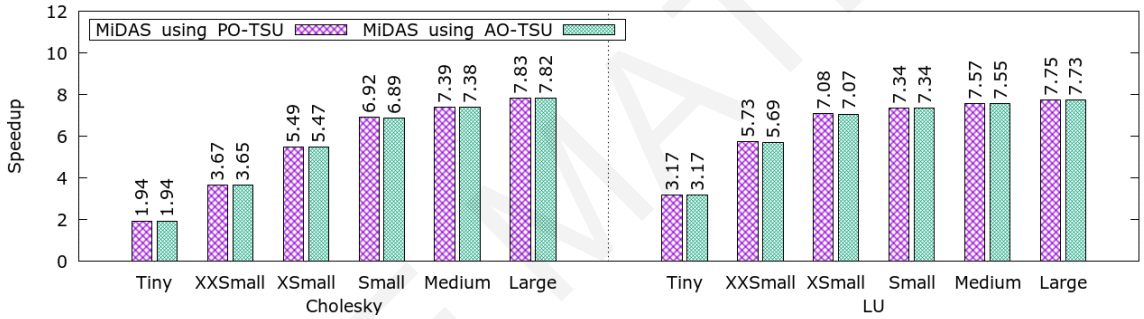


Figure 64: Performances of MiDAS using the PO-TSU vs. MiDAS using the AO-TSU under the Cholesky and LU benchmarks.

Figure 65 compares the latencies of the basic operations carried out by the DynamicSM for each of the two TSU implementations. The results are normalized with respect to the latencies of the PO-TSU's DynamicSM, and are obtained using waveform outputs from the Xilinx ISim simulator [182]. It is clear that the DynamicSM of the PO-TSU is much faster compared to the one of the AO-TSU. The reason for this, is that, in the case of the AO-TSU the CSEs manage more SMI entries, that is, 512 SMI entries for each CSE, where in the PO-TSU each CSE manages 32 SMI entries. This affects the Update operations that target RC Blocks which do not exist in the DynamicSM (each CSE checks all its SMI entries to identify whether the RC Block does not exist). In a similar manner, invalidate operations are also affected since each SMI entry should be checked whether it holds the TID of a specific DThread. However, the Update operations that target RC Blocks which exist in the DynamicSM require almost the same latency to complete. The reason for this is that hashing operations are performed based on the TID and Context attributes, which locate the RC Blocks faster.

However, as demonstrated above, the latencies of the DynamicSM do not affect the performance of the system when real benchmarks are executed, such as those of Cholesky and LU.

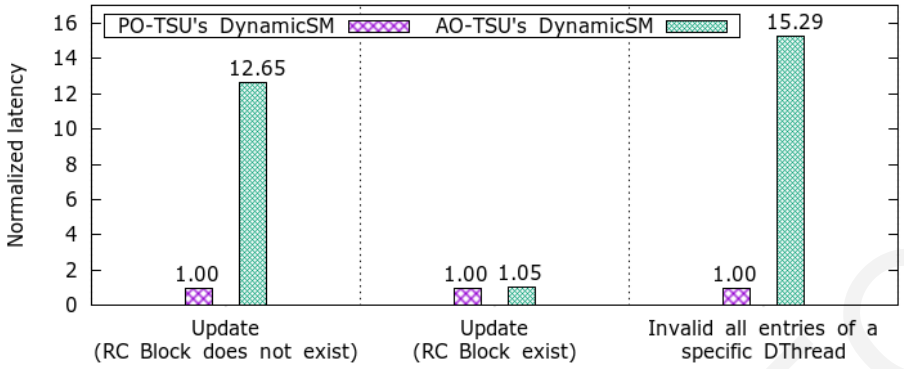


Figure 65: Comparing the DynamicSM's latencies of PO-TSU and AO-TSU.

7.4.4 FPGA Resource Requirements and Power Consumption Estimations for MiDAS

The FPGA resource requirements and power consumption estimations in constructing MiDAS using the PO-TSU and the AO-TSU implementations are depicted in Table 11. These overhead results were generated using the Xilinx XPS tool where power estimations were estimated using the Xilinx XPower Analyzer tool. We would like to note that more accurate resource requirement results can be taken using the Xilinx ISE tool which utilizes post-synthesis information, however, the tool does not provide per-component results. The XPower Analyzer tool uses post-synthesis results and for our experiments the tool was used with the default configurations. The FPGA resource overheads for each individual component are shown in parentheses in terms of percentages. In the same table the component labelled "other" includes the clock generator, the MDM, the Memory Controller, etc., all of which are necessary to ensure the proper system functionality.

Comp.	MiDAS using PO-TSU					MiDAS using AO-TSU				
	Slice Registers	Slice LUTs	BRAM	DSPs	Power (W)	Slice Registers	Slice LUTs	BRAM	DSPs	Power (W)
Core x 8	31784 (10.5%)	36528 (24.2%)	152 (36.5%)	40 (5.2%)	0.32	31784 (10.5%)	36528 (24.2%)	152 (36.5%)	40 (5.2%)	0.32
FSL x 16	368 (0.1%)	2576 (1.7%)	0 (0.0%)	0 (0.0%)	0.03	368 (0.1%)	2576 (1.7%)	0 (0.0%)	0 (0.0%)	0.03
TSU	7478 (2.5%)	16510 (11.0%)	114 (27.4%)	0 (0.0%)	0.07	3124 (1.0%)	4981 (3.3%)	12 (2.9%)	0 (0.0%)	0.04
Other	8005 (2.7%)	10935 (7.3%)	2 (0.5%)	0 (0.0%)	0.17	8005 (2.7%)	10935 (7.3%)	2 (0.5%)	0 (0.0%)	0.16
Total	**47635 (15.8%)**	**66549 (44.2%)**	**268 (64.4%)**	**40 (5.2%)**	**0.59**	**43281 (14.4%)**	**55020 (36.5%)**	**166 (39.9%)**	**40 (5.2%)**	**0.55**

Table 11: Virtex-6 FPGA resource requirements and power consumption estimations in implementing MiDAS incorporating either the PO-TSU or the AO-TSU.

The hardware device requirements for MiDAS using the PO-TSU are rather low, taking up 15.8% of the slice registers and 44.2% of the LUTs of the target FPGA device. As such, this facilitates feasible future extensions in the functionality of our system. The Block RAM (BRAM) requirements are set at 64.4%. The BRAM is mainly used to (i) model big caches (32KB for each cache) in order to improve the performance of the system and (ii) to implement a large DynamicSM. Obtained results

GEORGE MATHEOU

show that the hardware PO-TSU can easily fit on the Virtex-6 FPGA since it utilizes about 2.5% of its slice registers, 11.0% of its slice LUTs, and 27.4% of its BRAM. Further, it utilizes a small proportion (11.9%) of the overall power of the multi-core system. The AO-TSU utilizes a small proportion of the Virtex-6 FPGA resources; 1.0% of the slice registers, 3.3% of the slice LUTs, and 2.9% of the BRAM. This enables the AO-TSU to be used in much smaller FPGA devices as compared to the Virtex-6 FPGA.

Figures 66 and 67 present the percentage of the resources and power consumption that each component of MiDAS consumes, when PO-TSU and AO-TSU are used. PO-TSU consumes 15% of the slice registers, 25% of the slice LUTs, 42% of the BRAM and 12% of the power, of the entire system. Moreover, AO-TSU consumes 7% of the slice registers, 9% of the slice LUTs, 7% of the BRAM and 7% of the power, of the entire system.

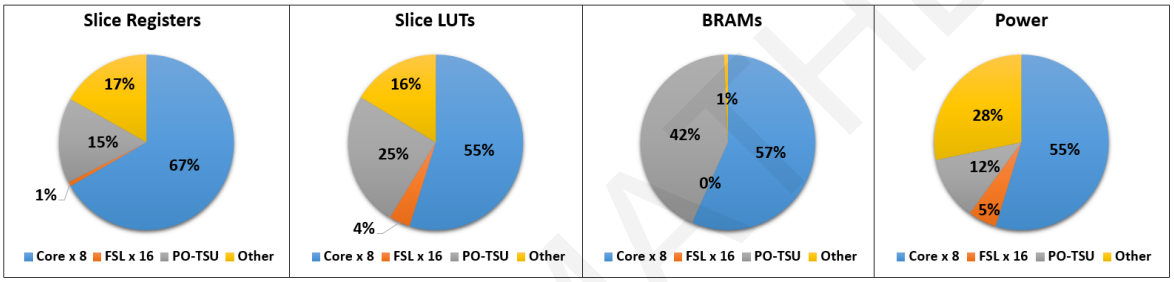


Figure 66: Per-component resource utilization and power consumption of MiDAS using PO-TSU.

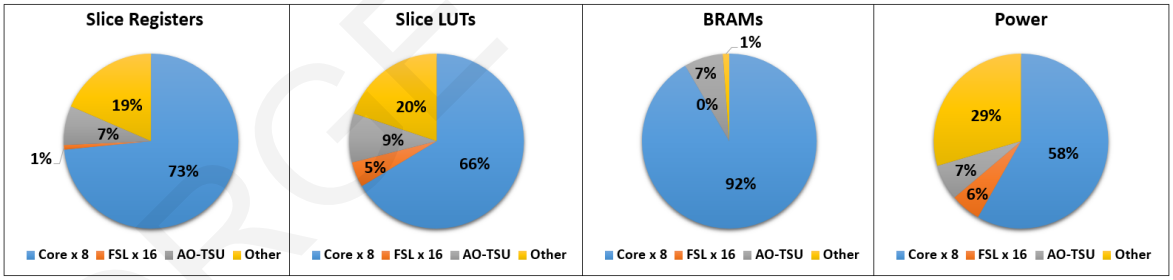
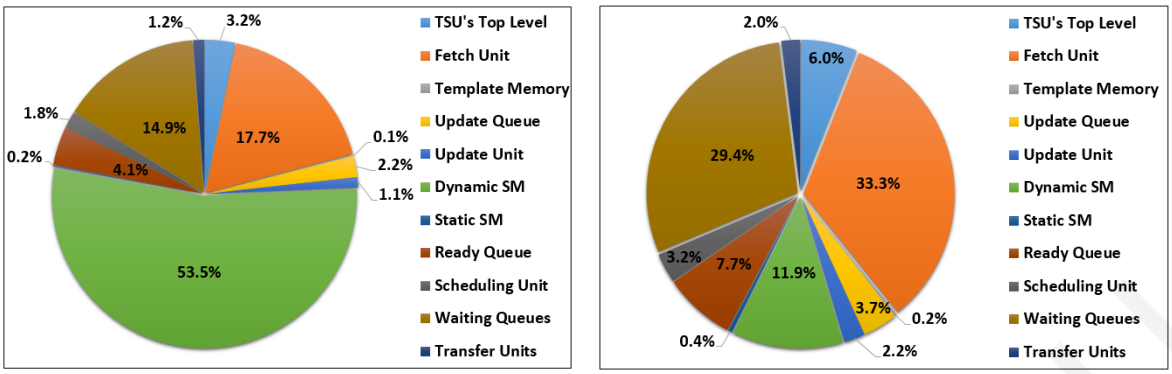


Figure 67: Per-component resource utilization and power consumption of MiDAS using AO-TSU.

Figure 68 shows the power consumption of each component in the PO-TSU and the AO-TSU. The largest power consumer of the PO-TSU is the DynamicSM, set at about 54%. In the case of the AO-TSU which uses a smaller DynamicSM implementation, the largest power consumers are the Fetch Unit (33.3%) and the Waiting Queues (29.4%). Finally, the DynamicSM of the AO-TSU takes up only 11.9% of the AO-TSU's overall power consumption.

GEORGE MARKEOU



(a) PO-TSU

(b) AO-TSU

Figure 68: Per-component power consumption of the PO-TSU and AO-TSU.

7.4.5 DDM Architectural Support in MiDAS vs. Task Superscalar Architecture

We compare our hardware TSU implementation with that of the Task Superscalar implementation [4, 46]. Unfortunately, the HDL code of Task Superscalar is inaccessible, and, as such, we cannot compare the performance (e.g., latencies of executing threads/tasks) of our TSU implementation to those of Task Superscalar. Hence, we used the synthesis results of Task Superscalar which are provided in [46] to compare both implementations with regards to the resource requirements and macro statistics. Task Superscalar was synthesized for two different Xilinx Virtex-7 devices, those of xc7vh290t and xc7v2000t, using the Xilinx ISE 14.2 tool. In this work we make use of the Xilinx ISE 14.7 tool which only supports the xc7v2000t device. Thus, our comparisons will only be provided for the xc7v2000t device. Task Superscalar utilizes Distributed RAM in implementing its data structures, such as, Task Reservation Stations (TRSs), Object Renaming Tables (ORTs), and Object Versioning Tables (OVTs).

It is important to note that Task Superscalar was not interconnected and evaluated with real processing elements (e.g., MicroBlaze soft-cores) and real benchmarks. Thus, Task Superscalar does not include modules like CMD_Buffers, Waiting Queues and Transfer Units which are used for the communication with the cores. The authors of Task Superscalar used a trace memory to test and simulate the functionality of of their system. The trace memory can save up to 1024 tasks (each task includes 17 80-bits entries). The Task Superscalar prototype incorporates two TRSs which can store up to 1024 in-flight tasks (similar to our DThread instances). The Task Superscalar's components are connected using an interconnection network which includes arbiters and 4-element FIFOs.

In order to obtain fair comparisons with the Task Superscalar implementation we synthesized the hardware TSU to store at least 1024 DThread instances with $RC > 1$, using a DynamicSM with 128 SMI entries, 128 RC Blocks, 8 RCs per RC Block, and 2 CSEs. The Template Memory and StaticSM are configured to handle 256 entries (i.e., TID Size=8-bits), thus facilitating 256 DThreads with Nesting=0 and $RC \neq 1$. Also, the TSU implementation allows an unlimited amount

of DThread instances with $RC = 1$ and supports 8 cores and 8-bit RC values. Finally, all FIFOs-/Buffers and data structures (i.e., Template Memory, StaticSM, and DynamicSM) of the proposed TSU were implemented using Distributed RAM instead of Block RAM. This is achieved using the *(* RAM_STYLE="distributed" *)* directive for each memory allocation. For inferring Block RAM in the default TSU configuration we have used the following directive: *(* RAM_STYLE="block" *)*.

Table 12 compares the resource requirements consumed in a Virtex-7 xc7v2000t device, along with macro statistics, of the proposed hardware DDM-supporting TSU with those of Task Superscalar. The synthesis results of the hardware TSU were taken using the Xilinx ISE tool configured with the default configurations. The results show that Task Superscalar consumes much more resources compared to TSU. In particular, Task Superscalar is $4.94\times$ larger than the DDM's TSU with respect to the slice registers and $11.3\times$ larger with respect to the slice LUTs. Furthermore, Task Superscalar utilizes more registers, multiplexers, adders/subtractors, tristate buffers and xor gates. The presented results clearly show that implementing a data-flow/data-driven system with dynamic dependency resolution in hardware, like Task Superscalar, demands many more resources and consumes more power compared to a system which supports static dependency resolution like our proposed TSU in MiDAS.

System	Slice Utilization		Macro Statistics					
	Slice Registers	Slice LUTs	Registers	Multiplexers	Adders/Subtractors	Comparators	Tristate Buffers	Xors
Hardware TSU	5900	9766	275	1311	50	50	16	1
Task Superscalar	29138	110729	13094	3407	68	23	268	8924

Table 12: Resource requirements and macro statistics of the proposed TSU vs. Task Superscalar.

7.4.6 Hardware vs. Software TSU - Preliminary Results

In this work we have implemented DDM's TSU in both software and hardware. The software TSU is used by FREDDO to schedule DThread instances on conventional/commodity multi-cores. The hardware TSU is used by MiDAS to schedule DThread instances on MicroBlaze soft-cores. It is important to know how much faster is the hardware TSU compared to a software implementation with the same functionalities. To answer this question we have developed FREDDO's TSU as a C++ library targeting the MicroBlaze soft-core. For the development process we have used the Xilinx SDK tool and the MicroBlaze GCC compiler (mb-g++). In order to have fair comparisons between the two TSU implementations, the software TSU was configured to support 32-bit Context values. Also we have disabled the recursion support, the Unlimited Input Queues (UIQs), the Graph Memory (used to store the consumers of each DThread) and the Pending Template Memory (used to compute the RC values automatically). In the original FREDDO's TSU implementation, the DynamicSM is implemented using a hash-map data-structure (the default implementation uses the C++11 *unordered_map*). For our comparisons we have implemented a software DynamicSM which is similar to the hardware

GEORGE MATHEOU

DynamicSM implementation. In both DynamicSM implementations the RC values are allocated and deallocated dynamically in the form of blocks (called RC Blocks).

	Simple DThread		MultipleDThread			MultipleDThread2D			MultipleDThread3D			Total		
	RC Value	#Single Updates	RC Value	#Single Updates	#Multiple Updates	RC Value	#Single Updates	#Multiple Updates	RC Value	#Single Updates	#Multiple Updates	#Updates Send to TSU	#Updates executed by TSU	#Ready Instances
Synth-NoSM-Small	1	1	1	1	1 of 1024 Instances	1	1	1 of 1024 Instances	1	1	1 of 1024 Instances	4	3076	3076
Synth-NoSM-Medium	1	1	1	1	1 of 2048 Instances	1	1	1 of 2048 Instances	1	1	1 of 2048 Instances	4	6148	6148
Synth-NoSM-Large	1	1	1	1	1 of 4096 Instances	1	1	1 of 4096 Instances	1	1	1 of 4096 Instances	4	12292	12292
Synth-SM-Small	2	2	2	2	2 of 1024 Instances	2	2	2 of 1024 Instances	2	2	2 of 1024 Instances	8	6152	3076
Synth-SM-Medium	2	2	2	2	2 of 2048 Instances	2	2	2 of 2048 Instances	2	2	2 of 2048 Instances	8	12296	6148
Synth-SM-Large	2	2	2	2	2 of 4096 Instances	2	2	2 of 4096 Instances	2	2	2 of 4096 Instances	8	24584	12292

Table 13: Versions of *Synth* benchmark used for comparing software and hardware TSUs.

In our experiments we developed a synthetic application, called *Synth*, which executes the basic TSU operations, such as load/remove DThreads, Update RC values, and read ready DThread instances. *Synth* creates four different types of DThreads comprising SimpleDThread, MultipleDThread, MultipleDThread2D, and MultipleDThread3D, and performs Update operations on all of them. We used six different versions of *Synth* where each version has different characteristics, i.e., diverse numbers of Single and Multiple Updates, and various RC values. Table 13 depicts the characteristics of all *Synth*'s versions. Three of them do not use an SM implementation, those with their RC value of all DThreads set to 1. The *Synth* benchmark was executed on MiDAS's master core using both the software TSU as well as the hardware-constructed TSU. The benchmark performs the following tasks:

- **Task 1:** Four different DThreads are created, one of each type.
- **Task 2:** Single Updates are sent to the TSU through the Input Queue 0 (or Input FSL Bus 0).
- **Task 3:** Multiple Updates are sent to the TSU through the Input Queue 0 (or Input FSL Bus 0).
- **Task 4:** The TSU executes all Update commands. Also, the TSU schedules the ready DThread instances for execution on the master core (i.e., it sends the ready DThread instances to the Output Queue 0 or Output FSL Bus 0).
- **Task 5:** A software function is called which dequeues all ready DThread instances from the Output Queue 0 (or Output FSL Bus 0).
- **Task 6:** The DThreads are removed from the TSU.

For our preliminary evaluation we measured the execution time (in milliseconds) needed to perform the Tasks 2 to 5. Figure 69 compares the software implemented TSU to the hardware-constructed TSU under all *Synth* versions that do not use an SM implementation, while Figure 70 does the same for all *Synth* versions that use an SM implementation. We note that in our experiments we used the PO-TSU implementation.

GEORGE MATHEOU

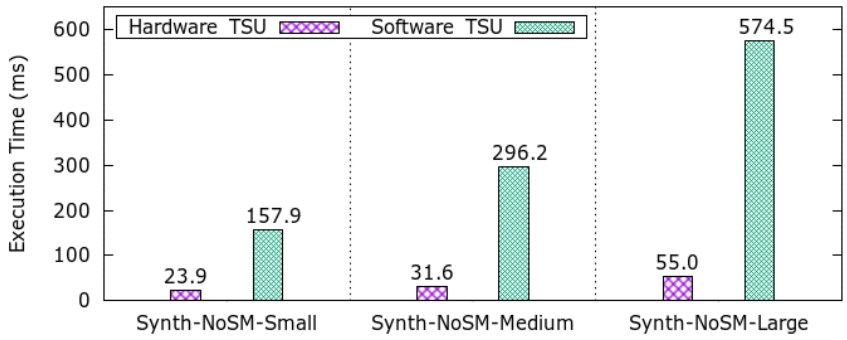


Figure 69: Hardware vs. Software TSU on *Synth*'s versions that do not use an SM implementation.

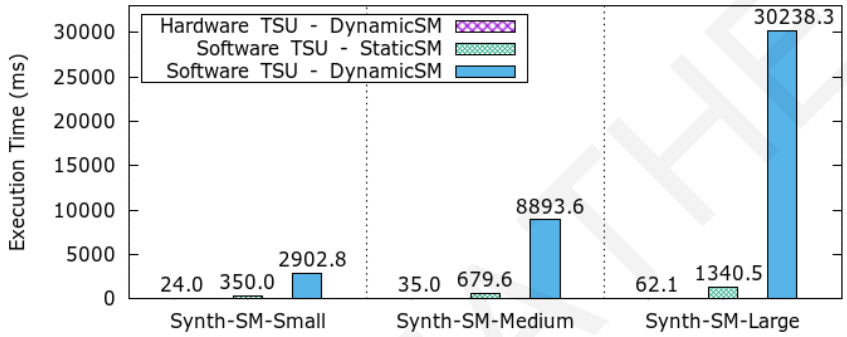


Figure 70: Hardware vs. Software TSU on *Synth*'s versions that use an SM implementation.

The software-implemented TSU was configured with two different SM implementations, DynamicSM and StaticSM. As mentioned earlier, the DynamicSM provides the same functionalities as those of the DynamicSM in the hardware-constructed TSU. The software DynamicSM is configured with 1024 SMI entries, 1024 RC Blocks and 32 RCs per block. The software StaticSM allocates the RC value of all DThread instances immediately at the time when creating the Thread Templates. Accessing an RC entry in the StaticSM is a direct operation that uses the Context of the DThread's instance. Thus, the StaticSM executes Update operations faster than the DynamicSM does. It is important to note that the StaticSM of the software-implemented TSU is different than that of the StaticSM implemented in the hardware-constructed TSU, where in the former case the StaticSM is used to allocate the RC values of DThreads with $RC > 1$, while in the latter case the StaticSM is used to allocate the RC values of DThreads with $Nesting = 0$ and $RC > 1$.

The comparison results show that the hardware-constructed TSU significantly outperforms the software-implemented TSU under all test scenarios. In particular, the hardware-constructed TSU is up to $10.4\times$ faster than the software-implemented TSU when the SM is not utilized. When an SM is needed, the hardware-constructed TSU is up to $21.6\times$ faster than the software-implemented TSU using StaticSM, and up to $487.1\times$ faster when using DynamicSM. The software-implemented TSU that uses the DynamicSM achieves the lowest performance obtained due to the runtime overheads

GEORGE MATHEOU

incurring from the dynamic allocation/deallocation of the RC blocks. The hardware-constructed TSU achieves the best performance for the following reasons:

1. All the software-implemented TSU data structures are allocated in RAM which is much slower than the Block RAM which is used to implement the data-structures of the hardware-constructed TSU.
2. The majority of the hardware-constructed TSU operations is executed in parallel. On the contrary, all the operations of the software-implemented TSU are executed sequentially.
3. The hardware-constructed TSU communicates with the master core through a hardware FIFO (FSL Bus) which is faster than a software FIFO implementation.
4. In the hardware-constructed TSU the DynamicSM is implemented using 32 CSEs which use 64 ports simultaneously to search the SMI module. In the software-implemented DynamicSM, however, the SMI entries are accessed sequentially.
5. In the hardware-constructed DynamicSM the entries of an RC Block are initialized simultaneously, whereas in the software-implemented DynamicSM the entries are initialized sequentially.

To conclude, a hardware-constructed TSU can achieve much faster data-driven scheduling compared to a software TSU implementation given the same set of functionalities. This enables efficient benchmark execution with fine-grained threads. Future HPC systems (e.g. exascale architectures) are projected to have up to billions of fine-grained threads executing asynchronously [33, 168]. Scheduling efficiently such numbers of fine-grained threads is an important issue. We believe that our hardware TSU implementation can be used as an efficient thread scheduler in future multi-core/many-core architectures that will constitute the basic building blocks of future massively parallel HPC systems.

7.5 Single-node FREDDO Evaluation

7.5.1 Experimental Setup

For the single-node evaluation of FREDDO we have used an AMD node (see Table 5) and twelve benchmarks from our benchmark suite (see Section 7.2). The size of the Context values was set to 64-bit. Table 14 illustrates the characteristics of the benchmarks used in our experiments. For the benchmarks working on matrices, the matrices are dense single-precision floating-point. The problem sizes are separated into three categories: *Small*, *Medium* and *Large*. For the block/tile algorithms, we choose three different granularities (32×32, 64×64 and 128×128 block/tile). The last three columns of Table 14 depict the average sequential execution time (in seconds) of five executions for each problem size of all benchmarks. For each problem size of each block/tile algorithm, the average sequential time is the best among all granularities. The execution time measurements were collected

GEORGE MATHEOU

using the *gettimeofday* system call. For the recursive algorithms we have used *thresholds* in order to control the number of DThread instances that are used for executing the recursive calls.

No	Benchmark Acronym	Granularity	Problem Size			Average Sequential Time (sec)		
			Small	Medium	Large	T_{Small}	T_{Medium}	T_{Large}
1	BMMULT	32x32 Block 64x64 Block 128x128 Block	1024x1024 Matrix	2048x2048 Matrix	4096x4096 Matrix	3.67	29.63	235.09
2	Conv2D	32x32 Block 64x64 Block 128x128 Block	1024x1024 Matrix	4096x4096 Matrix	8192x8192 Matrix	1.26	20.15	81.10
3	LU	32x32 Tile 64x64 Tile 128x128 Tile	1024x1024 Matrix	2048x2048 Matrix	4096x4096 Matrix	1.23	9.90	79.36
4	Cholesky	32x32 Tile 64x64 Tile 128x128 Tile	1024x1024 Matrix	4096x4096 Matrix	8192x8192 Matrix	0.40	21.75	170.10
5	QR	32x32 Tile 64x64 Tile 128x128 Tile	1024x1024 Matrix	2048x2048 Matrix	4096x4096 Matrix	8.40	69.75	597.96
6	Swaptions	20000 Simulations	256 Swaptions	512 Swaptions	4096 Swaptions	24.56	50.53	390.88
7	Blackscholes	Variable	16K Options	64K Options	10M Options	2.20	8.80	1319.30
8	Fibonacci	Variable	N=40	N=45	N=48	1.12	12.59	53.09
9	NQueens	Variable	N=13	N=14	N=15	1.49	9.63	65.30
10	Knights-Tour	Variable	4x6 Board	5x5 Board	5x6 Board	4.81	11.21	1164.70
11	PowerSet	Variable	N=30	N=32	N=34	5.83	18.61	88.54
12	Mandelbrot	Variable	8192x8192 Image	16384x16384 Image	32768x32768 Image	12.66	55.34	226.22

Table 14: The benchmark suite characteristics for the FREDDO evaluation.

The FREDDO framework and all benchmarks were coded in C++11 and compiled using the g++ 4.8.4 compiler. The LAPACK kernels for the Cholesky and QR benchmarks were compiled using the gcc 4.8.4 compiler. All source codes and libraries/packages were compiled using the *-O3* optimization flag. For the performance evaluation, all experimental results are reported as speedups. The speedup of a certain configuration is defined by the following formula:

$$speedup = \frac{average\ sequential\ execution\ time}{average\ parallel\ execution\ time},$$

where the *average parallel execution time* indicates the average of five parallel executions using the FREDDO framework. Notice that the maximum possible speedup is 31, since we reserve one core out of the 32 cores for the execution of the TSU.

7.5.2 Performance Evaluation

We have performed a scalability study in order to evaluate the performance of the single-node FREDDO implementation. Our benchmark suite includes applications that are embarrassingly parallel (Swaptions, Blackscholes and Mandelbrot), applications that have a combination of memory-bound and compute-bound nature (BMMULT and Conv2D), and applications with complex data-dependencies (LU, Cholesky and QR). Additionally, four benchmarks with recursion were implemented (Fibonacci, NQueens, Knights-Tour and PowerSet) in order to evaluate the ability of

GEORGE MATHEOU

FREDDO to provide recursion support. Fibonacci, NQueens and PowerSet were implemented using the RecursiveDThread and ContinuationDThread classes, where Knights-Tour was implemented using the RecursiveDThreadWithContinuation class.

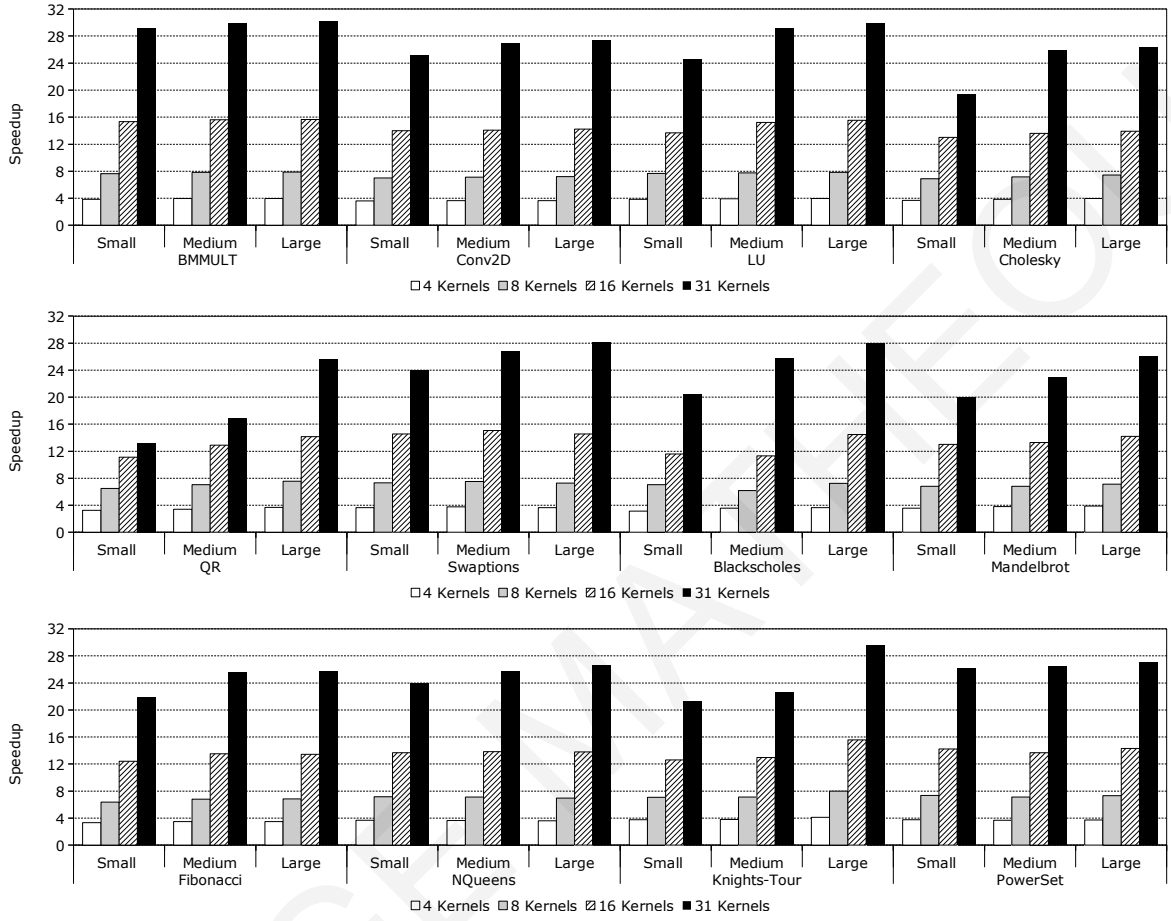


Figure 71: Performance scalability of FREDDO for different number of computation cores (Kernels) and problem sizes.

We have evaluated the performance of FREDDO on different number of cores and problem sizes. The evaluation is shown in Figure 71. Four different Kernel configurations are used: 4, 8, 16 and 31. The evaluation shows that FREDDO scales very well across the range of the benchmarks and it achieves very good speedups, especially for the *Large* problem size. This is justified by the fact that, as the benchmark's execution time increases, the parallelization overhead is amortized. For instance, the LU benchmark, achieves the following speedups: 3.94 out of 4, 7.83 out of 8, 15.53 out of 16 and 29.83 out of 31, for the *Large* problem size. To conclude, the results of executing all the benchmarks demonstrate that overall, the system scales well over the range of the benchmarks and achieves - when utilizing all the cores/Kernels - an average speedup of:

- 22.42 out of 32 for the *Small* problem size (70% efficiency).
- 25.37 out of 32 for the *Medium* problem size (79% efficiency).

GEORGE MASON

- 27.51 out of 32 for the *Large* problem size (86% efficiency).

As such, our framework effectively leverages the decoupling of synchronization and execution for the maximum tolerance of synchronization overheads.

7.5.3 Comparisons

FREDDO is compared with OmpSs [157, 40] using four benchmarks: Cholesky, LU, NQueens and PowerSet. For our experiments we have used the Nanos++ runtime V0.13a-2017-06-02 and the Mercurium compiler V2.0.0-2017-06-02. The comparison results are shown in Figure 72. All benchmarks were executed on an AMD node utilizing all the available cores. For the NQueens benchmark, both frameworks achieve very good speedups and have similar results. In the case of PowerSet, FREDDO achieves an average improvement of 29.7% for the *Small* problem size, 7.6% for the *Medium* problem size and 7.1% for the *Large* problem size. In the case of benchmarks with complex dependency graphs, FREDDO outperforms OmpSs for all problem sizes. Particularly, for the largest problem size, FREDDO achieves an average improvement of 23.8% for Cholesky and 25.6% for LU. FREDDO outperforms OmpSs, especially in the case of benchmarks with complex dependency graphs, since the latter builds the dependency graph at runtime. This can incur more delay to the critical path of the application compared to our model which creates the dependency graph statically. Moreover, OmpSs makes only a part of the graph available to the scheduler and consequently only a fraction of the concurrency opportunities in the applications is visible at any given time.

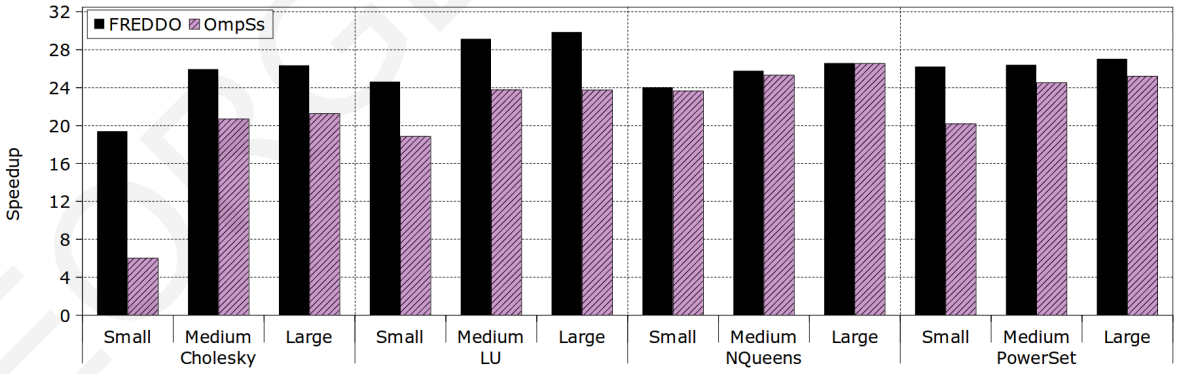


Figure 72: FREDDO vs. OmpSs on an AMD node using 32 cores.

FREDDO is also compared with OpenMP [19] on benchmarks with complex data-dependencies (LU and Cholesky) as well as on benchmarks which are embarrassingly parallel (BMMULT and Blackscholes). The performance results are depicted in Figure 73. The maximum possible speedup when we use OpenMP is 32 whereas when we use FREDDO is 31. In the case of BMMULT and Blackscholes, which are embarrassingly parallel benchmarks, both frameworks scale very well and achieve very good performance, especially for the largest problem size. However, in the case of LU

GEORGE MATHEOU

and Cholesky, which are benchmarks with high-complexity graphs, FREDDO outperforms OpenMP in all cases. For the largest problem size, FREDDO is $2.64\times$ faster than OpenMP for Cholesky and $1.24\times$ faster for LU. FREDDO achieves better performance than OpenMP since the former allows asynchronous data-driven execution while the latter relies on the fork-join paradigm which incurs more overheads (e.g., barriers are used between phases of computation).

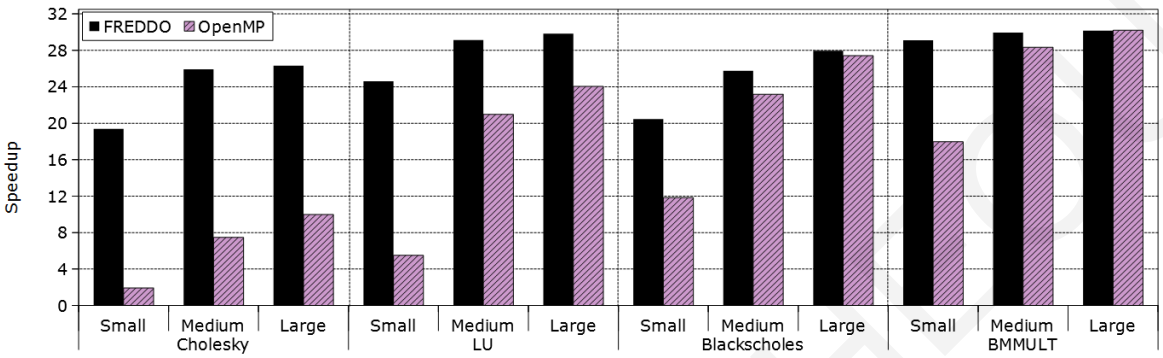


Figure 73: FREDDO vs. OpenMP on an AMD node using 32 cores.

7.6 Distributed FREDDO Evaluation

7.6.1 Experimental Setup

For the distributed FREDDO evaluation we have used two different systems, AMD and CyTera (see Table 5). AMD is a 4-node system with a total of 128 cores. CyTera is a 64-node Intel HPC system with a total of 768 cores. The benchmark suite used for the distributed evaluation contains three applications which require low communication between the nodes (BMMULT, Blackscholes and Swaptions), three benchmarks with complex dependency graphs that require heavy inter-node communication (LU, QR and Cholesky) and two recursive algorithms (Fibonacci and PowerSet) that require medium inter-node communication. The description of the benchmarks can be found in Section 7.2.

For the benchmarks working on tile/block matrices we have used both single-precision (SP) and double-precision (DP) floating-point dense matrices. All source codes and libraries/packages were compiled using the *-O3* optimization flag. For the performance results which are reported as *speedup*, speedup is defined as $^{Savg}/_{Pavg}$, where $Savg$ is the average execution time of the sequential version of the benchmark (without any FREDDO overheads) and $Pavg$ is the average execution time of the FREDDO implementation. For the average execution times we have executed each benchmark (both sequential and parallel) five times. In the parallel execution time of each execution we have included the time needed for gathering the results to the *RootNode*.

We have executed benchmarks using FREDDO with Custom Network Interface support (called FREDDO+CNI) and with MPI support (called FREDDO+MPI). Currently, FREDDO+CNI supports

GEORGE MATHEOU

only Ethernet-based interconnects. The default FREDDO implementation for the AMD system is FREDDO+CNI. In CyTera, the MPI libraries provided to us are configured for the InfiniBand interconnect. As such, we are using the FREDDO+MPI implementation as the default since it provides faster communication compared to FREDDO+CNI. For the FREDDO+MPI implementation we are using the OpenMPI library (V1.8.4 for CyTera and V2.0.1 for AMD). Notice that for both implementations, the size of the Context values is set to 64-bit.

7.6.2 Performance Evaluation

We have performed a scalability study in order to evaluate the performance of the proposed framework, by varying the number of nodes on the two systems. Each benchmark is executed with three different problem sizes. For the tiled algorithms (BMMULT, LU, Cholesky and QR) we choose the optimal tile size, for both the sequential and parallel implementation of each algorithm. For each different execution (problem size and number of nodes), we run experiments with three different tile sizes: 32×32, 64×64 and 128×128. Out of the total number of cores in each node, one of them is used for executing the TSU code while the rest are used for executing the Kernels. Unlike the Kernels and the TSU which are pinned to specific cores, the Network Manager's *receiving thread* is not pinned to any specific core. This gives the opportunity to the operating system to move the *receiving thread* to an idle core, or to migrate it regularly between the cores. For the single-node execution of the benchmarks, the Network Manager's *receiving thread* is disabled. Given this configuration, the maximum possible speedup for the single-node execution is 31 on the AMD system and 11 on the CyTera system. When all nodes are used, the maximum possible speedup is 124 and 704, on AMD and CyTera, respectively.

Figures 74 and 75 depict the results for the AMD and CyTera systems, respectively. On the former system we have executed all the benchmarks, including both single-precision and double-precision versions of the algorithms working on tile/block matrices. On the latter system we have executed the two recursive algorithms and the single-precision versions of the tiled algorithms. Blackscholes and Swaptions as well as the double-precision versions of the tiled algorithms are excluded from our performance evaluation on the CyTera system in order to save computational resources (CPU hours). *Ideal speedup* refers to the maximum speedup that can be achieved in relation to the number of cores used for the parallel execution. For example, on CyTera for the 64-node configuration, the *Ideal speedup* is equal to 768.

From the performance results we observe that generally as the input size increases, the system scales better (especially for the benchmarks with the complex dependency graphs). This is expected, as larger problem sizes allow for amortizing the overheads of the parallelization. Table 15 depicts the average sequential time (in seconds) of the sequential version of the benchmarks that were executed on both systems. The double-precision versions of the algorithms achieve slightly lower speedups

GEORGE MATHEOU

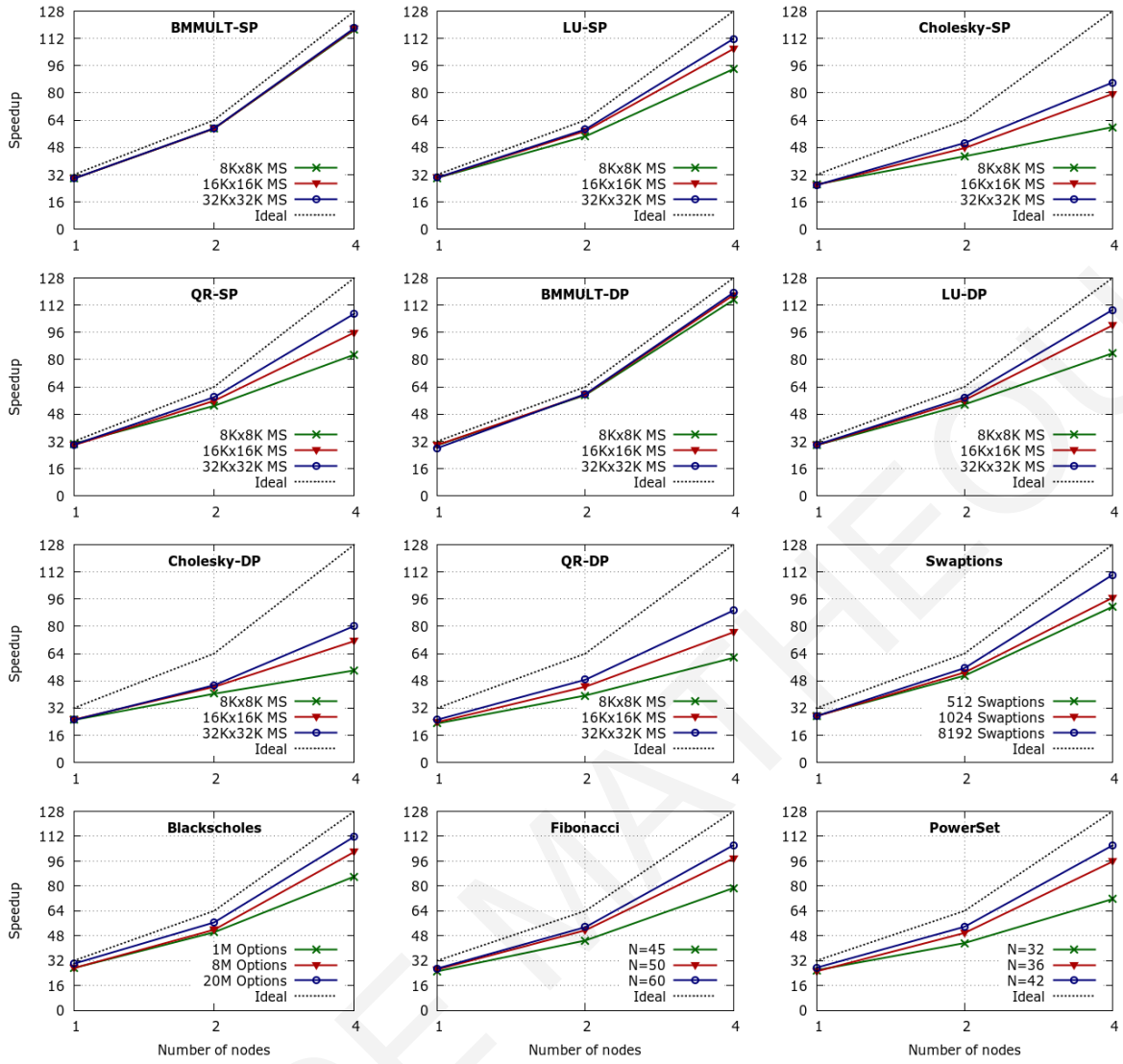


Figure 74: Strong scalability and problem size effect on the AMD system using FREDDO+CNI (MS=Matrix Size, SP=Single-Precision, DP=Double-Precision, $K = 2^{10}$, $M = 10^6$).

compared to the single-precision ones, since in the former case the data exchanged in the network is doubled.

BMMULT, Blackscholes and Swaptions achieve very good speedups due to the low data sharing and low data exchange between the nodes. On AMD, for the 4-node configuration and the largest problem size, they achieve up to 93% of the ideal speedup. On CyTera, for the 64-node configuration and the largest problem size, BMMULT achieves 84% of the ideal speedup. LU, QR and Cholesky are classic dense linear algebra workloads with complex dependency graphs. FREDDO ended up with lower speedups as the number of nodes increases, due to the heavy inter-node communication and the complexity of the algorithms. When utilizing all the nodes of CyTera for the largest problem size, FREDDO achieves up to 61% of the ideal speedup for these complex algorithms. However, it is expected that for larger problem sizes, a better performance can be achieved.

GEORGE MATHEOU

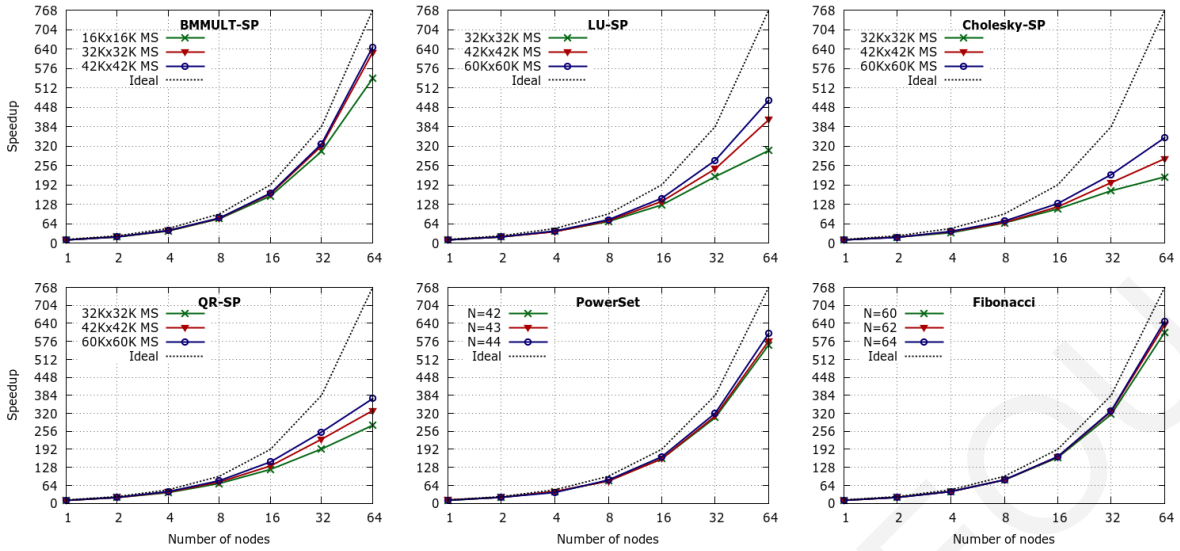


Figure 75: Strong scalability and problem size effect on the CyTera system using FREDDO+MPI (MS=Matrix Size, SP=Single-Precision, $K = 2^{10}$).

Benchmark	AMD			CyTera		Benchmark	AMD		CyTera	
	Matrix Size	Execution Time of SP	Execution Time of DP	Matrix Size	Execution Time of SP		Problem Size	Execution Time	Problem Size	Execution Time
BMMULT	8Kx8K	1,893.34	2,143.96	16Kx16K	5,664.72	Fibonacci	N=45	12.59	N=60	8,885.24
	16Kx16K	15,110.28	17,196.18	32Kx32K	45,340.37		N=50	140.70	N=62	23,250.33
	32Kx32K	120,816.96	139,346.05	42Kx42K	102,586.97		N=60	17,291.87	N=64	60,795.54
LU	8Kx8K	636.48	709.37	32Kx32K	15,350.72	PowerSet	N=32	18.61	N=42	9,721.38
	16Kx16K	5,135.44	5,745.75	42Kx42K	34,411.99		N=36	304.29	N=43	19,295.20
	32Kx32K	41,572.70	47,079.94	60Kx60K	101,666.51		N=42	20,098.89	N=44	38,261.29
Cholesky	8Kx8K	170.10	181.76	32Kx32K	4,106.64	Swaptions	512 Swaptions	50.53	-	-
	16Kx16K	1,310.03	1,404.20	42Kx42K	9,273.48		1024 Swaptions	97.67	-	-
	32Kx32K	10,313.71	11,068.11	60Kx60K	27,034.89		8192 Swaptions	781.77	-	-
QR	8Kx8K	5,016.25	3,900.37	32Kx32K	131,237.58	Blackscholes	1M Options	141.19	-	-
	16Kx16K	40,585.11	31,867.58	42Kx42K	307,914.23		8M Options	1,123.27	-	-
	32Kx32K	324,817.37	268,552.39	60Kx60K	921,452.84		20M Options	2,826.60	-	-

Table 15: Average sequential execution time (in seconds) of the sequential version of the benchmarks.

System	Problem Size	Benchmark	Number of Nodes			System	Problem Size	Benchmark	Number of Nodes						
			1	2	4				1	2	4	8	16	32	64
AMD	Smaller	Fibonacci	29	29	29	CyTera	Smaller	Fibonacci	35	35	35	35	35	35	35
		PowerSet	14	12	11			PowerSet	15	14	15	15	16	16	16
	Medium	Fibonacci	30	30	30		Medium	Fibonacci	35	35	35	35	35	35	35
		PowerSet	13	14	15			PowerSet	15	15	15	15	16	16	17
	Largest	Fibonacci	33	33	33		Largest	Fibonacci	35	35	35	35	35	35	37
		PowerSet	13	13	15			PowerSet	15	15	16	16	15	15	17

Table 16: Thresholds used for the execution of the recursive algorithms.

The recursive algorithms (Fibonacci and PowerSet) also achieve very good speedups. For the 4-node configuration on the AMD system and the largest problem size, FREDDO achieves about 83% of the ideal speedup (106 out of 128). For the 64-node configuration on CyTera, FREDDO achieves 84% (648 out of 768) for the Fibonacci and 79% (604 out of 768) for the PowerSet of the ideal speedup, also for the largest problem size. For minimizing the overheads of the parallel recursive implementations we have used *thresholds* in order to control the number of DThread instances that are used for executing recursive calls. For each problem size of the algorithms we test various *thresholds*

GEORGE ATHEOU

and we choose the one that provides the best performance. Table 16 depicts the thresholds used to achieve the best performance.

To conclude, distributed FREDDO scales well and effectively leverages the decoupling of synchronization and execution. Table 17 depicts the minimum, maximum and average speedup results, on both systems, for each problem size and number of nodes. Next to each speedup value the utilization percentage of the available cores is presented. The results show that FREDDO utilizes the resources of both systems efficiently, especially for the largest problem size. For the largest problem size and when all the available nodes are used, FREDDO achieves an average of 82% of the ideal speedup on AMD and 67% on CyTera.

System	Problem Size	Speedup	Number of Nodes						
			1	2	4	8	16	32	64
AMD	Smaller	Avg	27.5 (86%)	49.2 (77%)	83.1 (65%)	-	-	-	-
		Min	22.9 (72%)	39.3 (61%)	54.1 (42%)	-	-	-	-
		Max	30.6 (96%)	59.2 (92%)	117.0 (91%)	-	-	-	-
	Medium	Avg	27.5 (86%)	52.7 (82%)	96.5 (75%)	-	-	-	-
		Min	23.7 (74%)	44.5 (69%)	71.3 (56%)	-	-	-	-
		Max	30.2 (94%)	59.5 (93%)	117.9 (92%)	-	-	-	-
	Largest	Avg	28.0 (87%)	54.9 (86%)	104.9 (82%)	-	-	-	-
		Min	25.0 (78%)	45.4 (71%)	80.2 (63%)	-	-	-	-
		Max	30.3 (95%)	59.8 (93%)	119.3 (93%)	-	-	-	-
CyTera	Smaller	Avg	10.5 (88%)	20.5 (85%)	38.9 (81%)	75.2 (78%)	139.1 (72%)	251.3 (65%)	419.1 (55%)
		Min	10.4 (86%)	19.7 (82%)	34.1 (71%)	66.6 (69%)	112.5 (59%)	172.5 (45%)	218.2 (28%)
		Max	10.8 (90%)	21.2 (89%)	41.9 (87%)	82.8 (86%)	162.5 (85%)	316.1 (82%)	608.3 (79%)
	Medium	Avg	10.7 (89%)	20.6 (86%)	40.0 (83%)	76.8 (80%)	145.9 (76%)	271.5 (71%)	475.8 (62%)
		Min	10.4 (86%)	19.8 (82%)	37.2 (78%)	67.8 (71%)	119.6 (62%)	199.3 (52%)	278.0 (36%)
		Max	11.9 (99%)	21.2 (88%)	42.3 (88%)	83.1 (87%)	165.3 (86%)	326.4 (85%)	634.7 (83%)
	Largest	Avg	10.5 (88%)	20.7 (86%)	40.3 (84%)	79.8 (83%)	153.5 (80%)	288.1 (75%)	514.6 (67%)
		Min	10.3 (86%)	18.8 (78%)	38.4 (80%)	72.5 (76%)	130.1 (68%)	225.9 (59%)	347.4 (45%)
		Max	10.9 (91%)	21.5 (90%)	41.9 (87%)	83.4 (87%)	166.3 (87%)	329.0 (86%)	647.9 (84%)

Table 17: Speedup results along with the utilization percentage of the available cores in each case.

7.6.3 FREDDO: CNI vs MPI

In this section we study the performance penalties of using MPI instead of CNI, for the benchmarks that have medium and heavy inter-node communication. For our experiments we have used the AMD system with all the available nodes. The results are shown in Figure 76 and are normalized based on the average execution time of FREDDO+CNI. The comparisons show that FREDDO+CNI is 80%, 25% and 5% faster than FREDDO+MPI on average, for the smallest, medium and largest problem sizes, respectively. This indicates that MPI has more overheads which affect the performance of the Network Manager's receiving thread as well as the sending operations of the Kernels. MPI has more overheads than CNI since it's a much larger library which contains more functionalities than CNI. However, the MPI's overheads are hidden as the benchmark's input size increases. Thus, FREDDO+MPI can be used for real life applications that have enormous input sizes (at least in the order of our largest problem size). This solution can provide better portability to FREDDO applications, especially when targeting large-scale HPC systems with different architectures.

GEORGE THEOU

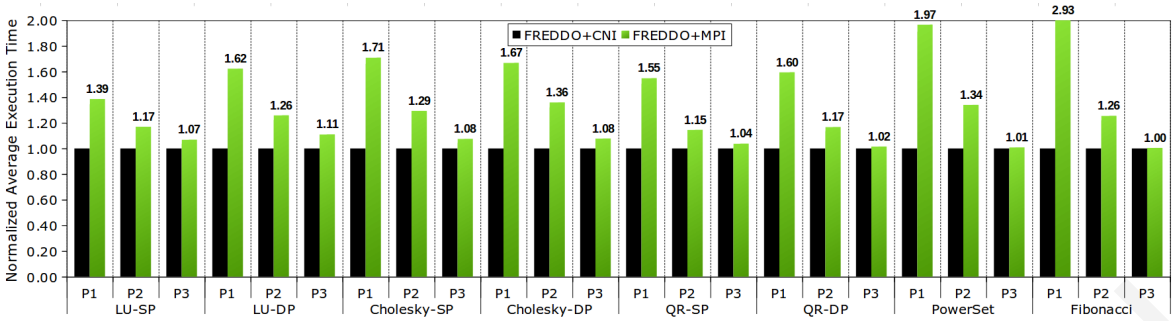


Figure 76: FREDDO+CNI vs. FREDDO+MPI on AMD for the 4-node configuration. P1, P2 and P3 indicate the smaller, medium and largest problem sizes, respectively.

7.6.4 Performance comparisons with other systems

Distributed FREDDO implementation is compared with MPI, DDM-VM [63] and OmpSs@Cluster [129, 130]. MPI is the reference programming model in HPC systems and clusters. DDM-VM is a data-flow system similar to FREDDO which supports static dependency resolution, i.e., the programmer/compiler is responsible for constructing the dependency graph. OmpSs@Cluster is a data-flow-based system which supports dynamic dependency resolution, i.e., the dependency graph is constructed dynamically, at runtime. For the comparisons between FREDDO and MPI we have used Cholesky, LU, Fibonacci and PowerSet. For the MPI implementation of Cholesky, we used the *pspotrf* and *pdpotrf* routines of ScaLAPACK [185] (V2.0.2). The MPI implementation of LU was retrieved from [186] and utilizes MPI's one-sided communication functionalities. The recursive algorithms implemented in MPI are based on the Fibonacci algorithm presented in [187] which uses the MPI's dynamic process management support through *MPI_Comm_spawn*. Additionally, the recursive algorithms were modified to support thresholds in order to improve the performance. For the comparisons between FREDDO and DDM-VM, we used Cholesky and LU. Finally, we compare FREDDO with OmpSs@Cluster using Cholesky, LU and QR. For all frameworks we choose the configurations that achieve the optimal performance (e.g., tile sizes, thresholds for the recursive algorithms and grid configurations for the ScaLAPACK implementations).

OmpSs@Cluster was installed with GASNet [188] (V1.28.2) and OpenMPI (V1.8.4) on CyTera. We used the latest stable OmpSs package which includes the Mercurium compiler V2.0.0-28-06-2017 and the Nanos++ runtime V0.13-29-06-2017. The OmpSs benchmarks were executed using GASNet's *ibv-conduit*. In order to have fair comparisons, FREDDO is compared with DDM-VM using the FREDDO+CNI implementation and with MPI and OmpSs@Cluster using the FREDDO+MPI implementation. The reason is that the MPI library incurs more overheads (as shown in Figure 76) compared to a custom network interface with less functionalities.

GEORGE MATHEOU

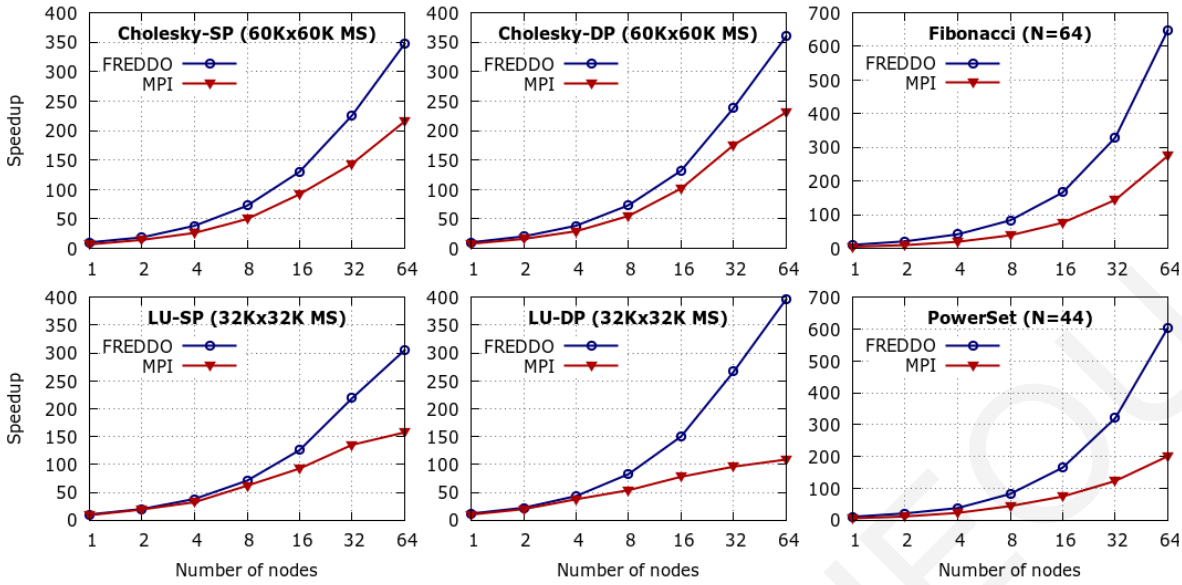


Figure 77: FREDDO+MPI vs. MPI on CyTera.

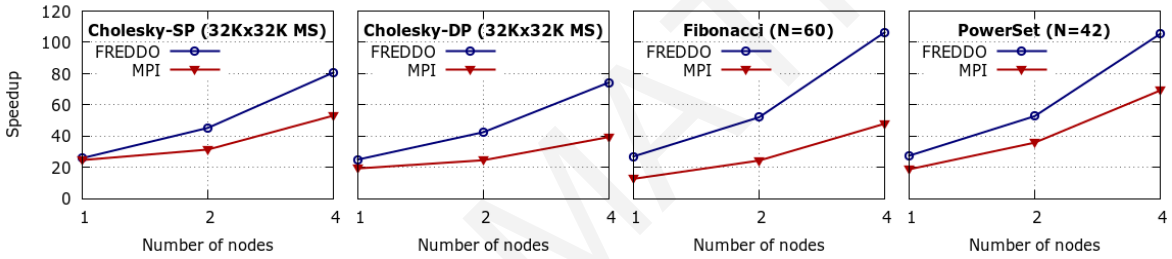


Figure 78: FREDDO+MPI vs. MPI on AMD.

7.6.4.1 FREDDO vs. MPI

The comparison results of FREDDO and MPI, on CyTera and AMD, are depicted in Figures 77 and 78, respectively. LU is not included in our comparisons on AMD since the system does not support Remote Direct Memory Access (RDMA), which is required by the MPI implementation of the algorithm. Our framework scales better than MPI. On average, FREDDO is $1.79\times$ faster than MPI, on AMD, for the 4-node configuration and $2.35\times$ faster, on CyTera, for the 64-node configuration. In the case of Cholesky and LU, FREDDO performs better since it exploits the advantages of having fine-grained threads/tasks executing asynchronously in a data-driven manner. MPI relies on the fork-join paradigm and it uses barriers to synchronize between phases of computation. Synchronization constructs, like barriers, can have a negative impact on performance, especially when the number of nodes increases. Contrary, FREDDO does not use such synchronization constructs as it relies solely on data-driven mechanisms for its operations.

In the MPI implementations of the recursive algorithms, the *MPI_Comm_spawn* routine is used to spawn children recursive calls through new MPI processes. A parent-instance waits its children-instances to complete before it proceeds with computations, thus increasing the runtime overheads (a

GEORGE MATHEOU

parent-instance blocked until all its children-instances finished their execution). Also, additional overheads are introduced by the dynamic allocation of new MPI processes. In FREDDO such overheads are eliminated by executing recursive algorithms based on data-flow with continuations; a continuation DThread instance is activated to process the results of the children-instances when all of them finished their execution. Moreover, all instances are executed by the Kernels, thus there is no need to allocate new resources (processes, threads, etc.).

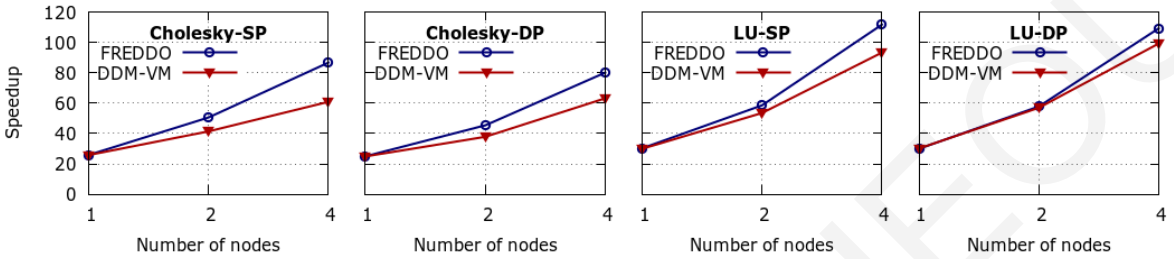


Figure 79: FREDDO+CNI vs. DDM-VM on AMD (MS: $32K \times 32K$).

7.6.4.2 FREDDO vs. DDM-VM

The comparison results of FREDDO and DDM-VM, on the AMD system, are depicted in Figure 79. On average, FREDDO is $1.25\times$ faster than DDM-VM, for the 4-node configuration. Although DDM-VM and FREDDO are based on the same execution model, FREDDO achieves better performance for three main reasons:

1. DDM-VM follows a Context-based distribution scheme (similar to this work) where each DThread instance is mapped and executed on a specific core of the distributed system. In FREDDO, the DThread instances are mapped to specific nodes and TSU's Scheduler distributes them to the Kernels with the least work-load. This approach can better improve the load-balancing in each node. For example, consider that two DThread instances with the same Context value and different TID are scheduled to run on node x. In DDM-VM, the two instances will be scheduled for execution on the same core, sequentially. In FREDDO, the two instances will be scheduled to run on the cores with the least amount of work. This allows the two instances to be executed in parallel, on two different cores.

2. FREDDO provides optimized TSU and Network Manager. For example, FREDDO uses *atomic variables* to implement the distributed termination detection algorithm, where DDM-VM uses lock/unlock operations which incur more overheads.

3. In DDM-VM, the TSU and the *receiving thread* of the Network Interface Unit (similar to the *receiving thread* of FREDDO's Network Manager) are pinned on the same core. This approach can affect the scheduling and network operations, thus, increasing the runtime overheads. In

GEORGE MATHEOU

FREDDO, the *receiving thread* is not pinned to any specific core which gives the flexibility to the operating system to schedule it appropriately.

7.6.4.3 FREDDO vs. OmpSs@Cluster

Figure 80 compares FREDDO with OmpSs@Cluster on CyTera. We have used the *affinity* scheduling policy which is used for running applications on a cluster. We have executed the OmpSs benchmarks with different tile sizes (from 32×32 to 2048×2048) and we found that the 512×512 tile size achieved the best performance, in all cases. Our experiments show that OmpSs does not perform well on benchmarks with small tiles. Smaller tile sizes increase the number of tasks which increase the Nanos++'s workload required to determine the data-dependencies between those tasks. Notice that OmpSs@Cluster reserves one core for network functionalities.

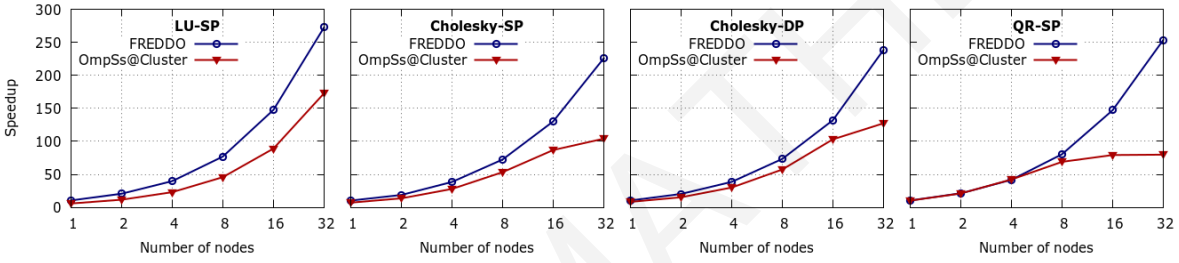


Figure 80: FREDDO+MPI vs. OmpSs@Cluster on CyTera (MS: $60K \times 60K$).

FREDDO outperforms OmpSs@Cluster, especially for the configurations with large number of nodes. For the 32-node configuration on CyTera, FREDDO is up to $3.18\times$ faster than OmpSs@Cluster ($2.20\times$ faster on average). The reason is twofold. First, OmpSs determines the dependencies at runtime. This eases programmability since programmers only need to annotate the sequential code with compiler directives. However, it increases runtime overheads. In FREDDO, the dependencies are provided by programmers (through Update commands) which increases the programming effort but runtime overheads are reduced. Second, the execution model of OmpSs@Cluster is based on a master-worker design where the master is responsible for: (1) assigning tasks to the remote nodes and (2) preserving data coherency. OmpSs@Cluster aims to create an identical address space on each node, which gives the view of a single distributed address space [129], like in FREDDO. However, a master-worker scheme suffers from scalability issues on large clusters due to the bottleneck constituted by the master. Although OmpSs supports task nesting on cluster nodes to reduce the pressure on the master node, our results show that FREDDO scales better. FREDDO implements a peer-to-peer network (a node can send data and Updates to any other node) and a lightweight distribution scheme based on static mapping. It also uses data forwarding to reduce latencies.

GEORGE MATHEOU

7.6.5 Network Traffic Analysis

In order to study the efficiency of the proposed mechanisms for reducing the network traffic, during a distributed data-driven execution, we have performed a traffic analysis for both FREDDO and DDM-VM. We are comparing our system with DDM-VM since the latter does not provide any mechanisms for reducing the network traffic in DDM applications. The experiments were conducted on the AMD system for two benchmarks from our benchmark suite, Cholesky and LU (single precision versions). For our experiments a root access was required for capturing the network traffic. Thus, the AMD system was used since it is the only system where we have root access. Figure 81 depicts the total TCP packets (in Millions) and the total data (in GB) that are exchanged between the nodes of the AMD system, for the 4-node configuration and the largest problem size ($32K \times 32K$). The benchmarks were executed with three different tile sizes: 32×32, 64×64 and 128×128.

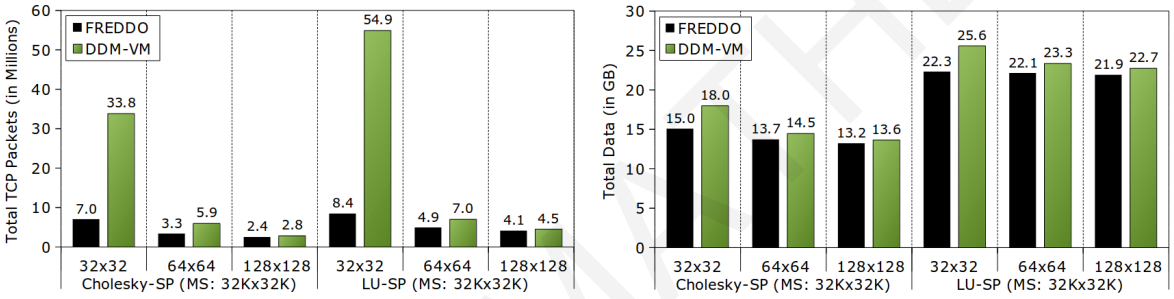


Figure 81: Network traffic analysis: FREDDO against DDM-VM on the AMD system, for the 4-node configuration and the largest problem size ($32K \times 32K$).

For the traffic analysis experiments, we used the *TShark* tool [189] (V2.2.3) and configured it to capture the traffic that is exchanged between the TCP ports that were reserved for the inter-node communication. It is important to note that in FREDDO, the size of the Context values is set to 64-bit, whereas DDM-VM supports only 32-bit Context values. Larger Context values allow to execute benchmarks with large problem sizes and fine-grained threads (e.g., the LU benchmark on the CyTera system with $60K \times 60K$ matrix size and 32×32 tile size).

The comparison results show that FREDDO reduces the total TCP packets and data, especially for the smallest tile size where the frequency of the communication is increased between the nodes of the system. In the case of Cholesky, FREDDO reduces the total TCP packets by $4.85\times$, $1.79\times$ and $1.16\times$, for the 32×32, 64×64 and 128×128 tile sizes, respectively. In the case of LU, FREDDO reduces the total TCP packets by $6.55\times$, $1.44\times$ and $1.11\times$, for the 32×32, 64×64 and 128×128 tile sizes, respectively.

Furthermore, FREDDO reduces the total amount of data by 16.7%, 5.5% and 2.9% for Cholesky and 12.9%, 5.2% and 3.5% for LU, for the 32×32, 64×64 and 128×128 tile sizes, respectively. It is easy to observe that the total number of TCP packets and the total amount of data are not reduced with the same ratio. This is because the largest percentage of the total amount of data consists of

the computed matrix tiles that are forwarded from the producer to consumer nodes. This percentage is approximately the same in both frameworks since FREDDO reduces the network traffic mainly through optimizations on sending Update operations. The number of Update operations is high in benchmarks with high-complexity dependency graphs, thus, a higher number of TCP packets is used to carry such operations in DDM-VM.

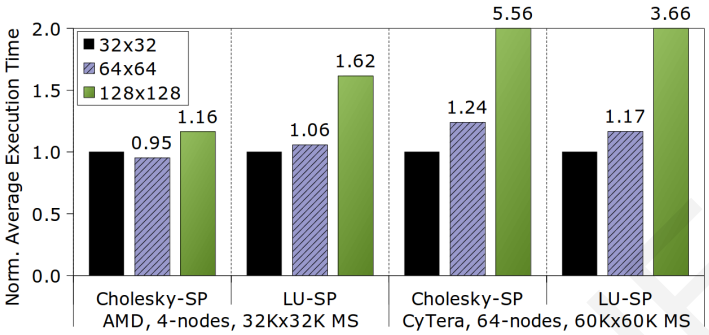


Figure 82: Tile size effect on the AMD and CyTera systems using FREDDO.

Although the proposed mechanisms for reducing the network traffic are performing better for relatively small tile sizes (fine-grained threads), we expect to have a high positive impact on benchmarks that run on HPC systems with a large number of cores/nodes. Future HPC systems (e.g., exascale architectures) are projected to have up to billions of fine-grained threads executing asynchronously [168, 33, 32]. As a very small indication, in Figure 82, we provide the normalized average execution time of the LU and Cholesky benchmarks that are executed on both systems for three different tile sizes. The timings are normalized based on the execution time of the 32×32 tile size. The results show that larger tile sizes (i.e., coarse-grained threads) can negatively affect the performance, especially in the CyTera system with the 64-node configuration. We would like to note that we have tested experiments with even smaller tile sizes (e.g., 16×16) and the performance was not good, as expected, especially, on the AMD system. This is because smaller tile sizes can increase the runtime overheads since more DThread instances will be created which stress the TSU. However, smaller tile sizes increase significantly the Update operations, thus our mechanisms may further reduce the network traffic compared to the DDM-VM system.

7.6.6 Execution Times

In this subsection we present the best average execution times of FREDDO+CNI, FREDDO+MPI, MPI, DDM-VM and OmpSs@Cluster, that were used for the performance evaluation of the distributed FREDDO implementation as well as for the performance comparisons, on both AMD and CyTera. In each case, we present the average execution times (in seconds) for all benchmarks, node configurations and problem sizes. Notice that, in the case of the tile algorithms,

GEORGE MATHEOU

the best average execution time of an experiment (problem size and number of nodes) is the minimum among all the execution times of all tile/block sizes tested for that experiment. Similarly, in the case of recursive algorithms, the best average execution time of an experiment (problem size and number of nodes) is the minimum among all the execution times of all thresholds tested for that experiment. The execution times are separated in two categories: execution times on the AMD system (Section 7.6.6.1) and execution times on the CyTera system (Section 7.6.6.2).

7.6.6.1 AMD System

Problem Size	BMMULT-SP			BMMULT-DP			LU-SP			LU-DP		
	1	2	4	1	2	4	1	2	4	1	2	4
Smaller	63.87	32.08	16.18	72.10	36.22	18.64	21.15	11.69	6.77	23.91	13.22	8.47
Medium	509.21	255.86	128.58	575.85	289.03	145.85	170.31	89.13	48.52	193.70	102.15	57.37
Larger	4064.04	2039.16	1023.46	4970.50	2331.35	1167.92	1372.83	709.77	372.06	1572.88	814.67	430.97
Problem Size	**Cholesky-SP**			**Cholesky-DP**			**QR-SP**			**QR-DP**		
	1	2	4	1	2	4	1	2	4	1	2	4
Smaller	6.48	3.98	2.84	7.28	4.50	3.36	174.36	100.86	64.27	170.05	99.15	63.40
Medium	50.62	27.53	16.50	55.70	31.58	19.69	1365.79	727.23	423.21	1345.62	715.61	416.58
Larger	398.39	203.94	119.79	443.05	243.72	138.01	10802.81	5604.97	3035.84	10675.07	5521.82	3007.10
Problem Size	**Fibonacci**			**PowerSet**			**Blackscholes**			**Swaptions**		
	1	2	4	1	2	4	1	2	4	1	2	4
Smaller	0.50	0.28	0.16	0.72	0.43	0.26	5.18	2.81	1.64	1.84	0.99	0.55
Medium	5.34	2.73	1.44	12.10	6.11	3.18	41.28	21.65	11.02	3.60	1.85	1.01
Larger	641.07	322.98	161.81	732.48	373.62	189.44	94.12	49.95	25.30	28.65	14.10	7.09

Table 18: Best average execution time (in seconds) for FREDDO+CNI on AMD.

Framework	Cholesky-SP (32K x 32K MS)			Cholesky-DP (32K x 32K MS)			Fibonacci (N=60)			PowerSet (N=42)		
	1	2	4	1	2	4	1	2	4	1	2	4
FREDDO	398.39	228.01	127.89	443.05	260.53	148.91	641.07	332.39	162.57	732.48	380.93	190.70
MPI	418.24	328.86	194.24	573.24	451.01	281.11	1368.21	711.21	361.12	1072.26	561.57	290.48

Table 19: FREDDO+MPI vs. MPI: best average execution time (in seconds) on AMD.

Framework	Cholesky-SP (32K x 32K MS)			Cholesky-DP (32K x 32K MS)			LU-SP (32K x 32K MS)			LU-DP (32K x 32K MS)		
	1	2	4	1	2	4	1	2	4	1	2	4
FREDDO	398.39	203.94	119.79	443.05	243.72	138.01	1372.83	709.77	372.06	1572.88	814.67	430.97
DDM-VM	399.82	248.99	169.45	444.08	292.58	175.14	1388.15	776.75	445.56	1562.33	826.56	474.27

Table 20: FREDDO+CNI vs. DDM-VM: best average execution time (in seconds) on AMD.

GEORGE MARKOU

7.6.6.2 CyTera System

Problem Size	BMMULT-SP							Cholesky-SP						
	1	2	4	8	16	32	64	1	2	4	8	16	32	64
Smaller	546.77	281.61	142.51	70.60	36.55	18.74	10.44	393.99	208.92	120.34	61.69	36.49	23.81	18.82
Medium	4373.33	2199.56	1103.73	555.09	279.67	141.62	72.15	887.23	468.81	249.26	136.70	77.52	46.54	33.35
Larger	9883.14	4963.75	2468.96	1241.79	623.20	313.68	158.73	2629.21	1439.06	701.94	372.99	207.73	119.69	77.83
	LU-SP							QR-SP						
Problem Size	1	2	4	8	16	32	64	1	2	4	8	16	32	64
Smaller	1461.62	758.89	400.58	216.02	122.02	70.08	50.31	12619.00	6408.24	3433.46	1861.38	1087.89	679.11	473.25
Medium	3277.83	1690.10	917.33	467.32	250.40	140.66	84.68	29550.31	15019.81	7702.91	4071.76	2318.05	1350.93	933.83
Larger	9673.31	4910.79	2556.44	1320.69	688.60	372.92	216.27	88261.77	43236.43	22070.06	11449.27	6262.52	3635.15	2476.29
	Fibonacci							PowerSet						
N	1	2	4	8	16	32	64	1	2	4	8	16	32	64
Smaller	826.13	422.59	212.01	107.30	54.67	28.11	14.61	918.10	457.53	236.08	121.69	61.49	31.90	17.27
Medium	2171.38	1104.12	555.28	279.63	140.64	71.24	36.63	1614.48	911.93	456.13	244.53	121.97	62.12	33.43
Larger	5681.77	2882.58	1451.11	728.86	365.65	184.80	93.84	3509.56	1780.81	995.51	461.56	231.43	119.38	63.39

Table 21: Best average execution time (in seconds) for FREDDO+MPI on CyTera.

	Cholesky-SP (60K x 60K MS)							Cholesky-DP (60K x 60K MS)						
	1	2	4	8	16	32	64	1	2	4	8	16	32	64
FREDDO	2629.21	1439.06	701.94	372.99	207.73	119.69	77.83	3681.79	1893.62	997.48	524.37	291.62	161.37	106.95
MPI	3938.92	1854.57	1029.40	541.32	293.95	188.08	125.41	4764.66	2389.62	1319.07	697.31	378.65	219.45	166.50
	Fibonacci (N=64)							PowerSet (N=44)						
	1	2	4	8	16	32	64	1	2	4	8	16	32	64
FREDDO	5681.77	2882.58	1451.11	728.86	365.65	184.80	93.84	3509.56	1780.81	995.51	461.56	231.43	119.38	63.39
MPI	11659.85	6117.45	3072.30	1563.87	796.71	422.15	221.27	6102.92	3178.64	1654.62	857.14	512.73	310.21	190.29
	LU-SP (32K x 32K MS)							LU-DP (32K x 32K MS)						
	1	2	4	8	16	32	64	1	2	4	8	16	32	64
FREDDO	1461.62	758.89	400.58	216.02	122.02	70.08	50.31	1604.54	864.25	444.85	233.31	128.58	72.58	48.77
MPI	1672.34	793.47	473.08	247.26	165.46	113.43	97.35	1865.90	970.67	513.59	358.70	247.78	201.17	177.53

Table 22: FREDDO+MPI vs. MPI: best average execution time (in seconds) on CyTera.

	Cholesky-SP (60K x 60K MS)						Cholesky-DP (60K x 60K MS)					
	1	2	4	8	16	32	1	2	4	8	16	32
FREDDO	2629.21	1439.06	701.94	372.99	207.73	119.69	3681.79	1893.62	997.48	524.37	291.62	161.37
OmpSs@Cluster	3784.07	1971.60	969.44	506.80	311.38	259.74	4648.45	2516.09	1281.95	671.22	373.56	302.44
	LU-SP (60K x 60K MS)						QR-SP (60K x 60K MS)					
	1	2	4	8	16	32	1	2	4	8	16	32
FREDDO	9673.31	4910.79	2556.44	1320.69	688.60	372.92	88261.77	43236.43	22070.06	11449.27	6262.52	3635.15
OmpSs@Cluster	17392.46	8784.97	4408.04	2222.77	1141.83	588.35	89843.32	43642.95	21793.56	13363.54	11621.30	11548.15

Table 23: FREDDO+MPI vs. OmpSs@Cluster: best average execution time (in seconds) on CyTera.

GEORGE MANIEOS

Chapter 8

Conclusions and Future Work

8.1 Conclusions

In this thesis we presented two different projects based on Data-Driven Multithreading (DDM), a non-blocking multithreading model that allows data-driven scheduling on sequential processors. DDM is utilizing the Thread Scheduling Unit (TSU), a special module that is responsible for scheduling DDM threads (DThreads) in a data-driven manner.

The first project includes the design, development and evaluation of DDM's TSU in hardware. TSU was implemented as a fully-parameterizable IP core using the Verilog HDL. It was synthesized with different configurations and several results are provided, including resource utilization statistics, power consumption estimations and latencies (in cycles) of various TSU operations. The hardware TSU implementation was integrated into a multi-core processor with non-coherent in-order cores, called MiDAS. The processor was prototyped and evaluated on a Xilinx Virtex-6 FPGA using benchmarks with different characteristics. The benchmarks were developed in C/C++ using a software API. MiDAS was evaluated with two different TSU implementations, the Performance Optimized TSU (PO-TSU) and the Area Optimized TSU (AO-TSU). The difference between the two implementations is that PO-TSU has larger and faster Dynamic Synchronization Memory (SM) than AO-TSU. The performance evaluation shows that MiDAS using both TSU implementations scales well and achieves very good results, even on benchmarks with very small problem sizes (e.g., 16×16 matrices).

In the context of this work we provided FPGA resource requirements and power consumption estimations of the MiDAS system. The results show that PO-TSU can easily fit on the Virtex-6 FPGA since it utilizes about 2.5% of its slice registers, 11.0% of its slice LUTs and 27.4% of its BRAM. AO-TSU utilizes less Virtex-6 FPGA resources, compared to PO-TSU: 1.0% slice registers, 3.3% slice LUTs and 2.9% BRAM. Additionally, TSU utilizes a small proportion (12% for the PO-TSU and 7% for the AO-TSU) of the overall power of the MiDAS system. We are very encouraged by the

GEORGE MATHEOU

performance, resource requirements and power estimation results of our hardware prototype. Thus, we are confident that our hardware TSU implementation can support larger number of cores.

We compared our hardware DDM-based TSU that adopts static dependency resolution with the Task Superscalar architecture that implements the StarSs programming framework, a hardware task-based data-flow scheduler that adopts dynamic dependency resolution. Our experimental results show that data-flow schedulers which adopt dynamic dependency resolution, like Task Superscalar, require significant amounts of resources to be implemented; Task Superscalar is $4.94\times$ larger than our TSU implementation regarding the slice registers, while it is $11.34\times$ larger regarding the slice LUTs. Last, our hardware TSU was compared against a software TSU implementation, with both being run on an FPGA fabric under a synthetic application and offering identical functionalities. Our experimental results show that the hardware TSU implementation significantly outperforms in terms of speedup the software TSU implementation, by up to $21.6\times$ when the software TSU utilizes a StaticSM implementation, and by up to $487.1\times$ when the software TSU utilizes a DynamicSM implementation.

The second project, called FREDDO, is an efficient and portable object-oriented implementation of DDM that enables data-driven scheduling on conventional single-node and distributed multi-core systems. It provides new features to the DDM model like recursion support and it extends the DDM's programming interface with the object-oriented programming paradigm. Experiments were performed on two distributed systems, AMD and CyTera. AMD is a 4-node system with a total of 128 cores. CyTera is an open-access HPC system which provides up to 64 nodes per user with a total of 768 cores. Our evaluation analysis demonstrates that FREDDO scales well and achieves comparable or better performance when is compared with other systems, such as OpenMP, MPI, DDM-VM and OmpSs. Last but not least, FREDDO proposes simple and efficient techniques for reducing network traffic during the execution of distributed DDM applications. Our experiments on the AMD system show that FREDDO can reduce the total amount of TCP packets by up to $6.55\times$ and the total amount of data by up to 16.7% when compared to the DDM-VM system.

8.2 Future Work

The analysis presented in the previous chapters proved that FREDDO is able to allow efficient data-driven execution on single-node and distributed multi-core systems. Furthermore, DDM's TSU can be implemented in hardware with a small hardware budget and it can be integrated on conventional multi-core systems. As a proof of concept, we have developed MiDAS, a shared-memory multi-core processor augmented with a hardware TSU implementation and non-coherent in-order processing elements. Our evaluation shows that DDM's architectural support (TSU) allows efficient data-driven scheduling even in the case of benchmarks with very small problem sizes. In this chapter, we present directions to further improve and extent MiDAS and FREDDO.

GEORGE MATHEOU

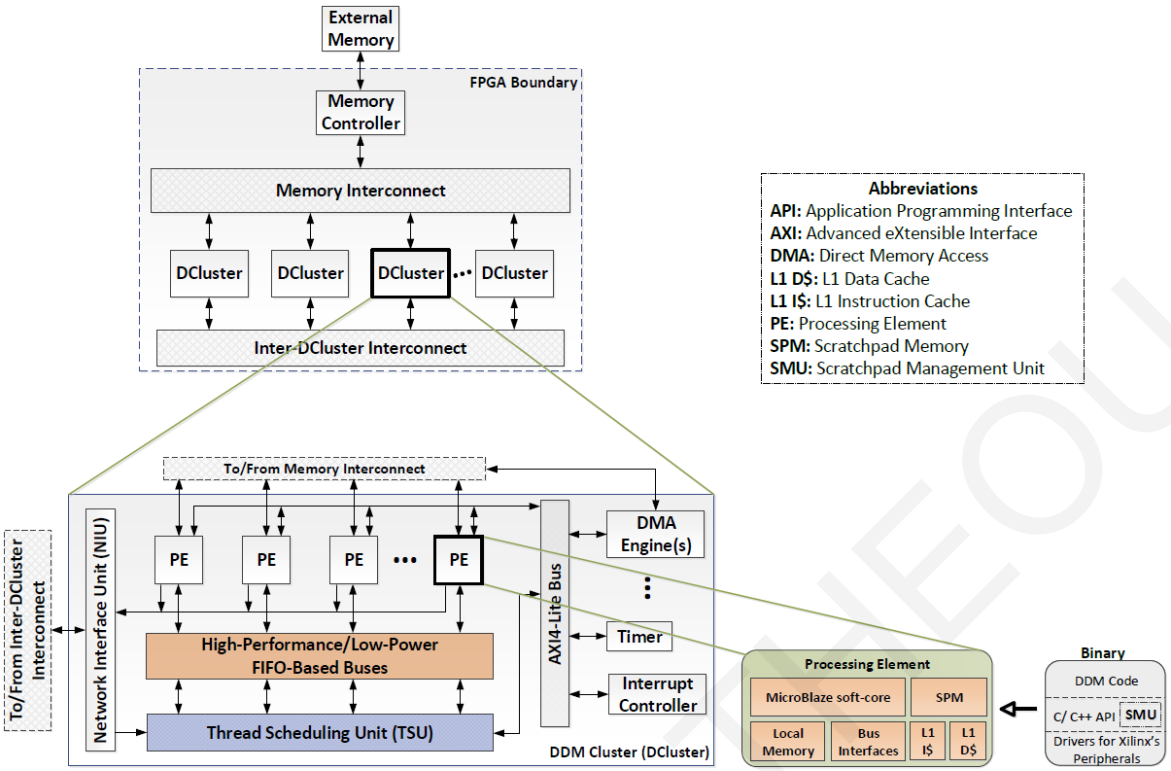


Figure 83: Future distributed data-driven many-core implementation.

8.2.1 MiDAS

8.2.1.1 Implementing a many-core processor based on MiDAS

The MiDAS architecture can be used as a vehicle for building high-performance low-power many-core systems. In particular, we plan to design and develop a many-core system consisting of low-power and low-complexity non-coherent processing elements (PEs) organized into clusters (called DDM clusters or DClusters), and a power-efficient memory model consisting of lightweight caches and scratchpad memories (SPMs) operating simultaneously in parallel. The SPMs will be managed implicitly based on data-driven principles and the DDM's CacheFlow policy [143] in order to simplify programmability and increase application performance. The lightweight caches (non-coherent small L1 caches) will be used to increase the performance of the sequential parts of an application.

A high-level block diagram of a many-core processor, based on MiDAS, is depicted in Figure 83. The DClusters are connected to an off-chip shared Main Memory using a Memory Interconnect (e.g., Network on Chip (NoC) or Hierarchical AXI-4 Bus) and a Memory Controller. Distributed data-driven execution across DClusters can be implemented using the functionalities/techniques provided by the FREDDO framework. Such functionalities include: the inter-node scheduling mechanism, Network Manager's functionalities and compression techniques for reducing the network traffic.

Each DCluster will be an enhanced MiDAS system. It will feature a separate Network Interface Unit (NIU) that will be responsible for receiving/sending data from/to the Inter-DCluster interconnect.

GEORGE MATHEOU

Additionally, each PE of a DCluster will feature its own software-controlled SPM. Implicit memory management of the SPMs, based on the data-driven dependencies of the threads, will be implemented by the Scratchpad Management Unit (SMU). SMU will be an optimized controller of an SPM that will be responsible for loading the data of ready threads from the Main Memory, in its associated SPM, implicitly, using DMA functionalities. After a thread finishes its execution, the SMU will transfer the thread's output data in the Main Memory in order to be read by its consumers. The SMU will be implemented as a part of the C/C++ API.

8.2.1.2 Optimizing the hardware DynamicSM

Updating the RC value of a DThread instance which its associated RC Block is not allocated in DynamicSM requires a significant number of cycles. The reason for this, is that, each Context Search Engine (CSE) checks all its entries to determine if the RC Block does not exist in DynamicSM. An alternative approach is to implement the DynamicSM's SM Indexer (SMI) using Content-Addressable Memories (CAMs) [183]. A CAM is a memory that implements a lookup-table function in a single clock cycle using dedicated comparison circuitry. However, implementing the SMI using CAMs significantly increases the FPGA resource requirements and associated power consumption. In order to justify this, we synthesized a Verilog-based CAM implementation retrieved from [190]. The CAM-based SMI implementation was configured with 4096 entries of 40-bit widths (8 bits for the TID and 32 bits for the Context) targeting the Xilinx ML605 Evaluation Board. Results show that such configuration occupies 41021 slice registers (13% of the FPGA resources), 141947 slice LUTs (94% of the FPGA resources), and 5 BRAMs (1% of the FPGA resources). The CAM-based SMI implementation is much larger than the entire AO-TSU which holds 8192 SMI entries. We note that the AO-TSU occupies 3124 slice registers, 4981 slice LUTs, and 12 BRAMs. To conclude, implementing DynamicSM's SMI using CAMs accelerates the search process, however at the expense of requiring significantly elevated hardware resources and increased power consumption.

8.2.1.3 Provide Recursion Support

The current MiDAS implementation does not provide recursion support. We plan to extend MiDAS's functionalities in order to provide recursion support, based on the functionalities implemented in FREDDO. This will need modifications in the hardware TSU implementation and the C/C++ API.

8.2.1.4 Evaluation of the hardware/software and static/dynamic implementations of the TSU

It would be interesting to further evaluate the hardware and software implementations of the TSU as well as to compare them on FPGA devices. In Section 7.4.6 we presented preliminary results for comparisons between a software and a hardware TSU implementation using a synthetic application.

GEORGE MATHEOU

We can extend our evaluation analysis by comparing MiDAS (8-core with a hardware TSU) with a 9-core system where the one core will be responsible for executing the software TSU implementation. For the comparisons we can use real benchmarks, such as, Cholesky, LU and Blocked Matrix Multiplication.

Furthermore, we plan to extend the functionalities of the TSU to support dynamic dependency resolution, based on the techniques implemented in the DDM-VM system [11, 36]. DDM-VM uses I-Structures [97] for handling the synchronization between the producer and consumer threads in a split-phase manner, i.e., a request issued to an I-structure is independent in time from the received response. This approach will allow us to increase application parallelism in applications where the producer-consumer dependencies cannot be determined at compile-time. Additionally, implementing dynamic and static dependency resolution in the same system will allow us to compare the two techniques in terms of performance, resource requirements and power consumption.

Finally, we plan to implement an efficient Synchronization Memory (SM) implementation for handling the RC values of DThreads with $RC > 1$ and $Nesting \neq 0$, similarly to the StaticSM data-structure of FREDDO. Accessing an RC entry in FREDDO's StaticSM is a direct operation that uses the Context of the DThread's instance. This will allow us to accelerate the Update operations in the hardware TSU implementation as well as to evaluate the overheads of the MiDAS's DynamicSM implementation (see Section 3.2.2.4). However, a direct-mapped SM implementation will need the allocation of all RCs to be performed at the time of creating the Thread Template.

8.2.2 FREDDO

8.2.2.1 Apply data-driven scheduling on heterogeneous HPC systems

Current FREDDO implementation allows data-driven execution on conventional multi-core processors. Future work will be focused on applying data-driven scheduling on heterogeneous HPC systems. Particularly, we are interested on many-core accelerators with software-controlled scratch-pad memories. An example of this architecture is the Sunway SW26010 processor which is the basic building block of the Sunway TaihuLight [191] (ranked 1st in TOP500). Deterministic data prefetching into scratchpad memories using data-driven techniques can improve the locality of sequential processing [10]. We believe that this approach can further improve the performance of HPC systems.

8.2.2.2 Improving the distributed memory model

FREDDO's DSM implementation requires the shared objects to have the same memory size in each node. This simplifies the implementation of the proposed programming model but it limits the total amount of memory used by a DDM program. This limitation can be solved by partitioning a shared data object (e.g., a matrix) across the nodes. If a DThread instance needs a data segment which

GEORGE MATHEOU

belongs to a remote node a data request should be sent to that node requesting the data. Another idea is to use Partitioned Global Address Space (PGAS) libraries like DASH [192]. DASH is a C++ template library that offers distributed data structures and parallel algorithms and implements a compiler-free PGAS approach. This implementation will allow us to explore the benefits and drawbacks of using a data-flow+PGAS model. However, data partitioning will increase the runtime overheads since the DThread instances will wait input data from remote nodes. Such overheads could be mitigated using multithreading (e.g., use more Kernels than the available cores and perform the *yield* function when a DThread waits for input data) and the MPI's one-sided communication functionalities.

8.2.2.3 Improving data locality

FREDDO schedules ready DThread instances to cores using dynamic scheduling, i.e., it locates the Output Queue (OQ) with the least amount of work and it sends the ready DThread instance to that OQ. However, dynamic scheduling can affect the locality. An alternative solution which can increase data locality is to schedule consumer DThread instances to the cores of their producers. This functionality can be implemented using a special data-structure that will store the Kernel IDs of the producer-instances that *fire* (send the last Update operation) the consumer-instances.

8.2.3 Extending the functionalities of DDM

The DDM model can be extended with additional functionalities, such as (1) efficient management and reuse of Context values, (2) eliminating redundant dependencies from applications and (3) parallelism control to optimize the use of resources. For example, in the latter case, we can use techniques like loop throttling to limit the number of DThread invocations that are active concurrently. Such functionalities can be implemented in both MiDAS and FREDDO.

GEORGE MATHEOU

Bibliography

[1] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fourth Edition: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006.

[2] B. D. de Dinechin, R. Ayrignac, P.-E. Beaucamps, P. Couvert, B. Ganne, P. G. de Massas, F. Jacquet, S. Jones, N. M. Chaisemartin, F. Riss *et al.*, "A clustered manycore processor architecture for embedded and accelerated applications." in *HPEC*, 2013, pp. 1–6.

[3] R. Sass and A. G. Schmidt, *Embedded systems design with platform FPGAs: principles and practices*. Morgan Kaufmann, 2010.

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

[5] Maxeler. Dataflow computing. [Online]. Available: https://www.maxeler.com/technology/dataflow-computing/

[6] C. Kyriacou, "Data driven multithreading using conventional control flow microprocessors," Ph.D. dissertation, University of Cyprus, 2005.

[7] K. Stavrou, P. Evripidou, and P. Trancoso, "Ddm-cmp: data-driven multithreading on a chip multiprocessor," in *Embedded Computer Systems: Architectures, Modeling, and Simulation*. Springer, 2005, pp. 364–373.

[8] K. Stavrou, "The TFLUX platform: A portable platform for data-driven multithreading on commodity multiprocessor systems," Ph.D. dissertation, University of Cyprus, 2009.

[9] K. Stavrou, M. Nikolaides, D. Pavlou, S. Arandi, P. Evripidou, and P. Trancoso, "TFlux: A portable platform for data-driven multithreading on commodity multicore systems," in *Parallel Processing, 2008. ICPP'08. 37th International Conference on*. IEEE, 2008, pp. 25–34.

[10] S. Arandi and P. Evripidou, "DDM-VMc: the data-driven multithreading virtual machine for the cell processor," in *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*. ACM, 2011, pp. 25–34.

[11] A. Samer, "The data-driven multithreading virtual machine," Ph.D. dissertation, University of Cyprus, 2011.

[12] The N-queens Problem. Accessed on 10 Aug 2016. [Online]. Available: https://developers.google.com/optimization/puzzles/queens

[13] Knight's tour. Accessed on 10 Aug 2016. [Online]. Available: https://en.wikipedia.org/wiki/Knight%27s_tour

[14] G. E. Moore, "Cramming more components onto integrated circuits," vol. 38, no. 8, 1965.

GEORGE MATHEOU

[15] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang, "The case for a single-chip multiprocessor," *SIGOPS Oper. Syst. Rev.*, vol. 30, no. 5, pp. 2–11, Sep. 1996. [Online]. Available: http://doi.acm.org/10.1145/248208.237140

[16] A. Yarkhan and J. Dongarra, "Lightweight superscalar task execution in distributed memory," 2014.

[17] S. Fuller and L. Millett, "Computing performance: Game over or next level?" *Computer*, vol. 44, no. 1, pp. 31–38, Jan 2011.

[18] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: portable parallel programming with the message-passing interface*. MIT press, 1999, vol. 1.

[19] OpenMP Architecture Review Board, "OpenMP application program interface version 4.5," Nov. 2015. [Online]. Available: http://www.openmp.org/mp-documents/openmp-4.5.pdf

[20] Arvind and R. A. Iannucci, "Two fundamental issues in multiprocessing," in *4th International DFVLR Seminar on Foundations of Engineering Sciences on Parallel Computing in Science and Engineering*. New York, NY, USA: Springer-Verlag New York, Inc., 1988, pp. 61–88. [Online]. Available: http://dl.acm.org/citation.cfm?id=52797.52802

[21] P. Kogge, "Next-generation supercomputers," *IEEE Spectrum, February*, 2011.

[22] S. Zuckerman, J. Suetterlein, R. Knauerhase, and G. R. Gao, "Using a codelet program execution model for exascale machines: position paper," in *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*. ACM, 2011, pp. 64–69.

[23] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' guide*. Siam, 1999, vol. 9.

[24] E. Agullo, B. Hadri, H. Ltaief, and J. Dongarrra, "Comparative study of one-sided factorizations with multiple software packages on multi-core hardware," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM, 2009, p. 20.

[25] A. Haidar, H. Ltaief, A. YarKhan, and J. Dongarra, "Analysis of dynamically scheduled tile algorithms for dense linear algebra on multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 24, no. 3, pp. 305–321, 2012.

[26] J. Kurzak, H. Ltaief, J. Dongarra, and R. M. Badia, "Scheduling linear algebra operations on multicore processors," 2009, LAPACK Working Note 213.

[27] J. B. Dennis, "First version of a data flow procedure language," in *Programming Symposium*. Springer, 1974, pp. 362–376.

[28] B. Lee and A. R. Hurson, "Dataflow architectures and multithreading," *Computer*, vol. 27, no. 8, pp. 27–39, 1994.

[29] J. R. Gurd, C. C. Kirkham, and I. Watson, "The manchester prototype dataflow computer," *Communications of the ACM*, vol. 28, no. 1, pp. 34–52, 1985.

[30] K. Arvind and R. S. Nikhil, "Executing a program on the mit tagged-token dataflow architecture," *Computers, IEEE Transactions on*, vol. 39, no. 3, pp. 300–318, 1990.

[31] J. Landwehr, J. Suetterlein, A. Márquez, J. Manzano, and G. R. Gao, "Application characterization at scale: lessons learned from developing a distributed open community runtime system for high performance computing," in *Proceedings of the ACM International Conference on Computing Frontiers*. ACM, 2016, pp. 164–171.

GEORGE MATHOU

[32] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey, "Hpx: A task based programming model in a global address space," in *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. ACM, 2014, p. 6.

[33] S. Amarasinghe, M. Hall, R. Lethin, K. Pingali, D. Quinlan, V. Sarkar, J. Shalf, R. Lucas, K. Yelick, P. Balanji *et al.*, "Exascale programming challenges," in *Proceedings of the Workshop on Exascale Programming Challenges, Marina del Rey, CA, USA. US Department of Energy, Office of Science, Office of Advanced Scientific Computing Research (ASCR)*, 2011.

[34] W. M. Johnston, J. Hanna, and R. J. Millar, "Advances in dataflow programming languages," *ACM Computing Surveys (CSUR)*, vol. 36, no. 1, pp. 1–34, 2004.

[35] S. Arandi and P. Evripidou, "Programming multi-core architectures using data-flow techniques," in *Embedded Computer Systems (SAMOS), 2010 International Conference on*. IEEE, 2010, pp. 152–161.

[36] S. Arandi, G. Michael, P. Evripidou, and C. Kyriacou, "Combining compile and run-time dependency resolution in data-driven multithreading," in *Data-Flow Execution Models for Extreme Scale Computing (DFM), 2011 First Workshop on*. IEEE, 2011, pp. 45–52.

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

[38] C. Lauderdale, M. Glines, J. Zhao, A. Spiotta, and R. Khan, "Swarm: A unified framework for parallel-for, task dataflow, and distributed graph traversal," *ET International Inc., Newark, USA*, 2013.

[39] J. M. Perez, R. M. Badia, and J. Labarta, "A dependency-aware task-based programming environment for multi-core architectures," in *Cluster Computing, 2008 IEEE International Conference on*. IEEE, 2008, pp. 142–151.

[40] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "Ompss: a proposal for programming heterogeneous multi-core architectures," *Parallel Processing Letters*, vol. 21, no. 02, pp. 173–193, 2011.

[41] K. Hiraki, K. Nishida, S. Sekiguchi, T. Shimada, and T. Yuba, "The sigima-1 dataflow supercomputer: A challenge for new generation supercomputing systems," *Journal of information processing*, vol. 10, no. 4, pp. 219–226, 1988.

[42] Y. Kodama, S. Sakai, and Y. Yamaguchi, "A prototype of a highly parallel dataflow machine em-4 and its preliminary evaluation," *Future Generation Computer Systems*, vol. 7, no. 2, pp. 199–209, 1992.

[43] G. M. Papadopoulos and D. E. Culler, "Monsoon: an explicit token-store architecture," in *ACM SIGARCH Computer Architecture News*, vol. 18, no. 2SI. ACM, 1990, pp. 82–91.

[44] V. G. Grafe and J. E. Hoch, "The epsilon-2 multiprocessor system," *Journal of Parallel and Distributed Computing*, vol. 10, no. 4, pp. 309–318, 1990.

[45] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore, "Exploiting ilp, tlp, and dlp with the polymorphous trips architecture," in *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*. IEEE, 2003, pp. 422–433.

[46] F. Yazdanpanah, D. Jimenez-Gonzalez, C. Alvarez-Martinez, Y. Etsion, and R. M. Badia, "Fpga-based prototype of the task superscalar architecture," in *7th HiPEAC Workshop on Reconfigurable Computing (WRC 2013), Berlin, Germany*, 2013.

GEORGE MATHEOU

[47] O. Pell and V. Averbukh, "Maximum performance computing with dataflow engines," *Computing in Science & Engineering*, vol. 14, no. 4, pp. 98–103, 2012.

[48] J. Benson, R. Cofell, C. Frericks, C.-H. Ho, V. Govindaraju, T. Nowatzki, and K. Sankaralingam, "Design, integration and implementation of the dyser hardware accelerator into opensparc," in *IEEE International Symposium on High-Performance Comp Architecture*. IEEE, 2012, pp. 1–12.

[49] D. Capalija and T. S. Abdelrahman, "Microarchitecture of a coarse-grain out-of-order superscalar processor," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 2, pp. 392–405, 2013.

[50] C. Wang, X. Li, J. Zhang, P. Chen, Y. Chen, X. Zhou, and R. C. Cheung, "Architecture support for task out-of-order execution in mpsocs," *IEEE Transactions on Computers*, vol. 64, no. 5, pp. 1296–1310, 2015.

[51] A. K. Jain, X. Li, S. A. Fahmy, and D. L. Maskell, "Adapting the dyser architecture with dsp blocks as an overlay for the xilinx zynq," *ACM SIGARCH Computer Architecture News*, vol. 43, no. 4, pp. 28–33, 2016.

[52] H. H. J. Hum, O. Maquelin, K. B. Theobald, X. Tian, X. Tang, G. R. Gao, P. Cupryk, N. Elmasri, L. J. Hendren, A. Jimenez, S. Krishnan, A. Marquez, S. Merali, S. S. Nemawarkar, P. Panangaden, X. Xue, and Y. Zhu, "A design study of the earth multiprocessor," in *Proceedings of the IFIP WG10.3 Working Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '95. Manchester, UK, UK: IFIP Working Group on Algol, 1995, pp. 59–68. [Online]. Available: http://dl.acm.org/citation.cfm?id=224659.224685

[53] K. M. Kavi, R. Giorgi, and J. Arul, "Scheduled dataflow: Execution paradigm, architecture, and performance evaluation," *Computers, IEEE Transactions on*, vol. 50, no. 8, pp. 834–846, 2001.

[54] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin, "Wavescalar," in *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2003, p. 291.

[55] C. Kyriacou, P. Evripidou, and P. Trancoso, "Data-driven multithreading using conventional microprocessors," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 17, no. 10, pp. 1176–1188, 2006.

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

[57] A. Mondelli, N. Ho, A. Scionti, M. Solinas, A. Portero, and R. Giorgi, "Dataflow support in x86_64 multicore architectures through small hardware extensions," in *Digital System Design (DSD), 2015 Euromicro Conference on*. IEEE, 2015, pp. 526–529.

[58] P. Evripidou, "Thread synchronization unit (tsu): A building block for high performance computers," in *High Performance Computing*. Springer, 1997, pp. 107–118.

[59] K. Sankaralingam, R. Nagarajan, R. McDonald, R. Desikan, S. Drolia, M. Govindan, P. Gratzf, D. Gulati, H. Hanson, C. Kim *et al.*, "Distributed microarchitectural protocols in the trips prototype processor," in *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*. IEEE, 2006, pp. 480–491.

[60] S. Swanson, A. Schwerin, M. Mercaldi, A. Petersen, A. Putnam, K. Michelson, M. Oskin, and S. J. Eggers, "The wavescalar architecture," *ACM Transactions on Computer Systems (TOCS)*, vol. 25, no. 2, p. 4, 2007.

GEORGE MATHEOU

[61] Xilinx, "Microblaze processor reference guide," *reference manual*, vol. 23, 2006.

[62] xilinx.com, "All programmable technologies from xilinx inc." 2017. [Online]. Available: http://www.xilinx.com/

[63] G. Michael, S. Arandi, and P. Evripidou, "Data-flow concurrency on distributed multi-core systems," in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2013, p. 515.

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

[65] P. G. Harrison and M. J. Reeve, "The parallel graph reduction machine, alice," in *Graph Reduction*. Springer, 1987, pp. 181–202.

[66] P. Watson and I. Watson, "Evaluating functional programs on the flagship machine," in *Functional Programming Languages and Computer Architecture*. Springer, 1987, pp. 80–97.

[67] I. Watson, V. Woods, P. Watson, R. Banach, M. Greenberg, and J. Sargeant, "Flagship: a parallel architecture for declarative programming," in *ACM SIGARCH Computer Architecture News*, vol. 16, no. 2. IEEE Computer Society Press, 1988, pp. 124–130.

[68] J. B. Dennis and D. P. Misunas, "A preliminary architecture for a basic data-flow processor," *SIGARCH Comput. Archit. News*, vol. 3, no. 4, pp. 126–132, Dec. 1974. [Online]. Available: http://doi.acm.org/10.1145/641675.642111

[69] D. W. Wall, "Limits of instruction-level parallelism," in *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS IV. New York, NY, USA: ACM, 1991, pp. 176–188. [Online]. Available: http://doi.acm.org/10.1145/106972.106991

[70] W. A. Wulf and S. A. McKee, "Hitting the memory wall: implications of the obvious," *ACM SIGARCH computer architecture news*, vol. 23, no. 1, pp. 20–24, 1995.

[71] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, and Others, "Introduction to the cell multiprocessor," *IBM journal of Research and Development*, vol. 49, no. 4/5, p. 589, 2005.

[72] A. Olofsson, T. Nordström, and Z. Ul-Abdin, "Kickstarting high-performance energy-efficient manycore architectures with epiphany," in *2014 48th Asilomar Conference on Signals, Systems and Computers*. IEEE, 2014, pp. 1719–1726.

[73] S. P. Crago, D.-I. Kang, M. Kang, R. Kost, K. Singh, J. Suh, and J. P. Walters, "Programming models and development software for a space-based many-core processor," in *Space Mission Challenges for Information Technology (SMC-IT), 2011 IEEE Fourth International Conference on*. IEEE, 2011, pp. 95–102.

[74] T. G. Mattson, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl *et al.*, "The 48-core scc processor: the programmer's view," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society, 2010, pp. 1–11.

[75] B. D. de Dinechin, P. G. de Massas, G. Lager, C. Léger, B. Orgogozo, J. Reybert, and T. Strudel, "A distributed run-time environment for the kalray mppa-256 integrated manycore processor," *Procedia Computer Science*, vol. 18, pp. 1654–1663, 2013.

[76] J. Jeffers and J. Reinders, *Intel Xeon Phi coprocessor high-performance programming*. Newnes, 2013.

GEORGE MATHEOU

[77] Xilinx, "Field programmable gate array (fpga)." [Online]. Available: http://www.xilinx.com/training/fpga/fpga-field-programmable-gate-array.htm

[78] S. D. Automation, "VHDL Reference Manual," *Rev. March*, pp. 9–12, 1997.

[79] M. McNamara *et al.*, "IEEE Standard Verilog Hardware Description Language. The Institute of Electrical and Electronics Engineers," *Inc. IEEE Std*, pp. 1364–2001, 2001.

[80] P. P. Chu, *FPGA prototyping by Verilog examples: Xilinx Spartan-3 version*. John Wiley & Sons, 2011.

[81] R. J. Francis, "Technology mapping for lookup-table based field-programmable gate arrays," Ph.D. dissertation, Citeseer, 1993.

[82] P. K. Gupta, "Xeon+fpga platform for the data center," in *Fourth Workshop on the Intersections of Computer Architecture and Reconfigurable Logic*, vol. 119, 2015.

[83] W. Ackermann and J. Dennis, "Val: A value-oriented algorithmic language," in *Tech. Report LCS/TR-218*. Massachusetts Inst. of Technology Cambridge, 1979.

[84] R. S. Nikhil, P. Fenstermacher, J. Hicks, and R. Johnson, "Id world reference manual," *CSG Memo, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA (April 1987)*, 1987.

[85] J. McGraw, S. Skedzielewski, S. Allan, D. Grit, R. Oldehoeft, J. Glauert, I. Dobes, and P. Hohensee, "Sisal: streams and iteration in a single-assignment language. language reference manual, version 1.1," Lawrence Livermore National Lab., CA (USA), Tech. Rep., 1983.

[86] M. Amamiya, R. Hasegawa, and S. Ono, "Valid, a high-level functional programming language for data flow machines," *Review of the Electrical Communication Laboratories*, vol. 32, no. 5, pp. 793–802, 1984.

[87] S. S. Thakkar, "Selected reprints on dataflow and reduction architectures," 1987.

[88] A. R. Hurson and K. M. Kavi, "Dataflow computers: Their history and future," *Wiley Encyclopedia of Computer Science and Engineering*, 2008.

[89] Arvind and D. E. Culler, "Dataflow architectures." Annual Reviews Inc., 1986, pp. 225–253.

[90] J. B. Dennis, "Data flow supercomputers," *Computer*, no. 11, pp. 48–56, 1980.

[91] A. L. Davis, "The architecture and system method of ddm1: A recursively structured data driven machine," in *Proceedings of the 5th Annual Symposium on Computer Architecture*, ser. ISCA '78. New York, NY, USA: ACM, 1978, pp. 210–215. [Online]. Available: http://doi.acm.org/10.1145/800094.803050

[92] A. Plas, D. Comte, O. Gelly, and J. Syre, "Lau system architecture: A parallel data driven processor based on single assignment," in *Proceedings of the International Conference on Parallel Processing*, 1976, pp. 293–302.

[93] M. Cornish, "The ti data flow architectures- the power of concurrency for avionics," *Challenge of the'80 s*, pp. 19–25, 1979.

[94] I. Watson and J. Gurd, "A prototype data flow computer with token labelling," in *Proceedings of the National Computer Conference*, vol. 1979, 1979, pp. 623–628.

[95] Arvind and V. Kathail, "A multiple processor data flow machine that supports generalized procedures," in *Proceedings of the 8th Annual Symposium on Computer Architecture*, ser. ISCA '81. Los Alamitos, CA, USA: IEEE Computer Society Press, 1981, pp. 291–302. [Online]. Available: http://dl.acm.org/citation.cfm?id=800052.801882

GEORGE MASON UNIVERSITY

[96] Arvind and R. E. Thomas, *I-Structures: An efficient data type for functional languages*. Laboratory for Computer Science, Massachusetts Institute of Techcnology, 1981.

[97] Arvind, R. S. Nikhil, and K. K. Pingali, "I-structures: Data structures for parallel computing," *ACM Trans. Program. Lang. Syst.*, vol. 11, no. 4, pp. 598–632, Oct. 1989. [Online]. Available: http://doi.acm.org/10.1145/69558.69562

[98] T. Shimada, K. Hiraki, K. Nishida, and S. Sekiguchi, "Evaluation of a prototype data flow processor of the sigma-1 for scientific computations," in *Proceedings of the 13th Annual International Symposium on Computer Architecture*, ser. ISCA '86. Los Alamitos, CA, USA: IEEE Computer Society Press, 1986, pp. 226–234. [Online]. Available: http://dl.acm.org/citation.cfm?id=17407.17383

[99] L. M. Patnaik, R. Govindarajan, and N. Ramadoss, "Design and performance evaluation of exman: An extended manchester data flow computer," *Computers, IEEE Transactions on*, vol. 100, no. 3, pp. 229–244, 1986.

[100] B. Lee, A. R. Hurson, and B. Shirazi, "A hybrid scheme for processing data structures in a dataflow environment," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 3, no. 1, pp. 83–96, 1992.

[101] D. E. Culler and G. M. Papadopoulos, "The explicit token store," *Journal of Parallel and Distributed Computing*, vol. 10, no. 4, pp. 289–308, 1990.

[102] A. Hurson and B. Lee, "Issues in dataflow computing," *Adv. in Comput*, vol. 37, no. 285-333, pp. 38–39, 1993.

[103] K. M. Kavi and B. Shirazi, "Dataflow architecture: Are dataflow computers commercially viable?" *IEEE Potentials*, pp. 27–30, 1992.

[104] G. M. Papadopoulos and K. R. Traub, "Multithreading: A revisionist view of dataflow architectures," in *Proceedings of the 18th Annual International Symposium on Computer Architecture*, ser. ISCA '91. New York, NY, USA: ACM, 1991, pp. 342–351. [Online]. Available: http://doi.acm.org/10.1145/115952.115986

[105] F. Yazdanpanah, C. Alvarez-Martinez, D. Jimenez-Gonzalez, and Y. Etsion, "Hybrid dataflow/von-neumann architectures," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 25, no. 6, pp. 1489–1509, 2014.

[106] J. Silc, B. Robic, and T. Ungerer, "Asynchrony in parallel computing: From dataflow to multithreading," *Parallel and Distributed Computing Practices*, vol. 1, no. 1, pp. 3–30, 1998.

[107] B. Robic, J. Silc, and T. Ungerer, "Beyond dataflow," *Journal of Computing and Information Technology*, vol. 8, no. 2, pp. 89–102, 2000.

[108] R. A. Iannucci, *Toward a dataflow/von Neumann hybrid architecture*. IEEE Computer Society Press, 1988, vol. 16, no. 2.

[109] S. Sakai, K. Hiraki, Y. Kodama, T. Yuba *et al.*, "An architecture of a dataflow single chip processor," in *ACM SIGARCH Computer Architecture News*, vol. 17, no. 3. ACM, 1989, pp. 46–53.

[110] M. Sato, Y. Kodama, S. Sakai, Y. Yamaguchi, and Y. Koumura, "Thread-based programming for the em-4 hybrid dataflow machine," in *ACM SIGARCH Computer Architecture News*, vol. 20, no. 2. ACM, 1992, pp. 146–155.

[111] P. Evripidou and J.-L. Gaudiot, "A decoupled graph/computation data-driven architecture with variable-resolution actors," University of Southern California, Los Angeles, CA (United States). Dept. of Electrical Engineering, Tech. Rep., 1990.

GEORGE MATHEOU

[112] ——, "A decoupled data-driven architecture with vectors and macro actors," in *CONPAR 90—VAPP IV*. Springer, 1990, pp. 39–50.

[113] ——, "The usc decoupled multilevel data-flow execution model," *Advanced topics in data-flow computing*, pp. 347–379, 1991.

[114] G. R. Gao, "An efficient hybrid dataflow architecture model," *Journal of Parallel and Distributed Computing*, vol. 19, no. 4, pp. 293–307, 1993.

[115] R. S. Nikhil, "Can dataflow subsume von neumann computing?" in *Proceedings of the 16th Annual International Symposium on Computer Architecture*, ser. ISCA '89. New York, NY, USA: ACM, 1989, pp. 262–272. [Online]. Available: http://doi.acm.org/10.1145/74925.74955

[116] R. S. Nikhil, G. M. Papadopoulos, and Arvind, "*t: A multithreaded massively parallel architecture," *SIGARCH Comput. Archit. News*, vol. 20, no. 2, pp. 156–167, Apr. 1992. [Online]. Available: http://doi.acm.org/10.1145/146628.139715

[117] S. S. Nemawarkar and G. R. Gao, "Measurement and modeling of earth-manna multithreaded architecture," in *Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 1996. MASCOTS'96., Proceedings of the Fourth International Workshop on*. IEEE, 1996, pp. 109–114.

[118] W. Zhu, Y. Niu, and G. R. Gao, "Performance portability on earth: a case study across several parallel architectures," *Cluster Computing*, vol. 10, no. 2, pp. 115–126, 2007.

[119] D. E. Culler, A. Sah, K. E. Schauser, T. von Eicken, and J. Wawrzynek, "Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine," in *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS IV. New York, NY, USA: ACM, 1991, pp. 164–175. [Online]. Available: http://doi.acm.org/10.1145/106972.106990

[120] D. E. Culler, S. C. Goldstein, K. E. Schauser, and T. Voneicken, "Tam-a compiler controlled threaded abstract machine," *Journal of Parallel and Distributed Computing*, vol. 18, no. 3, pp. 347–370, 1993.

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

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

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

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

[125] J. M. Pérez, P. Bellens, R. M. Badia, and J. Labarta, "Cellss: Making it easier to program the cell broadband engine processor," *IBM Journal of Research and Development*, vol. 51, no. 5, pp. 593–604, 2007.

[126] E. Ayguadé, R. M. Badia, F. D. Igual, J. Labarta, R. Mayo, and E. S. Quintana-Ortí, "An extension of the starss programming model for platforms with multiple gpus," in *Euro-Par 2009 Parallel Processing*. Springer, 2009, pp. 851–862.

[127] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta, "Hierarchical task-based programming with starss," *International Journal of High Performance Computing Applications*, vol. 23, no. 3, pp. 284–299, 2009.

GEORGE MASON

[128] V. K. Elangovan, R. M. Badia, and E. A. Parra, "Ompss-opencl programming model for heterogeneous systems," in *Languages and compilers for parallel computing*. Springer, 2013, pp. 96–111.

[129] J. Bueno, L. Martinell, A. Duran, M. Farreras, X. Martorell, R. M. Badia, E. Ayguade, and J. Labarta, "Productive cluster programming with ompss," in *European Conference on Parallel Processing*. Springer, 2011, pp. 555–566.

[130] J. Bueno, X. Martorell, R. M. Badia, E. Ayguadé, and J. Labarta, "Implementing ompss support for regions of data in architectures with multiple address spaces," in *Proceedings of the 27th international ACM conference on International conference on supercomputing*. ACM, 2013, pp. 359–368.

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

[132] J. M. Arul and K. M. Kavi, "Scalability of scheduled data flow architecture (sdf) with register contexts," in *Algorithms and Architectures for Parallel Processing, 2002. Proceedings. Fifth International Conference on*. IEEE, 2002, pp. 214–221.

[133] F. YAZDANPANAH and C. ALVAREZ-MARTINEZ, "Supplementary file of" hybrid dataflow/von-neumann architectures."

[134] D. Burger, S. W. Keckler, K. S. McKinley, M. Dahlin, L. K. John, C. Lin, C. R. Moore, J. Burrill, R. G. McDonald, and W. Yoder, "Scaling to the end of silicon with edge architectures," *Computer*, vol. 37, no. 7, pp. 44–55, 2004.

[135] S. Swanson, A. Putnam, M. Mercaldi, K. Michelson, A. Petersen, A. Schwerin, M. Oskin, and S. J. Eggers, "Area-performance trade-offs in tiled dataflow architectures," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 2, pp. 314–326, 2006.

[136] C. Kyriacou and P. Evripidou, "Communication assist for data driven multithreading," in *Advances in Informatics*. Springer, 2001, pp. 351–367.

[137] P. Evripidou, "D3-machine: a decoupled data-driven multithreaded architecture with variable resolution support," *Parallel Computing*, vol. 27, no. 9, pp. 1197–1225, 2001.

[138] C. Kyriacou and P. Evripidou, "Network interface for a data driven network of workstations (d2now)," in *High Performance Computing*. Springer, 1999, pp. 257–268.

[139] P. Evripidou and C. Kyriacou, "Data driven network of workstations (d2now)." *J. UCS*, vol. 6, no. 10, pp. 1015–1033, 2000.

[140] I. Watson and et al, "A prototype data flow computer with token labelling," in *Managing Requirements Knowledge, International Workshop on*. IEEE Computer Society, 1989, pp. 623–623.

[141] G. Matheou and P. Evripidou, "Architectural support for data-driven execution," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 11, no. 4, pp. 52:1–52:25, Jan. 2015. [Online]. Available: http://doi.acm.org/10.1145/2686874

[142] ——, "FREDDO: an efficient framework for runtime execution of data-driven objects," in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, 2016, pp. 265–273.

[143] C. Kyriacou, P. Evripidou, and P. Trancoso, "Cacheflow: A short-term optimal cache management policy for data driven multithreading," in *Euro-Par 2004 Parallel Processing*. Springer, 2004, pp. 561–570.

GEORGE MATHEOU

[144] P. Trancoso, P. Evripidou, K. Stavrou, and C. Kyriacou, "A case for chip multiprocessors based on the data-driven multithreading model," *International Journal of Parallel Programming*, vol. 34, no. 3, pp. 213–235, 2006.

[145] P. Trancoso, K. Stavrou, and P. Evripidou, "Ddmcpp: The data-driven multithreading c preprocessor," *Proceedings of the 11th Interact-11*, pp. 32–39, 2007.

[146] A. Diavastos, G. Stylianou, and P. Trancoso, "Tfluxscc: Exploiting performance on future many-core systems through data-flow," in *Parallel, Distributed and Network-Based Processing (PDP), 2015 23rd Euromicro International Conference on*. IEEE, 2015, pp. 190–198.

[147] "Simics." [Online]. Available: https://en.wikipedia.org/wiki/Simics

[148] A. Diavastos, P. Trancoso, M. Luján, and I. Watson, "Integrating transactions into the data-driven multi-threading model using the tflux platform," in *Data-Flow Execution Models for Extreme Scale Computing (DFM), 2011 First Workshop on*. IEEE, 2011, pp. 19–27.

[149] ——, "Integrating transactions into the data-driven multi-threading model using the tflux platform," *International Journal of Parallel Programming*, pp. 1–21, 2015. [Online]. Available: http://dx.doi.org/10.1007/s10766-015-0369-2

[150] P. Felber, C. Fetzer, and T. Riegel, "Dynamic performance tuning of word-based software transactional memory," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '08. New York, NY, USA: ACM, 2008, pp. 237–246. [Online]. Available: http://doi.acm.org/10.1145/1345206.1345241

[151] R. Giorgi, R. M. Badia, F. Bodin, A. Cohen, P. Evripidou, P. Faraboschi, B. Fechner, G. R. Gao, A. Garbade, R. Gayatri *et al.*, "Teraflux: Harnessing dataflow in next generation teradevices," *Microprocessors and Microsystems*, vol. 38, no. 8, pp. 976–990, 2014.

[152] S. Palnitkar, *Verilog HDL: a guide to digital design and synthesis*. Prentice Hall Professional, 2003, vol. 1.

[153] xilinx.com, "Fsl v20," 2014. [Online]. Available: http://www.xilinx.com/support/documentation/ipembedprocess_processorinterface_fsl.htm

[154] I. Xilinx, "Logicore ip axi interconnect (v1.06.a)," December 2012. [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/axi_interconnect/v1_06_a/ds768_axi_interconnect.pdf

[155] T. Feist, "Vivado design suite," *White Paper*, vol. 5, 2012.

[156] G. Matheou and P. Evripidou, "Freddo: an efficient framework for runtime execution of data-driven objects," Department of Computer Science, University of Cyprus, Nicosia, Cyprus, Tech. Rep. TR-16-1, January 2016. [Online]. Available: https://www.cs.ucy.ac.cy/docs/techreports/TR-16-1.pdf

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

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

[159] G. Matheou and P. Evripidou, "Verilog-based simulation of hardware support for data-flow concurrency on multicore systems," in *SAMOS XIII, 2013*. IEEE, 2013, pp. 280–287.

[160] B. Eckel, *Thinking in JAVA*. Prentice Hall Professional, 2003.

[161] TutorialsPoint, "Data encapsulation in c++," 2016. [Online]. Available: http://www.tutorialspoint.com/cplusplus/cpp_data_encapsulation.htm

GEORGE MATHEOU

[162] J. Protic, M. Tomasevic, and V. Milutinović, *Distributed shared memory: Concepts and systems*. John Wiley & Sons, 1998, vol. 21.

[163] D. A. Koufaty, X. Chen, D. K. Poulsen, and J. Torrellas, "Data forwarding in scalable shared-memory multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, no. 12, pp. 1250–1264, 1996.

[164] D. K. Poulsen and P.-C. Y. P.-C. Yew, "Data prefetching and data forwarding in shared memory multiprocessors," in *Parallel Processing, 1994. ICPP 1994 Volume 2. International Conference on*, vol. 2. IEEE, 1994, pp. 280–280.

[165] J. Matocha and T. Camp, "A taxonomy of distributed termination detection algorithms," *Journal of Systems and Software*, vol. 43, no. 3, pp. 207–221, 1998.

[166] E. W. Dijkstra and C. S. Scholten, "Termination detection for diffusing computations," *Information Processing Letters*, vol. 11, no. 1, pp. 1–4, 1980.

[167] R. Stevens, A. White, S. Dosanjh, A. Geist, B. Gorda, K. Yelick, J. Morrison, H. Simon, J. Shalf, J. Nichols *et al.*, "Architectures and technology for extreme scale computing," in *ASCR Scientific Grand Challenges Workshop Series, Tech. Rep*, 2009.

[168] R. Rosner *et al.*, "The opportunities and challenges of exascale computing," *US Dept. of Energy Office of Science, Summary Report of the Advanced Scientific Computing Advisory Committee (ASCAC) Subcommittee*, 2010.

[169] Z. Budimlic, A. M. Chandramowlishwaran, K. Knobe, G. N. Lowney, V. Sarkar, and L. Treggiari, "Declarative aspects of memory management in the concurrent collections parallel programming model," in *Proceedings of the 4th workshop on Declarative aspects of multicore programming*. ACM, 2009, pp. 47–58.

[170] A. Diavastos, G. Matheou, P. Evripidou, and P. Trancoso, "Data-driven multithreading programming tool-chain," Department of Computer Science, University of Cyprus, Nicosia, Cyprus, Tech. Rep. TR-17-3, September 2017. [Online]. Available: https://www.cs.ucy.ac.cy/docs/techreports/TR-17-3.pdf

[171] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, "LU decomposition and its applications," *Numerical Recipes in FORTRAN: The Art of Scientific Computing*, pp. 34–42, 1992.

[172] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," in *ACM SIGARCH Computer Architecture News*, vol. 23, no. 2. ACM, 1995, pp. 24–36.

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

[174] "Mandelbrot set," 2017. [Online]. Available: https://en.wikipedia.org/wiki/Mandelbrot_set#cite_note-John_H._Hubbard_1985-1

[175] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, J. J. Dongarra, J. Du Croz, S. Hammarling, A. Greenbaum, A. McKenney, and D. Sorensen, *LAPACK Users' Guide (Third Ed.)*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1999.

[176] L. S. Blackford, A. Petitet, R. Pozo, K. Remington, R. C. Whaley, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry *et al.*, "An updated set of basic linear algebra subprograms (blas)," *ACM Transactions on Mathematical Software*, vol. 28, no. 2, pp. 135–151, 2002.

GEORGE MATHEOU

[177] F. Song and J. Dongarra, "Scaling up matrix computations on shared-memory manycore systems with 1000 cpu cores," in *Proceedings of the 28th ACM international conference on Supercomputing*. ACM, 2014, pp. 333–342.

[178] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov, "Numerical linear algebra on emerging architectures: The plasma and magma projects," in *Journal of Physics: Conference Series*, vol. 180, no. 1. IOP Publishing, 2009, p. 012037.

[179] Eight queens puzzle. Accessed on 10 Aug 2016. [Online]. Available: https://en.wikipedia.org/wiki/Eight_queens_puzzle

[180] BSC. BSC Application Repository. Accessed on 10 Aug 2016. [Online]. Available: https://pm.bsc.es/projects/bar/wiki/Applications

[181] T. C. Institute, "Cy-Tera," http://web.cytera.cyi.ac.cy, 2017, [Online; accessed 25-Mar-2017].

[182] Xilinx, *ISim User Guide (UG660)*, 2012.

[183] K. Pagiamtzis and A. Sheikholeslami, "Content-addressable memory (cam) circuits and architectures: A tutorial and survey," *IEEE Journal of Solid-State Circuits*, vol. 41, no. 3, pp. 712–727, 2006.

[184] P. R. Panda, N. D. Dutt, and A. Nicolau, "On-chip vs. off-chip memory: the data partitioning problem in embedded processor-based systems," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 5, no. 3, pp. 682–704, 2000.

[185] L. S. Blackford, J. Choi, A. Cleary, E. D'Azeuedo, J. Demmel, I. Dhillon, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, *ScaLAPACK User's Guide*, J. J. Dongarra, Ed. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1997.

[186] mpich.org, "LU factorization," https://trac.mpich.org/projects/armci-mpi/browser/tests/contrib/lu/lu.c, 2017, [Online; accessed 09-Oct-2017].

[187] M. C. Cera, J. V. Lima, N. Maillard, and P. O. A. Navaux, "Challenges and issues of supporting task parallelism in mpi," in *EuroMPI*. Springer, 2010, pp. 302–305.

[188] D. Bonachea, "Gasnet specification, v1.1," EECS Department, University of California, Berkeley, Tech. Rep. UCB/CSD-02-1207, Oct 2002. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2002/5764.html

[189] U. Lamping and E. Warnicke, "Wireshark user's guide," *Interface*, vol. 4, no. 6, 2004.

[190] A. Forencich, "Verilog content addressable memory module," 2016. [Online]. Available: https://github.com/alexforencich/verilog-cam

[191] H. Fu, J. Liao, J. Yang, L. Wang, Z. Song, X. Huang, C. Yang, W. Xue, F. Liu, F. Qiao *et al.*, "The sunway taihulight supercomputer: system and applications," *Science China Information Sciences*, vol. 59, no. 7, p. 072001, 2016.

[192] K. Fürlinger, T. Fuchs, and R. Kowalewski, "Dash: a c++ pgas library for distributed data structures and parallel algorithms," in *High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS), 2016 IEEE 18th International Conference on*. IEEE, 2016, pp. 983–990.

GEORGE MATHEOU

Appendices

GEORGE MATHEOU

Appendix A

Publications

Journals

1. **G. Matheou** and P. Evripidou. "Architectural support for data-driven execution". ACM Transactions on Architecture and Code Optimization (TACO) 11.4 (2015): 52. Presented in HiPEAC 2015, Amsterdam, January 2015. DOI: 10.1145/2686874.
2. S. Arandi, **G. Matheou**, C. Kyriacou, and P. Evripidou. "Data-Driven Thread Execution on Heterogeneous Processors." International Journal of Parallel Programming, February 8, 2017. DOI: 10.1007/s10766-016-0486-6.
3. **G. Matheou** and P. Evripidou. "Data-Driven Concurrency for High Performance Computing." ACM Transactions on Architecture and Code Optimization (TACO) 14.4 (2017): 53. DOI: 10.1145/3162014

Conferences

1. **G. Matheou** and P. Evripidou. "Verilog-based simulation of hardware support for data-flow concurrency on multicore systems." Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII), 2013 International Conference on. IEEE, 2013. DOI: 10.1109/SAMOS.2013.6621136.
2. **G. Matheou**, P. Evripidou, and C. Kyriacou. "Paradigm Shift for EXASCALE Computing." In Proceedings of the 3rd International Conference on Exascale Applications and Software (EASC 2015), A. Gray, L. Smith, and M. Weiland (Eds.). University of Edinburgh, Edinburgh, Scotland, UK, 109-114, 2015. ISBN: 978-0-9926615-1-9.
3. **G. Matheou** and P. Evripidou, "FREDDO: an efficient Framework for Runtime Execution of Data-Driven Objects," Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA), 2016. ISBN: 1-60132-444-8.

Workshops

1. **G. Matheou**, I. Watson, and P. Evripidou, "Recursion support for the data-driven multithreading model," Fifth Workshop on Data-Flow Execution Models for Extreme Scale Computing (DFM 2015), in conjunction with PACT 2015, San Francisco, October 2015.
2. **G. Matheou**, C. Kyriacou, and P. Evripidou, "Data-Driven execution of the Tile LU Decomposition," Sixth Workshop on Data-Flow Execution Models for Extreme Scale Computing (DFM 2016), in conjunction with PACT 2016, Haifa, September 2016.

GEORGE MATHEOU

Technical Reports

1. **G. Matheou** and P. Evripidou, "FREDDO: an efficient Framework for Runtime Execution of Data-Driven Objects," Department of Computer Science, University of Cyprus, Nicosia, Cyprus, Tech. Rep. TR-16-1, January 2016. [Online]. Available: www.cs.ucy.ac.cy/docs/techreports/TR-16-1.pdf
2. **G. Matheou** and P. Evripidou, "Data-Driven Concurrency for High Performance Computing," Department of Computer Science, University of Cyprus, Nicosia, Cyprus, Tech. Rep. TR-17-1, May 2017. [Online]. Available: www.cs.ucy.ac.cy/docs/techreports/TR-17-1.pdf
3. A. Diavastos, **G. Matheou**, P. Evripidou and P. Trancoso, "Data-Driven Multithreading Programming Tool-chain," Department of Computer Science, University of Cyprus, Nicosia, Cyprus, Tech. Rep. TR-17-3, September 2017. [Online]. Available: www.cs.ucy.ac.cy/docs/techreports/TR-17-3.pdf

GEORGE MATHEOU