SEVENTH FRAMEWORK PROGRAMME
THEME
FET proactive 1: Concurrent Tera-Device Computing (ICT-2009.8.1)

PROJECT NUMBER: 249013

Exploiting dataflow parallelism in Teradevice Computing

D6.4 – Evaluation of the TERAFLUX Abstraction Layer and Fine-tuned Model

Due date of deliverable: 31st March 2014
Actual Submission: 19th May 2014

Start date of the project: January 1st, 2010 Duration: 51 months

Lead contractor for the deliverable: UCY

Revision: See file name in document footer.

Project co-founded by the European Commission within the SEVENTH FRAMEWORK PROGRAMME (2007-2013)	
Dissemination Level: PU	
PU	Public
PP	Restricted to other programs participant (including the Commission Services)
RE	Restricted to a group specified by the consortium (including the Commission Services)
CO	Confidential, only for members of the consortium (including the Commission Services)

Change Control

Version#	**Date**	**Author**	**Organization**	**Change History**
1	**10.03.2014**	**Rosa M. Badia Daniel Jiménez Carlos Álvarez**	**BSC**	**First update**
2	**20.03.2014**	**Alberto Scionti, Bruce Jacob, Roberto Giorgi**	**UNISI**	**Final UNISI contribution**
7	**10.05.2014**	**Skevos Evripidou, Pedro Trancoso**	**UCY**	**Last fixes**
8	**11.05.2014**	**Roberto Giorgi**	**UNISI**	**review**

Release Approval

Name	**Role**	**Date**
Skevos Evripidou	**Originator**	**10.03.2014**
Skevos Evripidou	**WP Leader**	**10.05.2014**
Roberto Giorgi	**Project Coordinator for formal deliverable**	**11.05.2014**

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

List of contributors to the writing of the document.

Rosa M. Badia, Daniel Jiménez, Carlos Álvarez
BSC

Arne Garbade, Sebastian Weis, Theo Ungerer
UAU

Andreas Diavastos, George Matheou, Pedro Trancoso, Paraskevas Evripidou
UCY

Behram Khan, Salman Khan, William B Toms, Mikel Lujan, Ian Watson, Mikel Lujan
UNIMAN

Roberto Giorgi, Alberto Scionti, Bruce Jacob
UNISI

Glossary

Cluster	Group of cores (synonymous of NODE)
DDM	Data-Driven Multithreading
DF-Thread	A TERAFLUX Data-Flow Thread
DF-Frame	The Frame memory associated to a Data-Flow thread
DTA	Decoupled Threaded Architecture
DTS	Distributed Thread Scheduler (the whole set of D-TSUs and L-TSUs)
D-FDU	Distributed Fault Detection Unit (per-node FDU, also L2-FDU)
D-TSU	Distributed Thread Scheduling Unit (per-node TSU, also L2-TSU)
Emulator	Tool capable of reproducing the Functional Behavior; synonymous in this context of Instruction Set Simulator (ISS)
FPGA	Field-Programmable Gate Array – reconfigurable hardware
HTSS	Hardware TaskSuperScalar initial hardware implementation based on the original design [3] used as a base to perform a hardware space design exploration.
L-Thread	Legacy Thread: a thread consisting of legacy code
L-FDU	Local Fault Detection Unit (per-core FDU, also L1-FDU)
L-TSU	Local Thread Scheduling Unit (per-core TSU, also L1-TSU, or LSU)
MMS	Memory Model Support
NIU	Network Interface Unit
NoC	Network on Chip
Non-DF-Thread	An L-Thread or S-Thread
Node	Group of cores (synonymous of Cluster)
OWM	Owner Writeable Memory
OS	Operating System
Per-Node-Manager	A hardware unit including the TSU and the FDU
Picos	Final Hardware proposed implementation of the TaskSuperScalar. It improves the original design in performance, uses fewer resources and better supports the OmpSs programming model.
PhyGAS	Physical Global Address Space
SCC	Intel 48-core experimental research processor
Sharable-Memory	Memory that respects the FM,OWM,TM semantics of the TERAFLUX Memory Model
SimTSS	Cycle-Accurate Software Simulator of the Picos Hardware implementation tuned with the latencies obtained from the hardware implementation of HTSS.
S-Thread	System Thread: a thread dealing with OS services or I/O
StarSs	A programming model introduced by Barcelona Supercomputing Center
Simulator	Emulator that includes timing information; synonymous in this context of "Timing Simulator"
TFlux	Thread Flux DDM-Style runtime implementation
TLS	Thread Local Storage
TM	Transactional Memory
TMS	Transactional Memory Support

TP	Threaded Procedure
UAS	Unified Address Space (synonymous of PhyGAS, Physical Global Address Space)
Virtualizer	Synonymous of “Emulator”
VCPU	Virtual CPU or Virtual Core

Executive Summary

This document describes the work that was performed during the fourth year (m37-51) of the TERAFLUX project within the context of Task 6.4 (m25-51) "Fine-Tuned Execution Model" and Task 6.5 (m25-51) "Abstraction Layer". Since both tasks extend for the last 27 months of the project we also include a short summary of the work performed in the third year (m25-M36) for these tasks.

The work in Task 6.4 was a collaboration work by the different partners involved and thus we report here the contribution of each partner.

- UCY implemented and tested the execution of a DDM-Style application on an FPGA-based system (year 4), a single-node many-core system (year 4), and a multi-node multi-core system (year 3 and 4)
- UCY implemented and tested the dynamic scheduling of DThreads for DDM-Style applications (year 4)
- UNISI proposed the advanced memory management of TERAFLUX (year 4)
- BSC presented the evolution of the HTSS implementation (year 3 and 4)
- UCY developed and evaluated a runtime dependency resolution mechanism for the DDM-Style of the TERAFLUX execution model (year 3)
- UNISI designed and evaluated a multi-node TSU working thanks to the implementation of the T* ISE (year 3)
- UNIMAN showed the TM support for the TERAFLUX architecture (year 3)
- UCY presented a program analysis tool based on PARAVER (year 3)
- HP extended the TSU design (TSUF) for supporting a common memory model for FM, OWM, TM (year4); this work is only mentioned here for completeness but is documented in detail in D7.5 Section 9.

The work in Task 6.5 was a collaboration work by the different partners involved and thus we report here the contribution of each partner.

- UAU developed and evaluated an abstraction layer for reliability
- UCY developed and evaluated the dynamic scheduling policy for the DDM-Style model
- UNISI proposed the virtual memory implementation and the TLB integration in the TERAFLUX architecture
- UNIMAN proposed the TM interface

Our achievements show that our goals for this period have been met.

1 Introduction

In the last period of this project, partners contributed in different ways as to achieve the goals that had been set for the project. Regarding the TERAFLUX Architecture Work Package (WP6), the partners contributed to the fine-tuning of the TERAFLUX execution model (Task 6.4) and to the definition and implementation of the TERAFLUX Abstraction Layer (Task 6.5). In terms of the TERAFLUX execution model, UCY, BSC, and UNIMAN have contributed to the evolution of the support for the programming model. UCY proposes different implementations for the support of the DDM-Style TERAFLUX execution. In particular, UCY presents DDM-Style implementations for real systems: an FPGA-based system, a distributed multi-node system, and a many-core system. BSC and UNIMAN present the architecture support for the TaskSs model, and the execution of Transactions, respectively. Both UCY and BSC present an evaluation of the proposed architectural support for a set of applications showing near-linear speedup for different scenarios.

Also, UCY and BSC contribute to the definition of advanced scheduling mechanisms. UCY proposes the dynamic scheduling for the DDM-Style execution while BSC presents the changes in the Hardware TaskSuperScalar.

Within the context of the evolution of the execution model, UNISI describes the TERAFLUX advanced memory management. In particular, the memory consistency mechanisms and estimated overheads are presented.

Another major topic covered in year 4 was the TERAFLUX Abstraction Layer. Regarding this topic, UAU contributes with the definition of the technique and mechanisms for the support of reliability by the Abstraction Layer. UNISI, UCY, and UNIMAN present techniques that focus on the improving the performance through the Abstraction Layer. In particular, UCY focuses on dynamic scheduling for better utilization of the system, UNISI focuses on the virtual memory implementation, and TLB integration to the TERAFLUX architecture, and UNIMAN on the TM support.
Overall, the work described in this document presents the collaborative efforts by the different partners regarding the evolution of the TERAFLUX Execution Model and the Abstraction Layer.

1.1 Document structure

In Section 2 we briefly recall the background work, i.e. the work that has been performed during year 3 of the project within the context of Tasks 6.4 and 6.5. Section 3 presents the work regarding the TERAFLUX execution model that was performed within the context of Task 6.4. In Section 4 we the work regarding the TERAFLUX abstraction layer that was performed within the context of Task 6.5. Finally in Section 5 we present the conclusions for the work performed.

1.2 Relation to other deliverables

In addition to the deliverables of the previous years for this work package (D6.1 and D6.2), the current deliverable is strongly related to deliverable D6.3 ("Fine-tuned TERAFLUX Execution Model") it also describes the work performed in the context of Tasks 6.4 and 6.5 during year 3 of the project.

1.3 Activities referred by this deliverable

The activities described in this deliverable are part of the work performed in TERAFLUX in the context of WP6 in year four of the project (m37-51). This work was performed within the context of the two active tasks for this work package for this period:

- Task 6.4 (m25-51) “Fine-Tuned Execution Model” and
- Task 6.5 (m25-51) “Abstraction Layer”.

2 Background Work

As presented before, the active tasks for period 4 of the project for WP6 are Tasks 6.4 and 6.5. Both these tasks were active for period 3 and period 4 of the project. Given that D6.3 already reported the activities performed within the context of Task 6.4 in year 3, this current document focuses on the activities performed during year 4 for Task 6.4 and both year 3 and period 4 for Task 6.5. For the sake of a better understanding of the activities performed for both Tasks, we start by giving a brief summary of the work that was done within the context of Task 6.4 in year 3 of the project. A more detailed description can be found in deliverable D6.3.

The WP6 work package focuses on the TERAFLUX execution model and the architectural support for this model [34]. The basic TERAFLUX execution model was presented in the first two years of the project. This model is based on the dataflow concepts, where dataflow is used as the policy for scheduling threads (sequences of instructions). Transactions are added to the dataflow threads as a way to explore more parallelism and improve the programmability. Several different types of dataflow threads were defined, as well as the memory model. In addition, we have adopted a template for the architecture proposed within the WP6, which is depicted in Figure 1. More details on the execution model and architecture can be found in D6.1 and D6.2.

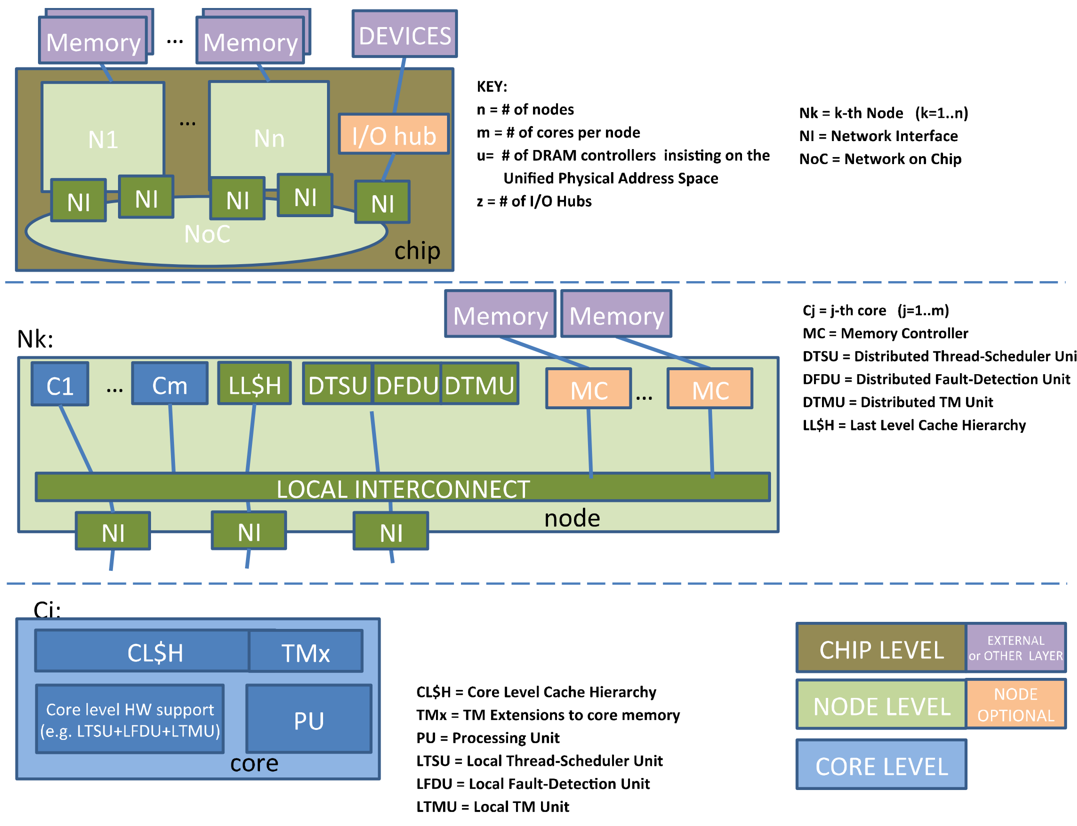


Figure 1: TERAFLUX Architecture Template

In year 3 of the TERAFLUX project, the partners have proposed ways to extend the model as to allow for the efficient execution across different nodes of multi-cores. This required extensions to the D-TSU, which are reported in deliverable D6.3. The implementation of the memory model proposed in previous years and the extension of the T* instruction set have also been part of the efforts performed during year 3.

While it is well known that the dataflow model is able to exploit the maximum available parallelism, making it efficient is a challenge. This is especially true for execution models that depend on the static definition of the dependencies. For this analysis programmers are many times faced with the task of identifying the dependencies among threads. In some cases this might not be possible as dependencies may only be determined at runtime. During year 2 the WP6 partners have developed and tested the use of I-structures at the Node level. During year 3 the WP6 partners have experimented with an efficient mechanism to extend the execution for distributed systems. An alternative approach is to allow the use of dynamic dependence through the TSCHEDULE instruction as done in the T* approach (c.f. D7.1, D6.2), widely adopted and described in D4.6, D5.3 and D7.4.

In terms of hardware modules to support the execution model, in addition to the Thread scheduling modules for the support of DTA- and DDM-style dataflow threads, which are reported in D7.4, during year 3 there was a special effort in developing the modules for support of coarse grain threads (the TaskSs module) and transactions (TM module). The former allows for the system to explore dynamically coarse-grain dataflow threads as a combined or alternate model to the fine-grain DTA- and DDM-style dataflow threads. Recent evolution in the project developed the support of the efficient execution of transactions for exploring the access to shared modifiable variables within dataflow threads.

Lastly, the successful execution of a parallel application depends also on the careful analysis of its execution and overcoming eventual bottlenecks in either the application or the runtime support for the proposed model. In year 3 we have adopted an existing tool for the analysis of the execution of TERAFLUX applications. With this tool it is possible to analyze the status of the different queues in the runtime and the time spent in different routines of both the application and runtime. This analysis helps in tuning the runtime and also determining bottlenecks in the application.

3 Fine-tuned Execution Model

In this section we present the work that was performed by the different partners in enhancing the TERAFLUX execution model.

3.1 Support for the Evolution of the Programming Model

3.1.1 DDM-Style Execution (UCY)

3.1.1.1 TSU++: Multi-node Implementation for DDM-Style Execution (UCY)

TSU++, an object-oriented software implementation of TSUs for the DDM-style applications, supports both single-node and multi-node execution of DDM-style applications. The communication (data or updates) is conducted through network messages using the IP addresses of the participant nodes. During the start of the execution of a DDM program a connection between all participating nodes is conducted and the execution starts from the root node. Programmer must be careful to add extra dependencies in order to avoid having both read and write dependencies on the same task as in Figure 2.

Data reside in a global memory space and tasks exchange data by reading and writing at that global space. Dependencies are formed whenever a number of tasks want to read the global memory data produced by a parent task or when a single task wants to modify that data. As soon as a task produces data, the TSU will ensure that the data will be exchanged with the remote nodes that will run tasks that are dependent on that data.

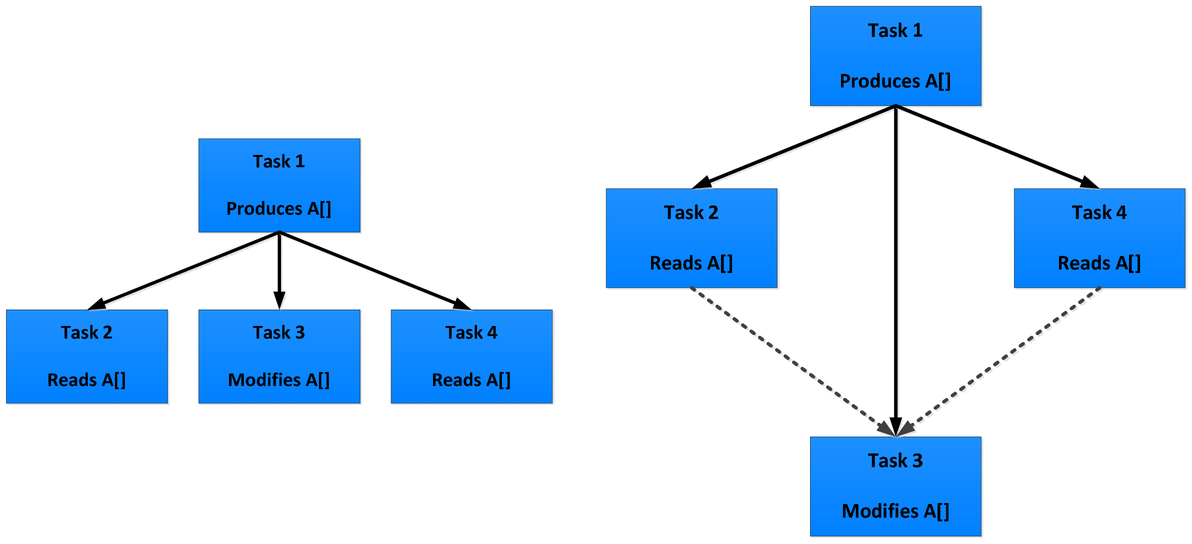


Figure 2: Task 3 may modify A[], but Task 2 and 4 want to read A[] as produced by Task 1. The solution is adding control dependencies between Task2->Task 3 and Task 4->Task3

Each task informs the TSU about the data it produced (one call per produced variable). Each dependency contains information about what data are required by the consuming thread. The TSU processes data dependencies one by one. It determines the node that will execute the dependent task and checks to see if the data was sent to that node. If data not yet sent, it sends the data to the node and marks them as sent to that specific node. Remote nodes that receive the data, update their version of the global memory, and then afterwards decrement the *readycount* of the dependent task.

For the evaluation of this system we have executed different application on a multi-node multi-core platform of Four AMD Opteron 6276, with 2 CPU (2.30GHz 16 cores), 48 GB RAM with Gigabit Ethernet and no cache coherence across the Nodes. We have chosen the Blocked Cholesky Factorization benchmark because it is the most complex application ported to TSU++ system so far. It has a very complex dependency graph. Also, it's a computationally intensive and performance-sensitive benchmark. **Table 1** illustrates the characteristics of the execution scenario of the Cholesky benchmark. We have run four different execution scenarios on four DDM nodes. Each node is equipped with 32 cores (AMD Opteron 6276).

In TSU++ the TSU is implemented as a software module running on one of the machine's cores, while the threads' execution takes place on the other cores. Notice that in the Scenario_4 we have evaluated our system by using all the cores for the threads' execution. As such, the TSU's code is switched with the threads' code on the cores. The speedup results are depicted in Figure 3. The Cholesky application achieves very good speedups despite its complex dependency graph.

Table 1: The execution scenarios of Cholesky benchmark

Scenario	Number of Nodes	Number of Cores/Node	Total Cores	Matrix Size	Block Size
Scenario_1	4	31	124	16 K	128
Scenario_2	4	31	124	32 K	256
Scenario_3	4	31	124	32 K	128
Scenario_4	4	32	128	32 K	128

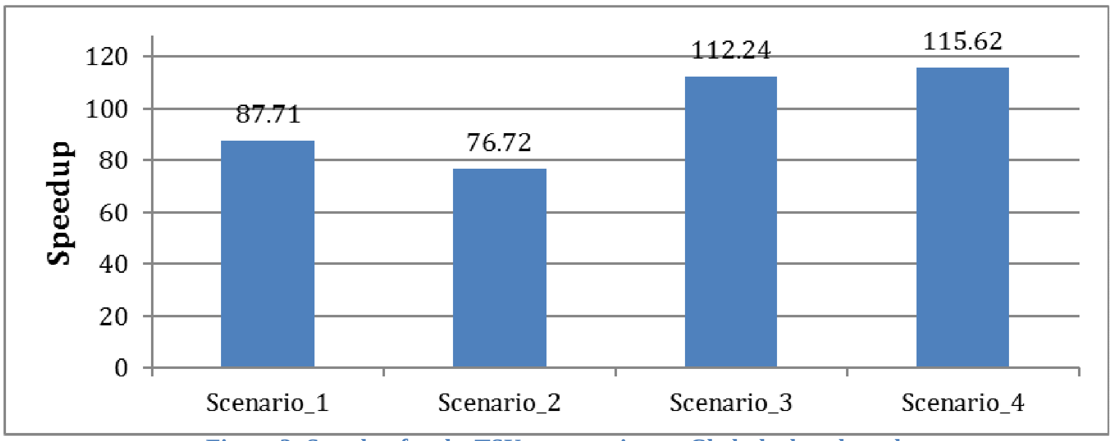


Figure 3: Speedup for the TSU++ execution on Cholesky benchmark

We have developed an OpenMP version of the Cholesky benchmark (Matrix size 4096 with Block size 32x32) and compared it with the Data-Flow version (DDM-style of execution) a 32 core machine (AMD Opteron 6276). The OpenMP achieved speedup of 9.1 versus 25.9 for the Data-Flow version.

Finally, we have compared the TSU++ implementation with the MPI implementation of the Blocked Cholesky benchmark from github (https://github.com/pawnbot/Matrix-Inverse) on a distributed multi-core environment. From the results below we can see that our system outperforms the MPI framework due to its data-driven execution as well as its efficient and lightweight network interface. For this comparison we have used 1, 2 and 4 AMD Opteron 6276 machines. The most notable test case is the last one where we have run the benchmark using 128 cores on 4 different machines. For 32 cores DDM gets speedup close to 25 and MPI around 11. Overall DDM achieves speedup slightly above

115 where the MPI achieves only 4. This is due to the fact that the Cholesky algorithm has very complex data dependencies that cannot be handled well in the MPI implementation.

Configuration					*DDM*			*MPI*		
Matrix Size	*Block Size*	*Cores Per Node*	*# of Nodes*	*Total Cores*	*Serial Time*	*Parallel Time*	*Speedup*	*Serial Time*	*Parallel Time*	*Speedup*
4K	64	32	1	32	215.53	8.32	**25.89**	42.90	5.38	**7.97**
8K	64	32	1	32	1065.70	41.84	**25.47**	334.48	30.06	**11.13**
8K	64	32	2	64	971.74	24.56	**39.56**	335.25	306.93	**1.09**
16K	128	32	2	64	13595.94	197.66	**68.78**	2730.15	1360.47	**2.01**
32K	128	32	4	128	105386.21	911.48	**115.62**	21682.60	5432.17	**3.99**

3.1.1.2 TFluxSCC Implementation for Intel SCC (UCY)

TFluxSCC [2] is a software platform for the execution of DDM-style applications on the Intel SCC processor [5]. TFluxSCC is based on the TFlux [8] Data-Driven Multithreading (DDM) platform that was developed for commodity multicore systems. This is an efficient implementation of the DDM-style model on a clustered many-core that is used as a case study to achieve high degree of parallelism. With TFluxSCC we achieve scalable performance in a cluster of many simple cores using global address space without the need of cache-coherency support. Our scalability study shows that applications can scale, with speedup results ranging from 30x to 48x for 48 cores.

We want to show that using the DDM-style model of execution on the Intel SCC we are able to avoid the restrictions or the limitations of the architecture that may affect the performance or the programming style. Although the Intel SCC provides a global address space, it doesn't allow caching data coming from this memory location as it doesn't have any support for hardware cache-coherency. The DDM-style model though, and consequently our TFluxSCC implementation, doesn't require any hardware to maintain coherency as it disallows simultaneous access to shared data.

Contrary to the original TFlux implementation, in TFluxSCC we integrate the TSU functionality with the application thread as shown in Figure 4.We remove the busy wait loop from the TSU and call its operations at the end of the execution of an application thread, which is the only time that the TSU will have real operations to execute (send update messages to consumers). This solution allows us to utilize the execution unit of the core to the maximum.

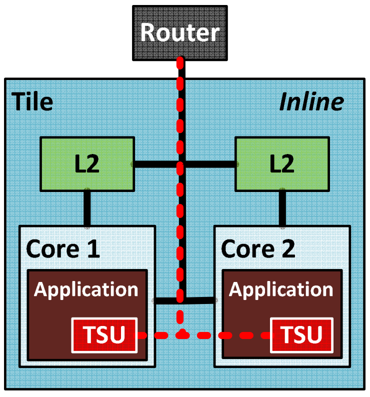


Figure 4: TSU Implementation for TFluxSCC

We performed a scalability study of the performance for six applications with different characteristics. In our workload we have included applications that are embarrassingly parallel, applications that are compute-bound and others that have a combination of memory- and compute-bound, as well as more complex dependencies among the different parallel threads. Three of them are kernels that represent common scientific operations; two belong to the *MiBench* suite [4] and one to the *NAS* suite [1]. Our hardware setup was an Intel SCC experimental processor, RockyLake version. The system has a total of 32GB of main memory. The operating system used for the Intel SCC cores was the Linux_dcm kernel provided by Intel SCC Communities repository that supports caching the data coming from the off-chip shared-memory to L2 cache. To cross compile our benchmarks for the Intel SCC we used the GCC v.3.4.5 compiler with the optimization flag O3. For porting and executing the applications on the Intel SCC we used the RCCE v1.4.0 tool-chain [7].

Our study emphasizes on the scalability of the DDM-style model, thus in Figure 5 we present Speedup values for 2-to-48 cores using the large input data set size. The results in Figure 5 show a large speedup for most applications. The application with the largest overall speedup is TRAPEZ, which is an application that is compute-bound and suffers no memory overheads. RK4, which has a considerable number of threads and dependencies, achieves also a good speedup and thus it shows that the execution of the TSU code does not incur in a large overhead for the execution of the application. QSORT* that performs the partial sorting application shows considerable improvement in performance compared to the original QSORT. MMULT that is both a compute- and memory-bound application performance a maximum of 38x speedup. Finally, FFT and QSORT show the smallest speedup of all applications. QSORT is split into two phases. The first one is like QSORT* where the total vector is split into smaller parts and each core sorts its part independently. This phase has linear speedup. The following phase combines the results of all sorted parts as to build the complete sorted vector. This is done as a reduction using the merge sort algorithm.

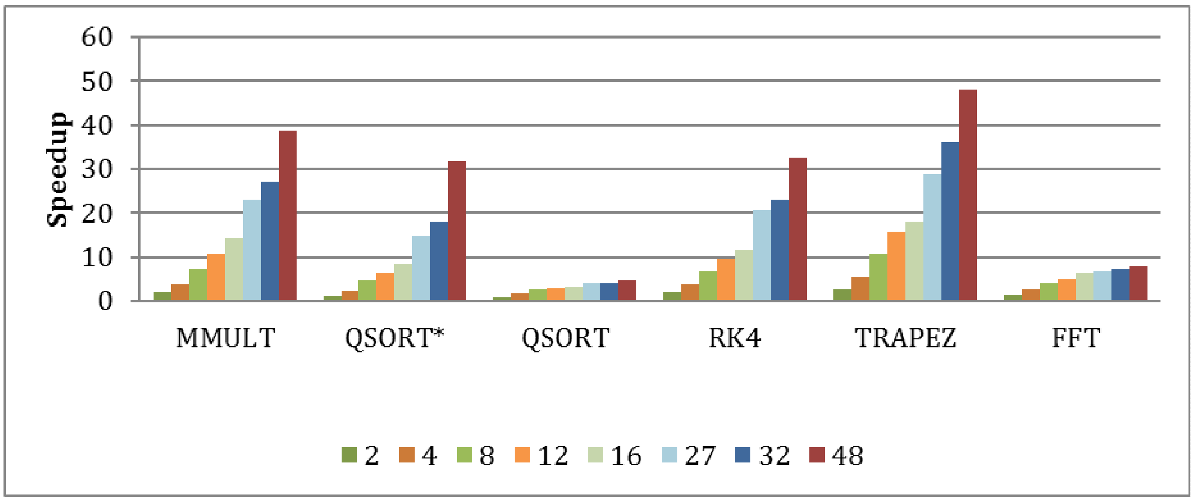


Figure 5: Speedup results for TFluxSCC for MMULT, QSORT*, QSORT, RK4, TRAPEZ, and FFT, on the Intel 48-core SCC system

3.1.1.3 DDM-mc: The hardware DDM-style implementation on FPGA (UCY)

In this section we present the implementation of the Data-Driven Multithreading Multi-core (DDM-mc) system, a novel parallel system that supports the DDM model. DDM-mc has been implemented on a Xilinx Virtex-6 FPGA [9]. The proposed system consists of two major modules: 1) the Multi-core Processor, an eight core shared memory system that is utilizing a hardware TSU implementation, 2) and the Runtime System, a software support that handles the communication between the DDM-mc applications and the Multi-core Processor. DDM-mc allows getting real values about time, power consumption and overheads of a Threaded Dynamic Dataflow implementation.

The performance evaluation has shown that DDM-mc gets good results for both large problem sizes and for very small. With this work we have demonstrated that Threaded Dataflow concurrency can be supported in hardware with negligible overheads and a very small hardware and power budget. DDM-mc provides dataflow concurrency without the need for cache coherence and at the same time it is low complexity and low power system.

3.1.1.3.1 The DDM-mc Multi-core Processor

The DDM-mc Multi-core Processor (Figure 6: Block diagram of the DDM-mc Multi-core processor. Figure 6) is a shared memory octa-core that implements the DDM execution paradigm by utilizing a hardware version of the TSU. It consists of eight MicroBlaze Blocks (MBBs), each of the featuring a Xilinx MicroBlaze [10] soft-core with its caches and local memory. The MicroBlaze is a 32-bit RISC Harvard architecture processor that operates at 100-MHz. The data accesses are cached by a 32-KB L1 cache (D-Cache), while the instruction accesses are cached by a 16-KB L1 cache (I-Cache). The MBBs exchange data with the TSU through the Fast Simplex Link (FSL) Buses [11]. An FSL Bus is a fast 32-bit wide interface that provides unidirectional FIFO-based communication. The TSU dispatches the threads that are going to be executed to the MBBs, through the Output FSLs. The MBBs send DDM commands (updates, thread templates, etc.) to the TSU through the Input FSLs.

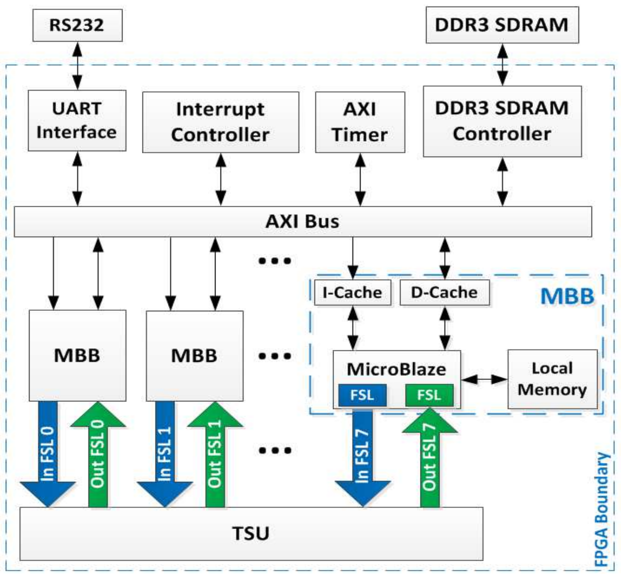


Figure 6: Block diagram of the DDM-mc Multi-core processor.

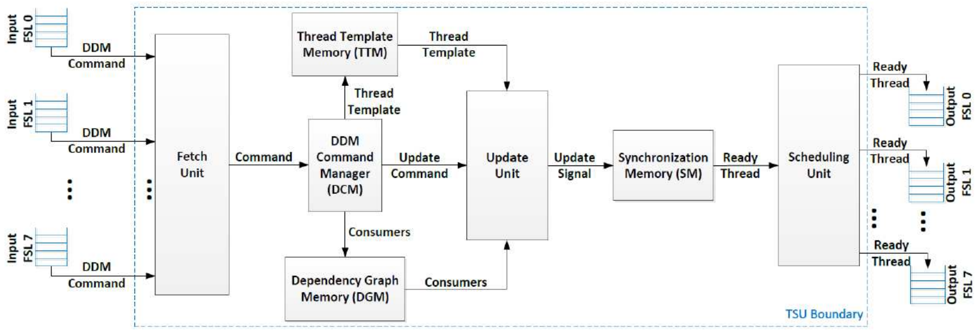


Figure 7: Block diagram of the TSU Micro-architecture supporting eight cores.

In this case, the Thread Scheduling Unit (TSU) is developed as a hardware peripheral using the Verilog HDL. It uses threads' meta-data for the data-driven scheduling of the threads, such as the Thread ID, the scheduling policy, the Instruction Frame Pointer and the consumers of the thread. Figure 7 depicts the block diagram of the TSU's microarchitecture that supports eight cores. The TSU uses three main units for storage, the Thread Template Memory (TTM), the Dependency Graph Memory (DGM) and the Synchronization Memory (SM). The TTM contains the Thread Template of each thread, i.e. the thread's meta-data, while the DGM contains the consumers of each thread. The consumer threads are kept separately from the TTM to facilitate simultaneously access from the TSU. The SM contains the Ready Count (RC) values for each thread. An RC value indicates the number of producer-threads that a thread needs to wait before its execution. A thread that implements a loop has multiple instances, one for each iteration, hence the TSU holds a separate entry for each instance of a thread in the SM.

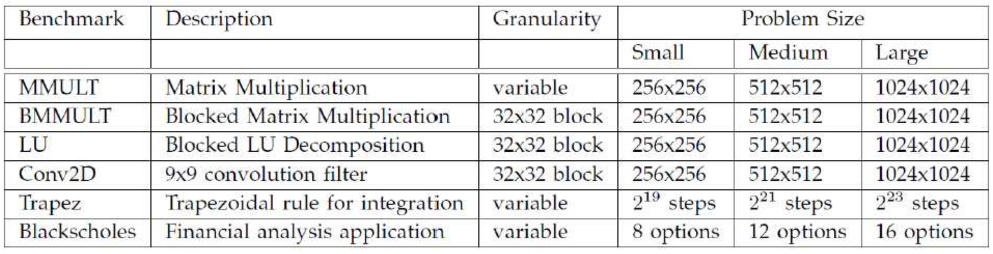
The Fetch Unit dequeues the DDM commands, send by the cores, from the Input FSLs in a round-robin fashion and it forwards them to the DDM Command Manager (DCM) for further processing. The DCM is responsible for storing and removing the information into/from the TTM and DGM modules. Also, it forwards the update commands to the Update Unit. The Update Unit reads the Thread Template attributes from the TTM. Also, it locates the thread's consumers in DGM if it's necessary. After that, the Update Unit decreases the RC of the corresponding threads in the SM. If the RC value of a thread reaches zero, then it will be deemed that it is ready to be executed and so it is sent to the Scheduling Unit.

The Scheduling Unit enforces the scheduling policy by assigning a ready thread to the corresponding Output FSL. Two scheduling methods have been implemented: dynamic and static. The dynamic method distributes the thread invocations to the cores in order to achieve load-balancing. In the static method the thread instances are assigned to a specific core.

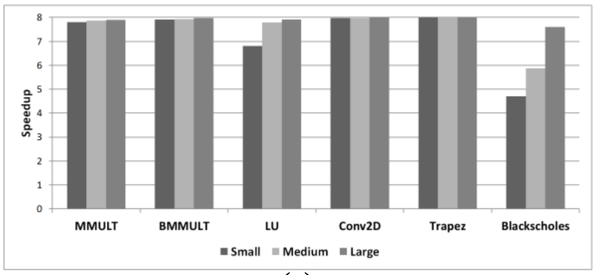
3.1.1.3.2 System Evaluation

For the performance evaluation we use a suite of six different benchmarks widely used in scientific and image processing applications. Table 2 illustrates the characteristics of the benchmarks along with the problem sizes. The execution time measurements were collected using the AXI Timer module of the system. Figure 8 depicts the system evaluation. The speedup results of all six applications are depicted on the left graph of the figure. Figure 8: System Evaluation. (a) Speedup for the DDM-mc execution. (b) Speedup for the DDM-mc execution on very small sizes. The results show that the DDM-mc system scales very well across the range of the benchmarks achieving almost linear speedup. We have also evaluated the ability of the DDM-mc to handle small problem sizes and ultra-lightweight threads: the speedups of the DDM-mc system on very small sizes (16x16, 32x32 and 64x64) are shown on the right graph of the figure. The three applications, MMULT, BMMULT and Conv2D achieve speedups from 7 to 7.96. The high complexity of the LU ended up with smaller speedups in the order of 2 to 4.4. Note that we kept the thread size of the blocked algorithms to 4x4 for these experiments.

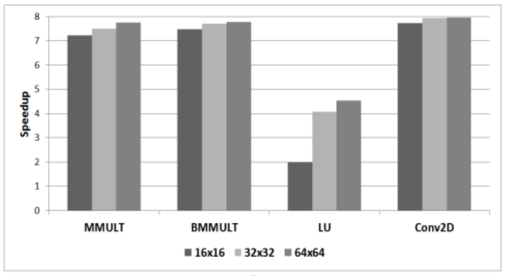
Table 2: The benchmark suite characteristics.



Benchmark	Description	Granularity	Problem Size		
			Small	Medium	Large
MMULT	Matrix Multiplication	variable	256x256	512x512	1024x1024
BMMULT	Blocked Matrix Multiplication	32x32 block	256x256	512x512	1024x1024
LU	Blocked LU Decomposition	32x32 block	256x256	512x512	1024x1024
Conv2D	9x9 convolution filter	32x32 block	256x256	512x512	1024x1024
Trapez	Trapezoidal rule for integration	variable	2^{19} steps	2^{21} steps	2^{23} steps
Blackscholes	Financial analysis application	variable	8 options	12 options	16 options



(a)

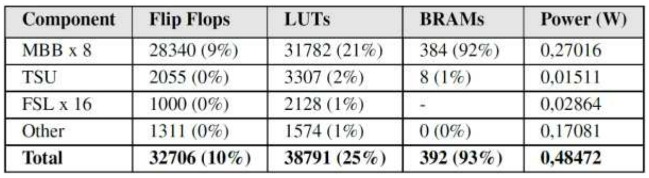


(b)

Figure 8: System Evaluation. (a) Speedup for the DDM-mc execution. (b) Speedup for the DDM-mc execution on very small sizes.

Table 3 depicts the FPGA resource utilization and power consumption estimations for the prototype of the DDM-mc processor. The utilization percentage of each component is shown in parenthesis. The component labeled “other” includes the clock generator, the interrupt controller, the timer etc., all necessary for the proper functionality of the system, but outside the scope of this work. The hardware device utilization of our prototype is rather low, which will enable us to extend the functionality of our system in the future. The Block RAM (BRAM) utilization on the other hand is quite high (93%). We choose to utilize as much as possible BRAMs, to model big caches and local memories in order to increase the system’s performance. The maximum operating frequency of the TSU peripheral is 198-MHz. Furthermore, the TSU consumes a small proportion of the overall power of the system. Particularly, an MBB consumes 223.5% more power than TSU. We believe that this feature will allow the implementation of a power efficient multi- and many-core DDM system.

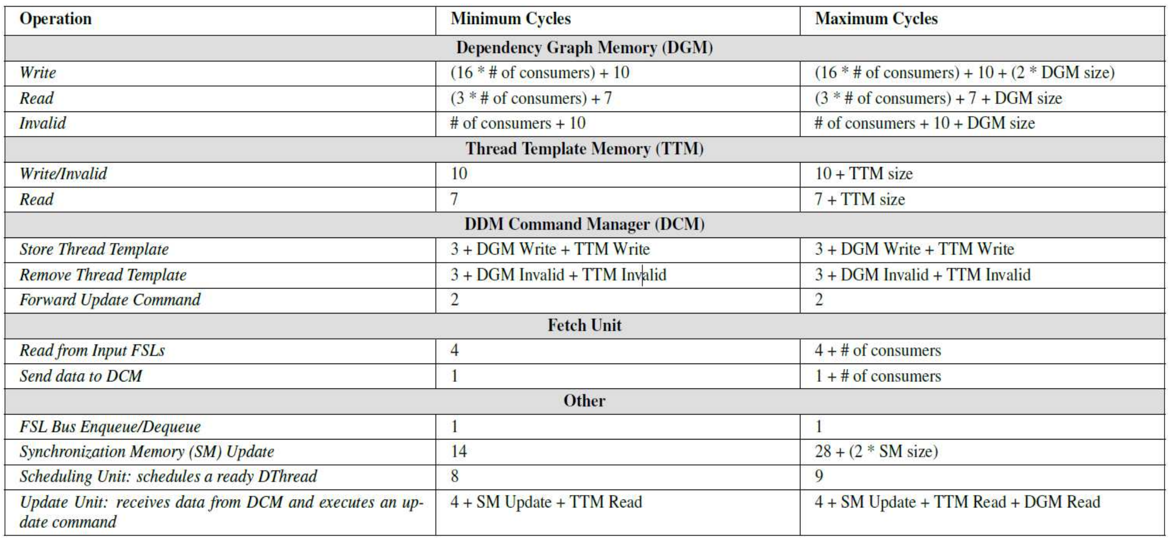
Table 3: Virtex-6 FPGA resource utilization and power consumption estimations.



Component	Flip Flops	LUTs	BRAMs	Power (W)
MBB x 8	28340 (9%)	31782 (21%)	384 (92%)	0,27016
TSU	2055 (0%)	3307 (2%)	8 (1%)	0,01511
FSL x 16	1000 (0%)	2128 (1%)	-	0,02864
Other	1311 (0%)	1574 (1%)	0 (0%)	0,17081
Total	**32706 (10%)**	**38791 (25%)**	**392 (93%)**	**0,48472**

Table 4 depicts the minimum and maximum cost (in cycles) of the TSU operations. Since the TSU operates dynamically, the majority of its operations depends on the size of its structures. For instance, the maximum cost of the TTM Write/Invalid operation is equal to its minimum cost plus the size of the TTM structure in cycles. The cost of the operations that manage consumers, such the DGM Write operation, depend on the number of the consumers (# of consumers) that it will manage. Moreover, the cost of some operations depends on the cost of other operations. For instance, the minimum cost of the DCM’s store operation depends on the minimum cost of the DGM Write and TTM Write operations. The size of the TSU structures are as follows: the Input and Output FSLs consist of 128 entries, the DGM and TTM consist of 16 entries and the SM consists of 1024 entries. The SM Update operation costs a lot because the SM size is equal to 1024. The reason for this is that the SM module allocates and deallocates the RC values dynamically. In particular, the maximum cost of the SM Update operation depends on the allocation procedure. To solve this issue, we allocate blocks of RC values (32 RC values each time) in order to avoid frequent allocations. This technique improves the TSU performance since the SM Update operation is the most frequent operation. Notice that the TSU operations are performed in parallel. For example, while the Fetch Unit reads data from the Input FSLs, the Update Unit can perform updates and the Scheduling Unit can store ready threads in the Output FSLs.

Table 4: Timings of the TSU operations



Operation	Minimum Cycles	Maximum Cycles
	Dependency Graph Memory (DGM)	
Write	(16 * # of consumers) + 10	(16 * # of consumers) + 10 + (2 * DGM size)
Read	(3 * # of consumers) + 7	(3 * # of consumers) + 7 + DGM size
Invalid	# of consumers + 10	# of consumers + 10 + DGM size
	Thread Template Memory (TTM)	
Write/Invalid	10	10 + TTM size
Read	7	7 + TTM size
	DDM Command Manager (DCM)	
Store Thread Template	3 + DGM Write + TTM Write	3 + DGM Write + TTM Write
Remove Thread Template	3 + DGM Invalid + TTM Invalid	3 + DGM Invalid + TTM Invalid
Forward Update Command	2	2
	Fetch Unit	
Read from Input FSLs	4	4 + # of consumers
Send data to DCM	1	1 + # of consumers
	Other	
FSL Bus Enqueue/Dequeue	1	1
Synchronization Memory (SM) Update	14	28 + (2 * SM size)
Scheduling Unit: schedules a ready DThread	8	9
Update Unit: receives data from DCM and executes an update command	4 + SM Update + TTM Read	4 + SM Update + TTM Read + DGM Read

3.1.1.3.3 CacheFlow on DDM-MC

DDM can improve the locality of sequential processing by implementing deterministic data prefetching using data-driven caching policies called CacheFlow [6]. CacheFlow policies include firing a thread for execution only if the code and data of the thread are present in the cache. Furthermore, blocks associated with threads scheduled to execute in the near future are not replaced until the thread finishes its execution. Results of applying CacheFlow have shown that CacheFlow reduces cache misses considerably, even on caches of small sizes. Figure 9 depicts a proposed design for the implementation of CacheFlow on the DDM-mc system. More specifically we show how the MBB has to be modified to support CacheFlow. For this functionality we will use a Scratch Pad Memory (SPM) instead of Data Cache (D-Cache). The SPM will be used for storing the prefetched threads' data. The SPM controller will be responsible for transferring the data of the ready threads from main memory (External DDR3 SDRAM) into the SPM. When the data will be allocated in SPM for a specific thread, the SPM controller will inform the MicroBlaze to execute this thread through the Output FSL Bus. Moreover, when a thread finishes its execution, the SPM controller will be also responsible for writing back the modified thread's data into the main memory. Currently we are working on the implementation of the SPM controller. Particularly, we are working on the part that is responsible for transferring the data from SPM to main memory and vice versa.

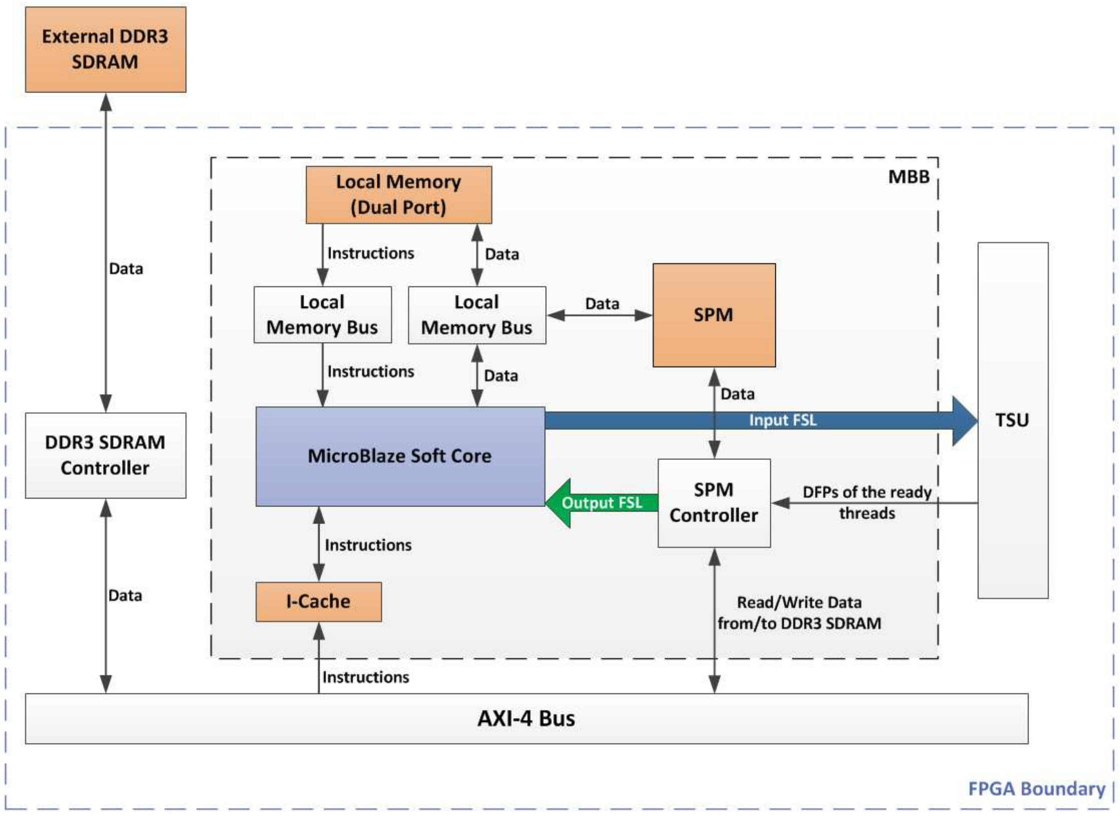


Figure 9: CacheFlow implementation on DDM-mc.

3.1.2 TaskSs – another approach for a hardware TSU (BSC)

In this work we have contributed with a significantly improved Hardware TaskSuperScalar design (cf. D6.1, D6.2, and D6.3): Picos [12] [13]. Picos hardware is a major revision of the Hardware Task Superscalar architecture with several improvements in its work-flow. The main improvements are related with architectural changes to add support to nested tasks, better memory management and faster task dispatching (more details are explained in section **Error! Reference source not found.**). Figure 10 shows the organization of a computing system that includes the Picos hardware. It is composed by a many-core with any number of threads that send two kind of task information to the Picos hardware: (1) the dependency information of new tasks, and (2) the notification of ending a task. The Picos hardware is composed by one gateway (GW), one or more Dependence Chain Trackers (DCT), one or more Task Reservation Stations (TRS) and one Task Scheduler (TS). All those components work together in parallel in order to build the dynamic task dependency graph and generate a list of ready tasks that are sent back to the threads to be executed. The connections between the modules are decoupled by FIFO queues that are interconnected by arbiter modules (not drawn in Figure 10 for clarity). There is one arbiter module between the output queues of one type of module and the input queues of a different type of module (for example, one arbiter reads from a single output queue from the GW and writes to one of the input queues of the adequate TRS).

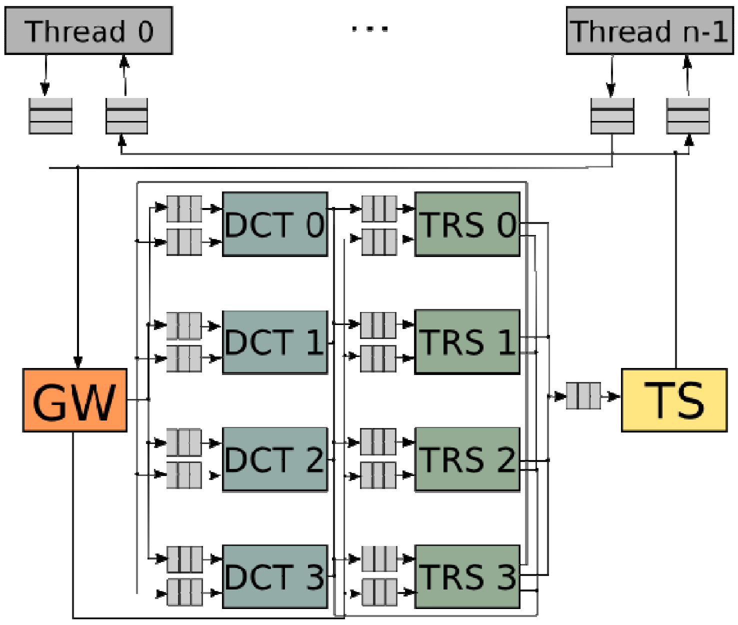


Figure 10: Computing system with Picos pipeline hardware

Operational Flow Overview

Once a thread reaches a task creation it creates a new Work Descriptor (WD) (similar to the concept of continuation, cf. D6.1) that basically is a memory structure containing the necessary information for the new task to be executed. This information mainly includes the address of the task code to execute and the address of all its dependencies with their directions (input, output, input and output - inout, or direct for immediate values). Once this WD is created, it is sent to the Picos hardware that reads its information and stores the data of the corresponding task until all its dependencies are fulfilled. For the first task created all its dependencies are ready because all its input and inout dependencies are already in memory. However, the most common case is that a task has to wait until one or more of its dependencies become ready after other tasks finish. The information (finishing messages) of those finished tasks is sent to the Picos hardware by the threads that execute those tasks. With this finishing message Picos will delete the corresponding WD in the system and proceed to mark as ready all the task dependencies that could be waiting for the dependencies of the just finished task. Then, the Picos hardware will try to send the new ready task/s to execute. That means sending the WD to the TS, which will put it available to all the threads in the system. When one thread that is not busy realizes that a new WD is available it starts executing the corresponding task. If a task creates new tasks, new WDs are created and the dependency information is sent to Picos as explained above.
The main differences with previous versions are:

- Now original ORT and OVT modules are joined in the new DCT module.
- The memories and packets have been redesigned to use the minimum amount of resources.
- The GW module has been redesigned with TRSs memory availability information in order to increase the task scheduling throughput.

- Also the GW module now receives the Finished Task packets to preserve the order between task finishing packets and children tasks creation packets as a way to support nested tasks.

3.1.3 TM Support (UNIMAN)

First we briefly recall the context of Transactional memory (TM). TM attempts to simplify concurrent programming by allowing a group of load and store operations to execute in an atomic way. It is a concurrency control mechanism analogous to database transactions for controlling access to shared memory in concurrent computing. TM systems can exist in hardware, software or as hybrid implementations [16] [17] [18] [19] [20]. This discussion is related to the hardware implementation of TM systems.

TM hardware must perform a number of tasks.

1. Transaction modifications are isolated from the rest of the system until commit time through data versioning.
2. The system detects and resolves read-set and write-set conflicts.
3. Transaction commits appear to occur atomically.
4. In case of a conflict leading to an aborted transaction, a consistent state is reached after rewinding.

Transactional mechanisms are being designed while keeping following requirements under consideration.

1. Performance should be achievable without an undue burden on the programmer.
2. The mechanisms should scale gracefully to large systems with large amounts of concurrency.
3. The system should be able to cope with, and if possible exploit, a hierarchical organization of cores into nodes.

Our research at Manchester University aimed to answer the following questions regarding a TM hardware system.

1. Is it better to exchange information about sharing between transactions as they go along or to do so only at the commit time?
2. How can we leverage the node (clustered) architecture to provide good performance for transactions?
3. What sharing patterns exist across a broad range of workloads?
4. What is the best balance between communication, storage and false sharing? It may be that consistent performance can only be achieved through adaptive mechanisms.

At Manchester we have developed a scalable transactional memory system in COTSon. The scalable system is a purely lazy implementation but the commit process takes advantage of a hierarchical organization of cores into nodes (clusters). Figure 11 shows a high-level view of the scalable model that we are evaluating, which is conformant to the TERAFLUX architectural template (cf. Figure 1, Section 2).

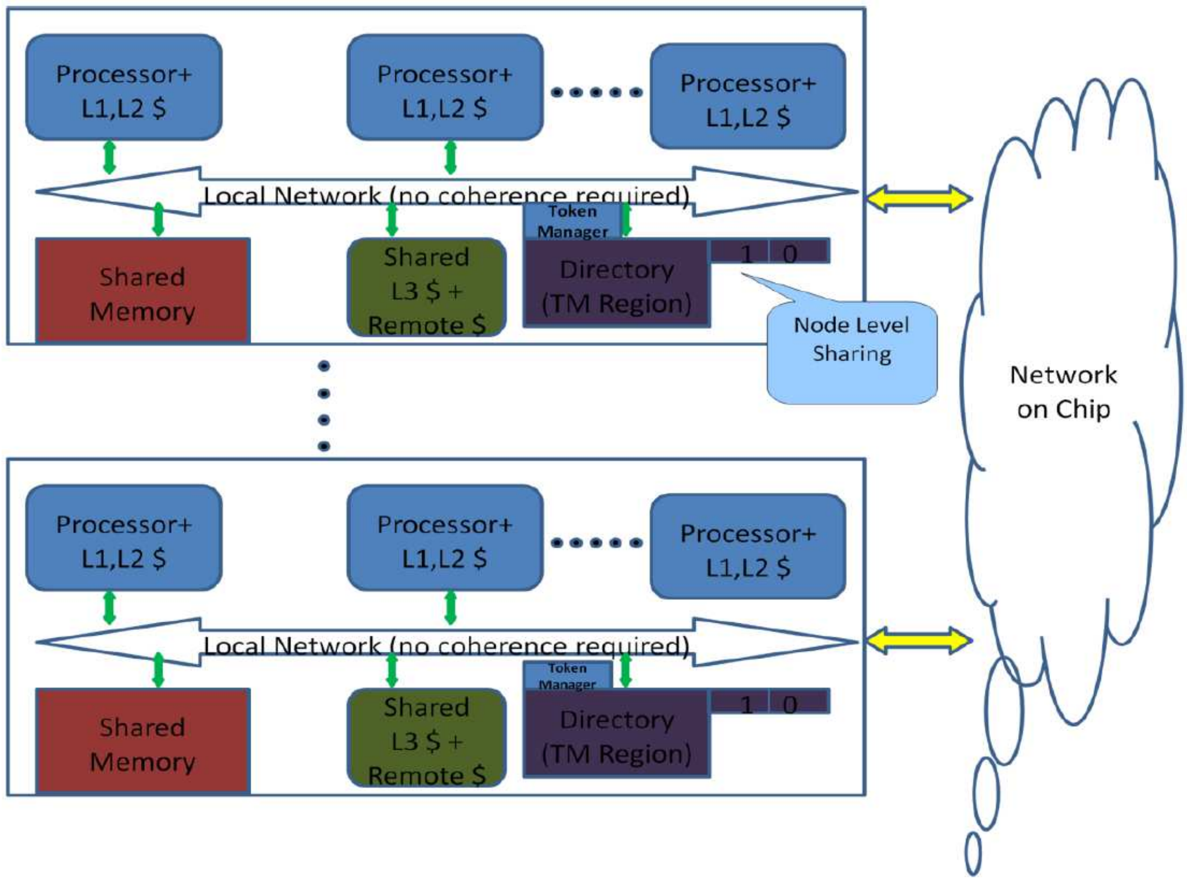


Figure 11: Clusters with extensions to cache and directory to support Transactional Memory

The system consists of multiple nodes with each node having multiple cores. Each core has its own private L1 and L2 cache. Within each node (cluster) there is a shared L3 cache, a directory and part of distributed memory. The directory tracks transactional memory regions and maintains information at cache line granularity. Each directory entry contains a bit vector to represent sharers. Sharers are maintained at the node level.

L1 and L2 caches are used to maintain data versioning. During commit, a transaction first *occupies* all the directories in its R/W-set and *marks* all the cache lines in its write-set. The “occupy and mark” process is similar to Scalable TCC [14]. After completing the occupy phase the transaction locks the L3 controller and then writes back all its modified TM lines to the shared L3. After writing back all its data, the transaction *unlocks* the L3 controller and then sends commit messages to all the directories in its write set so that they can send the relevant invalidations. The write back is required so that L3 contains the most up to date copy and can respond to any requests to the cluster.

There are many optimizations possible to our implementation. For example using bloom filters to reduce the size of our read/write-set and to lock directories at lower level of granularity [15].

Evaluation:
For the evaluation purposes we have run two of the STAMP benchmarks on the scalable TM hardware. In Figure 12 we show the performance results of these benchmarks.

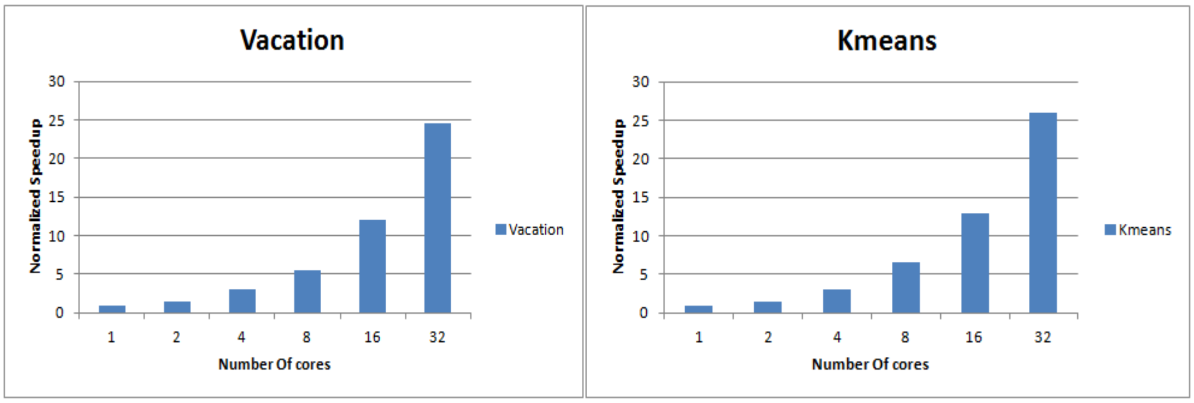


Figure 12: Speedup, normalized to single core execution

Token based transactional memory

Here we describe our design for a token based transactional memory system that keeps a check on the number of potentially conflicting transactions running in a TM system. One of our major work in TERAFLUX is to separate memory into multiple regions (WP3). An interesting area of research is to divide transactional memory regions into further smaller sub-regions with special hardware maintaining the number of concurrent access to these sub-regions. We call this hardware the *transaction token manager*. The sub-regions can be as small as a cache line and can be as big as multiple page sizes. The token manager maintains tokens for each sub-region and keeps a small history of the number of conflicts that has happened between transactions due to accesses to particular sub-regions. The history needs to be small to make it feasible to be implemented in the hardware. Based on the history, the hardware *controls* the number of concurrent transactions that can access a particular transactional sub-region. If the number of concurrent transactions reach a threshold, any further accesses to the sub-region by new transactions are *delayed*.

The token manager maintains tokens for each TM sub-region. When a transaction requests an address from a particular sub-region the hardware *protecting* that region looks at its history and based on the history provides a transaction with a token to this sub-region. Token gives *access rights* to a transaction. Once a transaction finishes it returns the token back to the hardware.

The token manager protecting the sub-regions is distributed with one token manager per cluster in a multi-cluster system. Figure 13 shows the token managers in a multi-cluster TM system.

We have been working on the design of the system but due to lack of time we were not able to implement the complete system in order to thoroughly test it for evaluation.

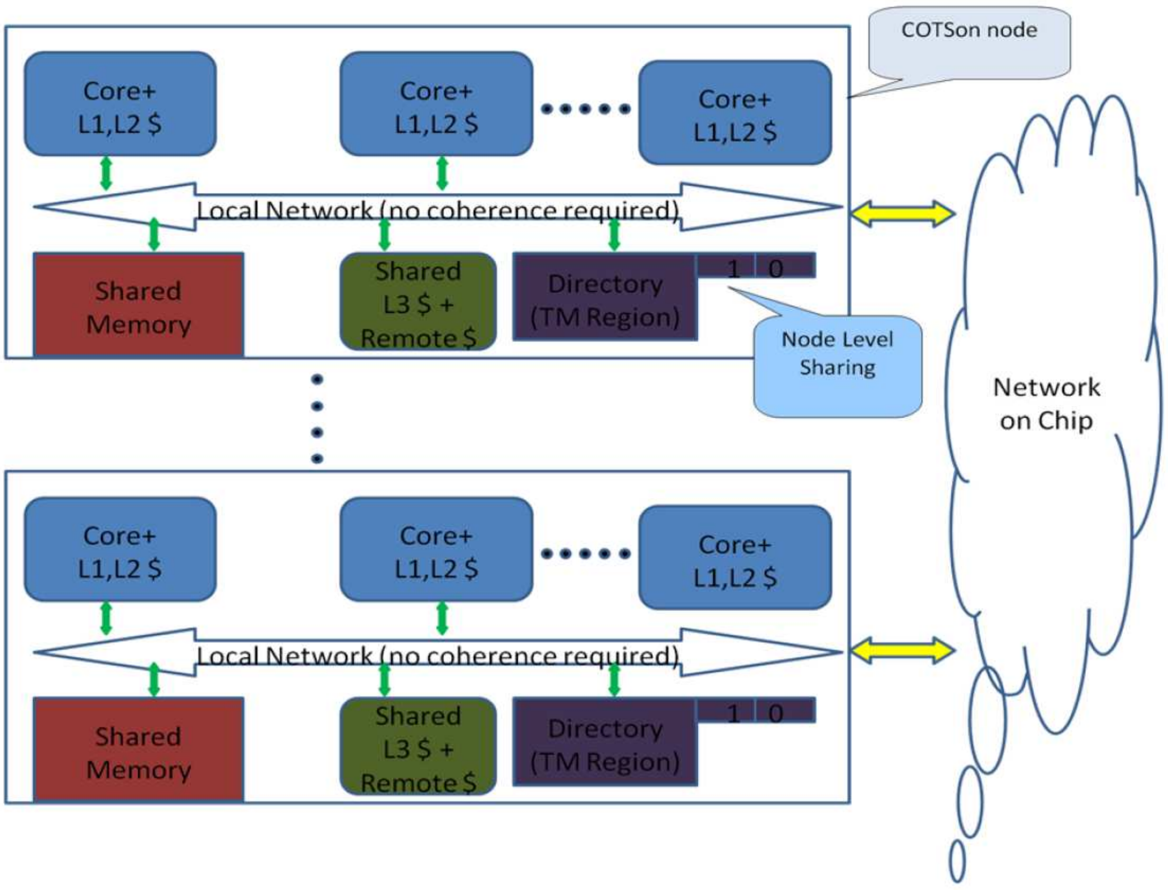


Figure 13: Transactional Token Managers in a cluster based Transactional Memory system.

3.1.4 Architectural Support for Task Scheduling (UNIMAN)

In this section we describe UNIMAN's hardware implementation and testing of scheduling approaches for dataflow programming models. This is developed on the basis of the same line of reasoning of UNISI, BSC, UCY (cf. D7.1, D6.1, D6.2, D6.3), but provides also novel elements.

The scheduler can make decisions that improve data locality if it is sufficiently aware of the data requirements of the tasks by placing tasks on cores whose caches contain the required data. In the general case information about which tasks use which data is absent as a consequence of the way in which conventional imperative programming models have been extended to include the ability to perform parallel execution. However, models like dataflow programming implicitly allow for parallel execution and contain additional information about the access patterns of the computation. The main characteristics of the dataflow model are that the execution of an operation is constrained only by the availability of its input data. The computation is performed by side effect free tasks and the execution is triggered by the presence of data instead of the explicit flow of control. These constraints prevent both deadlocks and race conditions.

In this section we demonstrate how the structure provided to programs by the dataflow programming model can be incorporated into task schedulers, making them aware of a task's data requirements without any further help from the programmer.

Our major contributions are:

—An examination of how the use of a dataflow programming model can by allowing more intelligent scheduling techniques improve system performance.

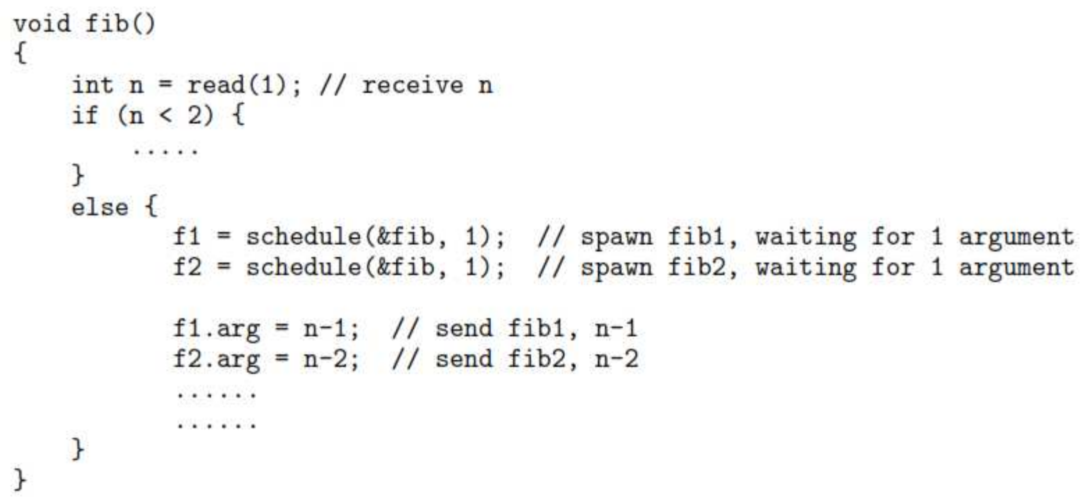
—We propose two novel scheduling policies, 'Token Scheduling' and 'Reference Scheduling' and demonstrate how these scheduling policies result in better resource utilization.

—We propose the design of a scalable hardware scheduler that has low hardware complexity and is relatively insensitive to the access latency of the hardware queues.

—A demonstration that the proposed architectural support has significant performance benefits and the scheduling policies have much better resource utilization when compared with other scheduling policies. Our scheduling policies result in a reduction in cache misses by up to 72% and 95% for the L1 and L2 caches respectively compared to FIFO scheduling (see next sections).

SCHEDULING

Dataflow is an asynchronous and self-scheduling model where the execution of nodes is constrained only by data dependencies. From the code, Figure 14, we can see that each 'fib' function creates two new tasks 'fib1' and 'fib2'. The schedule(&fib, 1) creates a dataflow task which executes fib as its function once all the dependencies are computed. It also informs the dataflow scheduler that this task is waiting for only one piece of input data to be ready for execution. Finally fib1's and fib2's argument is set to the value n-1 and n-2 respectively, making them ready for execution.



```
void fib()
{
    int n = read(1); // receive n
    if (n < 2) {
        .....
    }
    else {
           f1 = schedule(&fib, 1);  // spawn fib1, waiting for 1 argument
           f2 = schedule(&fib, 1);  // spawn fib2, waiting for 1 argument

           f1.arg = n-1;  // send fib1, n-1
           f2.arg = n-2;  // send fib2, n-2
           ......
           ......
    }
}
```

Figure 14: A dataflow function for computing Fibonacci numbers.

Figure 15 shows how tasks are managed by a dataflow scheduler using example code of Figure 14. Note that this diagram is just an abstract view of the scheduling of a task taking place, the discussion about the actual design and implementation of the scheduler is in later sections.

In step 1, a task, which requires a single argument is created by a dataflow thread t1 running on core 1. The task is sent to the scheduler, which maintains the information about this created task in its pending queue. In step 2, f1.arg provides the required argument to the waiting task. After receiving the arguments the task now has all its input data and is ready to execute. The scheduler moves the task from the pending to the ready queue making it available to be scheduled to any core. In step 3, a request is sent by core 2 to the scheduler for a ready task. When a scheduler gets this request, it sends a task from the ready queue to the requesting thread (this is also in line with work proposed by UNISI, UCY, cf. D6.2 D6.3).

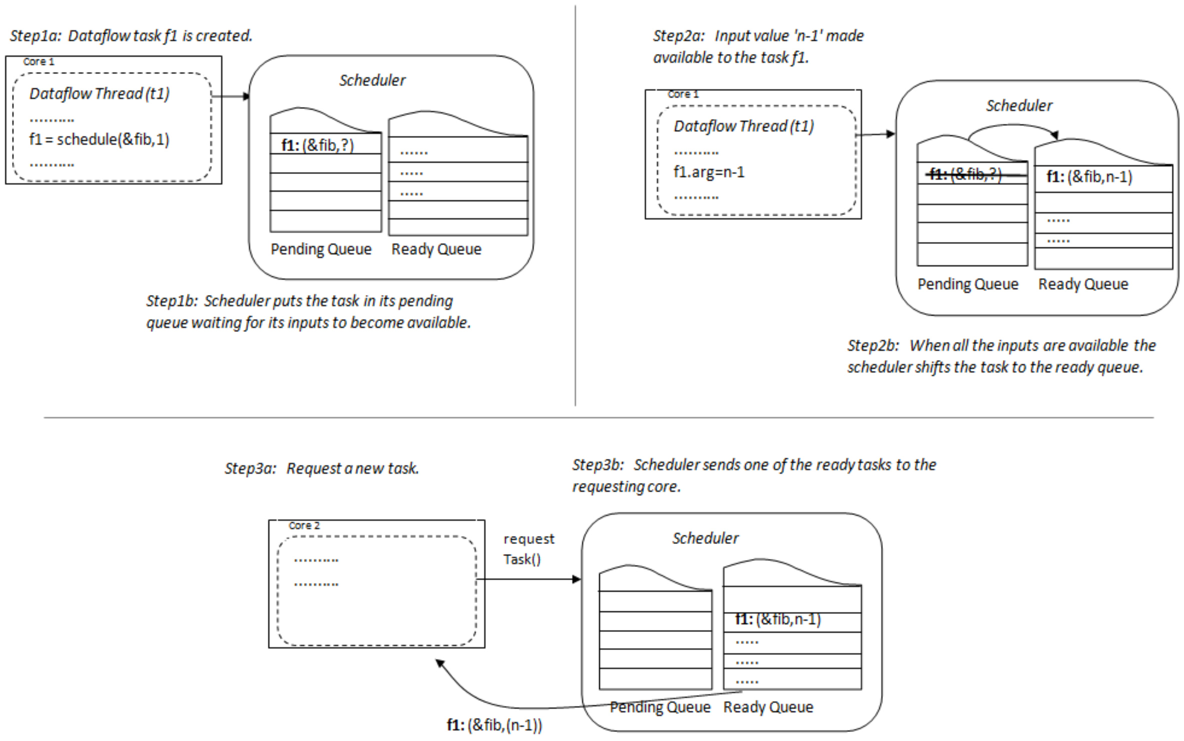

Step1a: Dataflow task f1 is created.
Core 1
Dataflow Thread (t1)
f1 = schedule(&fib,1)
Scheduler
f1: (&fib,?)
Pending Queue
Ready Queue
Step1b: Scheduler puts the task in its pending queue waiting for its inputs to become available.
Step2a: Input value 'n-1' made available to the task f1.
f1.arg=n-1
f1: (&fib,?)
f1: (&fib,n-1)
Step2b: When all the inputs are available the scheduler shifts the task to the ready queue.
Step3a: Request a new task.
Step3b: Scheduler sends one of the ready tasks to the requesting core.
Core 2
request Task()
f1: (&fib,(n-1))

Figure 15: Dataflow Scheduling (abstract view)

NOVEL SCHEDULING POLICIES

In this section we describe how the structure provided to programs by the dataflow programming model can be exploited to provide useful information to the task scheduler. By using this information the scheduler is aware of a task's data requirements and can make better decisions. We propose two scheduling policies and discuss potential performance advantages that can be gained from them.

Token Scheduling

This scheduling strategy relies on the assumption that if a reference to a data structure is passed from a task running on core x, the probability of that structure is cached by core x is high. This only works for passed references as passed values have already been copied in the process of passing them so do not allow for data reuse.

Consider the example shown in Figure 16. In step 1a, task running on core1 passes 2 arguments to task t2. In step 1b, core2 sends an argument to task t2 thus making it ready for execution. As the scheduler knows where data for the task came from, it assigns a processor affinity to task t2, in this case core 1 based on the proportion of references it received from each processor.

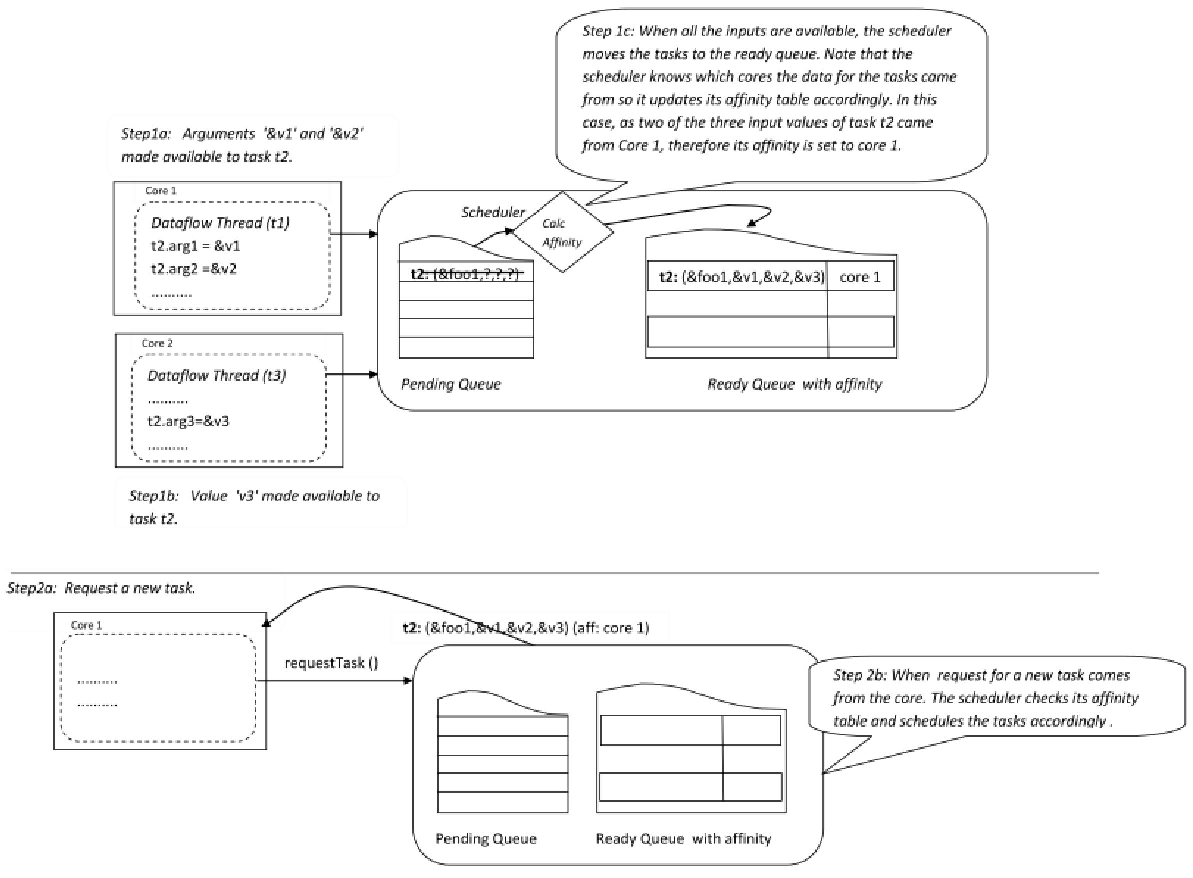

Step 1c: When all the inputs are available, the scheduler moves the tasks to the ready queue. Note that the scheduler knows which cores the data for the tasks came from so it updates its affinity table accordingly. In this case, as two of the three input values of task t2 came from Core 1, therefore its affinity is set to core 1.
Step1a: Arguments '&v1' and '&v2' made available to task t2.
Core 1
Dataflow Thread (t1)
t2.arg1 = &v1
t2.arg2 =&v2
....
Scheduler
Calc Affinity
~~t2: (&foo1,?,?,?)~~
t2: (&foo1,&v1,&v2,&v3)
core 1
Core 2
Dataflow Thread (t3)
....
t2.arg3=&v3
....
Pending Queue
Ready Queue with affinity
Step1b: Value 'v3' made available to task t2.
Step2a: Request a new task.
Core 1
....
....
t2: (&foo1,&v1,&v2,&v3) (aff: core 1)
requestTask ()
Step 2b: When request for a new task comes from the core. The scheduler checks its affinity table and schedules the tasks accordingly .
Pending Queue
Ready Queue with affinity

Figure 16: Token Scheduling

When a core requests a new task in step 2a, the scheduler looks at its list of ready tasks and then assigns the task which has the highest affinity value for that processor as shown in step 2b.

Token scheduling does not base the affinity of a task on the core where it was spawned but records where the data of the task is coming from and based on that information assigns an affinity. This technique is explored besides other miss-reduction techniques like CacheFlow (cf. Section 3.1.1.3.3 or [6]). We are not making any claim on the effective presence of data in the assigned core cache.

Reference Scheduling

In reference scheduling, instead of the task recording where its inputs came from, the scheduler records which references each core has recently accessed and tasks are assigned to the cores by matching the set of references used by the task to the cores reference history. This is explained below.

Figure 17 shows how the reference history is obtained and maintained by the scheduler using the task information present in the programming model. When task t2 becomes ready, the scheduler compares references passed to it in its argument list to the reference history of the cores. In this case the scheduler sees that the reference to data structure X is present in core 1's history, so it assigns the task t2 to core 1.
Later on when the task t2 is sent to core 1 for execution, the reference history of core 1 is updated so that it now contains the reference to structure Z as well. The reference histories do not need to be large and are bounded to a very small storage requirements to reflect that older items are less likely to still remain in the cache by decaying items in the history until ultimately they are evicted.

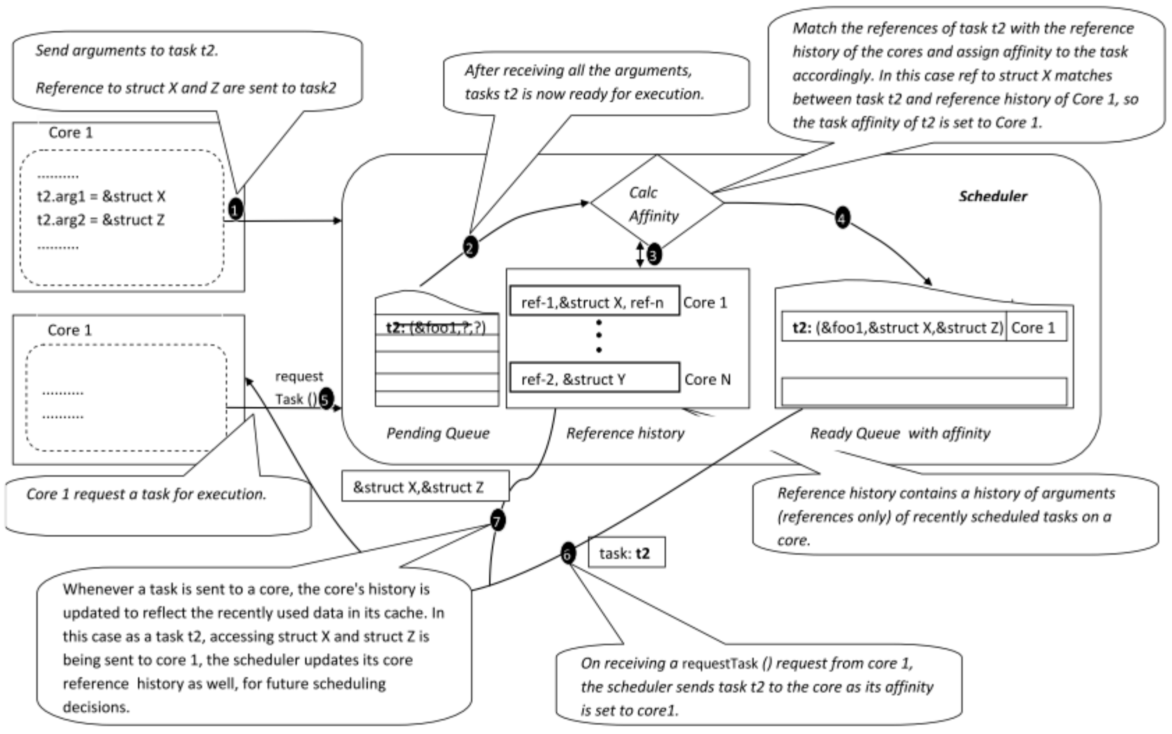

Send arguments to task t2.
Reference to struct X and Z are sent to task2
Core 1
t2.arg1 = &struct X
t2.arg2 = &struct Z
After receiving all the arguments, tasks t2 is now ready for execution.
Match the references of task t2 with the reference history of the cores and assign affinity to the task accordingly. In this case ref to struct X matches between task t2 and reference history of Core 1, so the task affinity of t2 is set to Core 1.
Calc Affinity
Scheduler
t2: (&foo1,?,?)
ref-1,&struct X, ref-n
Core 1
ref-2, &struct Y
Core N
t2: (&foo1,&struct X,&struct Z)
Core 1
Core 1
request Task ()
Pending Queue
Reference history
Ready Queue with affinity
Core 1 request a task for execution.
&struct X,&struct Z
Reference history contains a history of arguments (references only) of recently scheduled tasks on a core.
task: t2
Whenever a task is sent to a core, the core's history is updated to reflect the recently used data in its cache. In this case as a task t2, accessing struct X and struct Z is being sent to core 1, the scheduler updates its core reference history as well, for future scheduling decisions.
On receiving a requestTask () request from core 1, the scheduler sends task t2 to the core as its affinity is set to core1.

Figure 17: Reference Scheduling

ARCHITECTURAL SUPPORT

Distributed Task Queuing

A centralized queue is the simplest way to implement task queues in scheduling, and the block diagrams demonstrating our scheduling policies showed centralized queues for simplicity. In a centralized system, all the tasks are enqueued and dequeued from a single shared queue. While this is sometimes acceptable, a single queue can quickly become a bottleneck as the number of cores scale up. To address this bottleneck and allow better throughput and latency times, software and hardware schedulers often use distributed tasks queues with task stealing [24] [25] [26] [27] [28]. Figure 18 shows the distributed queue structure we propose for the scheduler in order to group the tasks based on their affinity to specific processors.

Design

Our design provides for low overhead distributed task queues and is tolerant to increasing on-die latency as the number of cores in the system scales. This is achieved by implementing the distributed task queues in the hardware. The tasks are stored in the hardware queues, scheduling is implemented in hardware and we have hardware task prefetchers so that each hardware thread can start a new tasks as soon as it finishes its current one. The design has also many similarities with Carbon [22], DTA, DDM, TaskSuperScalar (cf. D6.1, D6.2, D6.3).

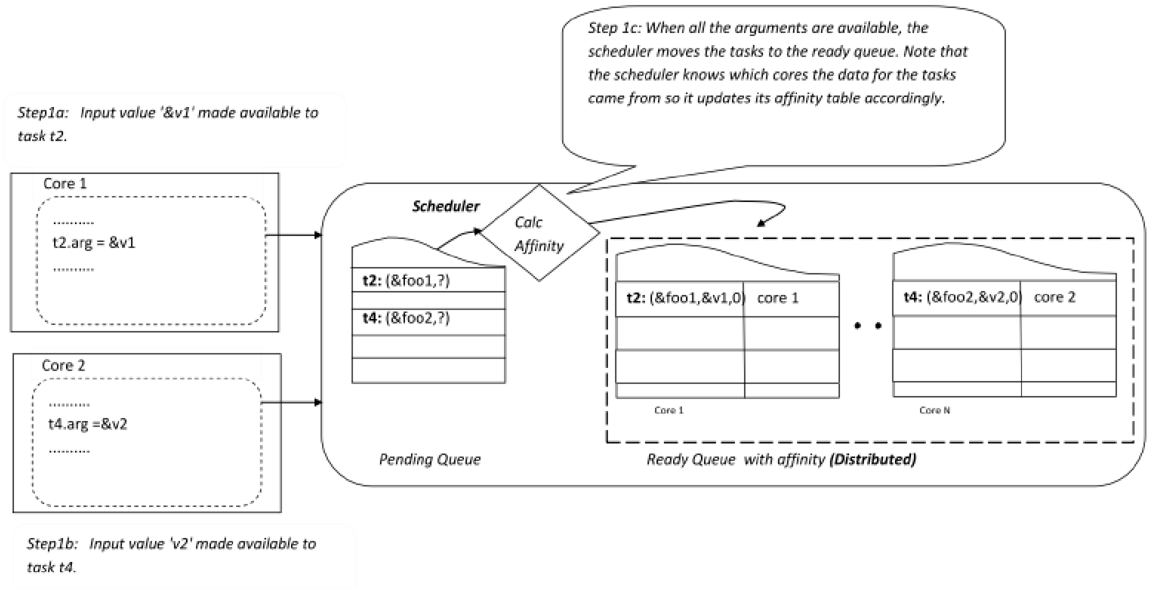

Step1a: Input value '&v1' made available to task t2.
Core 1
t2.arg = &v1
Core 2
t4.arg =&v2
Step1b: Input value 'v2' made available to task t4.
Step 1c: When all the arguments are available, the scheduler moves the tasks to the ready queue. Note that the scheduler knows which cores the data for the tasks came from so it updates its affinity table accordingly.
Scheduler
Calc Affinity
t2: (&foo1,?)
t4: (&foo2,?)
t2: (&foo1,&v1,0) core 1
t4: (&foo2,&v2,0) core 2
Core 1
Core N
Pending Queue
Ready Queue with affinity (Distributed)

Figure 18: Distributed token scheduling

From the tasks queue hardware perspective, a task is simply a tuple [22] (or “continuation”). In this implementation, it is a tuple of four 64-bit values; a function pointer and pointers to shared data passed to a task as arguments. Similarly the core reference history is also a tuple of 64bit values. Reference histories can be implemented as bloom filters [29] to make the comparison between task reference and history references cheaper, thus making the hardware for comparison much simpler, quicker and more energy efficient. The hardware task queues have limited capacity, in order to support a virtually unbounded number of tasks for a given processor, and to support virtually unbounded number of processors, we can extend our model to move tasks out of the hardware tasks queues into the memory system using a mechanism similar to [22].

According to the TERAFLUX architectural template, our design considers a CMP where the cores and last-level caches are connected by an on-die network. This design has two main components: a centralized global scheduling unit (GSU, similar to a single D-TSU) and a per-core local scheduling unit (LSU, i.e., an L-TSU). Figure 19 shows our design.

Global Scheduling Unit (GSU). The Global scheduling unit holds enquired tasks in a set of hardware queues with one queue per core in the system. This could be extended to implement a hardware queue per hardware context. The global scheduling unit implements the task scheduling policies described earlier. Since the queues are physically located close to each other, the communication latency between the queues is minimized.

Local Scheduling Unit (LSU). Centralized scheduling systems may not scale with the number of cores in the system. This issue is addressed with Local Scheduling Units.

Each core in the system has a local scheduling unit that provides an interface between a core and the GSU. The LSU is used to hide the latency of dequeueing a task from the GSU by buffering a small number of ready tasks. The LSU includes a task prefetcher and a small task prefetch buffer (similar to CacheFlow [6]). Without the LTU, if a thread sends a task request to a GSU it will stall waiting for the response from the GSU.

When a task is requested by the core's hardware thread, the LSU returns a task to the core and sends a prefetch request for the next task to the GSU. The LSU buffer should be large enough to hide the latency of accessing the GSU. In our benchmarks we find that buffering a single task is sufficient to hide the GSU dequeue latency. Because the dataflow graph has provided in advance the information about which references a task will use, a buffer of 1 can be implemented without loss of precision.

Benchmarks and Evaluation:

As our concern is the effect of scheduling on locality, precision in the micro architectural model was not necessary, so we model comparatively simple in-order x86 processors, accompanied by MESI coherence protocol. While we vary the number of processors simulated, all the experiments were carried out with 64KB private L1 data caches and a 2 MB unified private L2 caches. All caches were 4-way set associative. Figure 20 summarizes the base system configuration.

For experiments with hardware scheduling, we add the hardware described in the previous sections. We applied a 20 cycle delay for an access (e.g., enqueue or dequeue) to the global task unit. This is in addition to the latency for the cores to message the GSU over the on-chip network. Within this simulator we implemented the five different scheduling policies: Random, FIFO, Source, Token and Reference Scheduling. These different strategies represent increasing levels of complexity for the scheduler. Our proposed policies of Token and Reference scheduling are already explained in detail in the previous sections. Here we will briefly discuss the other scheduling policies with which we will be comparing our proposed schemes:

- ***Random***. Each processor is randomly assigned a task from a set of tasks currently available to run. If there are more processors than available tasks, processors are selected at random to assign the available tasks to. This strategy is included to demonstrate that any improvements are not simply because we are introducing an element of randomness to the scheduling.

- ***FIFO***. It is our baseline and schedules tasks strictly in the order that they become ready.

- ***Source Scheduling***. It is a strategy that can take advantage of programs which are split into distinct parts. With this strategy, cores will preferentially run tasks created by other tasks on that core. This approach is similar to one used by Carbon [22].

- Token Scheduling, as explained before.

- Reference Scheduling, as explained before.

Benchmarks

To test the effects of our scheduling policies in the scheduler, we used a set of six benchmarks:
a) Block matrix multiplication, b) iterative refinement for motion estimation, c) index searching, d) route planning, and e) and f) two versions of kmeans. If we constrain these relatively small examples to the dedicated hardware available in HPC it is in principle possible to manually achieve the same potential using conventional solutions, however this makes solutions that are fragile and may not exhibit performance portability. As programs increase in size and become more complex, or in more general scenarios where resources are not dedicated, but are shared with other programs, effective hard coding become untractable. The hard coding of strategies into the program also assumes that the programmer is able to correctly determine the appropriate strategy, real world problems are often too complex especially when such problems include input that is outside of the programmers' control.

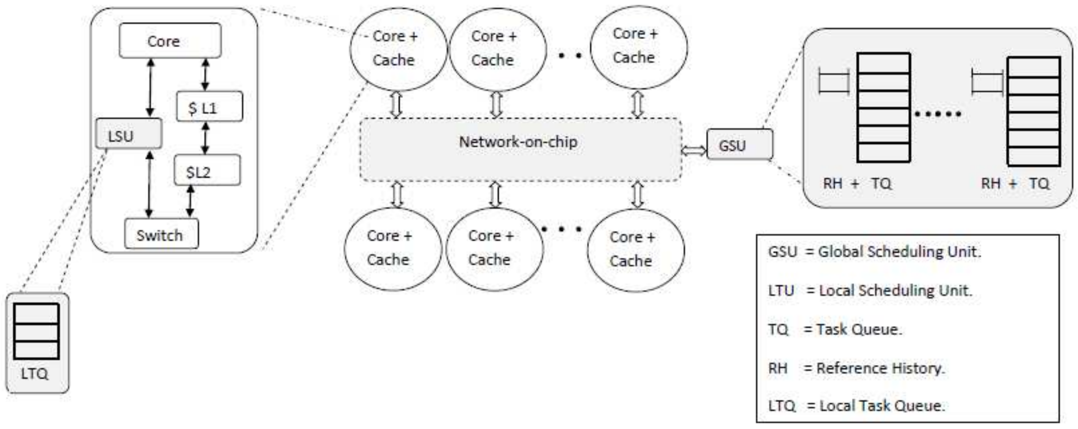
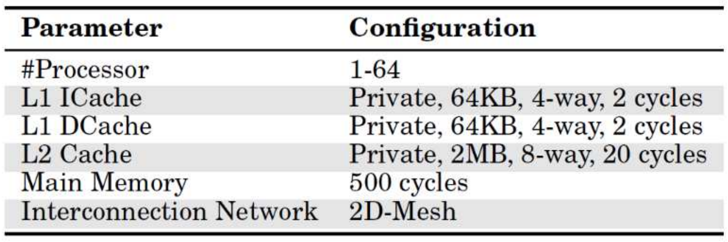


Figure 19: Example of many-core system with our hardware support for task queues. The shaded portions are additional hardware for the scheduler

Results

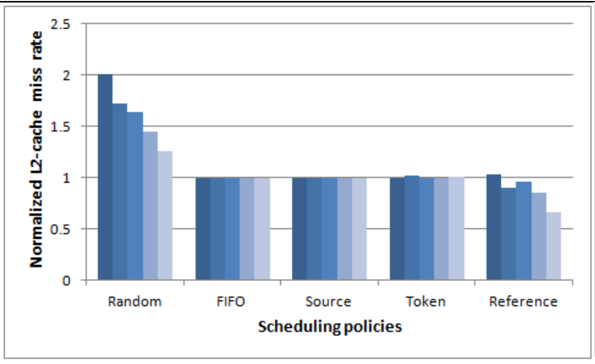
When evaluating the scheduling policies, we observe improvements in locality, measured through cache misses which can be seen in [21]. This shows the number of L2 misses as a percentage of those seen when using FIFO scheduling. We see that for all benchmarks our Reference and Token scheduling policies perform at least as well as the best alternate policy. In case of Routing benchmark, the reference scheduling policy reduces the cache misses to 30% of the FIFO level on 32 cores. On the average the Reference scheduling policy reduces the L2 cache misses to about 50% of the FIFO level. The advantage of the Reference scheduling policy is that the scheduler knows exactly which data was most recently sent to which processor, which as expected results in the best data locality.



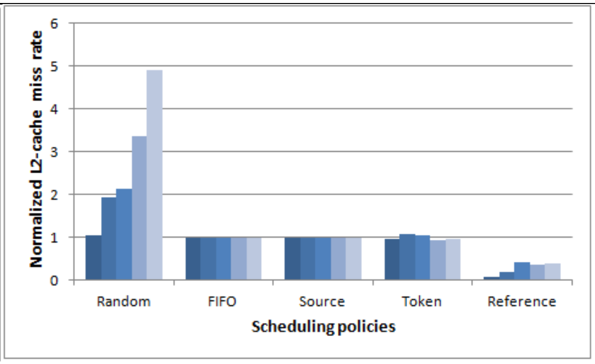
Parameter	Configuration
#Processor	1-64
L1 ICache	Private, 64KB, 4-way, 2 cycles
L1 DCache	Private, 64KB, 4-way, 2 cycles
L2 Cache	Private, 2MB, 8-way, 20 cycles
Main Memory	500 cycles
Interconnection Network	2D-Mesh

Figure 20: Architectural Parameters used

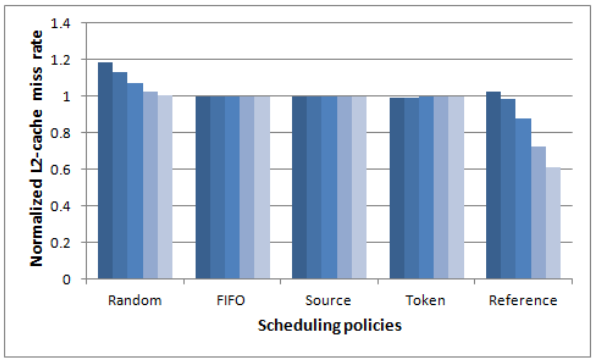
The Source policy which prioritizes the processor where a given task was created shows almost identical performance to FIFO. This policy relies on the observation that a tasks children are more likely to share data requirements than a random task elsewhere in the system. Unfortunately this observation fails to work for a range of models including those that converge to a single task to perform some control logic before returning to work on the dataset. Examples of this model include MapReduce [30] and the ForkJoin [31] frameworks. Random, as expected, performs progressively worse as the number of processors increases. This is because the probability of finding a good schedule by chance decreases as the number of processors rises.



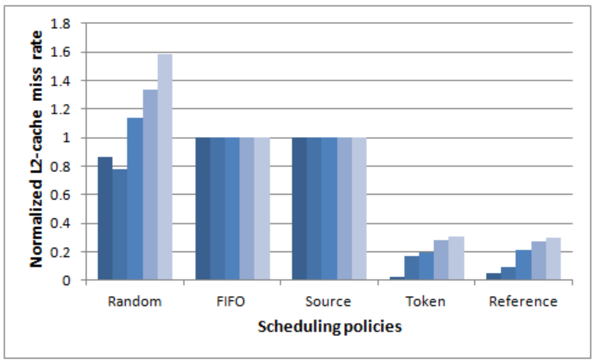
(a) Matrix Multiply



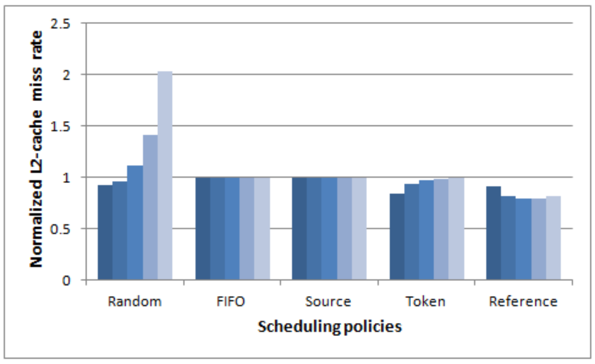
(b) Iterative Refinement



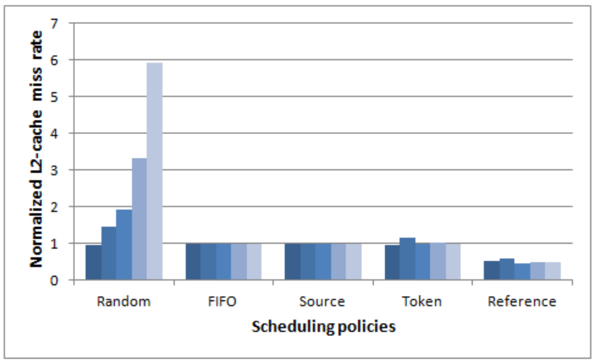
(c) Shared Index



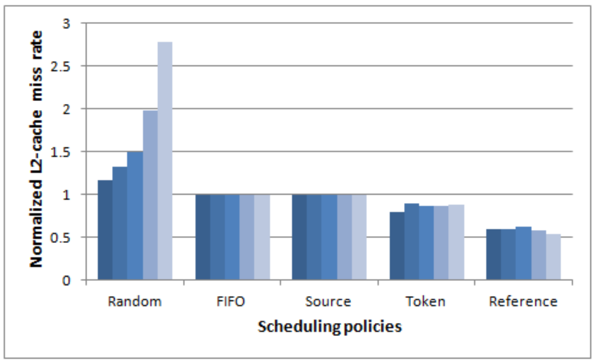
(d) Routing



(e) Kmeans-a



(f) Kmeans-l



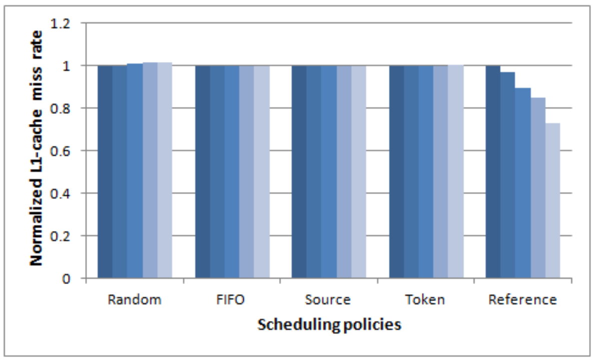
(g) Average

Number of Cores

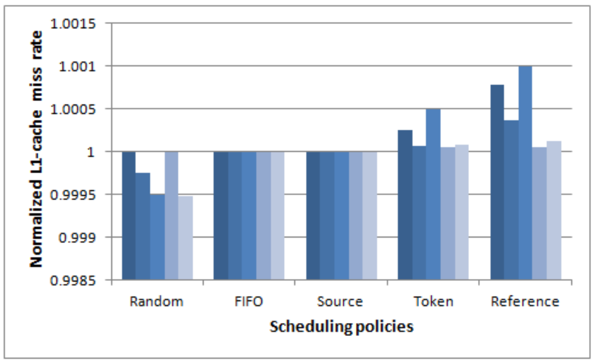
- ▨ 2
- ▨ 4
- ▨ 8
- ▨ 16
- ▨ 32

Figure 21: Number of L2 misses, as a percentage of those seen with FIFO scheduling

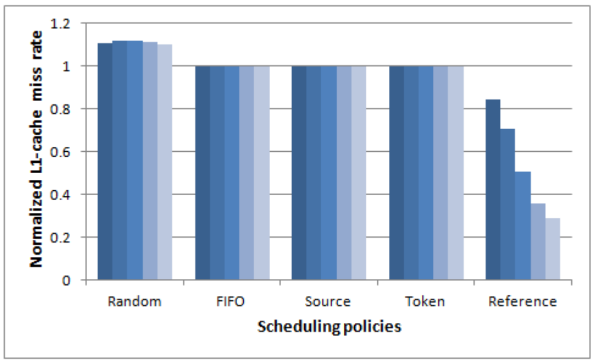
In Figure 22, we see the reduction in L1 cache misses for four of the benchmarks. Iterative refinement and Routing show no significant effect on the L1 cache misses from changing scheduling policy. This is because the inputs for threads are too large to fit in the L1 cache. This emphasizes that any scheduling policy will only obtain advantage from locality if the data is partitioned such that it remains in the cache between threads. Clever scheduling does not obviate the need for appropriate partitioning of the problem.



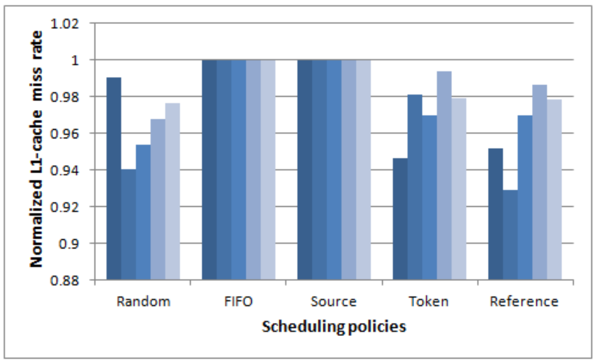
(a) Matrix Multiply



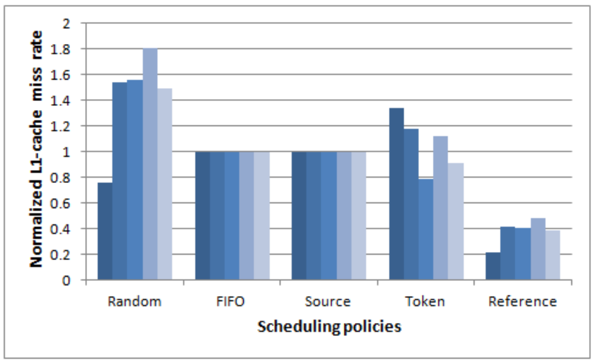
(b) Iterative Refinement



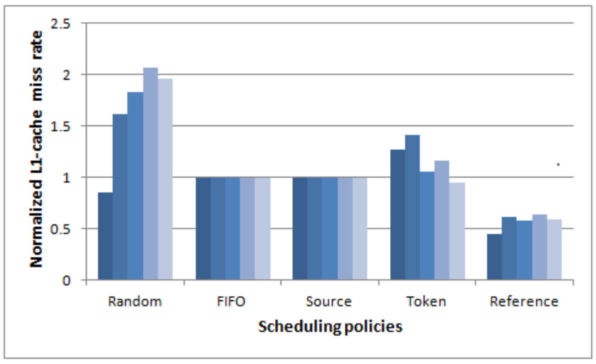
(c) Shared Index



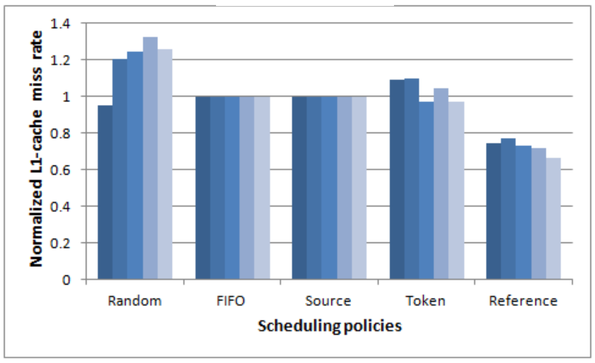
(d) Routing



(e) Kmeans-a



(f) Kmeans-l



(g) Average

Number of Cores

▨ 2

▨ 4

▨ 8

▨ 16

▨ 32

Figure 22: Number of L1 misses, as a percentage of those seen with FIFO scheduling

3.2 Advanced Scheduling Mechanism

3.2.1 Dynamic Scheduling (UCY)

One relevant component in the Abstraction Layer is the ability for threads to be mapped dynamically to cores. This can be used for example to avoid, dynamically at runtime, to schedule a thread to a faulty core, or to a core that is near its thermal threshold.

For the DDM-Style execution, the TSU++ (see Section 3.1.1.1) runtime implements a number of scheduling policies that permit the programmer/compiler to control the mapping of threads to the cores. The scheduling policies are assigned per-thread allowing for maximum flexibility. The supported scheduling policies are as follows:

- **Dynamic:** The default policy distributes the threads invocations among the cores in a way that maximizes load-balancing. We denote this policy as the dynamic scheduling policy. It takes the load status in consideration and so in the case of scheduling threads with similar execution durations, it provides optimal performance.
- **Static:** The static policy distributes the invocations of a specific thread to a specific core.
- **Round-Robin:** The round-robin policy distributes the invocations of threads across the cores in a round-robin fashion. It requires no information of the core status and aims at distributing the threads among the cores uniformly.
- **Modulo:** The modulo policy uses the context, uniquely distinguishing each thread invocation modulo the number of cores to select the target core.

3.2.1.1 Scheduling Example: Dynamic and Round-Robin Policies

Figure 15 demonstrates the dynamic and round-robin scheduling policies used for an arbitrary program with multiple different threads. As time is relevant in the case of these two policies, the length of the threads rectangles represent their execution duration. The vertical distance between the dependency arrows start and end represent the time needed for performing the following tasks: informing the TSU that a thread has finished, decrementing the Ready Count (RC) of its consumers, fetching the data of ready threads and eventually inserting the thread in the Firing Queue (FQ). For simplicity we assume that these steps are accomplished in a constant time and that the scheduling decision is made mid-way. The empty dashed rectangles represent waiting time for an execution unit to become available.

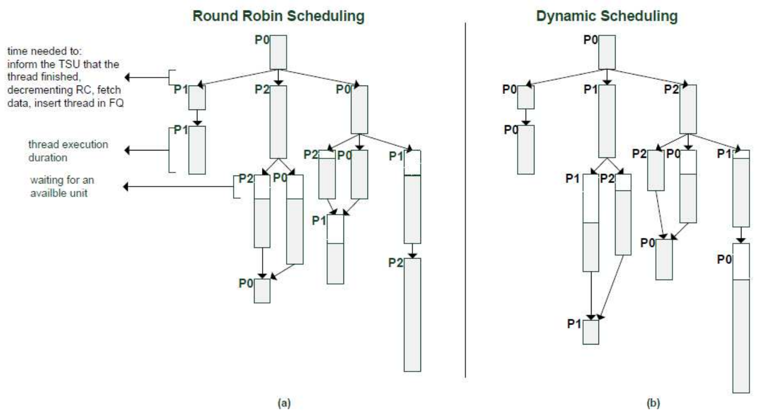

Round Robin Scheduling
Dynamic Scheduling
time needed to: inform the TSU that the thread finished, decrementing RC, fetch data, insert thread in FQ
thread execution duration
waiting for an availble unit
P0
P1
P2
(a)
(b)

Figure 23: Dynamic and Round-Robin Scheduling Policies

In part (a) of Figure 23 the round-robin policy is used for all the threads. This policy keeps a common counter value initialized to zero (i.e. the id of the first core) and every time a scheduling decision is required the current value of the counter is returned as the core identifier and the counter is then incremented by one. In part (b) of the figure the dynamic scheduling policy is used for all the threads. This policy selects the core with the least load, which promotes better load-balancing. Both policies result in different schedules as shown in the figure.

3.2.2 Changes in the Hardware Task Superscalar task scheduler structure in order to simplify the overall system network and reduce system stalls (BSC)

As explained in section **Error! Reference source not found.** our previous design of a Hardware Task Superscalar Architecture [13] has been improved. The new hardware has been called Picos. In order to dimension the new hardware, a space design exploration has been performed to find the amount of necessary resources to fully exploit current and future many-core architectures. As a result of this work, the resource utilization of the system has been reduced while the task scheduling throughput has been increased. The resulting proposed configuration (called High Performance Configuration, HPCConf) for a Picos Hardware machine is composed by 10 modules (whose functionality is explained in section 3.1.2): 1 Gateway, 4 TRSs, 4 DCTs and 1 TS. Each TRS has a 256-entry TM. Each DCT module has 2 memories: the VM is an indexed array of 512 entries while the DM is an 8-way set associative memory with 64 entries (to also amount a total of 512 entries). That configuration is able to efficiently manage real applications with huge potential parallelism.

Figure 24 and Figure 25 show a comparison of the benchmark performances when using the aforementioned configuration (HPCConf) and the performance of the OmpSs benchmark versions using Nanos++ runtime for Cholesky and LU problems. OmpSs results are for a machine with 12 cores at 2.4 GHz, using one thread per core. The Figure shows in the Y-axis the speed-ups obtained against the sequential execution when we change the number of threads (X-axis) and the parallel approach (the block size). The executions shown in each graph solve the same problem: a Cholesky and a LU application for a 2048 problem size (matrix), but with different block sizes. Each bar label shows the selected block size (up to 1024) and if it has been obtained executing with Nanos++ (Nanos bars) or simulated with Picos hardware (Picos bars). To avoid the variability of comparing different executions, all the tests have been executed three times and the best results have been chosen. Also it is important to note that while Nanos++ real executions are influenced by the parallel memory behavior of the application, Picos results are based in a sequential execution that can exhibit a different memory behavior.

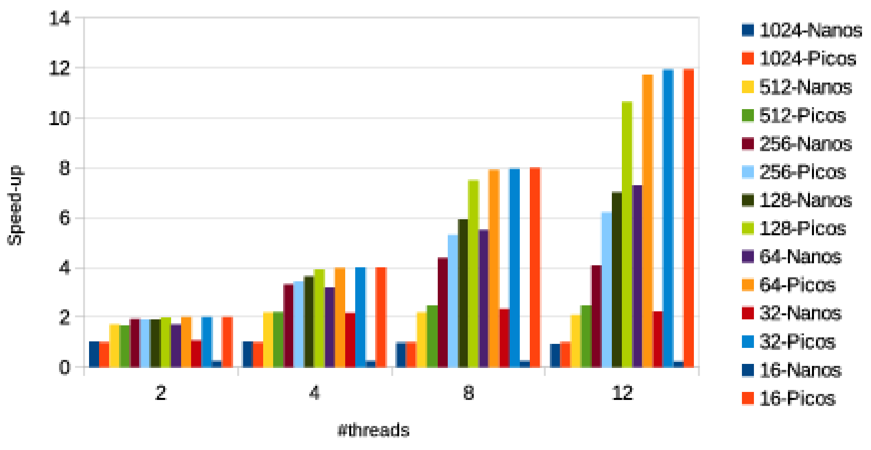


Figure 24: Comparison of Nanos++ and Picos with different number of threads and tasks for block Cholesky application with matrix size 2048x2048 (each bar is labelled with the block size).

In Figure 24 and Figure 25 it can be seen that when the parallelism is increased (bars with diminishing block sizes) Nanos++ and Picos take advantage at the increasing number of tasks (Cholesky with a matrix size of 2048 and a block size of 1024 has only 4 tasks while with the same matrix size and a block size of 16 has 357760 tasks). However as the task granularity diminishes (the problem size is the same in all the executions) the overhead introduced by the software runtime scheduler starts to introduce diminishing returns in the obtained speed-up. This effect can be observed in the last execution configurations in Figure 24: bars 64-Nanos, 32-Nanos and 16-Nanos. The Picos hardware, on the other side, can take profit of the parallelism of the application regardless of the parallelism granularity and, in fact, the more aggressive the parallelism, the better Picos exploits it. This behavior is really desirable as it decouples the application parallelization approach from the hardware in which it is going to be executed, making parallel programmers' life easy.

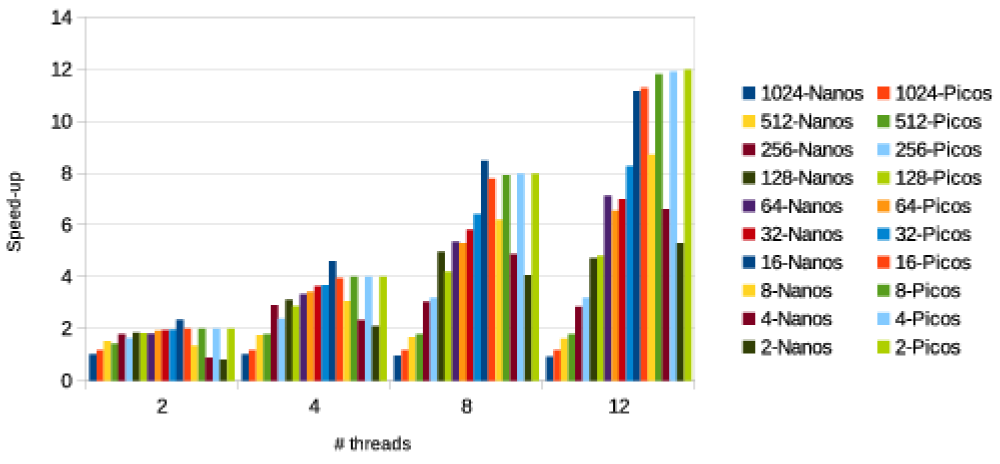


Figure 25: Comparison of Nanos++ and Picos with different number of threads and tasks for block LU application with matrix size 2048x2048 (each bar is labelled with the block size).

Figure 25 shows another interesting effect: superlinear speed-up for the LU parallelization with block size 16. This effect can be seen in bars 16-Nanos with two threads (2.34x), with four threads (4.59x) and with eight threads (8.58x). Note that the superlinear speed-up is measured in real executions using the software OmpSs approach not developed in this project. It can be explained by the memory access pattern of the LU application that creates sequences of dependent tasks that access the same memory locations and are executed in the same physical core (allowing very good cache behaviors). This effect couldn’t be achieved in a sequential execution where the control-flow order of execution of the tasks prevents two consecutive ones from accessing the same memory locations. This superlinear speed-up makes Nanos++ perform better than the hardware for this case. The problem is that this effect cannot be observed in Picos because its results are note executed but are extrapolated from the sequential execution. However, the same behavior is expected to occur in a real machine allowing the hardware to be at least as good as the software.

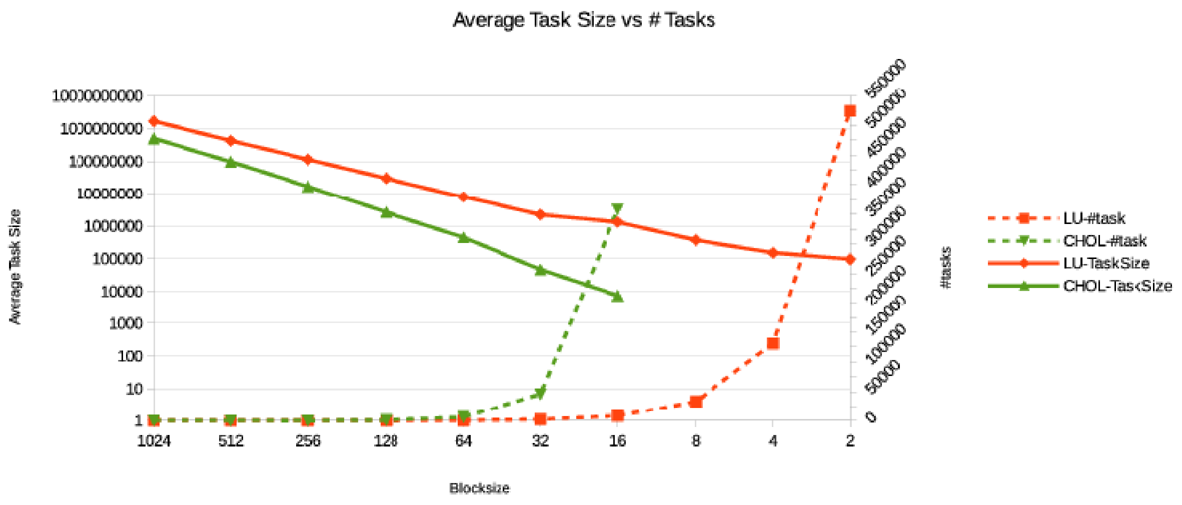


Figure 26: Number of tasks and average task size in cycles of block Cholesky and LU applications both executed with a matrix size of 2048x2048 as a function of the block size.

Figure 26 shows the number of task instance executions (right axis) and the average task size in cycles (left axis) of the executions in Figure 24 and Figure 25 as a function of the block sizes. As it can be seen comparing the three figures, the software approach suffers not only when the tasks are small but also when the number of tasks grows exponentially. The hardware, on the other side, transforms its limited memory storage drawback in an advantage. Picos hardware keeps obtaining good results as it only maintains a limited number of in-flight tasks at the same time, but processing them very fast.

Another interesting side effect of using the Picos hardware instead of the software approach is that the hardware doesn’t suffer from contention when the number of threads increases. This effect can be seen comparing 12-thread and 8-thread bars for block size 256 in Figure 24. While bars 256-Picos show that the hardware takes profit of the increasing in available resources, bars 256-Nanos show that the runtime obtains less speed-up with more resources. The reason for this different behavior is the decoupled design of the hardware that allows it to work in parallel in the different dependence chains that the application generates avoiding contention caused by shared data structures.

In fact, taking the above example of contention to the limit to better illustrate it, Cholesky with a matrix size of 2048 and a block size of 64 has a maximum speed-up of 86× and the selected configuration can extract a speed-up of up to 72× with 256 workers. Note that in this experiment both the problem size and the parallelism approach are different from the ones presented in section 3.1.1.1 where a matrix size of 32768 and block sizes of 256 and 128 were used. As a result the experiment presented in this section has smaller tasks and a lower maximum theoretical speed-up. An even more parallel configuration (with eight TRS and eight DCT modules) with double number of workers can scale up to 83×. This effect is fully shown in Figure 27 where it can be seen, for the selected parallelization of the chosen applications (both of them solving a 2048×2048 problem size), the speed-up obtained when they are executed in a system environment with 256 workers using the selected HPCConf. For the sake of comparison, Figure 27 also shows the best speed-up that can be obtained for these traces ("Ideal" bars) and the improvement that will result when using 512 workers, with a doubled configuration (that is the same as the HPCConf but doubling the number of TRS and DCT modules, labelled "2x HPCConf"). Figure 27 shows that the selected configuration reaches speed-ups close to the ideal for all the benchmarks. For the most demanding applications this speed-ups can still be improved by simply increasing the number of modules in the system showing that even for very aggressive machines and demanding applications the decoupled Picos system would be able to deal with the challenge.

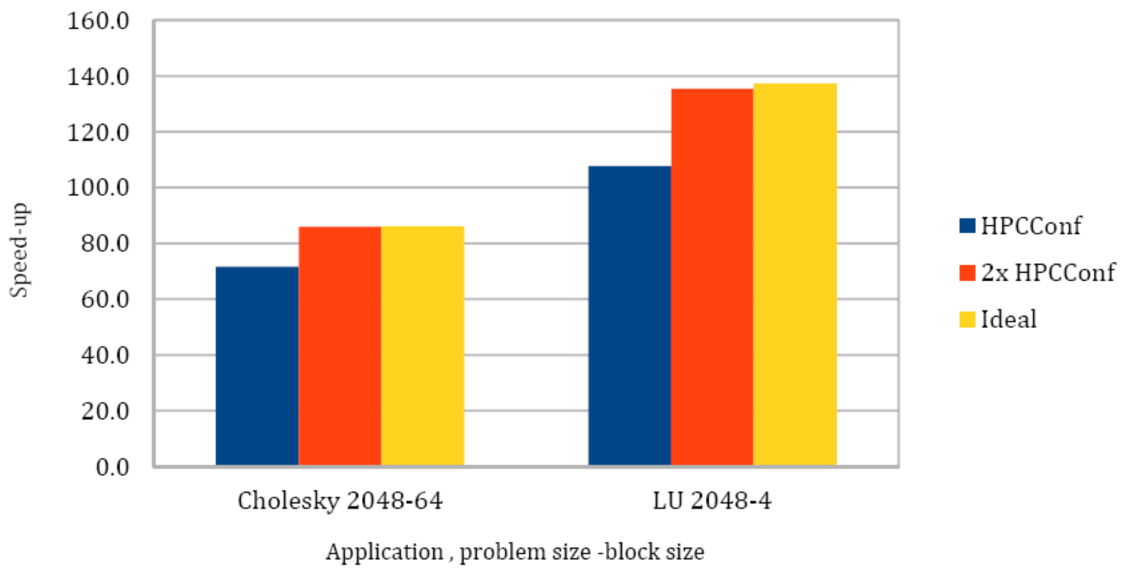


Figure 27: Picos Hardware performance for Cholesky and LU applications.

Regarding the Picos hardware the results obtained show that the runtime task management hardware approach is much more efficient than the software alternative (Nanos++ runtime of OmpSs) for the selected applications when they are divided in several small tasks. Furthermore, the hardware approach efficiently decouples the parallelization applied to the applications from the resources (physical threads) used in performing the computation allowing the applications to be easily optimized for a wide range of target platforms. These advantages are due to two main factors: the speed at which the hardware can manage the tasks dependencies and its decoupled design that allows different processes (as chaining dependencies while sending tasks to execute) to be performed in parallel. Both, the minimized overhead and reduced contention imply that smaller tasks can be executed efficiently, providing a suitable bases for building and exploiting next-generation many-core architectures.

3.3 Advanced Memory Management

3.3.1 Memory consistency mechanisms (UNISI)

TERAFLUX requires several different consistency models (cf. D7.1, Section 7), including:

- Frame Memory – FM;
- Thread-Local Storage – TLS;
- Code Memory – CM;
- Owner-Writable Memory – OWM (which, as discussed below, could also be used to implement Frame Memory, Thread-Local Storage, and Code Memory);
- Transactional Memory – TM;

There are at least two memory-related operations in the system that would require consistency-model communication across the chip (and, depending on the chosen implementation of transactional memory, more operations could be required):

- **region-acquire**:
  - a region is acquired by the writer;
  - all other potential readers/writers of that region are notified, their write permissions are removed, but their read permissions are *optionally* removed;
- **region-release**:
  - any suspended read permissions are restored;

In the exact same way that one can choose the placement of the TLB and thus the level to which address (re-)mappings must be broadcast, so, too, one can choose the level to which these acquire/release consistency operations must be broadcast. Simply put, they must be broadcast to the level at which they are expected to take effect [32]. For instance, the following types of access protections are used in many architectures to decide whether a load/store operation is allowed to progress:

- readable;
- writable;
- executable;
- kernel-owned;

If each of these is checked and enforced at the *core level*, then it is to the core level that all acquire/release events must be broadcast (assuming, for the sake of argument, that access-level protections are the mechanism by which one grants or denies read/write access to shared regions). This, of course, can be optimized by delivering the information to the node level, and designing the node keep track of usage within itself (e.g., through frame metadata in the TSU, providing *de facto* back-pointers), thereby requiring “selective” broadcast to the core level. This would still be considered a core-level broadcast, as opposed to delivering information to each node and having the process stop right there.

By comparison, using a *node-level* access mechanism would require no information to pass higher than the node, and thus individual cores would not see any changes in consistency status. This design

choice would necessarily allow each core to freely access the data in its local caches, pausing only to check when accesses miss that local cache. Thus, it would not support owner-writable memory in which read operations to a shared region cause protection violations prior to the release of that region.
The following illustration indicates the level to which memory-consistency information propagates and shows the implications for each design choice (L2/TSU refers to what we called D-TSU in the TERAFLUX architectural template, cf. Figure 1).

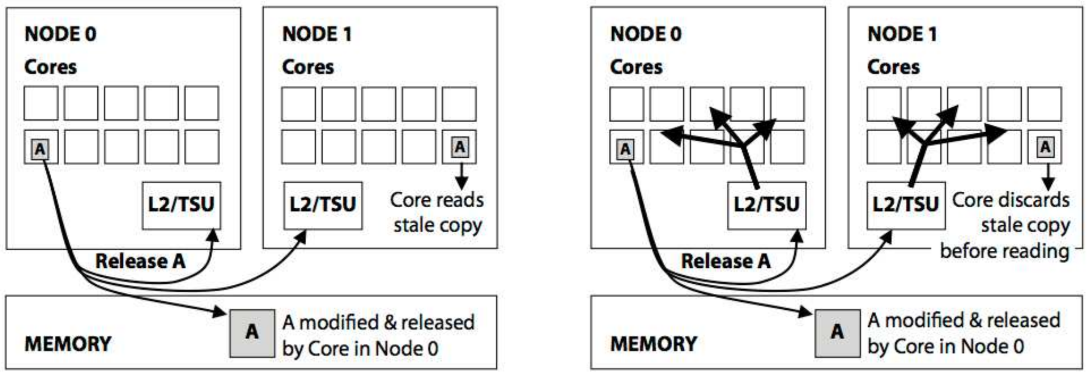


Figure 28: Propagation of memory-consistency information. On the left, consistency information is propagated to the node/TSU level. Therefore, after the core in Node 0 releases its copy of A, the core in the Node 1, which has a cached copy, continues to read a stale copy from its cache. On the right, all consistency information is propagated to the individual cores. Therefore, after the core in Node 0 releases its copy of A, the core in the Node 1 discards its cached copy and obtains a new copy.

As Figure 28 shows, broadcasting only to the Node/TSU level is simpler, requires less overhead (in terms of both performance and power), but can result in core-level inconsistencies. However, the only way that this particular inconsistency can happen is if cached copies are allowed after the owner-core in Node 0 acquires a writable copy of item A. Thus, the problem is avoided if one can ensure that, at the time of the acquire operation by the Core in Node 0, no cached copies exist anywhere in the system. The implication is that, to avoid the inconsistency, one must do one of two things: (1) broadcast at the core level the *availability* of an updated copy of item A upon its *release*, or (2) broadcast at the core level the *unavailability* of item A at the time of the *acquire* operation. Either way, one performs a core-level broadcast.

Most likely, the ability to *optionally* disseminate consistency beyond the node level would be ideal, so that one could use the same scheme to implement Frame Memory, Code Memory, and Owner Writable Memory. The following subsection describes a mechanism that would do exactly this, using the TSU system in an operator-overloaded manner (i.e., use core-level and node-level mechanisms where appropriate to manage the protection and consistency status of the memory, for example to turn off read and write permissions for a particular region until the writer releases it, or to allow read permissions while a different core is writing a region) to handle all or nearly all of the memory consistency models required by TERAFLUX.

3.3.2 Consistency Required Mechanisms (UNISI)

Given the previous hypothesis, the following are the only mechanisms that need to be implemented in the system. Using these primitives, one can implement all (with perhaps the exception of transactional memory) of the consistency domains discussed earlier.

- TSU metadata, one such data structure to be created for each region:
  - Frame ID
  - a list of any "writers" (i.e., the threads that are producing the values needed by the reader thread, which is the thread to wake up), or **nil** … if **nil**, then this is just regular memory, such as local storage or code memory or shared memory—but it is not "Frame Memory"[1];
  - a "written" status bit for each writer[1];
  - thread owner (i.e., "reader" or thread to wake up);
  - permissions: read/write/execute/etc. for both the owner and non-owners;

- **region-acquire:** privileged operation
  arguments: id, region, writers, permissions (owner, non-owner);
  If such a frame exists, modify its metadata. If it does not exist, create it as well as its metadata. Metadata is held in the TSU. If the non-user-permissions are **nil**, then the TSU must ensure that there are no cached copies within its node (alternatively, one could create a broadcast mechanism in the **release** operation in which a "scope" is defined for the release, indicating whether it is meant to apply at only the node level, or all the way to the core level).

- **region-release:** privileged operation
  arguments: region, permissions (owner, non-owner);
  If such a frame exists, modify the permissions accordingly.

- Core L1 cache miss acquires cache-block metadata from TSU;

- If a TSU acts in concert with its local L2 cache and ensures that, for any data in the L2 cache (which must also enforce inclusion with the L1 caches), there is corresponding metadata held in the TSU, then broadcasts are simplified in the case of newly-created data frames (which would not be cached anywhere and thus should require no core-level broadcasts).

As mentioned, these mechanisms are all that is necessary to create the various forms of memory listed earlier. How each can be implemented is described in the subsections below.

1. Note that the writer list and status bit for each TSU entry can be more simply implemented through a single count value, provided that system software can ensure that a given data-producer-thread ("writer") only generates a single write reference to a given Frame ID.

3.3.2.1 Frame Memory consistency (UNISI)

On thread initialization, allocate a Frame identifying the data-producers that will wake up the thread and cause it to execute:

- *region-acquire*
  id: **thread**
  region: frame
  writers: **{set of data-producers}**[1]
  permissions: owner: read + write, non-owner: nil

This creates an entry in the TSU system with the expected information. The thread's "writers" write to the memory system, using the Frame ID as an address. The TSU system collects these memory references and updates the matching database entries by setting the corresponding **written** bits (or by decrementing the counter). Once the writers finish, the reader thread is awakened automatically by the TSU.

3.3.2.2 Thread-Local Storage consistency (UNISI)

On the creation of a thread (at initialization or later), allocate regions as follows:

- *region-acquire*
  id: thread
  region: frame
  writers: nil
  permissions: owner: read + write, non-owner: nil

This creates a read/write region owned by the specified thread, not shared with other threads.

At the end of a thread's life (as part of the thread-destroy operation), its non-persistent memory regions are de-allocated (i.e., those that are non-shared):

- *region-release*
  region: frame
  permissions: owner: nil, non-owner: nil

This destroys a region. Note that the OS must manage what regions are shared and/or persistent, so that it does not accidentally destroy a region that is meant to remain in the system.

3.3.2.3 Owner-Writable Memory consistency (UNISI)

To either create or take ownership of an existing region of memory, perform the following. There are two variants, one of which allows concurrent read access, the other of which does not. The first example assumes that other threads CAN read the region while the owner thread is writing it (the readers just get stale data).

- *region-acquire*
  id: thread

region: frame
writers: nil
permissions: owner: **read + write**, non-owner: **read**

- [write region]
- *region-release*
region: frame
permissions: owner: read, non-owner: read

This creates a read/write region (or simply changes the status of an existing region) owned by the invoking thread, and other threads are allowed to keep readable copies, but not writable copies, in their L1 and L2 caches. When the region is released, it is marked as readable by all; if afterward a thread wants write access, it must perform an acquire operation. The second example assumes that other threads CANNOT read the region while the owner thread is writing it (they get permission-violations when they try).

- *region-acquire*
id: thread
region: frame
writers: nil
permissions: owner: **read + write**, non-owner: **nil**
- [write region]
- *region-release*
region: frame
permissions: owner: read, non-owner: read

This creates a read/write region (or simply changes the status of an existing region) owned by the specified thread, and other threads are **not** allowed to keep readable **or** writable copies in their L1 and L2 caches. When the region is released, it is marked as readable; if afterward a thread wants write access, it must perform an acquire operation.

3.3.2.4 Code Memory consistency (UNISI)

To create an executable block of code, perform the following:

- *region-acquire*
id: kernel
region: frame
writers: nil
permissions: owner: **read + write**, non-owner: **nil**
- [initialize region]

- *region-release*
  region: frame
  permissions: owner: read, non-owner: read + execute

This creates a read/write region (or simply changes the status of an existing region) owned by the specified thread, which in this case is the kernel, perhaps acting on behalf of the compiler. While the kernel writes the region, other threads are **not** allowed to keep readable **or** writable copies in their L1 and L2 caches. When the region is released, it is marked as both readable and executable.

3.3.2.5 Transactional Memory consistency (UNISI)

It depends upon the Transactional Memory (TM) implementation (please see Section 3.1.3).

3.3.3 Memory consistency overheads and TSU impact (UNISI)

Several scenarios make use of the **release** operation, and this operation indicates to the L2/TSU that the status of a region of memory has changed. The only reason to provide separate acquire/release operations is if they behave differently (otherwise one could simply have a single "set-region-permissions" operation that handles everything).

Both implementations are equally viable. Just for the sake of clarity, we will assume we keep the dual acquire/release pair. If we assume that both acquire and release are used, then the difference is that one broadcasts, and the other does not. We will assume that the acquire operation broadcasts whenever necessary, and that the release operation does not.

Why this is important is that, for correct operation, this scenario (in which a release operation does not automatically notify the cores of updates) requires either polling by threads expecting a particular region to become available, or wakeups sent from the TSU (which, yes, are a form of broadcast, thereby breaking our "no broadcast" model, which begs the question again of broadcast).

Consider the example of an owner-writable memory in which concurrent readers are not desired—in such a scenario, a previous acquire operation has removed copies of the region from all L1 and L2 caches in the system; the owner thread writes to the region and then releases it. How do the waiting threads know that the region is now available? Answer: they have to perform a load instruction that misses their L1 cache, then send a request to the L2/TSU, which sees the updated metadata and obtains the most recently written copy from memory. How do they know to perform this load instruction? Either they have been looping on the same load instruction for a while (each time, causing a form of SEGFAULT in which the hardware realizes at the TSU level that the requesting thread is not trying to reference a non-existent frame but instead is referencing a temporarily unavailable frame), or the TSU has a list of threads to wake up when a particular frame is released.

The only reason to consider the broadcast-wakeup operation is to avoid the excessive power dissipation that would result from numerous spinning (polling) threads.

While keeping the advantages of supporting programmability (cf. D3.5) and reliability (cf. D5.4) the TSU will face the same types of communication overhead seen in many-core designs and their memory-consistency mechanisms—namely the overhead of managing data-protection information that is not centralized but is instead distributed widely throughout the system. The overheads include both space for storage (i.e., the amount of memory required to hold the data-protection information) and the time for metadata movement (which affects both performance and energy consumption). In particular:

- **Number of Frame Records**. The amount of space needed to hold the metadata for all of the frames and regions referenced by the cores of a node will scale with the numbers of cores on the chip as well as a given application workload and the degree of sharing that it exhibits. However, practically speaking, there is no need for more metadata than that needed to manage the amount of cache storage. For example, if a chip had 100 blocks of cache storage, it would not need significantly more than 100 records of frame/region metadata. Providing additional space for metadata would be unlikely to yield significant performance benefits for normal memory accesses, as it would only improve the uncommon case of last-level cache misses—reducing only slightly the reload penalty, assuming a metadata record is smaller than a cache block. The main benefit of extra frame records would be for updating the consistency status of temporarily unavailable regions that are not currently held in the cache. It is safe to assume that these would not be more numerous than the currently available regions, and it is also safe to assume that most regions would have more than one block held in the cache. Therefore the space allocated to frame records at the node level should scale with the L2 cache.

  The number of unique frames and/or regions that can be stored in the L2 cache (i.e., the maximum number of metadata records needed) is limited by the number of unique cache blocks in the L2 cache. For example, a 1MB cache with a 64 byte cache block would need a maximum of 16K frame records; a cache of half this size or twice the block size would need a maximum of 8K records; etc.

- **Frame Record Size**. As described earlier, the following metadata information is required to manage each frame or region:

  - Frame ID: 22–32 bits (see Figure)
  - “Reader/writer” dependence information (could be a simple counter value, for example 16 bits) (e.g., the “synchronization counter”)
  - Thread owner: 16–32 bits (depends on scope of uniqueness)
  - Permissions: 6 bits at minimum (r|w|x for each of *owner* and *non-owner*)

  This yields a metadata record of 60–86 bits (8–11 bytes) in size, significantly smaller than the cache block.

- **Time Requirements**. The performance impact of managing frame records can only be determined through detailed simulation; however, it is possible to get a rough idea of per-operation costs. Worst-case operations require some degree of broadcast (e.g., see discussion above contrasting **acquire** and **release** operations), which means the entire chip’s interconnect network would be dedicated for a length of time required to transmit roughly a dozen bytes of information. The following graph shows for a range of parameters what the cost to the system is, in units of cycles per instruction, for updating frame records. This is the number of cycles spent broadcasting frame-record update information on a dedicated frame-record broadcast bus, per CPU cycle (during which 1000 cores are each executing their own instructions). The parameters include the following:

- Frame-record bus width: 4, 8, or 16 bits wide
- Average occurrence of memory read/write operations: 0.25–0.40 per instruction (typical application percentages are roughly one out of three)
- Likelihood that any given memory reference causes a consistency update: 0.0005–0.0015 (0.05%–0.15%)—one would expect that this value would be zero for workloads of independent threads, low for workloads with little sharing between threads, and relatively high for workloads that share significant data between threads

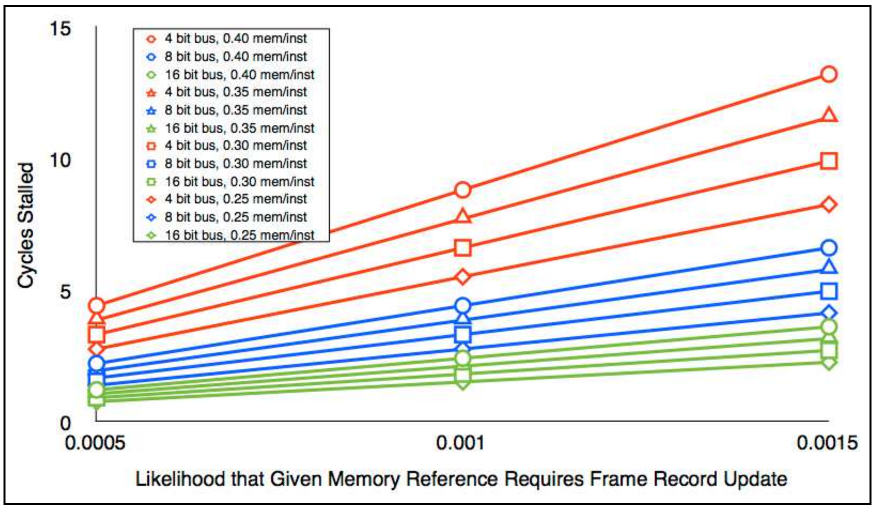


Figure 29: Cost per Cycle to Broadcast Frame Records

As the graph shows (Figure 29), the overhead of consistency traffic in a 1000-core chip is likely to be extremely high: in particular, note that any value greater than 1 indicates that the traffic level is unsustainable, and nearly every graph is above 1. Workloads that share any significant amount of data between threads—the very workload set for which TERAFLUX is intended—will require either wide consistency-information busses, or numerous busses, or both. For instance, a design using a 4-bit consistency bus and a typical consistency update ratio of one tenth of one percent (0.001) will require 5–10 independent consistency busses on the chip to sustain execution throughput of one instruction per core per cycle. A design using a 16-bit consistency bus and the same update ratio (0.001) will require 2–3 independent consistency busses. Either way, the total width of the bus required—just for consistency information—would be 20–50 bit lanes.

3.3.3.1 TSU storage and time requirements (UNISI)

Given the estimates of storage and time overheads, it is possible to propose an initial, approximate design for the memory structures held within the TSU and the interconnects tying the node-level TSUs together. The TSU and its distributed system will require several obvious structures:

- **Frame-Record Cache**. The frame-record cache holds the frame/region metadata; this structure is maintained at the node level, in conjunction with the L2 cache. As discussed earlier, the frame-record cache would need to hold roughly the same number of records as the number of unique regions and/or frames held in the L2 cache, plus extra records to manage the consistency for temporarily unavailable frames. The total number of records required can be estimated to be equal to the number of blocks in the L2 cache.

The frame-record cache would not need to maintain strict inclusion with the L2 cache; it is certainly possible to support independent lookups and refills for both the data/instruction cache and the frame-record cache. Thus, while the cache used to hold the frame records would need to be associative, it would not need to be a fully associative CAM. Given the large number of records, a moderate degree of associativity would likely suffice (this, however, should be verified in simulation). For example, a starting point for design exploration would be 8K entries, each of 8 bytes (requiring 64KB of storage), the whole cache organized as 8-way set associative.

- **Communication Buffers**. TSUs must handle inter-node communications to manage global consistency issues. Given that any core can request data from any other core (other than issues of address-space privacy enforced by the operating system, TERAFLUX insists that there be no limitations on the regions of memory that can be referenced by a given thread), TSUs could be handling as many simultaneous outgoing requests as there are cores in a node. Further, if requests take multiple cycles, say N, the number of outstanding requests could grow by a factor of N. Therefore, each TSU will require a set of read/write communication buffers to hold messages until they have been processed. A starting point for design exploration, assuming a node design of 32 cores, would be a 64-entry outgoing message buffer and separate 64-entry incoming message buffer, each organized as a fully searchable CAM but also able to maintain local message ordering (e.g., FIFO insert, fully associative lookup, much like a traditional reorder buffer).

- **Consistency Bus**. As briefly analyzed above, it is likely that the chip-wide consistency bus will require several dozen bit-lanes to ensure sustainable data movement; note that this is in addition to the chip-wide data bus. Assuming that the ratio of consistency messages to data messages is slightly higher than 1 (every data miss will likely require corresponding consistency information, but not every consistency message will require corresponding data), the consistency bus will need slightly higher message throughput than that of the data bus. It is also likely that an optimal choice will match the number of cycles needed to transmit a consistency packet on the consistency bus to the number of cycles needed to transmit a data packet on the data bus. E.g., transmitting an L1 cache block of size 32 bytes on a data bus of 32 bits will require a minimum of 8 cycles; the consistency bus for such an organization should be designed to take that many cycles or fewer to deliver a consistency packet. A starting point for design exploration, assuming a 32-byte L1 cache block and a 32-bit data bus, would be an 8-bit consistency bus. This would require 8 cycles to transmit up to 64 bits of frame record and op-code. The graph above suggests that a minimum of 3–5 independent consistency busses would be required to maintain throughput, depending on memory-request mix. The final number of consistency busses would have to be a least one more than the number of independent data busses. As the section below suggests that 2–3 such data busses would be required, the number of consistency busses will not have to be increased simply to accommodate a large number of data busses.

More efficient solution are under exploration, such as merging this information with the TLBs, but the above discussion can clarify the limits of a traditional separated implementation. More recently, BSC has shown that the current protection mechanisms are quite rudimentary and a finer-grain isolation can be done efficiently (ISCA-2014 paper [33]).

Other details are provided in Section 4.2.2.

4 Abstraction Layer

4.1 Abstraction layer for reliability (UAU)

From the reliability perspective, the abstraction layer has to hide the effects induced by hardware faults from the application layer. Therefore, the TERAFLUX Node Manager (D-FDUs and D-TSUs) provides an abstraction of the underlying system, even when low-level components are suffering from permanent, transient, or intermittent faults.

4.1.1 Abstracting from faults

To describe the abstraction of faults to the application level we illustrate the fault tolerance workflow in Figure 30. The D-FDU implements two main functionalities to provide the fault abstraction. First, it gathers health state information from its affiliated processor cores by the heartbeat message mechanism. Thereby, the D-FDU identifies corrupted/failed components, such as failed processor cores and routers. In addition, faulty dataflow thread executions are determined by a Double Execution presented in Deliverable D5.4. Second, the D-FDU creates two different datasets from the gathered information regarding the performance capabilities of its node. One detailed dataset is published to the D-TSU to allow a fine grained task placement, including precise information of failed or failure prone components. The other dataset is more general and includes an aggregated and abstract health status of the whole node. The latter dataset contains no direct information of which component is broken, since this information is only used by the D-TSU.

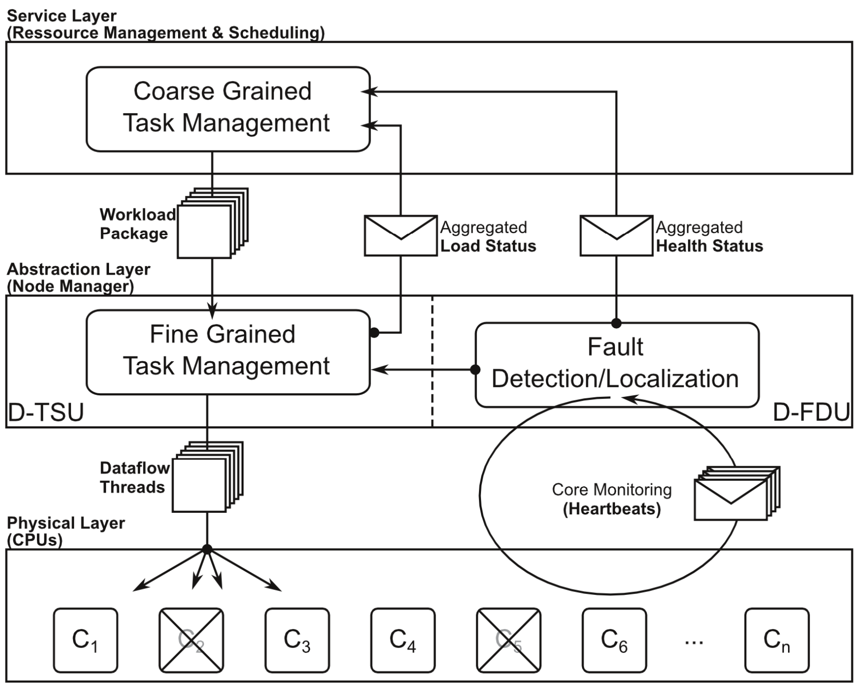


Figure 30: The TERAFLUX abstraction layer from the fault tolerance point of view.

The high-level administrative units, such as the top-level scheduler rely on the abstracted health status, as well as on the work load status provided by the D-TSU. This status is needed since faults will eventually degrade the amount of processing capabilities of that node, arising local overload situations when more workload is assigned to this node. Based on these two datasets, the top-level scheduler is able to create coarse grained workload packages, without the need for detailed information about the actual node's physical condition.

The nodes' D-TSUs, however, rely on those detailed information, in order to react on faulty components. While it informs the high-level administrative units, it is also responsible to perform recovery actions, to ensure a correct thread execution. Therefore, it migrates dataflow threads from permanent or intermittent faulty cores to more reliable ones. Furthermore, it prevents newly assigned threads from the higher level units to broken cores.

Since the recovery mechanisms play an essential role in the abstraction layer, we also present a short introduction in local thread recovery and global system recovery. Later on, we will also motivate the interconnection network fault localization for its use in the TERAFLUX abstraction layer.

4.1.2 FDU/TSU Checkpoint/Restart Mechanism

For thread recovery and its check pointing system, we need to distinguish between three different levels within the TERAFLUX architecture; core-level, node-level, and system-level. Each level poses different challenges but also provides advantages upon the other. We start with a short description of the core-level mechanisms and continue with the node-level and system-level mechanisms. Please note, that more detailed information are available in the Deliverables D5.3 and D5.4.

4.1.2.1 Core-Level Thread Restart Recovery

On core-level the restart functionality leverages the inherent existing checkpoints between dataflow threads. These checkpoints are natively generated by two factors. First, a dataflow thread is only allowed to be executed, if all its input operands are available. Second, the communication of threads takes place only at the end of a thread execution and is deferred until the D-FDU confirmed the fault free thread execution. This means, if a thread was successfully and fault-free executed, the resulting outputs from the thread execution are stored to its successor threads. By the time all input operands of a certain successor thread are available, the checkpoint for this particular successor thread is reached, because no other thread is now able to alter the data within the successor's input operands.

A faulty thread execution, however, will trigger the node manager's thread restart recovery mechanism. This means in the first step, the D-FDU notifies the D-TSU about the failed thread execution and the core, which was responsible for the failed thread execution. The D-TSU in turn stops all activities of this particular core and re-allocates all threads assigned to the faulty processor core. The information of all thread assignments is stored in the D-TSU's Thread-To-Core list.

For the case where faults are rather seldom, a more optimistic approach may be preferable. Therefore we extended the double execution based fault detection including a speculative start of successor threads as soon as the starting condition for this thread is met. This also includes, that the successor thread may work on corrupted operands induced by a fault within a former thread execution. This extension of double execution raises the need for a broader checkpointing mechanism covering more than the core's write buffers, which will be discussed in the next two subsections. For detailed insights in both pessimistic and optimistic double execution, we refer the reader to Deliverable D5.4 section.

4.1.2.2 Global Recovery

The node-level recovery mechanism is dedicated for situations, where optimistic double execution is applied and the communicating dataflow threads reside in the same node. In such a case, the D-FDU can create a checkpoint after an arbitrary thread commitment, including the node's frame memory region from within the global memory. In addition, the D-TSU creates a backup of its own context, which also contributes to the checkpoint.

All subsequent writes to the node's frame memory are then *logged* by maintaining a log of all changes to the node's frame memory. In case of a faulty thread execution after the checkpoint was created, the D-TSU only needs to restore the backup thread frame memory and its own context for recovery. As a side effect, newly allocated thread frames do not need to be part of this log, since these thread frames will be re-created after the node recovery

In the last case we consider thread communication across node borders along with the optimistic double execution. In this case, the node management needs to safe guard the full system to recover from faults, as faults may be propagated to the global memory and thereby may be spread out into several nodes. Since the node management units D-TSU and D-FDU are autonomous within each node, there is a need for negotiations between all nodes in order organize both creating a consistent global checkpoint and the fault recovery. We implemented a two-staged mechanism to create a consistent global checkpoint.

In the first stage, the node managers negotiate to agree that a new checkpoint should be created. This implicates, that all node managers wait until the pending write operations from inflight dataflow threads are finished. All subsequent write attempts are prohibited and thereby stopped. After a successful negotiation, each node manager start to create a checkpoint for its own node as described above. After this has been done, the system can proceed with the thread execution. The system-level recovery and checkpointing is also detailed in D5.4.

The reliability abstraction layer guarantees that the hardware always appears fault-free, even in the case of intermittent, permanent, or transient faults in the cores. In the following, we will describe, how the architecture guarantees the transparency of faults to the application layer.

4.1.2.3 Abstracting from Transient Faults

In the case of a transient fault, our fault recovery mechanism ensures that the faulty thread will be re-executed. Since the fault was transient, the faulty core is still used for further workloads. If the system uses thread restart recovery, the fault just leads to an increased thread execution time, since the execution of the faulty thread must be aborted and re-executed.

However, if the system is operating in global system recovery mode, then the whole system is rolled back to safe, fault-free state. In this case, the global execution is degraded, since all threads and the global memory must be recovered to a global and safe state.

4.1.2.4 Abstracting from Permanent/Intermittent Faults

If the system is suffering from a permanent or intermittent fault, identified by the D-FDU the node's D-TSU will decided to shut-down the core. Then the core will no longer be used by D-TSU to execute subsequent threads. In this case, the node's performance is permanently or temporarily degraded. This means that the workload can no longer be evenly distributed to all nodes. Instead, the D-FDU determines the current performance of its node, and informs all other D-FDUs about the reduced performance of the node. In this case, the node-local D-TSUs adjust their scheduling, in order to prevent overloading situations of certain nodes.

4.1.2.5 Fault localization on NoC-Level

The interconnection network has a significant impact on the processor performance as soon as bottlenecks may arise, which consequently slows down communication. These counts especially if threads communicate with each other and do not start executing until all their operands are available. Therefore, it is a good idea to take advantage of the health state information of the interconnection network as well, since work package assignments are performed on *all* levels. However, as the abstraction layer hides specific insights of physical conditions from the high level administration units, the D-FDU also aggregate the health status of the node's interconnection network. So the top-level scheduler may also consider the available bandwidth of a node, when creating a work package for that node.

The Localization is performed by the D-FDU and incorporates health state messages (or so called heartbeats) sent from the monitored cores within a node. Although these messages are not altered for fault localization within the node's network, the restrictive way these heartbeats are transmitted through net network allows us to extract additional information regarding the network. The main restriction for heartbeat messages is the strict timing determinism, achieved by isolating these messages from each other and application messages.

Any deviation from the timing determinism, that is a delayed arrival of a heartbeat message, can be interpreted as an indicator for a faulty network component. These delays are caused by detours a heartbeat message needs to take to circumvent a faulty router/link. Since the routing for heartbeat messages is also well known, the D-FDU consider for the moment all affiliated network components as suspicious. Heartbeats that arrive in time at the D-FDU rehabilitate all affiliated components on the message's path. Delayed heartbeats instead strengthen the suspicious values for certain network components. So in other words, the more heartbeat messages arrive at the D-FDU, the more accurate becomes the assumption where the fault is located in the network. A detailed description with examples is given in the Deliverables D5.3 (basic localization concept) and D5.4 (extensions to multiple fault localization).

4.2 Abstraction layer for performance (UNISI, UCY)

4.2.1 Dynamic Scheduling in DDM (UCY)

During year 3 of the project, a dynamic scheduling policy for the DDM-style runtime was created. This policy distributes the threads to the cores in a way that maximizes load-balance. It schedules ready threads to the Waiting Queue that has the least number of entries. In case a core is idle for a certain time period, it might take some of the workload of another core. Since all threads are side-effect-free, work stealing is allowed. As described, this mechanism can be used as a support for the abstraction layer. It is possible to have a virtual-to-physical translation table to allow for the virtualization. Work is assigned to cores according to their availability. If a core is determined to be faulty from the FDU, the entry in that table can be marked as unavailable and thus dynamically no more work will be sent to that core. Also if a core reaches a thermal limit, it may also be marked as unavailable for a certain period of time. In addition it is possible to change dynamically the total number of cores that are used for the application just by making some of the cores unavailable.

4.2.2 Virtual memory implementation in TERAFLUX architecture (UNISI)

The virtual memory model is intended to simplify sharing, as the chip is clearly intended to support extremely wide parallel threaded applications—as opposed to running 1000 different, unrelated applications at the same time. Because protection is nonetheless important (it will be used to implement the various memory-consistency models), its implementation must be simple and low-overhead. Translation from virtual to physical addresses must be done, but it can be done at a relatively large granularity—e.g., there is no need to retain the traditional 4K page size, especially considering the typical multi-GB-per-socket main memories available today. Shared memory can be both simple (e.g., in a way that avoids the cache problems of virtual-address aliasing) and flexible (e.g., in a way that allows virtual-address aliasing).

Proposal

Understanding that the specific sizes are yet to be determined (e.g., the size of the physical page size and the number of regions in the virtual space), the virtual-memory implementation presented here has the following general characteristics:

- The architecture uses a 64-bit virtual address, an 80-bit global address, and a 48-bit physical address;
- Process/thread address spaces are comprised of between hundreds and tens of thousands of equal-sized regions (which can be called “frames”), each uniquely identified by a region ID, managed by the TSU system. It is likely that region IDs would be between 6 bits and 16 bits, so as to keep the size of the ID tables as small as possible, thereby enabling them to be held entirely in hardware (as opposed to being cached, as in a page-table-cum-TLB arrangement);
- Frames/regions are mapped through the ID table onto the global address space, which is comprised of between 4 **million** frames (corresponding to a 6-bit region ID) and 4 **billion** frames (corresponding to a 16-bit region ID); thus, each frame is uniquely identified within the global space by a large (22-bit to 32-bit) Frame ID;
- There is a one-to-one mapping from the 80-bit global address space to the 48-bit physical address space, enforced by the operating system through a global page table. Note that no such limitation on mapping exists between the 64-bit virtual space and either the global space or the physical space; each thread can map its memory to the global space however it desires, up to the limitations of the protections enforced by the operating system;
- There is a many-to-many mapping from each thread’s address space to the global space, enabling virtual-address aliasing at the process/thread level;

- Memory is shared at the frame granularity, and each thread can map a given frame using a different protection (e.g., one thread can see the frame as read-only, while the other can see it as write-only);

- Memory is translated, i.e. also relocated, at the granularity of large (e.g., 64KB–1MB) pages; this is the minimum amount of memory that can be allocated. However, it could very well be that space could be *allocated* but not actually *used* (e.g., reserved in main memory but never written to).

Figure 31 shows the relationship between the various addresses and the hardware mechanisms used to provide translation and protection. It shows the extremes of the possible design choices: the top example illustrates a 16-bit region/frame ID (corresponding to a 64K-entry ID Table) and a 1MB physical page size; the bottom example illustrates a 6-bit region/frame ID (corresponding to a 64-entry ID Table) and a 64KB physical page size.

As is well known, this type of segmented mapping arrangement, in conjunction with a global page table, solves the aliasing problem for virtual caches.

The ID Table should be small enough to be an actual table in the hardware, or—if simulations support such a design decision—a sparsely populated table such as a small CAM. Its purpose is to handle the verification of protections & privileges, and to map addresses from the process/thread virtual address space to the global address space, both at the granularity of regions or frames. Because the ID table is a core-level hardware resource, its overhead must be as small as possible, in terms of both performance impact and power impact. The TLB should reside at the chip level, but if simulation suggests that inter-thread sharing will be minimal, a node-level TLB could be used instead. As mentioned above, the virtual memory architecture would use virtual caches while allowing cache synonyms.

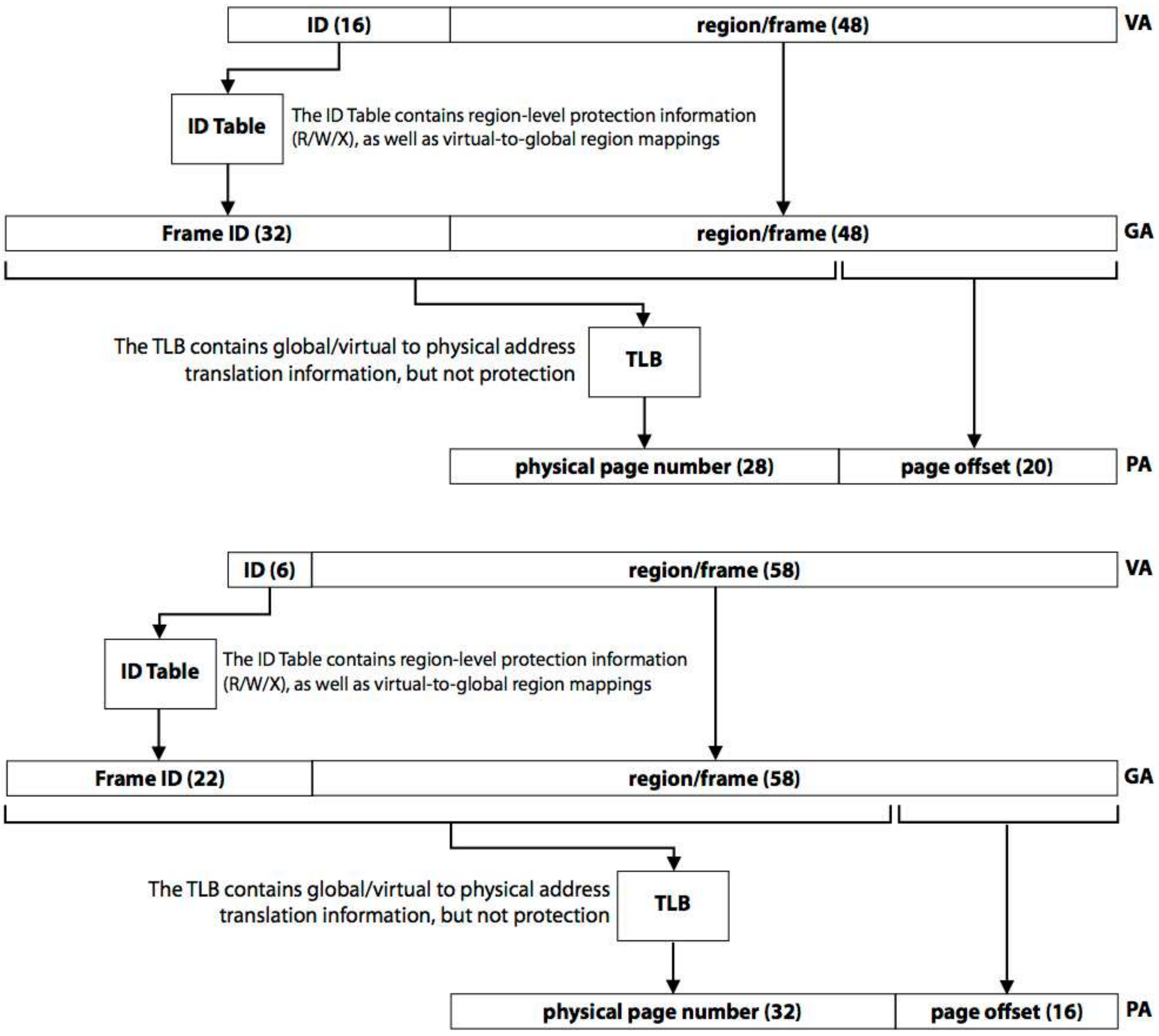

ID (16)
region/frame (48)
VA
ID Table
The ID Table contains region-level protection information (R/W/X), as well as virtual-to-global region mappings
Frame ID (32)
region/frame (48)
GA
The TLB contains global/virtual to physical address translation information, but not protection
TLB
physical page number (28)
page offset (20)
PA
ID (6)
region/frame (58)
VA
ID Table
The ID Table contains region-level protection information (R/W/X), as well as virtual-to-global region mappings
Frame ID (22)
region/frame (58)
GA
The TLB contains global/virtual to physical address translation information, but not protection
TLB
physical page number (32)
page offset (16)
PA

Figure 31: Virtual memory architecture, two extremes. Top and bottom figures illustrate different extremes for Region IDs, Frame IDs, and physical page sizes. The thread uses 64-bit addresses that are mapped at a region/frame granularity onto the global address space. Each process/thread address space is comprised of over a thousand of such regions. The global address space is comprised of between millions and billions of such regions. Protection information is held in the ID Table, one of which is maintained for every process/thread address space, and which is held in a per-core hardware structure while the thread is running. The TLB caches page- table entries and translates addresses from the global address space to the physical address space at the granularity of large pages (between 64KB and 1MB).

Hardwiring the Address Bits

Given appropriate TSU-level support, it is not necessary to hardwire regions of the virtual, global, or physical space as having particular consistency characteristics (e.g., by using one or more bits at the top of the address to signify cached/non-cached, kernel/user, transactional/normal, etc.).

Hardwiring specific address bits to specific (and necessarily software-exposed) memory behavior is best done (a) when it is unavoidable, as in the MIPS architecture, or (b) when it would be useful to allow a user thread to turn such characteristics, such as cached/non-cached, on and off for a given object without having to go through the operating system. E.g., if the top virtual bit is ignored as an address and simply indicates to the hardware whether or not to cache the referenced data, the thread can change an object's characteristics by changing the pointer, without having to invoke a system call. The implementation proposed here certainly *allows* the hardwiring of address bits; it simply does not *require* it. This flexibility means that, should it be decided that it would be beneficial to expose consistency in such a way, the implementation can be accomplished easily.

The downside of hardwiring address bits is that it necessarily exposes hardware implementation details to software: a design decision that cannot easily be undone (e.g., branch delay slots in the MIPS architecture, which were lauded for in-order pipelines but caused enormous headaches for the high-performance out-of-order implementations that came much later). In general, any design decision that blurs the distinction between implementation and architecture should be thought through very carefully to avoid any unnecessary future limitations.

4.2.3 TLBs integration in the TERAFLUX architecture (UNISI)

Virtual memory is a requirement of the system, which suggests the use of one or more TLBs. While eliminating all TLBs is certainly a realistic design option, the performance of doing so is only acceptable when the cache system is large enough to accommodate both the application and the operating system, i.e., significantly larger than what the application needs by itself. Given that integrating 1024 cores on a single chip is likely to limit significantly the amount of on-chip cache available, it seems likely that a TLB will be required.

As for the TLB architecture and the location of the *translation point*, there are three obvious possibilities to consider:

- *One TLB per core*

  This allows each core to have its own virtual space; sharing between cores must be through the physical space. Any shared caches use physical addresses, thereby simplifying the virtual address aliasing problem. Any modification to the TLB-resident meta-state of an address (e.g., its virtual-to-physical mapping) would require broadcast to the core-level.

  Expected power per chip, only for TLB-based translation: 200c, where c is the expected power dissipation for a single core. Therefore, 20% on top of expected chip power[2].

  Advantages: simple programming model

  Disadvantages: high power dissipation, TLB shootdown issues scale with cores

- *One TLB per node*

  This effectively creates a separate virtual space for each node, but all communication outside of the node is physical. Sharing between cores within the node is through virtual addresses; sharing across nodes is through physical addresses. Node-level shared caches would most likely be physically indexed, virtually tagged (i.e., what is commonly used in x86-compatible

2. The power overhead of the TLB is approximated as 20% of the power of a single core, consistent with recent results.

architectures today). Any modification to the TLB-resident meta-state of an address (e.g., its virtual-to-physical mapping) would require broadcast only to the node-level, not the core-level.

Expected power per chip, only for TLB-based translation: 10c, where c is the expected power dissipation for a single core. Therefore, only 1% on top of expected chip power.

Advantages: low power dissipation, less consistency-related communication than a per-core TLB

Disadvantages: TLB shootdown overhead scales with the number of cores (i.e., with the degree of sharing, not with the number of TLBs), scheme creates a complex shared-memory model that depends on whether a collaborating thread is resident on the same node or not (e.g., threads can use the same virtual address for a shared region if they want to, but only if they are co-resident on the same node, otherwise they must use physical addresses to share), separate node- and/or core-level mechanism required for data protection at the frame-access and/or cache-access levels,

- *One TLB per memory channel*

  Given that the memory channels connected to the chip will be managed by the operating system, this effectively shares a single virtual space across the entire chip. Core- and node-level caches would be virtual; any chip-level cache would be physically indexed, virtually tagged. Modifications to the TLB-resident meta-state of addresses (e.g., virtual-to-physical mappings) would require no broadcast.

  Expected power per chip, only for TLB-based translation: 1c, where c is the expected power dissipation for a single core. Therefore, less than 1% overhead.

  Advantages: low power dissipation, TLB shootdown eliminated, extremely simple model for shared memory

  Disadvantages: must deal with virtual-cache aliasing problem (which is solved, see below), separate node- and/or core-level mechanism required for data protection at the frame-access and/or cache-access levels

Note that the TLB handles translation tasks, and, depending on the TLB's placement within the hierarchy, it could also handle *protection* tasks as well. If the TLB is located at the core level, then it can handle all operations, both translation and protection. If the TLB is located at the node or chip level, however, a separate mechanism will be required at the core level to handle protection tasks (e.g., verifying access privileges for read/write access to frames, read/write access to shared memory, correct sequencing of transactions, etc.). Note that the *ID Table*, described in the Virtual Memory section below, could provide this function.

4.2.4 TM Interface (UNIMAN)

In year 3 UNIMAN has developed an interface between the abstraction layer and the actual architecture with respect to TM. For the abstraction layer, we indicate that Transactions can be nested using closed nesting behavior.

In WP3 D3.3 there is more information about how dataflow and TM is allowed to be combined in abstract level.

The TM interface is the following:

- **tm_begin** - Marks the start of a transaction. Speculative execution begins and stores are isolated from this point on.
- **tm_end** - Ends a transaction and performs conflict detection. In case a conflict has occured, execution is squashed from the point of the matching tm_begin. Execution returns to the start of the transaction.
- **tm_abort** - Explicitly aborts a transaction. Execution is squashed from the point of the matching tm_begin. Execution returns to the start of the transaction.

Transactions can be nested. Closed nesting behavior is provided. The simulator modules reported upon in Deliverable 7.4 implement this interface.

5 Conclusions

In the last period, WP6 focused on the TERAFLUX Fine-tuned Execution Model and the Abstraction layer. In the TERAFLUX Fine-tuned Execution Model we provided (i) support for the evolution of the programming model, (ii) advance scheduling mechanism and (iii) Advance Memory management.

Two Hardware implementations, one for the DDM-style execution model and the second for the HTTS (Hardware TaskSuperScalar) have been developed, using FPGAs. Both implementations provided actual timings that can be used in the TERAFLUX architecture and they achieve very good results.

We have developed two DDM-Style software platforms the TSU++ and the TFluxSCC. The hardware HTSS has improved significantly the TaskSs model. A token based transactional memory system that keeps a check on the number of potentially conflicting transactions running in a Transactional Memory system.

Two contributions were made for the Advance Scheduling: (i) The Scheduling of DDM-style execution that supports: Dynamic, Static, Round robin, and Modulo scheduling. (ii) The improvements to the Hardware TaskSuperScalar have reduced the resource utilization compared to the initial design and at the same time the task scheduling throughput has been increased.

For the advance memory management we have introduced a possible implementation of the memory consistency mechanisms for all the memory types of TERAFLUX: Frame memory, Thread local storage, Owner-Writable, Code memory and Transactional memory.

The abstraction layer was designed to hide the effects induced by hardware faults from the application layer. The collaboration between WP5 and WP6 has enabled TERAFLUX to tolerate faults at all levels of the system. Dynamic scheduling, support for transactions and virtual memory implementation are other features supported by the abstraction layer.

Overall, WP6 has achieved all of its goals and have also added work that was not envisioned at the proposal such as the Hardware (FPGA) development.

References

[1] David H. Bailey et al. "The NAS parallel benchmarks." International Journal of High Performance Computing Applications 5.3 (1991): 63-73.

[2] Andreas Diavastos, Giannos Stylianou and Pedro Trancoso. "TFluxSCC: A Case Study for Exploiting Performance in Future Many-core Systems". To be published in the ACM International Conference on Computing Frontiers 2014.

[3] Yoav Etsion, Felipe Cabarcas, Alejandro Rico, Alex Ramirez, Rosa M. Badia, Eduard Ayguade, Jesus Labarta, and Mateo Valero. 2010. Task Superscalar: An Out-of-Order Task Pipeline. Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '43). IEEE Computer Society, Washington, DC, USA, 89-100.

[4] Matthew R. Guthaus et al. "MiBench: A free, commercially representative embedded benchmark suite." Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on. IEEE, 2001.

[5] Jason Howard et al. "A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS." Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International. IEEE, 2010.

[6] Costas Kyriacou, Paraskevas Evripidou, and Pedro Trancoso. Cacheflow: A Short-Term Optimal Cache Management Policy for Data Driven Multithreading, Proc. EuroPar-04, pages 561–570, Aug. 2004.

[7] Timothy G. Mattson et al. "The 48-core SCC processor: the programmer's view." Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE Computer Society, 2010.

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

[9] Xilinx Inc.Virtex-6 FPGA family. [Online]. Available: http://www.xilinx.com/products/silicon-devices/fpga/virtex-6/index.htm

[10] Xilinx Inc. Microblaze soft processor core. [Online]. Available: http://www.xilinx.com/tools/microblaze.htm

[11] Xilinx Inc. FSL v20. [Online]. Available: http://www.xilinx.com/support/documentation/ipembedprocess_processorinterface_fsl.htm

[12] Fahimeh Yazdanpanah, Carlos Álvarez, Daniel Jiménez-González, Rosa M. Badia, Mateo Valero. Picos: A Hardware Runtime Architecture Support for OmpSs. Submitted to Journal Future Generation Computing Systems. Under Revision. 2014.

[13] Fahimeh Yazdanpanah, Daniel Jiménez-González, Carlos Álvarez Martínez, Yoav Etsion, Rosa M. Badia, Analysis of the Task Superscalar Architecture Hardware Design, Proceedings of the International Conference on Computational Science, ICCS, 2013, 339-348.

[14] H. Chafi, J. Casper, B. D. Carlstrom, A. McDonald, C. Cao Minh, W. Baek, C. Kozyrakis, and K. Olukotun. A scalable, non-blocking approach to transactional memory. Proceedings of the International Symposium on High Performance Computer Architecture, February 2007.

[15] Xuehai Qian, Wonsun Ahn, and Josep Torrellas. Scalablebulk: Scalable cache coherence for atomic blocks in a lazy environment. Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture, pages 447–458, 2010.

[16] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded Transactional Memory. In Proc. of the Eleventh IEEE Symposyum on High-Performance Computer Architecture, Feb. 2005.

[17] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The Notions of Consistency and Predicate Locks in a Database System. Communications of the ACM, 19(11):624–633, 1976.

[18] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional Memory Coherence and Consistency. In Proc. of the 31st Annual Intl. Symposyum on Computer Architecture, June 2004.

[19] H. T. Kung and J. T. Robinson. On Optimistic Methods for Concurrency Control. ACM Transactions on Database Systems, pages 213–226, Jun. 1981.

[20] Ravi Rajwar, Maurice Herlihy, and Konrad Lai. Virtualizing Transactional Memory. In Proc. of the 32nd Annual Intl. Symposyum on Computer Architecture, Jun. 2005.

[21] Scalable Object-Aware Hardware Transactional Memory. Behram Khan, Matthew Horsnell, Mikel Lujan and Ian Watson. In the 16th International Euro-Par conference on Parallal processing (EuroPar'10), 2010.

[22] S. Kumar, C.J. Hughes, and A. Nguyen. 2007. Carbon: architectural support for fine-grained parallelism on chip multiprocessors. ACM SIGARCH Computer Architecture News 35, 2 (2007), 162–173.

[23] Architectural Support for Task Scheduling: How to take advantage of explicit data dependencies. Behram Khan, Daniel Goodman, Salman Khan, Will Toms, Paolo Faraboschi, Mikel Lujan and Ian Watson. Submitted to ACM TACO 2014.

[24] M.T. Vandevoorde, E.S. Roberts, Workcrews. 1998. An abstraction for controlling parallelism.

[25] Kranz, B. Venners, R.H.H. Jr. 1990. Lazy task creation: a technique for increasing the granularity of parallel programs. E. Mohr, D.A.

[26] D. Hendler, N. Shavit. 2002. Non-blocking steal-half work queues.

[27] D. Chase, Y. Lev. 2005. Dynamic circular work-stealing deque.

[28] U.A. Acar, G.E. Blelloch, R.D. Blumofe. 2000. The data locality of work stealing.

[29] Bloom, B.H. 1970. Space/time trade-offs in hash coding with allowable errors.

[30] J. Dean, S. Ghemawat, Commun. http://doi.acm.org/10.1145/1327452.1327492.

[31] Lea, D. S. L. In Proceedings of the ACM 2000 conference on Java Grande, 2000. pp. 36-43.

[32] B. Jacob, S. Ng, and D. Wang. 2007. *Memory Systems: Cache, Dram, Disk*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[33] L. Vilanova, M. Ben-Yehuda, N. Navarro, Y. Etsion, M. Valero "CODOMs: Protecting Software with Code-centric Memory Domains", ISCA' 14, June 2014 (In Press).

[34] R. Giorgi, R. M. Badia, F. Bodin, A. Cohen, P. Evripidou, P. Faraboschi, B. Fechner, G. R. Gao, A. Garbade, R. Gayatri, S. Girbal, D. Goodman, B. Khan, S. Koliaï, J. Landwehr, N. Minh L, F. Li, M. Lujàn, A. Mendelson, L. Morin, N. Navarro, T. Patejko, A. Pop, P. Trancoso, T. Ungerer, I. Watson, S. Weis, S. Zuckerman, M. Valero "TERAFLUX: Harnessing dataflow in next generation teradevices", Journal of Microprocessors and Microsystems: Embedded Hardware Design (MICPRO), April 2014, doi: doi.org/10.1016/j.micpro.2014.04.001