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

PROJECT NUMBER: 249013

TERAFLUX

Exploiting dataflow parallelism in Teradevice Computing

D9.3 – Evaluation of the Codelet Runtime System on a Teradevice

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: UD

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#	Author	Organization	Change History
1	Stephane Zuckerman	UD	Initial Skeleton
2	Stephane Zuckerman	UD	First draft – still some parts missing
3	Stephane Zuckerman	UD	Added work by Chen et al. on automatic static codelet scheduling.
4	Stephane Zuckerman	UD	Added model for optimal tile size search.
5	Stephane Zuckerman	UD	Added Energy vs. Performance tiling discussion.
6	Stephane Zuckerman	UD	Added table of contents, list of figures, authors list, etc.
7	Stephane Zuckerman	UD	Restructured document; fixed cross-references.
8	Stephane Zuckerman	UD	Added x86/Intel perf and power numbers for DARTS.
9	Stephane Zuckerman	UD	Added perspective on applying power aware scheduling techniques to TERAFLUX arch. Added conclusion
10	Stephane Zuckerman, Jaime Arteaga	UD	Integrated Jaime's update on LDCS section.
11	Joshua Suetterlein	UD	Updates to Sections 3 and 4
12	Stephane Zuckerman	UD	Added missing references
13	Stephane Zuckerman	UD	Fixed DMM perf. and power numbers; added g500 perf and power numbers for Intel/x86 + analysis. Added Fib and DGEMM COTSon experiments.
14	Stephane Zuckerman	UD	Added parallel merge sort experiments on COTSon
15	Stephane Zuckerman	UD	Final version, minus one data point for 16core OpenMP merge sort.
16	Stephane Zuckerman	UD	Final version.
17	Stephane Zuckerman	UD	Corrections after internal review.
18	Roberto Giorgi	UNISI	Updates
19	Stephane Zuckerman	UD	Added clarifications

Release Approval

Name	Role	Date
Guang R. Gao	WP Leader	29.04.2014
Robert Giorgi	Project Coordinator for formal deliverable	16.04.2014

Contents

Table of Figures

List of contributors to the writing of the document.

Stephane Zuckerman **Jaime Arteaga**
Joshua Suetterlein **Haitao Wei**
Elkin Garcia **Guang R. Gao**
University of Delaware

Alberto Scionti, Roberto Giorgi
University of Siena

Glossary	
Codelet	Sequence of non-preemptive machine instructions
COTSon	Software framework provided under the MIT license by HP-Labs
C64	IBM Cyclops-64, a many-core chip designed for HPC
DARTS	Delaware Adaptive Run-Time System
DARTS-TSUF	The port of DARTS on the TSUF version of the Thread Scheduling Unit.
DDG	Data Dependency Graph
DMM	Dense Matrix Multiplication
DS	Dynamic Scheduling
FPU	Floating Point Unit
FMAD	Floating Multiply-Add Double
HPC	High Performance Computing
LDCS	Locality-Driven Code Scheduling
MKL	Matrix Kernel Library
MMU	Memory Management Unit
Percolation	Mechanism used to move data and/or code across the machine in a smart way
PXM	Program Execution Model
SS	Static Scheduling
Supertask	Coarse-grain structure used in LDCS to group tasks processing a common block of data
TU	Thread Unit

Executive Summary

This is the final report relating University of Delaware's work in the TERAFLUX Consortium. We expose the impact of fine-grain execution models on three different platforms: the IBM Cyclops-64 general-purpose many-core processor; off-the-shelf general-purpose multi-core x86-64 systems (from both Intel and AMD vendors); and finally, the TERAFLUX simulation environment for teradevices.

We propose several techniques that leverage the use of fine-grain multithreading to achieve high-performance and energy-efficient executions by exploiting code and data locality. One of them is a way to reduce the search space for optimal tile sizes analytically on systems which rely on programmer-managed memory, evaluating empirically the best tile shape for either performance or energy efficiency purposes. Experiments were conducted on Cyclops-64, using matrix multiplication and the LU factorization as target kernels. We provide results for both performance and energy and power efficiency.

Another technique is Locality-Driven Code Scheduling (LDCS), a way to leverage the knowledge of block-based algorithms such as LU factorization and Cholesky decomposition to perform dataflow-inspired task co-scheduling when a group of tasks are accessing the same data block. Such tasks are grouped and inlined into one *supertask*, which relaxes the traditional dataflow constraints, by allowing data-driven tasks to become *phases* in the supertask that are fired when their data dependencies are satisfied. LDCS thus allows signaling to occur in the middle of a supertask. Experiments were conducted on both Cyclops-64 and Intel Xeon-based platforms. We provide results for performance and energy and power efficiency.

A third technique describes how to automatically exploit locality using the Codelet Model and efficient static scheduling. Three algorithmic variants were evaluated and compared to a basic dynamic scheduling scheme. It was applied to several kernels, ranging from classical matrix multiplication to graph-based algorithms.

Finally, we present results based on the University of Delaware's implementation of the Codelet Model: the Delaware Adaptive Run-Time System (DARTS). We first introduce the implementation of DARTS on regular x86 platforms, including the performance and (when available) power/energy consumption results of several kernels running on off-the-shelf computing systems. We then present the port of DARTS on the TERAFLUX simulation infrastructure. We detail the trade-offs that were required to implement the Codelet Model on top of the DF-Thread/T* implementation using the T* (T-star) instruction set extension implemented in COTSon. The resulting runtime system provides a hybrid data-driven execution model that makes both DF-Threads and Codelet models converge into one. We conducted experiments to compare the execution of DARTS using the regular x86 software scheduling implementation, and the DARTS-TSU port, which uses the native Thread Scheduling Units.

1 Introduction

The main objective of the workpackage WP9 is to study the impact of the DARTS codelet runtime on teradevices. This study is decomposed into 3 steps:

1. The first step gives a preliminary study of the possibilities of applying DARTS on a large scale platform. In particular, this task consists in evaluating the opportunity to port DARTS to the COTSon simulator [Argollo09], performing an in-depth study of existing techniques of percolation, and also studying the existing works on the popular topic of energy efficiency and power-awareness.
2. The second step of WP9 is to evaluate the impact of DARTS on teradevices. This consists in a proof of concept of the percolation and power-aware scheduling techniques by performing ad-hoc development of the techniques.
3. Finally, WP9 concludes with a refined study of the impact of DARTS on teradevices.

This deliverable is focusing on the third and last step of WP9: an evaluation of a codelet runtime system on a teradevice. This document is structured as follows:

- The present section explains what is the relationship with previous deliverables (WP8, WP9), and gives a summary of the previous work;
- Section 2 introduces and describes techniques leveraging fine-grain data-driven multithreading to improve performance and energy efficiency for compute-intensive workloads.
- Section 3 presents the mechanisms that compose the Delaware Adaptive Run-Time System (DARTS), an implementation of the Codelet Model. We also present experimental results for both performance and energy efficiency.
- Section 4 describes the port of DARTS to the TERAFLUX architecture using the COTSon simulation infrastructure. We discuss the resulting execution model, which is a hybrid between the DF-Thread and Codelet models, and is the result of trade-offs that were necessary to port DARTS on the TERAFLUX simulation environment.

The integration of University of Delaware's work into TERAFLUX was therefore successful.

1.1 Relation with other deliverables

This document extends our previous work, described in D9.1 and D9.2, and in particular our efforts to the evaluation of DARTS.

1.2 Activities Referred by this Deliverable

This deliverable reports on the research carried out in the context of Task 9.3.

1.3 Summary of Previous Work

Reported in D8.1, D8.2, D9.1 and D9.2.

1.4 Description of Work of Work Package 9

We report here the DoW, for the reader convenience.

Task ID	Pre-Requisite	Partners (PM)	Task Title (start, end month) and Description
T9.1	-	UD (3), UNISI (1)	**Preliminary study of the impact of the codelet model on teradevices (start m1 (TERAFLUX_m28) - end m3 (TERAFLUX_m30))** This initial task will allow UD to study in more details the possibilities of applying the codelet model on a large scale platform such as the one proposed in TERAFLUX. In WP8, the TERAFLUX toolchain will be studied to evaluate the opportunity to port UD's codelet runtime system to the COTSon simulator and environment; in this WP9 we aim to focus more on the theoretical foundations and the study will be mostly based at UD. In particular, this task will allow UD to perform an in-depth study of existing techniques that apply percolation or any code/data movement technique close enough to be reused with percolation. Likewise, power-aware/energy-efficient scheduling is becoming a very popular topic and the study of existing literature in the field should provide very useful insights. Once this study is completed, UD will be able to determine the most productive strategy for applying data percolation and the codelet model on teradevices. The initial results will then be exposed early in the project (m3). We also wish to agree on the benchmarks (including TERAFLUX applications) to be considered.
T9.2	T9.1	UD (15), UNISI (3)	**Evaluation of the impact of the codelet model on teradevices (start m4 (TERAFLUX_m31) - end m9 (TERAFLUX_m36)** The comparison between codelets and DF-threads should help UD and its partners to decide how to best implement/adapt data percolation techniques. Whether codelets are implemented using DF-threads (fully utilizing their features or a subset) or the other way around, UD will characterize what is needed to perform the actual implementation and make a first attempt. Ad-hoc developments using the TERAFLUX toolchain and UD's codelet runtime system are performed in WP8, here mostly UD based studies will be performed and reported. UD will then evaluate the impact of these techniques on performance with respect to a selection of representative workloads. Both TERAFLUX's DF-threads and UD's codelets will be evaluated using these techniques. A first step toward generalization and automation will be proposed for percolation and power-aware scheduling techniques.
T9.3	T9.2	UD (15), UNISI (4)	**Refinements on the impact of the codelet model on teradevices (start m10 (TERAFLUX_m37) - end m21 (TERAFLUX_m48)** Once the usefulness of percolation and power-aware task scheduling are characterized and quantified, we aim to use extensively these techniques for assessing the system performance and its power estimation. The Run Time System from UD and the COTSon based implementation will be closely compared. The availability of the Cyclops-64 platform will allow us to compare the results on such platform as well. In particular, we aim to provide an in-depth analysis of: percolation applied

			to DF-threads and codelets. Power-aware task scheduling techniques will then be explored. As a final step a multi-constraint scheduling encompassing power, performance and other metrics such as temperature or fault rate will be evaluated.

Deliv. No	**Delivery month**	**Nature**	**Dissemination level**	**Deliverable Title**
D9.1	m30	R	RE	Executing a codelet runtime on teradevices: a feasibility study
D9.2	m36	R	RE	Report on data percolation on teradevices
D9.3	m51	R	PU	Evaluation of the codelet runtime system on a tera device

2 Leveraging Fine-Grain Dataflow-Inspired Multithreading on Multi and Many Core Systems

The TERAFLUX project emphasizes Program eXecution Models (PXMs) that feature fine-grain synchronization as well as event and data driven tasks, *e.g.*, DF-Threads and codelets. Using such PXMs leads to design solutions to well-known compute-intensive kernels which differ from their coarse-grain counterparts.

One important topic when targeting teradevices is power and energy efficiency. One of UD's goals for TERAFLUX was to demonstrate power-aware scheduling techniques, leveraging fine-grain event-driven execution models, and to evaluate the trade-offs between performance and power and energy efficiency.

The remainder of this section presents several techniques that leverage fine-grain synchronization and the scheduling of fine-grain dataflow-inspired threads as well as their impact on both performance and energy consumption and/or power efficiency. We also present some results of updated techniques presented in the previous deliverable (D9.2), as well as the ones presented therein, to both a general-purpose many-core processor as well as off-the-shelf multi-core x86 systems when available.

2.1 Experimental Testbed

2.1.1 A Brief Reminder of the IBM Cyclops-64 Architecture

Cyclops-64 (C64, pictured in Figure 1) is a many-core architecture designed for High Performance Computing (HPC) [Denneau11]. A C64 chip contains 160 independent single-issue thread units (TUs), up to 4.8MB of shared on-chip memory (SRAM) and 1GB of external memory (DRAM). Each pair of TUs shares one 64-bit floating point unit (FPU), one memory bank and a memory controller. The FPUs can fire one double precision "Floating point *Multiply and Add" (FMAD)* instruction per cycle for a total performance of 80 GFLOPS per chip when running at 500MHz. A 96-port crossbar network with a bandwidth of 4GB/s per port connects all TUs and SRAM banks. Execution on a C64 chip is non-preemptive and there is no hardware virtual memory manager.

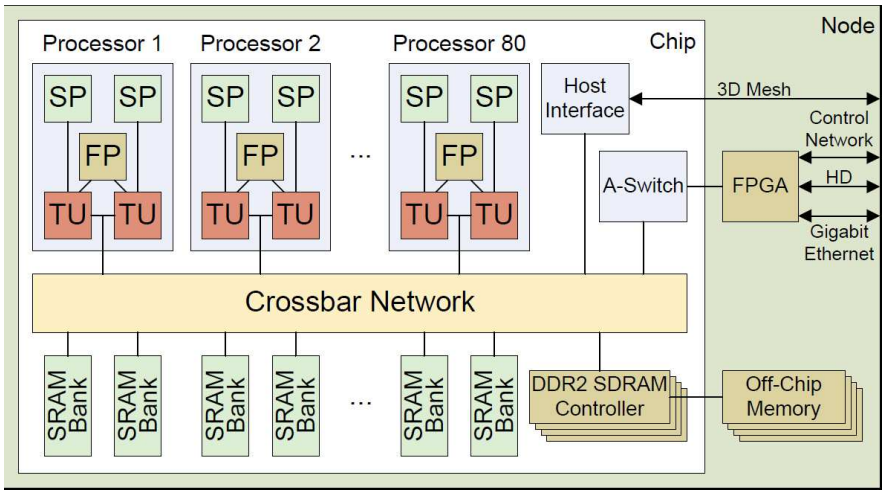


Figure 1 A Cyclops-64 node (block diagram)

2.1.2 Off-the-Shelf x86-64 Systems Used in this Study

In this study, we used several 64-bit x86-based machines. This section only uses on of them, which we call *DataServer*, and which is based on a 2-socket Intel Xeon E5-2670 CPU, featuring 8 cores per socket, two hardware threads per core, and clocked at 2.60GHz. Each Xeon processor features a shared unified 20MB L3 cache, a private unified 256KB L2, and 32KB L1 data and instruction caches. Hence, two hardware threads share both the L1 and L2 caches. This processor is based on the Sandy Bridge micro-architecture, which allows us to perform power and energy measurements using hardware counters. We obtained our power efficiency numbers using *likwid* [TreibigHagWel10].

2.2 Description and Evaluation of Techniques to Exploit Fine-Grain Event-Driven Execution Models

2.2.1 Optimizing Performance and Energy with Optimal Tile Size Search

In the past, we proposed an optimization framework, which integrates optimal tiling [GarciaEtAl10, GarciaOroGao11], as well as dynamic scheduling and dynamic percolation techniques [GarciaEtAl13, GarciaGao13] in our work to improve both performance and energy on a general purpose many-core architecture, the IBM Cyclops-64. However, the original framework features several limitations:

1. Only DRAM, shared on-chip SRAM and registers were considered;
2. Tiling was restricted to the registers; and
3. Tiles had to be square.

We have extended the previous work to integrate on-chip scratch-pad memory (SPM), which plays an important role in performance and energy efficiency in our framework. Improvements to the previous framework include:

1. Extending the tiling technique to the shared on-chip SRAM;
2. Search for optimal tile shapes, including rectangular ones; and
3. The SPM is now used as a cache to improve performance and energy consumption.

Please note that the following has not been published yet.

2.2.1.1 Improving our Optimal Tile Search on C64

We formulate the energy optimization for matrix multiply on Cyclops in Equations (1)-(3). There are two levels of tiling: from DRAM to SRAM, and from SRAM to register. We perform the matrix multiplication $C = A \times B$, assuming that the matrices A, B and C are in DRAM, and of dimensions $N \times N$. We define the tiled matrix multiplication from DRAM to SRAM as $C_{Y \times Y} = A_{Y \times X} \times B_{X \times Y}$, and the tile size from SRAM to register as $C_{L_2 \times L_2} = A_{L_2 \times L_1} \times B_{L_1 \times L_2}$. We used the double-buffering technique to perform latency-hiding. Therefore, we have 6 buffers in SRAM or registers. All the following equations (from Eq. (1) to Eq. (5)) are derived from the research done by Garcia et al. [GarciaOroGao11].

$$2 * L_1 L_2 + L_2^2 \leq R_{max} \quad (1)$$

$$2 * XY + X^2 \leq SRAM_{max} \quad (2)$$

Where $SRAM_{max}$ is the capacity of SRAM, and R_{max} is the maximum number of registers. $L_2 \times L_1$: tile size of row×column tile from SRAM to register; $X \times Y$: tile size of row×column tile from DRAM to SRAM. Inequations (1) and (2) show the capacity constraints for the buffers. N is the size of the matrices in DRAM – we assume matrices of size $N \times N$.

$$Energy_{Static} = \frac{e_0 2N^3 \left(\frac{1}{L_2} + \frac{1}{Y} + 1\right)}{Pmax * Freq} \quad (3)$$

$$Energy_{Dynamic} = N^3 \times \left[\left(\frac{2}{L_2} + \frac{1}{Y}\right) e_{Rsram} + \frac{1}{Y} e_{Wsram} + \frac{2}{X} e_{Rdram} + e_{Fmad}\right] + N^2 \times e_{Wdram} \quad (4)$$

Where e_0 is the static energy coefficient, and e_{Rsram}, e_{Wsram} , e_{Rdram}, , e_{Wdram} , and $e_{Fmad}$the energy spent by SRAM read instructions, SRAM write instructions, DRAM read instructions, DRAM write instructions, and FMAD (Floating Multiply-Add Double) instructions respectively.

Equation (3) describes static energy consumption, while Equation (4) represents dynamic energy consumption. Equation (5) minimizes the total energy consumption of matrix multiply by accumulating both static energy and dynamic energy consumption.

$$\text{Minimize } Energy_{Total} = Energy_{Static} + Energy_{Dynamic} \quad (5)$$

The larger the tile size, the better the solution will be. An efficient way of narrowing the search space is to us the method of Lagrange Multipliers to obtain an optimal tile size. To do so, Inequations (1) and (2) must be turned into regular equations. We can then apply the Lagrange Multipliers method. We then finally get the solution by empirically and exhaustively search the tile size space, which was drastically reduced thanks to our previous computations: $L1 = 1$ and $L2 = 6$ are the best values on the C64 platform. In order to simplify our implementation, we chose the multiple of 6 that is the closest to the optimal solution of X and Y as our final choice.

2.2.1.2 Evaluation of our Optimal Tile Size Search Methodology

By applying the methodology for optimal tile shape and size finding described in Section 2.2.1, we find that the optimal $X \times Y$ tile dimensions should be 288×48 for the C64 platform. We compared our solution with the previously (non-optimal) square solution yielding $X = Y = 192$ in our experiments.

We compared the performance and energy between the one using SPM and the one without in Figure 2 and Figure 3. Both of them use our tiling technique at the register level, dynamic scheduling for task execution and dynamic percolation from DRAM to SRAM (See D9.2). We list the main differences below:

1. The blue bar (“n192-m192”) represents the case using fixed square tile sizes in SRAM, without using the scratchpads. 192x192 is the maximum square size which can be held in SRAM.

2. The red bar ("n288-n48-spm") represents the case using a more efficient tiling at both SRAM SPM levels. 288x48 is the optimal tile size for energy and performance under the capacity limitation of SRAM.

From the figures, we observe a significant advantage of using scratchpad memory to enhance energy efficiency: efficient tiling shows a constant energy savings for different matrix sizes. Overall, we observe performance improvements in a 6.4%-7.5% range, and energy efficiency improvements in a 7.3%-8.3% range.

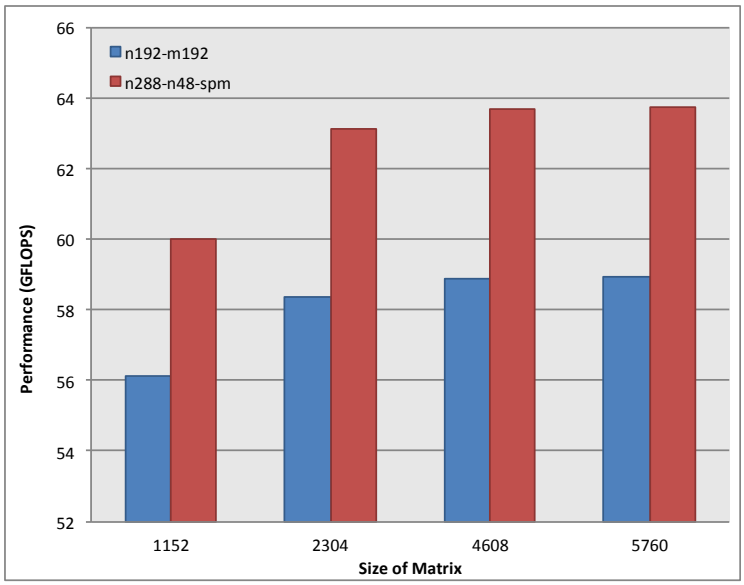


Figure 2 Performance comparison using fixed-size square tiles without scratchpad memory (blue bars) and variable-size tiles with scratchpad memory (red bars), while varying problem size (n=size of matrix).

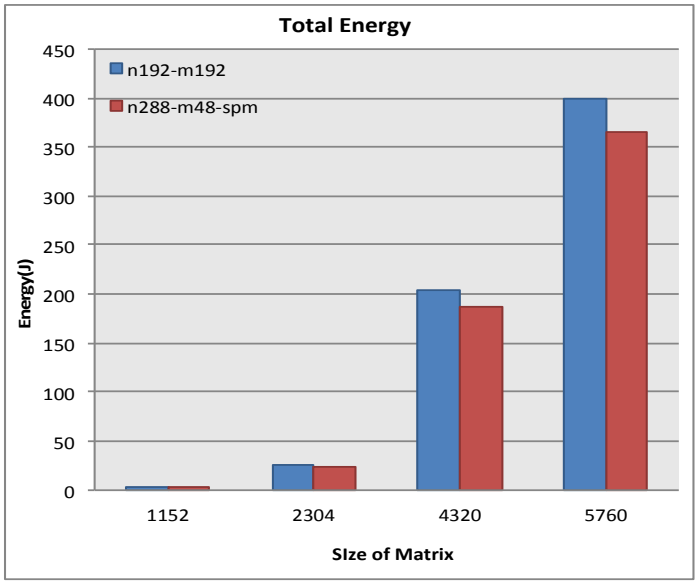


Figure 3 Energy comparison between: without (blue bars) and with (red bars) scratchpad memory, while varying problem size (n=size of matrix).

2.2.1.3 Further Discussion – Performance Tiling vs. Energy Tiling

It may seem intuitive that by optimizing for performance, the resulting tile shape and size will also automatically result in a more energy efficient execution. However, this is not true in several cases. Figure 4 illustrates this fact using matrix multiplication (and leveraging the energy model we described above).

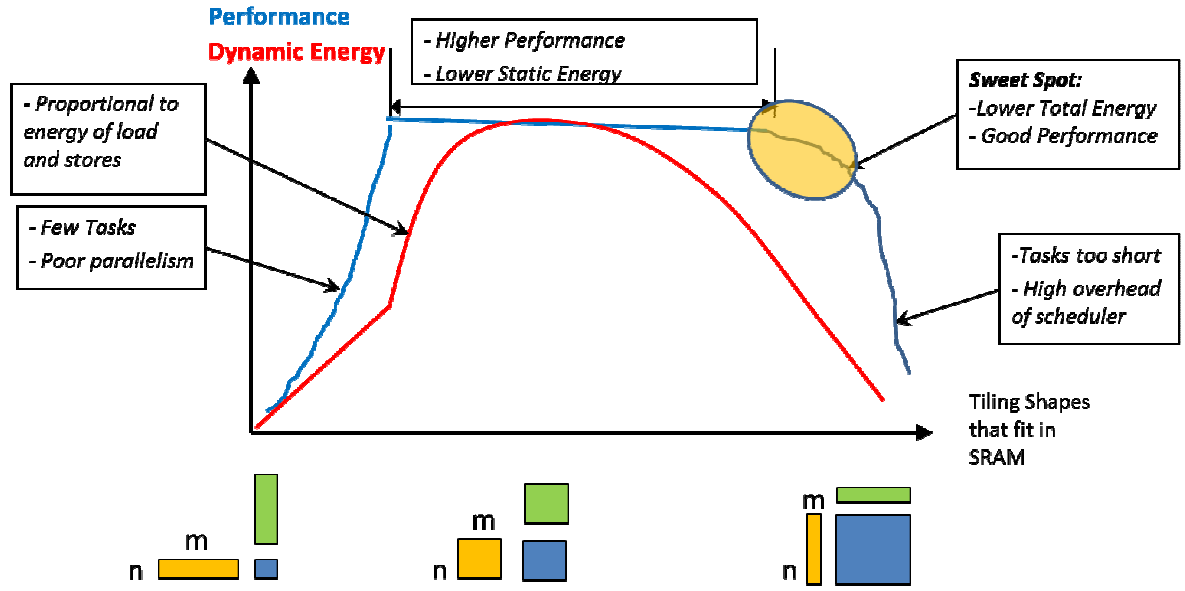


Figure 4 Tiling for Performance vs. Tiling for Energy. Not all tile shapes yield the best result according to a user goal.

We studied the tradeoffs between optimizations for performance and energy efficiency. We found that while optimizing for performance decreases also the static energy consumption (related to leakage currents and total execution time), it does not necessarily decrease dynamic energy (related to instructions executed). We explored the tiling space for matrix multiply in C64 and found that while several tiles shapes can produce similar performance (and similar static energy consumption), variations in dynamic energy make a major difference. In this order of ideas we found that the best tiling for performance is not necessarily the best tiling for energy consumption.

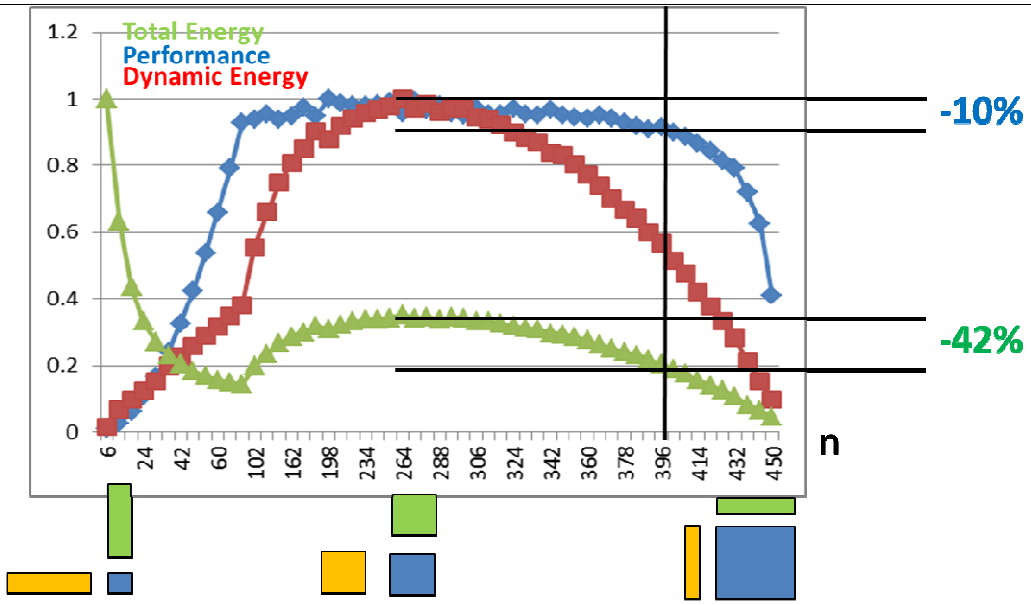


Figure 5 Impact of tile shape and size on performance and energy consumption. Bottom: various tile shapes produce various read/write patterns, thus resulting in vastly different memory transfer and reuse, impacting both performance and energy consumption.

In our particular example, allowing a 10% loss on the maximum performance can result in energy savings of up to 40%. The results we just discussed are presented in Figure 5. These experimental results confirm that our energy model is sound.

Some of our recent work also shows evidence of this trend for different kinds of kernels, *e.g.* LU factorization on C64 [GarciaEtAl13].

2.2.1.4 Considerations to Apply Optimal Tile Size Search for the TERAFLUX Architecture

The work presented in this section was successfully applied to the Cyclops-64 architecture. They rely (among other things) on dataflow-inspired multithreading (*i.e.*, tasks run only when their data is available, thus reducing data movement to times when it is only needed), coupled with the availability of on-chip cache-less SRAM and scratchpad memory directly addressed by the running program.

The TERAFLUX architecture experimented on x86-based, off-the-shelf components with added units, such as fault-detection and thread scheduling units. This will make the dataflow part of the program run much better than a software-only solution like the one we used for C64. However, while we could rely on a rather precise data movement and locality analysis to compute the ideal tile size for either performance or energy efficiency (or a trade-off between the two), the addition of cache components will require further experiments, and to add new constraints to our model – or at the very least, to our empirical search space once it has been computed. Indeed, replacing scratchpads with caches imply dealing with coherence protocols, and their possible downsides: false-sharing, capacity misses, etc. In no way does this make this technique non-applicable to x86-based processors with caches, but it does mean that further modeling is required to adapt it to cache-based architectures such as the one proposed in TERAFLUX. This being said, caches bring better programmability to the high-level programmer, who does not have to deal with explicit data movements from one memory level to the

other: in the C64 platform, the programmer must explicitly declare where in the memory hierarchy a piece of memory must be allocated (*i.e.*, DRAM, SRAM, or scratchpad), and any movement from one level to the other must be made explicit through copy instructions. This is likely to greatly hamper a programmer's productivity for application programming.

2.2.2 Locality-Driven Code Scheduling

One of the main causes for elevated power consumption in an application could be a large amount of data movements across the different levels of the memory hierarchy. As a result, several techniques have been proposed to increase the reuse and the locality of data in parallel applications. These techniques are known as locality-aware scheduling algorithms and they focus on determining the best scheduling algorithm for the assignment of work to hardware threads considering constraints such as spatial and temporal data locality, latencies, cache misses and hits, etc. in a single chip with a software-managed memory hierarchy.

The remainder of this section introduces the concept of supertask, as described by Arteaga et al. [ArteagaEtAl14].

2.2.2.1 Introduction to Supertasks

We believe that an application can benefit more from a locality-aware scheduling technique in a fine-grain programming model by grouping tasks that process a common block of data in a single coarse-grain construct called *supertask*, which requires dependence satisfaction in the middle of its execution. We call this technique Locality-Driven Code Scheduling (LDCS).

Operational semantics of supertasks are derived from dataflow semantics, and in particular macro-dataflow. A supertask is comprised of several phases that execute in sequence. Each phase is tied to a set of dependence signals and is triggered when external data it depends on has been fully updated. Supertasks provide several advantages: they improve data reuse, drastically reduce scheduling overheads, and, as phases are inlined within a supertask, they make the economy of function calls. This reduction in the amount of data movement is directly translated into improvements in the execution time of the application.

In order to achieve this, the programmer must select an appropriate size for the block of data so this one can fit in one of the upper levels of the memory hierarchy of the target platform, along with any other data required by the supertask for its processing. If the Data Dependence Graph (DDG) of an application is known, the steps a programmer needs to follow to implement LDCS are:

1. Determine the number of blocks of data to be produced by the application and their associated tasks.
2. For each block, create a supertask with all the corresponding tasks.
3. Assign dynamically supertasks to available hardware threads. Prioritize supertasks containing tasks in the critical path of the DDG.
4. Execute each supertask following the algorithm in Figure 6.
5. Repeat steps 3 and 4 until all supertasks have been assigned and processed.

Algorithm 1 LDCS: Execution of a Super-Task by a Hardware Thread

```
1: procedure PROCDATABLOCK(NP = Number of Phases, ND[NP] = Number of Dependencies for each Phase)
2:     for (p=0; p < NP; p++) do
3:         Wait for ND[p]'s dependencies to be satisfied
4:         if p==0 then
5:             Read the block of data of the super-task.
6:         end if
7:         Read any other data required.
8:         Process the block of data with phase p
9:     end for
10:    Write the block of data back into main memory
11:    Signal any thread(s) waiting for this block of data
12:    Make hardware thread available
13: end procedure
```

Figure 6 The Supertask Execution Algorithm: a High-Level View.

2.2.2.2 LU Factorization using LDCS

As an example of an application that can benefit from LDCS, Figure 7 presents the DDG for the LU factorization of an $M \times M$ matrix A, which has been divided in blocks of $T \times T$elements for an efficient blocked implementation. LDCS can be implemented by identifying in the DDG and grouping in a single supertask those tasks processing a common block of data, as shown in Figure 8.

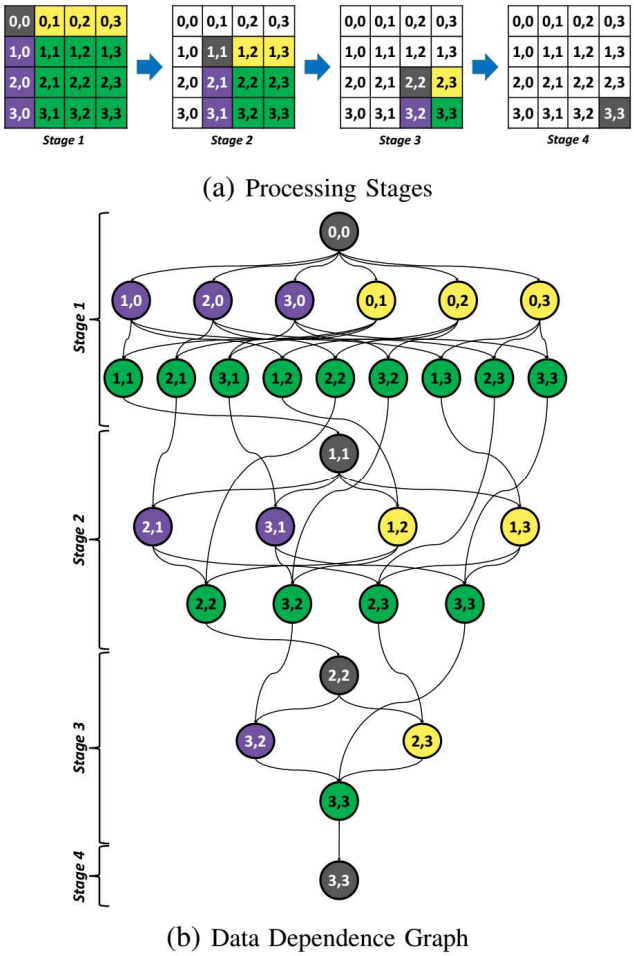


Figure 7 Classical Blocked LU factorization: GETRF tasks are dark gray, TSTRF tasks are purple, GESSM tasks are yellow, SSSSM tasks are green.

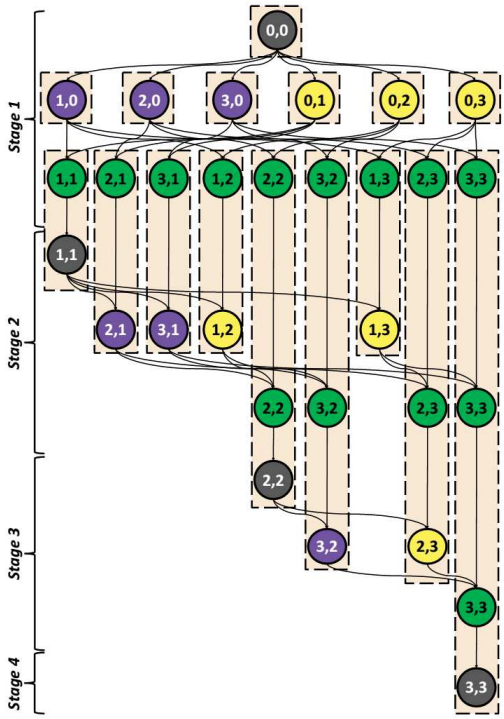
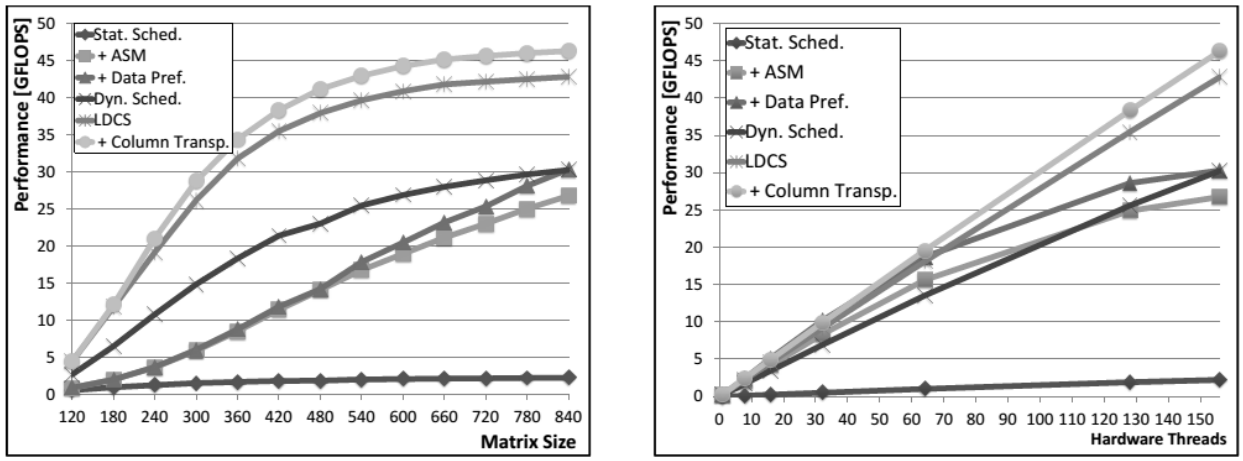


Figure 8 Data Dependence Graph of the LU factorization algorithm using LDCS. GETRF tasks are dark gray, TSTRF tasks are purple, GESSM tasks are yellow, SSSSM tasks are green, and light-orange dashed boxes enclose tasks computed by the same hardware thread and containing a supertask.

2.2.2.3 Evaluating Locality-Driven Code Scheduling on C64

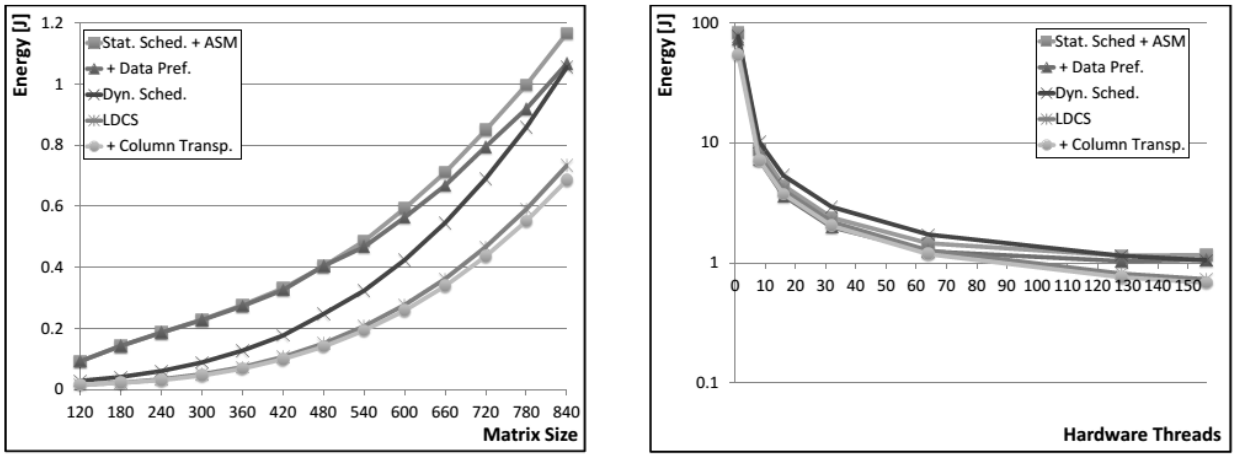
The following are the versions of LU factorization implemented on C64 in order to evaluate the benefits obtained with LDCS.

Version	**Features**
Stat. Sched.	Static scheduling of tasks in C
+ASM	Supertasks written in assembly
+Data pref.	Software pipelining and loop unrolling
Dyn. Sched.	With dynamic task scheduling, using 6×6 tiles
LDCS	With LDCS as described in Figure 6
+Column Transp.	With transposed storing of data to exploit C64 features



(a) Weak Scaling using 156 Hardware Threads. (b) Strong Scaling using an 840×840 Matrix.

Figure 9 Performance of LU factorization on C64. Higher is better.



(a) Weak Scaling using 156 Hardware Threads. (b) Strong Scaling using an 840×840 Matrix (logarithmic scale).

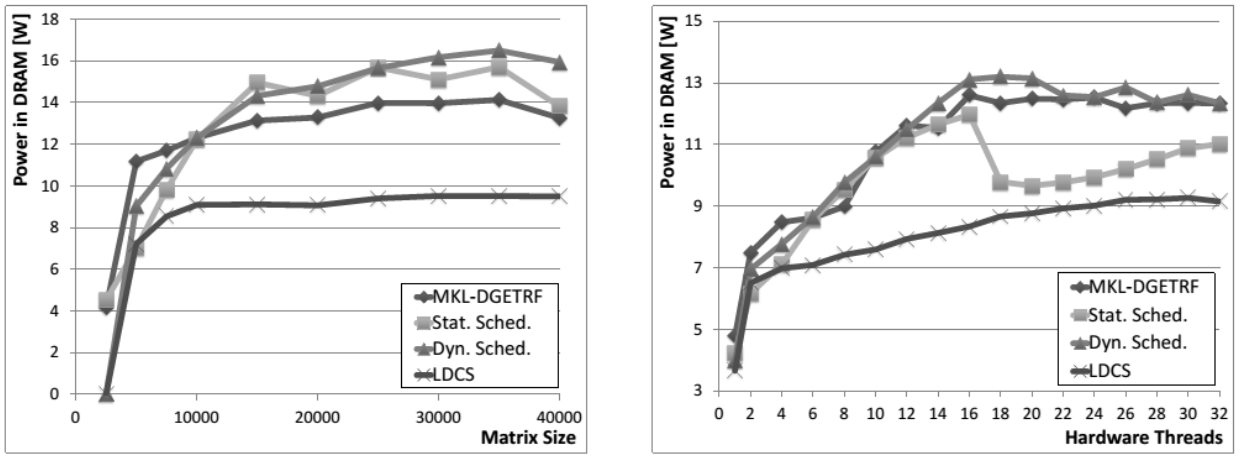
Figure 10 Total Energy Consumption of LU factorization on C64. Lower is better.

Figure 9 and Figure 10 show that LDCS can effectively improve the power efficiency of an application. On architectures with software-managed memory hierarchy such as C64, an improvement of 72% on average in weak scaling was obtained in comparison with a dynamic scheduling version of the application. Performance is also greatly increased thanks to the complete control the programmer has on the content of all the memory levels.

2.2.2.4 Evaluating Locality-Driven Code Scheduling on Off-the-Shelf Multi-Core Systems

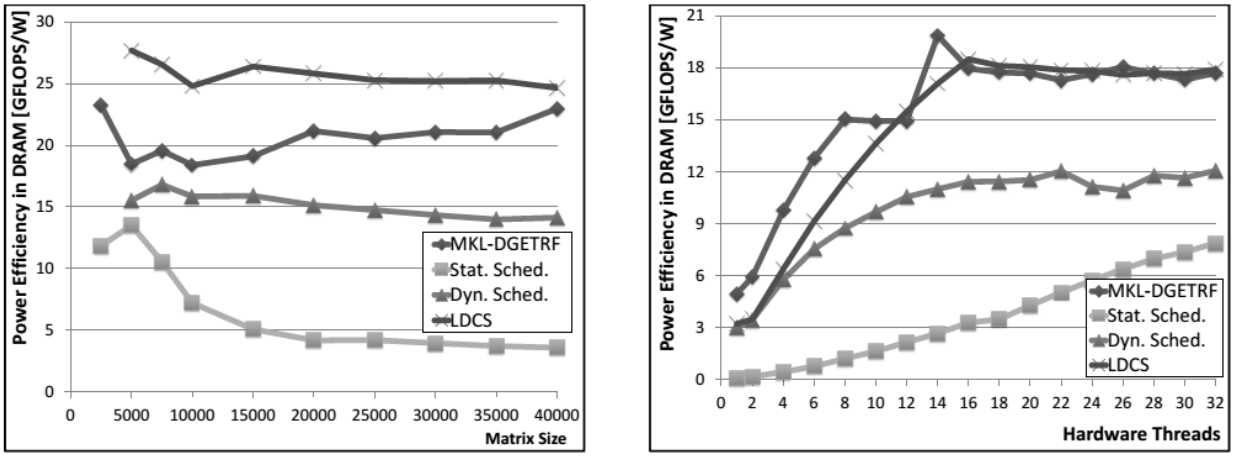
LDCS was also evaluated on x86 platforms using *DataServer* described in Section 2.1.2 and compared against an implementation using Intel's Matrix Kernel Library (MKL).The variants used in our experiments with x86 are described below.

Version	Features
MKL-DGETRF	With CBLAS' `DGETRF`
Stat. Sched.	Pthreads with CBLAS' `DAXPY` and `DGEMM`, using 100×100 tiles
Dyn. Sched.	Same with dynamic task scheduling
LDCS	Same using LDCS as described in Figure 6



(a) Weak Scaling using 32 Hardware Threads. (b) Strong Scaling using a $10k \times 10k$ Matrix.

Figure 11 Average DRAM Power Consumption of LU factorization on DataServer. Lower is better.



(a) Weak Scaling using 32 Hardware Threads. (b) Strong Scaling using a $10k \times 10k$ Matrix.

Figure 12 DRAM Power Efficiency of LU factorization on Intel Xeon. Higher is better.

On an architecture with hardware data caches, such as x86, LDCS improves the DRAM power efficiency of the application by 28% on average in weak scaling, versus a highly optimized version of the application using Intel's MKL, as can be seen on Figure 11 and Figure 12. On this architecture, LDCS' performance is competitive due to the increase in data locality obtained by executing with the same hardware thread all the tasks that process a common data block and by inlining such tasks in a single supertask.

2.2.2.5 Steps Required to Make Supertasks Run on the TERAFLUX Architecture

Some trade-offs must be made to allow supertasks on a TERAFLUX machine. While our current implementation of supertasks is based on codelet-like data-driven threads that are inlined to reduce scheduling overhead, it is not reasonable to expect the Thread Scheduling Units (TSUs) to be able to do so using the T* Instruction Set Extensions [Giorgi12] (cf. D6.2) implemented in the TERAFLUX simulation environment. However, at the cost of a slightly more complex scheduler, it could probably be possible to obtain a result that will be very close in practice to what we obtained in a fully software way.

The INRIA and HP partners have worked on a new version of the TSU which, among other things, allow newly created DF-Threads that are not yet scheduled for execution to be assigned to a given node. The (Distributed) TSUs still decide to perform work-stealing however they want within a given node. However, if a variant of the `df_constrain` instruction could be produced, it could not only pin a given DF-Thread to a node, but a supertask could be emulated as follows:

1. Run the first phase of a supertask as a DF-Thread.
2. As a first action, create a new DF-Thread for the next phase. ID will be stored somewhere in shared-memory so that the other supertasks can signal it when they are done processing their data block.
3. The dependence count of this phase is equal to the number of outside supertasks which will signal it, plus one (so that the current phase can signal it when it is done).
4. Add a constraint on the new DF-Thread handle so that it can only run on the current core.
5. Each phase follows the same steps from 2-4 for its successor phase.

Appendix A proposes some pseudo code to show how supertasks could be generated.

2.2.3 Automatic Locality Exploitation Using Static Codelet Scheduling

2.2.3.1 Problem Description

Current codelet scheduling approaches primarily focus on balancing workloads and reducing scheduling overhead in effort to increase performance. This leaves programmers responsible for manually exploiting locality at the cost of programming productivity. The following technique explores the automation of locality exploitation among codelets.

To motivate the exploitation of locality in the codelet model, we present the following example. Figure 13 shows 6 codelets and their dependencies.

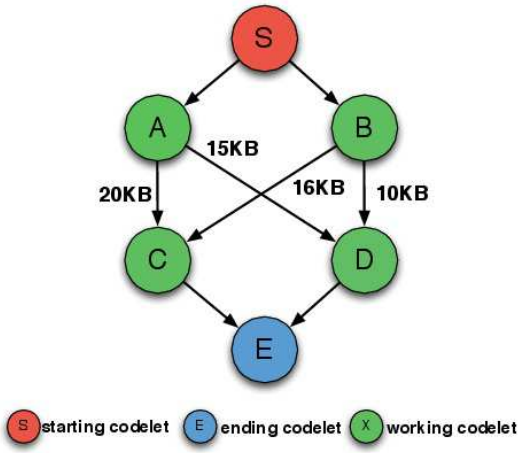


Figure 13: An example of codelet graph. Arcs are data dependencies weighted with the amount of data which will be manipulated by each codelet.

The starting and ending codelets do not affect the exploitation of locality. The 4 codelets in the middle are working codelets. The arrows from A and B to C and D indicate data dependencies between the codelets. A codelet is unable to begin its execution until its dependencies are satisfied. The numbers on each arrow signifies the data generated by source of the arrow and consumed by the sink. This number also indicates potential locality. For example, the arrow between A and C specifies that 20KB of data produced by A will be consumed by C. In general, A has to store this 20KB data into shared memory since it guarantees that C is able to access the data no matter where C is executed. However, if A and C are scheduled to the same core, A does not need to store the 20KB data into the shared memory. Instead, A may store the data into the core's local storage for future access of C. In such a way, we exploit the locality between A and C. In general, there may be multiple choices to exploit locality. For example, we may schedule AC on one core and BD on the other. This scheduling plan exploits 30KB locality. However, the best plan for this example is to schedule AD on one core and BC on the other, which exploits 31KB locality. For a more complicated case that contains many codelets and dependencies, there may be exponential selections. It would be hard for a programmer to figure out the optimal schedule exploiting maximum locality.

With the previous motivation we can informally introduce the *Best Schedule Problem*. Assuming the CDG is static and the information of potential locality is known, can a schedule be generated that partitions the CDG into groups of adjacent codelets which may be assigned to cores that maximizes locality? By partitioning the CDG into several groups of codelets, we can generate a static schedule. Each group may be assigned to a single core. Then adjacent codelets (a pair of codelets that are executed contiguously on the same core) may use local storage as a buffer to pass data. It reduces not only the latency of the memory access, but also saves energy since data is produced and consumed in place. Since the schedule is static, the execution order of the codelets assigned to the same group must be fixed. In other words, the codelets belonging to the same group are totally ordered in the CDG. The generated schedule should guarantee the maximum amount of potential locality is exploited.

We propose three algorithms to solve the Best Scheduling Problem. The three algorithms have different trade-offs in the algorithmic complexity, locality exploitation, program performance, energy efficiency, and required computation resources. The features of the three algorithms are as follows:

The first solution converts the Best Scheduling Problem to a "min-cost" flow problem. Given the weighted codelet graph, we can create a flow network that has two properties:

1. Each scheduling plan corresponds to a flow in the flow network, and vice versa; and
2. The sum of available weights in a scheduling plan and the cost of the corresponding flow are anticorrelated.

A min-cost flow algorithm finds the flow that has minimum cost among all possible flows. Applying the above two properties, we know that the corresponding scheduling plan is the one with maximum sum of available weights among all the plans. Therefore, the solution of the min-cost flow problem corresponds to the solution of the Best Scheduling Problem. This solution is guaranteed to be optimal. Furthermore, the time complexity is $O(knm\log(n))$ where *k* is the number of cores, *n* is the number of codelets, and *m* is the number of dependencies in the codelet graph.

The second approach uses a heuristic algorithm (called "max-first" algorithm) to provide a near-optimal solution for the Best Scheduling Problem. The main idea of the algorithm is to schedules the two codelets with maximum potential locality to some adjacent position on the same core at every step. The max first algorithm has lower time complexity than the min-cost flow based algorithm. Leveraging a heap data structure in its implementation, the max first algorithm has a time complexity of $O(n\log(n)+m)$ where *n* is the total number of codelets and *m* is the total number of dependencies.

The final approach converts the Best Scheduling Problem to a graph partitioning problem. A graph partitioning algorithm partitions the vertices of a weighted graph into multiple groups. It guarantees that the sum of inter-group weights (*i.e.*, the weights of edges that go across groups) is minimal or nearly minimal. By applying the graph partitioning algorithm on a codelet graph, we may partition the codelets into groups equal to the total number of cores. Then the codelets belonging to the same group will be scheduled to the same core. The minimum sum of inter-group weights indicates that the schedule minimizes the waste of inter-core locality. This approach has a time complexity of $O(m\log(k))$ where *m* is the number of dependencies, and k is the number of cores. This approach's generated schedule may suffer unnecessary serialization and is not guaranteed to be optimal.

2.2.3.2 Evaluation of Automatic Static Codelet Scheduling on C64

To study the three approaches presented, we developed an emulation platform of the IBM Cyclops-64 many-core architecture. The platform consists of two parts, the scheduling plan generator and the runtime scheduling emulator. The scheduling plan generator uses a static codelet graph, the potential locality information, and the total number of cores to generate a schedule according to one of four approaches listed below.

- Base – basic scheduling without locality exploitation
- MCF – Min-cost flow based algorithm
- MF – Max-first algorithm

- GP – Graph partitioning based algorithm

The generated schedule is then inputted into the runtime scheduling emulator along with the number and types of instructions in each codelet. The outputs of the module are the exploited locality, performance, and energy consumption.

We use the following six applications in our experiments:

- `mm` (matrix multiplication kernel): This benchmark is based on the previous study of matrix multiplication on C64 [GarciaEtAl10].
- `ms` (merge sort kernel): This benchmark computes a sorting of 10K integers via a 7-level merge process.
- `rt_ci` (random tree with computation-intensive codelets): This is a randomly generated tree-structure codelet graph. The codelet graph contains 160 compute-intensive codelets.
- `rt_mi` (random tree with memory-intensive codelets): This is also a randomly generated tree-structure codelet graph with 160 codelets. Each codelet however is memory-intensive.
- `rg_ci` (random graph with computation-intensive codelets): This benchmark is similar to rt_ci. However, the codelet graph is a randomly generated graph with 160 codelets and 320 dependency edges.
- `rg_mi` (random graph with memory-intensive codelets): This benchmark is similar to rt_mi. However, the codelet graph is a randomly generated graph with 160 codelets and 320 dependency edges.

Figure 14 shows the best locality exploitation of three algorithms (MCF, MF, and GP, described in Section 2.2.3) applied on the six applications. We do not show the result of Base because it does not exploit locality.

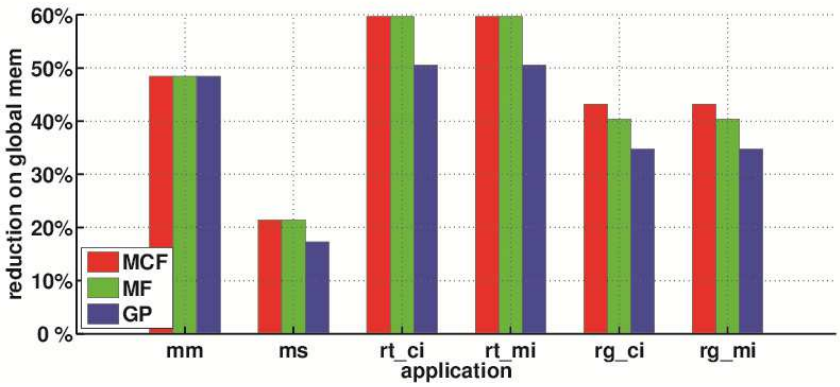


Figure 14 Reduction of memory movements using various automatic static codelet scheduling. The X-axis presents the six kernels on which we experimented. The Y-axis yields the locality exploitation value, that is, the percentage of global memory accesses that have been reduced via buffer in local storages.

Figure 15 shows the performance evaluation of the four algorithms on various applications. To make the comparison fair, all the algorithms use the same amount of cores. We set the amount to be equivalent to the requirement of MF because it is the only algorithm that does not support arbitrary number of cores.

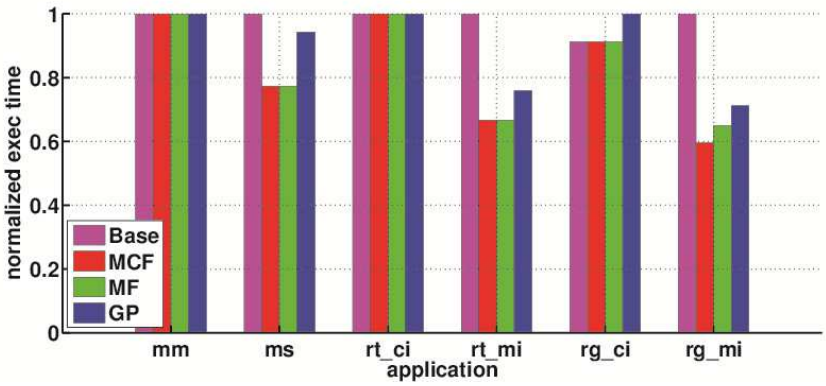


Figure 15 Performance evaluation of automatic static codelet scheduling. The X-axis represents the various kernels. The Y-axis features the normalized execution time of each application by using the four scheduling algorithms, respectively.

Lastly in Figure 16 we present the normalized overall energy consumption of the four algorithms on various kernels. The overall energy consumption of an application consists of static and dynamic energy consumptions. The static energy consumption is determined by the execution time. The dynamic energy is determined by the number and type of instructions executed.

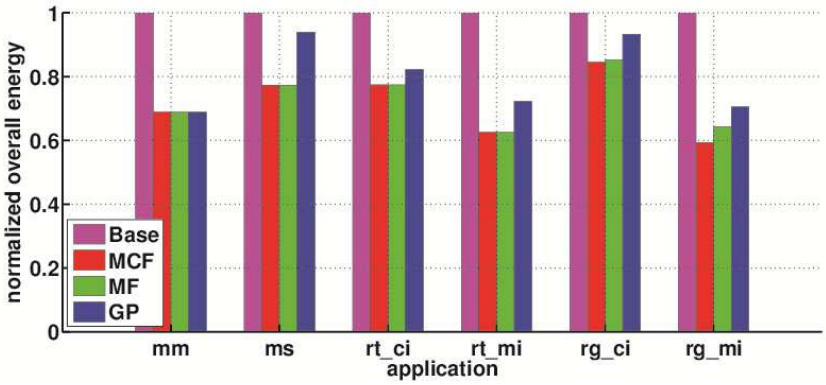


Figure 16 Overall normalized energy consumption using different variants on selected kernels.

Our major observations can be summarized as follows:

- MCF always exhibits best locality exploitation. It reduces up to 59.7% of global memory accesses. MF is the second best (within 7.0% of difference comparing to MCF).
- The applications using MCF outperform the same applications using the other scheduling algorithms. MCF achieves up to 68.1% of performance improvement comparing to Base. MF is the second best (within 9.1% of difference comparing to MCF).

MCF exhibits best energy reduction on both overall and dynamic energy consumptions. It reduces up to 40.7% overall energy and 59.2% dynamic energy comparing to Base. MF is the second best (within 8.5% of difference on overall energy and 3.6% on dynamic energy comparing to MCF).

2.2.3.3 Discussion – Applying Automatic Locality Exploitation on the TERAFLUX Architecture

To be applied to the TERAFLUX architecture using the DF-Thread or codelet models, this technique combines the caveats expressed to apply optimal tile size selection for performance and energy in Section 2.2.1.4, and the additional mechanisms required by locality-driven code scheduling and emulating supertasks, as explained in Section 2.2.2.5.

Since our technique statically schedules tasks, weighting data dependency arcs according to the amount of data moved between them, generating the actual code is not the problem. However, as explained before, the presence of caches, while a boon for the programmer, makes it harder to evaluate latencies to correctly place the weights on the arcs. It is far from impossible, but still requires further experimentation, as well as consider several cases, namely in-cache behaviors (say, by considering accesses to the last level of cache as the upper bound for memory latencies), and out-of-cache ones (to take into account cache misses and DRAM latencies and bandwidth). In addition, as with LDCS, this technique requires the ability to schedule several tasks to the same core. Hence, some additional TSU instruction to constrain DF-Threads to the same core is necessary.

3 Implementing the Codelet Model on Off-the-Shelf Multi-Core Systems

The Delaware Adaptive Run-Time System (DARTS) is the University of Delaware's implementation of the Codelet Model. There already exist runtime system implementations of the codelet model currently under development, such as SWARM [LauderdaleKhan12]. While they reuse the codelet object as the central unit of computation, they generally tend to stray from the original specification. Hence, our goal is to build a runtime system which will be true to the codelet model, but also serve as a research vehicle to evaluate and advance the model itself. We thus emphasized the following goals when we designed DARTS:

- ***Faithfulness***: DARTS is implemented to be faithful to the base codelet model. Hence, it employs codelets as the base unit of computation, but it also requires the use of threaded procedures as the containers for codelets.

- ***Portability and Modularity***: DARTS is written in C++. This language is low-level enough to ensure full control of the underlying hardware, while offering an object-oriented model which encourages modularity and component reuse. The latter point is important as we intend to use DARTS to explore and stretch the limit of the codelet PXM.

Codelets are (small) pieces of sequential code that are non-preemptive and event-driven: they are ready to be scheduled when all their data dependencies are satisfied, and that all required resources (*e.g.*, bandwidth or power envelope requirements) are fulfilled.

The codelet Abstract Machine Model (codelet AMM) consists of many nodes connected together via an interconnection network. Each node is expected to have several chips containing hundreds of cores. Interconnects with varying latencies will connect components at multiple levels. We envision two types of cores. The first is a simple Computation Unit (CU) which is responsible performing

operations. The other is a Scheduling Unit (SU) which is responsible for steering computation. Figure 17depicts the proposed abstract machine model.

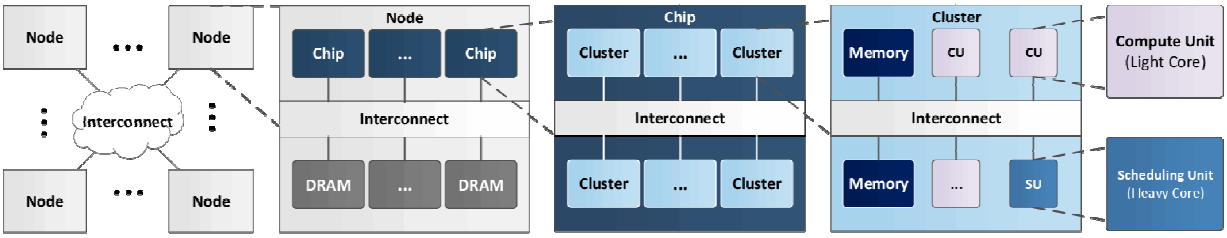


Figure 17 The Codelet Abstract Machine Model

This abstract machine is very close to the one used by the TERAFLUX project (cf. D6.2, D6.4, in fact, the difference is mostly in terms of terminology). The Codelet Model itself was already presented in previous deliverables [D9.1, D9.2] as well as peer-reviewed publications [ZuckermanEtAl11, SuetterleinZucGao13].

The codelet AMM we just described requires a concrete mapping to a physical machine. We reused the `hwloc` library [BroquedisEtAl10] to obtain the topology of the underlying computation node. Once discovered, the runtime decides how to decompose the hardware resources (processing elements, caches, etc.) according to the user-programmer's selection of preset configurations. For example, one can elect a single socket of an SMP system to act as the AMM's cluster, and a single core on the socket to act as the scheduling unit. New mappings can easily be added to the description of the codelet AMM.

Each cluster contains two types of cores, one SU and several CUs. Each core runs one of two types of schedulers. Each CU runs a micro-scheduler, responsible primarily for executing codelets. An SU runs a Threaded Procedure scheduler (TP scheduler) which is responsible for load balancing TPs between clusters, instantiating codelets, and distributing codelets within a cluster. Having designed DARTS with modularity as a guiding principle, each scheduler is capable of running several different scheduling algorithms. For the scope of this work, we use a work-stealing policy similar to Cilk [BlumofeEtAl95] to perform load balancing between TP schedulers. Within a cluster, micro-schedulers use a centralized queue to get work.

The codelet specification is implemented as a `Codelet` class containing a synchronization slot (*sync slot*) and a method called `fire`. The sync slot is used to keep track of the outstanding dependencies. The `Codelet` class must be specialized (i.e. derived) and can be instantiated once the `fire` method is expressed. `fire` is applied on a codelet by a CU's micro-scheduler when the codelet is chosen for execution. Each sync slot is initialized with the number of events the codelet requires to run. Codelets within a TP are known statically and can be accessed through the TP frame. The address of a codelet is required to signal codelets outside a TP, and can be provided at runtime. DARTS implements a form of argument fetching dataflow [GaoHumWon90], as the act of signaling is dissociated from passing data. For this reason data is written first, and then a codelet is signaled.

DARTS uses asynchronous functions called Threaded Procedures as the main way to instantiate portions of the computation graph. Much like codelets, threaded procedures are implemented as classes that must be derived by the programmer. The `ThreadedProcedure` class embeds an active codelet counter (to know when all the codelets it contains have finished executing), a pointer to a parent TP (the one which invoked it), and a member function to add a new codelet within the TP. The address of the TP frame (in practice, the pointer to the TP instance) is passed along to codelets so that

they can access shared variables. Once the last codelet of an instantiated TP has finished running, the TP is deallocated along with all the codelets it contained.

Currently, DARTS implements three types of loops: a serial loop, a TP parallel for loop, and a codelet parallel for loop. Parallel for loops (*forall*) prohibit loop-carried dependencies, conceptually executing all iterations in parallel. Practically, the iterations are executed when sufficient hardware is available. The TP forall creates a TP for each iteration of the loop, permitting the iterations to run on any cluster. The codelet forall loop adds all the iterations to the invoking TP, pinning them to a single cluster.

Conceptually, a codelet loop requires two codelets, as shown in Figure 18. These "loop controllers" act as a source and sink. The source codelet is signaled normally. Upon execution, the source schedules copies of the enclosed CDG. After the loop body has finished executing, the "leaf" codelets of each iteration signal the sink codelet. Once all iterations have completed, the sink codelet deallocates the copied iterations, and signals the next codelet in the CDG. In practice, the source and sink codelets which control the loop are merged into one, to avoid useless memory allocations. Once it has performed its source action, the loop controller is reset to the number of "leaf" codelets multiplied by the number of iterations prior to scheduling the loop iterations. This approach is sufficient for supporting nested loops.

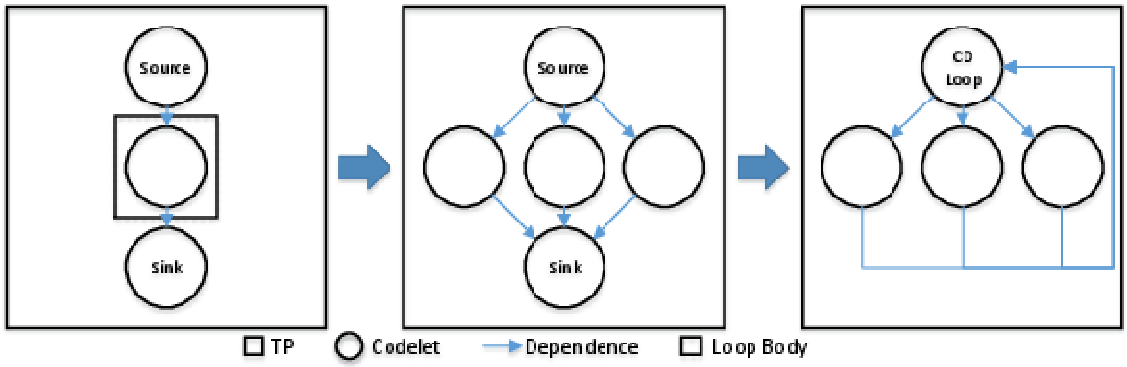


Figure 18 Implementation of Loops in DARTS

3.1.1 DARTS' Performance on x86-64

We propose the study of two benchmarks to evaluate the potential of fine-grain, event-driven multithreading on off-the-shelf x86-based machines: Matrix multiplication, and Graph500.

3.1.1.1 Experimental Testbed

For these experiments, we used a 4-socket AMD Opteron 6234 ("Interlagos") multicore system. Each socket yields two 6MB unified L3 caches; 6 2MB unified L2 caches (each shared by two cores); 12 private 16KB L1 data caches; and 12 private 32KB instruction caches. There are 12 cores in total (per socket), each sharing a floating-point unit with another core.

Software-wise, we used GCC v4.6.1 for all our tests, as well as the AMD Core Math Library (ACML) v5.3.0.

3.1.1.2 First Benchmark: Dense Matrix Multiplication

We compared our DARTS implementation to the ACML parallel implementation of the Double GEneral Matrix Multiplication (DGEMM). The ACML's implementation uses OpenMP to parallelize the computation. To ensure fairness, we reused the serial version of ACML's DGEMM, and used it within our codelets to compute DGEMM. Details on how we parallelized DGEMM for DARTS, the number of repetitions for each case, *etc.*, are available in our Euro-Par publication [SuetterleinZucGao13]. We use the sequential execution time of DGEMM/ACML as our baseline.

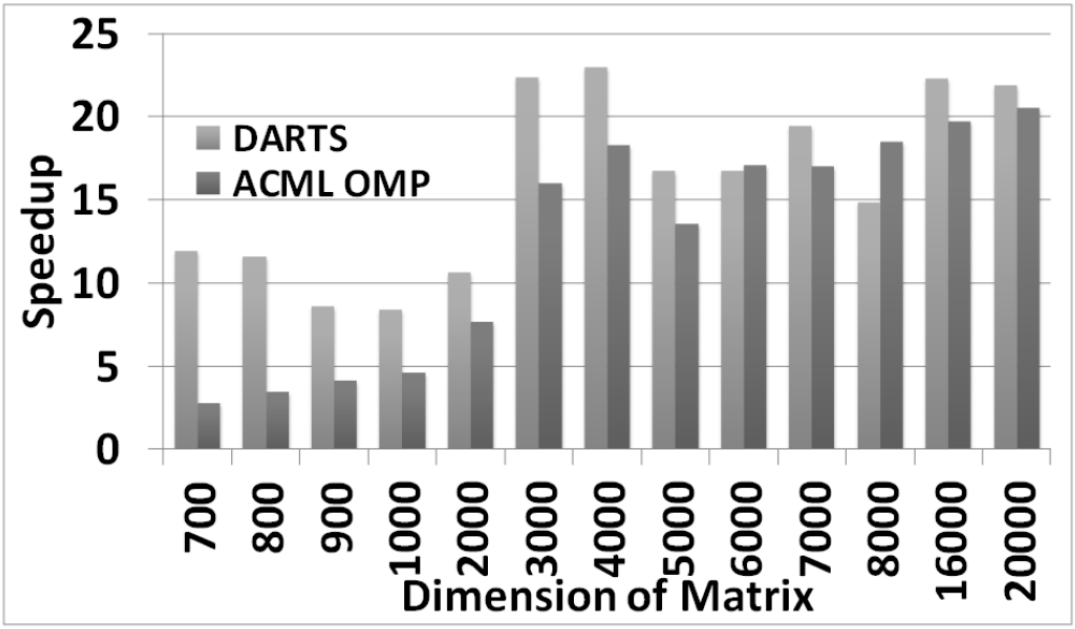


Figure 19 DGEMM Weak Scaling Case: OpenMP vs. DARTS. 48 cores are being used. All matrices are square. Higher is better.

As Figure 19 shows, for matrix sizes that are sufficiently big (700 and above), DARTS shows a better speedup, with a 1.4 × improvement on average. DARTS wins on a fully loaded machine because of two phenomena:

1. FPUs are completely contended, as OpenMP statically scheduled parallel-for loops launch all the work at the same time, and
2. Memory banks get more contended in the OpenMP case, while in the DARTS case, signaling allows for some delay between load and store requests.

It is worth noting that on a relatively low number of cores, such contention on memory banks is difficult to notice, as the bandwidth is sufficient and caches hide the remaining latencies well enough. With a high core count however, contention is unavoidable.

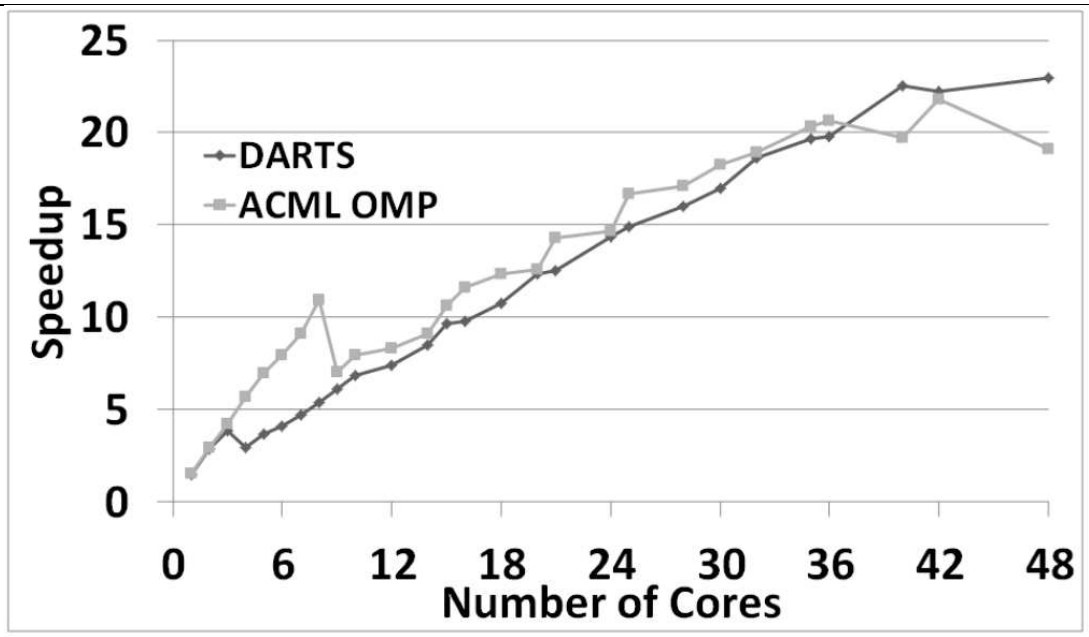


Figure 20 DGEMM Strong Scaling Case: OpenMP vs. DARTS. Higher is better.

Figure 20 shows strong scalability results for DGEMM. Up to 12 cores on a single socket, OpenMP clearly outperforms DARTS. The single-level scheduling featured by OpenMP is clearly helped by such as configuration. However, as the number of cores increases and "overflows" on more than one socket, DARTS' hierarchical scheduling catches up, to the point where it outperforms ACML/OpenMP at 48 threads. Still, there is a ~8% gap between DARTS and OpenMP in favor of the latter. This gap is seriously narrower as soon as more than 12 cores are solicited.

The use of DGEMM as a benchmark is meant to show that, even though more software mechanisms are required to run codelets, *e.g.,* to keep track of threaded-procedures and their frames, to ensure codelets are signaled when some of their input data is made available, *etc.*, DARTS remains competitive with state-of-the-art implementations on off-the-shelf multi-core systems, even without hardware help. On the contrary, the next benchmark is intended to show where DARTS, and dataflow-inspired program execution models in general, can easily shine: programs that feature irregular data and/or control flow.

3.1.1.3 Second Benchmark: Graph500

Graph500 [MurphyEtAl10] is a benchmarks that proposes to measure several kernels related to graph processing. We have compared our implementation of Graph500's second kernel, *breadth-first search* (BFS), to its reference implementation. To ensure fairness, we used the exact same tools to generate the same pseudo-randomly generated graph as an input to both versions. We also kept most of the code from the original reference implementation, only adding the boiler plate necessary to run DARTS. As with the DGEMM case, implementation details on how we built the codelet graph for the BFS kernel are available in our publication [SuetterleinZucGao13].

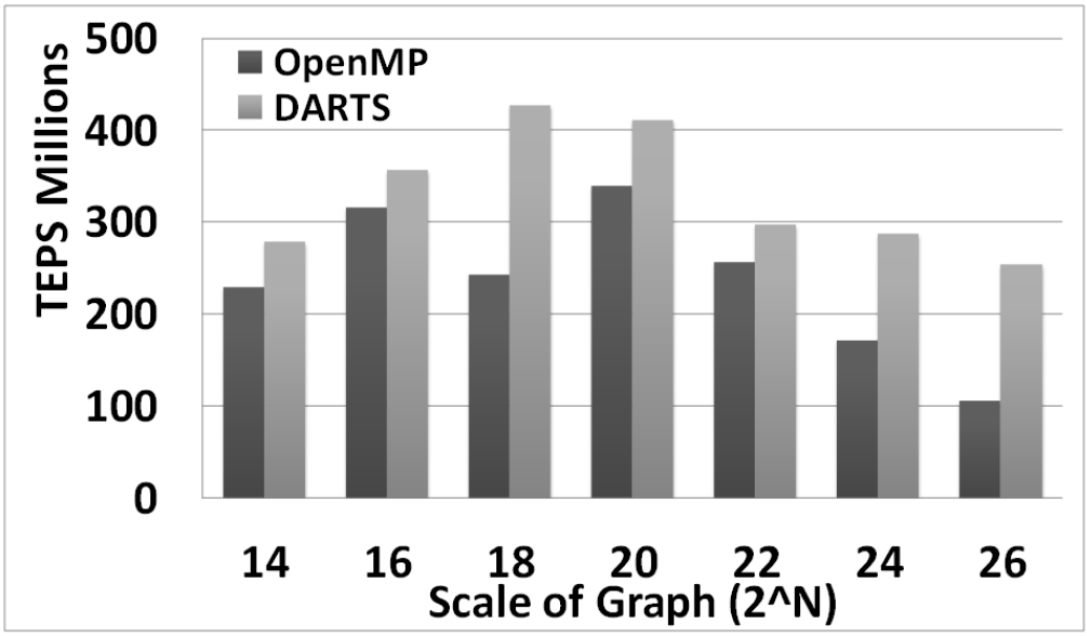


Figure 21 Graph500: OpenMP vs. DARTS. X-axis: The number of input vertices. On the Y-axis: The number of traversed edges per second (TEPS).

As Figure 21 illustrates, DARTS' implementation significantly outperforms OpenMP. There are several important notes to make, however:

1. As we stated earlier, we only modified the BFS part of the code itself. The point was to compare two execution models: OpenMP's and DARTS', and not optimize the underlying data structures. There are many publications which describe how to improve the execution of Graph500 by reshaping the vertex list, and careful use of atomic operations, but none which made their implementation available to the best of our knowledge, at the time these experiments were conducted.
2. We only reported the results of OpenMP obtained with static scheduling. We did try to modify the reference code to use dynamic scheduling, thus enabling chunks to have different sizes, but in our experiments, static scheduling always finished first.

On average, our implementation yields a speedup from 1.15 × to 2.38 × as the graph size increases.

3.1.2 Energy and Power Efficiency of DARTS on x86-64 Platforms: DGEMM

We evaluated our DGEMM kernel on *DataServer*, a 32-thread machine described in Section 2.1.2. We recompiled DARTS with Intel's C++ Compiler (ICC) v14.0, and used Intel's Math Kernel Library (MKL) to run our experiments: We either ran the parallel DGEMM using MKL's implementation based on icc's OpenMP runtime, or we used the serial DGEMM from the MKL in conjunction with DARTS. We first present performance numbers for both strong and weak scaling (as we did in Section 3.1.1.2), and then introduce power efficiency numbers.

In addition, we have decided to evaluate three different scheduling policies for DARTS: a purely static scheduling policy, akin to OpenMP's, a dynamic one which also follows OpenMP, and a work-

stealing policy. Until now, work-stealing had been described to move codelet sub-graphs across the machine DARTS was running upon. With this last scheduling policy, even within a given cluster of cores, codelets can be stolen between local schedulers.

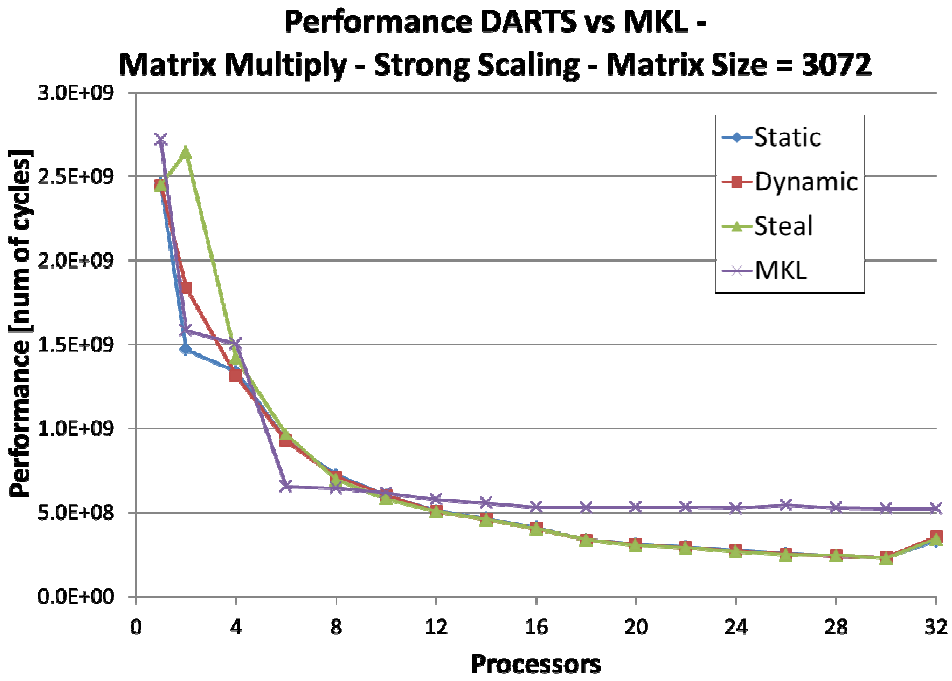


Figure 22 DGEMM, Strong Scaling case: DARTS vs. parallel MKL. Performance for strong scaling. Matrix size: 3072×3072. Lower is better.

As Figure 22 shows, dynamic and/or work-stealing policies perform slightly better than the MKL, while the static policy is on par with it.

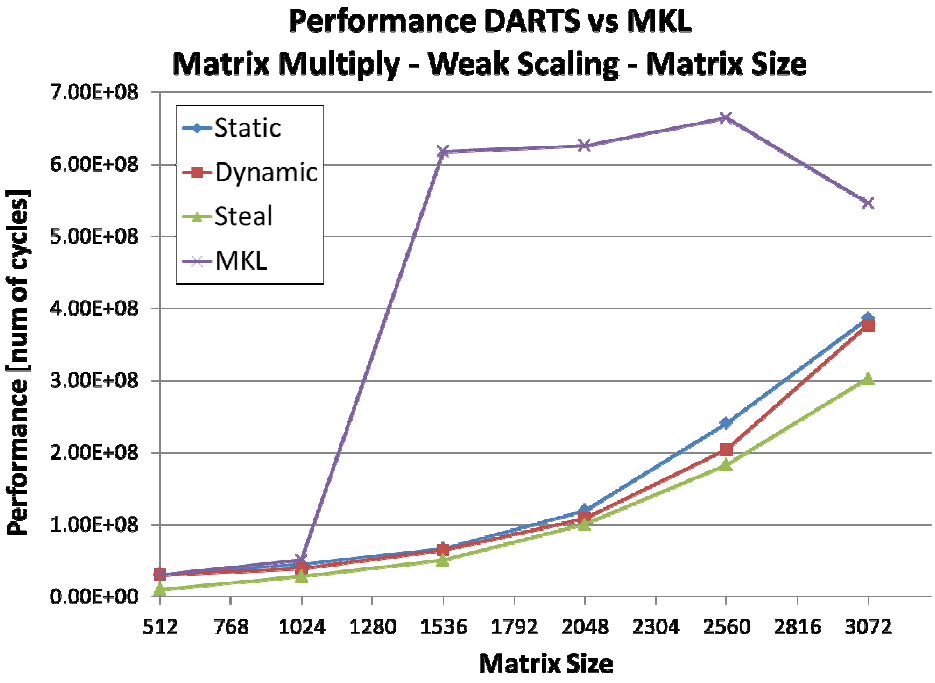


Figure 23 DGEMM, Weak Scaling Case: DARTS vs. parallel MKL. Running on 32 hardware threads. Lower is better.

Likewise, weak scaling shows a slight advantage in favor of DARTS, for all scheduling policies. The results are not as good in terms of power efficiency however.

From Figure 24 it is obvious that the MKL is much more power-efficient than any of the DARTS scheduling policies in the strong scaling case. Likewise, Figure **25** shows a similar trend in the weak scaling case.

This requires further investigation, but we suspect that individual threads are more solicited with "meta-work" (*e.g.*., local codelet queue management as well as threaded procedure management) than the more "flat" parallelism yielded by the MKL. In addition, our experience with MKL is that the "shape" of the blocks that are passed to the sequential MKL's DGEMM kernel have a strong impact on the overall performance [ZuckermanPerJal08].

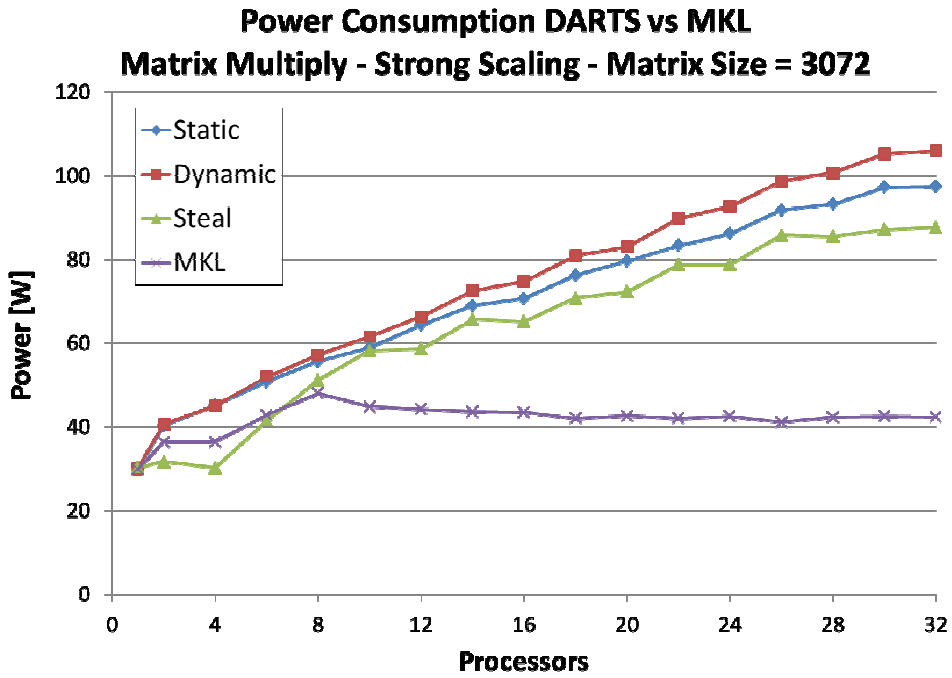


Figure 24 DGEMM, Strong Scaling Case: DARTS vs. parallel MKL. Power Consumption. Matrix size: 3072×3072. Lower is better.

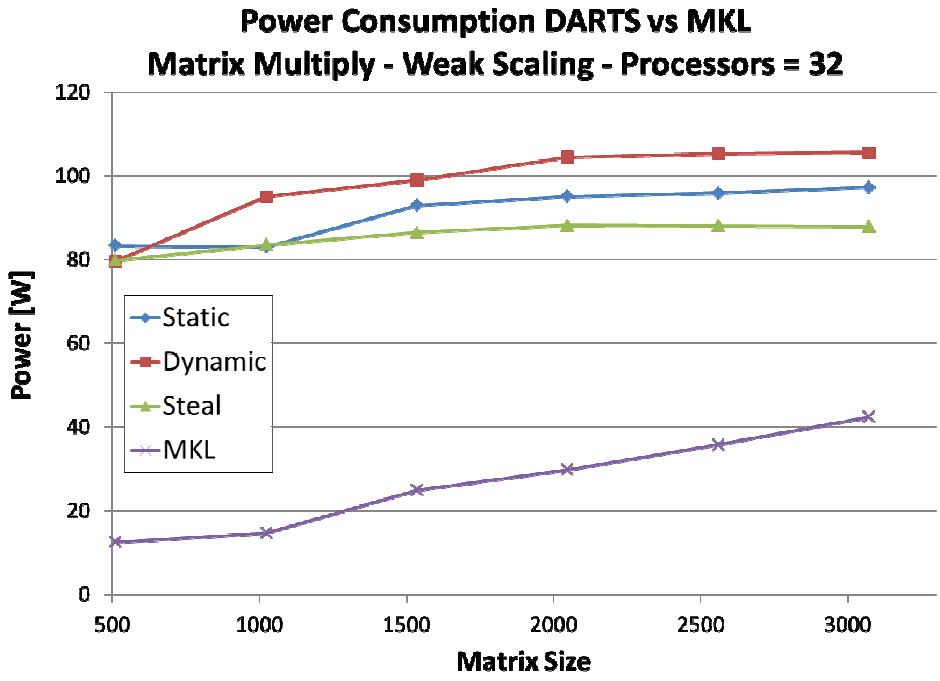


Figure 25 DGEMM, Weak Scaling Case: DARTS vs. parallel MKL. Power Consumption. Lower is better.

3.1.3 Energy and Power Efficiency of DARTS on x86-64 Platforms: Graph500

We also ran a new set of experiments using the Graph500 benchmark, and using the *DataServer* machine to also measure performance and power on this machine.

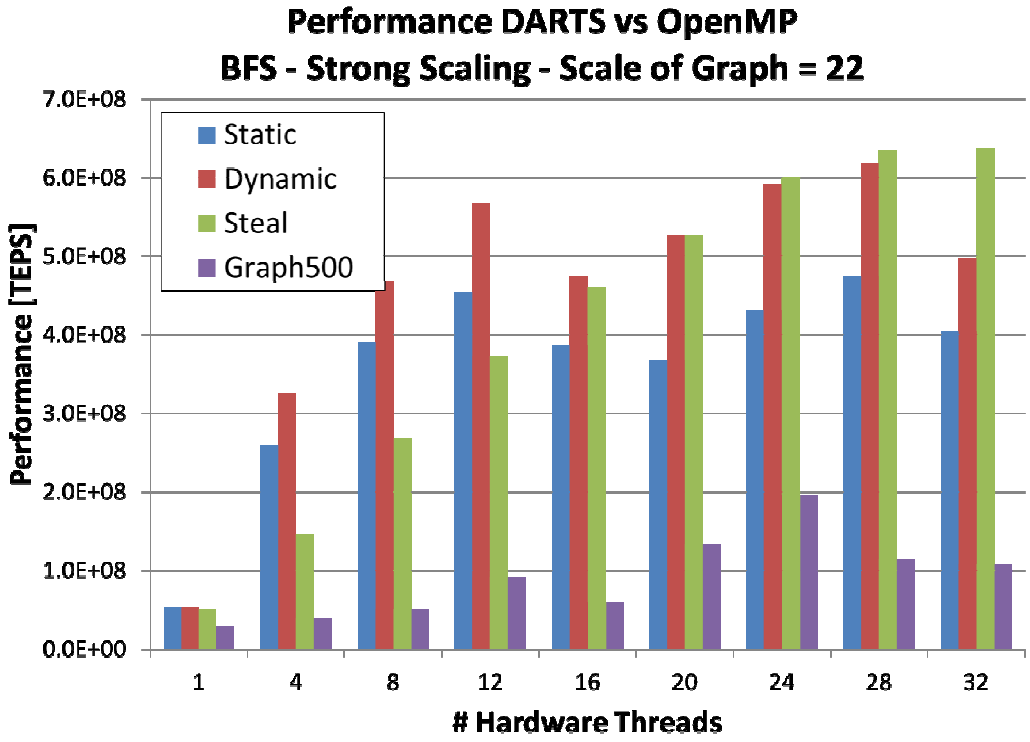


Figure 26 Graph500, Strong Scaling Case: Performance. "Graph500" is the performance of the reference code running with OpenMP. Higher is better.

Again, we evaluated the strong scaling performance of DARTS using our three different policies (Static, Dynamic, and Work Stealing) against the OpenMP version of the reference code. The implementation was not changed compared with the one described in Section 3.1.1.3, with the exception of the experimentation with different scheduling policies. In the original experiments, for both DGEMM and Graph500 (shown in Sections 3.1.1.2 and 3.1.1.3), we used the default dynamic scheduling policy. Whether for strong or weak scaling (shown in Figure 26 and Figure 27), DARTS' implementation clearly outperforms OpenMP's. An interesting observation is that although the static scheduling policy is mimicking OpenMP's static scheduling for *parallel for* loops, the 2-level scheduling (threaded procedures and codelets) allows to efficiently deal with variable granularity and work load imbalance much better than what OpenMP alone. In addition, the dataflow mechanisms used by codelets allow for a better resource usage, as only those threads that are effectively ready to run will actually access the memory subsystems.

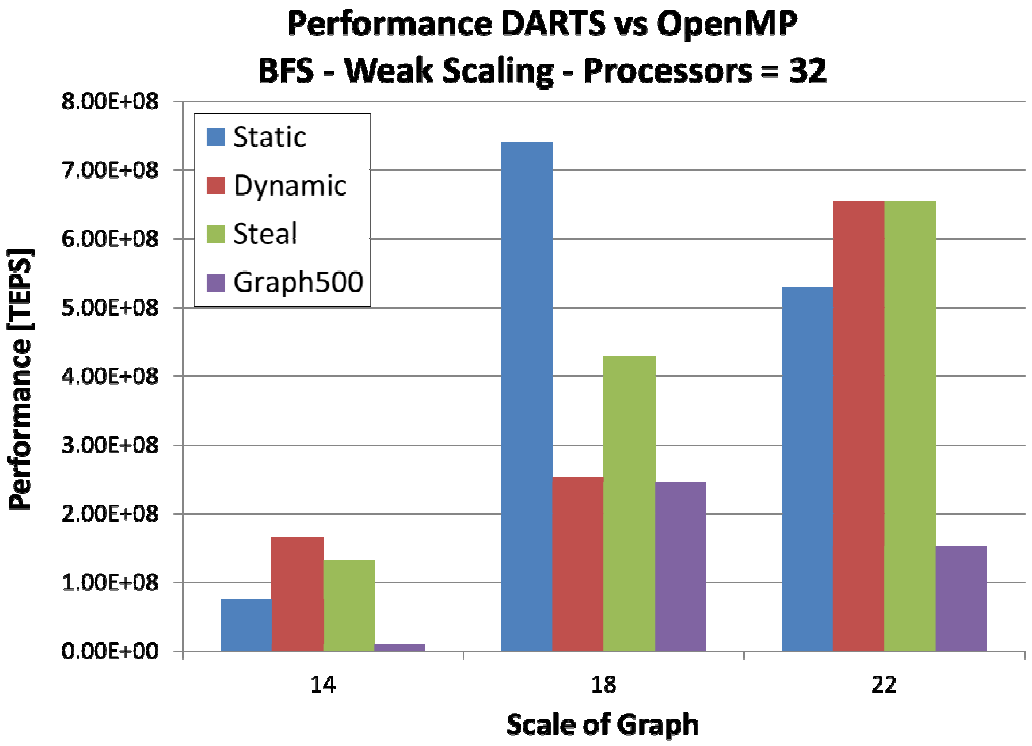


Figure 27 Graph500, Weak Scaling Case: Performance. "Graph500" is the performance of the reference code running with OpenMP. Higher is better.

The data collected for power consumption is presented (respectively) in Figure 28 in the case of strong scaling and Figure 29 for weak scaling. As with DGEMM, we used the *likwid* tool to measure power. However, due to some counter overflow, not all data is represented.

Overall, power efficiency is much better for the DARTS implementation in the strong scaling case, as pictured in Figure 28. This is probably a "mechanical" effect of the execution time being so much lower than the reference code time, and is likely to improve as the input data set grows. With smaller input sets, such as those depicted in Figure 29, and as with the DGEMM case, power consumption does not quite match the performance we observe. While lower, it is clear that the threaded procedure schedulers, as well as the micro-schedulers running on each core have a cost on power consumption.

However this is only a preliminary set of results and further investigation must be performed. In addition, we haven't taken advantage of the DVFS capabilities of the Sandy Bridge architecture yet. While in the DGEMM case this is probably going to yield poor results with respect to performance, in the Graph500 case, there are certainly opportunities to lower the voltage/frequency levels so that additional power is saved.

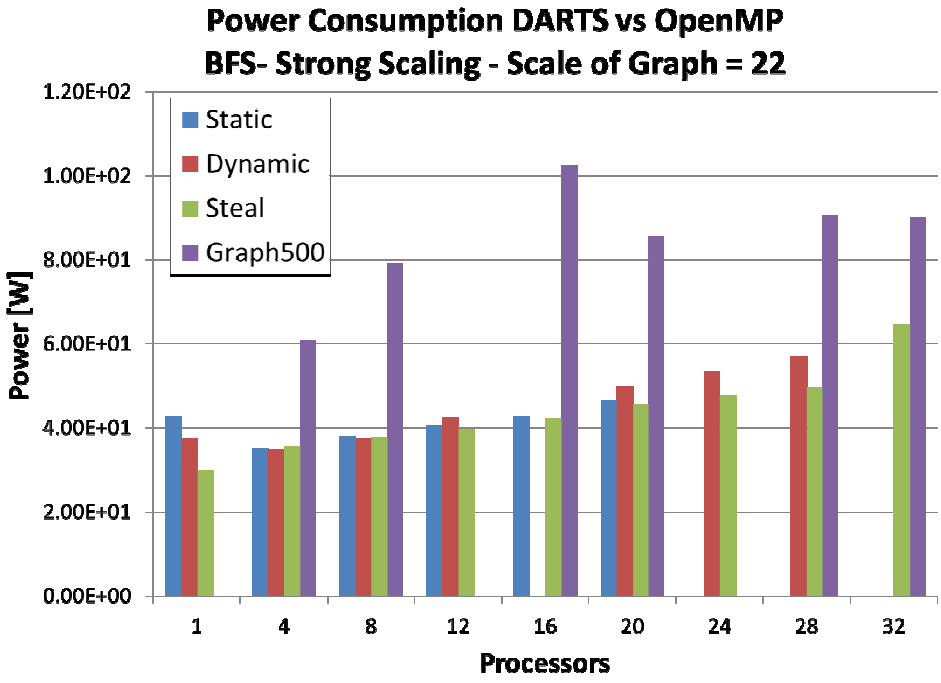


Figure 28 Graph500, Strong Scaling Case: Power Consumption. "Graph500" is the power consumption of the reference code running with OpenMP. Lower is better.

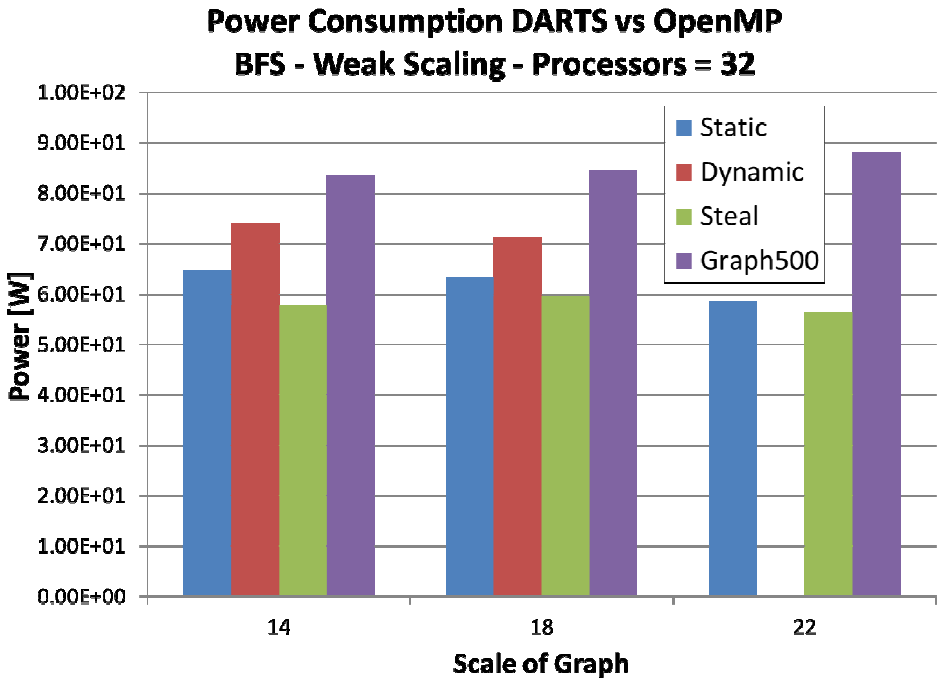


Figure 29 Graph500, Weak Scaling Case: Power Consumption. "Graph500" is the power consumption of the reference code running with OpenMP. Lower is better.

4 Porting DARTS to the TERAFLUX Simulation Infrastructure

This section describes the efforts to port DARTS to the TERAFLUX simulator, COTSon, as well as the implementation trade-offs that were made as a result. We first start by describing how DARTS was ported, and continue with some selected results on the simulation platform. It is worth mentioning the timeline of our porting DARTS to COTSon. It started as a simple feasibility study (as described in D9.1), where we only ported the "pure" DARTS runtime, with full software scheduling. We then proceeded to modify our runtime to take advantage of the DF-Threads/T* (cf. D7.2, D6.2) TSU. A first port of DARTS for the TSU4 (implemented by UNISI) was thus finalized in January 2013. A second port of DARTS, this time for the TSUF version (implemented by HP) was completed in January 2014. Both TSU-based implementations are confined to a single-node execution for now, but we plan on modifying DARTS to be able to run on TSUF with multiple nodes.

Most of the details that follow are valid for both the TSU4 and TSUF implementations of DARTS. The experiments detailed in Section 4.2 were all performed using DARTS-TSUF. A first mention of this work was expressed in a collective TERAFLUX publication [GiorgiEtAl2014] [SolinasEtAl13]

4.1 Merging Codelets and DF-Threads: DARTS-on-COTSon

4.1.1 Overview of the DF-Threads/Codelets merge

While both execution models are very close, DF-Threads and codelets differ in a few but significant ways, as explained in D8.2. We explain again some of those aspects when they are relevant to the discussion of porting DARTS to COTSon.

4.1.2 Units of computations: Accessing data from DF-Threads and Codelets

Both DF-Threads access data using a frame. However who can see such data being read or written, and what can be accessed varies widely between the two models.

1. A DF-Thread has its own input frame (the *df-frame*) and can access several output frames. DF-Threads are supposed to access variables in four different types of memory (Frame Memory (FM), Owner Writable Memory (OWM), Transactional Memory (TM), and Thread Local Storage (TLS)) each with its own consistency (cf. D7.1, D7.2, D6.2, and D3.5).
2. By contrast, codelets are contained within a threaded procedure, and have no real frame of their own: They access variables from the *TP frame*, and all codelets belonging to the same TP can access the same variables and memory locations.

We have opted to port DARTS on top of COTSon, which implies that we want to use the native instruction set extensions of the already-implemented DF-Thread/T* model as building blocks. The compromise we came to is the following: Each codelet will be assigned to a DF-Thread. To ensure that the various codelets instantiated in a TP frame are fully initialized before they can be run, an extra dependency is added to them. When the `add` call is issued, a call to `df_tdecrease` is issued to remove this extra dependency.

4.1.3 Invoking Threaded Procedures

According to the Codelet Model specification, codelets cannot be invoked directly from the code by the programmer: They must be contained in asynchronous functions called Threaded Procedures. Our goal was to keep as much of the original DARTS API as possible. However, the reference counting required in the original DARTS implementation is useless on COTSon, since the Thread Scheduling Unit has no knowledge of codelets, and the memory used to allocate the TP frame and the codelets contexts are also invisible to it. Thus a compromise had to be reached.

Calling a threaded procedure is done by issuing a call to `invoke`:

`invoke<SomeThreadedProcedure>(parameters)`

The class **`SomeThreadedProcedure`** is the actual asynchronous function we desire to run. We feed it the parameters required by the function. The actual sequence of operations is a bit complex, and involves the creation of a temporary DF-Thread. When calling a TP, the `invoke` function creates an instance of `tpClosure`, a set of template classes that simply holds all the arguments required by the TP to be instantiated. To each of the template classes, there is a corresponding `TPFactory` template function whose sole purpose is to instantiate the real TP for execution. Once a TP is effectively instantiated, all the codelets it needs to run are also created. Thus, by storing the function pointer to the adequate `TPFactory` as well as its corresponding `tpClosure`, one can migrate work that has not yet started anywhere on a teradevice. We create a new DF-Thread through the `df_tschedulez` call, *giving the size of the corresponding* ***`tpClosure`*** in the invoke function, so that it stores the right <`TPFactory, tpClosure`> pair, and upon firing, as many DF-Threads as there are codelets to run inside the TP are created.

4.1.4 Firing Codelets

For obvious reasons, C++ prevents the execution of a non-static member function without a corresponding instance context. Hence it is not possible to store the `Codelet::fire` member function call as a valid DF-Thread function pointer. We have elected to use an external function to perform the task. Its code is very simple, and is used for all codelets firings:

```
void Fire(void){
	Codelet* codelet = static_cast<Codelet*> df_tload();
	codelet->fire();
	df_destroy();
}
```

Thus, once a codelet is done firing, it also has a guarantee that the df-frame it was using is also deallocated.

4.1.5 Running DARTS Programs on COTSon

As we mentioned before, some compromises were necessary to be able to port DARTS on COTSon. There are new features in the latest version:

1. The loop constructs are effectively useless in the DF-Thread context: contrary to the full-software implementation of DARTS on x86, it is not possible to ensure that a given set of codelets see their dependencies reset when they are done firing, thus avoiding useless cycles of reallocations. Instead, a loop must be simulated using a recursion, much like what the current DF-Thread implementation requires already.
2. Because the TSU knows nothing of codelets and their two-level parallelism scheme, it is necessary that the user explicitly deletes the TP frames that were allocated upon exit. This only involves a few control paths, and is usually easily done in our experience.
3. Finally, the `invoke` call has been simplified compared to the original DARTS implementation: as the parent TP is no longer required to keep track of TP frames allocation, we have decided to remove the parameter altogether in the COTSon port.

These three caveats can be overcome to a certain extent if need be: the loop construct could simply generate a sequence of DF-Threads with a source and a sink, with a counter to know when to stop iterating. The required deletion of the TP frame on COTSon could be encapsulated (say, using a `EXIT_TP()`macro), thus allowing conditional compilation to generate the call if it is needed. However, it would constrain the original runtime more than it should, since in the original implementation, there is no need to know when to free the TP frame. Finally, the call to `invoke` could also be encapsulated in a macro, thus allowing conditional compilation to decide which set of parameters should be used depending on the target platform.

4.2 Evaluation of DARTS on the TERAFLUX Simulation Environment

We have performed several experiments to show the ability of the various models implementations to scale, using several benchmark kernels: Fibonacci, merge sort, and dense matrix multiplication. As we ran our experiments using the COTSon simulator [ArgolloEtAl09], we used dynamic sampling [FalconFarOrt07] to speed up simulation times.

As we explained in Section 4.1, all DARTS experiments were performed on DARTS-TSUF (cf. D7.5). The configuration we used for COTSon is the following:

- We used dynamic sampling, with a sampling size of 5 million instructions;
- we used 4, 8, and 16 cores on a single node for our experiments;
- all cores have access to a 1GB DRAM bank, with a latency of 100 cycles;
- a unified 4MB L3 cache, with a latency of 10 cycles, is shared by all cores in the node;
- all cores have access to a private 64KB unified L2 cache, with a latency of 5 cycles;
- all cores have access to a private 16KB L1 data cache, with a latency of 2 cycles;
- all cores have access to a private 32KB L1 instruction cache, with a latency of 2 cycles;
- all cache lines yield a size of 64 bytes

We compared the results of `fib`, `msort`, and `mm` using three different implementations: “pure” DF-Threads, DARTS-TSU, and OpenMP (*i.e.*, running using a normal multicore approach, without a TSU to help with the scheduling). Also, please note that the OpenMP variant was invoked with the `OMP_PROC_BIND` set to true, to ensure threads are not migrated from the core they initially got assigned to.

It should also be noted that we only report single-node results in this report, as a multi-node configuration could not be correctly exploited by a regular implementation of OpenMP. A multi-node implementation of DARTS is currently underway.

4.2.1 A Benchmark to Measure Pure Scalability: Naïve Fibonacci

Running a naïve Fibonacci computation is of course highly inefficient. However, it is a perfect tool to evaluate the scalability of a system, including the overhead required to create threads. In our experiments, when the recursion reaches $n = 18$, the computation calls a serial Fibonacci function to finish the computation.

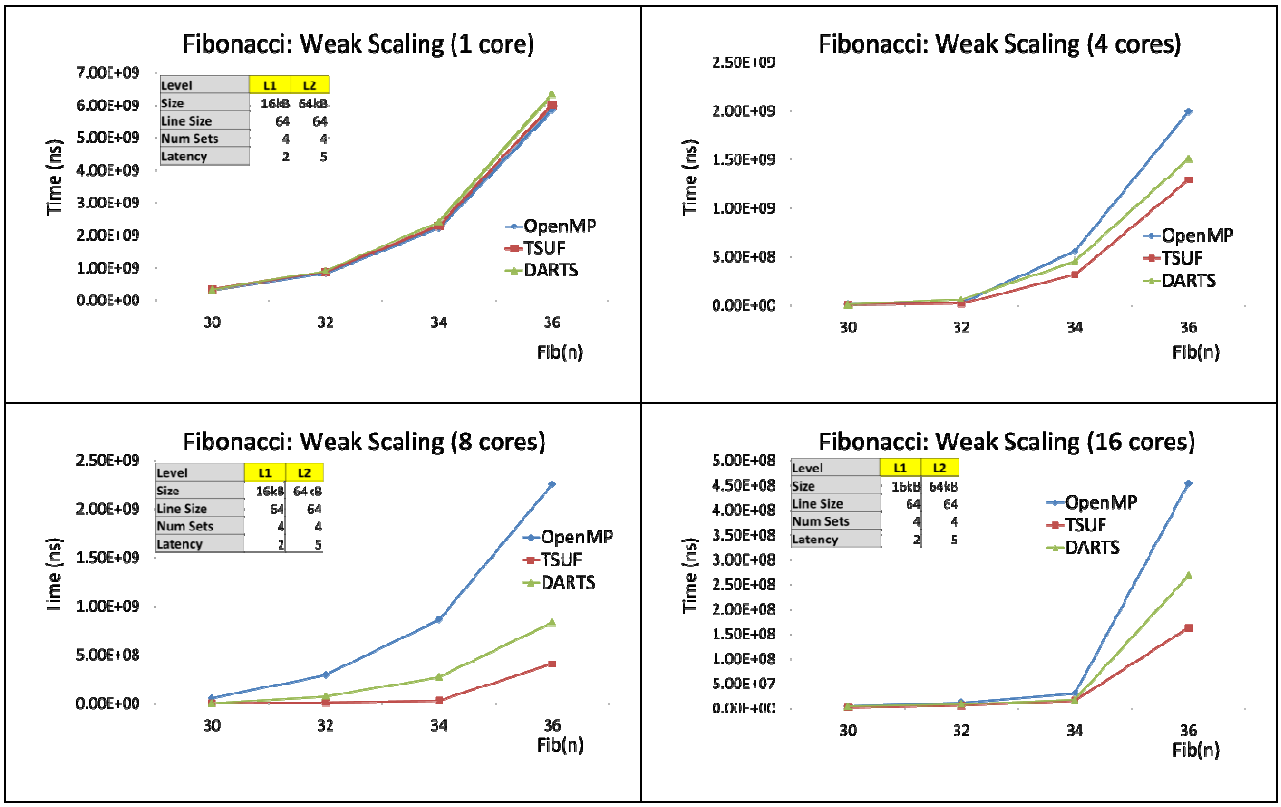


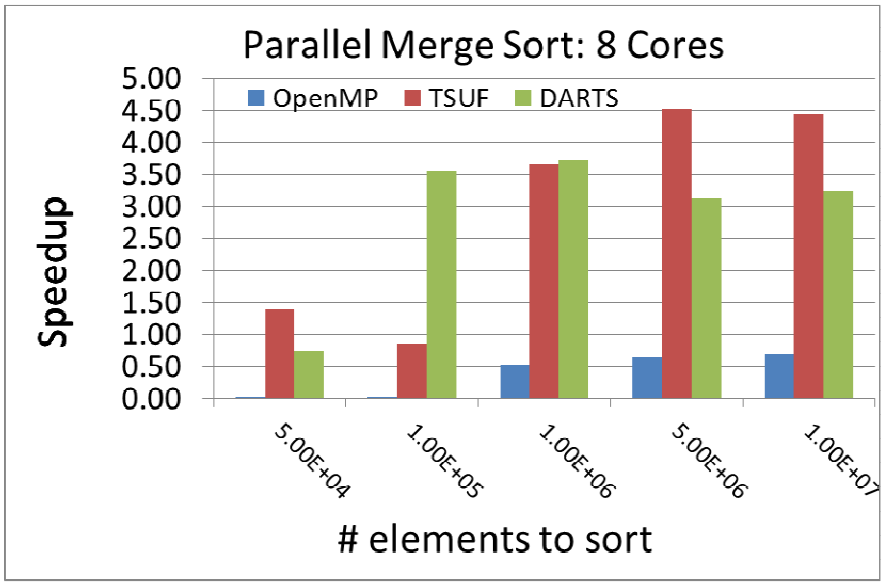
Figure 30 COTSon Experiments: Weak Scaling for Fibonacci. Threshold value: $n = 18$. Lower is better.

As Figure 30 shows, when running the Fibonacci number computation on a single core, all implementations ("pure" DF-Threads, DARTS, and OpenMP) spend roughly the same amount of time, which was expected. As we increase the number of cores however, there is a clear increase in overhead for OpenMP, while the other two stay reasonably low. The pure DF-Thread implementation outperforms DARTS-TSUF. This is due to the additional level of parallelism that DARTS implements. In a single-node environment, it basically means that we are allocating additional structures on the heap which, while still being very small (~256B on average), offer no clear benefit: Threaded Procedures were designed for a more distributed environment. However, since we map codelets to DF-Threads, overall DARTS benefits from the hardware scheduling they offer, and its performance stays close to the one demonstrated by pure DF-Threads.

4.2.2 An Intermediate Benchmark for Scalability: Parallel Merge Sort

We chose to run a parallel merge sort to have a certain balance between an almost computation and data movement free kernel (Fibonacci numbers computations) and a computationally intensive one (DGEMM, see Section 4.2.3), so underline cases where dataflow-inspired execution models may play a significant role, while not staying in the realm of "toy" kernels.

We used a parallel merge sort algorithm proposed by C. Leiserson for Cilk[1], and sketched in Introduction to Algorithms 3rd Edition [CormenEtAl09]. The implementation itself was inspired by two articles in the *Dr Dobbs* C++ Journal [Duvanenko11a, Duvanenko11b], which draws on Leiserson's lecture. Instead of using Intel Threading Building Blocks however, we ported the algorithm to use DF-Threads, DARTS-TSUF, and OpenMP. We used a cutoff value of 500 for both the divide-and-conquer step which performs the "array splitting" and the parallel merge step. When the threshold of 500 is reached by the "divide-and-conquer" part of the algorithm, a call to `qsort` (from the standard C library) is issued. When the cumulated length of both arrays used in the merge sort algorithm falls below 500 elements in the merge step, a sequential, iterative (instead of recursive) call to `serial_merge` is issued to speed-up merging the arrays. We used a pure sequential implementation of merge sort as the baseline to our experiments.



[1] Lecture slides available at http://supertech.csail.mit.edu/cilk/lecture-2.ppt

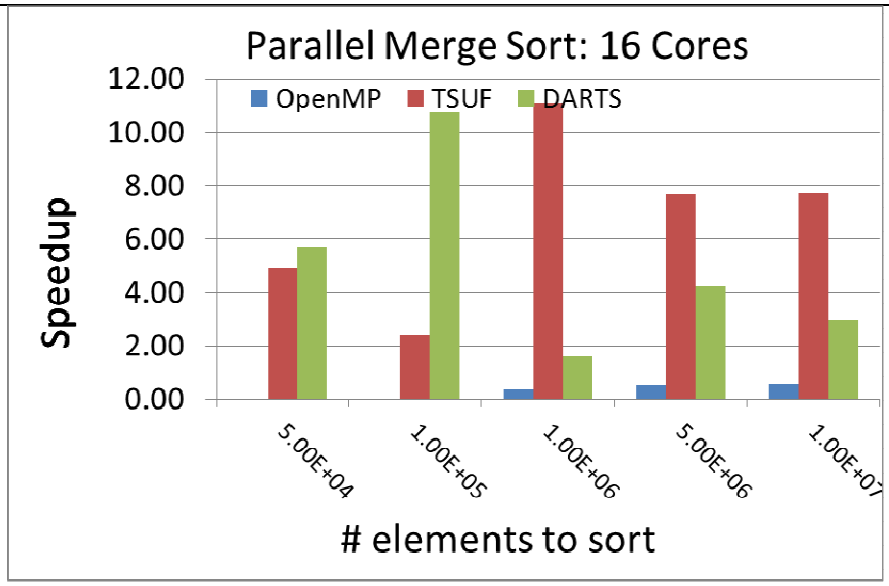
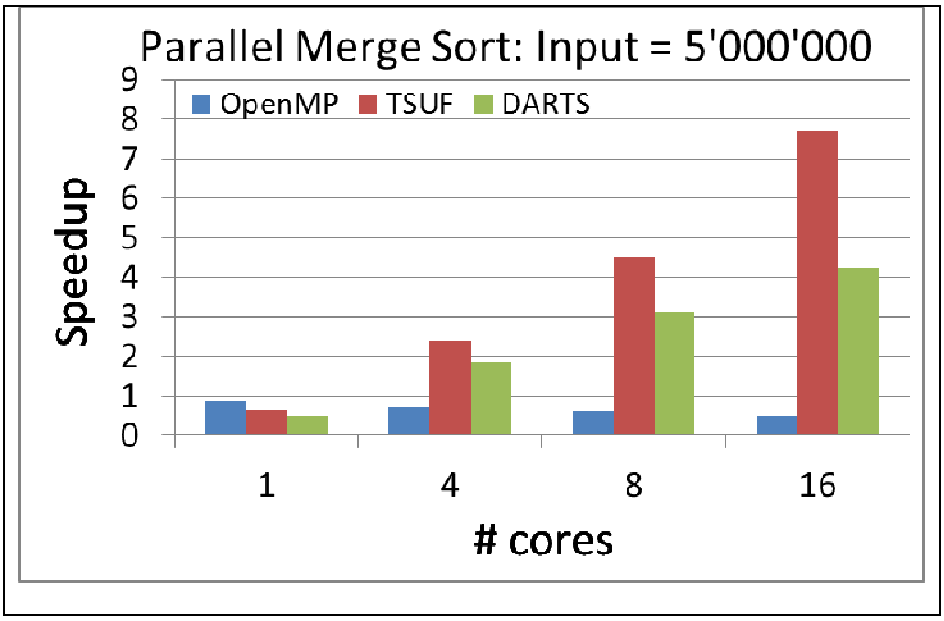


Figure 31 COTSon Experiments: Weak Scaling for Parallel Merge Sort. Higher is better.

Weak scaling is shown in Figure 31. As we increase the number of available cores on a node, full-software scheduling becomes more costly (the scheduler must decide where to assign threads, on which core). Overall, both pure DF-Threads and DARTS-TSUF outperform OpenMP as input size increases.

Figure 32 illustrates strong scaling abilities of our three systems for inputs of 5'000'000 and 10'000'000 32-bit integers to sort (resp. 40MB and 80MB worth of input, *i.e.* 10 to 20 times bigger than the shared L3 cache). Both DARTS-TSUF and pure DF-Threads clearly scale better than OpenMP. Pure DF-Threads seem to better scale at that point than DARTS-TSUF. We are still investigating the reasons. There clearly is some additional overhead involved in DARTS 2-level threading scheme, which can be alleviated when running on multiple nodes.



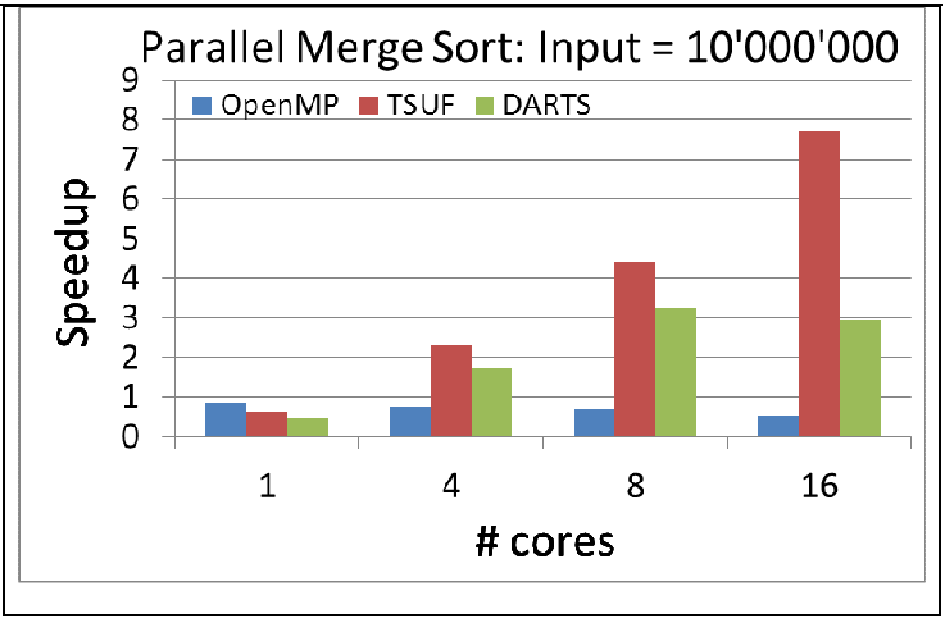


Figure 32 COTSon Experiments: Strong Scaling for Parallel Merge Sort. Higher is better.

4.2.3 A Compute-Intensive Benchmark: Matrix Multiplication

Figure 33 allows us to study the scalability of the system for computation-intensive kernels such as dense matrix multiplication. As with the Fibonacci and parallel merge sort examples, pure DF-Threads show the lowest overhead, followed by DARTS-TSUF.

Figure 33 shows the results obtained with `mm` on a 256×256 matrix multiplication. There is a relatively high discrepancy between DF-Threads/DARTS-TSUF and the OpenMP implementation, especially as we increase the number of cores.

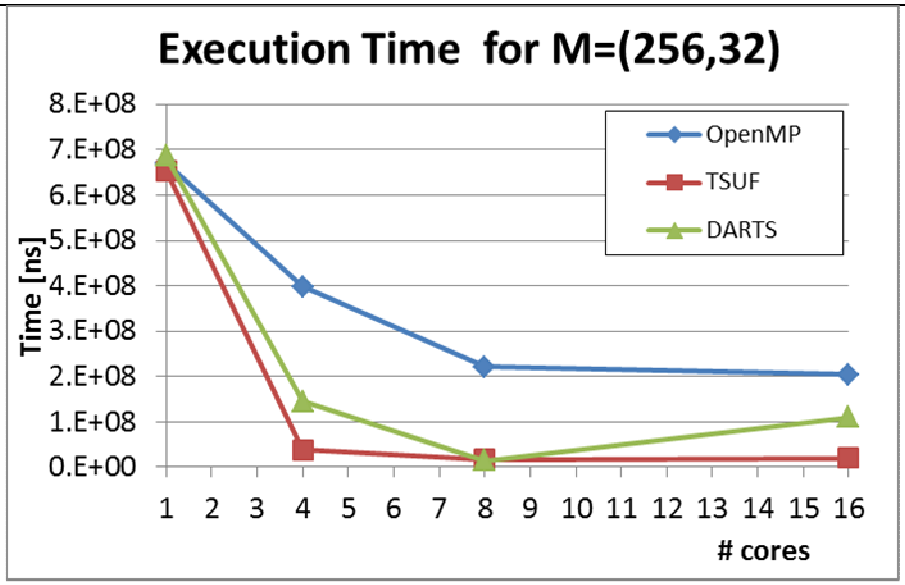


Figure 33 Performance of a 256×256 double-precision matrix multiplication, using tile sizes of 32×32. Lower is better.

5 Conclusions

This document presented our achievements with respect to power-aware scheduling leveraging dataflow-inspired, fine-grain program execution models, and the possibilities to apply them to the TERAFLUX architecture. One fundamental result is that, contrary to common wisdom, increasing performance does not automatically increase energy efficiency on tiled computations: the tile size and shape have a clear impact on the performance/energy efficiency trade-offs to make. These techniques are applicable to teradevices, at the cost of additional experiments and measurements to compute the optimal tile size in the presence of caches (instead of scratchpads), and/or the addition of a core-affinity constraint instruction to guide the TSU when creating a new DF-Thread.

It also described our implementation of the codelet model, the Delaware Adaptive Run-Time System (DARTS) to off-the-shelf multi and many core systems, and its subsequent port to the TERAFLUX simulation infrastructure, COTSon. We provide performance numbers for two widely different kernels on regular 64-bit x86: we show that DARTS is on par with state-of-the-art OpenMP implementations of parallel DGEMM (both using highly optimized Intel and AMD math libraries on their respective platforms), and greatly outperforms the Graph500 reference implementation, with almost no change to the original code. On the TERAFLUX port, we provide results where we compare our TSU-leveraged version of DARTS against the regular, software-implemented scheduling system. TSU based implementations outperform OpenMP, when the core count is higher. The DF-Threads/T* provided a solid basis for supporting flexible models such as codelets.

While this project is at an end, we plan on furthering this research and perform additional experiments to evaluate the performance gap between DF-Thread programs, our DARTS-TSU port, which leverages a merging of both DF-Thread and Codelet models, and regular OpenMP programs. A journal paper is currently being written in collaboration with the UNISI partner to explore those three models, while varying the cache hierarchy.

Appendix A – Pseudo-Code to Run Supertasks on the TERAFLUX Architecture

We provide below some pseudo-code that illustrates our proposed strategy to modify the TSU's instruction set to allow for constraining a set of DF-Threads to a given core. Please note that this proposed solution to implement supertasks means that there must be a way to signal that a given DF-Thread in a Local TSU (L-TSU) must be marked as unavailable for stealing. The best way to perform this is probably to propose two task queues, a "regular" one, from which everyone can steal, and a "supertask" queue, which should typically be bounded and very short in general (probably less than 10 slots should be available, or even less). The first phase of a supertask should be considered a regular DF-Thread, and as such should be stored in the regular DF-Thread queue. Calling the proposed **df_constrain_to_current_core** is what should move a task from one queue to the other.

```
// Stores the DF IDs of the next phases to signal
df_tid_t* global_tid_map;
typedef void (*next_phase_t)(void);
typedef struct phase2_s {
      long              supertask_id;
      next_phase_t      next_phase;
      /* The rest of the structure contains the data blocks,  *
       * local variables, etc.                                */
} phase2_t;
void phase2(void);
typedef struct phase1_s {
      long              supertask_id;
      next_phase_t      next_phase;
      /* The rest of the structure contains the data blocks,  *
       * local variables, etc.                                */
} phase1_t;
void phase1(void) {
  stask_phase1_t* frame = (stask_phase1_t*) df_tload();
  long supertask_id = get_supertask_id_self(frame);
  // Allocate new DF-Thread for the next phase
```

```
  df_tid_t next_phase = df_tschedulez(phase2,1+num_deps, sizeof(phase2_t));
  // Retrieve next phase's DF-Frame
  phase2_t* fp_phase2 = (phase2_t*) df_tcache(next_phase);
// Prefill next phase's frame with the data blocks to read and/or write
  fill_phase2(get_data_blocks(frame),supertask_id,fp_phase2);
/* Ensure that the next phase will execute on the same core. *
 * This *instruction does not exist at the moment.           */
  df_constrain_to_current_core(next_phase);
/* Store next phase's dataflow ID in a shared container accessible to    *
 * all workers involved, using the current supertask ID.                */
  store_df_tid(global_tid_map,supertask_id,next_phase);
  compute_step(get_data_blocks_from(frame));
  /* retrieve the DF IDs of other supertasks whose phases are waiting   *
   * for us to update their data blocks                                  */
  signal_neighbors( get_supertask_id(global_tid_map,supertask_id) );
  /* next_phase will be fired when all supertasks on which it depends   *
   * will have signalled it.                                             */
  df_tdecrease(next_phase);
  df_destroy();
}
```

References

[ArgolloEtAl09] Eduardo Argollo, Ayose Falcón, Paolo Faraboschi, Matteo Monchiero, and Daniel Ortega. 2009. “COTSon: infrastructure for full system simulation”. *SIGOPS Oper. Syst. Rev.* 43, 1 (January 2009), 52-61.

[ArteagaEtAl14] Jaime Arteaga, Stephane Zuckerman, Elkin Garcia, and Guang R. Gao, "Position Paper: Locality-Driven Scheduling of Tasks for Data-Dependent Multithreading". *In Proceedings of Workshop on Multi-Threaded Architectures and Applications (MTAAP 2014)*, May 2014,

[BlumofeEtAl95] Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H., and Zhou, Y. (1995) “Cilk: an efficient multithreaded runtime system” In PPOPP ’95: Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming, pages 207–216, New York, NY, USA. ACM.

[BroquedisEtAl10] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst. “hwloc: a generic framework for managing hardware affinities in HPC applications” In Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on, pages 180-186, Feb. 2010.

[ChenEtAl13] Chen Chen, Yao Wu, Joshua Suetterlein, Long Zheng, Minyi Guo, and Guang R. Gao, 2013. “Automatic Locality Exploitation in the Codelet Model”. In *Proceedings of the 2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications* (TRUSTCOM '13). IEEE Computer Society, Washington, DC, USA, 853-862.

[CormenEtAl09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009, *Introduction to Algorithms, Third Edition* (3rd ed). The MIT Press.

[Denneau11] M. Denneau, “Cyclops,” *in Encyclopedia of Parallel Computing: SpringerReference (www.springerreference.com) (D. Padua, ed.), Springer-Verlag Berlin Heidelberg*, 2011.

[Duvanenko11a] V.J. Duvanenko, “Parallel Merge Sort”, *in Dr Dobbs*, March 24, 2011. URL: http://www.drdobbs.com/parallel/parallel-merge-sort/229400239.

[Duvanenko11b] V.J. Duvanenko, “Parallel Merge”, *in Dr Dobbs*, February 24, 2011. URL: http://www.drdobbs.com/parallel/parallel-merge-sort/229204454

[FalconFarOrt07] A. Falcon, P. Faraboschi, and D. Ortega “Combining Simulation and Virtualization through Dynamic Sampling”, *Performance Analysis of Systems & Software, 2007. ISPASS 2007. IEEE International Symposium on*, vol., no., pp.72, 83, 25-27 April 2007
URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4211024&isnumber=4211007

[Ga0HumWon90] G. Gao, H. Hum, and Y.-B. Wong, “Parallel function invocation in a dynamic argument-fetching dataflow architecture” In Databases, Parallel Architectures and Their Applications, PARBASE-90, International Conference on, pages 112-116, Mar 1990.

[GarciaEtAl10] Elkin Garcia, Ioannis E. Venetis, Rishi Khan and Guang R. Gao, "Optimized Dense Matrix Multiplication on a Many-Core Architecture", *In Proceedings of International European Conference on Parallel and Distributed Computing (Euro-Par'10)*, Ischia, Italy. August 31- September 3, 2010.

[GarciaOroGao11] Elkin Garcia, Daniel Orozco and Guang R. Gao, "Energy efficient tiling on a Many-Core Architecture", *In Proceedings of 4th Workshop on Programmability Issues for Heterogeneous Multicores*

(MULTIPROG 2011); 6th International Conference on High-Performance and Embedded Architectures and Compilers (HiPEAC), Heraklion, Greece. January 23, 2011.

[GarciaEtAl13] E. Garcia, D. Orozco, R. Khan, I. Venetis, K. Livingston, and G.R. Gao, "A Dynamic Schema to increase performance in Many-core Architectures through Percolation operations", *In Proceedings of the 2013 IEEE International Conference on High Performance Computing (HiPC 2013)*, Hyderabad, India, December 18 - 21, 2013.

[GarciaGao13] E. Garcia and G. R. Gao, "Strategies for improving Performance and Energy Efficiency on a Many-core", *In Proceedings of 2013 ACM International Conference on Computer Frontiers (CF 2013)*, May 14-16, Ischia, Italy, ACM, 2013.

[Giorgi12] R. Giorgi, "TERAFLUX: Exploiting Dataflow Parallelism in Teradevices", ACM Computing Frontiers, Cagliari, Italy, May 2012, pp. 303-304

[GiorgiEtAl14] 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

[LouderdaleKhan12] C. Lauderdale and R. Khan, "Towards a codelet-based runtime for exascale computing: position paper". In Proceedings of the 2nd International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era (EXADAPT '12). ACM, New York, NY, USA, p. 21-26

[MurphyEtAl10] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. Ang. Introducing the graph 500. Cray Users Group (CUG), 2010.

[SolinasEtAl13] M. Solinas, R.M. Badia, F. Bodin, A. Cohen, P. Evripidou, P. Faraboschi, B. Fechner, G. R. Gao, A. Garbade, S. Girbal, D. Goodman, B. Khan, S. Koliai, F. Li, M. Lujan, L. Morin, A. Mendelson, N. Navarro, A. Pop, P. Trancoso, T. Ungerer, M. Valero, S. Weis, I. Watson, S. Zuckerman, and R. Giorgi, "The TERAFLUX Project: Exploiting the DataFlow Paradigm in Next Generation Teradevices," *Digital System Design (DSD), 2013 Euromicro Conference on* , vol., no., pp.272,279, 4-6 Sept. 2013.

[SuetterleinZucGao13] Joshua Suetterlein, Stéphane Zuckerman, and Guang R. Gao. 2013. "An implementation of the codelet model". In *Proceedings of the 19th international conference on Parallel Processing* (Euro-Par'13), Felix Wolf, Bernd Mohr, and Dieter Mey (Eds.). Springer-Verlag, Berlin, Heidelberg, 633-644.

[TreibigHagWel10] J. Treibig, G. Hager, and G. Wellein, "LIKWID: A Lightweight Performance-Oriented Tool Suite for x86 Multicore Environments," *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on*, vol., no., pp.207, 216, 13-16 Sept. 2010.
URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5599200&isnumber=5599082

[ZuckermanPerJal08] S. Zuckerman, M. Pérache, and W. Jalby. "Fine Tuning Matrix Multiplications on Multicore", *In Proceedings of the High-Performance Computing (HiPC) International Conference, pages 30–41*, Bangalore, 2008.

[ZuckermanEtAl11] Stéphane Zuckerman, Joshua Suetterlein, Rob Knauerhase, and Guang R. Gao, 2011. "Using a "codelet" program execution model for exascale machines: position paper". In *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era* (EXADAPT '11). ACM, New York, NY, USA, 64-69.