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

PROJECT NUMBER: 249013

Exploiting dataflow parallelism in Teradevice Computing

D2.1 –Report on the reference set of applications chosen, and initial characterization of the applications

Due date of deliverable: 31st December 2010
Actual Submission: 31st December 2010

Start date of the project: January 1st, 2010 Duration: 48 months

Lead contractor for the deliverable: Barcelona Supercomputing Center (BSC)

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
0.1	**Rosa M. Badia**	**BSC**	**Table of content**
0.2	**Rosa M. Badia**	**BSC**	**Editions in sections 2 and 3**
0.3	**Yoav Etsion**	**BSC**	**Editions in sections 3 and 4**
0.4	**Sylvain Girbal**	**Thales**	**Input for sections 2, 3 and 4**
0.5	**Antoni Portero**	**UNISI**	**Input for section 3**
0.6	**Mikel Lujan**	**UNIMAN**	**Input for section 3 & section 4**
1.0	**Rosa M. Badia**	**BSC**	**Integration of internal review and final edition**

Release Approval

Name	Role	Date
Rosa M. Badia	**Originator**	
Nacho Navarro	**WP Leader**	
Roberto Giorgi	**Project Coordinator for formal deliverable**	**31.12.2010**

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

The following list of authors will be updated to reflect the list of contributors to the writing of the document.

Rosa M. Badia, Yoav Etsion, Jesus Labarta, Nacho Navarro, Milan Pavlovic, Harald Servat, Carlos Villavieja
BSC

Sylvain Girbal
Thales

Mohammad Ansari, Mikel Lujan, Chris Kirkham, Ian Watson
UNIMAN

Roberto Giorgi, Antoni Portero
UNISI

Albert Cohen
INRIA

© 2009 TERAFLUX Consortium, All Rights Reserved.
Document marked as PU (Public) is published in Italy, for the TERAFLUX Consortium, on the www.teraflux.eu web site and can be distributed to the Public.
The list of author does not imply any claim of ownership on the Intellectual Properties described in this document.
The authors and the publishers make no expressed or implied warranty of any kind and assume no responsibilities for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information contained in this document.
This document is furnished under the terms of the TERAFLUX License Agreement (the "License") and may only be used or copied in accordance with the terms of the License. The information in this document is a work in progress, jointly developed by the members of TERAFLUX Consortium ("TERAFLUX") and is provided for informational use only.
The technology disclosed herein may be protected by one or more patents, copyrights, trademarks and/or trade secrets owned by or licensed to TERAFLUX Partners. The partners reserve all rights with respect to such technology and related materials. Any use of the protected technology and related material beyond the terms of the License without the prior written consent of TERAFLUX is prohibited. This document contains material that is confidential to TERAFLUX and its members and licensors. Until publication, the user should assume that all materials contained and/or referenced in this document are confidential and proprietary unless otherwise indicated or apparent from the nature of such materials (for example, references to publicly available forms or documents).
Disclosure or use of this document or any material contained herein, other than as expressly
permitted, is prohibited without the prior written consent of TERAFLUX or such other party that may grant permission to use its proprietary material. The trademarks, logos, and service marks displayed in this document are the registered and unregistered trademarks of TERAFLUX, its members and its licensors. The copyright and trademarks owned by TERAFLUX, whether registered or unregistered, may not be used in connection with any product or service that is not owned, approved or distributed by TERAFLUX, and may not be used in any manner that is likely to cause customer confusion or that disparages TERAFLUX. Nothing contained in this document should be construed as granting by implication, estoppel, or otherwise, any license or right to use any copyright without the express written consent of TERAFLUX, its licensors or a third party owner of any such trademark.
Printed in Siena, Italy, Europe.
Part number: *please refer to the File name in the document footer.*

DISCLAIMER
EXCEPT AS OTHERWISE EXPRESSLY PROVIDED, THE TERAFLUX SPECIFICATION IS PROVIDED BY TERAFLUX TO MEMBERS "AS IS" WITHOUT WARRANTY OF ANY KIND, EXPRESS, IMPLIED OR STATUTORY, INCLUDING BUT NOT LIMITED TO ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF THIRD PARTY RIGHTS.
TERAFLUX SHALL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL OR CONSEQUENTIAL DAMAGES OF ANY KIND OR NATURE WHATSOEVER (INCLUDING, WITHOUT LIMITATION, ANY DAMAGES ARISING FROM LOSS OF USE OR LOST BUSINESS, REVENUE, PROFITS, DATA OR GOODWILL) ARISING IN CONNECTION WITH ANY INFRINGEMENT CLAIMS BY THIRD PARTIES OR THE

SPECIFICATION, WHETHER IN AN ACTION IN CONTRACT, TORT, STRICT LIABILITY, NEGLIGENCE, OR ANY OTHER THEORY, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Glossary

BW	Bandwidth
CPI	Cycles per Instruction
DDM	Data-Driven Multithreading
DTA	Decoupled Threaded Architecture
HPC	High Performance Computing
HPEC	High Performance Embedded Computing
L	Latency
StarSs	Star Superscalar

Executive Summary

This document is the first deliverable of WP2, Benchmarks and Applications. The objective of this workpackage is to understand the runtime behaviour of applications in order to establish a guideline in the design of the other components of the computing system in TERAFLUX. As TERAFLUX explores the design of highly parallel teradevice systems, a key step in the project is understanding the fundamental requirements of highly parallel applications and their implications on all layers of a computing system that supports a data-flow programming and execution model – from the programming model itself, down to extensions to commodity architecture.

The deliverable describes the results of the first year of the project in tasks 2.1 and task 2.2. While task 2.1 is finished, task 2.2 will continue during the second year of the project. The activities performed in task 2.1 relate to the selection of the applications and benchmarks to be used in the project. The criteria used to select the applications include aspects implicit to the application (domain, inherent parallelism, data-flow characteristics, transactional memory requirements) and others related to practical aspects (availability of the code, availability of realistic input data, previous experience of partners with the code...). The codes have been classified in two lists: kernels/benchmarks and full applications.

In order to make the kernels, benchmarks and applications available to all partners in a uniform distribution, a disk image is based on a Linux installation where all the necessary libraries and environments have been installed (i.e. MPI, StarSs runtime). The disk contains both the compiled codes with all the necessary data to be able to run them, and also the source code, scripts and Makefile to be able to rebuild them. The format was chosen such that it would natively plug into the COTSon simulation platform, and the partners can therefore configure the simulation platform to simply boot Linux from the disk image and run the applications.

The activities performed in task 2.2 relate to the characterization of the applications. The parameters that will be used to characterise the applications have been classified into resource requirements (memory bandwidth, network latency and bandwidth, parallelism,...) and TERAFLUX specific requirements (transactional memory and data flow). A description of the methodologies to be used is given in section 3. While some of them are based on analytical methods, a large bunch of them are based on tracefile generation of real runs and different types of post-processing and automatic analysis methods. Section 4 presents a description of initial results for some of the project applications.

1 Introduction

This is the first deliverable of WP2, Benchmarks and Applications. Understanding the runtime behaviour of applications is a crucial guideline in the design of computing systems, as they are the effective consumers of the underlying compute power. As TERAFLUX explores the design of highly parallel teradevice systems, a key step in the project is understanding the fundamental requirements of highly parallel applications and their implications on all layers of a computing system that supports a data-flow programming and execution model – from the programming model themselves, down to extensions to commodity architecture. This exploration includes:

- Identify applications that can serve as reference applications for a programming model based on data-flow principles, and that can efficiently scale to utilize teradevice system.
- Characterize the resource requirements of these highly parallel applications, in terms of memory usage, bandwidth, and latency. Identify the performance requirements from underlying interconnection network. These characteristics will assist the architectural exploration performed in WP6.
- Uncover common data-flow and data-locality patterns implicit to the reference applications that can be disseminated into the programming model (WP3) as either data-flow or transactional semantics.
- Port a few applications to the programming models chosen by WP3. Initially, we will consider the StarSs (BSC) and DDM (UCY) programming model, and the DTA (UNISI) execution models. The ported applications will be used by the other work packages to guide their proposed designs.
- Extract the interesting patterns and data accesses and that will assist other work packages build sensible benchmarks that can test the proposed constructs in all domains: programming model (WP3), compilation platform (WP4), reliability (WP5) and architecture (WP6).

1.1 Document structure

The deliverable is organized as follows: this section introduces the deliverable and its structure, section 2 describes the criteria that have been used to select the applications and benchmarks and lists the actual selected ones (task 2.1). Section 3 presents the parameters that will be the object of the applications' characterization and explains the methodologies that will be used in this characterization. Section 4 presents some initial results of the characterization and finally section 5 concludes the document.

1.2 Relation to other deliverables

No specific one.

1.3 Activities referred by this deliverable

This deliverable refers to the activities performed in tasks 2.1 and 2.2 during the first year of the project. Task 2.2 will continue its activities during the second year of the project.

2 Identification of reference applications

The goal of task 2.1 is to identify a number of reference applications that can potentially utilize a tera-device. As this scale of parallelism is currently only available in the High-Performance Computing (HPC) and High-Performance Embedded Computing (HPEC) domains, and according to what was defined in the description of work, we have initially focused our search on HPC and HPEC applications that are expected to scale with technology. Such applications include physical, biological and physiological simulations, and sensor data fusion applications implying sonar/radar and video.

However, although initially it was not explicitly indicated in the description of work, only working on real (complete) applications can exclude more realistic research in the framework of the different workpackages: for example, to be able to effectively define the programming models features and to be able to compare different solutions, small benchmarks that expose specific behaviours are required.

With this objective, the workpackage has worked to identify whole reference applications and small kernels and benchmarks.

2.1 Criteria

To select the project applications, different parameters have been taken into account. Among them:

- Application domain: the task has considered applications from different domains in order to consider the different particularities of each type of problem
- Inherent parallelism: the use of task-based data-flow programming model with transactional semantics is expected to enable applications to expose their parallelism but it is necessary that enough inherent concurrency is present in the algorithm in order to scale to a large number of cores.
- Realistic input data: In order to convince the scientific and industrial community about the features of the TERAFLUX architecture and software layers real applications with real input data should be used.
- Existence of open source code: This requirement twofold, to enable the availability of the codes to the project partners and also to enable the project to publish modified versions of the code that would be made available to the community.
- Existence of research collaborations between the project partners and the code developers: this is not a requirement but it is a positive aspect, since it will help in the impact of the project and also in the progress of the research done with the applications.
- Existence of previous versions and experience by the project partners: the project does not have enough effort and time to start from scratch new applications. This can be only considered for small kernels/benchmarks.
- Exposure of data-flow characteristics: to enable a natural porting to data-flow programming model / architecture
- Exposure of transactional memory requirements: similarly, to enable to show the benefits of transactional behaviour.

2.2 Kernels / Benchmarks

The following table lists the kernels and benchmarks identified so far and that will be used in the project. For each of them we list the name, the partner that is responsible for the code, the source or programming model of the initial code, if the code initially presents need for transactional memory and which will be the performance reference used.

Benchmark	**Contributing partner**	**Initial code**	**Need for TM**	**Performance reference**
Matmul	BSC	StarSs		MKL
Radix Sort	UNISI	SPLASH-2		
Lonestar Delaunay	BSC	Galois		Galois
Lonestar Barnes-Hut	BSC	Galois		Galois
Cholesky	BSC	StarSs		MKL
Sparse LU	BSC	StarSs		MKL
FFT2D	BSC	StarSs		FFTW
SPECFEM3D	BSC	StarSs		Sequential
N Queens	BSC	StarSs		Sequential
Lee's Routing algorithm	UNIMAN	TM		TM
Vacation	UNIMAN	TM		TM
Bayes	UNIMAN	TM		TM
Genome	UNIMAN	TM		TM
Intruder	UNIMAN	TM		TM
Kmeans	UNIMAN	TM		TM
Laberynth	UNIMAN		See Lee-TM	
Ssca2	UNIMAN	TM		TM
Yada	UNIMAN	TM		

Table 1. List of kernels and benchmarks

2.3 Full Applications

The following applications have been initially selected due to their representativeness in the HPC world, their scalability in the current supercomputing machines, the availability of code and in some cases for the collaborations existent between the partners and the authors of the codes. Some of the codes have been initially made available with the current parallelization in MPI.

Benchmark	**Contributing partner**	**Initial code**	**Performance reference**
GADGET	BSC	MPI	Original code
GROMACS	BSC	MPI	Original code
PEPC	BSC	MPI	Original code
SWEEP3D	BSC	MPI	Original code
WRF	BSC	MPI	Original code
STAP (Radar)	THALES	C	Sequential
Viola & Jones (Pedestrian detection)	THALES	C	Sequential

Table 2 List of full applications

2.3.1 HiPEAC repository

The TERAFLUX partners are involved in the HiPEAC Network of Excellence (http://www.hipeac.net). Most of them are also active participants at the Task Force on Applications inside HiPEAC. As other projects that are largely driven by HiPEAC partners (http://www.hipeac.net/related), we share the goal of selecting a set of well-characterized applications and input data sets. This is a practical way of source code sharing and collaboration within the European Projects community.

The HiPEAC Task Force had previously proposed some of the applications selected in this WP as representatives of their respective domain. In TERAFLUX, we will explore their scalability by using our dataflow programming and execution model.

In the opposite direction, some applications have been contributed directly by our partners; after we have characterized them and agreed on their appropriateness, we will contribute them to the HiPEAC repository. As mentioned before, we would like other projects to use them as a reference, and therefore be able to compare their performance when ported and run on top of other platforms.

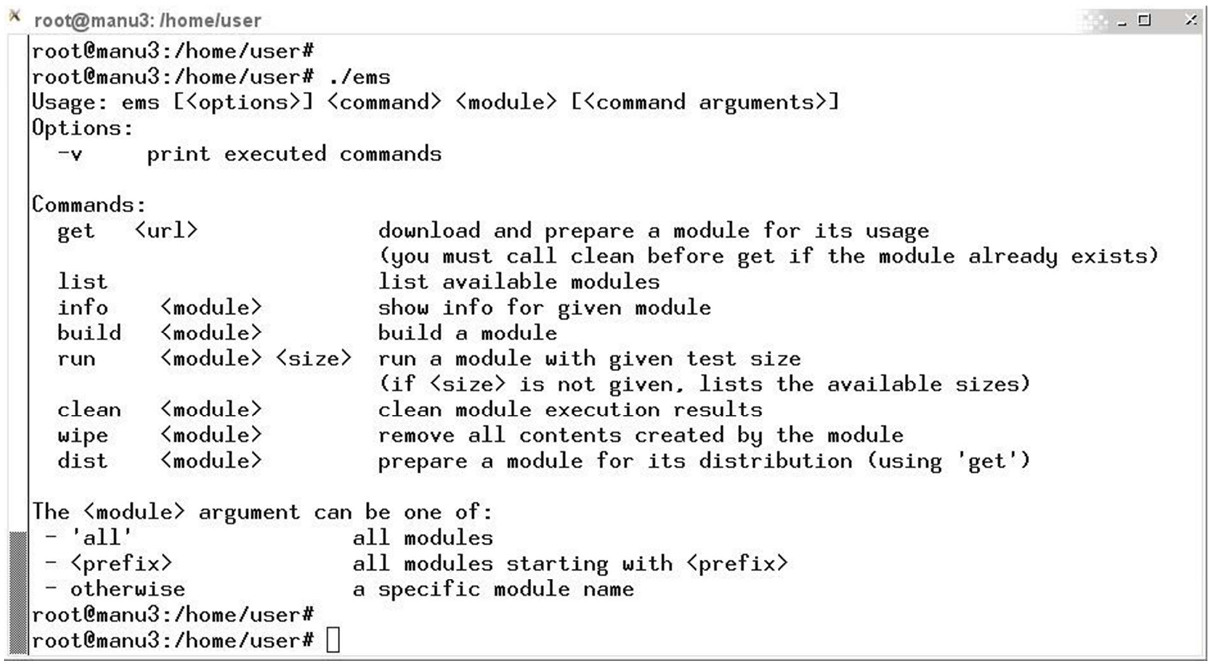
2.3.2 Initial Reference Kernel/Application Disk Image

Given the variety of kernels and applications explored in the project, which span multiple programming models, it is important to provide all partners with a baseline installation that can be used in conjunction with the simulation platform. Such a distribution was therefore prepared at BSC and distributed to the partners.

In order to avoid portability and compatibility issues, the baseline distribution is packed as a raw disk image containing a basic Linux installation. The format was chosen such that it would natively plug into the COTSon simulation platform, and the partners can therefore configure the simulation platform to simply boot Linux from the disk image and run the applications.

On top of the Linux installation, the disk image contains an initial set of benchmark kernels and applications, both pre-compiled and in binary form, as well as all the tools needed to modify and recompile the source files. Furthermore, the image includes an extensible *Experiment Management System* (EMS) that manages the execution and compilation of both the benchmarks and any external tool required, such as the compilation tool-chains for the different programming models. EMS is designed with extensibility in mind, so that more benchmarks and tools can be easily added to the disk image, either by re-distributing a new version of the image itself, or by distributing an archive containing only the additions.

The EMS is composed of a set of meta-scripts that abstracts the idiosyncrasies of the individual benchmarks by providing a uniform interface to build and execute benchmarks. The usage of the EMS script is outlined below.


root@manu3: /home/user

```
root@manu3:/home/user#
root@manu3:/home/user# ./ems
Usage: ems [<options>] <command> <module> [<command arguments>]
Options:
  -v     print executed commands

Commands:
  get   <url>             download and prepare a module for its usage
                          (you must call clean before get if the module already exists)
  list                    list available modules
  info    <module>        show info for given module
  build   <module>        build a module
  run     <module> <size>  run a module with given test size
                          (if <size> is not given, lists the available sizes)
  clean   <module>        clean module execution results
  wipe    <module>        remove all contents created by the module
  dist    <module>        prepare a module for its distribution (using 'get')

The <module> argument can be one of:
 - 'all'                all modules
 - <prefix>             all modules starting with <prefix>
 - otherwise            a specific module name
root@manu3:/home/user#
root@manu3:/home/user# 
```

Figure 1 EMS usage information

EMS abstracts the datasets for each benchmark into predefined names - *small*, *medium*, and *large*. These abstract names are translated by the main EMS meta-script into benchmark-specific runtime arguments. This translation is naïve by design and is implemented by executing sub-scripts named after the abstract dataset size. This enables users of the disk image to manually change dataset arguments and thereby utilize EMS for any additional benchmarks characterization performed. Moreover, upon benchmark execution, EMS creates a new run directory that stores the execution output thus providing a foundation for methodical comparative studies of the benchmarks.

Finally, EMS uses a hierarchical directory structure to store the benchmarks and external tools, including their binaries, source codes, datasets, execution scripts, and build scripts. This design, in turn, allows users to track script execution in order to simplify its extensibility by providing users with a glimpse into the internal workings of the EMS utility.

3 Methodologies used in the characterization and modeling of project kernels/benchmarks/applications

3.1 Parameters to be measured

Application requirements dictate the platform specification. But little is known about the resource requirements of current HPC applications, and how well they can utilize teradevice systems. This is even more critical in the evolving field of High Performance Embedded Computing (HPEC), where the application and platform are tailored together.

We will study and model the demands from the interconnect network, the memory bandwidth, and memory locality patterns. For example, increasing the number of computing cores is already pressuring the memory system, thus limiting available bandwidth and increasing access latencies. A better understanding of the actual use of data in applications will show the variability in memory requirements - in terms of size, bandwidth, and susceptibility to latency - of different computations inside the application. Additionally, we will focus on the inherent computational tasks inside HPC and HPEC applications, thus gaining more concise and effective insights of the transactional and data-flow semantics. This information will serve as a guideline in the design of asymmetric distributed memory systems, as well as in the mapping of computations to computing elements using data/computation scheduling algorithms.

Of particular interest will be to understand the characteristics of applications using the programming model that combines dataflow and transactions.

Another way to characterize applications is the usage of architecture independent metrics [Amesfoort10, Strohmaier04]. We briefly explain the methodology below. Some tools like *Intel Parallel Studio XE* provides ways to measure “parallelism” as defined by Leiserson [Frigo98]. This is discussed in subsection 3.3

According to this, we have classified the parameters to be evaluated from the applications in *Resource requirements* and *TERAFLUX execution model requirements*. For these two categories, the following parameters have been identified:

Resource requirements

- Memory BW
- CPI Breakdown
- Interconnection: BW, L
- Memory hierarchy
- Memory locality patterns
- Parallelism

TERAFLUX execution model requirements

- Transactional needs
    - Proportion of time spent executing transactions

    - Characteristics of the executed transactions
    - Characteristics of the wasted work due to transaction aborts
- Data flow needs

We have also classified the type of evaluation as Analytic evaluation (those performed statically, in an analytical way) and Evaluation using performance/monitoring tools (those based on actual results of applications on real or simulated platforms).

Metrics to characterize Parallel Applications

We want a way to effectively address the problem of characterizing each parallel applications limiting its scale by splitting the potentially infinite application space into a limited set of application classes. This idea is not entirely new, as a similarity-based taxonomy has been recently proposed. We focus on a mostly quantitative application characterization based on [Amesfoort10, Stroghmaier04]. They introduce a set of metrics to characterize applications and we show how they can be evaluated for three case-studies. Once a wide range of applications have been characterized, we can identify a limited number of application classes. Next, for these classes, dedicated programming tools can be used to write high-performance parallel software in a productive way. In the following paragraph, we enumerate the set of metrics:

Metrics related to Computation [Amesfoort10]**:** (Available concurrency (*Conc*), Arithmetic Intensity (*AI*), Operation Mix (*OpMix*), Memory Footprint (*MemFt*).

Metrics related to Communication: (Count, Direction (*R/W*), Length (*Len*), Alignment (*Align*))

Metrics related to Synchronization: (Local synchronization (LSyncs), Global Synchronization (*GSyncs*), Update Conflicts (*UpdCf*))

If we assume that the data access of any code [Stroghmaier04] can be described as several concurrent streams of addresses which in turn can be characterized by a single, unique set of performance related factors. As performance factors for our characterization we chose:

Main factors of performance: (Regularity, Data Set Size, Spatial locality, Temporal Locality and reuse).

At the current moment, we do not have specific measurement but we consider these kind of characterizations very relevant to us.

The next sections present the methodologies that will be used in the project to perform the application characterization. We have classified them according to their nature, differentiating analytic methods from methods that use performance/monitoring tools.

3.2 Analytic evaluation

3.2.1 Dataflow analysis

For the purpose of analysing the STAP and Pedestrian Detection applications, Thales used the internal SpearDE co-design environment, allowing the implementation of signal and image processing dataflow applications onto heterogeneous distributed architectures.

We mostly used the first stages of the whole SpearDE workflow:

Application Analysis: The first stage consists of describing the dataflow application as an acyclic graph within the graphical environment of PtolemyII. The nodes of this acyclic graph represent statically affine nests of loops to be executed over the elementary operations to be applied on the dataflow, whereas the edges represent the input-output flow.

This representation uses a multi-dimensional Synchronous Data Flow-like model of computation based on the ArrayOL formalism, allowing to highlight several axes of parallelism beforehand. Task parallelism becomes obvious in the graphical representation and data parallelism is highlighted inside each node through the external loops to be executed.

At this point, despite the fact that the application design is architecture agnostic, thanks to the information describing an elementary operation, data sizes are computed by the tool on the fly allowing to compute a preliminary estimation of throughput and computational load.

This information will be later used in the SpearDE environment to prepare the user-assisted architectural mapping phase, to automatically generate the communications allowing to transfer data between processing elements, or to automatically fuse several elementary tasks together to reduce the communication overhead.

In order to test the validity of the application graph, sequential, executable C code can be generated and its results are compared to those from the reference implementation.

From this Application Analysis phase, the TERAFLUX project will take benefits from the data-flow description of the application, the associated estimated throughput, and from the sequential version of the application to validate against.

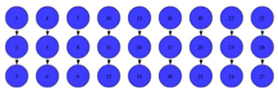
Communications and the Transposition problem: For signal processing applications, the dataflow usually consists of multidimensional data (e.g. radar), whereas the elementary tasks consist of various filters, dot products, averaging, interleaving and de-interleaving to be applied to several dimensions of this multi-dimensional data.

However those elementary operations make no assumptions on the data organization applying themselves to the first dimensions of the input data. Some extra data transpositions (or corner turns) are therefore necessary prior to applying the elementary operations. To avoid extra communication costs, those transpositions are performed alongside the communications between processing elements, trying both to maximize data-locality and to minimize memory occupancy.

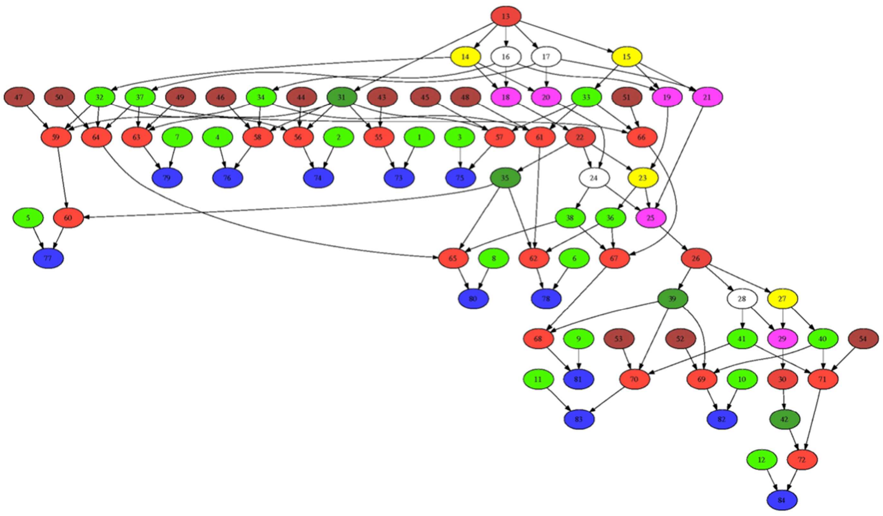
Within the TERAFLUX project, we will first consider those transpositions as extra elementary operations, allowing to directly use the TERAFLUX communication model at the cost of extra communications. In a second step we will consider embedding those transpositions into the communication scheme to reduce the communications and benefit from better data locality as presented above.

3.2.2 Task graph generation

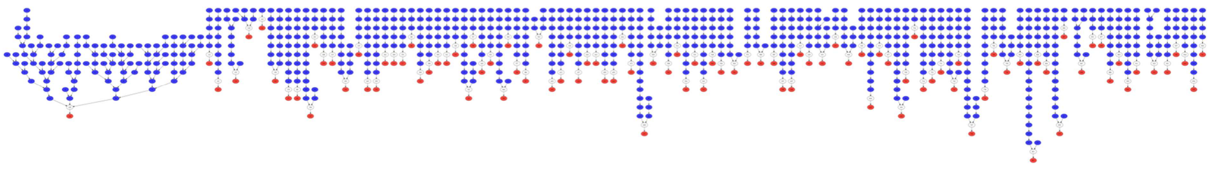
The StarSs programming model is based on the dynamic generation of a task graph that is used by the runtime to exploit the concurrency of the computation. The task graph is built taking into account the actual data dependences of the tasks. Although built at execution time, the StarSs runtime is able to generate a graphical version of this graph post-mortem in such a way that the task graph can be used to characterize the application. The basic characteristic that can be observed from the task graph is the potential concurrency, although in heterogeneous environments where given tasks may require specific hardware the graph may also reflect this information. Although currently not available, further information that could be extracted from the graph is the amount of data that flows through the graph and how it is transformed by each task. The next pictures show views of different task graphs that show the variability of their morphology:



(a)



(b)



(c)

Examples of StarSs Task Dependence Graphs: a) Matrix multiply; b) Check LU (composition of LU and check of results); c) PBPI

3.3 Evaluation using performance/monitoring tools

3.3.1 Memory BW

Consolidating multiple cores on a single chip imposes much higher bandwidth requirements on the shared components of the memory system — namely the off-chip memory bandwidth and the shared caches. Off-chip memory bandwidth is limited by the number of chip-to-board pins on one hand, and by the signalling frequencies of each pin on the other. On-chip caches are therefore commonly used to reduce the number of off-chip accesses, thereby reducing off-chip bandwidth (and access latencies), but are limited in size.

These design issues motivate the exploration of memory requirements of real parallel applications.

As we are not assuming parallel applications that explicitly target shared-memory CMPs, we base our predictions on the per-CPU memory requirements of distributed memory MPI applications. Although this methodology is imperfect (data may be replicated between nodes, which may result in pessimistic predictions when addressing shared-memory environments), we believe it provides a good indication of the requirements from a CMP memory system.

Methodology

We perform a full execution of the application, instrumented at the higher abstraction level: CPU bursts, synchronization and communication events. It produces a full time-stamped trace of events, annotated with hardware performance counters and memory usage statistics associated with each CPU burst. The full trace, representing hours of real execution, is still too large for processing. In order to reduce it to a manageable size, we apply non-linear filtering and spectral analysis techniques to determine the internal structure of the trace and detect periodicity of applications. Based on this analysis, we can cut a sample of the original trace, between 10 and 150 times smaller than the full trace [Casas07]. Next, we use a density-based clustering algorithm applied to the hardware counters to determine the most representative computation CPU bursts inside a period of the new trace [Gonzalez09]. At that point, we can analyse in isolation each one of the detected CPU phases (cluster of CPU bursts).

Our evaluation platform is a cluster of JS21 blades (nodes), each hosting 4 IBM Power PC 970MP processors running at 2.3 GHz. Each node has 8 GB of RAM, shared among its 4 processors, and is connected to a high-speed Myrinet type M3S-PCIXD-2-I port, as well as two Gigabit Ethernet ports. In order to avoid contention on the nodes' RAM, the benchmarks executed using only a single processor per node. Therefore, an application running on 64 processors actually had exclusive access to 64 nodes and 256 processors, of which 192 were idle (3 per node) such that each processor used by the application had 8 GB of memory and the full bandwidth at its disposal.

3.3.2 CPI Breakdown

A common metric used to evaluate processor performance is the average number of cycles required to complete the processing of a single instruction, or *cycles-per-instruction* (CPI). This metric, acquired by averaging the number of instructions processed over a period (measured in cycles), accounts for the different pipeline stalls that may occur during processing and provides a simple computational throughput metric.

Using hardware performance counters the basic CPI metric can be broken down to its specific components – the average delay induced by each of the functional units and different types of stalls. These include memory stalls, waiting for specific functional units, etc. Breaking down the basic metric into its individual components enables a better understanding the performance bottlenecks, and how the programming model and processor design affect performance.

Methodology

The results presented in next section were obtained by running selected MPI applications on the MareNostrum supercomputer, and sampling the different hardware performance counters throughout the run, thereby providing continuous performance measurement that facilitates performance analysis of individual computational phases.

The design of hardware performance counters does not support a continuous tracking all the counters on each processor. We have therefore used a temporal sampling of different counter sets among the different threads, which divides the entire counter sets into separate subsets and assigns a subset to each of the application's threads. Furthermore, in order to get a more accurate approximation of the CPI breakdown, the subset of performance counters tracked by each thread is switched regularly, while making sure that at any given time the different subsets are distributed among the threads such that the collection of subsets covers the entire set of performance counters

The direct result of this technique is a trace file containing the performance counter readings for all the threads, where at any given point in the trace each thread has accurate reading for a subset of all the performance counters. We then use CPI-based interpolation to approximate the values for the each thread's missing counters based on the accurate values sampled by other threads. This interpolation is based on the periodic nature on parallel scientific applications, according to which all the threads typically execute the same code at any given time and are therefore likely to experience the same behaviour.

Finally, the interpolated counter values are used to derive a continuous CPI breakdown estimate for the entire runtime of the application.

3.3.3 Interconnection: BW, L

One of the key elements of current large-scale computers is the interconnection network, and although an interconnection network can be characterized by many parameters, latency (L) and bandwidth (BW) are the more representative.

While latency (L) measures the time that it takes a message to travel from one node to another the bandwidth (BW) measures the number of bytes sent per unit of time. The characterization that we want to do is to evaluate the impact of the latency and bandwidth in current applications, in such a way that we can extrapolate the importance of these parameters in the future architectures.

The methodology explained in this section relates to the evaluation of latency and bandwidth in MPI applications by means of using the Dimemas[1] simulator.

[1] http://www.bsc.es/plantillaA.php?cat_id=475

Methodology

Dimemas is a performance analysis tool for message-passing programs. It enables the user to develop and tune parallel applications on a workstation, while providing an accurate prediction of their performance on the parallel target machine. The Dimemas simulator reconstructs the time behavior of a parallel application on a machine modelled by a set of performance parameters. Thus, performance experiments can be done easily. The supported target architecture classes include networks of workstations, single and clustered SMPs, distributed memory parallel computers, and even heterogeneous systems.

Several message-passing libraries are supported, while MPI is the more well-known nowadays. For communication, a linear performance model is used, but some non-linear effects such as network conflicts are taken into account. The simulator allows specifying different task to node mappings.

Dimemas generates trace files that can be analysed by performance analysis tools, like Paraver[2], that enable the user to conveniently examine any performance problems indicated by a simulator run.

The analysis module performs critical path analysis reporting the total CPU usage of different code blocks, as well as their importance for the program execution time. Based on a statistical evaluation of synthetically perturbed traces and architectural parameters, the importance of different performance parameters and the benefits of particular code optimizations can be analyzed.

Figure 2 represents how Dimemas can be used and the benefits of using the Dimemas environment for application analysis, development and tuning. These benefits are based in the possibility of not using a parallel machine to run the application to get the traces and the possibility to analyze different application choices without changing or re-running the application itself.

Using Dimemas instrumentation libraries, the tracefile can be obtained either in a dedicated parallel machine or in a shared sequential machine. Instrumentation libraries only record the CPU time in between communications and the communications primitives, thus the tracefile does not contain any record of network contention or processor preemption. With these records Dimemas will rebuild the application behaviour, using the tracefile and the architectural parameters defined. Per process CPU time is used as opposed to elapsed time. In this way, if a process suffers preemption during the instrumentation, Dimemas will not consider the preempted time and the predicted performance will approximate what would happen in a dedicated machine. In this way, Dimemas enables the analysis (or characterization) of applications for non-existent machines by means of using the architectural parameters and the traces obtained from the application runs.

The loop involving Dimemas and Paraver, the simulator and the visualization tool, is the second benefit of using this environment. The user has the chance to analyze the application behavior when some parameters are changed, for example, what will happen to application execution time if the execution time of a given function is reduced by 50%? In the output information section, some examples describe the different possible analysis.

[2] http://www.bsc.es/plantillaA.php?cat_id=485

In the description of the architectural parameters the user can define different types of parameter, like the architecture of the machine (processors per node, performance of the processors), but here we will focus on the network parameters. The interconnection network is represented with two parameters: number of links from a node to the network, represented by L, and number of buses in the network, represented by B. For each of the links, the parameters latency and bandwidth can be defined with different values to model heterogeneous network architectures.

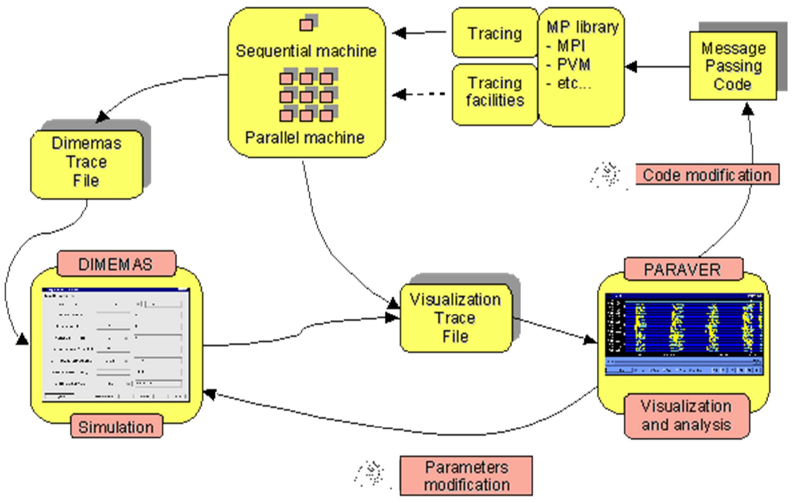


Figure 2 CEPBA-Tools[3] workflow

3.3.4 Memory hierarchy

Performance analysis tools allow the analyst to understand the idiosyncrasies of an application to finally improve it. However, these tools require monitoring regions of the application to provide information to the analyst, leaving the non-monitored regions of code unknown to him or her, which can result in lack of understanding of important parts of the application.

In TERAFLUX we will use a methodology that combining instrumentation and sampling mechanisms is able to enrich the performance analysis experience by supplying performance metrics on non-monitored regions. This methodology can be applied to study other aspects besides the memory hierarchy impact, but it is described here as a matter of example.

This methodology uses computation burst clustering and a mechanism called folding. While clustering automatically detects application structure, folding combines instrumentation and sampling to augment the performance analysis details. Folding provides fine grain performance information from coarse grain sampling on iterative applications. Folding results closely resemble the performance

[3] CEPBA-Tools is the set of performance analysis tools from BSC, mainly the Dimemas simulator, the Paraver visualizer and analyzer and a set of tracing and automatic analysis tools. See http://www.bsc.es/plantillaF.php?cat_id=52

data gathered from fine grain sampling with an absolute mean difference less than 5% and taking benefit of the marginal overhead imposed by sampling at coarse granularity.

Gathering information using sampling requires choosing a proper sampling rate. If the analyst does not know the application well enough, the election of the sampling frequency becomes a blind process and the analyst will first try a default frequency. If the analyst considers that the quantity of samples are not enough to characterize the region of interest within the application, then another execution will be needed with the consequent waste of computing resources.

The sampling frequency must not be too high in order not to disturb the application behavior. The objective of both sampling and folding is to provide large quantity of details while keeping the overhead at minimum. To accomplish our objective we use large sampling periods and we apply the folding process on iterative applications (which are very common in scientific computing). Folding is a post-mortem mechanism that reports the instantaneous evolution of performance metrics (i.e., MFlops) along delimited regions by gathering the sampled metrics along the whole execution.

Folding combines instrumented and sampled information contained in a tracefile to augment the details of the instrumented regions. In the folding process, instrumented and sampled information play different roles. Instrumented information is used to delimit regions and to determine to which region a sample belongs. Sampled information determines how the performance behaviour evolves within the region to which it belongs.

Instead of dealing with the cloud of samples, we perform a polynomial adjustment using the Kriging contouring algorithm. We benefit from the Kriging algorithm in several ways. First, as an analytical model we can compute its derivative and, from the derivative, we can compute the instantaneous rates. Second, and to serve further analyses, we sample the Kriging result to reintroduce the folded metrics as synthetic events into the tracefile at a user requested rate. Finally, the contouring algorithm serves also as a noise reduction mechanism.

As we have seen, the folding mechanism requires some tracefile events to delimit to which region a sample belongs. To delimit the regions, an option is to modify the application at compile or link time in order to obtain these delimiters using instrumentation. Another option is to enable the instrumentation package to do this automatically. In our case, the Extrae instrumentation package instruments MPI calls through the PMPI interface. The resulting tracefile contains information on entry and exit points to the MPI calls. We define a computation region as the code between a MPI exit point and the following MPI entry point. The computation regions are then used as delimiters in the folding process. In order to identify different instances of the same computation region to be folded on the synthetic regions we use a clustering tool. This clustering tool uses a post-mortem density-based clustering algorithm and groups similar computation regions using the metrics contained in the tracefile. The clustering tool reports two different outputs: pair-wise scatter-plots for each of the metrics used in the clustering process and an annotated tracefile indicating the computation clusters.

3.3.5 Memory locality patterns

In order to evaluate hardware and software locality aware policies, we use an application instrumentation tool to obtain application traces. Once we obtain the application traces, we perform different experiments using a cycle-accurate simulator to study and evaluate the different locality policies.

The applications evaluated in the project are large parallel applications, and they are too big to obtain traces of the full execution.

For this reason, we use SimPoint/Optima [Casas07] to identify the most relevant part of the application. This selected representative part of the application is usually a parallel section. Once we identify the most relevant part of a parallel application, we use an instrumentation tool (Pin, BProber or Valgrind) to instrument the application code and obtain the traces.

The traces of an application contain a set of files with information about the application basic blocks, instructions and memory addresses. Within this information, the simulator can reproduce the application behaviour. The instrumentation tool is programmed so that all thread traces maintain application synchronization.

As an example, for OpenMP applications, we split each thread trace in separate files when a barrier is found. Through this mechanism, execution on a simulator can reproduce the application including its threads' synchronization.

3.3.6 The parallelism

The parallelism denotes the amount of calculations that are carried out simultaneously, operating on the principle that large problems can often be divided into smaller ones, which are then solved concurrently ("in parallel").

The parallelism is defined as the ratio of total work to span. If you run an application on many fewer processors than the parallelism, then you should expect linear speedup. We need tools designed for general-purpose parallel programming, but also specialized effective for exploiting dynamic, highly asynchronous parallelism, which can be difficult to write in data-parallel or message-passing style.

The philosophy behind Cilk[CILK546] is that a programmer should concentrate on structuring his program to expose parallelism and exploit locality, leaving the runtime system with the responsibility of scheduling the computation to run efficiently on the given platform. The basic Cilk language is simple. It consists of C with the addition of three keywords to indicate parallelism and synchronization.

Methodology

Let us denote by T_P the execution time of a given computation on P processors. The **work** of the computation is the total time needed to execute all threads in the dag. We can denote the work with T_1, since the work is essentially the execution time of the computation on one processor.

Notice that with T_1 work and P processors, the lower bound $T_P \geq T_1/P$ must hold[4]. The second limit is based on the program's **span**, denoted by T_∞ , which is the execution time of the computation on an infinite number of processors, or equivalently, the time needed to execute threads along the longest path of dependency. The second lower bound is simply $T_P \geq T_\infty$.

[4] This abstract model of execution time ignores memory-hierarchy effects, but is nonetheless quite accurate [Blumofe95].

Cilk's[CILK546] work-stealing scheduler executes a Cilk computation on P processors in time $T_p \leq T1/P + O(T_\infty)$, which is asymptotically optimal. Empirically, the constant factor hidden by the big *O* is often close to 1 or 2 [Blumofe95], and the formula

$$T_P \approx T_1/P + T_\infty$$

is a good approximation of runtime. This performance model holds for Cilk programs that do not use locks. If locks are used, Cilk does not guarantee anything. (For example, Cilk does not detect deadlock situations.) We can explore this performance model using the notion of **parallelism**, which is defined as $\mathcal{P} = T_1/T_\infty$. The parallelism is the average amount of work for every step along the span. Whenever $P \ll \mathcal{P}$, that is, the actual number of processors is much smaller than the parallelism of the application, we have equivalently that $T_1/P \gg T_\infty$. Thus, the model predicts that $T_P \approx T_1/P$, and therefore the Cilk program is predicted to run with almost perfect linear speedup. The measures of work and span provide an algorithmic basis for evaluating the performance of programs over the entire range of possible parallel machine sizes. Cilk's timing instrumentation can measure these two quantities during a run of the program, no matter how many processors are used [CILK546].

3.3.7 TERAFLUX execution model requirements

TM requires a programmer to mark code blocks that access shared data as transactions. Whenever a transaction executes, a runtime system records the transaction's data accesses into a readset and a writeset. These sets are compared with the sets of other concurrently executing transactions for access conflicts (write/write or read/write). If conflicting accesses are detected then one of the conflicting transactions is aborted and restarted. A contention manager decides which transaction to abort. A transaction that completes execution of its code block without being aborted can commit its writeset.

We instrumented our software TM library to collect execution data from the execution of the applications. We present those metrics commonly used to characterize applications in the TM literature, and introduce two new metrics not seen in the TM literature; the transaction execution time histograms and the Instantaneous Commit Rate (ICR).

In transactions (InTX) is the percentage of total time the applications spent executing transactions. For the applications studied, the remaining percentage of time is spent executing serial code. A high InTX means an application spent most of its time executing transactions, thus possibly stressing the TM implementation more than an application with low InTX.

Wasted work shows the percentage of transaction execution time spent executing transactions that subsequently aborted. It is calculated by dividing the total time spent in aborted transactions by the time spent in all (committed and aborted) transactions. High amounts of wasted work can be an indicator of poor contention management decision-making, low amounts of parallelism in the application.

Aborts per Commit (ApC) shows the mean number of aborted transactions per committed transaction. ApC is not directly related to wasted work, but is an indicator for the same issues mentioned for wasted work. For example, high wasted work in combination with a low ApC (aborting a few long/large transactions, and favoring many short/small transactions) may indicate poor contention management decision-making, and studying the application may lead to better contention management policies.

Abort histograms detail how the ApC is spread amongst the transactions; e.g. is the ApC due to a minority of transactions aborting many times before committing, or vice versa?

Contention Management Time (CMT) measures the percentage of time the mean committed transaction spends in performing contention management when conflicts are detected. In combination with wasted work and abort histogram data, it is possible to understand which contention manager may be most effective for the profiled application.

Transaction execution time histograms show the spread of execution times of committed transactions. This metric describes how homogeneous or heterogeneous is the amount of work contained in transactions for a given application.

Instantaneous Commit Rate (ICR) graphs show the proportion of committed transactions at sample points during the execution of the application. ICR includes only completed, i.e. committed or aborted, transactions, and does not include active transactions. Low ICR is indicative of wasted work.

Readset & writeset sizes are a measure of the memory boundedness of committed transactions in an application. They can be used for selecting buffer or cache sizes for Hardware TM (HTM) implementations. Data from non-trivial TM applications gives higher confidence that the hardware will not overflow for a large proportion of transactions. In Section 4 a writeset is always a subset of its corresponding readset because all applications first read data before writing. For other applications, these sets may only overlap, or be distinct.

Readset-to-writeset ratio (RStoWS) shows the mean number of reads that lead to a write in a committed transaction. Execution usually involves reading a number of data elements, performing computation, and writing a result to a data element.

4 Initial characterization and modelling of project kernels/benchmarks/applications

This section presents results of the initial characterizations being performed using the different methodologies presented in the previous section.

4.1 Analytic evaluation

4.1.1 Dataflow analysis

STAP

The purpose of the Space-Time Adaptive Processing (STAP) for Radar application embedded in planes is to detect the position and radial speed of another flying target despite the presence of ground-based or flying jamming devices.

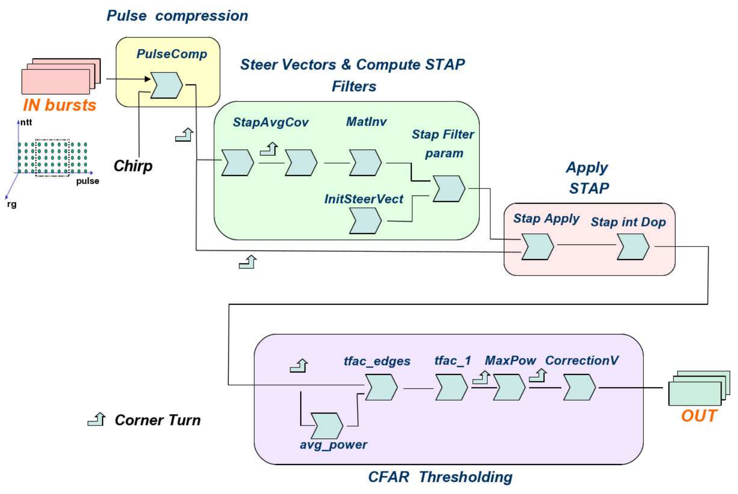


Figure 3 STAP Dataflow

The STAP algorithm, presented in Figure 3, is processing 3-dimensional input data bursts composed of (antenna, pulse rate and range gate), and is decomposed into the following phases:

1. During the Pulse Compression phase, the input signal is converted into complex floating point number data.
2. The Steer Vectors & STAP Filters computation phase which is preparing the necessary filters used by the STAP algorithm.
3. The STAP phase responsible of applying the STAP algorithm.
4. The CFAR Thresholding post-processing phase is responsible for reducing the number of false alarms.

Estimated Throughput / Data & Time profile: As the radar is using 5 different antennas, each with a pulse frequency of 0.5ms and bursts composed of 32 pulses, a full burst need to be processed in up to 5*0.5*32= 80ms. The estimated corresponding throughput and load is appearing on Figure 4.

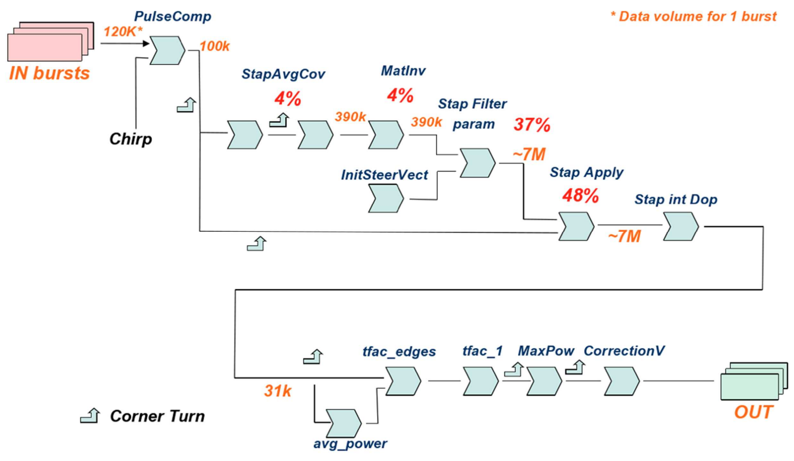


Figure 4 STAP throughput

Viola and Jones (Pedestrian detection)

The pedestrian detection algorithm consists of applying a cascade of simple classifiers in order to assess the existence of a pedestrian in a given subwindow of an image (video frame). The cascade is computed offline and contains several stages. Each stage contains one strong classifier computed from several features. The number of features grows exponentially with the stage index in the cascade. A classifier consists of a thresholded weighted sum of the feature values. In their turn the feature values are computed as a thresholded weighted sum of different simple rectangle filters.
The cascade is applied exhaustively on image tiles at a given scale, which represents the size of the detectable pedestrian. Several scales are to be handled between the smallest scale (given in the offline cascade file, e.g. 18x36 pixel tiles) and a maximum allowable scale (which is a function of the image size). Therefore the cascade file needs to be scaled in order to handle different tile sizes.

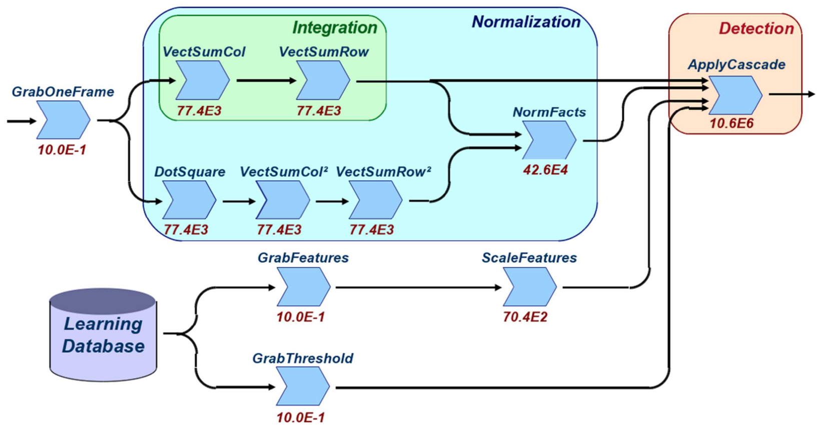


Figure 5 Viola & Jones Dataflow and throughput

The main stages of the pedestrian detection algorithm are the following:
1. Load one frame and original cascade file
2. Compute intermediate representation from the original image (used in computing the rectangle filters efficiently)
3. For all tiles at a given size compute tile variance (needed for normalizing classifier values).
4. Apply first stage of the cascade on all tiles. Retain only valid tiles (according to the stage threshold). Apply second stage on the valid tiles. Update valid tiles list, and so on.
5. Display valid candidate windows after the last stage in the cascade.
6. Pass to next image tile size and repeat 3-5.
7. Post-process results at each tile size.

Estimated Throughput / Data & Time profile: The pedestrian detection algorithm is to be applied with video recorder running at 15 frame per second, the search frame scanning the 32-bit image scaling from x1 (18x36 pixels) to x6 (108x216 pixels). The required throughput for a single frame therefore vary from 4 * 18 * 36 * (241-18) * (321-36) = 166MB to 4 * 108 * 216 * (241-108) * (321-216) = 33MB. At 15FPS, the throughput varies from 2.4GB to 495MB. Figure 5 presents the throughput associated with each elementary operation.

4.2 Evaluation using performance/monitoring tools

4.2.1 Memory BW

Application	Description
GADGET2	Cosmological N-body and smoothed particle hydrodynamics simulations
GROMACS	Molecular dynamics package simulating the Newtonian motion equations of large particle systems
LESLIE3D	Computational fluid dynamics code
MILC	Large scale simulations of four dimensional SU(3) lattice gauge theory
POP	Ocean circulation model
SOCORRO	Self-consistent electronic-structure calculations
SPECFEM3D	Southern California seismic wave propagation based upon the spectralelement method (SEM)
TERA_TF	3D Eulerian hydrodynamics application
VAC4	Solver astrophysical hydrodynamical and magnetohydrodynamical Problems
WRF	Mesocale numerical weather prediction system
ZEUS-MP	Computational fluid dynamics for astrophysical phenomena

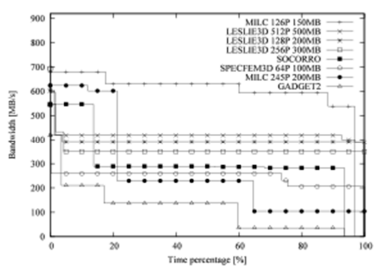
Table 3 List of examined applications

Results

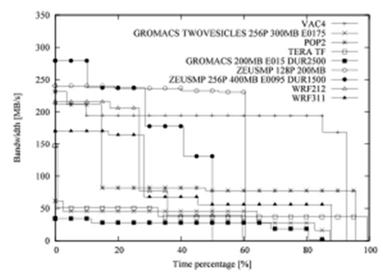
In order to predict the bandwidth requirements, we measured the per-processor bandwidth consumed by each benchmark at three levels: the off-chip memory bandwidth, L2 bandwidth, and L1 bandwidth. Figure 6 (a) and (b) depict the average per-processor off-chip bandwidth, and Figure 6 (c) and 2 depict the bandwidth between the L1 and L2 caches.

For the sake of readability, the benchmarks were split into two groups, based on their bandwidth consumption. As discussed Section 3, the entire execution time is divided into clusters. Phases of an application experiencing similar characteristics — specifically CPI, computation intensity, percentage of TLB misses, and bandwidth requirements — are grouped into the same cluster. For each benchmark, we focus our discussion on the four clusters dominating benchmark's runtime. Therefore, the X axis represents the percentage of the execution time spent in each cluster, and the Y axis shows the measured bandwidth in MB/s.

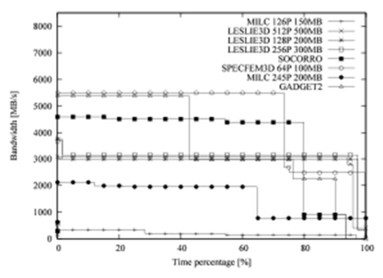
Figure 6 (a) and (b) show the off-chip memory bandwidth measured: Figure 6 (b) shows the results for benchmarks classified as having low memory bandwidth requirements, whereas Figure 6 (a) shows the results for benchmarks classified as bandwidth intensive.



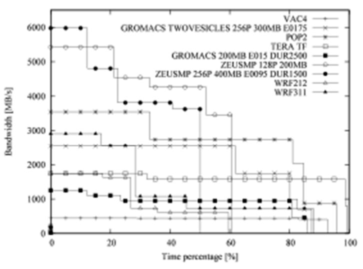
(a) Memory Bandwidth: High BW group



(b) Memory Bandwidth: Low BW group



(c) L2 Bandwidth: High BW group



(d) L2 Bandwidth: Low BW group

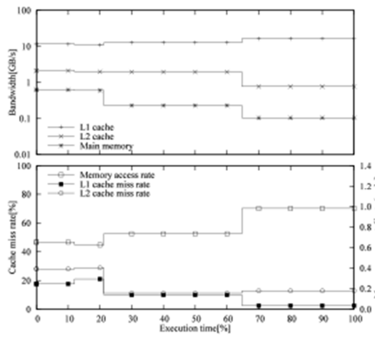
Figure 6 Bandwidth requirements

The figures show that the low-bandwidth benchmarks experience a typical off-chip memory bandwidth of 50–200 MB/s, whereas the high-bandwidth group typically requires between 100 and 400 MB/s, with peaks reaching as high as 700 MB/s. But as these values represent the per-processor average, they are likely to scale linearly when processors are consolidated on a single chip. Placing 100 processors on a chip is therefore likely to require sustained off-chip memory bandwidth of 10 GB/s to 40 GB/s, and may even peak to 70 GB/s.

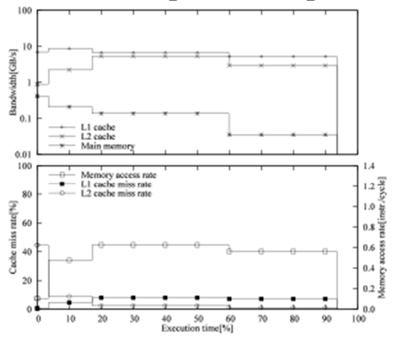
Compared with the off-chip bandwidth, the observed L2 cache bandwidth is an order of magnitude higher. This is understandable as the L2 cache hits filter bandwidth that would otherwise go off-chip (the same conclusion would apply for L1 vs. L2 bandwidth). To better understand the effectiveness of the caches, as well as variations in measured bandwidths between particular clusters, we have investigated three workload related metrics: frequency of memory accesses (the number of instructions per memory access) and the L1 and L2 miss rates.

Figure 7 depicts the above metrics for MILC 3(a), GADGET2 3(b), WRF 3(c) and SOCORRO 3(d), respectively. Each figure is split into two sub-plots — the top plot presents the bandwidth for the main memory, and L2 and L1 caches bandwidth, and the bottom plot shows the frequency of memory accesses, L2 and L1 miss rates of the respective clusters. Both sub-plots have the same X-axis, so that bandwidth variations can be correlated with variations in cache miss rates and/or frequency of memory accesses.

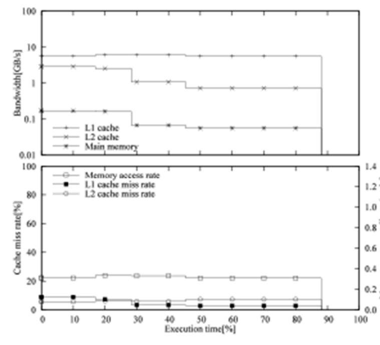
For MILC, the first significant change in bandwidth is observed between second and third cluster, as the memory bandwidth decreases (note the logarithmic bandwidth scale), despite the increase in memory access frequency. This is explained by the reduction in L1 and L2 miss-rates, which indicates that more data is fetched from the caches (mainly the L1, whose bandwidth increases), suggesting the third cluster experiences better data locality. A similar trend is observed between the third and forth cluster, in which the L1 bandwidth increases even further indicating a much higher data locality.



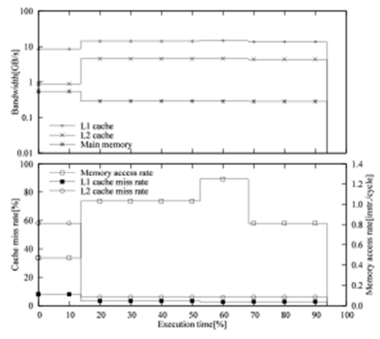
(a) MILC



(b) GADGET2



(c) WRF



(d) SOCORRO

Figure 7 Memory bandwidth requirements of selected applications, and their associated

A similar behaviour can be seen in WRF, when transitioning between second and third cluster, even though in this case we do not see an increase in memory access frequency.

For GADGET2, we observe the effects of different efficiencies of L2 cache. Between the first and second cluster, we see the a large increase in L2 bandwidth accompanied by a dramatic decrease in L2 miss-rate (and a similar, yet less noticeable effect for L1), again suggesting better data locality as L2 hits filter away more of the memory bandwidth. This trend continues between the second and the third cluster. Finally, in the fourth cluster, both L1 and L2 cache miss rates remain on the same level, while frequency of the memory accesses decreases, which leads to decrease in bandwidth demands of all L1 and L2 caches and main memory.

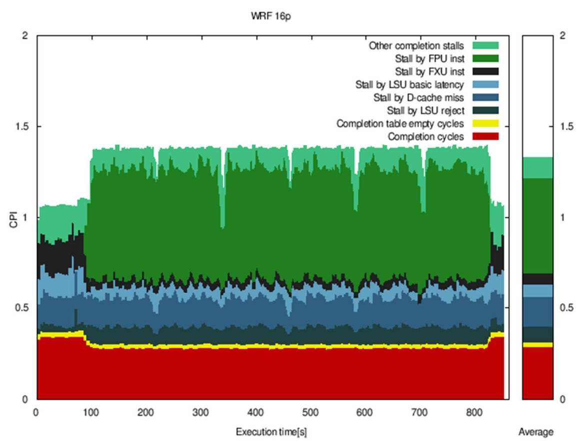
Finally, for SOCORRO we see a big increase in frequency of memory accesses between the first two clusters. The increase is large enough that even though the L1 cache miss-rate decreases slightly, it cannot fully absorb the larger number of memory accesses. Still, the L2 manages to capture most of the memory access frequency, as attested by its decreased miss-rate. As a result, both L1 and L2 bandwidth increase, and the main memory bandwidth decreases.

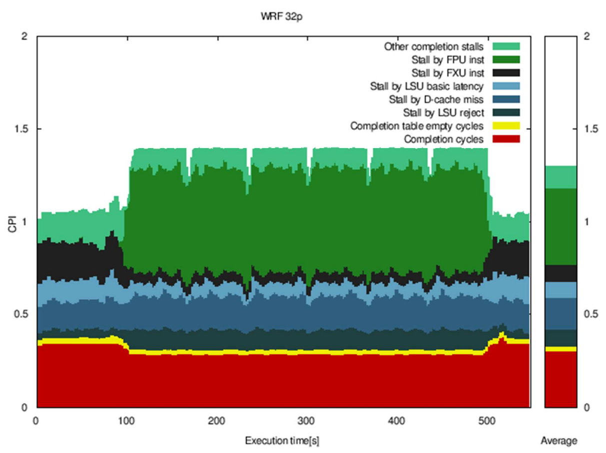
The results suggest that despite the fact that memory bandwidth requirements are seemingly high, in many cases data locality is substantial enough such that the caches capture most of the traffic. The evaluation suggests that, for CMPs consisting of up to ~100 processors, aggressive on-chip caching may suffice in bridging the gap between the memory bandwidth required by parallel applications and the effective off-chip bandwidth supported by current memory technologies. However, our observations also suggest that the tipping point lurks at ~200 processors on a chip, at which point existing memory technologies will not be able to provide applications with sufficient memory bandwidth.

4.2.2 CPI Breakdown

Results

Figure 8 and Figure 9 show the CPI breakdown for WRF and GADGET2 (due to lack of space, we only show results for two of the applications analyzed, see [Pavlovic10] for more details).





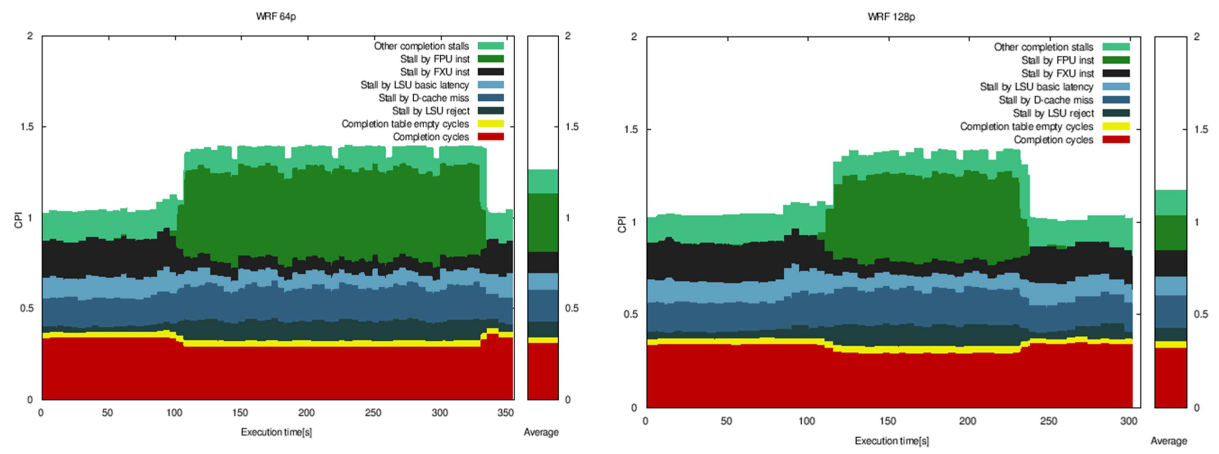
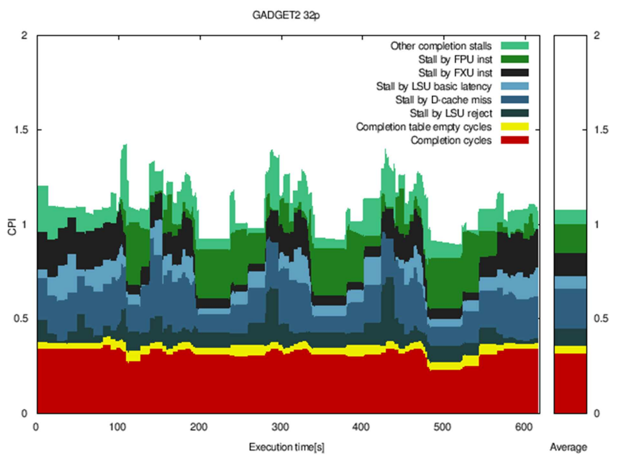


Figure 8 CPI breakdown for WRF, running on 16, 32, 64, 128 processors. Results are shown for a single sample thread.

Figure 8 illustrates the CPI breakdown for WRF. The CPI pattern clearly describe the data initialization part at the beginning of the execution, the parallel computational phase at the heart of the execution, followed by a termination phase. Furthermore, it is clear that the initialization and termination phases do not scale as the level of parallelism increases and become the dominant time consuming factor as the computational phase is accelerated with the increased parallelism.

When examining the scaling of the parallel phase, it is clear that WRF is a floating-point intensive application, as the biggest source of pipeline stalls in WRF is the functional units, contributing roughly 50% of the CPI throughout the computational phase. The rest of the pipeline stalls are mostly associated with the memory system, and are mostly attributed to data cache misses, either directly or indirectly (LSU full), or due to the basic latency of the load/store units.



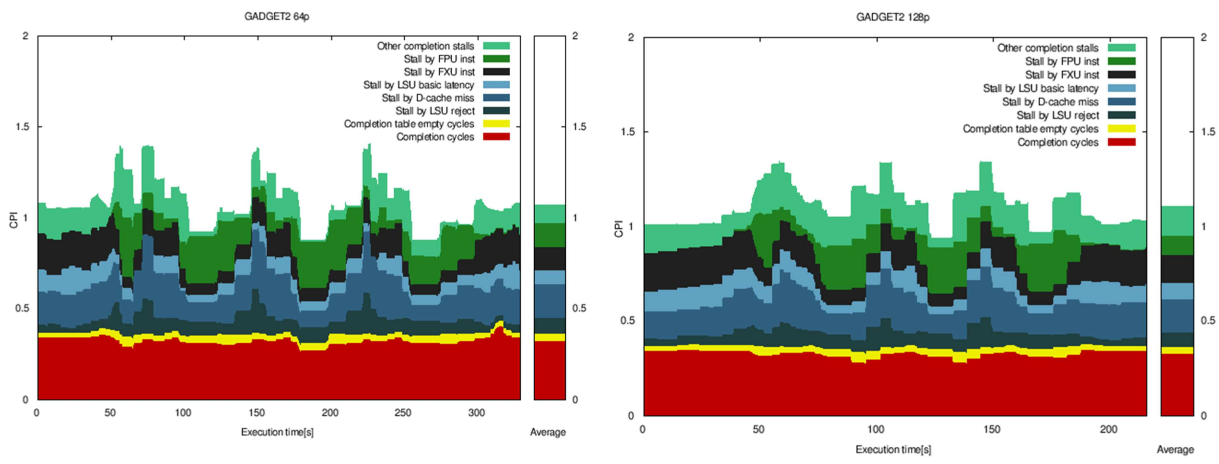


Figure 9 CPI breakdown for GADGET2, running on 32, 64, 128 processors (the dataset is too big to fit on a 16 processor configuration). Results are shown for a single sample thread.

A similar CPI breakdown is observed for GADGET2 as well, as shown in Figure 9. GADGET2's three parallel computational phases appear as the floating point intensive segments in mid-run. These phases are separated by MPI communication phases – an artifact of the programming model, rather than an intrinsic component of the computation. While the main causes for pipeline stalls are generally similar to WRF, GADGET2 also has a noticable portion of its stalls caused by the fixed-point functional units (FXU), and miscellenious pipeline stalls (branch mispredictions, I-cache misses, microcoding overheads, etc.).

Overall, these initial results are very promising in the context of the TERAFLUX project, as dataflow programming models are ideally suited for further parallelization of floating point and fixed point operations, which account for 50% of the pipeline stalls. Moreover, dataflow models facilitate judicious data scheduling and can typically provide highly efficient use of the memory heirarchy – much better than that of prevalent cache architectures that rely on demand fetching.

4.2.3 Interconnection: BW, L

This section presents initial characterization results that have been obtained with the application Gadget. The first step was to generate a tracefile of a real run in the MareNostrum supercomputer. For example, the tracefile on the top of the figure below shows a plot of the tracefile when using 256 processes. The tracefile is a timeline where each line represents a process of the application (although the lines are collapsed to make a more compact image). Different colors represent different activities. The light blue represents the phase of the application when computation is performed, an all the other colors represent the time when the application is executing an MPI call.

The image below is the reconstruction that Dimemas is able to perform when we simulate with an ideal network (latency =0, bandwidth = infinite). Different behaviours can be observed for different areas of the applications. For example, the first Allgather + sendrecv area is not affected by the use of an ideal network, denoting that the network is not limiting the performance of the application in this area. However, there are areas where the impact of the network is important: alltoall, sendrec and waitall.

Similarly to this analysis, with Dimemas we can perform exhaustive exploration of the sensitivity of the applications to network parameters, varying them and ploting the results in 2 or 3D charts.

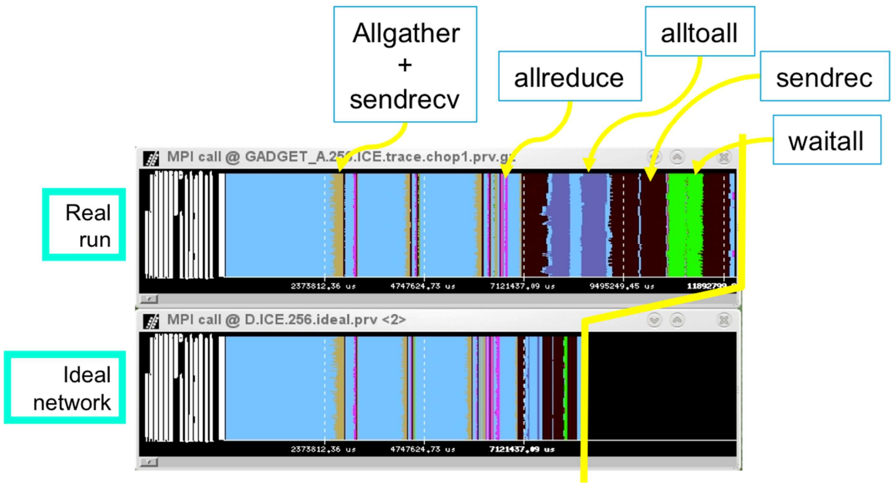


Figure 10 Impact of network in GADGET

4.2.4 Memory hierarchy

In the analyses we study some of the more time-consuming computation regions [Servat09, Servat11]. Every computation region is described by a figure of the folded hardware counters that were gathered in the execution using sampling. Moreover, each computation region is manually divided in different phases according to the variation on the slope in any of the shown metrics. Each application analysis contains also two informative tables per computation phase. One table contains hardware counter rates and phase duration. The other table describes the more time consuming code lines within the phase providing either file and line number locations or the name of the routine.

The plots and the table containing the hardware counter rates allows us determining potential performance problems, either by direct observation or by relation between them.

PEPC

PEPC was executed on a machine with four Intel Dunnington hex-core processors running at 2.4GHz. The binary was generated by GNU C and Fortran compilers version 4.3.4 using -O3 as optimization parameter and run using 16 cores. The application ran with a medium size input and gathering 10 samples per second. We show in Figure 11 the results for two of the more time consuming computation regions in the application, namely Cluster 3 and Cluster 4. Each plot contains the folded cumulative sample values for three different metrics: committed instructions, L2 cache misses and data TLB misses.

The overall behaviour of the computation regions shown is very bad in terms of MIPS rates. In fact, several phases present extremely low MIPS rates. Even the highest MIPS rate is 1500, that represents an IPC of 0.6 at the processor speed, which is far from the ideal IPC of the machine (4).

Cluster 3 presents four different phases. The proportion of MIPS to L2 cache miss rates outlines that 5 of every 100 instructions miss in L2. Phase B shows the better MIPS rate in this computation region. The L2 cache miss rate in this phase is similar to phase A. Then comes phase C, which runs at 80 MIPS, which is the worst rate in the region. The high L2 and TLB miss rates seem to indicate that they are the limiting factor of this phase. We also have to mention the variability of the TLB misses in this phase. In contrast, L2 cache does not suffer from this issue because its size and associativity is larger than the TLB (TLB is 4-way with 32 entries whereas L2 cache is 12-way with 49152 entries). Improving the performance on this phase would include using large memory pages. Finally, phase D that executes a large quantity of operations. This can increase the register pressure in the innermost part of the loop, so improvements using loop unrolling techniques may be limited because the number of operations surpasses the number of registers (8). However, the nested loop may benefit from vectorization due to the reduction of control instructions and the availability of additional registers.

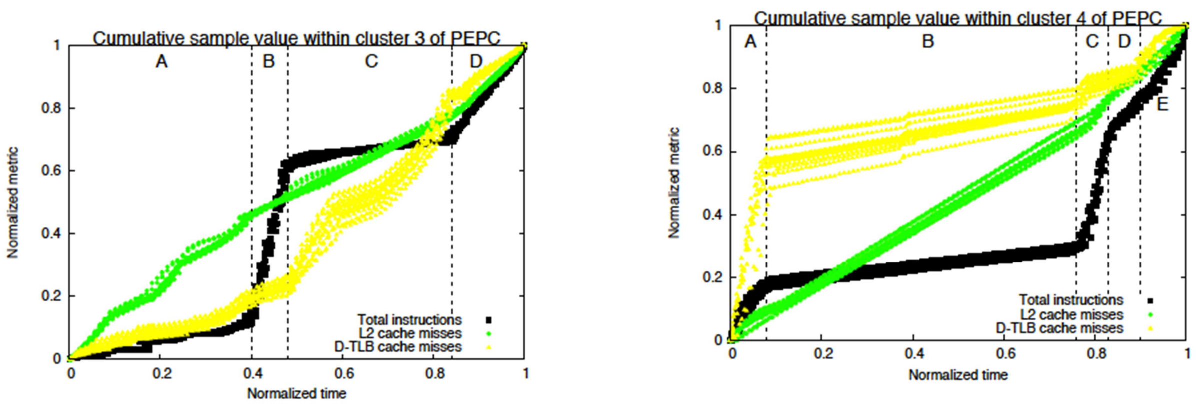


Figure 11 Impact of L2 cache misses and D-TLB misses in performance of PEPC

Cluster 4 is segmented in five phases. In phase A, we see a large variable increase in the number of data TLB misses which is derived from the large number of memory allocations done in the related part of the code. Phase B shows a extremely low MIPS rate (close to 16), committing less than 10% of instructions of the region in 60% of the computation time. Phase C shows the highest MIPS and L2 cache miss rates in this computation region, 400 and 4 million respectively. Phase D makes a list of unfinished particles using a loop that accesses consecutive addresses controlled by an if clause. Finally, phase E refers to a loop with an irregular access pattern, which is caused by accessing different locations of a hash table.

4.2.5 Memory locality patterns

For this task, we have performed a series of experiments using the PARSEC/NAS suites using a 16-core CMP configuration. We have evaluated Locality management policies using a comparison of two CMP architectures: a cache-based CMP architecture (L1 and L2), and a Local Memory-based architecture (L0 and LP as Local Memory).

The cache-based CMP divides its memory accesses between L1 cache, L2 cache, Remote L2 cache and Main Memory. The L1 cache is a 4-way 32KB cache, and the L2 is a 4-way 256KB cache.

In this architecture, Locality aware policies are automatically achieved due to the hardware cache coherence protocols.

The Local Memory-based CMP divides its memory accesses between L0 cache, LP or Local Memory, Remote LP, Main Memory and Page Migrations. The L0 cache is a 4-way, 4KB cache that only stores data on the Local Memory. Each LP is a scratchpad memory of 256KB. In this architecture, Locality is achieved through page migration from off-chip memory to the on-chip local memories.

When a page access count exceeds a certain threshold, a data page is automatically migrated to the on-chip memories. Figure 12 shows the data layout for all memory access for the PARSEC/NAS benchmarks.

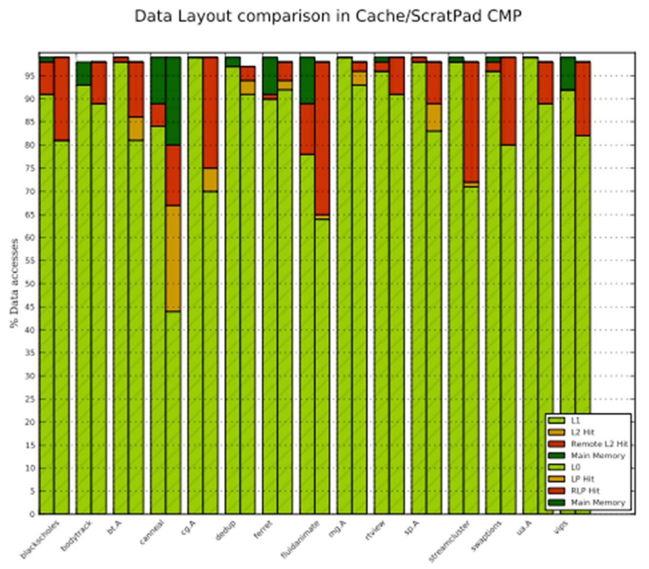


Figure 12 Data layout for PARSEC/NAS benchmarks

Each benchmark is represented by two stacked-bar graphs: the left bar is the cache-based CMP architecture, and the right bar is the local memory-based CMP architecture. Each bar shows stacked the layout of all memory accesses. Through these bars, we can observe where most of the data is allocated for all applications/architecture.

As it can be observed, using page migration, most benchmarks achieve a similar access time for the cache and local-memory based architectures.

4.2.6 Transactional Memory

This section presents the initial characterization results for the benchmark Lee-TM and a subset of the STAMP benchmark (Genome, KMeans, and Vacation).

Lee-TM is a circuit router that makes connections automatically between points. Routing is performed on a 3D grid that is implemented as a multidimensional array, and each array element is called a grid cell. The application loads connections (as pairs of spatial coordinates) from an input file, sorts them into ascending length order (to reduce ‘spaghetti’ routing), and then loads them into thread-local queues in a round-robin manner. Each thread then attempts to find a route from the first point to the second point of each connection by performing a breadth-first search, avoiding any grid cells occupied by previous routings. If a route is found, backtracking lays the route by occupying grid cells. Concurrent routing requires writes to the grid to be performed transactionally. Lee-TM is fully parallel, with conflicts at concurrent read/write or write/write accesses to a grid cell. A second version of Lee-TM has been implemented that uses early release. This version removes grid cells from the readset during the breadthfirst search. Two transactions may be routable in parallel, i.e. the set of grid cells occupied by their routes does not overlap, but because of their spatial locality, the breadth-first search of one transaction reads grid cells to which the second transaction writes its route, thus causing a read/write conflict. Removing grid cells from the readset during the breadth-first search eliminates such false-positive conflicts.

Genome is a gene sequencer that rebuilds a gene sequence from a large number of equal-length overlapping gene segments. Each gene segment is an object consisting of a character string, a link to the start segment, next segment, and end segment, and overlap length. The application executes in three phases. The first phase removes duplicate segments by transactionally inserting them into a hash set. The second phase attempts to link segments by matching overlapping string subsegments. If two segments are found to overlap then linking the two segments (by modifying the links in each gene segment object, and setting the overlap length) and removing them from the hash set is done transactionally, as multiple gene segments may match and result in conflict. The matching is done in a for-loop that starts by searching for the largest overlap (length-1 characters, since duplicates were removed in the first phase), down to the smallest overlap (1 character). Thus, conflict is likely to rise as execution progresses since smaller overlaps will lead to more matches. In the third phase, a single thread passes over the linked chain of segments to output the rebuilt gene sequence. The execution of Genome is completely parallel except for the third phase.

KMeans clusters objects into a specified number of clusters. The application loads objects from an input file, and then works in two alternating phases. One phase allocates objects to their nearest cluster (initially cluster centers are assigned randomly). The other phase re-calculates cluster centers based on the mean of the objects in each cluster. Execution repeatedly alternates between the two phases until two consecutive iterations generate, within a specified threshold, similar cluster assignments. Assignment of an object to a cluster is done transactionally, thus parallelism is controlled by the number of clusters. Execution consists of the parallel phase assigning objects to clusters, and the serial phase checking the variation between the current assignment and the previous.

Vacation simulates a travel booking database in which multiple threads transactionally book or cancel cars, hotels, and flights on behalf of customers. Threads can also execute changes in the availability of cars, hotels, and flights transactionally. Each customer has a linked list holding his reservations. The execution of Vacation is completely parallel, but available parallelism is limited by the number of relations in the database and the number of customers.

All experiments are performed on a 4 x dual-core 2.2GHz Opteron-based (i.e. an 8-core NUMA shared memory) machine with openSUSE 10.1, 16GB RAM, and using Sun Java 1.6.0 64-bit and our modified Software TM library. Note that the results presented will vary depending on the TM system implementation used for the experiments. A TM system takes non-deterministic decisions about scheduling and aborting specific transactions.

Configuration Name	Application	Configuration
Gen	Genome	gene length:16384, segment length:64, number of segments:4194304

KMeansL	KMeans	low contention min clusters:40, max clusters:40, threshold:0.00001, input file:random10000 12
KMeansH	KMeans	high contention min clusters:20, max clusters:20, threshold:0.00001, input file:random10000 12
VacL	Vacation	low contention relations:65536, % of relations queried:90, queries per transaction:4, number of transactions:1024768
VacH	Vacation	high contention as above, but % of relations queried:10, queries per transactions:8
Lee-TM-t	Lee-TM	Lee w/o early release early release:false, input file:mainboard.txt
Lee-TM-ter	Lee-TM	Lee with early release early release:true, input file:mainboard.txt

Table 4 Transactional memory benchmarks used in the initial characterization

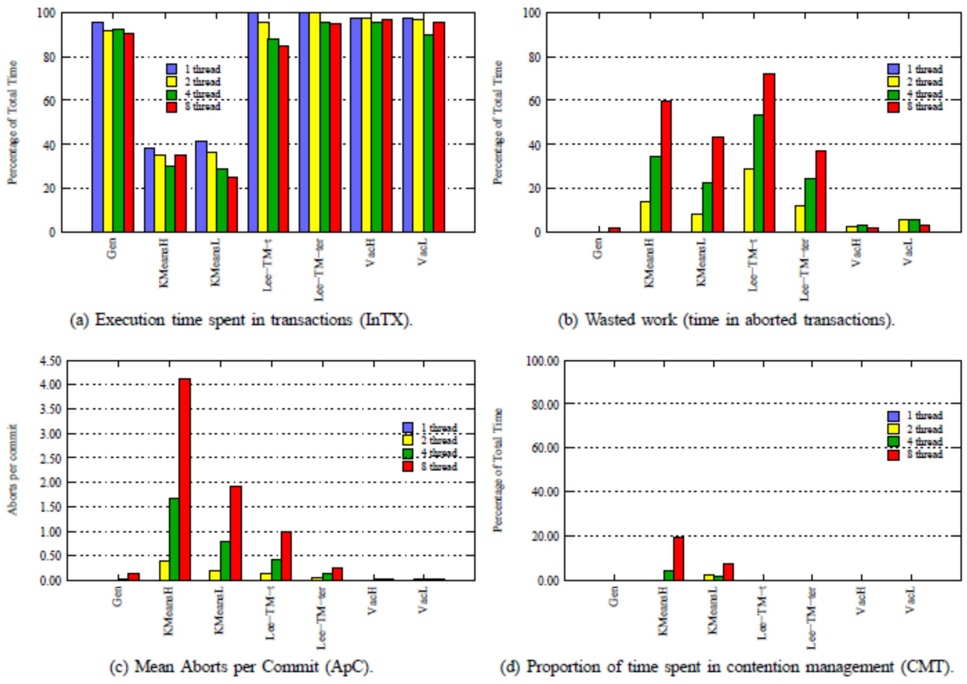


Figure 13 Transactional memory application level metrics

Figure 13(b) shows wasted work results. Gen and Vac have little wasted work (less than 10%). KMeans and Lee-TM-t have large amounts of wasted work, e.g. on 8 threads the wasted work is between 35% and 70%. Applications with large amounts of wasted work may be suitable candidates for studying contention management. KMeans speedup is limited by the significant sequential phase seen in Figure 13(a), and large amounts of wasted work. This shows that poor scalability can have its root in wasted work.

Figure 13(c) shows ApC results. KMeans has the highest ApC, followed by Lee-TM, Gen, and finally Vac. KMeansH has an ApC four times higher than Lee-TM-t, but 30% less wasted work. This suggests Lee-TM-t aborts large/long transactions as it has fewer aborts, yet large amounts of time spent in the aborted transactions. Figure 13(d) shows CMT results. CMT is negligible for Gen, Lee-TM, and Vac. At 8 threads KMeansH has 20% CMT, KMeansL has 10% CMT, but Lee-TM-t has almost none.

Figure 14 and Figure 15 present the histograms depicting the aborts (static view), while Figure 16 provides a dynamic view of how the aborts are spread during the execution using ICR.

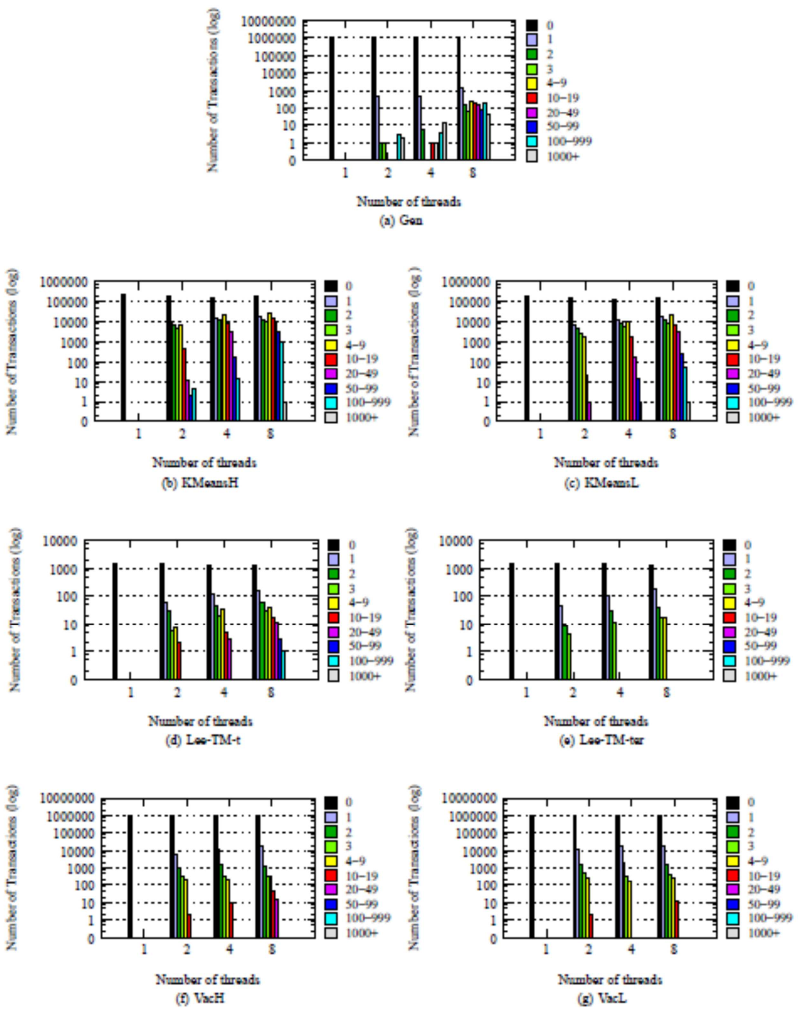


Figure 14 Abort histograms for TM benchmarks. Each bar represents the number of transactions that aborted a given number of times before actually committing. Note y-axis uses logarithmic scale.

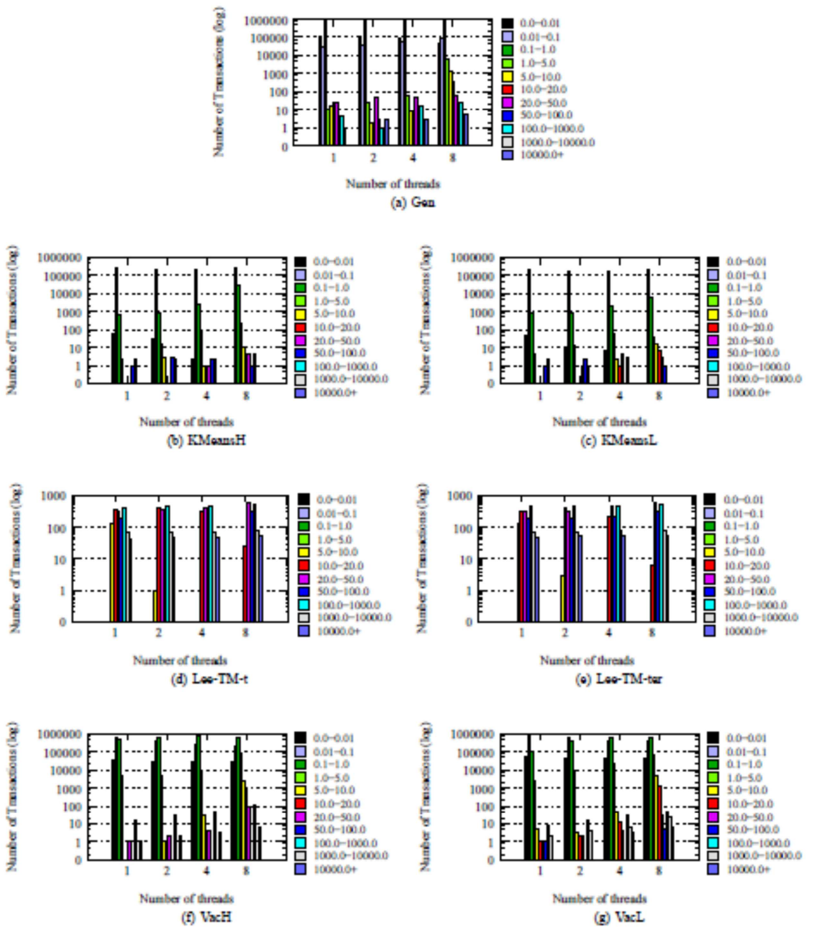


Figure 15 Transaction execution time histograms. The colour of each bar represents a range of elapsed execution times in milliseconds. The vertical axis represents the number of transactions completing within the time range.

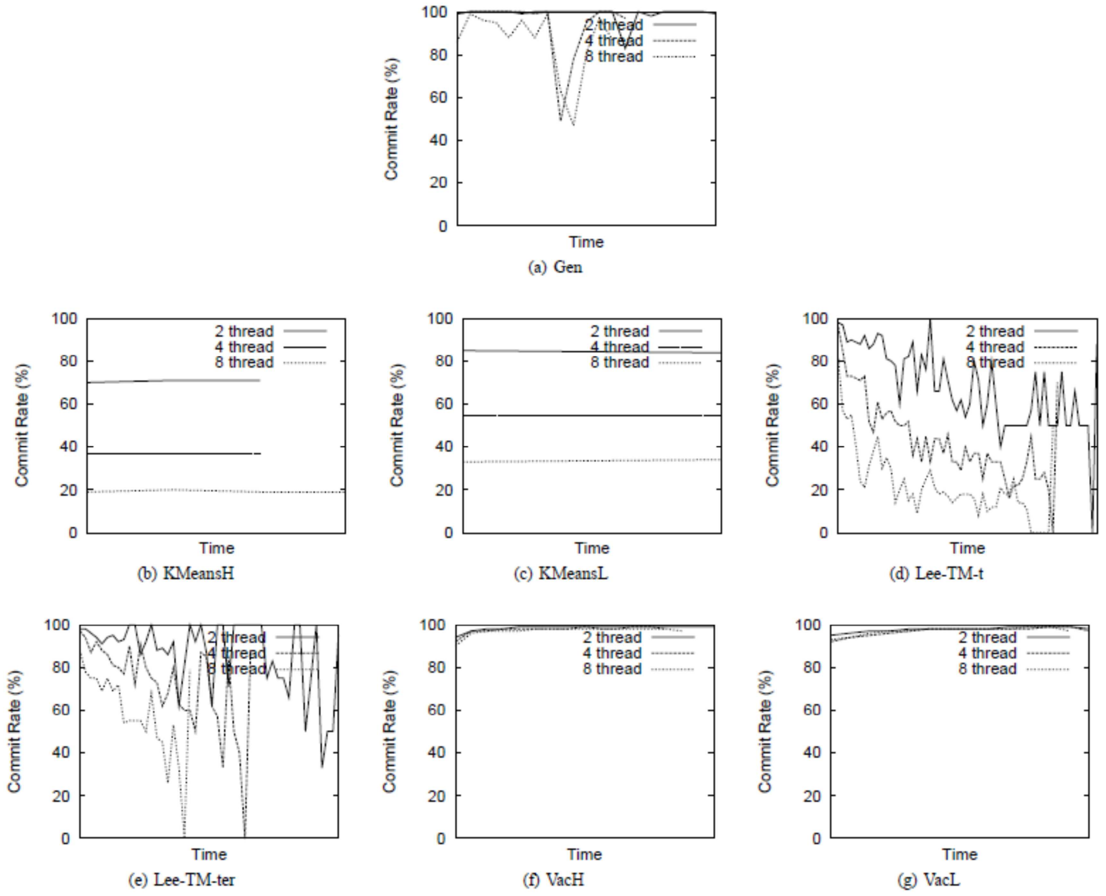


Figure 16 Instantaneous commit rate (ICR) graphs. In our experiments we sampled at 5 second intervals. The commit rate for 2,4 and 8 threads are plotted.

5 Conclusions

This document describes the first deliverable of TERAFLUX WP2. The activities in task 2.1 comprised the selection of the kernels, benchmarks and applications to be used in the project and the preparation of a disk image that enables all partners to have a uniform environment to test the applications. Although task 2.1 was planned to finish on M6 of the project and its objectives have been accomplished, we do not rule out adding more applications to the list in case the consortium considers this necessary.

Task 2.2 has begun during the second half of the first project year. The different methodologies to be used to characterize the applications have been identified and initial results presented. During the second year of the project, the task will continue its characterization activities.

References

[Amesfoort10] Alexander S. van Amesfoort, Ana Lucia Varbanescu, and Henk J. Sips, “Towards Parallel Application Classification using Quantitative Metrics”, Department of Software Technology, Delft University of Technology, 2010

[Blumofe95] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In Proceedings of the Fifth, ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), pages 207-216, Santa Barbara, California, July 1995

[Casas07] Marc Casas, Rosa M. Badia, Jesus Labarta, Automatic structure extraction from MPI applications tracefiles. In Europar, Aug 2007.

[CILK546] Cilk 5.4.6 Reference Manual, Supercomputing Technologies Group MIT Laboratory for Computer Science http://supertech.lcs.mit.edu/cilk

[Frigo98] Matteo Frigo, Charles E. Leiserson, Keith H. Randall, “The implementation of the Cilk-5 multithreaded language”, PLDI '98 Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation   ACM New York, NY, USA ©1998

[Gonzalez09] Gonzalez, J., Gimenez, J., Labarta, J.: Automatic detection of parallel applications computation phases. In Intl. Symp. On Parallel and Distributed Processing, May 2009.

[Pavlovic10] Milan Pavlovic, Yoav Etsion, and Alex Ramirez, "Can Manycores Support the Memory Requirements of Scientific Applications?". In Workshop on Applications for Multi and Many Core Processors (A4MMC), Jun 2010.

[Servat09] Harald Servat, Germán Llort, Judit Gimenez, Jesús Labarta: Detailed Performance Analysis Using Coarse Grain Sampling. Euro-Par Workshops 2009: 185-198.

[Servat11] Harald Servat, Germán Llort, Judit Gimenez, Kevin Huck, Jesus Labarta, Unveiling internal evolution of parallel application computation phases, Technical Report UPC-DAC-RR-2011-10.

[Strohmaier04] Erich Strohmaier, Hongzhang Shan, “Architecture Independent Performance Characterization and Benchmarking for Scientific Applications”, MASCOTS '04 Proceedings of the The IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems.