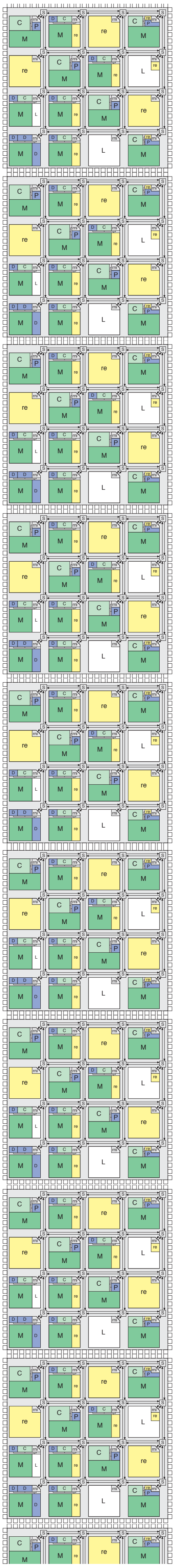
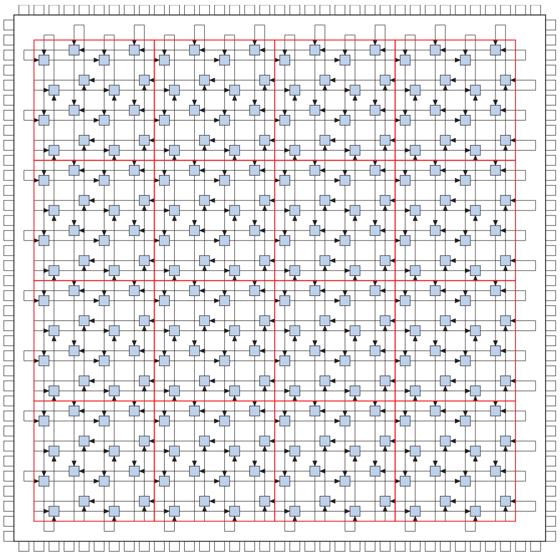
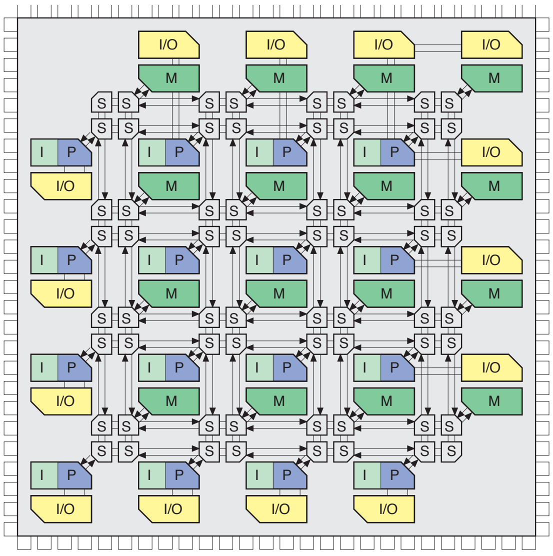






HPPC 2010

(Hand-out) Proceedings of the

4th Workshop on
Highly Parallel Processing
on a Chip

August 31, 2010, Ischia - Naples, Italy
Organizers Martti Forsell and Jesper Larsson Träff

in conjunction with

the 16th International European Conference on
Parallel and Distributed Computing (Euro-Par)
August 31-September 3, 2010, Ischia - Naples, Italy

Sponsored by VTT universität wien Euro-Par

(Hand-out) Proceedings of the

4th Workshop on Highly Parallel Processing on a Chip

August 31, 2010, Ischia - Naples, Italy
http://www.hppc-workshop.org/

in conjunction with

the 16th International European Conference on Parallel and Distributed Computing (Euro-Par)
August 31-September 3, 2010, Ischia - Naples, Italy

August 2010
Handout editors: Martti Forsell and Jesper Larsson Träff
Printed in Finland and Austria

CONTENTS

FOREWORD

Technological developments are bringing parallel computing back into the limelight after some years of absence from the stage of main stream computing and computer science between the early 1990ties and early 2000s. The driving forces behind this return are mainly advances in VLSI technology: increasing transistor densities along with hot chips, leaky transistors, and slow wires make it unlikely that the increase in single processor performance can continue the exponential growth that has been sustained over the last 30 years. To satisfy the needs for application performance, major processor manufacturers are instead planning to double the number of processor cores per chip every second year (thus reinforcing the original formulation of Moore's law). We are therefore on the brink of entering a new era of highly parallel processing on a chip. However, many fundamental unresolved hardware and software issues remain that may make the transition slower and more painful than is optimistically expected from many sides. Among the most important such issues are convergence on an abstract architecture, programming model, and language to easily and efficiently realize the performance potential inherent in the technological developments.

This is fourth time we organize the Workshop on Highly Parallel Processing on a Chip (HPPC). Again, it aims to be a forum for discussing such fundamental issues. It is open to all aspects of existing and emerging/envisaged multi-core processors with a significant amount of parallelism, especially to considerations on novel paradigms and models and the related architectural and language support. To be able to relate to the parallel processing community at large, which we consider essential, the workshop has been organized in conjunction with Euro-Par, the main European (and international) conference on all aspects of parallel processing.

The Call-for-papers for the HPPC workshop was launched early in the year, and at the passing of the submission deadline we had received 18 submissions, which were relevant to the theme of the workshop and of good quality. The papers were swiftly and expertly reviewed by the program committee, all of them receiving 3-4 qualified reviews. We thank the whole of the program committee for the time and expertise they put into the reviewing work, and for getting it all done within the rather strict timelimit. Final decision on acceptance was made by the program chairs based on the recommendations from the program committee. This year the themes of manuscripts matched well to the scope of the workshop and we were able to accept full 8 contributions, resulting in an acceptance ratio of about 44%. The 8 accepted contributions will be presented at the workshop today, together with two forward looking invited talks by Rolf Hoffmann and Jim Held on he massively parallel computing model GCA and Intel Lab's Single-chip Cloud Computer.

This handout includes the workshop versions of the HPPC papers and the abstracts of the invited talks. Final versions of the papers will be published as post proceedings in a Springer LNCS volume containing material from all the Euro-Par workshops. We sincerely thank the Euro-Par organization for giving us the opportunity to arrange the HPPC workshop in conjunction with the Euro-Par 2010 conference. We also warmly thank our sponsors VTT, University of Vienna and Euro-Par for the financial support which made it possible for us to invite Rolf Hoffmann and Jim Held, both of whom we also sincerely thank for accepting our invitation to come and contribute.

Finally, we welcome all of our attendees to the Workshop on Highly Parallel Processing on a Chip in the beautiful city of Ischia, Italy. We wish you all a productive and pleasant workshop.

HPPC organizers
Martti Forsell, VTT, Finland
Jesper Larsson Träff, University of Vienna, Austria

ORGANIZATION

Organized in conjuction with the 16th International European Conference on Parallel and Distributed Computing

WORKSHOP ORGANIZERS

Martti Forsell, VTT, Finland
Jesper Larsson Träff, University of Vienna, Austria

PROGRAM COMMITTEE

Martti Forsell, VTT, Finland
Jim Held, Intel, USA
Peter Hofstee, IBM, USA
Chris Jesshope, University of Amsterdam, The Netherlands
Ben Juurlink, Technical University of Berlin, Germany
Jörg Keller, University of Hagen, Germany
Christoph Kessler, University of Linköping, Sweden
Dominique Lavenier, IRISA - CNRS, France
Ville Leppänen, University of Turku, Finland
Lasse Natvig, NTNU, Norway
Sabri Pllana, University of Vienna, Austria
Jürgen Teich, University of Erlagen-Nürnberg, Germany
Jesper Larsson Träff, University of Vienna, Austria
Theo Ungerer, University of Augsburg, Germany
Uzi Vishkin, University of Maryland, USA

SPONSORS

VTT, Finland	http://www.vtt.fi
University of Vienna	http://www.univie.ac.at
Euro-Par	http://www.euro-par.org

PROGRAM

4th Workshop on Highly Parallel Processing on a Chip (HPPC 2010)

TUESDAY AUGUST 31, 2010 Ischia - Naples

SESSION 1 - Models and memory organizations

09:30-09:35 Opening remarks - *Jesper Larsson Träff and Martti Forsell, University of Vienna, VTT*
09:35-10:35 Keynote - The Massively Parallel Computing Model GCA - *Rolf Hoffmann, Technical University of Darmstadt*
10:35-11:00 Low-Overhead Organizations for the Directory in Future Many-Core CMPs - *Alberto Ros and Manuel E. Acacio, Technical University of Valencia, University of Murcia*

11:00-11:30 -- Break --

SESSION 2 - Programming multicores

11:30-11:55 A Work Stealing Algorithm for Parallel Loops on Shared Cache Multicores - *Marc Tchiboukdjian, Vincent Danjean, Thierry Gautier, Fabien Le Mentec and Bruno Raffin, CNRS - CEA/DAM, DIF, Grenoble University, INRIA*
11:55-12:20 Resource-agnostic programming for many-core microgrids - *Thomas Bernard, Clemens Grelck, Michael Hicks, Christopher Jesshope and Raphael Poss, University of Amsterdam*
12:20-12:45 Programming Heterogeneous Multicore Systems using Threading Building Blocks - *George Russell, Paul Keir, Alastair Donaldson, Uwe Dolinsky, Andrew Richards and Colin Riley, Codeplay Software, University of Glasgow, Oxford University*

12:45-15:30 -- Lunch --

SESSION 3 - Applications and optimizations

15:30-15:55 Fine-grained parallelization of a Vlasov-Poisson application on GPU - *Guillaume Latu, CEA, IRFM*
15:55-16:20 Highly Parallel Implementation of Harris Corner Detector on CSX SIMD Architecture - *Fouzhan Hosseini, Amir Fijany and Jean-Guy Fontaine, Italian Institute of Technology*
16:20-16:45 Static Speculation as Post-Link Optimization for the Grid Alu Processor - *Ralf Jahr, Basher Shehan, Sascha Uhrig and Theo Ungerer, University of Augsburg*

16:45-17:30 -- Break --

SESSION 4 - Networks and clouds

17:30-17:55 A Multi-Level Routing Scheme and Router Architecture to support Hierarchical Routing in Large Network on Chip Platforms - *Rickard Holsmark, Shashi Kumar and Maurizio Palesi, Jönköping University, University of Catainia*
17:55-18:55 Keynote - Intel Lab's "Single-chip Cloud Computer", an IA Tera-scale Research Processor - *Jim Held, Tera-Scale Computing Research, Intel*
18:55-19:00 Closing remarks - *Jesper Larsson Träff and Martti Forsell, University of Vienna, VTT*

KEYNOTE

The Massively Parallel Computing Model GCA

Rolf Hoffmann, Professor, Technical University of Darmstadt, Germany

Abstract: The Global Cellular Automata Model (GCA) is an extension of the Cellular Automata Model (CA). Whereas in the CA model each cell is connected via fixed links to its local neighbors, in the GCA model each cell is connected via data dependant dynamic links to any (global) cells of the whole array. The GCA cell state does not only contain data information but also link information. The cell state is synchronously updated according to a local rule, modifying the data and the link information. Similar to the CA model, only the own cell state is modified. Thereby write conflicts cannot occur. The GCA model is related to the CROW (concurrent read owners write) model and it can be used to describe a large range of applications. GCA algorithms can be described in the language GCA-L which can be compiled into different target platforms: a generated data parallel multi-pipeline architecture, a NIOS II multi-softcore architecture and a NVIDIA GPU.

Bio: *Rolf Hoffmann is Professor and leader of the Computer Architecture Group in the Computer Science Department of the Technical Unversity of Darmstadt Germany since 1978. He graduated 1970 at TU Berlin (Dipl.-Ing. Electrical Engineering), and received there 1974 the Ph.D. in Computer Science. He published a book on Microprogramming and Computer Design and many papers on special computer architectures and their FPGA implementations. Since 1994 several accelerators for Cellular Automata (CEPRA series) were implemented in his group. He is mainly working on novel massively parallel computing models; in particular he proposed the Global Cellular Automata model.*

Low-Overhead Organizations for the Directory in Future Many-Core CMPs*

Alberto Ros[1] and Manuel E. Acacio[2]

[1]Dpto. de Informática de Sistemas y Computadores
Universidad Politécnica de Valencia, 46022 Valencia (Spain)
[2]Dpto. de Ingeniería y Tecnología de Computadores
Universidad de Murcia, 30100 Murcia (Spain)
aros@gap.upv.es, meacacio@ditec.um.es

Abstract. If current trends continue, today's small-scale general-purpose CMPs will soon be replaced by multi-core architectures integrating tens or even hundreds of cores on-chip. These many-core CMPs will implement the hardware-managed, implicitly-addressed, coherent caches memory model. Cache coherence in these designs will be maintained through a directory-based cache coherence protocol implemented in hardware. The organization of the directory structure will be a key design point due to the requirements in area that it will pose. In this work we study the effects on performance, network traffic and area that the use of compressed sharing codes for the directory will have in many-core CMPs. In particular, we select two compressed sharing codes previously proposed by us in the context of large-scale shared-memory multiprocessors that have very small area requirements. Simulation results of 32-core CMPs show that degradations of up to 32% in performance and 350% in network traffic are experienced. Additionally, since some proposals for efficient multicast support in on-chip networks have recently appeared, we also consider the case of using this kind of support in combination with the compressed sharing codes. Unfortunately, we found that multicast support is not enough to remove all the performance degradation that the compressed sharing codes introduce and barely can reduce network traffic.

1 Introduction

In the last years we have witnessed the substitution of single-core processors by multi-core ones. Following the Moore's Law that establishes that the number of transistors doubles every 18 months, it is expected that current small-scale general-purpose chip-multiprocessors (CMPs) will soon be followed by multi-core architectures integrating tens or even hundreds of cores on-chip [3]. Architectures of this type are usually known as many-core CMPs.

Many-core CMPs will be probably designed as arrays of identical or close-to-identical building blocks (tiles) connected over a switched direct network [12, 16]. Tiled architectures provide a scalable solution for supporting families of products with varying computational power, managing the design complexity, and effectively using the resources available in advanced VLSI technologies. As an example, Intel has recently announced the 48-core Single-chip Cloud Computer [1], an experimental research microprocessor that has been developed in the context of the Tera-scale Computing Research Program. More specifically, the Single-chip Cloud Computer consists of 24 tiles with two IA cores per tile, which are interconnected by means of a 24-router mesh network providing 256 GB/s bisection bandwidth.

* This research was supported by the Spanish MEC and MICINN, as well as European Commission FEDER funds, under Grants CSD2006-00046 and TIN2009-14475-C04, and PROMETEO from Generalitat Valenciana (GVA) under Grant PROMETEO/2008/060.

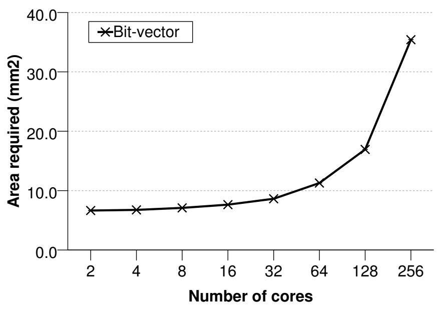


Fig. 1: Area (mm^2) required for a 1MB cache module when the bit-vector sharing code is used

On the other hand, if current trends continue, future many-core CMP architectures will implement the hardware-managed, implicitly-addressed, coherent caches memory model [6]. With this memory model, all on-chip storage is used for private and shared caches that are kept coherent in hardware by using a cache coherence protocol. In this way, each tile contains at least one level of cache memory that is private to the local core (the L1 in this work), and the first level of shared cache (commonly, the L2 cache) is physically distributed between the tiles of the system.

The cache coherence protocol will be a key design issue in these architectures since it will add requirements of area and energy consumption to the final design, and therefore, could restrict severely its scalability. When the number of cores is large, as is the case of many-core CMPs, the best way of keeping cache coherence is by implementing a directory-based protocol, which reduces energy consumption compared to broadcast-based protocols by keeping track of the caches that hold copies of each block in a directory structure. In tiled CMPs, the directory structure is distributed between the L2 cache banks, usually included into the L2 tags' portion [16]. In this way, each tile keeps the sharing information of the blocks mapped to the L2 cache bank that it contains. This sharing information comprises two main components[1]: the *state bits* used to codify one of the three possible states the directory can assign to the line (*Uncached*, *Shared* and *Private*), and the *sharing code*, that holds the list of current sharers. Most of the bits of each directory entry are devoted to codifying the sharing code. Since the directory must be stored as part of the on-chip L2 cache, it is desirable that its size be kept as low as possible. Moreover, a hard to scale directory organization could require to re-design the L2 cache to adapt the tile to the range of cores that is expected for the CMP.

In a traditional directory organization, each directory entry keeps track of the sharers of the corresponding memory block through a simple bit-vector (one bit per private cache). In Figure 1, we plot the area (in mm^2) that one 1MB 4-way L2 module would take as the number of cores grows from 2 to 256 (area estimations are based on CACTI. Refer to Section 4 for more details). As it can be seen, while the number of cores keeps below 16 the bit-vector sharing code barely impacts area requirements. However, from 16 cores on, the use of bit-vectors would entail too much area overhead and more area efficient sharing codes would be required.

One approach for reducing directory area requirements in the context of traditional shared-memory multiprocessors is the use of compressed sharing codes. Compressed sharing codes store the full directory information in a compressed way to use fewer number of bits, introducing a loss of precision compared to *exact* ones[2]. This means that when this information is reconstructed, some of the cores codified in the sharing code are real sharers and must receive the coherence

[1] Apart from other implementation-dependent bits.
[2] Bit-vector is an example of *exact* sharing code.

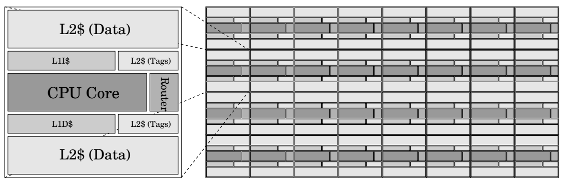


Fig. 2: Organization of the tile assumed in this work and a 4×8 tiled CMP

messages, whereas some other cores are not sharers actually and *unnecessary* coherence messages will be sent to them. *Unnecessary* coherence messages lead to increased miss latencies, since more messages are required to resolve caches misses. Moreover, *unnecessary* coherence messages also entail extra traffic in the interconnection network and useless cache accesses, which will increase energy consumption. Conversely, a bit-vector directory does not generate unnecessary coherence messages and thus shows the best results in terms of both performance and energy consumption.

In this work we study the effects on performance, network traffic and area required by the directory structure that the use of compressed sharing codes will have in many-core CMPs. In particular, we select two compressed sharing codes previously proposed by us in the context of large-scale shared-memory multiprocessors, Binary Tree (BT) and Binary Tree with Symmetric Nodes (BT-SN) [2], and that have very small area requirements. Simulation results of 32-core CMPs show that degradations of up to 32% in performance and 350% in network traffic are experienced. Additionally, since some proposals for efficient multicast support in on-chip networks have recently appeared [11], we also consider the case of using this kind of support in combination with the compressed sharing codes. Unfortunately, multicast support is not enough to remove completely the performance degradation that the compressed sharing codes introduce (performance degradations of 10% on average are still observed when BT is used) and barely can reduce network traffic.

The rest of the paper is organized as follows. First of all, we will give more details regarding the target CMP architecture in Section 2. Subsequently, in Section 3 we will present a couple of compressed sharing codes based on the concept of multilayer clustering that have small overhead in terms of area. Next, in Section 4, we will describe the evaluation environment that we are assuming, and the results of the evaluation will be shown in Section 5. Finally, Section 6 closes the work and points future directions to be explored.

2 Base Architecture

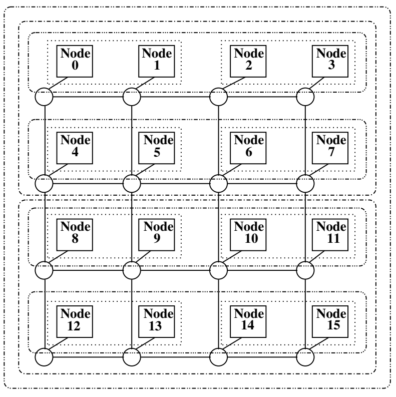
A tiled CMP architecture consists of a number of replicated *tiles* connected over a switched direct network. Each tile contains a processing core with primary caches (both instruction and data caches), a slice of the L2 cache, and a connection to the on-chip network. Cache coherence is maintained at the L1 caches. In particular, it is employed a directory-based cache coherence protocol, with directory information stored in the tags' part of the L2 cache modules. The L2 cache is shared among the different processing cores, but it is physically distributed between them. Therefore, some accesses to the L2 cache will be sent to the local slice while the rest will be serviced by remote slices (L2 NUCA architecture [5]). Moreover, for simplicity the L1 and L2 caches are inclusive, that is to say, all the blocks included in any L1 cache keep an entry in the L2 cache. Figure 2 shows the organization of a tile (left) and a 16-tile CMP (right). From now on, we will use the terms tile and node interchangeably.

3 Multi-layer Clustering Concept

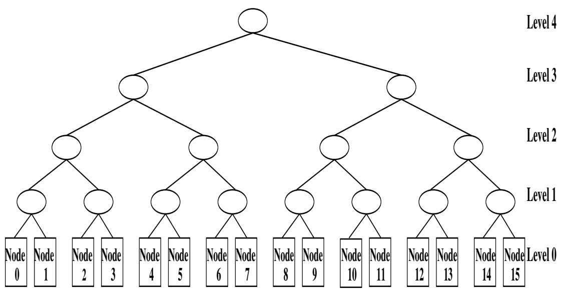
This section presents two compressed sharing code organizations based on the *multi-layer clustering* approach previously proposed in [2].

Multi-layer clustering assumes that nodes are recursively grouped into clusters of equal size until all nodes are grouped into a single cluster. Compression is achieved by specifying the smallest cluster containing all the sharers (instead of indicating *all* the sharers). Compression can be increased even more by indicating only the level of the cluster in the hierarchy. In this case, it is assumed that the cluster is the one containing the home node for the memory block. This approach is valid for any network topology.

Although clusters can be formed by grouping any integer number of clusters in the immediately lower layer of the hierarchy, we analyze the case of using a value equal to two. That is to say, each cluster contains two clusters from the immediately lower level. By doing so, we simplify binary representation and obtain better granularity to specify the set of sharers. This recursive grouping into layer clusters leads to a logical binary tree with the nodes located at the leaves.



(a) *Physical* system



(b) *Logical* system

Fig. 3: Multi-layer clustering approach example

As an application of this approach, two compressed sharing codes were previously proposed in [2]. The sharing codes can be shown graphically by considering the distinction between the *logical* and the *physical* organizations. For example, we have a 16-tile CMP with a mesh as the interconnection network, as shown in Figure 3(a), and we can imagine the same system as a binary tree (multi-layer system) with the nodes located at the leaves of this tree, as shown in Figure 3(b). Note that this tree only represents the grouping of nodes, not the interconnection between them. In this representation, each subtree is a cluster. Clusters are also shown in Figure 3(a) by using dotted lines. It can be observed that the binary tree is composed of 5 layers or levels ($\log_2 N + 1$, where N is a power of 2). From this, the following two compressed sharing codes were derived in [2]: *Binary tree* (BT) and *Binary tree with symmetric nodes* (BT-SN).

3.1 Binary Tree (BT)

Since nodes are located at the leaves of a tree, the set of nodes (sharers) holding a copy of a particular memory block can be expressed as the minimal subtree that includes the home node

Table 1: System parameters

32-core CMP			
GEMS Parameters		**SICOSYS Parameters**	
Processor frequency	4 GHz	Network frequency	2 GHz
Cache hierarchy	Inclusive	Topology	8x4 Mesh
Cache block size	64 bytes	Switching technique	Wormhole, Multicast
Split L1 I & D caches	128KB, 4 ways,	Routing technique	Deterministic X-Y
	4 hit cycles	Message size	4 flits data, 1 flit control
Shared unified L2 cache	1MB/tile, 4 ways,	Routing time	2 cycles
	7 hit cycles	Link latency (one hop)	2 cycles
Memory access time	300 cycles	Link bandwidth	1 flit/cycle

and all the sharers. This minimal subtree is codified using the level of its root (which can be expressed using just $\lceil \log_2 (\log_2 N + 1) \rceil$ bits). Intuitively, the set of sharers is obtained from the home node identifier by changing the value of some of its least significant bits to *don't care*. The number of modified bits is equal to the level of the above mentioned subtree. It constitutes a very compact sharing code (observe that, for a 128-node system, only 3 bits per directory entry are needed). For example, consider a 16-node system such as the one shown in Figure 3(a), and assume that nodes 1, 4 and 5 hold a copy of a certain memory block whose home node is 0. In this case, node 0 would store 3 as the tree level value, which is the one covering all sharers (see Figure 3(b)). Unfortunately, this would include as well nodes 0, 2, 3, 6 and 7 that do not have copy of such memory block and that, thus, would receive unnecessary coherence messages on a subsequent coherence event.

3.2 Binary Tree with Symmetric Nodes (*BT-SN*)

We also introduce the concept of symmetric nodes of a particular home node. Assuming that 3 additional symmetric nodes are assigned to each home node, they are codified by different combinations of the two most-significant bits of the home node identifier (note that one of these combinations represents the home node itself). In other words, symmetric nodes only differ from the corresponding home node in the two most significant bits. For instance, if 0 were the home node, its corresponding symmetric nodes would be 4, 8 and 12. Now, the process of choosing the minimal subtree that includes all the sharers is repeated for the symmetric nodes. Then, the minimum of these subtrees is chosen to represent the sharers. The intuitive idea is the same as before but, in this case, the two most significant bits of the home identifier are changed to the symmetric node used. Therefore, the size of the sharing code of a directory entry is the same as before plus the number of bits needed to codify the symmetric nodes (for 3 sym-nodes, 2 bits). In the previous example, nodes 4, 8 and 12 are the symmetric nodes of node 0. The tree level could now be computed from node 0 or from any of its symmetric nodes. In this way, the one which encodes the smallest number of nodes and includes nodes 1, 4 and 5 is selected. In this particular example, the tree level 3 must be used to cover all sharers, computed from node 0 or node 4.

4 Evaluation environment

We perform the evaluation using the full-system simulator Virtutech Simics [8] extended with Multifacet GEMS 1.3 [9], that provides a detailed memory system timing model. Since the

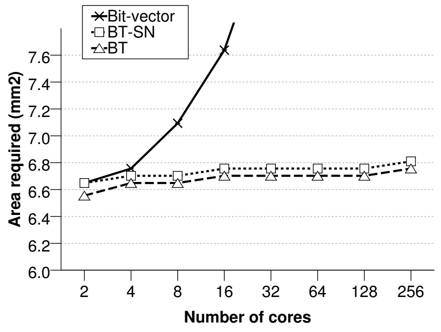


Fig. 4: Area (mm^2) required for a 1MB cache module when bit-vector, BT or BT-SN are used

network modeled by GEMS 1.3 is not very precise, we have extended it with SICOSYS [10], a detailed interconnection network simulator. We simulate a 32-tile CMP architecture as the one described in Section 2. The values of the main parameters used for the evaluation are shown in Table 1. Cache latencies have been calculated using the CACTI 5.3 tool [13] for 45nm technology. We also have used CACTI to measure the area of a 1MB 4-way L2 cache bank that includes the different sharing codes assumed in this work. In this study, we assume that the length of the physical address is 44 bits, like in the SUN UltraSPARC-III architecture [4].

The ten applications used in our simulations cover a variety of computation and communication patterns. *Barnes* (8192 bodies, 4 time steps), *FFT* (256K points), *Ocean* (258x258 ocean), *Radix* (1M keys, 1024 radix), *Raytrace* (teapot), *Volrend* (head) and *Water-Sp* (512 molecules, 4 time steps) are scientific applications from the SPLASH-2 benchmark suite [15]. *Unstructured* (Mesh.2K, 5 time steps) is a computational fluid dynamics application. *MPGdec* (525_tens_040.m2v) and *MPGenc* (output of *MPGdec*), are multimedia applications from the APLBench suite [7]. We account for the variability in multithreaded workloads by doing multiple simulation runs for each benchmark in each configuration and injecting random perturbations in memory systems timing for each run.

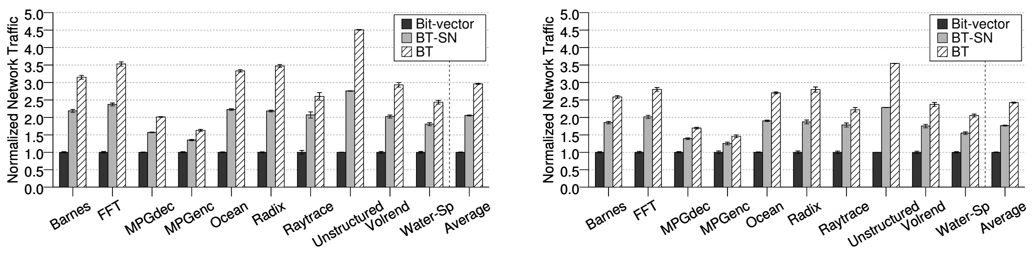
5 Evaluation results

We start this section by comparing the area overhead introduced by the different organizations for the sharing code considered in this work (i.e., bit-vector, BT and BT-ST). Next, we study the impact that the compressed sharing codes have on network traffic. For that, we consider both an interconnection network with and without multicast support. Finally, we end with a comparison between the three directory organizations in terms of the execution times that they obtain for the ten applications described in the last section.

5.1 Impact on area overhead

Figure 4 plots the total area (in mm^2) that would be required by a 1MB 4-way cache module when bit-vector, BT and BT-SN sharing codes are used. Due to the limited number of cores used in our simulations (32), we evaluate BT-SN assuming only one symmetric node. In this way, the size of BT-SN is equal to the size of BT plus 1 bit to codify whether the home node or the symmetric node is being used in the codification.

As shown in Figure 4 (and discussed in the introduction of this work), the area overhead that the bit-vector sharing code entails does not scale with the number of cores. Obviously,



(a) Without multicast (b) With multicast

Fig. 5: Normalized network traffic for bit-vector, BT and BT-SN

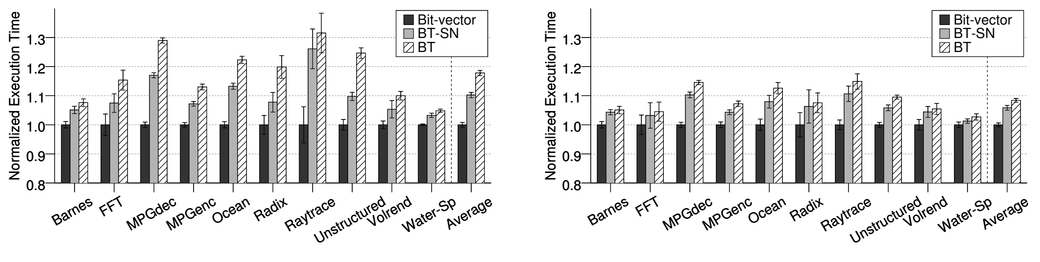
the size of the bit-vector (in bits) increases linearly with the number of cores. For this reason, the bit-vector could be a good option for a small number of cores. However, from 16 cores on the increase in area that the bit-vector conveys makes it unfeasible (the area overhead becomes almost 100% for the 64-core configuration). On the other hand, the size of BT and BT-SN barely increases with the number of cores. Moreover, the total number of bits needed by BT and BT-SN is very small in all cases ($\lceil \log_2 (\log_2 N + 1) \rceil$ bits and $\lceil \log_2 (\log_2 N + 1) \rceil + 1$ bits, respectively). In this way, the area overhead of BT and BT-SN is very low (less than 5% for the 256-core configuration) and keeps almost constant with the number of cores. This makes that BT and BT-SN can be considered as promising alternatives to bit-vector for future may-core CMPs, since besides introducing very small overheads in terms of area, these sharing codes would allow to support families of CMPs with varying number of cores and using exactly the same tile (without requiring any modifications in the directory structure).

5.2 Impact on network traffic

Although compressed sharing codes can drastically reduce the size of the directory, their counterpart is that they could increase the number of coherence messages as a consequence of the in-excess codification of the sharers that they perform. Increasing the number of coherence messages leads to more traffic being injected in the interconnection network of the CMP. Since previous works have identified the interconnection network as one of the most important elements of the CMP from the point of view of energy consumption (consuming almost 40% of the total energy budget in the Raw processor [14]), more traffic at the end means more energy.

Figure 5 shows the amount of network traffic that would be generated for bit-vector, BT and BT-SN for the 32-core CMP configuration assumed in this work. In particular, each bar plots the number of bytes transmitted through the interconnection network (the total number of bytes transmitted by all the switches of the interconnect) normalized with respect to the bit-vector case. We present results considering both a network with unicast support (a) and with multicast support (b).

As shown in Figure 5(a), the use of BT has severe impact on the amount of network traffic and degradations ranging from approximately 50% for *MPGenc* to 350% for *Unstructured* are found. The problem with BT is that when one of the sharers is far from the home node in the logical tree structure illustrated in Figure 3(b), the root of the tree is selected as the minimum tree level covering both the home node and the sharer, which results in all cores being actually codified. We have found that this situation occurs frequently in most applications, which explains the significant amount of extra traffic for BT. In particular, the average number of coherence



(a) Without multicast (b) With multicast

Fig. 6: Execution time for 32 cores

messages that are sent on a coherence event[3] increases from 2 in bit-vector to more than 20 in BT. On the contrary, when BT-SN is considered the tree level that covers all the sharers can be computed from either the home node or its symmetric node. This leads to noticeable reductions in the average number of coherence messages (12 in BT-SN), which leads to important savings in network traffic when compared with BT. Unfortunately, BT-SN does not mitigate completely the extra traffic introduced by BT and degradations of approximately 100% on average are still observed. Again, when two or more cores, distant in the logical tree, share a memory block, the root of the tree would be codified by BT-SN.

Obviously, the provision of multicast support at the interconnection network level can alleviate the levels of extra traffic. More specifically, in Figure 5(b) we show the results obtained when we take advantage of multicast support for sending coherence messages (invalidations and cache-to-cache transfer commands). Efficient implementations of such kind of multicast support in on-chip networks have recently been proposed [11]. Unfortunately, using multicast support for factorizing efficiently also the response messages is not a trivial issue. So, in this work we assume that responses to coherence commands are unicast messages. As it can be seen, the use of multicast support is an step forward in achieving the network traffic levels obtained by bit-vector, and it is especially useful when BT is considered (average traffic overhead is reduced from 200% without multicast support to 150%). Anyway, the fact that multicast support is available just for the coherence commands and not for their associated responses limits its benefits.

5.3 Impact on execution time

The degradations previously reported in terms of network traffic finally translate into increases in terms of execution time. In Figure 6 we show how the use of BT and BT-SN impacts applications' execution times, considering an interconnection network with and without multicast support, (a) and (b) respectively. Again, all results have been normalized with respect to the bit-vector case.

As observed in Figure 6(a), the use of BT without multicast support has important consequences on performance. In particular, the execution time grows from less than 10% for *Barnes* and *Water-Sp* to more than 30% for Raytrace (19% on average). In general, the greater number of messages that are needed with BT to resolve every coherence event leads to longer cache miss latencies, and therefore, execution times. Obviously, the extent of the degradation in execution time will depend on the particular characteristics of each application (L1 cache miss rate, average number of coherence messages per cache miss, kind of synchronization used, etc.). This is why there is no direct correlation between the amount of extra traffic reported in Figure 5(a) and the

[3] By coherence event we refer to a situation where the home node must use the sharing code to send coherence messages (invalidations or cache-to-cache transfer commands).

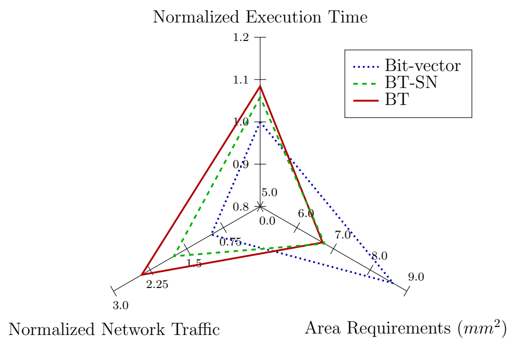


Fig. 7: Trade-off between area, performance and network traffic for BT, BT-SN and bit-vector (32 cores and multicast support are assumed)

degradation in execution time shown in Figure 6(a). On the other hand, when BT-SN is used instead of BT, the average overhead in terms of execution time is reduced to a half (10%). In this case, significant reductions in execution time are observed for most applications. The exceptions are *Barnes* and *Water-Sp*, that hardly see their execution times reduced when BT-SN is used, even when significant savings in terms of network traffic were reported.

The effects of using multicast support with BT and BT-SN are analyzed in Figure 6(b). As before, multicast support has significant impact on execution time when BT is assumed. In this case, average degradation falls from 19% to less than 10%. Although all applications benefit from multicast support, *FFT*, *MPGdec*, *Radix*, *Raytrace* and *Unstructured* are the most affected (in all these cases performance degradation entailed by BT is reduced to more than a half). Finally, and as it was reported for network traffic, multicast support does not help much in reducing performance overhead when BT-SN is considered. In this case, what dominates cache miss latencies is the time taken to collect all responses to a coherence event, which is not optimized with the assumed multicast support.

6 Conclusions and Future Work

The organization of the directory needed to maintain cache coherence will be a key design point in future many-core CMPs. In this work we have analyzed the effects that the BT and BT-SN compressed sharing codes have on area, network traffic (as representative of the energy consumed in the interconnection network) and performance in the context of many-core chip-multiprocessors. In particular, we have found that although very area-efficient directories could be derived based on these two sharing codes (with area overheads of less than 5%), the degradations in terms of network traffic (200% for BT and 100% for BT-SN) as well as execution time (20% for BT and 10% for BT-SN) that they entail could preclude them from being employed in future many-core CMPs. Moreover, we have studied the case of having an interconnection network with multicast support, and have found that although BT can significantly benefit from such kind of support (degradations in execution time and network traffic are reduced to 8% and 150% respectively), BT-SN barely finds any benefits from it. The reasons why multicast support is unable to hide the degradation that BT and BT-SN introduce are two. First, multicast support is only used for sending coherence commands but not for collecting the responses. An second, even when an efficient mechanism able to provide combined responses were used, more destinations for the coherence commands still implies more traffic and longer cache miss latencies. As a summary of the results, Figure 7 shows the trade-off between area, performance and network traffic for the sharing codes evaluated in this work.

Our future work includes new organizations for the sharing code aimed at reducing the amount of unnecessary coherence messages that BT-SN entails but having similar requirements in terms of area. Additionally, we are studying the possibility of including support in the interconnection network for discarding unnecessary coherence messages as they travel to their destination. Finally, we are extending our simulation tools to compare the different directory organizations in terms of their energy requirements (considering both static and dynamic energy consumption).

References

1. Single-chip Cloud Computer. *http://techresearch.intel.com/articles/Tera-Scale/1826.htm*.
2. M. E. Acacio, J. González, J. M. García, and J. Duato. A new scalable directory architecture for large-scale multiprocessors. In *7th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 97–106, Jan. 2001.
3. S. Borkar. Thousand core chips: A technology perspective. In *44th Annual Design Automation Conference*, pages 746–749, June 2007.
4. T. Horel and G. Lauterbach. UltraSPARC-III: Designing third-generation 64-bit performance. *IEEE Micro*, 19(3):73–85, May 1999.
5. C. Kim, D. Burger, and S. W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *10th Int. Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, pages 211–222, Oct. 2002.
6. J. Leverich, H. Arakida, A. Solomatnikov, A. Firoozshahian, M. Horowitz, and C. Kozyrakis. Comparing memory systems for chip multiprocessors. In *34th Int'l Symp. on Computer Architecture (ISCA)*, pages 358–368, June 2007.
7. M.-L. Li, R. Sasanka, S. V. Adve, Y.-K. Chen, and E. Debes. The ALPBench benchmark suite for complex multimedia applications. In *Int'l Symp. on Workload Characterization*, pages 34–45, Oct. 2005.
8. P. S. Magnusson, M. Christensson, and J. Eskilson, et al. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, Feb. 2002.
9. M. M. Martin, D. J. Sorin, and B. M. Beckmann, et al. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *Computer Architecture News*, 33(4):92–99, Sept. 2005.
10. V. Puente, J. A. Gregorio, and R. Beivide. SICOSYS: An integrated framework for studying interconnection network in multiprocessor systems. In *10th Euromicro Workshop on Parallel, Distributed and Network-based Processing*, pages 15–22, Jan. 2002.
11. S. Rodrigo, J. Flich, J. Duato, and M. Hummel. Efficient unicast and multicast support for CMPs. In *41st IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, pages 364–375, Nov. 2008.
12. M. B. Taylor, J. Kim, and J. Miller, et al. The raw microprocessor: A computational fabric for software circuits and general purpose programs. *IEEE Micro*, 22(2):25–35, May 2002.
13. S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi. CACTI 5.1. Technical Report HPL-2008-20, HP Labs, Apr. 2008.
14. H. Wang, L.-S. Peh, and S. Malik. Power-driven design of router microarchitectures in on-chip networks. In *36th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, pages 105–111, Dec. 2003.
15. S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *22nd Int'l Symp. on Computer Architecture (ISCA)*, pages 24–36, June 1995.
16. M. Zhang and K. Asanović. Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors. In *32nd Int'l Symp. on Computer Architecture (ISCA)*, pages 336–345, June 2005.

A Work Stealing Algorithm for Parallel Loops on Shared Cache Multicores

Marc Tchiboukdjian, Vincent Danjean, Thierry Gautier*, Fabien Lementec, and Bruno Raffin

MOAIS Project, INRIA- LIG
ZIRST 51, avenue Jean Kuntzmann
38330 Montbonnot Saint Martin, France
`(marc.tchiboukdjian, vincent.danjean, fabien.lementec, bruno.raffin)@imag.fr thierry.gautier@inrialpes.fr`

Abstract. Reordering instructions and data layout can bring significant performance improvement for memory bounded applications. Parallelizing such applications requires a careful design of the algorithm in order to keep the locality of the sequential execution. In this paper, we aim at finding a good parallelization of memory bounded applications on multicore that preserves the advantage of a shared cache. We focus on sequential applications with iteration through a sequence of memory references. Our solution relies on an adaptive parallel algorithm with a dynamic sliding window that constrains cores sharing the same cache to process data close in memory. This parallel algorithm induces the same number of cache misses as the sequential algorithm at the expense of an increased number of synchronizations. Experiments with a memory bounded application confirm that core collaboration for shared cache access can bring significant performance improvements despite the incurred synchronization costs. On quad cores Nehalem processor, our algorithms are 10% to 30% faster than algorithms not optimized for shared cache thanks to a reduced number of last level cache misses.

1 Introduction

Many applications in scientific computing are memory bounded. Favoring the locality of access patterns through data and computation reordering can bring significant performance benefits. When designing parallel algorithms, one must be extra careful not to lose the locality of the sequential application, which is the key for good performance.

In most last generation multicores, the last level of cache is shared among all cores of the chip. For instance the Intel Nehalem, the AMD Phenom and Opteron (only for the quadcores and hexacores) and the IBM Power7 all have a shared L_3 cache. Recent GPU architectures also adopt this cache design: the L_1 cache of a NVIDIA Fermi streaming multiprocessor is shared among 32 cores.

In this paper, we focus on one specific aspect of the parallelization of memory bounded applications: how to adapt the scheduling to take advantage of the shared

* Part of this work was done while the second author was visiting the ArTeCS group of the University Complutense, Madrid, Spain.

caches of multicore processors. The goal is to propose a scheduling algorithm that improves performance by reducing cache misses, compared to parallel algorithms that do not take into account the shared cache amongst several cores. We propose to have cores working on independent but close (regarding the memory layout) data sets that can all fit in the shared cache. If a core needs a data that is not in its data set, there is a good chance it will find it in the data set loaded in the cache by one of its neighbors, thus saving cache misses. The algorithm behaves as if each core would benefit from a full-size private cache, at the price of a few extra synchronizations required to ensure a proper collaboration between cores.

This paper focuses on algorithms that take an input sequence to produce an output sequence of results. Such algorithms encompass many of the C++ Standard Template Library (STL) functions like `for_each` or `transform`. Moreover, many parallel libraries such as Intel TBB or the GNU STL parallel mode provide parallel implementations of the STL. Thus providing shared cache aware parallelizations of these algorithms can improve performance of many applications running on multicores.

We provide a cache constraint that parallel algorithms should respect to induce no more cache misses than the sequential algorithms. We present two new algorithms respecting this cache constraint and two implementations, one based on PThread and the other one based on work-stealing allowing efficient dynamic load balancing. We also implement those new algorithms with the parallel library TBB and the GNU parallel STL and compare them with our implementations on the `for_each` function.

The paper is organized as follows. In section 2, we present the cache constraint and the associated algorithms. In section 3, we detail the implementation of these two algorithms using the work-stealing based framework Kaapi. Finally, we introduce the application we use to benchmark our algorithms in section 4 and the experimental data in section 5 before the conclusions.

2 Scheduling for Efficient Shared Cache Usage

2.1 Review of Work-Stealing and Parallel Depth First Schedules

Work Stealing (WS) is a scheduling algorithm that is very efficient both in theory and in practice. It has been implemented in many languages and parallel libraries including Cilk [1] and TBB [2]. In WS, each processor manages its own list of tasks. When a processor becomes idle, it becomes a thief, randomly chooses another processor, the victim, and try to steal some work. For an efficient load balancing, the thief should choose a task that represents a big amount of work far in memory from the work of the victim. This reduces the number of steal operations and thus synchronization costs. Unfortunately, stealing such tasks may not be optimal if one takes into account the shared cache of recent multicores.

Contrary to WS, the Parallel Depth First (PDF) schedule of [3] tries to optimize shared cache usage. This schedule is based on the sequential order of execution, which is supposed to be cache-efficient. When several tasks are available, a processor will preferably execute the earliest task in the sequential order. The authors showed that a PDF schedule induces no more cache misses than the sequential execution when the parallel execution uses a slightly bigger cache. However, computing and maintaining such a schedule is costly in practice.

Informally, one could think of the PDF scheduler as a WS scheduler where the thieves would choose the closest task in the victim list inducing lots of steal operations. This is not as simple as all processors, not only a victim and its thief, should work on data close in memory. In addition to the steal close operation, another mechanism is needed to prevent processors to deviate from each other after the steal operation. The cache constraint we present in the next section serves exactly this purpose. The processing order we proposed is a trade-off between WS and PDF. Processors work on data just close enough in memory to fit in the shared cache. This way the parallel application should not make more cache misses than the sequential application. The number of synchronizations is better than PDF but not as good as WS. However, as the number of cache misses is reduced, the overall performance should be improved over WS.

2.2 Window Algorithms for Sequence Processing

We consider algorithms that take an input sequence $i_1, i_2, \ldots, i_n$ (different input elements can share some data) and a function *op* to be applied on all elements of the input producing an output sequence $o_1, o_2, \ldots, o_{n'}$. Notice that treating one element may produce a different number of elements in the output sequence. Most STL algorithms are variations over this model. The sequential algorithm processes the sequence in order from i_1 to i_n. We assume that the sequential algorithm already performs well with respect to temporal locality of data accesses. Data processed closely in the sequential execution are also close in memory. We focus on the case where all elements of the sequence can be processed in parallel.

We introduce two parallel algorithms to process such a sequence in parallel. These two algorithms are parameterized by m, the maximum distance between the threads. In the first one, denoted *static-window*, the sequence is first divided into n/m chunks of m contiguous elements. Then, each chunk is processed in parallel by the p processors sharing the same cache. Several strategies can be used to parallelize the processing of each chunk. The m elements could be statically partitioned into p groups of m/p elements, one per processor, or a work-stealing scheme can be used to dynamically balance the load. The second parallel algorithm, denoted *sliding-window*, is a relaxed version of the *static-window* algorithm. At the beginning of the algorithm, the first m elements of the sequence are ready and can be processed in any order. Each time the first element i_k not yet processed in the sequence is treated by a processor, it enables the element i_{k+m} at the end of a window of size m. These two algorithms will be compared with an algorithm denoted *no-window* that do not respect the cache constraint. All the elements of the sequence can be processed in any order. This algorithm induces more cache misses than the sequential algorithm and the window algorithms, but it requires fewer synchronizations.

2.3 Cache Performance of Window Algorithms

The re-use distance captures the temporal locality of a program [4]. Let consider a series of memory references $(x_k)_{k \geq 0}$. When a reference x_k access an element for the first time, the re-use distance of x_k is infinite. If the element has been previously accessed, $x_{k'} = x_k$ with $k' > k$, the re-use distance of $x_{k'}$ is equal to the number of distinct elements accessed between these two references x_k and $x_{k'}$. Let h_d denote the number of memory references with a re-use distance d. The number of cache misses of a fully associative LRU cache of size C is equal to $M_{\text{seq}} = \sum_{d=C+1}^{\infty} h_d$. We can extend this definition to sequence processing algorithms: if processing i_k and $i_{k'}$ uses similar data, the re-use distance is $k' - k$.

We consider now p processors sharing the same cache that process the sequence in parallel in distant places like the *no-window* algorithm. As we assumed the sequence has good temporal locality, elements far-away in the sequence use distinct data. In this case, the re-use distance is multiplied by p as to each access of one processor corresponds $p-1$ accesses of the others to distinct elements. Thus, the number of cache misses is $M_{\text{no-win}} = \sum_{d=C+1}^{\infty} h_{d/p} \approx \sum_{d=C/p+1}^{\infty} h_d$. The *no-window* algorithm induces as many cache misses as the sequential algorithm with a cache p times smaller. We now restrain the processors to work on elements at distance less than m like in the window algorithms. Let $r(m)$ be the maximum number of distinct memory references when processing $m-1$ consecutive elements of the input sequence. In the worst case, when processing element i_k, all elements $i_{k+1}, \ldots, i_{k+m-1}$ have already been processed accessing at most $r(m)$ additional distinct elements compared to the sequential order. Thus the re-use distance is increased by at most $r(m)$. The number of cache misses is $M_{\text{window}} \leq \sum_{d=C+1}^{\infty} h_{d-r(m)} = M_{\text{seq}} + \sum_{d=C+1-r(m)}^{C} h_d$. As we assumed the sequence has good temporal locality, $r(m)$ is small compared to m and h_d is small for large d. Therefore $\sum_{d=C+1-r(m)}^{C} h_d$ is small and the window algorithms induce approximately the same number of cache misses as the sequential algorithm.

2.4 PThread Parallelization of Window Algorithms

We present here the implementation of the *no-window* and *static-window* algorithms using PThreads. The PThread implementation allows a fine grain control on synchronizations with very little overhead.

For the *no-window* algorithm, the sequence is statically divided into p groups. Each group is assigned to one thread bound to one processor and all threads synchronize at the end of the computation. For the *static-window* algorithm, the sequence is first divided into chunks of size m. Then each chunk is statically divided into p groups and all threads synchronize at the end of each chunk before starting to compute the next one. Each synchronization is implemented with a `pthread_barrier`. Threads wait at the barrier and are released when all of them have reached the barrier. Although we expect the threads in the *static-window* algorithm to spend more time waiting for other threads to finish their work, the reduction of cache misses should compensate this extra synchronization cost. The *sliding-window* algorithm has not been implemented in PThread because it would require a very complex code. We present in the next section a work-stealing framework allowing to easily implement all these algorithms.

```
typedef struct {
  InputIterator        ibeg;
  InputIterator        iend;
  OutputIterator       obeg;
  size_t               osize;
} Work_t ;

void dowork(...) {
  complete_work:
    while (iend != ibeg) {
      kaapi_stealpoint(..., &splitter);
      for(i=0; i<grain; ++i, ++ibeg)
        op(ibeg, obeg, &osize);
      kaapi_preemptpoint(..., &reducer);
    }
    if ( kaapi_preempt_next_thief(...) )
      goto complete_work ;
} // no more work -> become a thief

void reducer(Work_t *victim, Work_t *thief) {
  memmove( victim->obeg, thief->obeg,
           thief->osize );
  victim->osize += thief->osize;
  victim->ibeg   = thief->ibeg;
  victim->iend   = thief->iend;
} // victim -> dowork / thief -> try to steal
```

```
void splitter( Work_t *victim, int count,
               kaapi_request_t* request ) {
  int i = 0;
  size_t size = victim->iend - victim->ibeg;
  size_t bloc = size / (1+count);
  InputIterator local_end = victim->iend;
  Work_t *thief;

  if (size < gain)
    return;
  while (count >0) {
    if (kaapi_request_ok(&request[i])) {
      thief->iend  = local_end;
      thief->ibeg  = local_end - bloc;
      thief->obeg  = intermediate_buffer;
      thief->osize = 0;
      local_end   -= bloc;
      kaapi_request_reply_ok(thief,
                             &request[i]);
      --count;
    }
    ++i;
  }
  victim->iend  = local_end;
} // victim and thieves -> dowork
```

Fig. 1. C implementation of the adaptive *no-window* algorithm using the KAAPI API.

3 Work-Stealing Window Algorithms with Kaapi

In this section, we present the low level API of KAAPI [5] and detail the implementation of the windows algorithms.

3.1 Kaapi Overview

KAAPI is a programming framework for parallel computing using work-stealing. At the initialization of a KAAPI program, the middleware creates and binds one thread on each processor of the machine. All non-idle threads process work by executing a sequential algorithm (`dowork` in fig. 1). All idle threads, the thieves, send work requests to randomly selected victims. To allow other threads to steal part of its work, a non-idle thread must regularly check if it received work requests using the function `kaapi_stealpoint`. At the reception of `count` work requests, a `splitter` is called and divides the work into `count+1` well-balanced pieces, one for each of the thieves and one for the victim.

When a previously stolen thread runs out of work, it can decide to preempt its thieves with the `kaapi_preempt_next_thief` call. For each thief, the victim merges part of the work processed by the thief using the `reducer` function and takes back the remaining work. The preemption can reduce the overhead of storing elements of the output sequence in an intermediate buffer when the final place of an output element is not known in advance. To allow preemption, each thread regularly checks for preemption requests using the function `kaapi_preemptpoint`.

To amortize the calls to the KAAPI library, each thread should process several units of work between these calls. This number is called the *grain* of the algorithm. In particular, a victim thread do not answer positively to a work request when it has less than *grain* units of work.

Compared to classical WS implementations, tasks (`Work_t`) are only created when a steal occurs which reduces the overhead of the parallel algorithm compared to the sequential one [6]. Moreover, the steal requests are treated by the victim and not by the thieves themselves. Although the victim has to stop working to process these requests, synchronization costs are reduced. Indeed, instead of using high-level synchronization functions (mutexes, etc.) or even costly atomic assembly instructions (compare and swap, etc.), the thieves and the victim can communicate by using standard memory writes followed by memory barriers, so no memory bus locking is required. Additionally, the `splitter` function knows the number `count` of thieves that are trying to steal work to the same victim. Therefore, it permits a better balance of the workload. This feature is unique to Kaapi when compared to other tools having a work-stealing scheduler.

3.2 Work-Stealing Algorithm for Standard (*no-window*) Processing

It is straightforward to implement the *no-window* algorithm using Kaapi. The work owned by a thread is described in a structure by four variables: `ibeg` and `iend` represents the range of elements to process in the input sequence, `obeg` is an iterator on the output sequence and `osize` is the number of elements written on the output. At the beginning of the computation, a unique thread possesses the whole work: `ibeg=0` and `iend=n`. Each thread processes its assigned elements in a loop. Code of Fig. 1 shows the main points of the actual implementation.

3.3 Work-Stealing Window Algorithms

The *static-window* algorithm is very similar to the *no-window* algorithm of the previous section. The first thread owning the total work has a specific status, it is the *master* of the window. Only the master thread has knowledge of the remaining work outside the m-size window. When all elements of a window have been processed, the master enables the processing of the new window by updating its input iterators `ibeg = iend` and `iend += m`. This way, when idle threads request work to the master thread, the stolen work is close in the input sequence. Moreover, all threads always work on elements at distance at most m.

The *sliding-window* algorithm is a little bit more complex. In addition to the previous iterators, the master also maintains `ilast` an iterator on the first element after the stolen work in the input sequence (see Fig. 2). When the master does not receive any work request, then `iend == ilast == ibeg+m`. When the master receives work requests, it can choose to give work on both sides of the stolen work. Distributing work in the interval `[ibeg,iend]` corresponds to the previous algorithm. The master thread can also choose to distribute work close to the end of the window, in the interval `[ilast,ibeg+m]`. We implemented several variants of the `splitter`. The `local_splitter` gives in priority work in the interval `[ibeg,iend]`. It favors processing elements at the beginning to fast-forward the window thus enabling new elements to be processed. The `distant_splitter` gives in priority work in the interval `[ilast,ibeg+m]`. By distributing work at the end of the window, it should reduce the number of preemptions. The last one, `balanced_splitter` try to give well-balanced amount of work to all thieves by dividing the union of both intervals into equal size pieces. No piece of work can contains elements on both sides of the window as the resulting work would not be an interval.

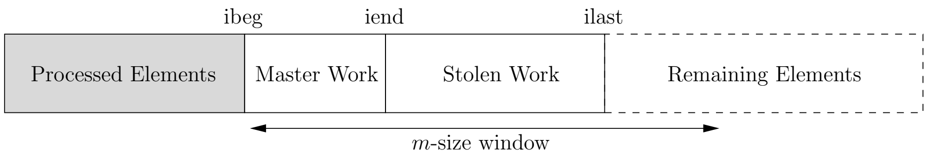


Fig. 2. Decomposition of the input sequence in the *sliding-window* algorithm.

4 Marching Tetrahedra for Isosurface Extraction

Isosurface extraction is one on the most classical filters of scientific visualization. It provides a way to understand the structure of a scalar field in a three dimensional mesh by visualizing surfaces of same scalar value. The marching tetrahedrons (MT) is an efficient algorithm for isosurface extraction [7]. For one cell of a mesh, the MT algorithm reads the point coordinates and scalar values and computes a linear approximation of the isosurface going through this cell. Applied on all mesh cells sequentially, it leads to a cost linear in the number of cells.

We now look at cache misses induced by MT. The mesh data structure usually consists of two multidimensional arrays: an array storing point attributes (e.g. coordinates, scalar values, etc.) and an array storing for each cell its points and attributes (e.g. type of the cell, scalar values, etc.). Points are accessed by following a reference from the cell array, e.g. reading coordinates of a point. As cells close in the cell array often use common points or points with close indices, processing cells in the same order as the sequential algorithm induces fewer cache misses when accessing the point array due to an improved temporal locality.

When implementing the window algorithms, the window size m should be chosen such that a sub-part of m cells of the mesh fits in the shared cache. Each point is coded on four doubles and each tetrahedron with four references (64bit integers) to points. On average, meshes have six times more tetrahedrons than points. So, for an 8MB cache, we approximately have $m = 225,000$. The same reasoning could apply to other mesh processing applications.

5 Experiments

We present experiments using the MT algorithm for isosurface extraction. We first calibrate the grain for the work-stealing implementation and the window size m for the window algorithms. Then, we compare the KAAPI framework with other parallel libraries on a central part of the MT algorithm which can be written as a `for_each`. Finally we compare the *no-window*, *static-window* and *sliding-window* algorithms implementing the whole MT.

All the measures reported are averaged over 20 runs and are very stable. The numbers of cache misses are obtained with PAPI [8]. Only last level cache misses are reported as the lower level cache misses are the same for all algorithms. Two different multicores are used, a quadcore Intel Xeon Nehalem E5540 at 2.4Ghz with a shared 8MB L_3 cache and a dualcore AMD Opteron 875 at 2.2Ghz with two 1MB L_2 private caches. If the window algorithms reduce the number of cache misses on the Nehalem but not on the Opteron, one can conclude that this is due to the shared cache.

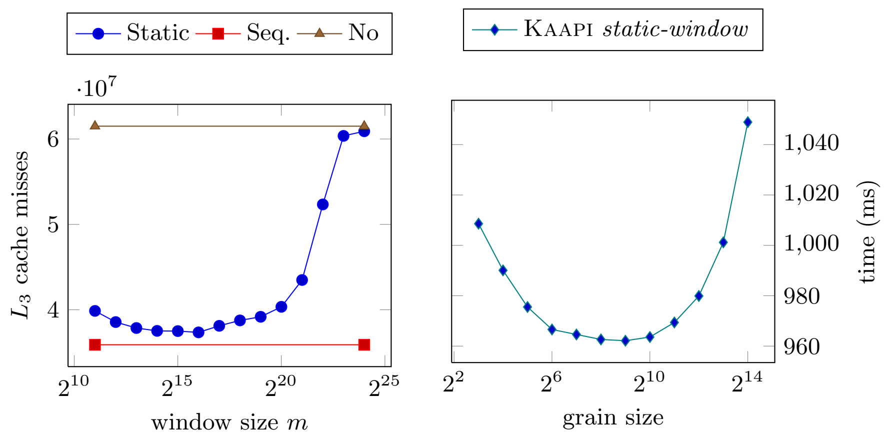


Fig. 3. (Left) Number of L_3 cache misses for the PThread implementation of the *static-window* algorithm —●— for various window sizes compared to the sequential algorithm —■— and the *no-window* —▲— algorithm. (Right) Parallel time for the KAAPI implementation of the *static-window* algorithm —◆— with various grain sizes. (Both) All parallel algorithms use the 4 cores of the Nehalem processor.

5.1 Calibrating the Window Algorithms

Fig. 3(left) shows the number of L_3 cache misses for the *static-window* algorithm compared to the sequential algorithm and the *no-window* algorithm. The *static-window* algorithm is very close to the sequential algorithm for window sizes less than 2^{20}. It does not exactly match the sequential performance due to additional `reduce` operations for managing the output sequence in parallel. With bigger windows, L_3 misses increase and tend to the *no-window* algorithm. For the remaining experiments, we set $m = 2^{19}$.

Fig. 3(right) shows the parallel time of the *static-window* algorithm with the KAAPI implementation for various grain sizes. Performance does not vary much, less than 10% on the tested grains. For small grains, the overhead of the KAAPI library becomes significant. For bigger grains, the load balancing is less efficient. For the remaining experiments, we choose a grain size of 128. We can notice that the KAAPI library allows very fine grain parallelism: processing 128 elements takes approximately 3μs on the Nehalem processor.

5.2 Comparison of Parallel Libraries on `for_each`

Table 1 compares KAAPI with the GNU parallel library (from gcc 4.3) (denoted GNU) and Intel TBB (v2.1) on a `for_each` used to implement a central sub-part of the MT algorithm. The GNU parallel library uses the best scheduler (parallel balanced). TBB uses the auto partitioner with a grain size of 128. TBB is faster than GNU on Nehalem and it is the other way around on Opteron. KAAPI shows the best performance on both processors. This can be explained by the cost of the synchronization primitives used: POSIX locks for GNU, compare and swap for TBB and atomic writes followed by memory barriers for KAAPI.

Time (ms)		Nehalem				Opteron			
Algorithms	#Cores	STL	GNU	TBB	KAAPI	STL	GNU	TBB	KAAPI
no-window	1	3,987	4,095	3,975	4,013	9,352	9,154	10,514	9,400
	4		1,158	1,106	1,069		2,514	2,680	2,431
static-window	1	3,990	4,098	3,981	4,016	9,353	9,208	10,271	9,411
	4		1,033	966	937		2,613	2,776	2,598

Table 1. Performance of the *no-window* and *static-window* algorithms on a `for_each` with various parallel libraries. GNU is the GNU parallel library. Time are in ms.

5.3 Performance of the Window Algorithms

We now compare the performance of the window algorithms. Table 1 shows that the *static-window* algorithm improves over the *no-window* algorithm for all libraries on the Nehalem processor. However, on the Opteron with only private caches, performances are in favor of the *no-window* algorithm. This was expected as the Opteron has only private caches and the *no-window* algorithm has less synchronizations. We can conclude that the difference observed on Nehalem is indeed due to the shared cache.

Fig. 4(left) presents speedup of all algorithms and ratio of cache misses compared to the sequential algorithm. The *no-window* versions induces 50% more cache misses whereas the window versions only 13% more. The window versions are all faster compared to the *no-window* versions. Work stealing implementations with KAAPI improves over the static partitioning of the PThread implementations. The *sliding-window* (with the best splitter: `balanced_splitter`) shows the best performance.

Fig. 4(right) focus on the comparison of the *sliding-window* and *static-window* algorithms. Due to additional parallelism, the number of steal operations are greatly reduced in the *sliding-window* algorithm (up to 2.5 time less for bigger windows) leading to an additional gain around 5%.

6 Related works

Previous experimental approaches have shown the interest of efficient cache sharing usage, on a recent benchmark in [9] and on data mining applications in [10]. In this paper, we go beyond those specific approaches by providing general algorithms for independent tasks parallelism which respect the sequential locality.

Many parallel schemes have been proposed to achieve good load balancing for isosurface extraction [11]. However, none of these techniques take into account the number of cache misses and the shared cache of multicore processors. Optimization of sequential locality for mesh applications has been studied through mesh layout optimization in [12].

7 Conclusions

This paper focuses on exploiting the shared cache of last generation multicores. We presented new algorithms to parallelize STL-like sequence processing. Experiments on several parallel libraries confirm that these techniques increase performance from 10% to 30% thanks to a reduced number of last level cache misses.

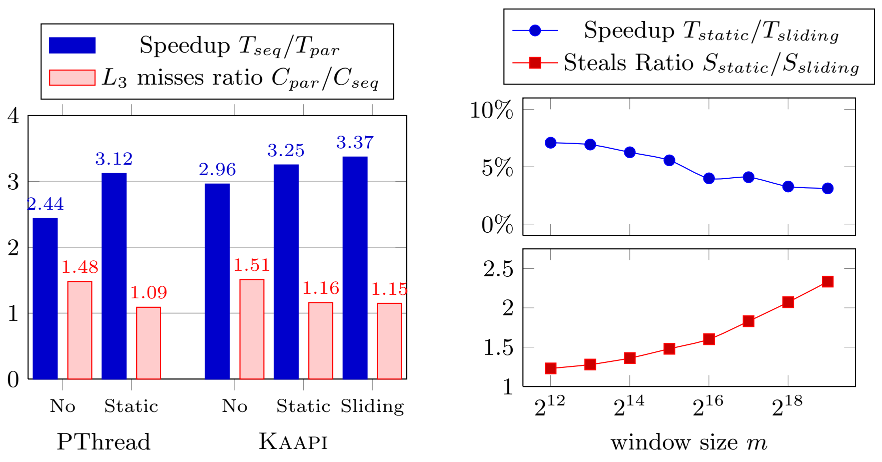


Fig. 4. (Left) Speedup ▬ and ratio of increased cache misses ▭ over the sequential algorithm for the *no-window*, *static-window* and *sliding-window* algorithms with PThread and Kaapi implementations. (Right) Speedup –●– and ratio of saved steal operations –■– for the *sliding-window* algorithm over the *static-window* algorithm with the Kaapi implementation. (Both) All algorithms run on the 4 cores of the Nehalem.

References

1. Blumofe, R., Joerg, C., Kuszmaul, B., Leiserson, C., Randall, K., Zhou, Y.: Cilk: An efficient multithreaded runtime system. Journal of Parallel and Distributed Computing **37**(1) (1996) 55–69
2. Robison, A., Voss, M., Kukanov, A.: Optimization via reflection on work stealing in TBB. In: IPDPS. (2008)
3. Blelloch, G.E., Gibbons, P.B.: Effectively sharing a cache among threads. In: SPAA. (2004)
4. Cascaval, C., Padua, D.A.: Estimating cache misses and locality using stack distances. In: Proc. of ICS. (2003)
5. Gautier, T., Besseron, X., Pigeon, L.: KAAPI: A thread scheduling runtime system for data flow computations on cluster of multi-processors. In: PASCO. (2007)
6. Traoré, D., Roch, J.L., Maillard, N., Gautier, T., Bernard, J.: Deque-free work-optimal parallel stl algorithms. In: Euro-Par. (2008)
7. Schroeder, W., Martin, K., Lorensen, B.: The Visualization Toolkit, An Object-Oriented Approach To 3D Graphics, 3rd ed. Kitware Inc. (2004)
8. Browne, S., Dongarra, J., Garner, N., Ho, G., Mucci, P.: A portable programming interface for performance evaluation on modern processors. The International Journal of High Performance Computing Applications **14** (2000)
9. Zhang, E.Z., Jiang, Y., Shen, X.: Does cache sharing on modern CMP matter to the performance of contemporary multithreaded programs? In: PPoPP. (2010)
10. Jaleel, A., Mattina, M., Jacob, B.: Last level cache (LLC) performance of data mining workloads on a CMP - a case study of parallel bioinformatics workloads. In: HPCA. (2006)
11. Zhang, H., Newman, T.S., Zhang, X.: Case study of multithreaded in-core isosurface extraction algorithms. In: EGPGV. (2004)
12. Tchiboukdjian, M., Danjean, V., Raffin, B.: Binary mesh partitioning for cache-efficient visualization. Visualization and Computer Graphics, IEEE Transactions on **16**(5) (sept.-oct. 2010) 815 –828

Resource-agnostic programming for many-core microgrids [1]

T.A.M. Bernard, C. Grelck, M.A. Hicks, C.R. Jesshope, R. Poss

University of Amsterdam, Informatics Institute, Netherlands
{t.bernard,c.grelck,m.a.hicks,c.r.jesshope,r.c.poss}@uva.nl

Abstract. Many-core architectures are a commercial reality, but programming them efficiently is still a challenge, especially if the mix is heterogeneous. Here granularity must be addressed, i.e. when to make use of concurrency resources and when not to. We have designed a data-driven, fine-grained concurrent execution model (SVP) that captures concurrency in a resource-agnostic way. Our approach separates the concern of describing a concurrent computation from its mapping and scheduling. We have implemented this model as a novel many-core architecture programmed with a language called μTC. In this paper we demonstrate how we achieve our goal of resource-agnostic programming on this target, where heterogeneity is exposed as arbitrarily sized clusters of cores.

Keywords: Concurrent execution model, many core architecture, resource-agnostic parallel programming.

1 Introduction

Many-core architectures provide the only solution to the various barriers opposing advances in mainstream computing performance [8]. However, programming applications on such platforms is still notoriously difficult [6,1,7]. Concurrency must be exposed, and in most programming paradigms it must be also explicitly managed [11]. For example, low-level constructs must be carefully assembled to map computations to hardware threads and achieve the desired synchronisation without introducing deadlocks, livelocks, race conditions, etc. From a performance perspective, any overhead associated with concurrency creation and synchronisation must be amortised with a computation of a sufficient granularity. The difficulty of the latter is under-estimated and in this paper we argue that this mapping task is too ill-defined statically and too complex to remain the programmer's responsibility. With widely varying resource characteristics, generality is normally discarded in favour of performance on a given target, requiring a full development cycle each time the concurrency granularity evolves.

[1] This work is supported by the European Union through the Apple-CORE project, grant no. FP7-ICT-215216.

We have addressed these issues in our work on *SVP* (for Self-adaptive Virtual Processor), which combines fine-grained threads with both barrier and dataflow synchronisation. Concurrency is created hierarchically and dependencies are captured explicitly. Hierarchical composition aims to capture concurrency at all granularities, without the need to explicitly *manage* it. Threads are not mapped to processing resources until run-time and the concurrency exploited depends only on the resources made available dynamically. Dependencies are captured using dataflow synchronisers and threads are only scheduled for execution when they have data to proceed. In this way, we automate thread scheduling and support asynchrony in operations. More detail on the model can be found in [3].

Asynchrony is exposed at the function level by delegating a unit of computation to independent processing resources where it can execute concurrently with its parent. It is also exposed in the dependencies captured between threads. In the context of this paper, where the model is implemented in a processor's ISA [5], we have efficient concurrency creation and synchronisation, requiring just a few processor cycles to distribute an arbitrary number of identical, indexed threads to a cluster of cores. Moreover, asynchronous operations are supported at a granularity of individual instructions and we can therefore tolerate latency in long-latency operations, such as loads from a distributed shared memory. The mapping of threads to a cluster of cores in our *Microgrid* chip architecture is automatic, and the compiled code may also express more concurrency than is available in a cluster. To resolve this mismatch, cores automatically switch from space scheduling to time scheduling when all hardware thread slots are in use. Hence, the minimal resource requirement for any SVP program is a single thread slot on a single core, which implies pure sequential execution, even though the code is expressed concurrently. It is through this technique and the latency tolerance that we achieve resource-agnostic code with predictable performance.

The main contribution of this paper is that we show simply implemented, resource agnostic SVP programs adapt automatically to the concurrency effectively available in hardware and can achieve extremely high execution efficiency. We also show that we can predict the performance of these programs based on simple throughput calculations even in the presence of non-deterministic instruction execution times. This demonstrates the effectiveness of the self-scheduling supported by SVP. In other words, we promote our research goal:

"Implement once, compile once, run anywhere."

2 The SVP concurrency model

We have built an implementation of SVP into a system language μTC and a compiler that maps this code to the Microgrid implementation. μTC is not intended as an end-user language; work is ongoing to target μTC from a data-parallel functional language (SaC [10]) and a parallelising C compiler [14,9].

In SVP programs *create* multiple threads at once as statically homogeneous, but dynamically heterogeneous *families*. The parent thread can then perform a barrier wait on termination of a named family using a *sync* action. This fork-join pattern captures concurrency hierarchically, from software component composition down to inner loops. A family is characterised by its index sequence, the initial PC for threads and the definition of unidirectional dataflow channels from, to and within the family. Channels are I-structures [2], i.e. blocking reads and single non-blocking writes; either from parent to all children ("*globals*") or sideways in the family ("*shareds*"). For more details see [5].

In the Microgrid implementation, the number of active threads per core is constrained by a block size specified for each family or by exhaustion of thread contexts. Additional expressed concurrency is then scheduled by reusing thread contexts non-preemptively. Deadlock freedom is guaranteed by restricting communication to forward-only dependency chains [17].

A key characteristic of SVP is the separation of concerns between the program and its scheduling onto computing nodes. Space scheduling is achieved by binding a collection of computing nodes, called a *place*, to a family upon its creation. This can happen at any level in the hierarchy, dynamically. Although in principle, SVP can be implemented at any level of granularity, we focus in this paper on the finest granularity, where clusters of cores implement an SVP run-time system in hardware. The SVP *create* distributes families equally to all cores in a cluster or locally depending on the place specifier. Clusters of cores are connected in rings and may be configured either at design-time or run-time.

On the Microgrid, SVP channels are mapped onto the cores' registers. Dependencies between threads mapped to the same core share the same physical registers to allow fast communication and when distributed between cores, communication is induced automatically upon register access. The latter is still a low-latency operation since constraints on dependency patterns ensure that communicating cores are adjacent on chip. Implementing I-structures on the registers also enforces scheduling dependencies between consumers and producers. Hence, long-latency operations may be allowed to complete asynchronously giving out-of-order completion with non-deterministic delay. Examples include memory operations, floating point operations (with FPU sharing between cores) and family synchronisation. This mechanism, together with support for a large number of threads per core provides the latency tolerance necessary to achieve a high utilisation of the cores' pipeline cycles. More information is available in [5].

3 An SVP implementation

The Microgrid evaluated in this paper comprises 128 cores sharing 64 FPUs with separate *add*, *mul*, *div* and *sqrt* pipelines. Each core supports up to 256 threads in 16 families using up to 1024 integer and 512 floating-point registers. On-chip memory comprises a modest 32×32KB L2 caches, shared in groups of 4 cores. There are 4 rings of 8 L2 caches; the 4 directories are connected in a top-level ring subordinated to a master directory. Two DDR3-1600 channels connect the master directory to external storage. The on-chip memory network implements a Cache-Only Memory Architecture (COMA) protocol with synchronisation at family creation, termination and on communication between threads. A cache line has no home location and migrates to the point of most recent use. This is described in more detail in [18].

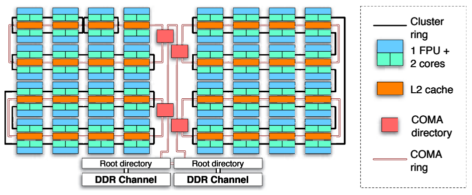


Fig. 1. Functional diagram of a 128 core Microgrid.

The following parameters are relevant to the numerical results: the two DDR channels provide 1600 million 64-bit transfers/s, i.e. a peak bandwidth of 25.6GB/s overall; each COMA ring provides a total bandwidth of 64GB/s, shared among its participants; the bus between cores and L2 caches provides 64GB/s of bandwidth; the SVP cores are clocked at 1GHz.

The Microgrid runs a minimal operating system. This includes initialisation, collection of system metrics, heap allocation, input of data from the environment through memory, and text output. A software *SVP place allocation* service allows to select dynamic cluster sizes, to subject benchmarks to heterogeneous concurrency parameters. We highlight that compiled program code is independent from all the architectural parameters of the Microgrid.

4 Experiments and results

Our aim in this paper is to show how we can obtain deterministic performance figures, even though the code is compiled from naive μTC code, with no knowledge of the target. We evaluate results from executing a range of benchmarks across a range of problem sizes on clusters of size 1-64 cores. These include both sequential and parallel algorithms with various data access patterns. The results are presented with performance on cold and warm caches. In order to analyse the performance, we need to understand the constraints on performance. For this we define two measures of arithmetic intensity (AI). The first AI_1 is the ratio of floating point operations to instructions issued. For a given kernel that is not I/O bound, this limits the floating point performance. For P cores at 1 GHz, the peak performance we can expect therefore is $P \times AI_1$. In some circumstances, we know that execution is constrained by dependencies between floating point operations and here we modify AI_1 to take this into account giving an effective intensity AI'_1. The second measure of arithmetic intensity is the ratio of Floating point operations to I/O operations, AI_2 FLOPs/Byte. I/O bandwidth IO is usually measured at the chip boundary (25.6GB/s) unless we can identify bottlenecks internally on the COMA rings (64GB/s). As these I/O bandwidths are independent of the number of cores used, this measure will provide a hard performance limit when $P \times AI_1 \geq AI_2 \times IO$.

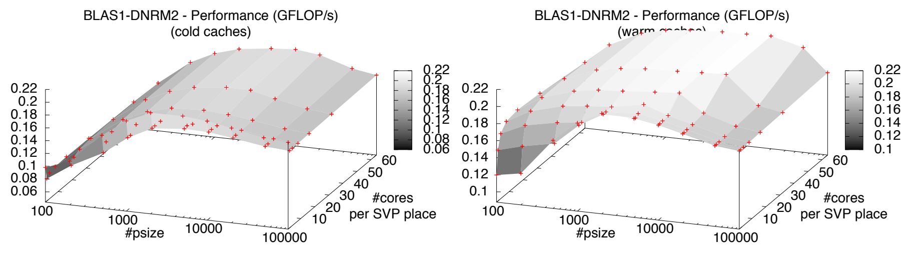
The results presented in this paper are produced using cycle-accurate emulation of a Microgrid chip that implements SVP in the ISA. It assumes custom silicon with current technology [5]. It defines all states that would exist in a silicon implementation and captures cycle-by-cycle interactions in all pipeline stages. We have used realistic multi-ported memory structures, with queueing and arbitration where we have more channels than ports. The timing assumptions are based on evaluation using CACTI [16]. We also simulate the timing of standard DDR3 channels. As details of the architecture have been described elsewhere we include only sufficient detail here to support the discussion.

4.1 Sequential code

The first kernel we consider is DNRM2 from the BLAS library, which computes the Euclidean norm of a vector. Here we do not parallelise the loop, which uses a carried dependency to calculate the sum. We are interested in how well the Microgrid tolerates the memory latency of hundreds of cycles. Branch prediction and out-of-order instruction issue can provide some latency tolerance, typically tens of cycles, which is sufficient to optimise performance when working from on-chip cache but not for larger data sets. Prefetching can do better on constant-stride accesses but as memory latencies rise, the probability that prefetched data will remain in cache diminishes. In our approach, the hardware provides latency hiding through interleaving multiple threads in the pipeline. In this kernel, a memory load and a *mul* form an independent prefix to the dependent *add* which computes the sum using a shared variable.

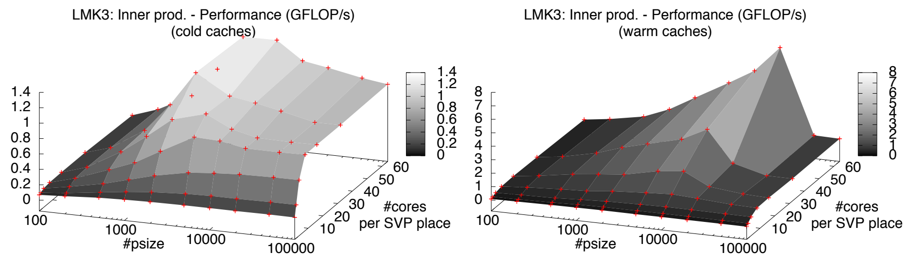
The thread code compiles to 4 instructions of which two are FP operations. So $AI_1 = 0.5$. However, every thread must wait for its predecessor to produce its result before computing its FP add. The cost of communicating the result from thread to thread requires between 6 and 11 cycles per add depending on the scheduling of threads, with the difference representing the cost of waking up a waiting thread and getting it to the read stage of the pipeline, which may be overlapped by other independent instructions in the pipeline. This implies $0.14 \leq AI'_1 \leq 0.22$, i.e. an expected single core performance of 0.14 to 0.22 GFLOP/s. As Figure 2 shows, provided we have enough threads we observe just under 0.20 GFLOP/s on one core.

We do not expect to see any performance increase by increasing the number of cores, because the independent prefix instructions that can be scheduled independently represent less than one third of the cycles required by the thread, i.e. $3 \div AI'_1$. Even with ideal scheduling and no overhead, Amdahl's law would limit speedup to a factor 1.5. The fact that we see a 10% increase is testament to the low overhead in this architecture of managing concurrency and communication.



(a) Cold caches (b) Warm caches

Fig. 2. Performance of DNRM2 on one SVP place. Working set: 8×#psize bytes.



(a) Cold caches (b) Warm caches

Fig. 3. IP performance, using N/P reduction. Working set: 16×#psize bytes.

4.2 Reductions

Any reduction can be parallelised for commutative and associative operations. The second benchmark is parallelised inner product (IP, Livermore kernel 3). The code is a straightforward extension of the naive implementation in μTC. It relies on the number of cores in the 'current place' being exposed to programs as a language primitive and splits the reduction into two stages, the first creates a family of one thread per core, which performs a local reduction and then completes the reduction between cores. When the number of threads per core is significantly larger than the number of cores, the cost of the final reduction is small and the performance should scale linearly with the number of cores. Figure 3 shows the experimental results for this code.

For IP, $AI_1 = 0.29$; however, again we must consider the effective intensity: $0.12 \leq AI'_1 \leq 0.17$, i.e. an expected single core performance of 0.12 to 0.17 GFLOP/s. The outer loop is parallel and hence we would expect a maximum performance of 0.15×64 or 9.6 GFLOP/s. However, for this code $AI_2 = 0.125$ FLOPs/byte and so performance would be memory limited to 3.2 GFLOP/s.

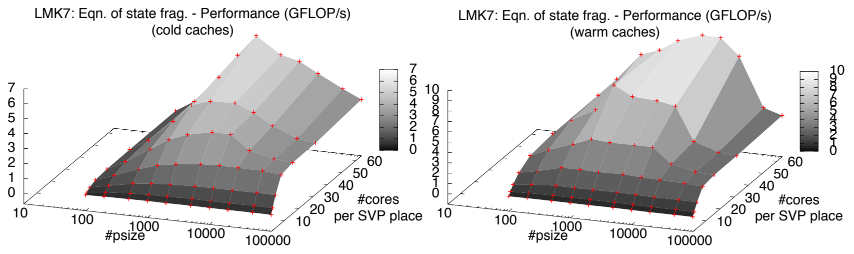
We achieve only 1.4 GFLOP/s, dropping to 0.88 GFLOP/s, for cold caches with the largest problem size. This deviation occurs when the working set does not fit in the L2 caches, because then loads to memory must be interleaved with line evictions. Even though evictions do not require I/O bandwidth, they do consume COMA ring bandwidth. It is more difficult to reason about ring bandwidth under such circumstances. In the worst case a single load may evict a cache line where the loaded line is used only by one thread before being evicted again. A single 8 byte load could require as much as two 64-byte line transfers, i.e. a perceived bandwidth for loads of 4 GB/s rising to 32GB/s if all 8 words are used. This translates into a peak performance of between 0.5 and 4 GFLOP/s with $AI_2 = 0.125$ FLOPs/Byte, when the caches become full. Note also, at a problem size of 20K on 64 cores, between 17 and 22% of the cycles required are for the sequential reduction, a large overhead and at a problem size of 100K, when this overhead is significantly smaller, only 1/6th of the problem fits in cache for up to 32 cores (1/3 for 64 cores).

With warm caches, this transition to on-chip bandwidth limited performance is delayed and more abrupt. For $P = 32$ the maximum in-cache problem size is N=16K and for $P = 64$, N=32K (ignoring code etc.). As would be expected for ring-limited performance, we see peak performance at N=10K and 20K resp. for these two cases. Any increase in problem size beyond this increases ring bandwidth to the same level as with cold caches.

4.3 Data-parallel code

We show here the behaviour of three data-parallel algorithms which exhibit different, yet typical communication patterns. Again, our μTC code is a straightforward parallelisation of the obvious sequential implementation and do not attempt any explicit mapping to hardware resources. The equation of state fragment (ESF, Livermore kernel 7) is a data parallel kernel with a high arithmetic intensity, $AI_1 = 0.48$. It has 7 local accesses to the same array data by different threads. If this locality can be exploited, then $AI_2 = 0.5$ FLOPs/Byte from off-chip memory. Matrix-matrix product (MM, Livermore kernel 21) has significant non-local access to data, in that every result is a combination of all input data. MM is based on multiple inner products and hence $AI_1 = 0.29$. However, for cache bound problems and best case for problems that exceed the cache size, $AI_2 = 3$ FLOPs/Byte from off-chip memory. Finally, FFT lies somewhere between these two extremes: it has a logarithmic number of stages that can exploit reuse but has poor locality of access. Here $AI_1 = 0.33$ and for cache-bound problems $1.6 \leq AI_2 \leq 2.9$ (logarithmic growth with problem size if there are no evictions). However, with evictions this is defined per FFT stage and $AI_2 = 0.21$.

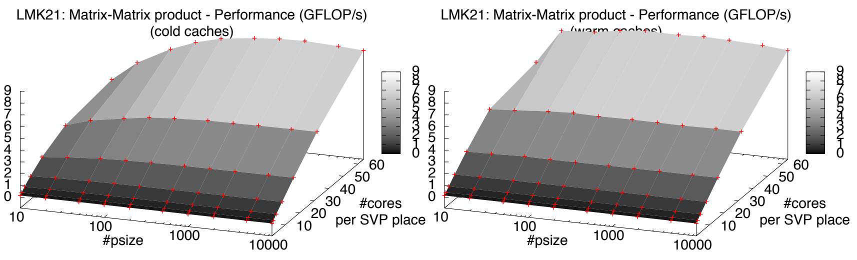
For ESF, with sufficient threads, the observed single core performance is 0.43 GFLOP/s, i.e. 90% of the expected maximum based on AI_1 for this problem (see Figure 4a). Also, while the problem is cache bound, for cold caches, we see linear speedup on up to 8 cores, 3.8 GFLOP/s. For 8 cores this problem size has 128 threads per core, reducing to 8 at 64 cores. This is an insufficient number of threads to tolerate latency and we obtain 6.6 GFLOP/s for 64 cores, 54% of the maximum limited by AI_2 (12.3 GFLOP/s). As the problem size is increased, cache evictions limit effective I/O bandwidth to 12.3GB/s at the largest problem sizes, i.e. an AI_2 constraint of around 6 GFLOP/s. We see saturation at 67% of this limit for both warm and cold caches. With warm caches and smaller



(a) Cold caches (b) Warm caches

Fig. 4. Performance of the ESF. Working set: 32×#psize bytes.

problem sizes, greater speedups can be achieved (see Figure 4b) and we achieve 9.87 GFLOP/s or 80% of the AI_2 constrained limit for a cache bound problem.



(a) Performance on cold caches (b) Performance on warm caches

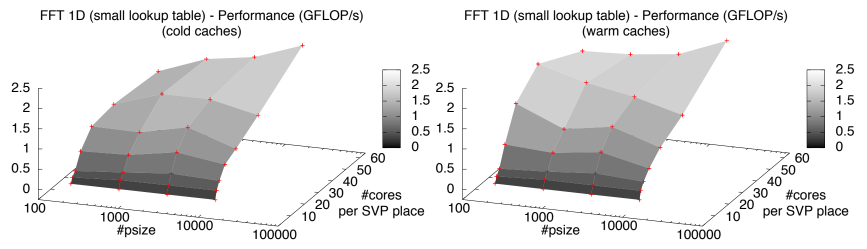
Fig. 5. Performance of the matrix-matrix product. Working set: ≈ 200×#psize bytes.

MM naively multiplies 25×25 matrices by 25×N matrices using a local IP algorithm. As $AI_2 = 3.1$ FLOPs/Byte, the I/O limit of 75 GFLOP/s exceeds the theoretical peak performance, namely 18.3 GFLOP/s. Our experiments show an actual peak of 8.57 GFLOP/s, or 47% of the maximum. As there are sufficient threads, we suspect the limit is on the COMA ring, as a significant amount of traffic is required to distribute rows and columns to cores.

For FFT, the observed performance (cf. Figure 6) on one core is 0.23 GFLOP/s, or 78% of the AI_1 limit. When the number of cores and the problem size increase, the program becomes AI_2 constrained, as now every stage will require loads and evictions, giving an effective bandwidth of 12.3GB/s and as $AI_2 = 0.21$, an I/O constrained limit of 2.6 GFLOP/s. We observe 2.24 GFLOP/s, or 86% of this.

5 Related work

SVP addresses many-core programming from hardware thread contexts up to the programming model. In this vertical approach, it relates to XMT [13].



(a) Cold caches (b) Warm caches

Fig. 6. Performance of the 1-D FFT. Working set: 8×#psize bytes + a lookup table.

However, the ability to define concurrency hierarchically and its data-driven scheduling bring it closer to Cilk [4] and the DDM architecture [12]. SVP differs from DDM mainly in that synchronisation is implemented in registers instead of cache, and that yet unsatisfied dependencies cause threads to suspend. Register-based synchronisation can also be found in the WaveScalar architecture [15], but WaveScalar requires pure dataflow program expression while SVP also allows thread-local sequential schedules using a regular RISC ISA.

6 Conclusion

The results presented in the previous section show efficient use of the hardware resources of single SVP places by naive implementations of computation kernels. We are able to analyse performance based on two bandwidth constrained measures and provided we have sufficient threads we observe performances that are very close (in the region of 80%) of the observed performance. Even in the worst cases we are within 50% of these predicted performances.

In conclusion, the SVP concurrency model facilitates the writing and generation of concurrent programs that need only be written and compiled once but yet can still exploit the varying parallel resources provided by particular hardware configurations. Programs can thus be expressed in the μTC language free from the restraints of resource awareness; the program only needs to express the available concurrency in algorithms and the desired synchronisations.

Acknowledgements

The development of SVP, the Microgrid architecture and the μTC compiler was initially supported by the NWO *Microgrids* project, then by the EU *Apple-CORE* project. SVP and its implementation is a group effort of the CSA group at the University of Amsterdam. The authors would like to thank especially Mike Lankamp for his work on emulation.

References

1. Amarasinghe, S.: (How) can Programmers Conquer the Multicore Menace? In: PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques. pp. 133–133. ACM, New York, NY, USA (2008)
2. Arvind, Nikhil, S., R., Pingali, K.K.: I-Structures: Data Structures for Parallel Computing. ACM Trans. Program. Lang. Syst. 11(4), 598–632 (1989)
3. Bernard, T., Bousias, K., Guang, L., Jesshope, C.R., Lankamp, M., van Tol, M.W., Zhang, L.: A General Model of Concurrency and its Implementation as Many-core Dynamic RISC Processors. In: Proc. Intl. Conf. on Embedded Computer Systems: Architecture, Modeling and Simulation, SAMOS-2008. pp. 1–9 (2008)
4. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., et al.: Cilk: an efficient multithreaded runtime system. SIGPLAN Not. 30(8), 207–216 (1995)
5. Bousias, K., Guang, L., Jesshope, C., Lankamp, M.: Implementation and Evaluation of a Microthread Architecture. J. Systems Architecture 55(3), 149–161 (2009)
6. Chapman, B.M.: The Multicore Programming Challenge. In: Advanced Parallel Processing Technologies. p. 3 (2007)
7. Gabb, H., Mattson, T., Breshears, C.: Thinking in Parallel - Three engineers' Viewpoints. Intel Software Insight Magazine 16, 24–26 (Feb 2009)
8. Geer, D.: Industry Trends: Chip Makers Turn to Multicore Processors. Computer 38(5), 11–13 (2005)
9. Grelck, C., Herhut, S., Jesshope, C., Joslin, C., Lankamp, M., Scholz, S.B., Shafarenko, A.: Compiling the Functional Data-Parallel Language SaC for Microgrids of Self-Adaptive Virtual Processors. In: 14th Workshop on Compilers for Parallel Computers (CPC'09), Zürich, Switzerland (2009)
10. Grelck, C., Scholz, S.B.: SAC: a functional array language for efficient multithreaded execution. Int. Journal of Parallel Programming 34(4), 383–427 (2006)
11. Kasim, H., March, V., Zhang, R., See, S.: Survey on Parallel Programming Model. In: Network and Parallel Computing. LNCS, vol. 5245, pp. 266–275. Springer (2008)
12. Kyriacou, C., Evripidou, P., Trancoso, P.: Data-driven multithreading using conventional microprocessors. IEEE Trans. Parallel Distrib. Syst. 17(10), 1176–1188 (2006)
13. Naishlos, D., Nuzman, J., Tseng, C.W., Vishkin, U.: Towards a first vertical prototyping of an extremely fine-grained parallel programming approach. In: SPAA '01: Proc. 13th annual ACM symposium on Parallel algorithms and architectures. pp. 93–102. ACM, New York, NY, USA (2001)
14. Saougkos, D., Evgenidou, D., Manis, G.: Specifying loop transformations for C2μTC source-to-source compiler. In: 14th Workshop on Compilers for Parallel Computers (Jan 2009)
15. Swanson, S., Schwerin, A., Mercaldi, M., Petersen, A., Putnam, A., Michelson, K., Oskin, M., Eggers, S.J.: The WaveScalar Architecture. ACM Trans. Comput. Syst. 25(2), 4 (2007)
16. Tarjan, D., Thoziyoor, S., Jouppi, N.: Cacti 4.0. Tech. rep., Western Research Laboratory, Compaq (2006)
17. Vu, T.D., Jesshope, C.R.: Formalizing SANE Virtual Processor in Thread Algebra. In: ICFEM. pp. 345–365 (2007)
18. Zhang, L., Jesshope, C.R.: On-Chip COMA Cache-Coherence Protocol for Microgrids of Microthreaded Cores. In: Bouge, et al. (eds.) Euro-Par Workshops. LNCS, vol. 4854, pp. 38–48. Springer (2007)

Programming Heterogeneous Multicore Systems using Threading Building Blocks[⋆]

George Russell[1], Paul Keir[2], Alastair F. Donaldson[3], Uwe Dolinsky[1], Andrew Richards[1] and Colin Riley[1]

[1] Codeplay Software Ltd., Edinburgh, UK
{uwe,andrew,george,colin}@codeplay.com
[2] Department of Computing Science, University of Glasgow, UK
pkeir@dcs.gla.ac.uk
[3] Oxford University Computing Laboratory, Oxford, UK
alastair.donaldson@comlab.ox.ac.uk,

Abstract. Intel's Threading Building Blocks (TBB) provide a high-level abstraction for expressing parallelism in applications without writing explicitly multi-threaded code. However, TBB is only available for shared-memory, homogeneous multicore processors. Codeplay's Offload C++ provides a single-source, POSIX threads-like approach to programming *heterogeneous* multicore devices where cores are equipped with private, local memories—code to move data between memory spaces is generated *automatically*. In this paper, we show that the strengths of TBB and Offload C++ can be combined, by implementing part of the TBB headers in Offload C++. This allows applications parallelised using TBB to run, without source-level modifications, across all the cores of the Cell BE processor. We present experimental results applying our method to a set of TBB programs. To our knowledge, this work marks the first demonstration of programs parallelised using TBB executing on a heterogeneous multicore architecture.

1 Introduction

Concurrent programming of multicore systems is widely acknowledged to be challenging. Our analysis is that a significant proportion of the challenge is due to the following phenomena:

Thread management: It is difficult to explicitly manage thread start-up and clear-down, inter-thread synchronization, mutual exclusion, work distribution and load balancing over a suitable number of threads to achieve scalability and performance.

Heterogeneity: Modern multicore systems, such as the Cell [1], or multicore PCs equipped with graphics processing units (GPUs) consist of cores with differing instruction sets, and contain multiple, non-coherent memory spaces. These heterogeneous features can facilitate high-performance, but require writing duplicate code for different types of cores, and orchestration of data-movement between memory spaces.

Threading Building Blocks (TBB) [2] is a multi-platform library for programming homogeneous, shared memory multicore processors in C++ using constructs such as parallel loop and reduction operations, pipelines, and tasks, that capture the parallelism

[⋆] This work was supported in part by the EU FP7 STREP project PEPPHER, and by EPSRC grant EP/G051100/1

inherent in large classes of applications. These constructs allow the programmer to specify what can be safely executed in parallel, with parallelisation coordinated behind-the-scenes in the library implementation, thus addressing the *thread management* issues identified above.

Offload C++ [3, 4] extends C++ to address *heterogeneity*. Essentially, Offload C++ provides single source, thread based programming of heterogeneous architectures consisting of a host plus accelerators. Thread management must be handled explicitly, but the burden of code duplication and movement of data between memory spaces is handled automatically by the compiler and runtime system. Offload C++ for the Cell processor under Linux is freely available [5].

In this paper, we combine the strengths of TBB and Offload C++ by using Offload C++ to implement an important part of TBB: the *parallel for* construct. This allows applications that use these constructs to run, *without source-level modifications*, across *all* cores of the Cell BE architecture.

We also discuss data-movement optimisations for Offload C++, and describe the design of a portable template-library for bulk data-transfers. We show that this template-library can be integrated with TBB applications, providing optimized performance when Offload C++ is used on Cell, and default performance otherwise. We evaluate our approach experimentally using a range of benchmark applications. In summary, we make the following contributions:

- We describe how an important fragment of TBB implemented using Offload C++ allows a large class of programs to run across all the cores of the Cell architecture
- We show how performance of TBB programs on Cell can be boosted using a *portable* template-library to optimize data-movement
- We demonstrate the effectiveness of our techniques experimentally

To our knowledge, this work marks the first demonstration of portable code parallelised with TBB executing on a heterogeneous multicore architecture.

2 Background

2.1 The TBB `parallel_for` construct

We illustrate the `parallel_for` construct using an example distributed with TBB that simulates seismic effects. Figure 1 shows a serial loop. In Figure 2 the loop body is expressed as a C++ function object, `UpdateVelocityBody`, which defines an **`operator`**`()` method to operate on elements in a given range. The `parallel_for` function template takes as parameters a function object and an iteration space. When invoked, the function object is applied to each element in the iteration space, and multiple elements of the iteration space can be processed in parallel. The programmer does not determine how many tasks are to be created, nor how many threads are to be used.

2.2 Offload C++

The central construct of Offload C++ is the *offload block*, a lexical scope prefixed with the **`__offload`** keyword. In the Cell BE implementation of Offload C++, code outside

```
void SerialUpdateVelocity() {
  for(int i=1; i<Height-1; ++i)
    for(int j=1; j<Width-1; ++j)
      V[i][j] = D[i][j]*(V[i][j]+L[i][j]*
        (S[i][j]-S[i][j-1]+T[i][j]-T[i-1][j]));
}
```

Fig. 1: A serial simulation loop

```
struct UpdateVelocityBody {
  void operator()(const blocked_range<int>& r) {
    for(int i=r.begin(); i!=r.end(); ++i)
      for(int j=1; j<Width-1; ++j)
        V[i][j] = D[i][j]*(V[i][j]+L[i][j]*
          (S[i][j]-S[i][j-1]+T[i][j]-T[i-1][j]));
  }
};
void ParallelUpdateVelocity() {
  parallel_for( blocked_range<int>(1, Height-1),
                UpdateVelocityBody() );
}
```

Fig. 2: Simulation loop body as a C++ function object, executable using `parallel_for`

an offload block is executed by the host processor (PPE). When an offload block is reached, the host creates an accelerator (SPE) thread that executes the code inside the block. This thread runs asynchronously, in parallel with the host thread. Multiple SPE threads can be launched concurrently via multiple offload blocks. Each offload block returns a handle, which can be used to wait for completion of the associated SPE thread.

3 Offloading TBB parallel loops on the Cell BE architecture

The example of Figure 2 shows that TBB makes it easy to parallelise regularly structured loops. However, TBB does not support heterogeneous architectures with multiple memory spaces, such as the Cell BE.

We now show that, by implementing the `parallel_for` construct in Offload C++ we can allow the code of Figure 2 to execute across *all* cores of the Cell. The key observation is that TBB tasks are an abstraction over a thread-based model of concurrency, such as that provided by Offload C++ for heterogeneous architectures.

We implement the parallel loop templates of TBB to distribute loop iterations across both the SPE and PPE cores of the Cell. These template classes are included in a small set of header files compatible with the Offload C++ compiler. Figure 3 shows a simple version of `parallel_for` implemented using Offload C++; `parallel_reduce` can be implemented similarly.

The implementation in Figure 3 performs static work division. Multiple distinct implementations with different static and dynamic work division strategies over various subsets of the available cores can be implemented via additional overloads of the `run` function. Dynamic work division is achieved by partitioning the iteration space dynamically to form a work queue, guarded by a mutex, from which the worker threads obtain units of work to perform. This provides dynamic load balancing, as workers with less challenging work units are able to perform more units of work. Overloaded versions of `parallel_for` allow the user to select a specific work partitioner, *e.g.* to select static or dynamic work division.

Work division between the SPE cores *and* the PPE core is performed in the `run` method of the `internal::start_for` template. Offload's automatic call graph duplication makes this straightforward, despite the differences between these cores: in Figure 3, `local_function` is called on both the SPE (inside the offload block) and PPE (outside the offload block) without modification to the client code.

```
template<typename Range, typename Body>
void parallel_for( const Range& range, const Body& body ) {
  internal::start_for<Range,Body>::run(range,body);
}

template<typename Range, typename Body>
class start_for<Range, Body> {
public:
  static void run(  const Range& range, const Body& body ) {
    typedef Range::const_iterator iter;

    // Query the runtime for the number of SPE cores we may use
    unsigned NUM_SPES = num_available_spes();
    offloadThread_t handles[NUM_SPES];
    iter start     = range.begin(); // Simple 1D range work division
    iter end       = range.end();
    iter size      = (end - start);
    // NUM_SPES+1 because the PPE will do some work
    iter chunksize = size/(NUM_SPES+1);

    const Body local_body = body;

    for (int i = 0; i < NUM_SPES; ++i) {
      iter local_begin = start + chunksize*i;
      iter local_end = local_begin + chunksize;

      if(local_end > end)
        local_end = end;
      // Partition iterations into sub-range
      Range local_range(local_begin,local_end);
      // Spawn asynchronous SPE thread for sub-range
      handles[i] = __offload(local_body, local_range) {
        local_body(local_range);
      };
    }
    {   // PPE also executes a sub-range
        iter local_begin = start + chunksize*NUM_SPES;
        Range local_range(local_begin,end);
        local_body(local_range);
    }
    for (int i = 0; i < NUM_SPES; i++)
      offloadThreadJoin(handles[i]); // Await completion of SPE threads
  }
};
```

Fig. 3: An Offload C++ implementation of `parallel_for` for the PPE and SPE cores

In Figure 3, `NUM_SPES` holds the number of SPEs available to user programs in addition to the PPE core. To use all the cores, we divide work between `NUM_SPES+1` threads. One thread executes on the PPE, the others on distinct SPEs. The body of `run` spawns offload threads parameterised with a single sub-range and the function object to apply; it then also applies the function object to a sub-range on the PPE, before finally awaiting the completion of each offload thread.

When passing function objects into template classes and template functions, the functions to invoke are all statically known. Therefore, the Offload C++ compiler is able to automatically compile the function object **`operator`**`()` routine for the SPE and for the PPE, generating the data transfer code needed to move data between global and SPE memory [3].

4 Portable tuning for performance

Offload C++ enables code written for a homogeneous shared memory multi-core architecture to run on heterogeneous multi-core architectures with fast local memories. A consequence of this is that the relative cost of data access operations differs, depending on the memory spaces involved. Thus the performance characteristics of code may change when offloaded.

We discuss the default data-movement strategy employed by Offload, a software cache (§4.1). We then discuss portable optimisations that can be applied: local shadowing (§4.2), and bulk transfers (§4.3). While these optimisations are generic to Offload C++, we demonstrate in §5 that they can improve the performance of TBB applications running on the Cell via Offload C++.

4.1 Default data-movement: software cache

The Offload C++ compiler ensures that access to data declared in host memory results in generation of appropriate data-movement code. The primary mechanism for data-movement on Cell is DMA. However, issuing a DMA operation each time data is read or written tends to result in many small DMA operations. This can lead to inefficient code, since providing standard semantics for memory accesses requires synchronous DMA transfers, introducing latency into data access.

A software cache is used to avoid this worst-case scenario. When access to host memory is required, the compiler generates a cache access operation. At runtime, a synchronous DMA operation is only issued if the required data is not in the software cache. Otherwise, a fast local store access is issued. When contiguous data is accessed, or the same data is accessed repeatedly, the overhead associated with cache-lookups is ameliorated by eliminating the much greater overhead associated with DMA. Writes to global memory can be buffered in the cache and delayed until the cache is flushed or the cache-entry is evicted to make room for subsequent accesses.

The software cache is small: 512 bytes by default. The cache is both a convenience and, in many cases, an optimisation. However, it is not suited to bulk data transfers where each cache-line is evicted without being reused. In such a case, the cache leads to overhead without benefit. We discuss mechanisms for bypassing the cache where appropriate in §4.2 and §4.3.

4.2 Local shadowing

Although use of a software cache can significantly improve performance over naïve use of DMA, accessing the cache is significantly more expensive than performing a local memory access, even when a cache hit occurs.

A common feature of code offloaded for Cell without modification is repeated access to the same region of host memory by offloaded code. In this case, rather than relying on the software cache, a better strategy can be to declare a local variable or array, copy the host memory into this local data structure *once*, and replace accesses to the host memory with local accesses throughout the offloaded code. If the offloaded code modifies the memory then it is necessary to copy the local region back to host memory before offload execution completes. We call this manual optimisation *local shadowing*: host data is shadowed by local data to improve performance.

We illustrate local shadowing with the following code, a fragment of the raytracer discussed in §5.1:

```
Sphere spheres[sphereCount]; // Allocated in host memory
...
__offload { ...
  RadiancePathTracing(&spheres[0], sphereCount, ... );
... };
```

Scene data allocated in host memory (the `spheres` array, declared outside the **`__offload`** block), and passed into the `RadiancePathTracing` function. This function repeatedly accesses elements of `spheres` via the software cache. We can apply local shadowing by copying the scene data from `spheres` into a locally-allocated array, `local`, declared inside the **`__offload`** block:

```
Sphere spheres[sphereCount]; // Allocated in host memory
...
__offload { ...
  Sphere local[sphereCount]; // Allocated in local memory
  for (int i = 0; i < sphereCount; ++i)
    local[i] = spheres[i];
  RadiancePathTracing(&local[0], sphereCount, ... );
... };
```

A pointer to `local` is now passed to `RadiancePathTracing`, redirecting accesses to scene data to fast, local memory. This optimisation reduces access to scene data via the software cache to the “copy-in” loop; after this, accesses are purely local. Since scene data is not modified during raytracing, there is no need for a “copy-out” loop.

Local shadowing does not compromise portability: in a system with uniform memory the copy-in and copy-out are unnecessary, but yield equivalent semantics. Assuming that the code using the locally shadowed data is substantial, the performance hit associated with local shadowing when offloading is not applied is likely to be negligible.

4.3 Bulk data transfers

Offload C++ provides a header-file library of portable, type-safe template classes and functions to wrap DMA intrinsics and provide convenient support for various data access use cases. Templates are provided for read-only (`ReadArray`), write-only (`Write-Array`) and read/write (`ReadWriteArray`) access to arrays in host memory.

The array templates follow the Resource Acquisition is Initialisation (RAII) pattern [6], where construction and automatic destruction at end of scope can be exploited to perform processing. Transfers into local memory are performed on construction of `ReadArray`/`ReadWriteArray` instances, and transfers to host memory are performed on destruction of `ReadWriteArray`/`WriteArray` instances.

```
struct UpdateVelocityBody {
  void operator()(const blocked_range<int>& range ) const {
    for( int i=range.begin(); i!=range.end(); ++i ) {
      ReadArray<float, Width> lD(&D[i][0]),
      ReadArray<float, Width> lL(&L[i][0]);
      ReadArray<float, Width> lS(&S[i][0]);
      ReadArray<float, Width> lT(&T[i][0]);
      ReadArray<float, Width> lpT(&T[i-1][0]);
      ReadWriteArray<float, Width> lV(&V[i][0]);
      for( int j=1; j < Width-1; ++j )
        lV[j] = lD[j]*(lV[j]+lL[j]*(lS[j]-lS[j-1]+lT[j]-lpT[j]));
    }
  }
};
```

Fig. 4: Using DMA template wrappers for efficient data transfer

Figure 4 illustrates optimising the example of Figure 2 with bulk transfers. The declaration `ReadArray<float, Width> lD(&D[i][0])` declares `lD` a local **`float`** array, of size `Width`, and issues a synchronous DMA to fill `ld` with data from host array `D` (hence `lD` stands for "local `D`"). The `ReadWriteArray` instance `lV` is similar, except that when destroyed (on scope exit), a synchronous DMA restores the contents of `lV` to `V`. Velocity update is now performed with respect to local arrays only.

Bulk transfer templates share similarities with local shadowing (§4.2). However, they hide details of copy-in and copy-out operations from the programmer, and bypass the software cache completely, which is often significantly more efficient than an element-by-element copy would be.

At compile time, when targetting the PPE, an implementation of the templates designed so that no performance penalty is incurred is selected. This implementation is also usable on systems with single memory spaces, maintaining portability of code using the templates. Additional data-movement use cases can be implemented by users using the same template functions abstracting transfer operations used to implement the array templates.

5 Experimental Evaluation

We demonstrate the effectiveness of our approach to offloading TBB programs to run on the Cell using a set of parallel TBB programs. Experiments are performed on a Sony PlayStation 3 (with six SPEs accessible), running Fedora Core 10 Linux and IBM Cell SDK v3.0. Parallel benchmarks are compiled using Offload C++ v1.0.4, optimisation level -03. Serial versions of the benchmarks are compiled using both GCC v4.1.1, and Offload C++ v1.0.4. The faster of the two serial versions is taken as the baseline for measuring the speedup obtained via parallelisation.

– **Seismic simulation** Simulation discussed in §2.1 for a 1120×640 pixel display
– **SmallPT-GPU Raytracer** A global illumination renderer generating 256×256 pixel images from scenes with between 3 and 783 spheres, computing sphere-ray intersections with specular, diffuse, and glass reflectance with soft shadows and anti-aliasing [7]
– **Image processing kernels** A set of 8 kernels operating on a 512×512 pixel image, performing black-and-white median, colour median and colour mean filtering; embossing; sharpening; greyscale conversion; Sobel and Laplacian edge detection
– **PARSEC Black-Scholes** Partial differential equations modelling the pricing of financial options, from the PARSEC benchmark suite [8] using the *large* data set
– **PARSEC Swaptions** Simulates pricing a portfolio of swaptions using the Heath-Jarrow-Morton and Monte Carlo methods; from PARSEC using the *large* data set

5.1 Results

We present results showing the performance increases obtained by parallelising each benchmark across all available cores of the Cell (6 SPEs + PPE), compared with PPE-only execution. We note that in some cases, the speedup using all cores is more than 7×. The SPE cores are significantly different to the PPE, so we would not expect them to be directly comparable; a specific program may run faster across the SPEs due to higher floating point performance, or efficient use of scratch-pad memory.

Seismic Simulation: After an initial offload of the original code, we found that the data transfer intensive nature of this code results in non-optimal performance on the SPE as the data being processed is still held in the global memory, and not in fast SPE local store. To address this, we used the `ReadArray` and `ReadWriteArray` templates, as shown in Figure 4. We then obtained a 5.9× performance increase in the simulation over using the PPE alone.

Image Processing Kernels: Figure 5 shows performance results. We used local shadowing (§4.2) to hold input pixel rows in stack allocated arrays, implementing a sliding window over the input image, in which a new pixel row is fetched to over-write the local buffer storing the oldest row. Row fetches were then replaced with bulk data transfer template operations (§4.3), and writes of individual output pixels were buffered and written out via bulk transfer.

SmallPT-GPU Raytracer: Figure 6 shows performance results for three versions of the SmallPT raytracer in raytracing six scenes compared to the serial baseline. The first version uses `parallel_for` to execute on the SPEs and PPE. The second version uses local shadowing of the scene data, as discussed in §4.2. Finally, the third version uses a dynamic scheduling implementation of `parallel_for` where the SPEs and PPEs threads dequeue work from a shared queue, and thereby load balance amongst themselves.

Kernel	B&W Median	Col. Mean	Col. Median	Emboss	Laplacian	Sharpen	Sobel	Greyscale
Speedup	7.7×	7.4×	4.5×	3.6×	3.1×	5.3×	5.7×	3×

Fig. 5: Speedup for Image Kernels.

Scene	caustic	caustic3	complex	cornell large	cornell	simple
Global scene data	2.5×	2.6×	1.4×	4.5×	4.4×	2.7×
Local scene data	2.8×	3.0×	7.1×	7.2×	7.1×	3.1×
Dynamic parallel_for	4.9×	5.2×	10.1×	8.9×	8.5×	5.1×

Fig. 6: Speedup for SmallPT Raytracer using `parallel_for`.

PARSEC Black-Scholes: Conversion of the Black-Scholes benchmark was straightforward. A single `parallel_for` template function call represents the kernel of the application. We obtained a speedup of 4.0× relative to the serial version on PPE.

PARSEC Swaptions: It was necessary to refactor the codes in two stages. First, dynamic memory allocations were annotated to distinguish between memory spaces. Secondly, unrestricted pointer usage was replaced with static arrays. The local shadowing technique described in §4.2 was also employed as an optimisation. After these modifications, a speedup of 3.0× was obtained. This may rise with the incorporation of bulk data transfer optimisations as described in §4.3.

6 Related Work

OpenCL [9] is a language and interface for programming in a heterogeneous parallel environment. *e.g.* GPUs, homogeneous multi-core systems, and Cell [10]. Unlike Offload, OpenCL introduces “boilerplate” code to transfer data between distinct memory spaces via an API, and requires accelerator code to be written in the OpenCL language.

OpenMP targets *homogeneous* shared-memory architectures, although distributed and heterogeneous implementations do exist [11–13]. In contrast to OpenMP on Cell, the Offload compiler can use C++ templates to reify information obtained statically from the call graph, allowing users to optimise code using “specialised” template strategies selected for a specific target architecture *e.g.* the SPE.

7 Conclusions

We have shown how, using Offload C++, the TBB parallel loop construct `parallel_for` can be readily used to distribute work across the SPE and PPE cores of the Cell processor. Our proof of concept implementation provides both static and dynamic work division and supports a subset of the TBB library; `parallel_for` and `parallel_reduce`; the associated `blocked_range` templates, and the `spin_mutex` class.

We have also demonstrated that data transfer operations can be portably implemented, exploiting target-specific DMA transfer capabilities when instantiated in the context of code to be compiled for the SPE processors.

The parallel loop constructs we have implemented are facades over a more general task based model of programming, provided for ease of use and to support common patterns of parallelism directly. The fully general model of fork-join parallelism is more challenging to implement. However, it does not seem unfeasible, although a considerable task.We plan to investigate the extent to which such an implementation is feasible.

In addition, we are keen to assess the performance of offloaded TBB code on more highly parallel Cell-based systems, such as the IBM Cell Blade, which has 16 available SPEs.

We are interested in extending Offload C++ to massively parallel systems, such as GPUs. However, GPU-like architectures are not a good fit for the current Offload C++ programming model, which is generally applicable to heterogeneous multicore systems as long as some means of random access to a shared global store is provided. Adapting existing application code and Offload C++ to work with the restricted programming models associated with GPUs will be a significant research challenge.

References

1. H. P. Hofstee, "Power efficient processor architecture and the Cell processor," in *HPCA*. IEEE Computer Society, 2005, pp. 258–262.
2. Intel, "Threading Building Blocks 2.2 for Open Source," http://www.threadingbuildingblocks.org/.
3. P. Cooper, U. Dolinsky, A. Donaldson, A. Richards, C. Riley, and G. Russell, "Offload - automating code migration to heterogeneous multicore systems," in *HiPEAC'10*, ser. LNCS, vol. 5952. Springer, 2010, pp. 337–352.
4. A. Donaldson, U. Dolinsky, A. Richards, and G. Russell, "Automatic offloading of C++ for the Cell BE processor: a case study using Offload," in *MuCoCoS'10*. IEEE Computer Society, 2010, pp. 901–906.
5. Codeplay Software Ltd, "Offload: Community Edition," http://offload.codeplay.com.
6. B. Stroustrup, *The Design and Evolution of C++*. Addison-Wesley, 1994.
7. D. Bucciarelli, "SmallPT-GPU," http://davibu.interfree.it/opencl/smallptgpu/smallptGPU.html.
8. C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *PACT'08*. ACM, 2008, pp. 72–81.
9. Khronos Group, "The OpenCL specification," http://www.khronos.org/.
10. IBM Research, "OpenCL Development Kit for Linux on Power," http://www.alphaworks.ibm.com/tech/opencl.
11. "Extending OpenMP to Clusters," http://www.intel.com/, 2006.
12. K. O'Brien, K. M. O'Brien, Z. Sura, T. Chen, and T. Zhang, "Supporting OpenMP on Cell," *International Journal of Parallel Programming*, vol. 36, no. 3, pp. 289–311, 2008.
13. IBM Research, "XL C/C++ for Multicore Acceleration for Linux," http://www-01.ibm.com/software/awdtools/xlcpp/multicore/features/.

Fine-grained parallelization of a Vlasov-Poisson application on GPU

Guillaume Latu[1,2]

[1] CEA, IRFM, F-13108 Saint-Paul-lezDurance, France.
guillaume.latu@cea.fr
[2] Strasbourg 1 University & INRIA/Calvi project
7 rue Descartes, 67084 Strasbourg Cedex, France

Abstract. Understanding finely turbulent transport in magnetised plasmas is a subject of major importance to optimise experiments in present and future tokamak fusion reactor. The Vlasov equation provides a useful framework to perform experimental study and modelling of such devices. In this paper, we focus on the parallelization of a 2D semi-Lagrangian solver dedicated to plasma physics on GPGPU. The originality of the approach lies in the needed overhaul of both numerical scheme and algorithms, in order to compute accurately and efficiently in the CUDA framework. Two main topics are addressed. First, we show how to deal with 32-bit floating point precision, and we look at accuracy issues when employing the GPU in this kind of application. Second, we exhibit a very fine grain parallelization that fits well on a many-core architecture. A speed-up of almost 80 has been obtained by using a GPU instead of one CPU core. As far as we know, this work presents the first semi-Lagrangian solver dedicated to plasma physics ported on GPGPU. Simulations of fusion plasma consume a great amount of CPU time on today's supercomputers; thus, we provide design insights for future plasma simulators running on GPU clusters.

1 INTRODUCTION

The present paper highlights the porting of a semi-Lagrangian Vlasov-Poisson code on a GPU device. The work, described herein, follows a previous study made on the LOSS code described in other papers [CLS06]. A classical approach in the Semi-Lagrangian community involves the use of cubic splines to achieve the many interpolations needed by this scheme. The application we describe here, uses a local spline method designed specifically to perform decoupled numerical interpolations, while preserving classical cubic spline accuracy. In previous papers (see [CLS06,CLS07,CLS09,LCGS07]), this scalable method was integrated in MPI codes and a set of simulators based on that scheme were described and benchmarked. Both one-dimensional and two-dimensional domain decompositions were considered in order to decouple computations on many processors. Only relatively small MPI inter-processor communication costs were induced and these codes scaled well over hundreds of cores.

We will describe how to enrich the existing algorithm and numerical scheme included in the LOSS code, in order to obtain a tuned algorithm that fits well in the CUDA framework. This research is performed in an interdisciplinary team of physicists, mathematicians and computer scientists within the INRIA CALVI project and the French Atomic Energy Authority (CEA). In the rest of the paper, the numerical scheme and the accuracy issues are briefly introduced and the parallelization of the main algorithm with CUDA is described. The speedup and accuracy of the simulations are reported and discussed.

2 MATHEMATICAL MODEL

In the present work, we consider a reduced model for two physical dimensions (instead of six in the general case), corresponding to x and v_x such as $(x, v_x) \in \mathbb{R}^2$. The 1D variable x represents the configuration space and the 1D variable v_x stands for the velocity along x direction. Moreover, the self consistent magnetic field is neglected because v_x is considered to be small in the physical configurations we are looking at. The Vlasov-Poisson system then reads:

$$\frac{\partial f}{\partial t} + v_x \,.\nabla_x f + (E + v_x \times B) \,.\nabla_{v_x} f = 0, \tag{1}$$

$$E(x,t) = -\nabla\phi, \tag{2}$$

$$-\varepsilon_0 \nabla^2 \phi = \rho(x,t) = q \int f(x, v_x, t) d\, v_x. \tag{3}$$

where $f(x, v_x, t)$ is the particle density function, ρ is the charge density, q is the charge of a particle (only one species is considered) and ε_0 is the vacuum permittivity.

Equations (1) and (3) are solved successively at each time step. Equation (2) gives the self-consistent electrostatic field $E(x,t)$ generated by particles. The density ρ of Eq. (3) is evaluated in integrating f over v_x. Our work focuses on the resolution of Equation (1) using a backward semi-Lagrangian method [SRBG99]. The physical domain is defined as $\mathcal{D}_p^2 = \{(x, v_x) \in [x_{\min}, x_{\mathrm{Max}}] \times [v_{x_{\min}}, v_{x_{\mathrm{Max}}}]\}$. For the sake of simplicity, we will consider that the size of the grid mapped on this physical domain is a square indexed on $\mathcal{D}_i^2 = [0, 2^j - 1]^2$. To have a rectangular logical grid, it is easy to break this assumption and to consider different values for j depending on the dimension. Concerning the type of boundary conditions, a choice should be made depending on the test cases under investigation. At the time being, only periodic extension is implemented.

3 ALGORITHMIC ANALYSIS

3.1 Global numerical scheme

The Vlasov Equation (1) can be decomposed by splitting. It is possible to solve it, through the following elementary advection equations:

$$\partial_t f + v_x \partial_x f = 0, \quad (\hat{x} \;\text{ operator})$$
$$\partial_t f + \dot{v_x} \partial_{v_x} f = 0. \quad (\hat{v}_x \;\text{ operator})$$

Each advection consists in applying a shift operator. A splitting of Strang [CK76] is employed to keep a scheme of second order accuracy in time. We took the sequence $(\hat{x}/2, \hat{v}_x, \hat{x}/2)$, where the factor 1/2 means a shift over a reduced time step $\Delta t/2$. Algorithm 1 shows how the Vlasov solver of Eq. (1) is interleaved with the field solver of Eq. (3).

Algorithm 1: One time step

```
  Input  : $f_t$
  Output: $f_{t+\Delta t}$
  // Vlasov solver, part 1
1 1D Advection, operator $\frac{\hat{x}}{2}$ on $f(.,.,t)$
  // Field solver
2 Integrate $f(.,.,t+\Delta t/2)$ over $v_x$
3       to get density $\rho(.,t+\Delta t/2)$
4 Compute $\Phi_{t+\Delta t/2}$ with Poisson solver
5       using $\rho(.,t+\Delta t/2)$
  // Vlasov solver, part 2
6 1D Advection, operator $\hat{v_x}$ (use $\Phi_{t+\Delta t/2}$)
7 1D Advection, operator $\frac{\hat{x}}{2}$
```

Algorithm 2: Advection in x dir., dt time step

```
1 forall $v_x$ do
2     $a(.) \leftarrow$ spline coeff. of sampled function $f(.,v_x)$
3     forall $x$ do
4         $x^0 \leftarrow x - v_x.dt$
5         $f^\star(x,v_x) \leftarrow$ interpolate $f(x^0,v_x)$ with $a(.)$
```

3.2 Local spline method

Each 1D advection (along x or v_x) consists in two substeps. First, the density function f is processed in order to derive the cubic spline coefficients. Hence, we get a continuous representation of f over dimension x or v_x. The specificity of the local spline method is that a set of spline coefficients covering a given subdomain can be computed concurrently with other subdomains. Thus, it improves the standard approach that unfortunately needs a coupling between all coefficients along one direction. Second, spline coefficients are used to interpolate the function f at specific points. This substep is intrinsically parallel wether with the standard spline method or with the local spline method: one interpolation involves only a linear combination of four neighbouring spline coefficients.

Algorithm 2 details one advection and shows how interpolations are integrated. In this algorithm x^o is called the origin of the characteristic. With the local spline method, we gain concurrent computations during the spline coefficient derivation (line 2 of the algorithm). Our main goal in this paper is to convert the algorithm into an form adapted to the CUDA framework.

3.3 Floating point precision

Usually, semi-Lagrangian codes make extensive use of double precision floating point operations. The double precision is required because pertubations of small amplitude often play a central role during plasma simulation. For example, we focus on the very classical linear Landau damping test case with $k=0.5, \alpha=0.01$. The initial distribution function is given by

$$f(x, v_x, 0) = \frac{e^{-\frac{{v_x}^2}{2}}}{\sqrt{2\,\pi}}(1 + \alpha\,\cos(k\,x))\ .$$

Let us mention that other test cases are available in our implementation (like strong Landau damping, or two stream instability). We incorporated essentially classical problems picked to test the numerical algorithm and benchmark the code. Herafter, we focus on the linear Landau damping which highlights the accuracy problem one can expect in Vlasov-Poisson simulations.

The problem arising with simple precision computations is shown on Figure 1. The LOSS code (written in Fortran 90 and using MPI) is used to perform linear Landau simulation. The L^2 norm of electric potential is shown on the picture (electric energy) with logarithmic scale along the Y-axis. The double precision curve represents the reference simulation. Obviously, the difference between the two curves indicates clearly that simple precision is not enough to get the right result; especially for long-time simulation. With an accurate look at the figure, one can notice that the double precision simulation is accurate until reaching a plateau value near 10^{-20}. To go beyond this limit, one shoud have even more accurate interpolation scheme.

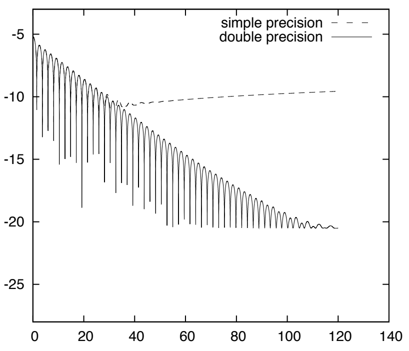


Fig. 1. Electric energy for Landau test case 1024×1024 with 32-bit precision advection versus advection with 64-bit precision (depending on time measured as a number of plasma period ${\omega_c}^{-1}$)

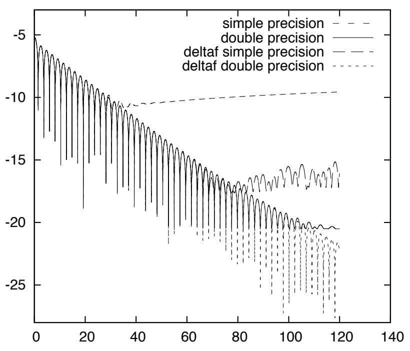


Fig. 2. Electric energy for Landau test case 1024×1024 using δf representation or standard representation. The floating precision is 32-bit or 64-bit accuracy.

3.4 Improvement of numerical precision

For the time being, one has to consider mostly simple precision (SP) computations to get maximum performance out of a GPGPU (General-Purpose Processing on Graphics Processing Units). The double precision (DP) is much slower than simple precision (SP) on today's devices. Furthermore, the memory bandwidth constraint is lighter with SP than with DP, considering the same number of elements to be transfered to floating point units.

The previous paragraph shows that SP leads to unacceptable numerical results. It turns out that our numerical scheme could be modified to reduce numerical errors even with only SP operations during the advection steps. In order to do so, we will introduce a new function

$$\delta f(x, v_x, t) = f(x, v_x, t) - f_{\mathrm{ref}}(x, v_x).$$

Working on the δf function could improve accuracy if the values that we are working on are sufficiently close to zero. Then, the reference function f_{ref} should be chosen such that the δf function remains relatively small (in L_∞ norm). As we will see hereafter, it is convenient to assume that f_{ref} is a constant along the x dimension. For the Landau test case, we choose

$$f_{\mathrm{ref}}(v_x) = \frac{1}{\sqrt{2\,\pi}}\, e^{-\frac{{v_x}^2}{2}}.$$

As the function f_{ref} is constant along x, the x-advection applied on f_{ref} leaves f_{ref} unchanged. Then, it is equivalent to apply $\hat{x}$ operator either on function δf or on function f. Working on δf is very worthwile: for the same number of floating point operations, we increase accuracy in working on small differences instead of large values. Concerning the $\hat{v_x}$ operator however, both f_{ref} and f are modified. For each advected grid point (x, v_x) of the $f^\star$ function, we have (v_x^o is the foot of the characteristic):

$$\begin{aligned} f^\star(x, v_x) &= f(x, v_x^o) = \delta f(x, v_x^o) + f_{\mathrm{ref}}(v_x^o), \\ \delta f^\star(x, v_x) &= f^\star(x, v_x) - f_{\mathrm{ref}}(v_x), \\ \delta f^\star(x, v_x) &= \delta f(x, v_x^o) - (f_{\mathrm{ref}}(v_x) - f_{\mathrm{ref}}(v_x^o)). \end{aligned}$$

Working on δf instead of f changes the operator $\hat{v_x}$. To advect in the v_x direction, we are looking for all values $f^\star(x, v_x)$ found thanks to the previous equations. To compute these values, we need to interpolate both $f(x, v_x^o)$ and $(f_{\mathrm{ref}}(v_x)–f_{\mathrm{ref}}(v_x^o))$. In doing so, we increase the number of computations ; because in the original scheme we had only one interpolation per grid point (x, v_x), whereas we have two in the new scheme. In spite of doubling the number of operations for evaluting the v_x operator, we expect the numerical accuracy to be enhanced using δf representation. Here is a sketch of the proposed δf based scheme that replaces that of Algorithm 1:

Algorithm 3: One time step with δf scheme

```
  Input  : $\delta f_t$
  Output: $\delta f_{t+\Delta t}$
  // Vlasov solver, part 1
1 1D advection on $\delta f$, operator $\frac{\hat{x}}{2}$
  // Field solver
2 Integrate $\delta f(.,.,t+\Delta t/2) + f_{\mathrm{ref}}(.)$ to get $\rho(.,t+\Delta t/2)$
3 Compute $\Phi_{t+\Delta t/2}$, with Poisson solver on $\rho(.,t+\Delta t/2)$
  // Vlasov solver, part 2
4 1D advection on $\delta f$, operator $\hat{v_x}$ (using $\Phi_{t+\Delta t/2}$)
5     $\rightarrow$ stored into $\delta f$
6 Interpolations of $f_{\mathrm{ref}}(v_x) - f_{\mathrm{ref}}(v_x^o)$ (using $\Phi_{t+\Delta t/2}$)
7     $\rightarrow$ results added into $\delta f$
8 1D advection on $\delta f$, operator $\frac{\hat{x}}{2}$
```

4 CUDA ALGORITHMS

4.1 CUDA Framework

Designed for NVIDIA GPUs (Graphics Processing Units), CUDA is a C-based general-purpose parallel computing programming model. Using the CUDA programming model, GPUs can be regarded as computation devices operating as coprocessors to the central processing unit (CPU). GPUs communicate with the CPU through fast PCI-Express ports. Over the past few years, a lot of successful experiments with GPGPU have been reported in the literature. An overview of the CUDA language and architecture will not be given here (for an introduction, see for example [NVI09]). Our reference implementation of LOSS used for comparison is written in Fortran 90 langage and uses the MPI library. The CUDA version of LOSS presented here mixes Fortran 90 code and external C calls (to launch CUDA kernels).

4.2 Data placement

We perform the computation on data δf of size $(2^j)^2$. Typical domain size varies from 128×128 (64 KB) up to 1024×1024 (4 MB). The whole domain fits easily in global memory of current GPUs. We could even store two data functions in global memory to avoid using an in-place algorithm. The main computational cost of our application is located in the four advection steps shown in Algorithm 3. In order to reduce unnecessary overheads, we decided to avoid transfering 2D data δf between the CPU and the GPU as far as we can. So we kept data function δf onto GPU global memory. Computation kernels directly update the 2D data stored on the GPU global memory. For diagnostics purposes only, the δf function is transfered to the CPU at a given frequency (a given number of time steps) and stored on disk. The end-user can then view or postprocess the diagnostic files.

4.3 Spline coefficients computation

Spline coefficients (of 1D discretized functions) are computed on patches of 32 values of δf. As explained elsewhere [CLS06], a smaller patch would introduce significant overhead because of the cost of first derivative computations on the patch borders. A bigger patch would increase the computational grain which is a bad thing for GPU computing that favors scheduling large number of threads.

The 2D domain is decomposed into small 1D vectors (named "patches") of 32 δf values. To derive the spline coefficients, small LU systems are solved. The assembly of right hand side vector used in this solving step can be summarized as follows: keep the 32 initial values, add 1 more value of δf at the end of the patch, and then add two derivatives of δf located at the border of the patch. Once the right hand side vector is available (35 floatint point numbers), two precomputed matrices L and U are inverted in order to derive spline coefficients. This step uses the classical forward and backward substitution. We decided not to try to parallelize this small LU solver: a single CUDA thread will be in charge of computing spline coefficients on one patch taking as input a right hand side vector, and two constant matrices L and U. That point could be improved in the future in order to use several threads instead of one.

4.4 Parallel interpolations

On one patch, 32 interpolations need to be done (excluding on domain boundaries where periodicity is taken into account). Each interpolation requires combining four spline coefficients. All these interpolations are decoupled. To maximize parallelism, one can even try to dedicate one thread per interpolation. Nevertheless, as some auxiliary computations could be factorized (for example the shift computed to find the foot of the characteristic), it is relevant to perform more than one interpolation per thread to reduce global computation cost. The exact number of such interpolations per thread is a parameter of our code and is has an impact on performance. In the following, we named this blocking factor B.

4.5 Data load

The computational intensity of the advection step is not that high. During the LU phase (*spline coefficients computation*), each input data is read and written twice and generates two multiplications and two additions in average. During the *interpolation step*, there are four reads and one write per input data and also four multiplications and four additions.
The low computational intensity implies that we could expect shortening the execution time in reducing loads and writes from GPU global memory to the floating point units. So, there is a benefit to group the spline computation and the interpolations in a single kernel. Several benchmarks have confirmed that with two distinct kernels (one for building splines and one for interpolations) instead of one, the price of load/store in the GPU memory increases. Thus, we now describe the solution with only one kernel that maximizes the computational intensity.

4.6 Domain decomposition and fine grain algorithm

To fit into the CUDA environment, the 2D computational domain is split into grids and blocks. We have designed three main kernels. Here is their short description:

KernVA	operator $\hat{v_x}$ on $\delta f(x, v_x)$
KernVB	adding $f_{\text{ref}}(v_x) - f_{\text{ref}}(v_x^o)$ to $\delta f(x, v_x)$
KernX	operator $\hat{x}$ on $\delta f(x, v_x)$

These kernels are very similar. Each of the three kernels begins with a load of a 2D rectangular shape of data into shared memory. Hence, the block of threads can share these data along the following computations. At the end of kernels, writes are performed in the GPU global memory in a rectangular 2D area. The main steps of an advection kernel (`KernVA` or `KernX`) are given in Algorithm 4. The computations of $8\,n$ threads acting on $32\,n$ real number values are described (which means that $B=4$ was hardcoded for this particular example).

```
Algorithm 4: Skeleton of an advection kernel
  Input  : $f_t$ in global memory of GPU
  Output: $f_{t+dt}$ in global memory of GPU
  // A) Load from global mem. to shared mem.
1 Each thread loads 4 floats from global mem.
2 Floats loaded are stored in shared memory
3 Boundary conditions are set (extra floats are read)
4 Synchro.: each block of threads owns $n$ vectors of 32 floats
  // B) LU Solver
5 1 thread over 8 solves a LU system
6         7 threads over 8 are idle
7 Synchro.: one block has $n$ vectors containing spline coeff.
  // C) Interpolations
8 Each thread computes 4 interpolations
  // D) Writing to GPU global memory
9 Each thread writes 4 floats to global mem.
```

In Algorithm 4, the first `A)` substep reads floats from GPU global memory and puts them into fast GPU shared memory. Then, a synchronization point waits for completion of all threads within one block of threads. When entering the `B)` substep, all input data have been copied into shared memory. Concurrently in the block of threads, small *LU* system are solved. During this small computation step, 87% of the threads are idle. Spline coefficients are finally known and stored in shared memory. In substep `C)`, each thread computes 4 interpolations (because $B=4$ in this example) using spline coefficients. This task is the most computation intensive part we have to tackle in the Vlasov-poisson solver. Finally, substep `D)` writes results into global memory.

5 PERFORMANCE

5.1 Machines

In order to develop the code and perform small benchmarks, a cheap personal computer has been used. The characteristic of the CPU are the following: Dual core E2200 Intel processor (2.2Ghz), 2 GB of RAM, 4 GB/s peak bandwidth, 4 GFLOPS peak, 1 MB L2 cache. The GPU is a GTX260 Nvidia card: 1.24 Ghz clock speed, 0.9 GB of global memory, 95 GB/s peak bandwidth, 750 GFLOPS peak, 27 multiprocessors, 8 cores per multiprocessor (for a total of 216 cores). The CPU-GPU transfer bandwidth is as small as 1 GB/s.

Another computer (at CINES, Montpellier, FRANCE) has been used for our benchmarks. The CPU part is a bi-socket quad-core E5472 (Harpertown), Xeon Intel 3 Ghz, 1 GB RAM , peak bandwidth 5 GB/s, 12 GFLOPS peak, L2 cache 2×6 MB. Concerning the GPU, the machine is connected to a Tesla S1070, 1.44Ghz, 4 GB global memory, 100 GB/s peak bandwidth, 1000 GLOPS peak, 30 multiprocessors, 8 cores per multiprocessor (for a total of 240 cores).

5.2 Small test case

Let us first have a look on execution time of the δf scheme on CPU or on GPU. We consider the small testbed (Dual core E2200 - GTX 260), and a reduced test case (data size = 256^2). The simulation ran on a single CPU core. Timing results are shown in the first column of Table 1. Then we ran the GPU version of the code on the same physical problem on the 216 cores of the GTX260 card. Timing results and speedup (reference execution time is the CPU single core timing) are given in the second column of the table.

Substeps in one time step	CPU (deltaf 4B)	GPU (deltaf 4B)
X Advection	5123 μs (*1.0*)	172 μs (*29.7*)
V Advection	4850 μs (*1.0*)	144 μs (*33.7*)
Field computation	133 μs (*1.0*)	93 μs (*1.4*)
Complete Iteration	10147 μs (*1.0*)	546 μs (*18.6*)

Table 1. Computation times inside a time step and speedup (in parentheses) averaged over 5000 calls - 256^2 test case, E2200/GTX260

The speedup is approximately 30 for the two significant computation steps, but is smaller for the field computation. The field computation part includes two substeps: first the integral computations over the 2D data distribution function, second the solving of a 1D poisson equation. The performance of integrals is bounded up by the loading time of 2D data from global memory of the GPGPU. This substep is not computationnally intensive, because there is only one addition to do per loaded float to do. The second substep that solves Poisson equation is a small 1D problem that could not easily be parallelized. Furthermore, we loose much time in lauching the poisson kernel on the GPU. We measured approximately a cost of 25 μs per kernel launch. During launch, no calculation and no loads takes place.

5.3 Large test case

We now have a look at a larger test case with data size equal to 1024^2. The two testbeds described earlier were used. Performance are slightly better using the Xeon/Tesla1070 compared to the E2200/GTX260.

Substeps in one time step	CPU (deltaf 4B)	GPU (deltaf 4B)
X Advections	79600 μs (*1.0*)	890 μs (*90*)
V Advections	89000 μs (*1.0*)	1000 μs (*89*)
Field computation	1900 μs (*1.0*)	180 μs (*11*)
Complete Iteration	171700 μs (*1.0*)	2250 μs (*76*)

Table 2. Computation time and speedups (in parentheses) averaged over 5000 calls - 1024^2 test case - E2200/GTX260

Substeps in one time step	CPU (deltaf 4B)	GPU (deltaf 4B)
X Advections	67000 μs (*1.0*)	780 μs (*86*)
V Advections	42000 μs (*1.0*)	960 μs (*43*)
Field computation	1500 μs (*1.0*)	200 μs (*7*)
Complete Iteration	110000 μs (*1.0*)	2200 μs (*50*)

Table 3. Computation time and speedups (in parentheses) averaged over 5000 calls - 1024^2 test case - Xeon/Tesla1070

Speedups of GPU over CPU are higher than in the previous smaller test case. The advection kernels reach speedups from 75 to 90 compared to one CPU single core computation. For this data size, the field computation does represent a small amount of computation (1D problem) compared to the advection kernels (2D problems). The relatively low speedup for the field solver does not penalize the global simulator performance. A complete iteration of the simulation is performed 76 times quicker on the 216 cores of the GTX260 than on a single core E2200.

CONCLUSION

We have developed a so-called δf model that tends to be more precise than the standard model. It turns out to be a valid approach to perform a Semi-Lagrangian Vlasov-Poisson simulation using only 32-bit floating-point precision instead of classical 64-bit precision simulations.

We have described the implementation on GPU of the advection operator used in Semi-Lagrangian simulation, which is subject to architectural constraints. We have discussed the kernel structure and the trade-offs made to accommodate the GPU hardware. A very fine grain parallelization of the advection step is presented that scales well on thousands of threads.

The original MPI application (before the porting to CUDA) is bounded up by memory bandwidth because computational intensity is small. It is well known that algorithms of high computational intensity can be efficiently implemented on the GPGPUs. We have demonstrated in this paper that algorithms of low computational intensity can also benefit from GPU hardware. We have built a GPU implementation reaching a significant speedup of overall 76 compared to a single core CPU computation. This allows solving quickly large Vlasov-Poisson test cases, on cheap and freely available personal computers. In the near future, we expect to integrate this solution into a 4D semi-Lagrangian code (with 2 dimensions both in space and velocity). The memory constraint imposed by such 4D simulations implies that we shall design a code that runs on multiple GPUs; it will allow for the enlargement of the available memory space. We are now targeting the design of a MPI+CUDA code that could run on a GPU cluster.

References

[CK76] C.Z. Cheng and Georg Knorr. The integration of the Vlasov equation in configuration space. *J. Comput Phys.*, 22:330, 1976.

[CLS06] N. Crouseilles, G. Latu, and E. Sonnendrücker. Hermite spline interpolation on patches for a parallel solving of the Vlasov-Poisson equation. Technical Report 5926, Research report INRIA, 2006. http://hal.inria.fr/inria-00078455/en/.

[CLS07] N. Crouseilles, G. Latu, and E. Sonnendrücker. Hermite spline interpolation on patches for parallelly solving the Vlasov-Poisson equation. *Applied Mathematics and Computer Science*, 17(3):335–349, 2007.

[CLS09] N. Crouseilles, G. Latu, and E. Sonnendrücker. A parallel Vlasov solver based on local cubic spline interpolation on patches. *J. Comput. Phys.*, 228(5):1429–1446, 2009.

[LCGS07] G. Latu, N. Crouseilles, V. Grandgirard, and E. Sonnendrücker. Gyrokinetic semi-lagrangian parallel simulation using a hybrid OpenMP/MPI programming. In *PVM/MPI*, pages 356–364, 2007.

[NVI09] NVIDIA. CUDA Programming Guide, 2.3, 2009. `http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf`.

[SRBG99] E. Sonnendrücker, J. Roche, P. Bertrand, and A. Ghizzo. The semi-lagrangian method for the numerical resolution of the Vlasov equations. *J. Comput. Phys.*, 149:201–220, 1999.

Highly Parallel Implementation of Harris Corner Detector on CSX SIMD Architecture

Fouzhan Hosseini, Amir Fijany, and Jean-Guy Fontaine

Tele Robotics and Applications Department, Italian Institute of Technology,
Via Morego 30, Genova, Italy
`{fouzhan.hosseini,amir.fijany,jean-guy.fontaine}@iit.it`
`http://www.iit.it`

Abstract. We present a much faster than real-time implementation of Harris Corner Detector (HCD) on a low-power, highly parallel, SIMD architecture, the ClearSpeed CSX 700, with application for mobile robots and humanoids. HCD is a popular feature detector due to its invariance to rotation, scale, illumination variation and image noises. Considering the CSX architecture, we have developed strategies for efficient parallel implementation of HCD, and we have achieved a performance of 465 frames per second (fps) for images of 640x480 resolution and 142 fps for 1280x720 resolution. For a typical real-time application with 30 fps, our fast implementation represents a very small fraction (less than %10) of available time for each frame and thus allowing enough time for performing other computations. Our results indicate that the CSX architecture is indeed a good candidate for achieving low-power supercomputing capability, as well as flexibility.

1 Introduction

Mobile robots and humanoids represent an interesting and challenging example of embedded computing applications. On one hand, in order to achieve a large degree of autonomy and intelligent behavior, these systems require a very significant computational capability to perform various tasks. On the other hand, they are severely limited in terms of size, weight, and particularly power consumption of their embedded computing system since they should carry their own power supply. The limitation of conventional computing architectures for these types of applications is twofold: first, their low computing power, second, their high power consumption. Emerging highly parallel and low-power SIMD and MIMD architectures provide a unique opportunity to overcome these limitations of conventional computing architectures. Exploiting these novel parallel architectures, our current objective is to develop a flexible, low-power, lightweight supercomputing architecture for mobile robots and humanoid systems for performing various tasks and, indeed, for enabling new capabilities.

Computer vision and image processing techniques are very common in robotic applications, e.g. motion detection, tracking, 3D reconstruction and object recognition. Feature detection is a low-level image processing task which is usually performed as the first step in many computer vision applications such as object tracking [1] and object detection/recognition [2]. Harris Corner Detector (HCD) [3] is a popular feature detector due to its invariance to rotation, scale, illumination variation and image noises.

Fast implementation of HCD has been considered on various architectures. Teixeira et al. [4] have implemented HCD on a graphics processing units (GPU). For an image of 640x480 resolution, the HCD is computed in 10.1 ms. They realized that the large number of memory accesses degrade performance. Therefore, by compressing each 2×2 pixels in the original image as one pixel, they reduced the computation time to 3.3 ms with one pixel imprecision. Another way to improve performance is to employ Field Programmable Gate Arrays (FPGAs). Dietrich [5] has implemented HCD on FPGA as part of a stereo vision system. The developed FPGA is capable of calculating HCD for images of the resolution 358×288 at the speed of 60 fps. Also, Cheng et al. [6] have proposed an ASIC implementation of HCD as part of a vision processor. The proposed architecture is capable of computing HCD for images of the resolution 128×128 at the speed of 1367 fps. Moreover, Saidani et al. [7] have employed Harris corner detector on Cell processor. ASICs and FPGAs could be used to design custom hardware for low-power high performance applications. GPU and Cell processor are more flexible, but the main limitation is the rather prohibitive power consumption. None of the above mentioned solutions satisfies our requirements for mobile system vision processing including low power consumption, flexibility, and real time processing capability simultaneously.

In this paper, we present a fast implementation of HCD on a highly parallel SIMD architecture, the ClearSpeed CSX 700. The CSX 700 has a peak computing power of 96 GFOLPS, while consuming less than 9 Watts. In fact, it seems that CSX provides one of the best (if not the best) performance in terms of GFLOPS/Watt among available computing architectures. Considering the CSX architecture, we have developed strategies for efficient parallel implementation of HCD.We have achieved a performance of 465 fps for images of 640x480 resolution and 142 fps for 1280x720 resolution. These results indeed represent a much faster than real-time implementation and better than those previously reported in the literature. For a typical real-time application with 30 fps, our fast implementation represents a very small fraction (less than %10) of available time for each frame and thus allowing enough time for performing other computations. Our experimental results, presented in this paper, clearly indicate that the SIMD architectures such as CSX can indeed be a good candidate for achieving low-power supercomputing capability, as well as flexibility, for embedded applications.

This paper is organized as follows. In Sect. 2, we briefly discuss the HCD algorithm. In Sect. 3, we briefly review the CSX architecture with emphasis on its salient features which have been exploited in our parallel implementation of HCD. In Sect. 4, our approach for parallel implementation of HCD on CSX architecture is described and experimental results are discussed in Sect. 5. Finally some concluding remarks and direction for our future works is presented in Sect. 6.

2 The Harris Corner Detector Algorithm

To detect corners in a given image, the HCD algorithm [3] proceeds as following. Let $I(x, y)$ denote the intensity of a pixel located at row x and column y of the image.

1. For each pixel (x, y) in the input image compute the elements of the Harris matrix $G = \begin{bmatrix} g_{xx} & g_{xy} \\ g_{xy} & g_{yy} \end{bmatrix}$ as follows:

$$g_{xx} = \left(\frac{\partial I}{\partial x}\right)^2 \otimes w \quad g_{xy} = \left(\frac{\partial I}{\partial x}\frac{\partial I}{\partial y}\right) \otimes w \quad g_{yy} = \left(\frac{\partial I}{\partial y}\right)^2 \otimes w, \tag{1}$$

   where $\otimes$ denotes convolution operator and w is the Gaussian filter.
2. For all pixel (x, y), compute Harris' criterion:

$$c(x, y) = \det(G) - k(trace(G))^2 \tag{2}$$

   where $\det(G) = g_{xx}.g_{yy} - g_{xy}^2$, k is a constant which should be determined empirically, and $trace(G) = g_{xx} + g_{yy}$.
3. Choose a threshold τ empirically, and set all $c(x, y)$ which are below τ to zero.
4. Non-maximum suppression, i.e. extract points (x, y), which have the maximum $c(x, y)$ in a window neighborhood. These points represents the corners.

3 The CSX 700 Architecture

In this section, we briefly review the ClearSpeed CSX 700 architecture with emphasis on some of its salient features that have been exploited in our implementation (see, for example, [8], [9] for more detailed discussion). As illustrated in Fig. 1(a), CSX700 has two similar cores, each core has a DDR2 memory interface and a 128KB SRAM, called external memory. Each core also has a standard, RISC-like, control unit, also called *mono execution unit*, which is coupled to a highly parallel SIMD architecture called *poly execution unit*.

Poly execution unit consists of 96 processing elements (PEs) and performs parallel computation (see Fig. 1(b)). Each PE has a 128 bytes register file, 6KB of SRAM, an ALU, an integer multiply-accumulate (MAC) unit, and an IEEE 754 compliant floating point unit (FPU) with dual issue pipelined add and multiply, as well as support for division and square root.

The CSX700 has clock frequency of $250MHz$[10]. Considering one add and one multiply floating point units working in parallel and generating one result per clock cycle, the peak performance of each PE is then 500 MFOPS, leading to a peak performance of 96 GFLOPS for two cores (one chip). However, sequential (i.e., scalar) operations, wherein single add or multiply is performed, take 4 clock cycles to be performed [10]. This results to a sequential peak performance of 12 GFLOPS for two cores. This indeed represents a drastic reduction in the peak,

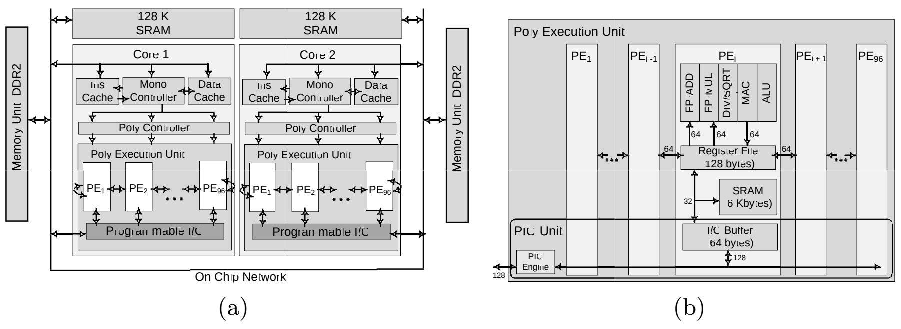


Fig. 1. (a)Simplified CSX Chip Architecture (b) Poly Execution Unit Architecture [8]

and hence, achievable performance. However, vector instructions which operate on sets of 4 data are executed much faster, e.g, vector add or multiply instructions take 4 cycles to be completed [10]. Therefore, vector instructions allow greater throughput for operations. However, the code generated by compiler may not be optimized. Therefore, in order to achive the best performance, we have also written part of our codes in assembly language of the CSX.

Poly execution unit includes a Programmable I/O (PIO) unit (Fig. 1(b)) which is responsible for data transfer between external memory and PEs' memories, called poly memory. It is important to note that the architecture of poly execution unit enables the computational units and the PIO unit to work in parallel, thus enabling overlapping of communication with computation. This feature is fully exploited in our implementation to reduce I/O overhead.

Moreover, each PE is capable of communicating with its two neighboring PEs by using a dedicated bus called *swazzle path*, which connects the register files of neighbors PEs (Fig. 1(b)). Consequently, on each cycle, PEs are able to perform a register-to-register data transfer to either their left or right neighbor, while simultaneously receiving data from the other neighbor.

4 Proposed Parallel Implementation

Considering the SIMD architecture of CSX, we have employed data parallel model of computation. Here, we first discuss our data decomposition strategy. Then, we discuss more details of our parallel implementation.

4.1 Data Decomposition

Having an image and an array of PEs, various data distributions schemes could be considered. The most obvious schemes are row (column)-stripe distribution, block distribution, and row (column)-cyclic distribution. Here, we discuss the effectiveness of each of these data distribution schemes for parallel implementation of HCD algorithms on the CSX architecture. An important consideration for the CSX architecture is the size of PE's memory which is rather limited. For the CSX architecture, various data distributions should be compared in terms of the following parameters: (a) required memory space for each PE; (b) redundant external memory communication; and (c) inter-PE communication time.

In the following, c and r denote the number of columns and rows in image matrix, respectively. According to the algorithm description in Sect. 2, HCD performs a set of operations in windows around each pixels. In fact, HCD uses windows which may have different sizes in 3 stages: calculating partial derivatives, Gaussian smoothing, and non-maximal suppression. Let ω be the sum of these window sizes. Also, let p indicate the number of PEs. Finally, in each memory communication, each PE reads or writes m bytes of data (pixel) from/into the external memory. Π is the memory space needed to calculate the elements of Harris matrix for m pixels.

Block distributions. In this scheme, as illustrated in Figure 2(a), the image is divided into $p = d * s$ blocks, with each block having c/d columns and r/s rows. The first block is assigned to the first PE, the second one to the second PE, and so on. Each block can be identified by an ordered pair (i, j) where $1 \leq i \leq s$ and $1 \leq j \leq d$. In the following, $P(i, j)$ denotes the PE which is responsible for processing the block (i, j) and refers to PE $((i - 1)s + j)$.

Figure. 2(b) depicts the boundary data needed for computation by $P(i, j)$ and its four immediate neighbors. To handle boundary data, needed by two neighboring PEs, there are two possible alternatives: transferring boundary data from external memory to both PEs, hence performing redundant data communication, or transferring to one PE and then using swazzling path to transfer it to the other PE. The former takes more time, and the latter requires more PE memory space to store boundary data as discussed in the following. To process first rows (columns), $P(i, j)$ requires the last rows (columns) of $P(i - 1, j)$ $(P(i, j-1))$, but these PEs have not yet received the data which $P(i, j)$ requires. Therefore, if the swazzling path is used then $P(i, j)$ should skip processing these

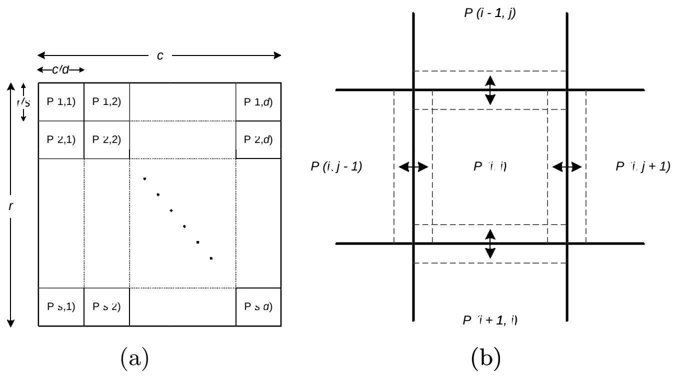


Fig. 2. (a)Block distribution. P(i, j) refers to $PE(i - 1)s + j$ (b) Boundary data for each PE in block distribution

boundary data, until $P(i-1,j)$ $(P(i,j-1))$ provides the required data. Also, for processing the last rows (columns) of data, $P(i,j)$ requires data which has already been sent to $P(i+1,j)$ $(P(i,j+1))$. For these PEs to provide the boundary data to $P(i,j)$, they need to store this part of data in their memory which is a limited resource. It should be noted that using block distribution scheme on the CSX architecture, the distance between $P(i,j)$ and $P(i+1,j)$ which process two neighboring blocks is d.

Row-strip distribution The first r/p rows are assigned to the first PE, the second r/p rows are assigned to the second PE, and so on. To handle boundary data, $PE(i)$ requires last rows of $PE(i-1)$ and first rows of $PE(i+1)$. In fact, like block distribution, boundary data could be transfered from external memory to both PEs or from one PE to another via swazzling path. The choice of PEs receiving the boundary data form external memory or via swazzle path depends on the trade-off between the required PE memory space and the cost of external memory communication.

Row-cyclic distribution In this scheme, the first row is assigned to the first PE, the second row to the second PE, and so on. Since one row is assigned to each PE, each PE needs to communicate with the PEs which are at most at the distance of $(w1)/2$. Here, each PE needs data just after its neighbor has finished processing that same data. So, swazzle path can be utilized without using extra poly memory space.

The parameters calculated for each data distribution strategy are summarized in Table 1. As can be seen, block and row-strip distribution schemes require either more PE memory space or more redundant external memory communications. In fact, for these schemes, the required poly memory space increases linearly with ω. Note that, the size of windows in HCD are determined empirically for each application. For larger ω, e.g. 7 or 11, using these data distributions, the required PE memory will be larger than poly memory space. Row-cyclic distribution needs less poly memory space and no redundant external memory communication. Although row-cyclic distribution uses inter-PE communication more than row-strip distribution by a factor of $\omega/2$, this overhead will be negligible since communication via swazzle path is very fast (see Sect. 3). Therefore, row-cyclic distribution scheme is the most efficient for implementing HCD on the CSX architecture.

4.2 Parallel Implementation of Harris Corner Detector Algorithm

In this section, we discuss parallel implementation of HCD on the CSX architecture, based on row-cyclic distribution scheme. Since, each CSX core includes 96 PEs, the input image is divided into groups of 96 rows. The computation of each group represents a sweep and sweeps are performed iteratively. Also, to utilize both cores of CSX700 processor, the input image is divided into two nearly equal parts. The first $\lceil r/2 \rceil + (\omega-1)/2$ rows are assigned to the first cores and the last $\lfloor r/2 \rfloor + (\omega-1)/2$ rows are assigned to the second core. Sending boundary lines to both cores enables each core to perform all computation locally. In the following, implementation of HCD on one core is explained (for one sweep).

Table 1. Figure of merit for different data distribution schemes. S indicates that boundary data is shared between PEs by using swazzling path. M indicates that boundary data is transferred from external memory

Data Dist.		Redundant External Memory Comm.	Inter-PE Comm.	PE Memory Space
Block Dist.	M	$cs(\omega-1)$	$r(\omega-1)$	$\omega\Pi+m$
	S	-	$(\omega-1)[cs+r]$	$(\omega+\frac{\omega-1}{2})\Pi+m$
Row-strip Dist.	M	$pc(\omega-1)$	-	$\omega\Pi+m$
	S	-	$c(\omega-1)$	$(\omega+\frac{\omega-1}{2})\Pi+m$
Row-cyclic Dist.		-	$c\frac{\omega(\omega-1)}{2}$	$\Pi+m$

Memory Communication Pattern For our parallel implementation, communication and computation overlapping can be greatly exploited due to the local nature of our computation. That is, the fact that there is no need to load the whole image into the PEs memory to start the computation. In fact, each image row is divided into segments of almost equal size (32 or 64 pixels, depending on the image size) and PEs can start the computation as soon as they receive the first segment of data. After receiving the firs segment of data, each PE initiates PIO data transfers to and from external memory, and continues to process the segment of data which is ready in its memory. In the background, PIO transfers new sets of data from external memory to memories of PEs and transfers the last sets of results to external memory. When the computation is finished and data is ready in PEs' memories, PEs start new PIO data transfers and continue computation of the new set of data. In our implementation, PEs never wait to receive data (except the initial phase). This overlapping of computation and communication significantly reduces the overhead in the parallel computation, thus enabling a much better performance.

Computation Steps In this section, we present processing of one segment of data. In our implementation of HCD, we have divided the algorithm into 5 steps: calculating partial derivation of I in direction x and y, Gaussian smoothing, computing Harris criterion, non-maximum suppression, followed by thresholding. Algorithm 1 shows the pseudocode for this processing.

To calculate partial derivation of I, we have used Prewitt operator. Prewitt operator uses two 3x3 kernels, P_X and P_Y, which are convolved with the original image to calculate approximations of the derivatives in x and y directions, respectively. In our implementation, we take advantages of the fact that convolution kernels used by Prewitt operator are separable, i.e. these kernels can be expressed as the outer product of two vectors.

$$P_X = \begin{bmatrix} -1\,0\,1 \\ -1\,0\,1 \\ -1\,0\,1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} * \begin{bmatrix} -1\,0\,1 \end{bmatrix} \quad P_Y = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix} * \begin{bmatrix} 1\,1\,1 \end{bmatrix} \tag{3}$$

So, the x and y derivation can be calculated by first convolving in one direction (using local data), then swazzling data and convolving in the other direction.

Next step is Gaussian smoothing. Elements of Harris matrix, g_{xx}, g_{xy}, and g_{yy} are calculated using (1). As stated in Sect. 3, the Gaussian smoothing can be performed using standard convolution methods. Gaussian kernel is also separable. Thus, the 2-D convolution can be performed by first convolving with a 1-D Gaussian in the x direction, and then swzzling the calculated values and convolving with another 1-D Gaussian in the y direction. The y component is exactly the same as x component but is oriented vertically.Then, Harris' criterion is computed using (2).

In the next step, non-maximum suppression, the maximum value of Harris criterion in each 3x3 neighborhood is determined. First, each PE obtains the maximum value in 1x3 neighborhood. Then, each PE swazzle the maximum values to both its neighbors. Receiving the maximal values of two neighboring rows, the maximum value in 3x3 neighborhood can then be obtained. Using this strategy, the maximum value of 9 element in a 3x3 neighborhood is obtained by just 4 comparisons.

Algorithm 1 Pseudocode of Parallelized HCD

```
$\omega_1$: Gaussian window size $\omega_2$: NMS window size

PEs in parallel do
1. Derivation of $I_x$ and $I_y$:
    $I_x = I \otimes [-1\ 0\ 1]$ ,    $I_x = [swazzle\_down(I_x), I_x, swazzle\_up(I_x)] \otimes [1\ 1\ 1]$
    $I_y = swazzle\_up(I) - swazzle\_down(I)$ ,    $I_y = I_y \otimes [1\ 1\ 1]$

2. Guassian Smoothing:
  $g_{xx} = I_x^2 \otimes \mathrm{x} - \mathrm{Guassian}$,   $g_{xy} = (I_x I_y 2) \otimes \mathrm{x} - \mathrm{Guassian}$,   $g_{yy} = I_y^2 \otimes \mathrm{x} - \mathrm{Guassian}$
  $g_{xx} = [swazzle\_down(g_{xx}), g_{xx}, swazzle\_up(g_{xx})] \otimes \mathrm{y} - \mathrm{Guassian}$
  $g_{xy} = [swazzle\_down(g_{xy}), g_{xy}, swazzle\_up(g_{xy})] \otimes \mathrm{y} - \mathrm{Guassian}$
  $g_{yy} = [swazzle\_down(g_{yy}), g_{yy}, swazzle\_up(g_{yy})] \otimes \mathrm{y} - \mathrm{Guassian}$

3. Computation Harris Criterion:    $c = g_{xx}g_{yy} - g_{xy}^2 - k(g_{xx} + g_{yy})$

4. Non-maximum suppression:
  for $k = 1$ to $m$
    $mx[k] = \max\{c[l] \mid k - (\omega_2 - 1)/2 \leq l \leq k + (\omega_2 - 1)/2\}$
  for $k = 1$ to $m$
    $mx[k] = \max\{mx[k],\ swazzle\_up(mx[k]),\ swazzle\_down(mx[k])\}$

5. Thresholding:
  for $k = 1$ to $m$
    if $c[k] \geq \tau$ and $c[k] == mxp[k]$
     corresponding pixel is corner

$*$ $swazzle\_up()$ and $swazzle\_down()$ represent communication with left and right
neighbors, respectively.
```

5 Results and Performance of Parallel Implementation

To evaluate the performance, we have implemented the following HCDs on the CSX700 architecture: $HCD_{3\times3}$ and $HCD_{5\times5}$ which uses a 3×3 and 5×5 Gaussian kernel, respectively. Since our proposed parallel approach provides flexibility, it can be easily applied to images with different sizes, and various sizes of Gaussian filter or non-maximum suppression window. The performance of implemented algorithms in terms of latency, fps, and sustained GFLOPS for different image resolutions are summarized in Table 2. As Table 2 shows, for all tested image resolutions, even for resolution of 1280x720, our implementation is much faster than real-time.

Table 2. Performance of HCD on CSX700 architecture using 3×3 and 5×5 Gaussian filter

Image	Latency (ms)		fps		Sustained GFLOPS	
Resolution	$HCD_{3\times3}$	$HCD_{5\times5}$	$HCD_{3\times3}$	$HCD_{5\times5}$	$HCD_{3\times3}$	$HCD_{5\times5}$
128x128	.165	.224	6060	4464	3.97	4.68
352x288	.8	1.22	1250	819	5.06	5.31
512x512	1.74	2.63	574	380	6.02	6.37
640x480	2.15	3.28	465	304	5.71	5.99
1280x720	7.04	10.89	142	91	5.23	5.41

The arithmetic intensity, i.e., number of operation per pixel, of $HCD_{3\times3}$ and $HCD_{5\times5}$ is 40 and 64 respectively. As Table 2 shows, the sustained GFLOP depends also on the image size. The reason is that in processing the last sweep of data, some PEs may be idle, and the number of idle PEs depends on image size. For example, performing $HCD_{3\times3}$ for images of resolution 640x480 and 1280x720, the number of idle PEs are 4 and 32, respectively. Due to more utilization of PEs, better GFLOPS is achieved for resolution 640x480.

Table 3 compares our implementation results with those reported in the literature. As can be seen, our approach provides much better performance in terms of latency or frame per second while providing a high degree of flexibility in terms of problem size and parameters.

Table 3. Comparison with other implementations in the literature

Image Resolution	fps reported in [ref]	fps achieved by our approach
128x128	1367 [6]	4464-6060
352x288	60 [5]	819
640x480	99 [4]	304

6 Conclusion and Future Work

We presented a much faster than real-time implementation of Harris Corner Detector (HCD) on a low-power, highly parallel, SIMD architecture, the ClearSpeed CSX 700. Considering the features of the CSX architecture, we presented strategies for efficient parallel implementation of HCD.We have achieved a performance of 465 fps for images of 640x480 resolution and 142 fps for 1280x720 resolution. These results indeed represent a much faster than real-time implementation. Our experimental results, presented in this paper, and our previous work [11] clearly indicate that the CSX architecture is indeed a good candidate for achieving low-power supercomputing capability, as well as flexibility, for embedded computer vision applications. We are currently implementing other more complex variants of HCD as well as more sophisticated and computationally more expensive feature detectors such as SIFT.

References

1. Yilmaz, A., Javed, O., Shah, M.: Object tracking: A survey. ACM Computing Surveys **38**(4) (2006) 13
2. Roth, P.M., Winter, M.: Survey of appearance-based methods for object recognition. Technical Report ICG-TR-01/08, Inst. for Computer Graphics and Vision, Graz University of Technology (2008)
3. Harris, C., Stephens, M.: A combined corner and edge detector. In: 4th Alvey Vision Conference. (1988) 147–151
4. Teixeira, L., Celes, W., Gattass, M.: Accelerated corner-detector algorithms. In: 19th British Machine Vision Conference(BMVC '08), Springer-Verlag (2008) 625–634
5. Dietrich, B.: Design and implementation of an fpga-based stereo vision system for the EyeBot M6. University of Western Australia (2009)
6. Cheng, C.C., Lin, C.H., Li, C.T., Chang, S.C., Chen, L.G.: iVisual: an intelligent visual sensor soc with 2790fps cmos image sensor and 205gops/w vision processor. In: 45th annual Design Automation Conference(DAC '08), ACM (2008) 90–95
7. Saidani, T., Lacassagne, L., Bouaziz, S., Khan, T.M.: Parallelization strategies for the points of interests algorithm on the cell processor. In: 5th International symposium on Parallel and Distributed Processing and Applications (ISPA'07). (2007) 104–112
8. ClearSpeed www.clearspeed.com: Clearspeed Whitepaper: CSX Processor Architecture. (2007)
9. ClearSpeed www.clearspeed.com: CSX600 Hardware Programming Manual. (Jan 2008) Document No. 06-RM-1305 Revision: 1.A.
10. ClearSpeed www.clearspeed.com: CSX600/CSX700 Instruction Set Reference Manual. (August 2008) 06-RM-1137 Revision: 4.A.
11. Hosseini, F., Fijany, A., Safari, S., Chellali, R., Fontaine, J.G.: Real-time parallel implementation of ssd stereo vision algorithm on csx simd architecture. In: Proceedings of the 5th International Symposium on Advances in Visual Computing (ISVC '09), Springer-Verlag (2009) 808–818

Static Speculation as Post-Link Optimization for the Grid Alu Processor

Ralf Jahr, Basher Shehan, Sascha Uhrig, Theo Ungerer
{jahr, shehan, uhrig, ungerer}@informatik.uni-augsburg.de

Institute of Computer Science
University of Augsburg
86135 Augsburg
Germany

Abstract. In this paper we propose and evaluate a post-link-optimization to increase instruction level parallelism by moving instructions from one basic block to the preceding blocks. The Grid Alu Processor used for the evaluations comprises plenty of functional units that are not completely allocated by the original instruction stream. The proposed technique speculatively performs operations in advance by using unallocated functional units.
The algorithm moves instructions to multiple predecessors of a source block. If necessary, it adds compensation code to allow the shifted instructions to work on unused registers, whose values will be copied into the original target registers at the time the speculation is resolved.
Evaluations of the algorithm show a maximum speedup of factor 2.08 achieved on the Grid Alu Processor compared to the unoptimized version of the same program. Reasons are a better exploitation of the instruction level parallelism and an optimized mapping of loops.

1 Introduction

The Grid Alu Processor (GAP, see Uhrig et al. [15]) was proposed to speed up the execution of single threaded sequential instruction streams. In difference to most other currently discussed designs it uses the available number of transistors not for complete cores but for a high number of functional units (FUs) set up as two-dimensional array. To configure it, a superscalar-like processor frontend loads a standard sequential instruction stream that is dynamically mapped onto the array by a special configuration unit. Execution speed is gained very much from the high level of parallelism supplied by the FUs.

The main influences on the mapping process are control and data flow dependencies as well as resource conflicts caused by limited resources. These dependencies restrict the level of instruction level parallelism that can be exploited. Thus, most of the time not all of the FUs of the GAP can be used although they could execute additional instructions at no or only little additional cost in terms of execution time.

The algorithm presented in this paper tackles this by moving parts of a basic block (source block) to one or more preceding blocks (target blocks). By this, results that might be required in the near future, e.g. after upcoming branches, are calculated speculatively on otherwise unused resources. At the time the reason for the speculation is resolved, these results are made visible by compensation instructions (if required).

As the GAP shall be able to replace a superscalar processor and, hence, be able to execute the same binaries, no recompilation would be needed to make use of it. To preserve this advantage, we suggest using a post-link optimizer to apply platform-dependent code optimizations because the source code of the program to optimize is not needed in this case. Therefore, the algorithm has been designed for use in a post-link-optimizer, hence after instruction selection, register assignment, and scheduling. [1]

The algorithm is able to handle all types of control flow independent from domination- or post-domination-relations or the number of the source block's predecessors. The only exceptions are basic blocks that are targets of indirect jumps. A binary analysis together with profiling of the application delivers information about the execution frequency of basic blocks that can be selected as candidates for the modification.

The paper is organized as follows. Section 2 gives an overview of related approaches followed by Section 3 which introduces the GAP (GAP) as target processor, with focus on the mapping of code to the array of FUs and the features exploited by the proposed algorithm. The algorithm is described in Section 4 followed by an evaluation of its effects on the execution of selected benchmarks in Section 5. Section 6 concludes the paper.

2 Related Work

The GAP is a unique approach and no other code optimizations are yet suggested for it. However, similar challenges arise in compilation for superscalar or VLIW architectures as well as in hardware design. This section gives an overview.

For VLIW architectures, trace scheduling [5] is used to expose parallelism beyond basic block boundaries. The Multiflow Trace Scheduling Compiler [9] is an example for its implementation. This compiler also tries to move instructions above splits in the control flow graph but does this only if no compensation instructions are necessary. Compensation code is not used/generated due to the author's point of view that this causes too much overhead. As we show later, this is not always correct. Other scheduling techniques for speculation have been introduced and evaluated by e.g. Bergmann [2] and Mahlke [10]. They work mainly the level of superblocks. These techniques require sophisticated knowledge of the program to optimize and, therefore, cannot be applied as post-link optimizations.

For scalar and superscalar architectures, moving of instructions to preceding blocks is also suggested by Bernstein et al. [1]. The authors also state, that it is possible to move instructions speculatively, but does not give any details. They focus on moving single instructions. Similar work is done by Tirumalai et al. [13]. Although this can be iterated many times it is a difference to the work presented here, because we try to move as many instructions of a basic block as possible or reasonable at one time. Hence, the

[1] Nevertheless, additional implementation effort arises from this and it can happen that the optimization performs not as well as if implemented directly in the compiler. Somehow this is the price to pay for not having access to the source code.

overhead for repeatingly executing the analyses, e.g., updating the data dependencies, is much smaller. Beyond this, we also cope with the duplication of instructions to execute them speculatively.

Similarities also exist with *tail duplication* (e.g. [6]), which creates copies of a basic block merging them with each of its preceding basic blocks. We also try to expose parallelism by duplicating instructions but handle only the important parts of basic blocks. Hence, the program is not as heavily rewritten but the modification effort is even smaller.

As shown in Section 4 our algorithm also has parallels with software pipelining (e.g., Llosa [8]) because it can split a loop formed by a single block into two parts and rearrange them (i.e., a prologue is formed). Nevertheless, it does not reach the complexity of most algorithms for software pipelining because we assume that instructions in blocks have already been scheduled. Accordingly, we do not try to divide the source block into equal blocks in terms of approximated execution time and support only one stage.

Regarding processor design techniques, out-of-order execution as implemented by scoreboarding [11, 12] or Tomasulo's scheme [14] – both in combination with branch prediction – executes instructions speculatively, too. The hardware-effort needed to allow out-of-order execution is very high and adds new limitations e.g. for the issue unit of a processor as shown by Cotofana et al. [4].

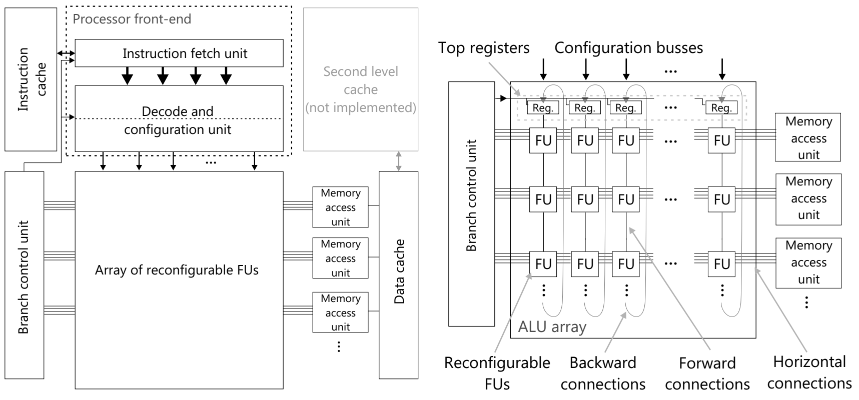
Hence, the outstanding features of the algorithm presented here are its large number of instructions which can be handled in one iteration, its ability to handle different constellations of blocks independent of the number of the source block's predecessors or the domination and/or post-domination relation between a source block and its predecessors. Beyond this, it is a post-link optimization that uses only information available from the analysis of the binary file and profiling. This causes also the struggle to modify only small parts of the program with the aim of achieving maximal effects.

3 Target Platform: Grid Alu Processor

The Grid Alu Processor (GAP) has been developed to speed up the execution of conventional single-threaded instruction streams. To achieve this goal, it combines the advantages of superscalar processor architectures, those of coarse grained reconfigurable systems, and asynchronous execution.

A superscalar-like processor front-end consisting of fetch- and decode unit is together with a novel configuration unit (see Figure 1(a)) used to load instructions and map them dynamically onto an array of functional units (FUs) accompanied by a branch control unit and several load/store units to handle memory accesses (see Figure 1(b)).

The array of FUs is organized in columns and rows. Each column is dynamically and per configuration assigned to one architectural registers. Instructions are then assigned to the column whose register matched the instructions output register. The rows of the array are used to model dependencies between instructions. If an instruction B is dependent of an instruction A than it must be mapped to a row below the row of A. This way it is possible for the in-order configuration unit to also "issue" dependent instructions without the need of complex out-or-order logic. After mapping a branch



(a) Block diagram of the GAP core (b) General organization of the ALU array

Fig. 1. Architecture of the Grid Alu Processor

the configuration unit continues with the most probable output, for this e.g. a bimodal branch predictor is used.

Execution starts in the first row of the array. It is done asynchronously between the FU and synchronously in and with the branch control unit and the L/S units. Synchronization between the FUs and the other elements of the array is controlled by tokens calculated by the configuration unit; this is similar to data-flow architectures.

When execution reaches the last row of the array, a branch is miss-predicted or there are no more columns available to map instructions, the array is cleared and the configuration units starts mapping in the first row of the array. To be able to save configurations for repeated execution all elements of the array are equipped with some memory cells which form configuration layers. The array is quasi three-dimensional and its size can be written as `columns x rows x layers`.

So, before clearing the array it is first checked if the next instruction to execute is equal to any first instruction in one of the layers. Then, in all cases, the new values of registers calculated in columns are copied to the register file at the top of the columns. If a match is found, the corresponding layer is set to active and execution continues there. If no match is found, the least recently used configuration is cleared and used to map new instructions. With this technique, the execution of loops can be accelerated very much because instructions do not have to be re-issued. This favors static speculation, too, because the maybe speculatively executed instructions are likely to be configured already on one of the configuration layers due to being often executed parts of the program.

To evaluate the architecture a cycle-accurate simulator has been developed. It uses the Portable Instruction Set Architecture (PISA), hence the simulator can execute the same program files as the SimpleScalar simulation tool set [3].

More detailed information about the processor are given by Uhrig et al. [15].

4 Static speculation

In this section, we shortly describe the algorithm to move instructions to preceding basic blocks together with the aim of the proposed program transformation

When running code generated by the default compiler (GCC 2.7.2 with `-O3`, the latest version available for SimpleScalar/PISA), in most rows of the array only a small number of FUs is configured with instructions (see Figure 2). So there are enough FUs that could be used to calculate additional results, even if they might not be needed, because they would consume only little or no additional time. To use these spare resources we try to speculatively execute instructions from following blocks.

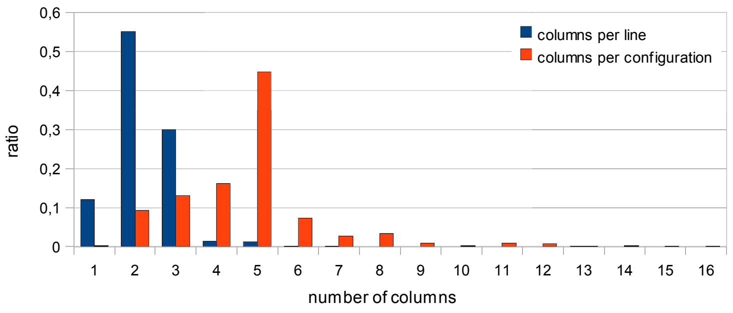


Fig. 2. Ratio between the number of used and available columns per row/configuration for benchmark `jpeg_encode` executed on GAP with 16x16x16 array

An example is demonstrated in Figure 3. It shows three basic blocks which could stand for an *if-then*-structure and have been placed into the array of the GAP. The influence of data dependencies on the instruction placement can be observed. Also, after each control flow instruction synchronization shown by a horizontal line in Figure 3 is required. In this example, all instructions of the second block can be moved to the first block and executed speculatively.

If the second block shall not be executed, i.e., the branch from the first block to the third block is taken, its effects must not be visible. In the example, `R3` is the only register which would have been modified by the speculatively executed instructions and overwrite a value which is used by subsequent instructions. Therefore, the moved instructions work on `R4`, which was initially unused, instead of the original `R3`. The overhead for executing the speculative instructions is zero in this example, because they are executed in parallel to the first block.

If the second block shall be executed, the content of `R4` must be copied to `R3`, the original target register. In the figure, this additional compensation instruction is marked by a dark box. Even if multiple compensation instructions would be required, they can be placed in the same row of the array and can be executed in parallel because they do not have any interdependencies. In our example, the overhead for the compensation instruction is zero because it is executed in parallel with the branch instruction. In the worst case, all compensation instructions can be executed in parallel because they do

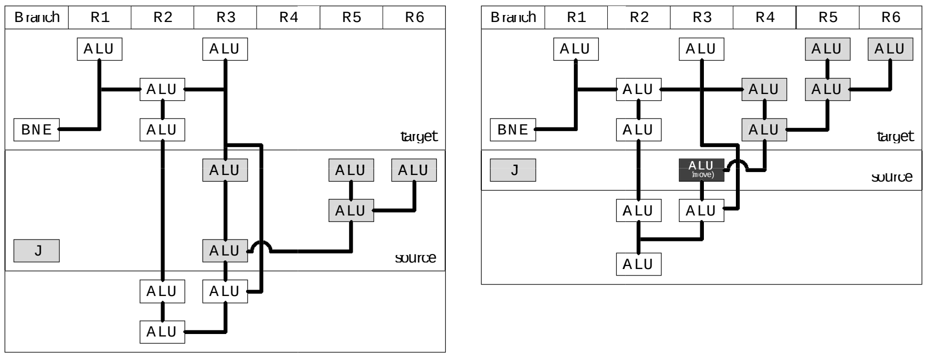


Fig. 3. Three basic blocks and their layout in the GAP array before (left) and after (right) moving of instructions

not have any dependencies on each other and, hence, consume the same time as a single additional *move* instruction.

Depending on the critical path and the resource utilization of the source and the target block, the number of rows of the array that are required to map the combined blocks should be lower. If a lower number of rows is needed for the modified blocks the average number of instructions per row increases. Hence, the number of instructions that can be executed in parallel increases resulting in a higher instructions per cycle (IPC) value.

To avoid executing too many unnecessary instructions, we move instructions only if the probability of the usage of the calculated results is above a fixed value, e.g. 30%. This value has been found to be a good tradeoff between performance and additionally executed instructions. Also we want to modify blocks only if they contribute significantly to the total program performance. Hence a block must be executed more often than a fixed boundary, e.g. more than 10 times.

Nevertheless, we cannot be sure that the total configuration length will be shorter after the modification of the blocks. This is because of the eventually required additional row for the compensation instructions, the may-be inconvenient layout of the critical paths of the blocks, and resource restrictions. For example, if memory operations are moved into a block which already uses many memory access units, than additional rows will be needed to map the moved memory operations. Hence, resource restrictions can also restrict the degree of parallelism of instructions. This problem is solved by introducing an objective function to estimate the height of the modified configuration. The additional height could also be limited by a parameter, at the moment this is set to 0.

This is mainly taken into account when selecting the number of instructions to move. it is maximized in respect to the objective function and the availability of enough registers to use them as temporary registers.

A special case is to move instructions across a loop branch. Figure 4 shows an example for such a situation. In this example, there is a block with a conditional branch to its first instruction, so it forms a very simple loop. If we shift a part of the source block to

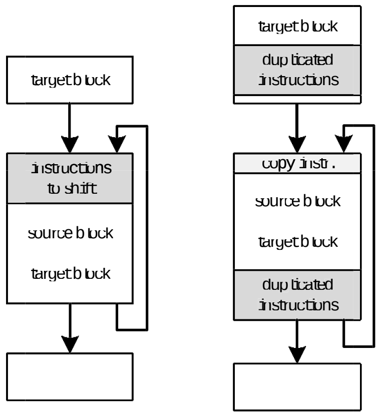


Fig. 4. A part of a block which forms a loop is moved over the loop back edge (left unmodified, right modified).

all the preceding blocks, we shift instructions from its beginning to one or more blocks with edges to the loop and also to the end of the loop. In other words, we divide the loop into two parts and copy the first one to blocks outside the loop and also move it to the end of the loop. Hence, when executing the loop, the degree of parallelism between its two parts is expected to be higher. Loop carried dependencies are also handled because the speculatively executed previously first part of the loop operates on temporary registers which do not influence the second part. The speculatively calculated results of the first part are not copied to the target registers until it is clear that the loop will be executed at least one more time.

To sum up, we expect better performance in terms of execution time. In an optimal case a better use of the FUs of the array is achieved because more columns and less rows are used. This leads to less reconfigurations of the array. Furthermore, the degree of parallelism of instructions inside the array increases.

5 Evaluation

We evaluated the static speculation algorithm using seven selected benchmarks of the MiBench Benchmark Suite [7]. We first compiled them using a standard compiler (GCC with optimizations turned on, $-O3$). Second, we performed a static analysis and applied our post-link-optimization to modify the binaries.

To be able to analyze the effects of every single modification, we set an upper bound for the maximum allowed number of modifications and increased it continuously.

The GAP, which is our target processor, is simulated by a cycle accurate simulator. It can be configured in terms of array size, configuration layers, cache size, and branch prediction. For all benchmarks, we used a *bimod* branch predictor and the identical cache configuration.

Figure 5 shows the speedup that can be gained for the seven selected benchmarks and several configurations of the GAP by the optimization technique over unchanged code. They have been distributed on the two charts according to the main

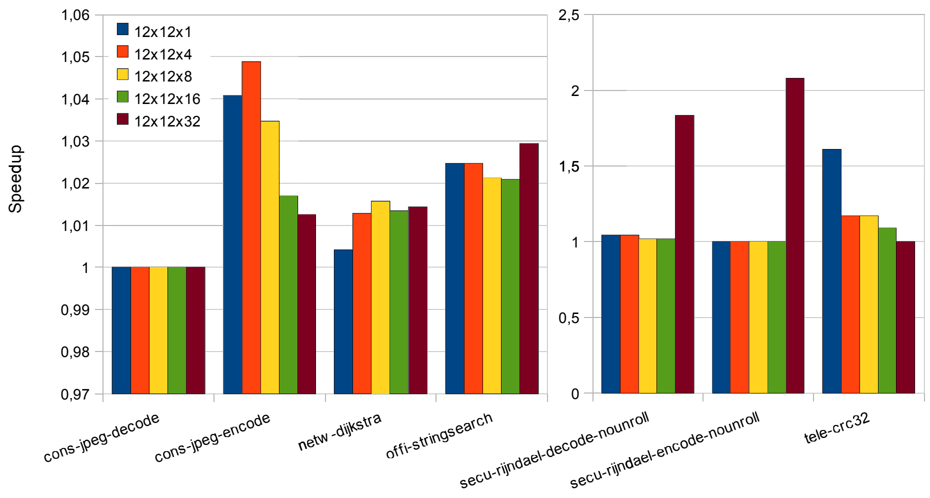


Fig. 5. Maximum speedup for selected benchmarks on GAP array of 12x12xN FUs

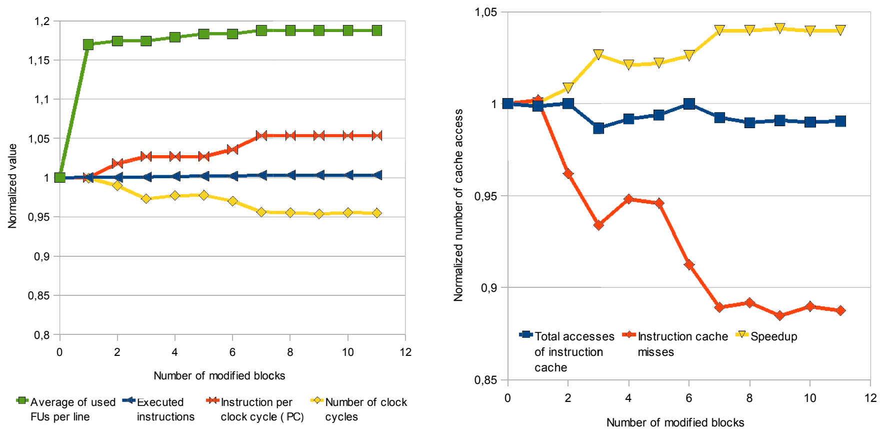
reason for the speedup. The maximum speedup of 2.08 is achieved for benchmark `secu-rijndael-encode` for GAP with an array of 12 rows, 12 columns and 32 layers (i.e. 12x12x32). The speedup is calculated as the number of total clock cycles required to completely execute the unmodified program divided by the number of clock cycles needed for the modified program executed on GAP with exactly the same configuration, too.

The speedup for the benchmarks `tele-crc32`, `secu-rijndael-decode-nounroll`, and `secu-rijndael-encode-nounroll` is caused by some effects beyond those described in 4. The main reason for the speedup is here GAP's ability to accelerate the execution of loops if the loop body fits completely into the array. This is more often the case if the program has been modified with reduction of the length of configurations as objective.

As example, GAP with the 12x12x1 configuration executing `tele-crc32` accelerates 108310 loop iterations after applying the algorithm instead of 428 without modifications. This is because the configuration of the loop is short enough after applying the static speculation optimization to map it onto the single available configuration layer. In other words, the static speculation and the hardware architecture are working hand in hand.

The more configuration layers are available the less is the impact of the optimization for `tele-crc32` and `cons-jpeg-encode`. This is because huger loops can be mapped to multiple layers anyway and, hence, the advantage of the static speculation is not as high as with only a single layer.

The acceleration of the two `secu-rijndael-*-nounroll` benchmarks is caused by the same effects. Hereby, a very long loop is mapped to multiple layers and by static speculation the number of layers required for the loop is reduced. Consequently, more layers are available to configure other code fragments.



(a) Different performance counters (b) Normalized total number of instruction cache accesses and misses as well as speedup

Fig. 6. Performance counters showing effectiveness of static speculation for `cons-jpeg-encode` for GAP with array of 12x12x1 FUs

But static speculation can also gain speedup for benchmarks without dominating loops. Figure 6(a) shows for `cons-jpeg-encode` four characteristics for different numbers of modified blocks. All values are normalized to an unmodified version of the benchmark (number of modified blocks is 0).

The modified benchmark shows a higher degree of instruction level parallelism and the execution time decreases while executing nearly the same number of instructions. Hence, we observe the intended effects.

The reason for the speedup from a hardware point of view is the better cache behaviour. Figure 6(b) shows that while the number of accesses of the instruction cache changes only marginally the number of instruction cache misses drops by more than 10 percent and speedup increases. If more configuration layers are available, the impact of the algorithm is reduced because more instructions can be accessed quickly and hence the influence of the instruction cache is decreased.

6 Conclusion

In this paper, we present an algorithm for a post-link-optimizer to increase the degree of instruction level parallelism in some parts of a program. Therefore, instructions are moved from one basic block to the preceding blocks. This modification allows in-order architectures with high fetch and execute bandwidth to execute these instructions speculatively. The speculative instructions are statically modified to use registers not required by the original program flow at that time. If the following branch is resolved the results are copied into the original target registers, if necessary. Otherwise, they are discarded. Additional hardware for speculative execution is not required. Our evaluations show a maximum speedup factor of 2.08 for a standard benchmark using GAP.

A side effect of the static speculation algorithm is that moving instructions over a loop back branch is similar to software pipelining. In the future we will focus more on this aspect. As example, it would be possible to add an additional step to reschedule the instructions of the source block before modification to increase the number of instructions that can be moved to the target blocks.

Another topic that we will examine is the real-time capability of the proposed approach. Speculative execution is also applied within out-of-order processors but, in contrast to our approach, its timing behavior is nearly unpredictable because of the dynamic nature.

References

1. D. Bernstein and M. Rodeh. Global instruction scheduling for superscalar machines. *SIGPLAN Not.*, 26(6):241–255, 1991.
2. R. A. Bringmann. *Enhancing instruction level parallelism through compiler-controlled speculation.* PhD thesis, University of Illinois, Champaign, IL, USA, 1995.
3. D. Burger and T. Austin. The simplescalar tool set, version 2.0. *ACM SIGARCH Computer Architecture News*, 25(3):13–25, June 1997.
4. S. Cotofana and S. Vassiliadis. On the design complexity of the issue logic of superscalar machines. *EUROMICRO Conference*, 1:10277, 1998.
5. J. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, 30:478–490, 1981.
6. D. Gregg. Comparing tail duplication with compensation code in single path global instruction scheduling. In *CC '01: Proceedings of the 10th International Conference on Compiler Construction*, pages 200–212, London, UK, 2001. Springer-Verlag.
7. M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and T. Brown. Mibench: A free, commercially representative embedded benchmark suite. *4th IEEE International Workshop on Workload Characteristics*, pages 3–14, December 2001.
8. J. Llosa. Swing modulo scheduling: A lifetime-sensitive approach. In *PACT '96: Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques*, page 80, Washington, DC, USA, 1996. IEEE Computer Society.
9. P. G. Lowney, S. M. Freudenberger, T. J. Karzes, W. D. Lichtenstein, R. P. Nix, J. S. O'Donnell, and J. Ruttenberg. The multiflow trace scheduling compiler. *J. Supercomput.*, 7(1-2):51–142, 1993.
10. S. A. Mahlke, W. Y. Chen, R. A. Bringmann, R. E. Hank, W. mei W. Hwu, B. Ramakrishna, R. Michael, and S. Schlansker. Sentinel scheduling: a model for compiler-controlled speculative execution. *ACM Transactions on Computer Systems*, 11:376–408, 1993.
11. J. E. Thornton. Parallel operation in the control data 6600. In *AFIPS '64 (Fall, part II): Proceedings of the October 27-29, 1964, fall joint computer conference, part II: very high speed computer systems*, pages 33–40, New York, NY, USA, 1965. ACM.
12. J. E. Thornton. *Design of a Computer—The Control Data 6600.* Scott Foresman & Co, 1970.
13. P. Tirumalai and M. Lee. A heuristic for global code motion. In *ICYCS'93: Proceedings of the third international conference on Young computer scientists*, pages 109–115, Beijing, China, China, 1993. Tsinghua University Press.
14. R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM J Res Dev*, 11(1):25–33, 1967.
15. S. Uhrig, B. Shehan, R. Jahr, and T. Ungerer. A two-dimensional superscalar processor architecture. In *The First International Conference on Future Computational Technologies and Applications, Athens, Greece*, 2009.

A Multi-Level Routing Scheme and Router Architecture to support Hierarchical Routing in Large Network on Chip Platforms

Rickard Holsmark[1], Shashi Kumar[1] and Maurizio Palesi[2]

[1] School of Engineering, Jönköping University, Sweden
{rickard.holsmark, shashi.kumar}@jth.hj.se
[2] DIIT, University of Catania, Italy
mpalesi@diit.unict.it

Abstract. The concept of hierarchical networks is useful for designing a large heterogeneous NoC by reusing predesigned small NoCs as subnets. It can also be helpful when analyzing and designing a large NoC as interconnection of subnets at a higher level of abstraction. Hierarchical deadlock-free routing is required to enable deadlock-free interconnection of sub-networks with different internal routing algorithms. In this paper we show that multi-level addressing is a cost-effective implementation option for hierarchical deadlock-free routing. We propose a two-level routing scheme, which is not only efficient, but also enables co-existence of algorithmic and table-based implementation in one router. A hierarchical view of the network simplifies addressing of network nodes and address decoding in the router. Synthesis results show that a 2-level hierarchical router design for an $8x8$ NoC, can reduce area and power requirements by up to ~20%, as compared to a router for the flat network. This work also proposes a new possibility for increasing the number of nodes available for subnet-to-subnet interfaces, while keeping the properties of hierarchical deadlock-freedom. We evaluate and discuss the communication performance in a 2-level hierarchical network for various subnet interface set-ups and traffic situations. A cycle accurate simulator has been developed and used for this purpose.

Keywords: Networks on Chip, Hierarchical Networks, Deadlock Free Routing, Router Architecture

1 Introduction

While SoCs consisting of tens of cores were common in the last decade, ITRS predicts that the next generation of many-cores SoC will contain hundreds of cores. Intel has recently announced the fabrication of a 48-core chip [1] using a Network-on-Chip as communication infrastructure. The concept of hierarchy will be very useful in designing and using such NoC platforms with growing number of cores. This concept will allow raising the level of abstraction of reuse during the design process. Instead of reusing IP cores, one can redesign large NoCs by integrating predesigned smaller NoCs as building blocks.

Whether hierarchical or not, the formation of packet deadlocks may be fatal to any network communication. To avoid this, several deadlock-free routing schemes have been proposed in literature, e.g. Turn model [2], Odd-Even [3] and Up*/Down* [4]. Deadlock freedom may be compromised when combining different networks, each with its own deadlock-free routing algorithm. For this reason, an important new issue in hierarchical NoCs is the design of deadlock-free routing algorithms. Holsmark et al. [5] proposed the concept of hierarchical deadlock-free routing and showed that if subnets are interconnected by "safe boundary" nodes, it is possible to design a deadlock-free global routing algorithm without altering any internal subnet routing algorithm. Although design and analysis of the routing algorithm was hierarchical, Holsmark et al. [5] assumed a flat implementation with a common address space for all network nodes. Non-homogeneity in such cases will often require the use of routing tables to implement the routing function.

In this work we propose that a hierarchical routing function is implemented in two levels. The higher level routing function will determine if the destination for a packet is inside or outside the local subnet. If the destination is outside the subnet, the packet is guided to a node at the boundary of the local subnet. From here the external routing function guides the packet (possibly through intermediate subnets) to a boundary node of the destination subnet. If the destination is within the subnet, the lower level routing function itself guides the packet to the destination node. The proposed structured router architecture enables significant reduction in area and power consumption. One important parameter which affects performance is the number of safe boundary nodes of a subnet. Since some routing algorithms provide very few safe nodes, we propose the concept of "safe channels" to attain higher connectivity. The performance of hierarchical routing is compared with common deadlock-free routing algorithms and the effect of varying the number of boundary nodes is explored.

Recently the topic of hierarchical NoCs has caught the attention of researchers. Several aspects have been studied, for example Bourduas et al. [6] have proposed a hybrid ring/mesh interconnect topology to remove limitations of lengthy diameter of large mesh topology networks. In [7], a hybrid mesh-ring NoC topology is proposed which is suitable for future 3-D ICs. A hierarchical on-chip approach is also taken in HiNoC [8], which offers both packet- and stream-based communication services. In HiNoC, the network has two levels of hierarchy; the asynchronously communicating mesh at the top level and an optional synchronously operating fat-tree structure attached to a mesh router network node. Deadlock-free routing in irregular networks often implies a strongly limited set of routing paths. To increase the available paths, Lysne et al. [9] developed a routing scheme, which avoids deadlock by assigning traffic into different layers of virtual channels.

2 Hierarchical Deadlock-Free Routing Algorithms

The methodology for hierarchical routing algorithms [5] enables deadlock-free interaction of independent subnet routing algorithms in hierarchical networks (sub-networks interconnected by external links). Deadlock freedom is guaranteed by acyclic channel dependency graphs (CDG) [10] [11]. In [5] it is shown that if all subnets are deadlock free and all nodes which interconnect subnets are "safe", it is possible to design a deadlock-free routing algorithm for the whole network. Whether a boundary node is "safe" or not depends on the internal subnet routing algorithm and is easily checked by analysis of internal CDG paths. If there are no paths from any internal output to any internal input of a node, it is *safe* (see Fig. 1). If such a path exists, the node is *unsafe* and may enable formation of CDG cycles with paths in other subnets. The concept of safe boundary nodes helps to design a hierarchical routing algorithm by only considering the CDG paths among boundary nodes.

2.1 Safe Channels for Increasing Connectivity

The requirement that all boundary nodes should be safe often, depending on routing algorithm, reduces the number of possible boundary nodes in a network. For deterministic routing algorithms, like XY all nodes are safe boundary nodes. Several partially adaptive algorithms provide few safe boundary nodes, e.g. an *NxN* network with Odd-Even [3], or West-First [2] provides only *N* safe boundary nodes. Negative-First [2] would in this case provide *N*+ (*N*-1) safe nodes.

To remedy this situation we propose the concept of safe channels. Given a node *n*, and an internal output channel *c* of node *n*, *c* is a safe channel if there does not exist an internal CDG path from channel *c* to any input channel of *n*. Fig. 1 illustrates the differences between unsafe nodes, safe nodes and safe channels.

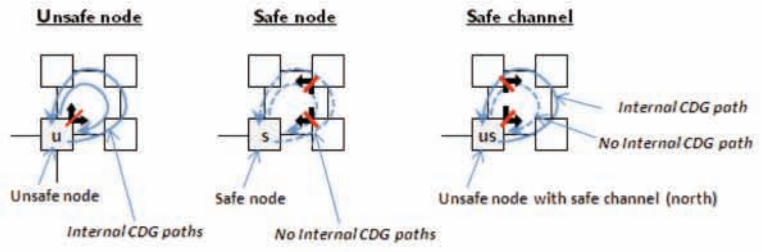


Fig. 1. Examples of unsafe boundary nodes, safe boundary nodes and safe channels

In the safe channel example, it is straightforward to see that only one of the internal output channels of node ***us*** (unsafe with safe channel) is on a CDG path to an input channel of ***us*** itself. Using this safe channel and restricting the use of the other channel would, from a deadlock-free perspective, be the same as using a safe boundary node. Note that safe channels cannot relax the requirement that there must be at least one safe boundary node in each neighboring subnet. The effect of adding unsafe nodes with safe channels is explored in the evaluation section.

3 Two-Level Routing Scheme

3.1 Addressing and Routing Protocol

Intuitively it seems that the destination address for a packet in a two-level NoC can be encoded using only two fields given in the form: [*subnet id, node id*]. However the availability of multiple boundary nodes requires that information of the destination subnet boundary node is added. Therefore a source node tags the header destination address with three fields [*subnet id, boundary node, node id*].

The routing protocol is identical for all nodes. Each node first checks whether a packet is destined to its own subnet or to an external subnet. If the destination is internal to the subnet, the packet is forwarded using the internal routing function. If the destination is in another subnet, the packet is forwarded by an external routing function (which provides paths identical to the internal routing protocols for internal link traversals).

If subnets are heterogeneous, the encoding of node address in the source subnet may differ from the encoding in the destination subnet, both with respect to size and topology. In general, the header field for node address must be adjusted according to the subnet requiring largest number of bits for node address. The size of the field for subnet addressing depends on the number of subnets.

3.2 Routing Function

The two-level routing function is partitioned into an external routing function R_G and a subnet internal routing function R_i. The internal routing function is identical to the routing function should the subnet be a stand-alone network. One feature which is enabled by two-level routing is the possibility to utilize different implementation techniques of the internal routing functions in different subnets. This implies that routers in some subnets may be table-based while other routers may implement algorithmic routing.

```
function R_H (curr, dst) returns c_out {
        if ((curr.sn=dst.sn) and (curr.addr = dst.addr)) {
                c_out:=Resource;}
        elsif (curr.sn = dst.sn) {                           //check if dest. in subnet
                c_out:=R_i (curr.addr, dst.addr);}           // local route
        else {
                c_out:= R_G (curr.addr, dst.sn, dst.bn);}    // global route
}
```

Fig. 2. Two-level routing function

Fig. 2 gives pseudo-code of the main hierarchical routing function R_H. The routing function takes *dst* which contains the destination subnet (*dst.sn*), destination boundary node (*dst.bn*) and node address (*dst.addr*). If both destination subnet and node address matches with current subnet and node address, the channel will be set to the local resource. Otherwise if the destination resides in the same subnet as the current node, the local routing function is called with the destination node address (*dst.addr*). The output channel (*c_out*) will in this case always be internal. Should the subnets not match, the external routing function is invoked with destination subnet (*dst.sn*) and boundary node (*dst.bn*). The external routing function can return both external and internal channels if current node is a boundary node. If current node is not a boundary node it will only return internal channels.

The two-level router tables are built using a similar algorithm (breadth first search) as was used for constructing flat router tables. The main difference is that only paths to destination subnets and boundary nodes are stored in the external table. This means that during the search, for each source-destination pair, the node where the last transition between different subnets was made, is stored as boundary node for the destination. This information is used for addressing by the source node. Simultaneously, the output channel from which the boundary node can be reached is stored in the router table.

Since all paths are obtained using the hierarchical deadlock-free routing methodology [5], it can be shown that the two-level scheme is deadlock free and connected as well. If the destination is in another subnet, such paths must traverse a boundary node in the source subnet and a boundary node in the destination subnet (and possibly also through some intermediate subnets).

3.3 A Small Example of Routing in Two-Level Router Networks

The following simplistic example illustrates routing in two-level networks as well as the necessity for addition of boundary node id for specifying the destination address. Consider Fig. 3 where each of the subnets S_1, S_2 and S_3 is a 2*x*2 mesh with routing algorithms XY, YX and XY respectively. The external algorithm in this case is assumed to be YX, which is the same as subnet S_2 algorithm. Nodes within subnets are addressed using (*row#*, *column#*) as shown in the figure. Boundary nodes are indicated by double border.

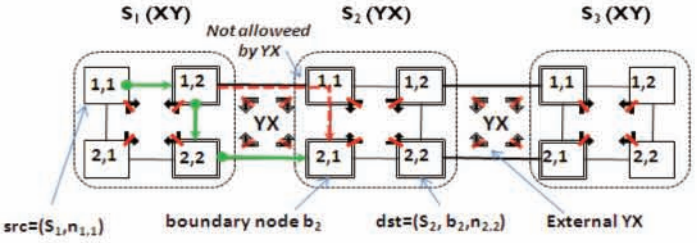


Fig. 3. Example of two-level addressing

Consider routing a message from source node $n_{1,1}$ in subnet S_1 to the destination node $n_{2,2}$ in subnet S_2. In two-level addressing, the source node is identified with subnet and node address, *src*= (S_1, $n_{1,1}$). The source appends the destination address with destination subnet, boundary node and node address, *dst*= (S_2, b_2, $n_{2,2}$). When the routing function is called in *curr*=*src*, the subnet fields do not match and the external function will be used. The external function returns the *East* channel, i.e. $R_G((n_{1,1}), S_2, b_2) = East$. Note that this is the only allowed route according to the internal XY algorithm. At node *curr*=(S_1, $n_{1,2}$), the external algorithm returns *South*. Note that *East* would also not violate the internal algorithm restriction.

However, this shows the necessity for boundary node specification. If the external address is specified using subnet id alone, it would be impossible to distinguish between destinations in row 1 and row 2 in subnet S_2. In this case, for reaching node *dst* the *only* allowed route is *South*, since the packet cannot make this turn at row 1 in subnet S_2 since both the internal algorithm and external algorithm is YX. After turning south, eventually the packet arrives at node $n_{2,1}$ in subnet S_2. Since the current subnet is now the same as the destination subnet, the node address and local algorithm is used for routing to the destination, i.e. $R_i = YX(n_{2,1}, n_{2,2}) = East$.

4 Router Architecture with Two-Level Routing Function

A block diagram of the two-level routing function in the router is given in Fig. 4. As shown in the upper part of the figure, the routing function takes destination subnet address (sn), destination boundary node address (bn), destination node address (addr), and returns the allowed output channel(s) (c_out).

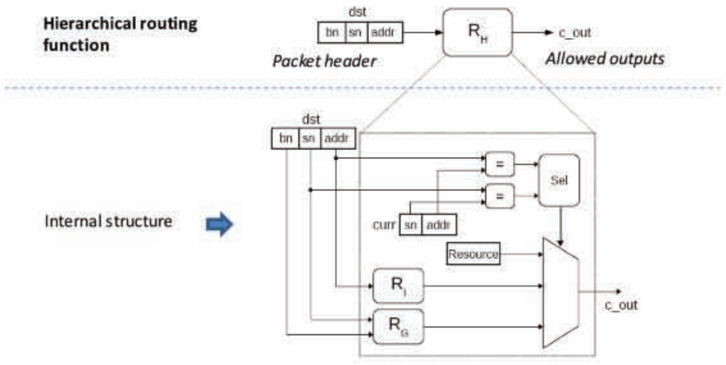
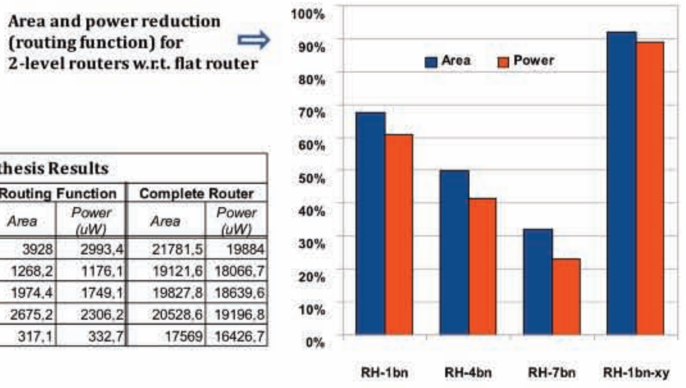


Fig. 4. Internal structure of two-level routing function

Studying the internal structure, it is seen that if both destination subnet and node addresses match with the current subnet and node addresses, the comparators will set the output of the multiplexor to *Resource*. If the subnet addresses match but not the node addresses, the destination is internal to the subnet and the output from the internal function R_i will be selected. If subnet addresses do not match, the output of the external function R_G will be selected, and the node address is not used. The table in Fig. 5 presents synthesis results from a 65nm technology library, assuming 1 GHz clock frequency. Network size is set to 64 nodes (8*x*8 mesh), which is considered as a two-level hierarchical network consisting of four equally sized subnets (4*x*4 mesh).

The results for implementation of a flat routing function are indicated by the label *RF*. Two level routing functions are synthesized for 1, 4 and 7 boundary nodes (*RH-1bn*, *RH-4bn* and *RH-7bn*). The table also provides data for two-level routing with one boundary node and algorithmic XY routing (*RH-1bn-xy*). Results are given for one routing function per router. The table gives area and power consumption separately for the routing function as well as the whole router. The main share of cost of the complete router is dominated by input buffers of 4 flits each.

Fig. 5 also summarizes the percentage of area and power reduction of the two level routing functions as compared to the flat routing function. The largest reduction for area, about 65 percent for the routing function (and ~12 percent for the complete router), is obtained by the configuration with one boundary node (*bn1)*, which only needs to store one entry per subnet. As the number of boundary nodes increase so do the resource requirements of the routing function. Power reduction is slightly less than area reduction for all configurations. Considering the algorithmic implementation with XY as local routing function, it is shown that it is possible to reduce the required area and power for the routing function by about 90 percent.



Synthesis Results					
Router	Description	Routing Function		Complete Router	
		Area	Power (uW)	Area	Power (uW)
RF	*Flat 8x8 mesh*	3928	2993,4	21781,5	19884
RH-1bn	*2L table 1 bn*	1268,2	1176,1	19121,6	18066,7
RH-4bn	*2L table 4bn*	1974,4	1749,1	19827,8	18639,6
RH-7bn	*2L table 7 bn*	2675,2	2306,2	20528,6	19196,8
RH-1bn-xy	*2L tbl/alg 1 bn*	317,1	332,7	17569	16426,7

Fig. 5. Area and power for different two-level router versions(RH-*x*bn) and a flat router (RF)

5 Performance Evaluations and Results

The evaluations compare performance of hierarchical routing with a few flat routing algorithms (XY, Up*/Down* [4]) with different configurations of boundary nodes and traffic scenarios.

5.1 Evaluation Parameters

The simulator is designed in SDL (Specification and Description Language) using Telelogic SDL and TTCN Suite 6.2 (now IBM Rational). Wormhole switching is employed, with packet size fixed at 10 flits. Routers are modeled with input buffers of size 4 and flit latency of 3 cycles per router. Packet injection rate *pir* is given in average number of packets generated per cycle. Thus *pir*=0.02 corresponds to that each node generates on average 2 packets per 100 cycles (Poisson process). For two-level routing, the simulator implements the two-level routing protocol described in Section 3, with algorithmically modeled internal subnet routing functions.

Simulations are performed with different levels of external subnet traffic w.r.t. local subnet traffic. This means that for 75% local traffic, 25% of the traffic is sent outside the source subnet. External traffic destinations are uniformly distributed over the whole network. The used subnet configurations are given in Fig. **6**(left). Each subnet exhibits a specific traffic type, which in the case of hierarchical routing is matched with a suitable routing algorithm (Subnet 1: Uniformly random, XY*;* Subnet 2: Transpose1, Negative-First; Subnet 3: Shuffle, East-First (mirrored West-First); Subnet 4: Bit Reversal, Odd-Even).

Fig. 6(right) illustrates the three configurations of boundary nodes and external routing restrictions used in the evaluations. Nodes labeled 1, and links connecting these nodes, are used in the case with one boundary node per subnet (bn1). The set-up with 4 boundary nodes per subnet (bn4) additionally uses the nodes labeled 4 and attached links. The case with 7 boundary nodes per subnet (bn7) uses, in addition to nodes labeled 1 and 4, the nodes labeled 7. The *bn1* and *bn4* set-ups utilize only safe nodes, where *bn4* represents the maximum attainable connectivity with safe nodes.

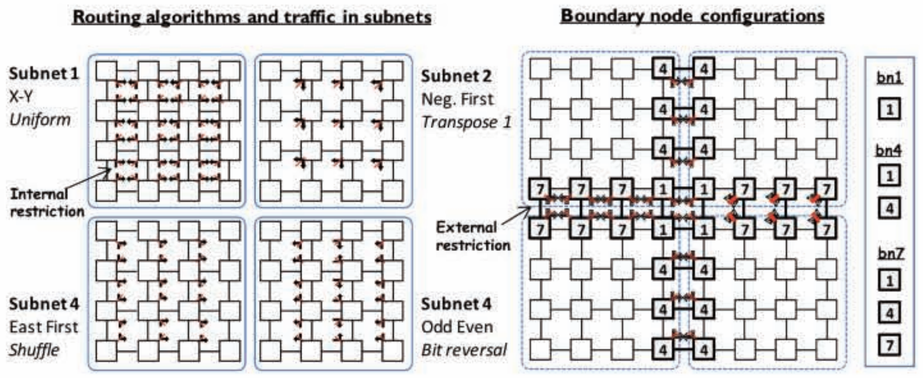


Fig. 6. Subnet and boundary node configurations

The *bn7* case allows *safe channels* of unsafe nodes in subnets 3 and 4 and, in this case, achieves the maximum connectivity of the topology. For flat algorithms, the same algorithm is used for all subnets. That is, in the case of XY this means that XY is used for routing over the whole network. Note that XY is only applicable to the *bn7* configuration. The Up*/Down* algorithm is applicable to all different configurations and a particular configuration is annotated similarly to the hierarchical (hr_bn*x*) cases, i.e. ud_bn*x*. The *latency* of a packet is the duration from when the packet was generated at the source to when its tail flit was received at the destination. Average latency is the average of all packet latencies in a simulation.

5.2 Comparison of Routing Algorithms and Boundary Node Configurations

Fig. 7(left) compares average latency of the hierarchical *hr_bn7* configuration with XY and Up*/Down* for 100, 95 and 75 percent of message subnet locality.

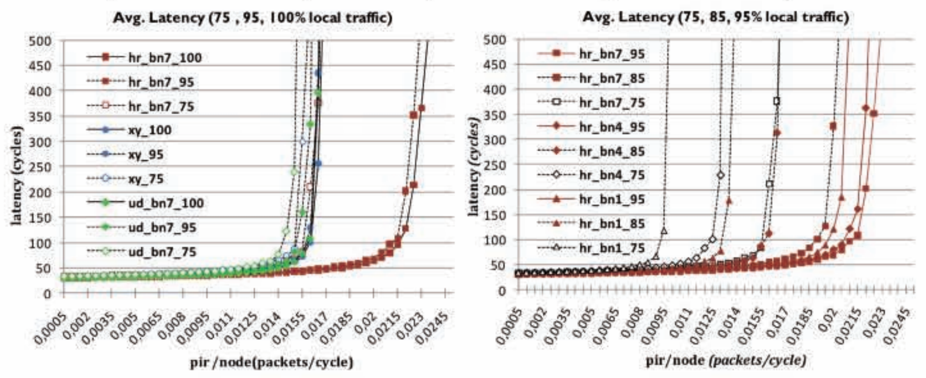


Fig. 7. Average latency: hr vs. other algorithms (left), different hr configurations (right)

As can be seen, the performance is adversely affected for all algorithms when reducing the level of internal traffic. The highest performance is, not surprisingly, obtained by *hr_bn7* with 100% local traffic (hr_bn7_100). One observation which is rather unexpected can be noted. This is seen for *hr_bn7* with 95 % local traffic (hr_bn7_95), which performs considerably better than both XY and Up*/Down* with 100% local traffic. For 75% of local traffic, the differences in performance are reduced, especially compared to XY. This is quite expected, since XY is known to be a very good algorithm for uniformly distributed traffic (which is the distribution of the external traffic).

When comparing the results in Fig. 7(left) with hierarchical routing for different configurations of boundary nodes in Fig. 7(right), it is notable that both the four- and one- boundary node hierarchical (hr_bn4_95 and hr_bn1_95 respectively) outperform XY for 95% local traffic, even though the average distances are higher due to the necessity of longer external routes. However, as the local traffic is reduced to 75% and 85%, for *hr_bn4* and *hr_bn1* respectively, the lesser connectivity of fewer boundary nodes result in notably higher average latency than XY. The very few external links in *hr_bn1* are effective bottlenecks and the congestion on these links propagates into the internal subnet traffic.

5.3 Comparison of Effects on Local and External Traffic

Fig. 8.(left) compares average latency for different algorithms and internal subnet traffic. Both *hr_bn7* and *hr_bn1* show considerably lower latency values for high load in the 95% local traffic scenario.

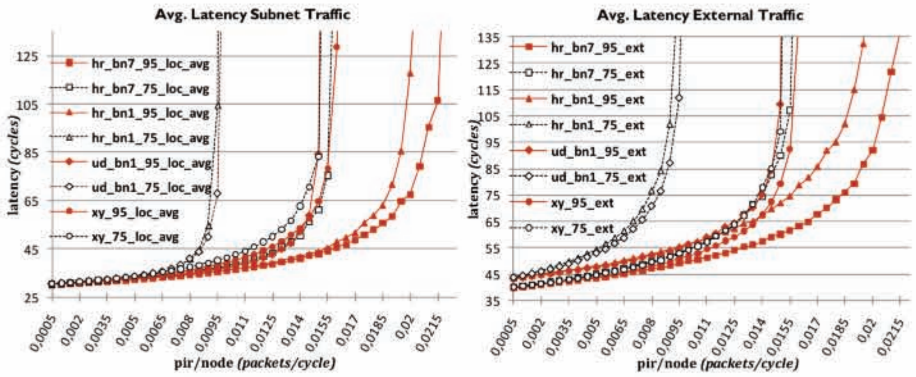


Fig. 8. Average latency for internal subnet traffic (left), external traffic (right)

Note that XY in this case follows a higher curve than Up*/Down* (ud_bn1_95) at low *pir* but improves as *pir* is increased. This indicates that Up*/Down* may have advantage of adaptive routes at lower *pir* compared to XY routing algorithm. Fig. 8(right) complement the subnet latency by showing the latency of the external traffic. The higher base latency for *ud_bn1* and *hr_bn1*, due to less number of external links is visible at both 75% and 95% of local traffic. Still, even though the base latency of *xy_95* is lower, it rapidly increases above the latency of *hr_bn1_95* at *pir* of 0.015.

6 Conclusions

In this paper we have proposed both a new routing scheme as well as a structured router design to support deadlock-free routing in a two-level hierarchical NoC. One important hierarchical network parameter is the number of safe interconnection nodes. We have compared the area and energy consumption of a router for two-level hierarchical networks for various values of this parameter. It is noticed that two-level routing is less costly as compared to a flat solution, especially when only considering the routing function. The importance of this advantage will increase with network size (due to larger tables), if buffer size is kept constant.

We have also evaluated the effect of the number of boundary nodes on communication performance. We observe that two-level hierarchical routing with maximum number of boundary nodes, in general, provides higher performance compared to flat routing algorithms. The advantage is higher when the ratio of external to local traffic is higher. For low external traffic, a single boundary node in each subnet enables routing performance comparable to flat algorithms on fully connected mesh. Multi-level routing embodies a multitude of exploration activities. For example, although the proposed 2-level scheme recursively extends itself to *n*-levels, implementation issues of such schemes will open new challenges.

References

1. Dighe, S., Hoskote, Y., Vangal, S., Finan, D., Ruhl, G., Jenkins, D., Wilson, H., Borkar, N., Schrom, G., Pailet, F., Jain, S., Jacob, T., Yada, S., Marella, S., Salihundam, P., Erraguntla, V., Konow, M., Riepen, M., Droege, G., Lindemann, J., Gries, M., Apel, T., Henriss, K., Lund-larsen, T., Steibl, S., Borkar, S., De, V., Wijngaart, R.V.D., Mattson, T., Howard, J.: A 48-Core IA-32 Message-Passing Processor with DVFS in 45nm CMOS. International SolidState Circuits Conference. 9, 58-59 (2010).
2. Glass, C., Ni, L.: The Turn Model for Adaptive Routing. Computer Architecture, 1992. Proceedings., The 19th Annual International Symposium on. pp. 278-287 (1992).
3. Ge-Ming Chiu: The odd-even turn model for adaptive routing. Parallel and Distributed Systems, IEEE Transactions on. 11, 729-738 (2000).
4. Schroeder, M., Birrell, A., Burrows, M., Murray, H., Needham, R., Rodeheffer, T., Satterthwaite, E., Thacker, C.: Autonet: a high-speed, self-configuring local area network using point-to-point links. Selected Areas in Communications, IEEE Journal on. 9, 1318-1335 (1991).
5. Holsmark, R., Kumar, S., Palesi, M., Mejia, A.: HiRA: A methodology for deadlock free routing in hierarchical networks on chip. Networks-on-Chip, 2009. NoCS 2009. 3rd ACM/IEEE International Symposium on. pp. 2–11IEEE Computer Society (2009).
6. Bourduas, S., Zilic, Z.: A Hybrid Ring/Mesh Interconnect for Network-on-Chip Using Hierarchical Rings for Global Routing. Proc. of the ACM/IEEE Int. Symp. on Networks-on-Chip (NOCS). (2007).
7. Rantala, V., Lehtonen, T., Liljeberg, P., Plosila, J.: Hybrid NoC with Traffic Monitoring and Adaptive Routing for Future 3D Integrated Chips. Digest of the Workshop on Diagnostic Services in Network-on-Chips. (2008).
8. Hollstein, T., Ludewig, R., Zimmer, H., Mager, C., Hohenstern, S., Glesner, M.: Hinoc: A Hierarchical Generic Approach for on-Chip Communication, Testing and Debugging of SoCs. VLSI-SOC: From Systems to Chips. pp. 39-54 (2006).
9. Lysne, O., Skeie, T., Reinemo, S., Theiss, I.: Layered Routing in Irregular Networks. IEEE Trans. Parallel Distrib. Syst. 17, 51-65 (2006).
10. Dally, W., Seitz, C.: Deadlock-Free Message Routing in Multiprocessor Interconnection Networks. Computers, IEEE Transactions on. C-36, 547-553 (1987).
11. Duato, J.: A new theory of deadlock-free adaptive routing in wormhole networks. Parallel and Distributed Systems, IEEE Transactions on. 4, 1320-1331 (1993).

KEYNOTE

Intel Lab's "Single-chip Cloud Computer", an IA Tera-scale Research Processor

Jim Held, Intel Fellow, Director Tera-Scale Computing Research, Intel, USA

Abstract: As part of our Tera-scale Computing Research Program, Intel Labs has created a second generation experimental "Single-chip Cloud Computer," (SCC). It contains the most Intel Architecture cores ever integrated on a silicon CPU chip – 48 cores. It incorporates technologies intended to scale multi-core processors to 100 cores and beyond, such as an on-chip network, advanced power management technologies and support for "message-passing."

Architecturally, SCC is a microcosm of a cloud datacenter. Each core can run a separate OS and software stack and act like an individual compute node that communicates with other compute nodes over the on-die packet-based network fabric, thus supporting the "scale-out" message passing programming models that have been proven to scale to 1000s of processors in cloud datacenters.

The SCC serves as an experimental platform for a wide range of software research and is currently being used by a worldwide community of academic and industry co-travelers. This talk will describe the architecture of the SCC platform and discuss its role in the broader context of our Tera-scale research.

Bio: *Jim Held is an Intel Fellow who leads a virtual team of architects conducting Tera-Scale Computing Research in Intel Labs. Since joining Intel in 1990, he has led research and development in a variety of Intel's labs concerned with media and interconnect technology, systems software, multi-core processor architecture and virtualization. He earned a Ph.D. (1988) in Computer and Information Science at the University of Minnesota.*

the 4th Workshop on
Highly Parallel Processing
on a Chip