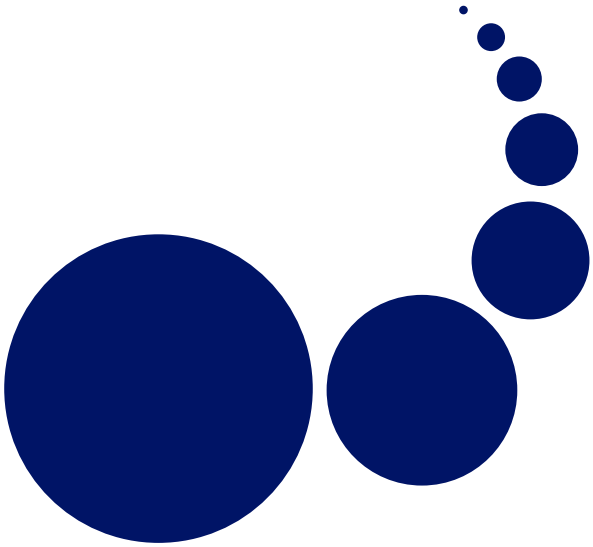
# SCALABLE COMPUTING
## Practice and Experience

## Special Issues:

### Practical Aspects of High-Level Parallel Programming
Editors: Anne Benoît and Frédéric Loulergue

### High Performance Reconfigurable Computing
Editors: Dorothy Bollman, Javier Díaz and Francisco Rodriguez-Henriquez



Volume 8, Number 4, December 2007

ISSN 1895-1767

SUBSCRIPTION INFORMATION: please visit http://www.scpe.org

Scalable Computing: Practice and Experience

Volume 8, Number 4, December 2007

## TABLE OF CONTENTS

Book Reviews:

Scalable Computing: Practice and Experience
Volume 8, Number 4, pp. i–iv. http://www.scpe.org
ISSN 1895-1767


# EDITORIAL: THE PRESENT AND THE FUTURE OF RECONFIGURABLE DEVICES FOR SPACE APPLICATIONS

**1. Introduction.** Space missions present important scientific and technological challenges. Electronic systems used in space applications require low power consumption, small weight and small size. Designers should take into account that each gram sent to space requires a very large amount of money. Furthermore, the electronics must be highly reliable in order to work for years in hostile environments [1]. One of the main inconveniences with instruments designed for space is that they must be resistant to radiation. This requirement depends mostly on the type of mission and its duration.

FPGAs began space missions by acting as "glue" logic. Only one chip could do the work of several more (controllers, clock divisors, decoders...). Currently, FPGAs technologies are more and more versatile and reconfigurable devices can be used as complex controllers, or as the main control system, combining several system functions on a single chip, including microprocessors functionality and small size memory. Because of the special environmental conditions in these kinds of missions, the designers are forced to utilize suitable devices adapted to support radiation and extreme temperatures. As a consequence, it is not possible to use the latest technologies in a space mission, because the use of these technologies in a space environment use is complex. In spite of this, the number of space missions has been increasing, and electronics manufacturers now have sections that are dedicated to the aerospace market. This is one of the reasons that has motivated the rapid evolution in the use of FPGAs devices.

In this editorial we present the current state of reconfigurable hardware and the feasible future evolution of this technology in this field. We review the configurable devices that are suitable for flight. We give some examples of missions where FPGAs have been used successfully and finally, we draw some conclusions about the use of these devices.

**2. Review of FPGAs for space applications.** As commented before, the devices onboard a spacecraft must be light, small size, with very low power consumption, radiation resistant, and as a main requirement, they must guarantee very high reliability. The choice of a suitable device depends mostly on the kind of mission, the environment and on the life span. The requirements for a satellite that will be working in space for one year are radically different from requirements for a spacecraft going to Mars and working for six years [2].

The farther we go, the more difficult it is to find suitable devices. This motivates the choice of devices that are known to be reliable and have a good track record over multiple missions. An example is the Intel 8086 processor, which is still used for space missions like the Mars Pathfinder and GIADA in Rosetta [3]. The same reasoning can be applied to FPGA devices. The ones that we can use are unfortunately not those that represent the latest technology advances. Nevertheless, FPGAs that are suitable for space have evolved and today designers can find products with a million usable equivalent gates that are suitable for flight.

Space agencies (NASA and ESA) offer public parts lists with preferred devices to go onboard a space mission [4, 5]. But these lists are not completely exhaustive, because they do not have all the space devices. In the lists, we can see European devices are nonvolatile, whereas NASA preferred devices are reprogrammable Few reprogrammable devices have been used on European spacecrafts due to their sensitivity to single event upsets (SEUs). But recently, FPGA vendors have begun to develop SEU mitigation techniques to make their devices usable in space applications [6].

There are two main FPGA manufacturers: Actel and Xilinx, although there are more vendors offering FPGAs for flight: Aeroflex and Atmel. In this editorial we focus on Actel and Xilinx. Actel offers non reprogrammable FPGAs for space applications, with the SX and Axcelerator families. They are antifuse-based devices [7], non reprogrammable, but radiation tolerant, with total ionization dose (TID) up to 300K rads. The main features are 250MHz system performance and from 48k up to 108k system gates for the SX family, and 350MHz system performance and 250k to 4 million system gates for the Axcelerator family. The advantages of these devices are the availability of prototyping using non-qualified devices, and Single Even upset (SEU) mitigation techniques such as triple modular redundancy (TMR) implemented automatically on the chip. These devices also feature error detection and correction (EDAC) for internal memory, and they are "power on and go", that is, they do not need other components to start working, because the program is already on the device when the system is powered on. In reprogrammable devices, it is necessary to add some other chip containing the program, so when the system is powered on, the program has to be written onto the FPGA.

Xilinx offers reprogrammable devices based on the QPro-R Virtex and QPro-R Virtex II devices [8]. They are claimed to be powerful and flexible alternatives to mask-programmed gate arrays. The main features are reprogrammability, 200 MHz system performance and system gates from 300K to 1M gates for Virtex and 300 MHz system performance and 1M to 6M gates for Virtex II devices. The TID is 100KRads for the first family and 200KRads for the second one. The advantage of these devices is the reprogramability itself and, because there are no significant differences compared to the commercial versions, the standard devices could be used for reducing system prototyping cost.

Some FPGA vendors (i.e. Atmel and Xilinx) provide the possibility of migrating FPGA designs to ASICs. This can be used, for example, in a constellation of satellites.

Traditionally only antifuse devices (such as the one provided by Actel) have been used for the space missions. The applicability of reprogrammable devices (such as the one offered by Xilinx) has started to be in use in the last years [9].

If we compare these two types of FPGAs (antifuse and reprogrammable devices), there are also other features that should be taken into account. On the one hand, antifuse families offer TRM implementation in an automatic way, and SEU protection for internal registers, whilst reprogrammable devices do not [7, 8]. On the other hand, reprogrammable devices offer a high integration density. In addition, they provide advanced interfacing solutions with a broad range of electrical standards, clock management features and internal memory capacity [10]. Nevertheless, this family requires an external device to load the bitstream at power on. This means more components and an additional risk during a very critical moment like the start up of the system. As a consequence, the choice between these two alternative devices depends on the application, and should be taken after a careful analysis of the mission requirements.

**3. FPGAs used in space missions.** In this section we will review some space missions that have successfully used FPGAs, and are already on space.

The Institute de Astrophysics of Andalucía has been involved in several space missions; the most extended experience is with OSIRIS and GIADA instruments, and we will refer these.

There are many differences between designing for space and designing for another application. Space designers must take into account the final implementation of their designs, that is, the logic generated from their code. This can be different from working in other area, where only the final result is significant. In space, the logic is important because of the necessity of preventing SEU and other radiation effects, making the instrument as safe as possible [11]. The OSIRIS instrument of the Rosetta mission is an optical spectroscopic and Infrared remote imaging system. It has several parts, and FPGAs have been used to improve the design. In the CCD readout box, there are two FPGAs, one for the clocks, and the other to handle high speed serial [12]. The mechanism controller board (MCB) of the two OSIRIS optical cameras also has two FPGAs to implement the digital control circuits [13]. Data gathering, packaging, transmission and command decoding are performed within the FPGAs (Actel RH1280). In this case, FPGAs have made significant advances, allowing the reduction of mass and making the design simpler. In fact. the motor controller FPGA makes an "a dco" controller optimized for this instrument with different and very specific functionalities.

The GIADA instrument, also of the Rosetta mission is a grain impact analyzer and dust accumulator experiment. It uses one FPGA to control the data obtained from the proximity electronics [3, 2]. This makes possible the use of an Intel 8086 processor, because there are tasks done by the FPGA, leaving the processor free to do other operations. All the FPGAs used in GIADA are Actel, RH1280. The use of Actel RH1280 allows significant reduction of power consumption and also makes the design simpler, because one chip implements the same function as a microcontroller with all the peripheries and EDAC techniques for mitigating SEU. The mass and size are obviously small in this solution.

Spirit and Discovery missions utilize XQVR1000 and XQR4062XLs from Xilinx. The XQR4062XLs were used during the descent and landing of the rovers on the surface of Mars, while XQVR1000s were used to control all of the brushed DC and stepper motors for the wheels, steering, antennas, camera, and other instruments on the rovers themselves [9].

Almost all future space missions that are presently being developed are going to use FPGAs onboard. An example of those in which the IAA is working is, for example, MEDUSA, the acronym for martian environmental dust systematic analyzer. It will analyze the dust on the surface of Mars, onboard the EXOMARS mission [14]. The main electronics will include an FPGA from Actel, with a programmable finite state machine inside, the

clock generation control, and all the controllers for communication, digital acquisition, data handling, and an embedded overall controller.

Other future projects include: (1) the SOPHI experiment which willl be onboard the solar orbiter mission. It is a polarimetric helyospheric image magnetograph for studying the sun. (2) the SODA experiment which will be onboard the Solar Orbiter mission also. It is an instrument devoted to study cosmic dust. (3) BELA is a Laser Altimeter going onboard the Bepi-Colombo mission. In all these missions FPGAs will play a relevant role making the system electronics simple and of high reliability at the same time.

**4. Conclusions.** Design for space environment is complicated due to the very high requirements. Factors such as weight, power and size must be taken into account. Among all of these, the most important requirement is reliability and one of the most important problems is the effect of radiation. For these reasons it is necessary to use devices especially built for space environments. The manufactures have specific areas for the aerospace market and this has motivated space agencies to have preferred parts lists to help designers to choose the devices to use onboard.

In the past, devices were more primitive. Therefore, FPGAs were used as glue logic. To use them, designers utilized schematic as a popular design technique. Nowadays, configurable devices can be employed as many complex systems on just one chip. We can find devices with a million gates on the market, and it is not the practice to use schematic, because of the complexity. For these reasons, designers are turning to hdl languages, which offer less control of the logic used, but yield a design simpler than a scheme.

There are two main types of configurable devices. One time programmable devices (OTP) like that offered by Actel, and reprogrammable devices like that offered for Xilinx. Both have advantages and disadvantages, and care should be taken before choosing one device.

Companies like Actel now have new FPGA devices, based on flash technology. These devices have all the advantages from reprogramability, and from "Power and Go". They are still not qualified for space, but in the future it is reasonable to believe they will be. In addition, new technological advances such as dynamic reconfiguration can represent an important step in the future of space missions. This would offer the possibility of making substantial changes in hardware on the fly, giving options to add new functionalities or function modes, and replacing software patches.

One important feature that we believe will be of relevant importance for the future of FPGA devices in space mission is their capability of developing programmable system-on-chip (PSoC) in just one single device. Nowadays there are FPGA synthesizable versions of processors such as the Leon processor [15], which has been qualified for space applications [16]. Codesign techniques could be applied to split the different tasks in such a way that algorithm and scheduling tasks are implemented in the processor while low level controlling tasks are developed using the device logic gates. Moreover, traditional glue logic tasks could still be done on the same chip. For demanding extensive computations, the FPGA logic could be used to developing a custom coprocessor to help the processor in these computations, making possible full parallel processing. This feature is not achievable using a single microprocessor., But using programmable hardware, we can do several tasks simultaneously.

FPGAs play a relevant role in space missions, and it is reasonable to believe that in the future the use of these devices will be even more extensive. Their capabilities to act as glue logic and as overall controllers, all in a single chip, make them the perfect devices to go onboard space craft. We believe FPGA devices will be the predominant digital device for space missions in the near future.

REFERENCES

[1] J. Rodríguez, L. Costillo, *Docencia Tercer Ciclo. Master en ingeniería de computadores: Dpmto Arquitectura y Tecnología de Computadores,* Univ. Granada, 2006.

[2] López-Jiménez A., *Aplicación de dispositivos FPGA a la instrumentacióespacial: los instrumentos GIADA y OSIRIS de la misión Rosett,* Granada, Doctoral dissertation, March 2006.

[3] *The Grain Impact Analyser and Dust Accumulator (GIADA) Experiment for the Rosetta Mission: Design, Performances and First Results,* L. Colangeli, J. Rodríguez, A. López-Jiménez et al (eds), Springer Netherlands. 1–4, 2007, Space Science Reviews, Vol. 128, pp. 803–821. DOI 10.1007/s11214-006-9038-5.

[4] Jet Propulsion Laboratory (NASA), Available Parts. JPL WebSite. `http://parts.jpl.nasa.gov/data/sources.htm` January 21, 2008.

[5] ESCIES, *European Space Components Information Exchange System. European Preferred Part Lists. ESCIES WebSite,* November 16, 2007, `https://escies.org/ReadArticle?docId=166`

[6] Habinc Sandi, *Suitability of reprogrammable FPGAs in space applications (compilation),* FPGA-002-01, Report ESA contract No. 15102/01/NL/FM(SC) CCN-3, September 2002.
[7] ACTEL, Actel Products and services: Devices. Actel Website. http://www.actel.com/products/devices.aspx January 17, 2008.
[8] XILINX, *Xilinx products & services: Aerospace and Defense,* Xilinx website, January 17, 2008: http://www.xilinx.com/products/silicon_solutions/aero_def/index.htm
[9] Ratter David, *FPGAs on Mars,* Issue 50, XCell Journal Online, 2004.
[10] *QPro Virtex 2.5V QML Datasheet,* XILINX. 2001. DS002 (v 1.5).
[11] Habinc Sandi, *Lessons Learned from FPGA Developments,* FPGA 001-01, Report ESA contract, No. 15102/01/NL/FM(SC) CCN-3, September 2002.
[12] *OSIRIS—The Scientific Camera System Onboard Rosetta,* H. U. Keller, J. Rodríguez, A. C. López-Jiménez et al (eds), Springer Netherlands. 1–4, 2007, Space Science Reviews, Vol. 128, pp. 433–506. 0038-6308 (Print) 1572–9672 (Online).
[13] *Mechanism controller system for the optical spectroscopic and infrared remote imagin system instrument on board the Rosetta space mission,* J. M. Castro Marín, V. J. G. Brown, A. C. López-Jiménez, J. Rodríguez Gómez (eds), American Institute of Physics, 5, May 2001, Review of Scientific Instruments, Vol. 72, pp. 2423–2427.
[14] European Space Agency (ESA), *Aurora Exploration Programme: Technologies: Dust and Water Vapour Instrument Suite. ESA Website,* http://www.esa.int/esaMI/Aurora/SEMIWP7X9DE_0.html January 25, 2008.
[15] Gaisler Research, *Leon3 Processor,* Gaisler Research WebSite. January 21, 2008: http://www.gaisler.com/cms/index.php?option=com_content&task=view&id=13&Itemid=53
[16] ESA, *ESA-Microelectronics-Leon2 FT,* ESA-Microelectronics WebSite. January 21, 2008: http://www.esa.int/TEC/Microelectronics/SEMUD70CYTE_0.html

Beatriz Aparicio del Moral,
Julio Rodríguez-Gómez,
Antonio C. López Jiménez.

*Institute of Astrophysics of Andalucía,*
*Intrumental and Technological Development Unit,*
*Higher Council of Scientific Research (CSIC).*

{bea, julio, antonio}@iaa.es


SCALABLE COMPUTING: PRACTICE AND EXPERIENCE ISSN 1895-1767
Volume 8, Number 4, p. v. http://www.scpe.org 

# INTRODUCTION TO THE SPECIAL ISSUE: PRACTICAL ASPECTS OF HIGH-LEVEL PARALLEL PROGRAMMING

As Computational Science applications are more and more demanding, both on computing power and complexity of development, it is necessary to provide programming languages and tools which offer a high degree of abstraction to ease the programming of parallel, distributed and grid computing systems. Moreover, these high-level languages are very often based on formal semantics. It is thus possible to certify the correctness of critical parts of the applications.

This special issue of *Scalable Computing: Practice and Experience* presents recent work of researchers in these fields. These articles are a selection of extended and revised versions of papers presented at the third international workshop on Practical Aspects of High-Level Parallel Programming (PAPP), affiliated to the International Conference on Computational Science (ICCS 2006). The PAPP workshop is aimed both at researchers involved in the development of high level approaches for parallel and grid computing and computational science researchers who are potential users of these languages and tools. The topics of the PAPP workshop include high-level models (CGM, BSP, MPM, LogP, etc.) and tools for parallel and grid computing; high-level parallel language design, implementation and optimisation; functional, logic, constraint programming for parallel, distributed and grid computing systems; algorithmic skeletons, patterns and high-level parallel libraries; generative (e.g. template-based) programming with algorithmic skeletons, patterns and high-level parallel libraries; applications in all fields of high-performance computing (using high-level tools); and benchmarks and experiments using such languages and tools.

The Java programming language increases the productivity of programmers for example by taking care of memory management, by providing a wide collection of data structures (safer with the recent introduction of genericity in the language), and many other features. JIT compilation techniques make Java virtual machines quite efficient. Thus Java is an interesting choice as a basis for high-level parallelism. The two papers selected from the PAPP 2006 workshop focus on parallel programming with Java. In their paper *MUSKEL: an expandable skeleton environment*, Marco Aldinucci, Marco Danelutto and Patrizio Dazzi propose a new skeleton language, Muskel, based on data flow technology. It implements both the usual predefined skeletons and user-defined parallelism exploitation patterns. Muskel is a pure Java implementation, and relies on the annotation and RMI facilities of Java. *A Buffering Layer to Support Derived Types and Proprietary Networks for Java HPC* by Mark Baker, Bryan Carpenter and Aamir Shafi, presents a new MPI-like binding for Java. MPJ Express combines two strength which can usually not be found in other MPI bindings for Java: it supports the implementation of derived datatypes and it is implemented in pure Java. The buffering layer used to provide these features also gives a way to implement efficient proprietary networks communication devices.

We would like to thank all the people who made the PAPP workshop possible: the organizers of the ICCS conference, the other members of the programme committee: Marco Aldinucci (CNR/Univ. of Pisa, Italy), Olav Beckmann (Imperial College London, UK), Alexandros Gerbessiotis (NJIT, USA), Stephen Gilmore (Univ. of Edinburgh, UK), Clemens Grelck (Univ. of Luebeck, Germany), Christoph Herrmann (Univ. of Passau, Germany), Zhenjiang Hu (Univ. of Tokyo, Japan), Casiano Rodriguez Leon (Univ. La Laguna, Spain), Alexander Tiskin (Univ. of Warwick, UK). We also thank the referees external to the PC for their efficient help. Finally we thank all authors who submitted papers for their interest in the workshop, the quality and variety of the research topics they proposed.

Anne Benoît,
*Laboratoire d'Informatique du Parallélisme,*
*Ecole Normale Supérieure de Lyon,*
*46 Allée d'Italie,*
*69364 Lyon Cedex 07, France.*

Frédéric Loulergue,
*Laboratoire d'Informatique*
*Laboratoire d'Informatique Fondamentale d'Orléans (LIFO),*
*University of Orléans,*
*rue Léonard de Vinci, B. P. 6759,*
*F-45067 Orléans Cedex 2, France.*

Scalable Computing: Practice and Experience
Volume 8, Number 4, pp. vii–viii. http://www.scpe.org
ISSN 1895-1767


# INTRODUCTION TO THE SPECIAL ISSUE: HIGH PERFORMANCE RECONFIGURABLE COMPUTING

High performance reconfigurable computing has become a crucial tool for designing application-specific processors/cores in a number of areas. Improvements in reconfigurable devices such as FPGAs (field programmable gate arrays) and their inclusion in current computing products have created new opportunities, as well as new challenges for high performance computing (HPC).

An FPGA is an integrated circuit that contains tens of thousands of building blocks, known as configuration logic blocks (CLBs) connected by programmable interconnections. FPGAs tend to be an excellent choice when dealing with algorithms that can benefit from the high parallelism offered by the FPGA fine-grained architecture. In particular, one of the most valuable features of FPGAs is their reconfigurability, i.e., the fact that they can be used for different purposes at different stages of a computation and they can be, at least partially, reprogrammed at run-time.

HPC applications with reconfigurable computing (RC) have the potential to deliver enormous performance, thus they are especially attractive when the main design goal is to obtain high performance at a reasonable cost. Furthermore they are suitable for use in embedded systems. This is not the case for other alternatives such as grid-computing.

The problem of accelerating HPC applications with RC can be compared to that of porting uniprocessor applications to massively parallel processors (MPPs). However, MPPs are better understood to most software developers than reconfigurable devices. Moreover, tools for porting codes to reconfigurable devices are not yet as well developed as for porting sequential code to parallel code. Nevertheless, in recent years considerable progress has been in developing HPC applications with RC in such areas as signal processing, robotics, graphics, cryptography, bioinformatics, evolvable and biologically-inspired hardware, network processors, real-time systems, rapid ASIC prototyping, interactive multimedia, machine vision, computer graphics, robotics, and embedded applications, to name a few. This special issue contains a sampling of the progress made in some of these areas.

In the first paper, Lieu My Chuong, Lam Siew Kei, and Thambillai Srikanthan propose a framework that can rapidly and accurately estimate the hardware area- time measures for implementing C-applications on FPGAs. Their method is able to predict the delays with average accuracy of the 97%. The estimation computation of this approach can be done in the order of milliseconds. This is an essential step to facilitate rapid design exploration for FGPA implementations and significantly helps in the implementation of FPGA systems using high-level description languages.

In the second paper, T. Hausert, A. Dsu, A. Sudarsanam, and S. Young design an FPGA based system to solve linear systems for scientific applications. They analyze the FPGA performance per wait (MFLOPS/W) and compare the performance with microprocessor-based approaches. Finally as the main outcome of this analysis, they propose helpful recommendations for speeding up FPGA computations with low power consumption.

In the third paper, S. Mota, E. Ros, and F. de Toro describe a computing architecture that finely pipelines all the processing stages of a space variant mapping strategy to reduce the distortion effect on a motion-detection based vision system. As an example, they describe the results of correcting perspective distortion in a monitoring system for vehicle overtaking processes.

In the fourth paper, Sadaf R. Alam, Pratul K. Agarwal, Melissa C. Smith, and Jeffrey S. Vetter describe an FPGA acceleration of molecular dynamics using the Particle-Mesh Ewald method. Their results show that time-to-solution of medium scale biological system simulations are reduced by a factor of 3X and they predict that future FPGA devices will reduce the time-to-solution by a factor greater than 15X for large scale biological systems.

In the fifth paper, Nazar A. Saqib presents a space complexity analysis of two Karatsuba-Ofman multiplier variants. He studies the number of FPGA hardware resources employed by those two multipliers as a function of the operands' bitlength. He also provides a comparison table against the school (classical) multiplier method, where he shows that the Karatsuba-Ofman method is much more economical than the classical method for operand bitlengths greater than thirty two bits. The complexity analysis presented in this paper is validated experimentally by implementing the multiplier designs on FPGA devices.

The help of the follow reviewers, who ensured the quality of this issue, is gratefully acknowledged:

- Mancia Anguita, University of Granada, Spain,
- Beatriz Aparico, Andalucia Astrophysics Institute, CSIC, Spain,

- AbdSamad Benkrid, Queen's University, Northern Ireland,
- Eunjung Cho, Georgia State University, USA,
- Nareli Cruz-Cortés,CIC-IPN, Mexico,
- Sergio Cuenca, University of Alicante, Spain,
- Jean-Pierre Deschamps, University Rey Juan Carlos, Spain,
- Edgar Ferrer, University of Puerto Rico at Mayaguez,
- Luis Gerardo de la Fraga, CINVESTAV-IPN, Mexico,
- Antonio Garcia, University of Granada, Spain,
- Javier Garrigos, University of Cartagena, Spain,
- Miguel Angel León-Chávez, BUAP, Mexico,
- Adriano de Luca-Pennacchia CINVESTAV-IPN, Mexico,
- Antonio Martinez, University of Alicante, Spain,
- Christian Morillas, University of Granada, Spain,
- Daniel Ortiz-Arroyo, Aalborg University, Denmark.

Dorothy Bollman,
*University of Puerto Rico at Mayaguez.*

Javier Díaz,
*University of Granada, Spain.*

Francisco Rodriguez-Henriquez,
*Center for Research and Advanced Study,*
*National Polytechnical Institute, Mexico.*

Scalable Computing: Practice and Experience
Volume 8, Number 4, pp. 325–341. http://www.scpe.org
ISSN 1895-1767


# MUSKEL: A SKELETON LIBRARY SUPPORTING SKELETON SET EXPANDABILITY*

MARCO ALDINUCCI† AND MARCO DANELUTTO† AND PATRIZIO DAZZI‡

**Abstract.** Programming models based on algorithmic skeletons promise to raise the level of abstraction perceived by programmers when implementing parallel applications, while guaranteeing good performance figures. At the same time, however, they restrict the freedom of programmers to implement arbitrary parallelism exploitation patterns. In fact, efficiency is achieved by restricting the parallelism exploitation patterns provided to the programmer to the useful ones for which efficient implementations, as well as useful and efficient compositions, are known. In this work we introduce `muskel`, a full Java library targeting workstation clusters, networks and grids and providing the programmers with a skeleton based parallel programming environment. `muskel` is implemented exploiting (macro) data flow technology, rather than the more usual skeleton technology relying on the use of implementation templates. Using data flow, `muskel` easily and efficiently implements both classical, predefined skeletons, and user-defined parallelism exploitation patterns. This provides a means to overcome some of the problems that Cole identified in his skeleton "manifesto" as the issues impairing skeleton success in the parallel programming arena. We discuss fully how user-defined skeletons are supported by exploiting a data flow implementation, experimental results and we also discuss extensions supporting the further characterization of skeletons with non-functional properties, such as security, through the use of Aspect Oriented Programming and annotations.

**Key words.** algoritmical skeletons, data flow, structured parallel programming, distributed computing, security.

**1. Introduction.** Structured parallel programming models provide the user (programmer) with native high-level parallelism exploitation patterns that can be instantiated, possibly in a nested way, to implement a wide range of applications [13, 23, 24, 8, 6]. In particular, such programming models do not allow programmers to program parallel applications at the "assembly level", i. e. by directly interacting with the distributed execution environment via communication or shared memory access primitives and/or via explicit scheduling and code mapping. Rather, the high-level native, parametric parallelism exploitation patterns provided encapsulate and abstract from these parallelism exploitation related details. For example, to implement an embarrassingly parallel application processing all the data items in an input stream or file, the programmer simply instantiates a "task farm" skeleton by providing the code necessary to process (sequentially) each input task item. The system, either a compiler and run time tool based implementation or a library based one, takes care of devising the appropriate distributed resources to be used, to schedule tasks on the resources and to distribute input tasks and gather output results according to the process mapping used. By contrast, when using a traditional system, the programmer has usually to explicitly program code for distributing and scheduling the processes on the available resources and for moving input and output data between the processing elements involved. The cost of this appealingly high-level way of dealing with parallel programs is paid in terms of programming freedom. The programmer is normally not allowed to use arbitrary parallelism exploitation patterns, but he must use only the ones provided by the system, usually including all those reusable patterns that happen to have efficient distributed implementations available. This is aimed mainly at avoiding the possibility for the programmer to write code that could potentially impair the efficiency of the implementation provided for the available, native parallel patterns. This is a well-known problem. Cole recognized its importance in his "manifesto" paper [13].

In this work we discuss a methodology that can be used to provide parallel application programmers with both the possibility of using predefined skeletons in the usual way and, at the same time, the possibility of implementing their own, additional skeletons, where the predefined ones do not suffice. The proposed methodology, which is based on data flow, preserves most of the benefits typical of structured parallel programming models. According to the proposed methodology, predefined, structured parallel exploitation patterns are implemented by translating them into data flow graphs executed by a scalable, efficient, distributed macro data flow interpreter (the term *macro* data flow refers to the fact that the computation of a single data flow instruction can be a substantial computation). User-defined, possibly unstructured parallelism exploitation patterns can be programmed by explicitly defining data flow graphs. These data flow graphs can be used in the skeleton system in any place where predefined skeletons can be used, thus providing the possibility of seamlessly integrating both kinds of parallelism exploitation within the same program.

*This work has been partially supported by Italian national FIRB project no. RBNE01KNFP GRID.it and by the FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265).
†Dept. Computer Science—Univ. of Pisa & Programming Model Institute, CoreGRID
‡ISTI–CNR, Pisa & IMT—Institute for Advanced Studies, Lucca & Programming Model Institute, CoreGRID

User-defined data flow graphs provide users with the possibility of programming new skeletons. However, in order to introduce a new skeleton, users need concentrate only on the data flow within the new skeleton, rather than on the implementation issues typically related to the efficient implementation of structured parallelism exploitation patterns. This greatly improves the efficacy of the parallel application development process as compared to classical parallel programming approaches such as MPI and OpenMP that instead provide users with very low level mechanisms and give them complete responsibility for efficiently and correctly using these mechanisms to implement the required parallelism exploitation patterns.

After describing how user defined skeletons are introduced and supported within our experimental skeleton programming environment, we will also briefly discuss other tools we are currently considering to extend the prototype skeleton environment. These tools extend the possibility for users to control some non-functional features of parallel programs in a relatively high-level way. In particular we will introduce the possibility of using Java 1.5 annotations and AOP (Aspect-Oriented Programming) techniques to associate to the skeletons different non-functional properties such as security or parallelism exploitation related properties.

**2. Template based vs. data flow based skeleton systems.** A skeleton based parallel programming environment provides programmers with a set of predefined and parametric parallelism exploitation patterns. The patterns are parametric in the kind of basic computation executed in parallel and, possibly, in the execution parallelism degree or in some other execution related parameters. For example, a pipeline skeleton takes as parameters the computations to be computed at the pipeline stages. In some skeleton systems these computations can be either sequential computations or parallel ones (i. e. other skeletons) while in other systems (mainly the ones developed at the very beginning of the skeleton related research activity) these computations may only be sequential ones.

The first attempts to implement skeleton programming environments all relied on the implementation template technology. Original Cole skeletons [12], Darlington's group skeleton systems [18, 20, 19], Kuchen's Muesli [23, 26] and our group's P3L [7] and ASSIST [36] all use this implementation schema. As discussed in [27], in an implementation template based skeleton system each skeleton is implemented using a parametric process network chosen from those available in a template library for that particular skeleton and for the kind of target architecture at hand (see [28], which discusses several implementation templates, all suitable for implementing task farms, that is embarrassingly parallel computations implemented according to a master-worker paradigm). The template library is designed once and for all by the skeleton system designer and captures the state of the art knowledge relating to implementation of the parallelism exploitation patterns modeled by the skeletons. Therefore the compilation process of a skeleton program, according to the implementation template model, can be summarized as follows:

1. the skeleton program is parsed and a skeleton tree representing the precise skeleton structure of the user application is derived. The skeleton tree has nodes marked with one of the available skeletons, and leaves marked with sequential code (sequential skeletons).
2. The skeleton tree is traversed, in some order, and templates from the library are assigned to each of the skeleton nodes, apart from the sequential ones, which always correspond to the execution of a sequential process on the target machine. During this phase, parameters of the templates (e.g. the parallelism degree or the kind of communication mechanisms used) are fixed, possibly using heuristics associated with the library entries.
3. The annotated skeleton tree is used to generate the actual parallel code. Depending on the system this may involve a traditional compilation step (e.g. in P3L when using the Anacleto compiler [11] or in ASSIST when using the `astcc` compiler tools [2, 1]) or use of a skeleton library hosting templates (e.g. Muesli [26] and eSkel [14] exploiting MPI).
4. The parallel code is eventually run on the target architecture, possibly exploiting some kind of loader/ deploy tool.

Figure 2.1 summarizes the process of deriving running code from skeleton source code using template technology.

More recently, an implementation methodology based on data flow has been proposed [15]. In this case the skeleton source code is used to compile a data flow graph and the data flow graph is then executed on the target architecture using a suitable distributed data flow interpreter engine. The approach has been used both in our group, in the implementation of Lithium [35, 6], and in Serot's SKIPPER skeleton environment [30]. In both cases the data flow approach was used to support fixed skeleton set programming environments. We adopted the very same implementation approach in the `muskel` full Java skeleton library, but in `muskel`

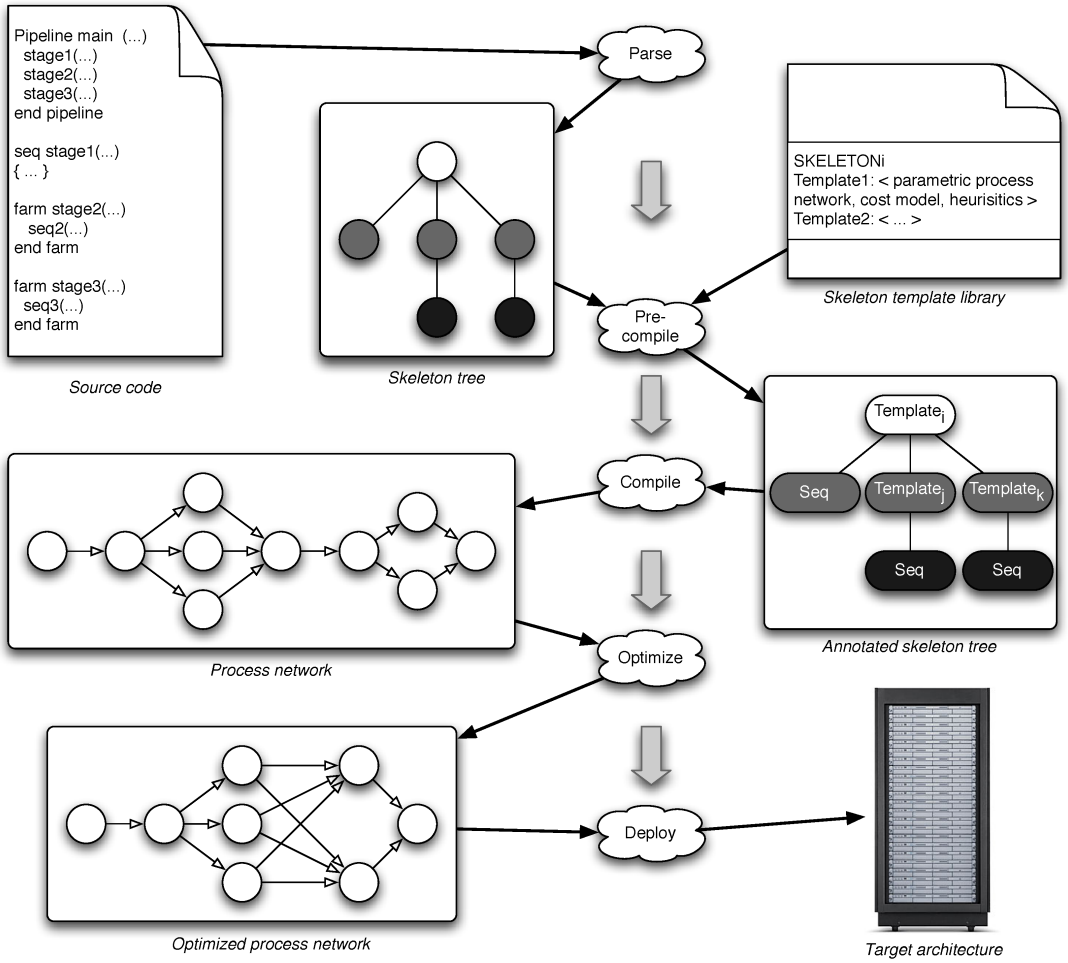



FIG. 2.1. *Skeleton program execution according to the implementation template approach.*

(as shown in the rest of this paper) the data flow implementation is also used to support extensible skeleton sets.

When data flow technology is exploited to implement skeletons, the compilation process of a skeleton program can be summarized as follows:

1. the skeleton program is parsed and a data flow graph is derived. The data flow graph represents the pure data flow behaviour of the skeleton tree in the program.
2. For each of the input tasks, a copy of the data flow graph is instantiated, with the task appearing as an input token to the graph. The new graph is delivered to the distributed data flow interpreter "instruction pool".
3. The distributed data flow interpreter fetches fireable instructions from the instruction pool and the instructions are executed on the nodes in the target architecture. Possibly, optimizations are taken into account (based on heuristics) that try to avoid unnecessary communications (e.g. caching tokens that will eventually be reused) or to adapt the computation grain of the program to the target architecture features (e.g. delivering more than a single fireable instruction to remote nodes to decrease the impact of communication set up latency, or multiprocessing the remote nodes to achieve communication and computation overlap).

Figure 2.2 summarizes the steps leading from skeleton source code to the running code using this data flow approach. It is worth pointing out that macro data flow implementation of skeletons is "pure data flow" compliant: no side effects, such as those deriving from the usage of global variables, are supported, nor can data flow graphs compiled from one skeleton in the program affect/modify the graphs compiled from the other skeletons

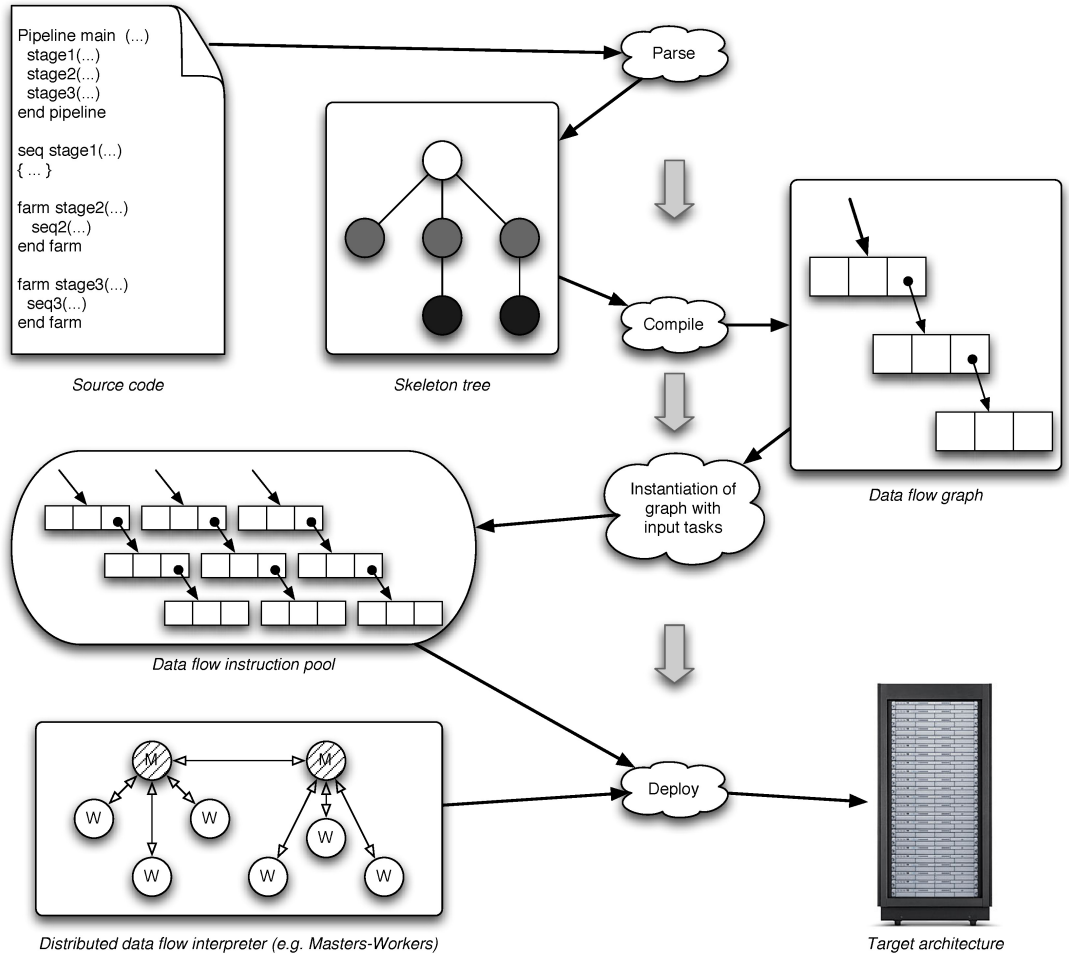



FIG. 2.2. *Skeleton program execution according to the data flow approach.*

in the program. This can be perceived as a limitation if we assume a non-structured parallel programming perspective. However, this represents a strong point in the structured parallel programming perspective as it guarantees that macro data flow graphs separately generated from skeletons appearing in the source code can be composed/unfolded safely in the global macro data flow graph eventually run on the distributed macro data flow interpreter.

The two approaches just outlined appear very different, but they have been successfully used to implement different skeleton systems. To support what will be presented in §4.2, we wish first to point out a quite subtle difference in the two approaches.

On the one hand, when using implementation templates, the process network eventually run on the target architecture is very similar to the one the user has in mind when instantiating skeletons in the source code. In some systems the "optimization" phase of Fig. 2.1 is actually empty and the program eventually run on the target architecture is built by simple juxtaposition of the process networks making up the templates of the skeletons used in the program. Even when the optimization phase does actually modify the process network structure (in Fig. 2.1 the master/slave service process of the two consecutive farms are optimized/collapsed, for instance), the overall structure of the process network does not change very much.

On the other hand, when a data flow approach is used the process network run on the target architecture has almost nothing to do with the skeleton tree described by the programmer in the source code. Rather, the skeleton tree is used to implement the parallel computation in a correct and efficient way, exploiting a set of techniques and mechanisms that are much closer to the techniques and mechanisms used in operating systems rather than to those used in the execution of parallel programs, both structured and unstructured. From a slightly different perspective, this can be interpreted as follows:

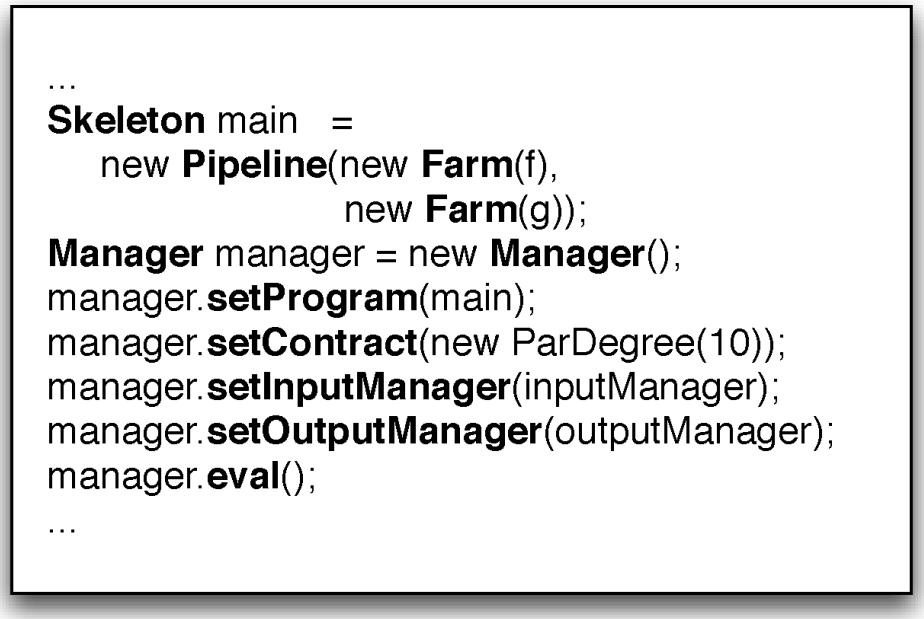



FIG. 3.1. *Sample* `muskel` *code: sketch of all (but the sequential portions of code) the code needed to set up and execute a two-stage pipeline with parallel stages (farms).*

- skeletons in the program "annotate" sequential code by providing the meta information required to efficiently implement the program in parallel;
- the support tools of the skeleton programming environment (the data flow graph compiler and the distributed data flow interpreter, in this case) "interprets" the meta information to accurately and efficiently implement the skeleton program, exploiting (possibly at run time, when the target architecture features are known) the whole set of known mechanisms supporting implementation optimization (e.g. caches, prefetching, node multiprocessing, etc.).

Viewed in this way, the data flow implementation for parallel skeleton programs presents a new perspective in the design of parallel programming systems where parallelism is dealt with as a "non-functional" feature, introduced by programmers via annotations or exploiting Aspect-Oriented Programming (AOP) techniques, and handled by the compiling/runtime support tools in the most convenient and efficient way with respect to the target architecture at hand (see §4.2).

**3.** `muskel`**.** `muskel` is a full Java skeleton programming environment derived from Lithium [6]. Currently, it provides only the stream parallel skeletons of Lithium, namely stateless task farm and pipeline. These skeletons can be arbitrarily nested, to program pipelines with farm stages, for example, and they process a single stream of input tasks to produce a single stream of output tasks. `muskel` implements skeletons using data flow technology and Java RMI facilities. The programmer using `muskel` can express parallel computations by simply using the provided `Pipeline` and `Farm` classes. For example, to express a parallel computation structured as a two-stage pipeline with a farm in each of the stages, the user should write code such as that shown in Fig. 3.1. `f` and `g` are two classes implementing the `Skeleton` interface, i. e. supplying a `compute` method with the signature `Object compute(Object t)` computing `f` and `g`, respectively. The `Skeleton` interface represents the "sequential" skeleton, that is the skeleton always executed sequentially and only used to wrap sequential code in such a way that it can be used in other, non-sequential skeletons.

In order to execute the program, the programmer first sets up a `Manager` object. Then, using appropriate methods, he indicates to the manager the program to execute, the performance contract required (in this case, the parallelism degree required for the execution), what is in charge of providing the input data (the input stream manager, which is basically an iterator providing the classical `boolean hasNext()` and `Object next()`

methods) and what is in charge of processing the output data (the output stream manager, providing only a `void deliver(Object)` method processing a single result of the program). Finally, he can request parallel program execution simply by issuing an `eval` call to the manager. When the call terminates, the output file has been produced.

Actually, the `eval` method execution happens in steps. First, the application manager looks for available processing elements using a simplified, multicast based peer-to-peer discovery protocol, and recruits the required remote processing elements. Each remote processing element runs a data flow interpreter. Then the skeleton program (the `main` of the example) is compiled into a macro data flow graph (capitalizing on normal form results shown in [3, 6]) and a thread is forked for each of the remote processing elements recruited. Then the input stream is read. For each task item, an instance of the macro data flow graph is created and the task item token is stored in the proper place (initial data flow instruction(s)). The graph is placed in the task pool, the repository for data flow instructions to be executed. Each thread looks for a fireable instruction in the task pool and delivers it for execution to the associated remote data flow interpreter. The remote interpreter instance associated to the thread is initialized by being sent the serialized code of the data flow instructions, once and for all, before the computation actually starts. Once the remote interpreter terminates the execution of the data flow instruction, the thread either stores the result token in the appropriate "next" data flow instruction(s) in the task pool, or it directly writes the result to the output stream, invoking the `deliver` method of the output stream manager. Currently, the task pool is a centralized one, associated with the centralized manager. We are currently investigating the possibility to distribute both task pool and manager so as to remove this bottleneck. The `manager` takes care of ensuring that the performance contract is satisfied. If a remote node "disappears" (e.g. due to a network failure, or to the node failure/shutdown), the manager looks for another node and starts dispatching data flow instructions to the new node instead [16]. As the manager is a centralized entity, if it fails, the whole computation fails. However, the manager is usually run on the user machine, which is assumed to be safer than the remote nodes recruited as remote interpreter instances.

The policies implemented by the `muskel` managers are *best effort*. The `muskel` library tries to do its best to accomplish user requests. If it is not possible to completely satisfy the user requests, the library establishes the closest configuration to the one implicitly specified by the user with the performance contract. In the example above, the library tries to recruit 10 remote interpreters. If only $n < 10$ remote interpreters are found, the parallelism degree is set exactly to $n$. In the worst case, that is if no remote interpreter is found, the computation is performed sequentially, on the local processing element.

In the current version of the `muskel` prototype, the only performance contract actually implemented is the `ParDegree` one, asking for the use of a constant number of remote interpreters in the execution of the program. The prototype has been designed to support at least another kind of contract: the `ServiceTime` one. This contract can be used to specify the maximum amount of time expected between the delivery of two program result tokens. Thus, with a call such as `manager.setContract(new ServiceTime(500))`, the user may request delivery of one result every half a second (time is in ms, as usual in Java). We do not discuss in more detail the implementation of the distributed data flow interpreter here. The interested reader can refer to [15, 16]. Instead, we will present more detail of the compilation of skeleton code into data flow graphs.

A `muskel` parallel skeleton code is described by the grammar:

$$\mathsf{P} ::= \mathsf{seq}(\mathit{className}) \mid \mathsf{pipe}(\mathsf{P}, \mathsf{P}) \mid \mathsf{farm}(\mathsf{P})$$

where the `className`s refer to classes implementing the `Skeleton` interface, and a macro data flow instruction (MDFi) is a tuple:

$$\mathsf{MDFi} \equiv Id \times Id \times Id \times \mathcal{I}^n \times \mathcal{O}^k$$

where the first $Id$ : $paper.tex, v1.352007/03/2316 : 45 : 59 marcodExp$ represents the MDFi identifier distinguishing that MDFi from other MDFi in the graph, the second represents the graph id (both are either integers or the special `NoId` identifier), the third the identifier of the Skeleton code computed by the MDFi; and, finally, $\mathcal{I}$ and $\mathcal{O}$ are the input tokens and the output token destinations, respectively. An input token is a pair $\langle \mathit{value}, \mathit{presenceBit} \rangle$ and an output token destination is a pair $\langle \mathit{destInstructionId}, \mathit{destTokenNumber} \rangle$. With these assumptions, a data flow instruction such as $\langle a, b, \mathtt{f}, \langle \langle 123, \mathtt{true} \rangle, \langle \mathtt{null}, \mathtt{false} \rangle \rangle, \langle \langle i, j \rangle \rangle \rangle$ is the instruction with identifier $a$ belonging to the graph with identifier $b$. It has two input tokens, one present (the integer 123) and one not present yet. It is not fireable, as one token is missing. When the missing token is delivered to this

instruction, either from the input stream or from another instruction, the instruction becomes fireable. To be computed, the two tokens must be given to the `compute` method of the `f` class. The method computes a single result that will be delivered to the instruction with identifier $i$ in the same graph, in the position corresponding to input token number $j$. The process compiling the skeleton program into the data flow graph can therefore be more formally described as follows. We define a pre-compile function $PC : \mathsf{P} \times Id \rightarrow (Id \rightarrow \mathsf{MDFi}^\star)$ as follows:

$$PC[\mathsf{P}]_g = \begin{cases} \lambda i.\ \{\langle newId(), g, \mathtt{f}, \langle\langle \mathtt{null}, \mathtt{false}\rangle\rangle, \langle\langle i, 1\rangle\rangle\rangle\} & \textit{if } \mathsf{P} = \mathsf{seq}\ (\mathtt{f}) \\ PC[\mathsf{P}_1]_g & \textit{if } \mathsf{P} = \mathsf{farm}(\mathsf{P}_1\ ) \\ \lambda i.((PC[\mathsf{P}_1]_{gid}\ (getId(T))) \cup (T(i))) & \\ \quad \text{where } T = PC[\mathsf{P}_2]_{gid} & \textit{if } \mathsf{P} = \mathsf{pipe}\ (\mathsf{P}_1, \mathsf{P}_2) \end{cases}$$

where $\lambda x.T$ is the usual lambda notation for functions and $getID()$ returns the $id$ of the first instruction in its argument graph, that is, the one assuming to receive the input token from outside the graph.
Then, we define the compile function $C : \mathsf{P} \rightarrow \mathsf{MDFi}^\star$ as follows:

$$C[\mathsf{P}\ ] = PC[\mathsf{P}\ ]_{newGid()}\ (\mathtt{NoId})$$

where $newId()$ and $newGid()$ are stateful functions returning a fresh (i. e. unused) instruction and graph identifier, respectively. The compile function therefore returns a graph, with a fresh graph identifier, containing all the data flow instructions defining the skeleton program. The result tokens are identified as those whose destination is `NoId`. For example, the compilation of the `main` program $\mathsf{pipe}(\mathsf{farm}(\mathsf{seq}(\mathtt{f})), \mathsf{farm}(\mathsf{seq}(\mathtt{g})))$ produces the data flow graph:

$$\{\langle 2, 1, \mathtt{f}, \langle\langle \mathtt{null}, \mathtt{false}\rangle\rangle, \langle\langle 1, 1\rangle\rangle\ , \langle 1, 1, \mathtt{g}, \langle\langle \mathtt{null}, \mathtt{false}\rangle\rangle, \langle\langle \mathtt{NoId}, 1\rangle\rangle\rangle\}$$

(assuming that identifiers and token positions start from 1).

When the application manager is told to compute the program, via an `eval()` method call, the input file stream is read looking for tasks to be computed. Each task found is used to replace the data field of the initial data flow instruction in a new $C[\mathsf{P}\ ]$ graph. In the example above, this results in the generation of a set of independent graphs such as:

$$\{\langle 2, i, \mathtt{f}, \langle\langle \mathtt{null}, \mathtt{false}\rangle\rangle, \langle\langle 1, 1\rangle\rangle\ , \langle 1, i, \mathtt{g}, \langle\langle \mathtt{null}, \mathtt{false}\rangle\rangle, \langle\langle \mathtt{NoId}, 1\rangle\rangle\rangle\}$$

for all the tasks ranging from $task_1$ to $task_n$.

All the resulting instructions are put in the task pool of the distributed interpreter in such a way that the control threads taking care of "feeding" the remote data flow interpreter instances can start fetching the fireable instructions. The output tokens generated by instructions with destination tag equal to `NoId` are delivered directly to the output file stream by the threads receiving them from the remote interpreter instances. Those with a non-`NoId` flag are delivered to the appropriate instructions in the task pool, which will eventually become fireable.

**4. Expanding `muskel` skeleton facilities.** In this section, we will discuss how the skeleton facilities provided by `muskel` can be extended to accomplish particular user requirements. Two issues are considered. First, the mechanisms used to allow programmers to define their own skeletons are discussed, along with their `muskel` implementation. Using these mechanisms, the programmers may declare and use arbitrary, possibly "unstructured"[1] new skeletons. Then, we discuss how alternative mechanisms based on Java annotations and/or AOP techniques are currently being used to provide further expandability of the `muskel` skeleton set, in particular characterizing existing skeletons with new, non-functional features.

**4.1. User-defined skeletons.** In order to introduce completely new parallelism exploitation patterns, `muskel` provides programmers with mechanisms that can be used to design arbitrary macro data flow graphs. A macro data flow graph can be defined creating some `Mdfi` (macro data flow instruction) objects and connecting them in a `MdfGraph` object.

For example, the code in Fig. 4.1 is that needed to program a data flow graph with two instructions. The first computes the `inc1 compute` method on its input token and delivers the result to the second instruction. The second computes the `sq1 compute` method on its input token and delivers the result to a generic "next" instruction (this is modelled by giving the destination token tag a `Mdfi.NoInstrId` tag). The `Dest` type in the code represents the destination of output tokens as triples containing the graph identifier, the instruction

[1] With respect to classical skeleton frameworks.

```
Skeleton inc1 = new Inc();
Dest d = new Dest(0, 2 ,Mdfi.NoGraphId);
Dest[] dests = new Dest[1];
dests[0] = d;
Mdfi i1 = new Mdfi(manager,1,inc1,1,1,dests);
Skeleton sq1 = new Square();
Dest d1 = new Dest(0,Mdfi.NoInstrId, Mdfi.NoGraphId);
Dest[] dests1 = new Dest[1];
dests1[0] = d1;
Mdfi i2 = new Mdfi(manager,2,sq1,1,1,dests1);
MdfGraph graph = new MdfGraph();
graph.addInstruction(i1);
graph.addInstruction(i2);
ParCompute userDefMDFg = new ParCompute(graph);
```

FIG. 4.1. *Custom/user-defined skeleton declaration.*

identifier and the destination input token targeted in this instruction. Macro data flow instructions are built by specifying the manager they refer to, their identifier, the code executed (must be a `Skeleton` object) the number of input and output tokens and a vector with a destination for each of the output tokens.

We do not present all the details of arbitrary macro data flow graph construction here (a complete description is provided with the `muskel` documentation). The example is just to give the flavor of the tools provided in the `muskel` environment. Bear in mind that the simple macro data flow graph of Fig. 4.1 is actually the same macro data flow graph obtained by compiling a primitive `muskel` skeleton call such as: `Skeleton main = new Pipeline(new Inc(), new Sq())` More complex user-defined macro data flow graphs may include instructions delivering tokens to an arbitrary number of other instructions, as well as instructions gathering input tokens from several distinct other instructions. In general, the mechanisms of `muskel` permit the definition of any kind of graph with macro data flow instructions computing sequential (side effect free) code wrapped in a `Skeleton` class. Any parallel algorithm that can be modeled with a data flow graph can therefore be expressed in `muskel`[2]. Non deterministic MDFi are not yet supported (e.g. one that merges input tokens from two distinct sources) although the firing mechanism in the interpreter can be easily adapted to support this kind of macro data flow instructions. Therefore, new skeletons added through the macro data flow mechanism always model pure functions.

`MdfGraph` objects are used to create new `ParCompute` objects. `ParCompute` objects can be used in any place where a `Skeleton` object is used. Therefore, user-defined parallelism exploitation patterns can be used as pipeline stages or as farm workers, for instance. The only limitation on the graphs that can be used in a `ParCompute` object consists in requiring that the graph has a unique input token and a unique output token.

When executing programs with user-defined parallelism exploitation patterns the process of compiling skeleton code to macro data flow graphs is slightly modified. When an original `muskel` skeleton is compiled, the process described in §3 is applied. When a user-defined skeleton is compiled, the associated macro data flow graph is directly taken from the `ParCompute` instance variables where the graph supplied by the user is maintained. Such a graph is linked to the rest of the graph according to the rules appropriate to the skeleton where the user-defined skeleton appears.

To show how the whole process works, let us suppose we want to pre-process each input task in such a way that for each task $t_i$ a new task

$$t'_i = h_1(f_1(t_i), g_2(g_1(f_1(t_i))))$$

[2]Note, however, that common, well know parallel application skeletons are already modelled by pre-defined `muskel Skeletons`.

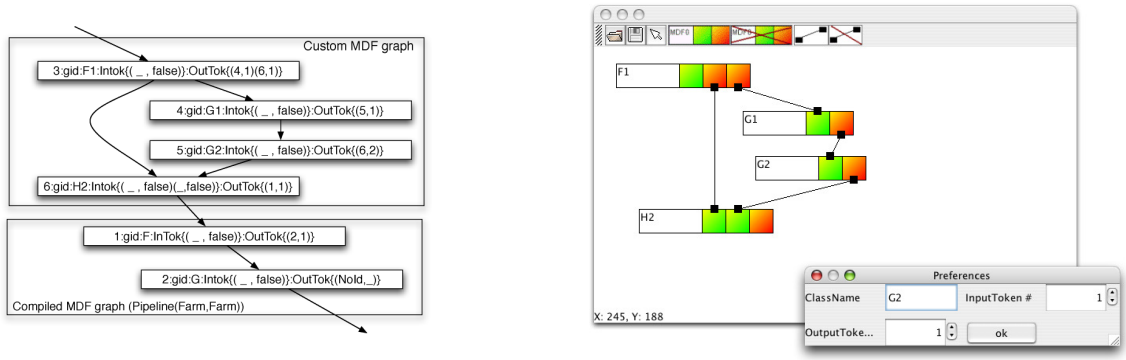



FIG. 4.2. *Mixed sample macro data flow graph (left): the upper part comes from a user-defined macro data flow graph (it cannot be derived using primitive* `muskel` *skeletons) and the lower part is actually coming from a three stage pipeline with two sequential stages (the second and the third one) and a parallel first stage (the user-defined one). GUI tool designing the upper graph (right).*

is produced. This computation cannot be programmed using the stream parallel skeletons currently provided by `muskel`. In particular, current pre-defined skeletons in `muskel` allow only processing of one input to produce one output, and therefore there is no way to implement the graph described here. In this case we wish to process the intermediate results through a two-stage pipeline to produce the final result. To do this the programmer can set up a new graph using code similar to the one shown in Fig. 3.1 and then use that new `ParCompute` object as the first stage of a two-stage pipeline whose second stage happens to be the postprocessing two-stage pipeline. When compiling the whole program, the outer pipeline is compiled first. As the first stage is a user-defined skeleton, its macro data flow graph is directly taken from the user-supplied one. The second stage is compiled according to the (recursive) procedure described in §3 and eventually the (unique) last instruction of the first graph is modified in such a way that it sends its only output token to the first instruction in the second stage graph. The resulting graph is outlined in Fig. 4.2 (left).

Making good use of the mechanisms allowing definition of new data flow graphs, the programmer can arrange to express computations with arbitrary mixes of user-defined data flow graphs and graphs coming from the compilation of structured, stream parallel skeleton computations. The execution of the resulting data flow graph is supported by the `muskel` distributed data flow interpreter in the same way as the execution of any other data flow graph derived from the compilation of a skeleton program. At the moment the `muskel` prototype allows user-defined skeletons to be used as parameters of primitive `muskel` skeletons, but not vice versa. We are currently working to extend `muskel` to alow the latter.

While the facility to include user-defined skeletons provides substantial flexibility, we recognize that the current way of expressing new macro data flow graphs is error prone and not very practical. Therefore we have designed a graphic tool that allows users to design their macro data flow graphs and then compile them to actual Java code as required by `muskel` and shown above. Fig. 4.2 (right) shows the interface presented to the user. In this case, the user is defining the upper part of the graph in the left part of the same Figure. It is worth pointing out that all that is needed in this case is to connect output and input token boxes appropriately, and to configure each MDFi with the name of the sequential `Skeleton` used. The smaller window on the right lower corner is the one used to configure each node in the graph (that is, each MDFi). This GUI tool produces an XML representation of the graph. Then, another Java tool produces the correct `muskel` code implementing the macro data flow graph as a `muskel ParCompute` skeleton. As a result, users are allowed to extend, if required, the skeleton set by just interacting with the GUI tool and "compiling" the graphic MDF graph to `muskel` code by clicking on one of the buttons in the top toolbar.

As a final example, consider the code of Fig. 4.3. This code outlines how a new `Map2` skeleton, performing in parallel the same computation on all the portions of an input vector, can be defined and used. It is worth pointing out how user-defined skeletons, once properly debugged and fine-tuned, can simply be incorporated in

```
public class Map2 extends ParCompute {

    public Map2(Skeleton f, Manager manager) {
        super(null);
        // first build the empty graph
        program = new MdfGraph();
        // build the emitter instruction
        Dest [] dds1 = new Dest[2];
        dds1[0]=new Dest(0,2);
        dds1[1]=new Dest(0,3);
        Mdfi emitter =
                new Mdfi(manager, 1,
                new MapEmitter(2), 1, 2, dds1);
        // add it to the graph
        program.addInstruction(emitter);
        // build first half map Skeleton node
        Dest [] dds2 = new Dest[1];
        dds2[0] = new Dest(0,4);
        Mdfi if1 = new Mdfi(manager,2, f, 1, 1, dds2);
        program.addInstruction(if1);
        // build second half map Skeleton node
        Dest []dds3 = new Dest[1];
        dds3[0] = new Dest(1,4);
        Mdfi if2 = new Mdfi(manager,3, f, 1, 1, dds3);
        program.addInstruction(if2);
        Dest[] ddslast = new Dest[1];
        ddslast[0] = new Dest(0,Mdfi.NoInstrId);
        Mdfi collector = new Mdfi(manager,4,new
                        MapCollector(), 2, 1, ddslast);
        program.addInstruction(collector);
        return;
    }
    ...
}
```

```
public class SampleMap {
    public static void main(String[] args) {
        Manager manager =
            new Manager();
        Skeleton worker = new Fdouble();
        Skeleton main =
            new Map2(worker,manager);

        InputManager inManager =
            new DoubleVectIM(10,4);
        OutputManager outManager =
            new DoubleVectOM();

        ParDegree contract =
            new ParDegree(10);
        manager.setInputManager(inManager);
        manager.setOutputManager(outManager);
        manager.setContract(contract);
        manager.setProgram(main);

        manager.compute();
    }
}
```

FIG. 4.3. *Introducing a new, user-defined skeleton: a map working on vectors and with a fixed, user-defined parallelism degree.*

the `muskel` skeleton library and used seamlessly, as the primitive `muskel` ones, but for the fact that (as shown in the code) the constructor needs the manager as a parameter. This is needed so as to be able to link together the macro data flow graphs generated by the compiler and those supplied by the user. It is worth noting that skeletons such as a general form of `Map` are usually provided in the fixed skeleton set of any skeleton system and users usually do not need to implement them. However, as `muskel` is an experimental skeleton system, we concentrate the implementation efforts on features such as the autonomic managers, portability, security and expandability rather than providing a complete skeleton set. As a consequence, `muskel` has no predefined map skeleton and the example of user defined skeleton just presented suitably illustrates the methodology used to expand the "temporary" restricted skeleton set of the current version of `muskel` depending on the user needs. The `Map2` code shown here implements a "fixed parallelism degree" *map*, that is the number of "workers" used to compute in parallel the skeleton does not depend on the size of the input data. It is representative of a more general `Map`skeleton taking a parameter specifying the number of workers to be used. However, in order to support the implementation of a map skeleton with the number of workers defined as a function of the input data, some kind of support for the dynamic generation of macro data flow graphs is needed, which is not present in the current `muskel` prototype.

**4.2. Non-functional features.** We briefly discuss here how annotations and aspect-oriented programming techniques and mechanisms can be used to introduce convenient ways of expressing non-functional features of parallel skeleton programs in `muskel`. Unlike the work discussed in the previous section, which has already been implemented in the current `muskel` prototype, this is more on-going work. We have preliminary results demonstrating the approach is feasible and we are currently working to transfer the experimental techniques to the "production" `muskel` prototype.

```
public aspect Normalize {

      pointcut callSetProgram(Skeleton c):
            call(public void Manager.setProgram(Skeleton)) && args(c);


         void around(Skeleton c)
      : callSetProgram(c) {
            proceed(new NormalForm(c));
         }

}
```

FIG. 4.4. *AspectJ code modeling normal form in* `muskel`.

In this context, let us consider as non-functional features all that is not related to the control flow that the programmer needs to set up to compute the final program result. For instance, we consider as non-functional features the necessity to secure code and data management in a program execution, the application of optimization rules transforming the user-supplied program into an equivalent, possibly more efficient one, or the hints given by programmers as to the features to exploit during the execution of the parallel skeleton code.

We wish to outline how these features can be implemented in the `muskel` framework using some innovative programming techniques.

First consider security issues. When executing a `muskel` program on a network of workstations, it may be the case that the workstations used happen to be in different local networks, possibly interconnected by public, untrusted network segments. Also, it may be the case that the user running the program does not have complete control of the machines used to run the remote data flow interpreter instances, and therefore cannot exclude malicious user activity on the remote machines aimed at reading or modifying the program or the data involved in the parallel program run. Therefore, it is appropriate to provide mechanisms that can be used in the `muskel` support to authenticate and encrypt all the communications happening during a `muskel` program run, both those relating to the transmission of the (serialized) program code and those relating to input and output token communications. As an example, an `ssl` transport layer can be used instead of plain TCP/IP to implement the `muskel` communications. However, the use of the `ssl` transport layer involves a communication cost which is definitely higher than the cost involved in plain TCP/IP configurations (see results shown in §5). Therefore, the user may wish to denote in the program which are the sensitive data or code segments that must not be transmitted in clear on untrusted networks. Java annotations can be used to the purpose, as follows:

- the programmer annotates (using some `@SensitiveCode` and `@SensitiveData` annotations) those `Skeleton`s whose code must be properly secured and those data that must be kept secret;
- then the `Manager`, in the `eval` implementation may use reflection to access these annotations and to process them properly. That is, in the case of `Skeleton` objects annotated as `@SensitiveCode` it provides for distribution of the code using `ssl` tunnelled RMI, in the case of tasks/tokens annotated as `@SensitiveData` it provides for invocation of remote `compute` execution again using `ssl` tunnelled RMI, while in all other cases it uses plain RMI over unencrypted, more efficient TCP/IP connections.

Now consider a different kind of non-functional feature: source-to-source program optimization rules. For example, let us consider our previous result on skeleton program *normal form*. Such result [3] can be informally stated as follows: an arbitrary `muskel` program whose structure is a generic skeleton tree made out of pipelines, farms and sequential skeletons may be transformed into a new, equivalent one, whose parallel structure is a farm with each worker made up of the sequential composition of the sequential skeletons appearing in the original skeleton tree taken left to right. This second program is the skeleton program normal form and happens to

```
public aspect Normalize {

	public boolean ContractSpecified = false;
	public boolean normalized = false;
	// contract is an integer, to simplify ...
	public int contract = 0;

	pointcut calledContract(int i): call(public void Manager.setContract(int)) && args(i);

	void around(int i): calledContract(i){
		ContractSpecified = true;
		contract = i;
		proceed(i);
	}

	pointcut callSetProgram(Skeleton c): call(public void Manager.setProgram(Skeleton)) && args(c);

	void around(Skeleton c):
	callSetProgram(c) {
		normalized = true;
		proceed(new NormalForm(c));
	}

	pointcut callEval(Manager m) : call(public void Manager.eval()) && target(m);

	before(Manager m):callEval(m){
		if(ContractSpecified)
			if(normalized)
				m.setContract(Manager.NormalizeContract(contract));
		else
			m.setContract(Manager.DefaultNormalizedContract);
		}
	}
}
```

FIG. 4.5. *AspectJ code handling performance contracts in* `muskel`.

perform better than the original one in the general case and in the same way in the worst case (this with respect to the service time). As an example, the code of Fig. 3.1 can be transformed into the equivalent normal form code: `Skeleton main = new Farm(new Seq(f,g));` where `Seq` is basically a pipeline whose stages are executed sequentially on a single processor.

In Lithium, normal form can be used by explicitly inserting statements in the source code. This means that the user must change the source code to use the normal form or the non-normal form version of the same program. Using AOP (and AspectJ, in particular) we can define an aspect dealing with normal form transformation by defining a pointcut on the execution of the `setProgram Manager` method and associating to the pointcut the action performing normal form transformation on the source code in the aspect, such as the one of Fig. 4.4. As a consequence, the user can decide whether to use the original or the normal form version of the program just by choosing the standard Java compiler or the AspectJ one. The fact that the program is left unchanged means the programmer may debug the original program and have the normal form one debugged too as a consequence, provided the AOP code in the normal form aspect is correct, of course. Moreover, if normal form is handled by aspects as discussed above, it is better to handle also related features by means of suitable aspects. For example, if the user provided a performance contract (a parallelism degree, in the simpler case) and then used the AspectJ compiler to request normal form execution of the program, it turns out to be quite natural to imagine a further aspect handling the performance contract consequently. Fig. 4.5 shows the AspectJ code handling this feature. In this case, contracts are stored as soon as they have been issued by the programmer, with the first pointcut, then, when normalization has been required (second pointcut) and program parallel evaluation is required, the contract is handled consequently (third pointcut); in this case it is either left unchanged or a new contract is derived from the original one according to some normal form related procedure.

At the moment we are experimenting with both annotations and AOP techniques to provide the `muskel` programmer with better tools supporting more and more possibilities to customize parallelism exploitation in `muskel` programs.

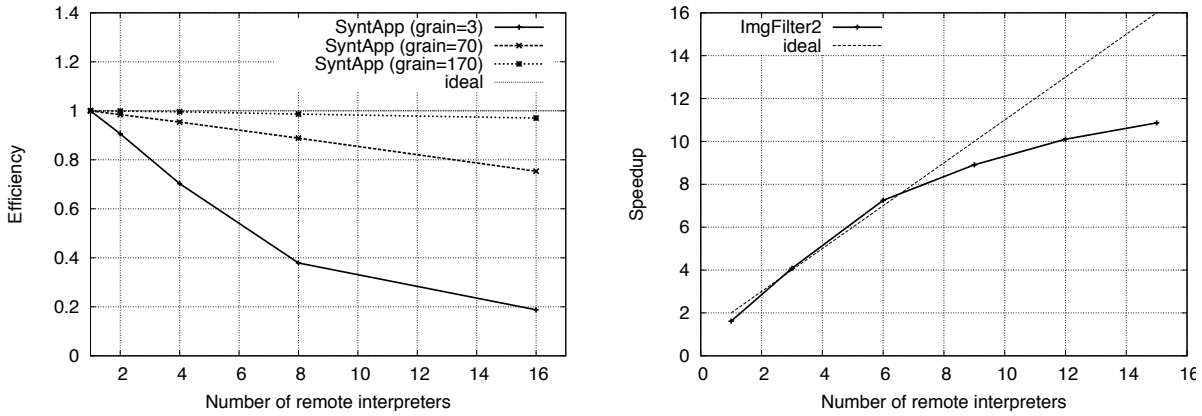



FIG. 4.6. *muskel performance versus number of remote interpreters on a homogeneous cluster. Left) Efficiency of* SyntApp *for several computation grains. Right) Speedup of* ImgFilter2 *compared with ideal speedup.*

In particular, we have investigated the possibility of relieving the programmer of the need to specify farm skeletons at all. Instead of declaring farm skeletons, programmers may simply annotate as `@Parallel` the `Skeleton` objects and the run time support directly manages to transform calls to the `compute` methods of such objects into farms [17]. This is not a completely new technique, but it can be used to evaluate the effectiveness of the approach, compared both to the original `muskel` farm handling and to a similar approach defining `Skeleton` objects to be computed in parallel in a farm by properly setting up a farm aspect with actions establishing task farm like computation patterns upon the invocation of the `Skeleton compute` method.

**5. Experimental results.** We ran some experiments aimed at validating the `muskel` prototype supporting user defined skeletons. The results shown refer to two applications. SyntApp is a synthetic application processing 1K distinct input tasks and designed in such a way that the macro data flow instructions appearing in the graph had a precise "average grain" (i. e. average ratio among the time spent computing the instruction at the remote interpreter and the time spent communicating data to and from the remote interpreter, $G = T_w/T_c$). ImgFilter2 is an image processing application based on the pipeline skeleton, which applies two filters in sequence to 30 input images. All input images are true-color (24 bit color depth) of 640x480 pixels size. ImgFilter2 basically applies "blur" and "oil" filters (available at `http://jiu.sourceforge.net`) from the Java Imaging Utilities in sequence as two stages of a pipeline. Note that these are *area filter operations*, i. e. the computation of each pixel's color does not only impact its direct neighbours, but also an adjustable area of neighboring pixels. By choosing five neighboring pixels in each direction as filter workspaces, we made the application more complex and enforced several iterations over the input data within each pipeline stage, which makes our filtering example a good representative of a compute intensive application [5].

Two types of parallel platforms are used for experimentation. The first is a dedicated Linux cluster at the University of Pisa. The cluster hosts 24 nodes: one node devoted to cluster administration and 18 nodes (P3@800MHz) exclusively devoted to parallel program execution. The second is a grid-like environment, including two organizations: the University of Pisa *(di.unipi.it)* and an institute of the Italian National Research Council in Pisa *(isti.cnr.it)*. The server set is composed of several different Intel Pentium and Apple PowerPC computers, running Linux and Mac OS X respectively (the detailed configuration is shown in Figure 5.1 left). In this case traditional measures like efficiency and speedup versus number of machines cannot be used due to the machines' power heterogeneity. To take the varying computing power of different machines into account, the performance increase is documented by means of the *BogoPower* measure, defined as the sum of individual BogoPower contributions of machines participating in the application run. The BogoPower of each machine is measured in terms of tasks/s the sequential version of the application can compute on the machine. BogoPower enables the comparison between an application's actual parallel performance and the application's ideal performance for each run [5].

Figure 4.6 summarizes the typical performance results of the enhanced interpreter. The left plot is relative to runs of SyntApp on the homogeneous cluster. The experiment shows that, in case of low grain, `muskel` rapidly loses efficiency with the number of machines involved in the computation. When the grain is high

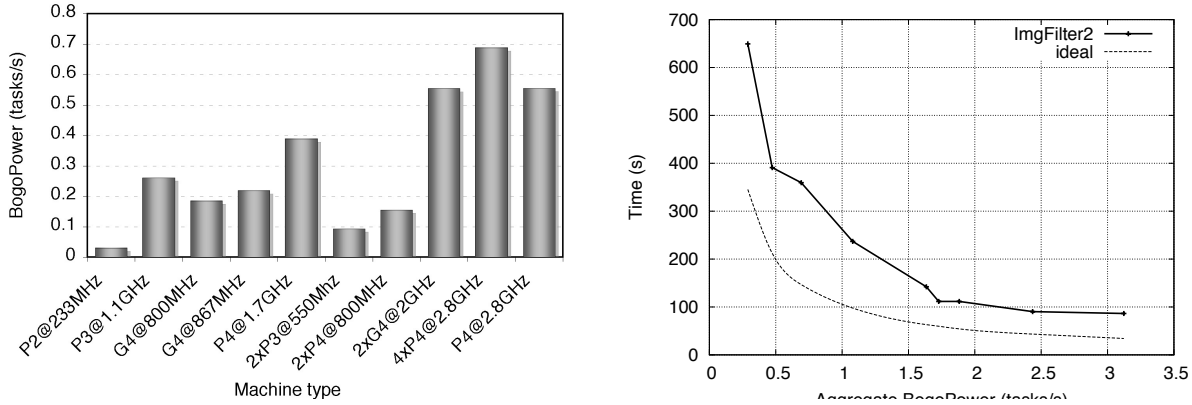



FIG. 5.1. *muskel performance on a grid-like testing environment. Left) Description of platforms in the testing environment (machine-type, Bogopower). Right) Completion time of ImgFilter2*

enough ($G = 200$ or more) the efficiency is definitely close to the ideal one. The right plot shows the speedup of ImgFilter2 on the same homogeneous cluster, instead.

Figure 5.1 right plots the completion time of ImgFilter2 executed on an heterogeneous network of Linux/Pentium and MacOsX PowerPC machines, whose relative performance is shown in the left part of the same Figure. The measured completion times show the same shape as the theoretical ones, confirming that the `muskel` run time efficiently and automatically balances the load when different (with respect to computing power) resources are used to allocate the `muskel` distributed macro data flow interpreter.

In addition to the evaluation of the scalability of the `muskel` prototype, we also have taken into account the possibility of using different mechanisms to support distributed data flow interpreter execution. We implemented several versions of `muskel` on top of ProActive [29], each exploiting different mechanisms, primitive to the ProActive library, to deploy and run remote macro data flow interpreter instances. In particular, we used ProActive XML deployment descriptors as well as RMI `ssh` tunnelling. When possible, we exploited the option to pre-allocate JVMs running the remote interpreter instances on the remote processing elements, to speed up program startup. The results showed that in the case where the remote JVMs are preallocated, the performance is definitely comparable to the performance of plain `muskel`. In the case of use of RMI tunelling through `ssh`, however, larger grain macro data flow instructions (close to 10 times larger grain) are needed to achieve almost perfect speedup.

As discussed in §3, appropriate security mechanisms, defined using Java 1.5 annotations, should be used to guarantee that data and code moved to and from the remote data flow interpreter instances are kept confidential and that intruders cannot use remote data flow interpreter instances to execute non-authorized macro data flow code. We conducted some experiments to evaluate the effectiveness of introducing selective security annotations in the code. We prepared a stripped `muskel` prototype version, using `ssl` to secure interaction between the main code running on the user machine and the remote data flow interpreter instances. With the `muskel` prototype exploiting `ssl` [34], we managed to measure the scalability penalty paid to introduce security. We verified that "secure" `muskel` scales close to ideal values when using up to 32 nodes for the remote macro data flow interpreter instances, similarly to plain `muskel`. However, due to the encrypting/decrypting activity taking place at the sending/receiving nodes, larger (i. e.more compute intensive) macro data flow instructions are required to achieve ideal scalability (see Figure 5.2 left). Also, we measured the load distribution in runs involving half "secure" and half "non-secure" remote interpreter instances. Communications with the secure interpreter instances are performed using plain TCP/IP, while communications with the non-secure ones are performed using SSL. With higher and higher amounts of data transferred to and from the remote interpreters more and more computation is performed on the secure nodes. This is due to the auto scheduling strategy of `muskel` that always dispatches computations to the free remote interpreter instances. As more data is transmitted, more time is spent securing communications through SSL and more time is spent computing a single MDFi. Therefore less MDFi are actually executed at the non-secure nodes (see Figure 5.2 right). The results shown are perfectly in line with what is stated in §4: securing `muskel` communications is quite costly and therefore it

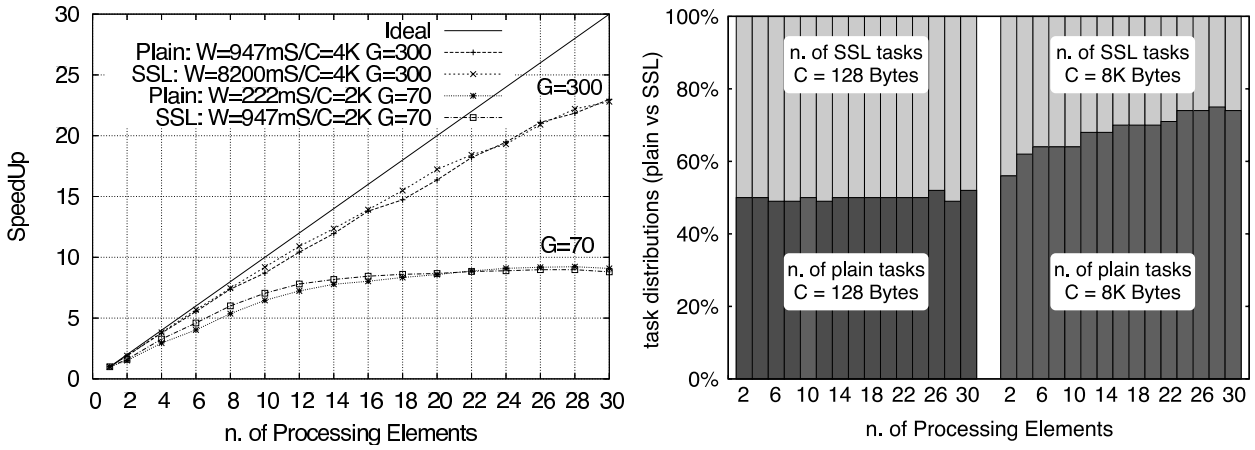



FIG. 5.2. *Effect of providing security in the distributed data flow interpreter: scalability of the* `muskel` *prototype using plain TCP/IP sockets vs. the one using SSL for different computational grains. W represent the average time spent computing a single MDFi, C the average amount of data sent/received to/from remote processing elements to compute the single MDFi*

is better to avoid securing communications not involving sensitive data and/or code. And this can be done by exploiting the annotation mechanisms just outlined in §4. Further details concerning security issues in `muskel` are discussed in [4].

**6. Related work.** Macro data flow implementation for the algorithmical skeleton programming environments was introduced by the authors in the late 90's [15] and subsequently has been used in other contexts related to skeleton programming environments [31]. Cole suggested in [13] that "we must construct our systems to allow the integration of skeletal and ad-hoc parallelism in a well defined way", and that structured parallel programming environments should "accommodate diversity", that is "we must be careful to draw a balance between our desire for abstract simplicity and the pragmatic need for flexibility". Actually, his eSkel [9, 14] MPI skeleton library addresses these problems by allowing programmers to program their own peculiar MPI code within each process in the skeleton tree. Programmers can ask to have a stage of a pipeline or a worker in a farm running on $k$ processors. Then, the programmer may use the $k$ process communicators returned by the library for the stage/worker to implement its own parallel pipeline stage/worker process. As far as we know, this is the only other attempt to integrate ad hoc, unstructured parallelism exploitation in a structured parallel programming environment. The implementation of eSkel, however, is based on process templates, rather than on data flow. Other skeleton libraries, such as Muesli [23, 24, 26], provide programmers with quite extensive flexibility in skeleton programming following a different approach. They provide a number of data parallel data structures along with elementary, collective data parallel operations that can be arbitrarily nested to get more and more complex data parallel skeletons. However, this flexibility is restricted to the data parallel part, and it is, in any case, limited by the available collective operations.

CO2P3S [25] is a design pattern based parallel programming environment written in Java and targeting symmetric multiprocessors. In CO2P3S, programmers are allowed to program their own parallel design patterns (skeletons) by interacting with the intermediate implementation level [10]. Again, this environment does not use data flow technology but implements design patterns using proper process network templates.

JaSkel [21] provides a skeleton library implementing the same skeleton set as `muskel`. In JaSkel, however, skeletons look much more like implementation templates, according to the terminology used in §2. However, it appears that the user can exploit the full OO programming methodology to specialize the skeletons to his own needs. As the user is involved in the management of support code too (e.g. he has to specify the master process/thread of a task farm skeletons) JaSkel can be classified as a kind of "low level, extensible" skeleton system, although it is not clear from the paper whether entirely new skeletons can be easily added to the system (actually, it looks like it is not possible at all).

There are several works proposing aspect-oriented techniques for parallel programming. [22] discusses an approach using AOP to separate concerns in scientific code. In [33, 32] a use of AOP is proposed aimed at separating the concerns of partitioning and distributing data and performing concurrent computations. This is

far from the usage we think to make of AOP techniques in this work, however, in that it requires a much more "template oriented" approach with respect to the one followed in `muskel`.

**7. Conclusions.** We discussed `muskel`, a full Java, parallel programming library providing users with the possibility to use skeletons to structure their parallel applications and exploiting macro data flow implementation technology. We discussed how `muskel` supports expandability of the skeleton set, as advocated by Cole in his "manifesto" paper [13]. In particular, we discussed how `muskel` supports both the introduction of new skeletons, modeling parallelism exploitation patterns not originally covered by the primitive `muskel` skeletons, and the introduction of non-functional features, i. e. features related to parallel program execution but not directly related to the functional computation of the application results. The former possibility is supported by allowing users to define new skeletons providing the arbitrary data flow graph executed in the skeleton and by allowing `muskel` to seamlessly integrate such new skeletons with the primitive ones. The latter possibility is supported by exploiting more innovative programming techniques such as annotations and aspect-oriented programming. This second part is under development, while the first is already available in the `muskel` prototype.

We also presented experimental results validating the whole `muskel` approach to expandability and customizability of its skeleton set. As far as we know, this is the most significant effort in the skeleton community to tackle problems deriving from a fixed skeleton set. Only Schaeffer and his group at the University of Alberta implemented a system were users can, in controlled ways, insert new parallelism exploitation patterns in the system [10], although the approach followed there is a bit different, in that users are encouraged to intervene directly in the run time support implementation, to introduce new skeletons, while in `muskel` new skeletons may be introduced using the intermediate macro data flow language as the skeleton "assembly" language.

Finally, we discussed how relatively new programming techniques, including annotations and AOP, can be usefully exploited in `muskel` to support details and features related to parallel program execution.

Preliminary versions of `muskel` have been released under GPL and are currently available on the muskel web site at `htpp://www.di.unipi.it/~marcod/muskel`. The new version, supporting the features discussed in this paper, is currently being developed. The support for new skeletons is already completed (and it is available, as a beta release, on the web site) and the other features will be released soon.

**Acknowledgements.** We wish to thank Daniele Licari, who implemented the graphic interface supporting user friendly design of new `muskel` skeletons. We wish also to thank Peter Kilpatrick for all the very useful and inspirational discussions we had on `muskel`, and for helping us proofread this paper.

REFERENCES

[1] M. Aldinucci, S. Campa, P. Ciullo, M. Coppola, M. Danelutto, P. Pesciullesi, R. Ravazzolo, M. Torquati, M. Vanneschi, and C. Zoccolo, *A framework for experimenting with structured parallel programming environment design*, in Parallel Computing: Software Technology, Algorithms, Architectures and Applications, PARCO 2003, G. R. Joubert, W. E. Nagel, F. J. Peters, and W. V. Walter, eds., vol. 13 of Advances in Parallel Computing, Dresden, Germany, 2004, Elsevier, pp. 617–624.

[2] M. Aldinucci, S. Campa, P. Ciullo, M. Coppola, S. Magini, P. Pesciullesi, L. Potiti, R. Ravazzolo, M. Torquati, M. Vanneschi, and C. Zoccolo, *The implementation of ASSIST, an environment for parallel and distributed programming*, in Proc. of 9th Intl Euro-Par 2003 Parallel Processing, H. Kosch, L. Böszörményi, and H. Hellwagner, eds., vol. 2790 of LNCS, Klagenfurt, Austria, Aug. 2003, Springer, pp. 712–721.

[3] M. Aldinucci and M. Danelutto, *Stream parallel skeleton optimization*, in Proc. of PDCS: Intl. Conference on Parallel and Distributed Computing and Systems, Cambridge, Massachusetts, USA, Nov. 1999, IASTED, ACTA press, pp. 955–962.

[4] ———, *The cost of security in skeletal systems*, in Euromicro PDP 2007: Parallel Distributed and network-based Processing, IEEE, February 2007. Naples, Italy.

[5] M. Aldinucci, M. Danelutto, J. Dünnweber, and S. Gorlatch, *Optimization techniques for skeletons on grid*, in Grid Computing and New Frontiers of High Performance Processing, L. Grandinetti, ed., vol. 14 of Advances in Parallel Computing, Elsevier, Oct. 2005, ch. 2, pp. 255–273.

[6] M. Aldinucci, M. Danelutto, and P. Teti, *An advanced environment supporting structured parallel programming in Java*, Future Generation Computer Systems, 19 (2003), pp. 611–626.

[7] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi, *$P^3L$: a structured high level programming language and its structured support*, Concurrency Practice and Experience, 7 (1995), pp. 225–255.

[8] B. Bacci, M. Danelutto, S. Pelagatti, and M. Vanneschi, *SkIE: A heterogeneous environment for HPC applications*, Parallel Computing, 25 (1999), pp. 1827–1852.

[9] A. Benoit, M. Cole, S. Gilmore, and J. Hillston, *Flexible skeletal programming with eSkel*, in Proc. of 11th Intl. Euro-Par 2005 Parallel Processing, J. C. Cunha and P. D. Medeiros, eds., vol. 3648 of LNCS, Lisboa, Portugal, Aug. 2005, Springer, pp. 761–770.

[10] S. Bromling, *Generalising pattern-based parallel programming systems*, in Parallel Computing: Advances and Current Issues. Proc. of the Intl. Conference ParCo2001, G. R. Joubert, A. Murli, F. J. Peters, and M. Vanneschi, eds., Imperial College Press, 2002.
[11] S. Ciarpaglini, M. Danelutto, L. Folchi, C. Manconi, and S. Pelagatti, *ANACLETO: a template-based P3L compiler*, in Proc. of the Parallel Computing Workshop (PCW'97), 1997. Camberra, Australia.
[12] M. Cole, *Algorithmic Skeletons: Structured Management of Parallel Computations*, Research Monographs in Parallel and Distributed Computing, Pitman, 1989.
[13] ———, *Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming*, Parallel Computing, 30 (2004), pp. 389–406.
[14] M. Cole and A. Benoit, *The eSkel home page*. `http://homepages.inf.ed.ac.uk/abenoit1/eSkel/`.
[15] M. Danelutto, *Dynamic run time support for skeletons*, in Proc. of Intl. PARCO 99: Parallel Computing, E. H. D'Hollander, G. R. Joubert, F. J. Peters, and H. J. Sips, eds., Parallel Computing Fundamentals & Applications, Imperial College Press, 1999, pp. 460–467.
[16] ———, *QoS in parallel programming through application managers*, in Proc. of Intl. Euromicro PDP: Parallel Distributed and network-based Processing, Lugano, Switzerland, Feb. 2005, IEEE, pp. 282–289.
[17] M. Danelutto, P. Dazzi, D. Laforenza, M. Pasin, L. Presti, and M. Vanneschi, *PAL: High level parallel programming with Java annotations*, in Proceedings of CoreGRID Integration Workshop (CIW 2006) Krakow, Poland, Academic Computer Centre CYFRONET AGH, Oct. 2006, pp. 189–200. ISBN 83-915141-6-1.
[18] J. Darlington, A. J. Field, P. Harrison, P. H. J. Kelly, D. W. N. Sharp, R. L. While, and Q. Wu, *Parallel programming using skeleton functions*, in Proc. of Parallel Architectures and Languages Europe (PARLE'93), vol. 694 of LNCS, Munich, Germany, June 1993, Springer, pp. 146–160.
[19] J. Darlington, M. Ghanem, and H. W. To, *Structured Parallel Programming*, in Programming Models for Massively Parallel Computers, Berlin, Germany, September 1993, IEEE Computer Society Press.
[20] J. Darlington, Y. Guo, H. W. To, Q. Wu, J. Yang, and M. Kohler, *Fortran-S: a uniform functional interface to parallel imperative languages*, in Third Parallel Computing Workshop (PCW'94), Fujitsu Laboratories Ltd., November 1994.
[21] J. F. Ferreira, J. L. Sobral, and A. J. Proença, *JaSkel: A Java skeleton-based framework for structured cluster and grid computing.*, in CCGRID, IEEE Computer Society, 2006, pp. 301–304.
[22] B. Harbulot and J. R. Gurd, *Using aspectj to separate concerns in parallel scientific java code.*, in AOSD, G. C. Murphy and K. J. Lieberherr, eds., ACM, 2004, pp. 122–131.
[23] H. Kuchen, *A skeleton library*, in Proc. of 8th Intl. Euro-Par 2002 Parallel Processing, B. Monien and R. Feldman, eds., vol. 2400 of LNCS, Paderborn, Germany, Aug. 2002, Springer, pp. 620–629.
[24] ———, *Optimizing sequences of skeleton calls*, in Domain-Specific Program Generation, D. Batory, C. Consel, C. Lengauer, and M. Odersky, eds., vol. 3016 of LNCS, Springer, 2004, pp. 254–273.
[25] S. McDonald, D. Szafron, J. Schaeffer, and S. Bromling, *Generating parallel program frameworks from parallel design patterns*, in Proc of. 6th Intl. Euro-Par 2000 Parallel Processing, A. Bode, T. Ludwing, W. Karl, and R. Wismüller, eds., vol. 1900 of LNCS, Springer, Aug. 2000, pp. 95–105.
[26] *The muesli home page*, 2006. `http://www-wi.uni-muenster.de/pi/personal/kuchen.php`.
[27] S. Pelagatti, *Structured Development of Parallel Programs*, Taylor & Francis, 1998.
[28] M. Poldner and H. Kuchen, *Scalable farms*, in Parallel Computing: Current & Future Issues of High-End Computing, Proc. of PARCO 2005, G. R. Joubert, W. E. Nagel, F. J. Peters, O. Plata, P. Tirado, and E. Zapata, eds., vol. 33 of NIC, Germany, Dec. 2005, Research Centre Jülich, pp. 795–802.
[29] *ProActive home page*, 2006. `http://www-sop.inria.fr/oasis/proactive/`.
[30] J. Serot, *Tagged-token data-flow for skeletons*, Parallel Processing Letters, 11 (2001), pp. 377–392.
[31] J. Serot and D. Ginhac, *Skeletons for parallel image processing: an overview of the SKIPPER project*, Parallel Computing, 28 (2002), pp. 1685–1708.
[32] J. L. Sobral, *Incrementally Developing Parallel Applications with AspectJ*, in Proc. of 20th Intl. Parallel & Distributed Processing Symposium (IPDPS), IEEE, April 2006.
[33] J. L. Sobral, M. P. Monteiro, and C. A. Cunha, *Aspect-oriented support for modular parallel computing*, in Proc. of the 5th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software, Y. Coady, D. H. Lorenz, O. Spinczyk, and E. Wohlstadter, eds., Bonn, Germany, March 2006, Published as University of Virginia Computer Science Technical Report CS–2006–01, pp. 37–41.
[34] *JSSE for the Java 2 SDK*, 2006. `http://java.sun.com/products/jsse/index-14.html`.
[35] P. Teti, *Lithium: a Java skeleton environment*, Master's thesis, Dept. Computer Science, University of Pisa, October 2001. In Italian.
[36] M. Vanneschi, *The programming model of ASSIST, an environment for parallel and distributed portable applications*, Parallel Computing, 28 (2002), pp. 1709–1732.

*Edited by:* Anne Benoît and Frédéric Loulergue
*Received:* September, 2006
*Accepted:* March, 2007

Scalable Computing: Practice and Experience
Volume 8, Number 4, pp. 343–358. http://www.scpe.org
ISSN 1895-1767


# A BUFFERING LAYER TO SUPPORT DERIVED TYPES AND PROPRIETARY NETWORKS FOR JAVA HPC*

MARK BAKER†, BRYAN CARPENTER‡, AND AAMIR SHAFI§

**Abstract.** MPJ Express is our implementation of MPI-like bindings for Java. In this paper we discuss our intermediate buffering layer that makes use of the so-called direct byte buffers introduced in the Java New I/O package. The purpose of this layer is to support the implementation of derived datatypes. MPJ Express is the first Java messaging library that implements this feature using pure Java. In addition, this buffering layer allows efficient implementation of communication devices based on proprietary networks such as Myrinet. In this paper we evaluate the performance of our buffering layer and demonstrate the usefulness of direct byte buffers. Also, we evaluate the performance of MPJ Express against other messaging systems using Myrinet and show that our buffering layer has made it possible to avoid the overheads suffered by other Java systems such as mpiJava that relies on the Java Native Interface.

**Key words.** Java, MPI, MPJ express, MPJ, mpiJava

**1. Introduction.** The challenges of making parallel hardware usable have, over the years, stimulated the introduction of many novel languages, language extensions, and programming tools. Lately though, practical parallel computing has mainly adopted conventional (sequential) languages, with programs developed in relatively conventional programming environments usually supplemented by libraries such as MPI that support a parallel programming paradigm. This is largely a matter of economics: creating entirely novel development environments matching the standards programmers expect today is expensive, and contemporary parallel architectures predominately use commodity microprocessors that can best be exploited by off-the-shelf compilers.

This argues that if we want to "raise the level" of parallel programming, one practical approach is to move towards advanced commodity languages. Compared with C or Fortran, the advantages of the Java programming language include higher-level programming concepts, improved compile-time and run-time checking, and as a result, faster problem detection and debugging. Its "write once, run anywhere" philosophy allows Java applications to be executed on almost all popular platforms. It also supports multi-threading and provides simple primitives like `wait()` and `notify()` that can be used to synchronize access to shared resources. Recent Java Development Kits (JDKs) provide greater functionality in this area, including semaphores and atomic variables. In addition, Java's automatic garbage collection, when exploited carefully, relieves the programmer of many of the pitfalls of lower-level languages. During the early days of Java, it was criticized for its poor performance [4]. The main reason was that Java executed as an interpreted language. The situation has improved with the introduction of Just-In-Time (JIT) compilers, which translate bytecode into the native machine code that then gets executed.

MPJ Express [14] is a thread safe Java HPC communication library and runtime system that provides a high quality implementation of the mpiJava 1.2 [6] bindings—an MPI-like API for Java. An important goal of our messaging system is to implement higher MPI [15] abstractions including derived datatypes in pure Java. In addition, we note the emergence of low-latency and high-bandwidth proprietary networks that have had a big impact on modern messaging libraries. In the presence of such networks, it is not *practical* to only use pure Java for communication. To tackle these issues of supporting derived datatypes and proprietary networks, we provide an intermediate buffering layer in MPJ Express. Providing an efficient implementation layer is a challenging aspect of a Java HPC messaging software. The low-level communication devices and higher levels of the messaging software use this buffering layer to write and read messages. The heterogeneity of these low-level communication devices poses additional design challenges. To appreciate this fully, assume that the user of a messaging library sends ten elements of an integer array. The C programming language can retrieve the memory address of this array and pass it to the underlying communication device. If the communication device is based on TCP, it can then pass this address to the socket's `write()` method. For proprietary networks like Myrinet [16], this memory region can be registered for Direct Memory Access (DMA) transfers, or copied to a DMA

*The authors would like to thank University of Portsmouth for supporting this research. The research work presented in this paper was conducted by the authors, as part of the Distributed Systems Group, at the University of Portsmouth.
†School of Systems Engineering, University of Reading, Reading, RG6 6AY, UK (mark.baker@computer.org)
‡Open Middleware Infrastructure Institute, University of Southampton, Southampton, SO17 1BJ, UK (dbc@ecs.soton.ac.uk)
§Center for High Performance Scientific Computing, NUST Institute of Information Technology, Rawalpindi, Pakistan (aamir.shafi@niit.edu.pk)

capable part of memory and sent using low level Myrinet communication methods. Until quite recently doing this kind of thing in Java was difficult.

The JDK 1.4 introduced the Java New I/O (NIO) [11] package. In NIO, read and write methods on files and sockets (for example) are mediated through a family of buffer classes handled by the Java Virtual Machine (JVM). The underlying `ByteBuffer` class essentially implements an array of bytes, but in such a way that the storage can be outside the JVM heap (so called *direct* byte buffers).

So now if a user of a Java messaging system sends an array of ten integers, they can be copied to a `ByteBuffer`, which is used as an argument to the `SocketChannel`'s `write()` method. Similary, if the user intends to communicate derived datatypes, the individual basic datatype elements of this derived type can be packed onto a contiguous `ByteBuffer`. The higher and lower levels of the software can use generic functionality provided by a buffering layer to communicate both basic and advanced datatypes, including Java objects and derived types. For proprietary networks like Myrinet, NIO provides a viable option because it is now possible to get memory addresses of direct byte buffers, which can be used to register memory regions for DMA transfers. Using direct buffers may eliminate the overhead [18] incurred by additional copying when using the Java Native Interface (JNI) [9]. On the other hand, it may be preferable to create a native buffer using JNI. These buffers can be useful for a native MPI or a proprietary network device.

For these reasons, we have designed an extensible buffering layer that allows various implementations based on different storage mediums, such as direct or indirect `ByteBuffer`s, byte arrays, or memory allocated in the native C code. The higher levels of MPJ Express use the buffering layer through an interface. This implies that functionality is not tightly coupled to the storage medium. The motivation behind developing different implementations of buffers is to achieve optimal performance for lower level communication devices. The creation time of these buffers can affect the overall communication time, especially for large messages. Our buffering strategy uses a pooling mechanism to avoid creating a buffer instance for each communication method. Our current implementation is based on Knuth's buddy algorithm [12], but it is possible to use other pooling techniques.

A closely related buffering API with similar gather and scatter functionality was originally introduced for Java in the context of the Adlib communication library used by HPJava [13]. In our current work, we have extended this API to support the derived datatypes in a fully functional MPI interface.

The main contribution of this paper is the in-depth analysis of the design and implementation of our buffering layer that allows high performance communication and supports implementing derived datatypes at the higher level. MPJ Express is the first Java messaging library that supports derived datatypes using pure Java. In addition, we have evaluated the performance of MPJ Express on Myrinet—a popular high performance interconnect. Also, we demonstrate the usefulness of direct byte buffers in Java messaging systems.

The remainder of this paper is organized as follows. Section 2 discusses the details of the MPJ Express buffering layer. Section 3 describes the implementation of derived datatypes in MPJ Express. In Section 4, we evaluate the performance of our buffering strategies, this is followed by a comparison of MPJ Express against other messaging systems on Myrinet. We conclude the paper and discuss future research work in Section 5.

**1.1. Related Work.** Under the general umbrella of exploiting Java in "high level" parallel programming, there are environments for *skeleton-based* parallel programming that are implemented in Java, or support Java programming. These include *muskel* [8] and *Lithium* [1]. At the present time muskel appears to be focussed on a coarse grained data flow style of parallelism, rather than the sort of Single Program Multiple Data (SPMD) parallelism addressed by MPI-like systems such as MPJ Express. Lithium encompasses SPMD parallelism through its *map* skeleton. So far as we can tell, Lithium is agnostic about how the processes in the "map" communicate amongst themselves, and in principle we see no reason why they could not use MPJ Express for this purpose. To this extent Lithium and our approach could be seen as complementary.

In a similar vein, Alt and Gortlatch [2] developed a prototype system for Grid programming using Java and RMI. Again the issues addressed in their work are somewhat orthogonal from the concerns of the present paper. But it is possible that some of their ideas for discovery of parallel compute hosts could be exploited by a future version of MPJ Express. Such approaches might supercede what we call the *runtime system* of the present MPJ Express—responsible for initiating node tasks on remote hosts.

In another related strand of research, there are systems that provide Java implementations of Valiant's Bulk Synchronous Parallel computing model (BSP). These include JBSP [10] and PUBWCL [5]. In the sense that these are providing a messaging platform to essentially do data parallel programming in Java, they compete more

directly with MPJ Express. They are distinguished from our work in focussing on a more specific programming model. MPI-based approaches embrace a significantly different, and in some respects wider, class of parallel programming models (on some platforms one could, of course, sensibly implement BSP in terms of MPI).

mpiJava [3] is a Java messaging system that uses JNI to interact with the underlying native MPI library. Being a wrapper library, mpiJava does not use a clearly distinguished buffering layer. After packing a message onto a contiguous buffer, a reference to this buffer is passed to the native C library. But in achieving this, additional copying may be required between the JVM and the native C library. This overhead is especially noticeable for large messages, if the JVM does not support pinning of memory.

Javia [7] is a Java interface to the Virtual Interface Architecture (VIA). An implementation of Javia exposes communication buffers used by the VI architecture to Java applications. These communication buffers are created outside the Java heap and can be registered for DMA transfers. This buffering technique makes it possible to achieve performance within 99% of the raw hardware.

An effort similar to Javia is Jaguar [18]. This uses compiled-code transformations to map certain Java bytecodes to short, in-lined machine code segments. These two projects, Jaguar and Javia, were the motivating factors to introduce the concept of direct buffers in the NIO package. The design of our buffering layer is based on direct byte buffers. In essence, we are applying the experiences gained by Jaguar and Javia to design a general and efficient buffering layer that can be used for pure Java and proprietary devices in Java messaging systems alike.

**2. The Buffering Layer in MPJ Express.** In this section, we discuss our approach to designing and implementing our MPJ Express buffering layer that is supported by a pooling mechanism. The self-contained API developed as a result is called the MPJ Buffering (`mpjbuf`) API. The functionality provided includes packing and unpacking of user data. The primary difficulty in implementing this is that the sockets do not directly access the memory and thus are unable to write or read the basic datatypes. The absence of pointers and the type safety features of the Java language make the implementation even more complex. Most of the complex operations used at the higher levels of the library, such as communicating objects and gather or scatter operations, are also supported by this buffering layer.

Before we go into the details of the buffering layer implementation, it is important to consider how this API is used. The higher-level of MPJ Express, specifically the point-to-point send methods, *pack* the user message onto a `mpjbuf` buffer. Once the user data, which may be primitive datatypes or Java objects, has been packed onto a `mpjbuf` buffer, the reference of this buffer is passed to lower communication devices that communicate data from the static and dynamic storage structures. At the receiving side the communication devices receive data into a `mpjbuf` buffer storing them in static and dynamic storage structures. Once the message has been received, the point-to-point receive methods *unpack* the `mpjbuf` buffer data onto user specified arrays.

**2.1. The Layout of Buffers.** An `mpjbuf` buffer object contains two data storage structures. The first is a *static* storage structure, in which the underlying storage primitive is an implementation of the `RawBuffer` interface. An implementation of the static storage structure, called `NIOBuffer`, uses direct or indirect `ByteBuffer`s. The second is a *dynamic* storage structure where a byte array is the storage primitive. The static portion of the `mpjbuf` buffer has predefined size, and can contain only primitive datatypes. The dynamic portion of the `mpjbuf` buffer is used to store serialized Java objects, where it is not possible to determine the size of the serialized objects beforehand.

A message consists of zero or more sections stored physically on the static or dynamic storage structure. Each section can hold elements of the same type, basic datatypes or Java objects. A section consists of a header, followed by the actual data payload. The data stored in a static buffer can be represented as big-endian or little-endian. This is determined by the encoding property of the buffer, which takes the value `java.nio.ByteOrder.BIG_ENDIAN` or `java.nio.ByteOrder.LITTLE_ENDIAN`. The encoding property of a newly created buffer is determined by the return value of the method `java.nio.ByteOrder.nativeOrder()`. A developer may change the format to match the encoding property of the underlying hardware, which results in efficient numeric representation at the JVM layer.

As shown in Figure 2.1, a message consists of zero or more sections. The message consists of a message header followed by the data payload. Padding of up to 7 bytes may follow a section if the total length of the section (header + data) is not a multiple of ALIGNMENT_UNIT, which has a value of 8. The general layout of an individual section stored on the static buffer is shown in Figure 2.2.

Figure 2.2 shows that the length of a message header is 8 bytes. The value of the first byte defines the elements type contained in the section. The possible values for static and dynamic buffers are listed in Table 2.1

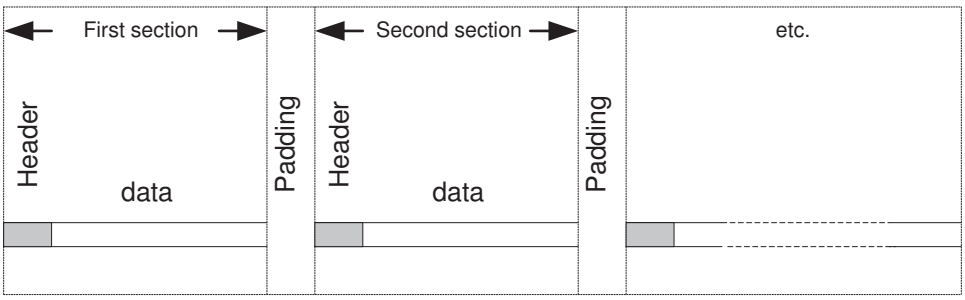



FIG. 2.1. *The Layout of a Static Storage Buffer*

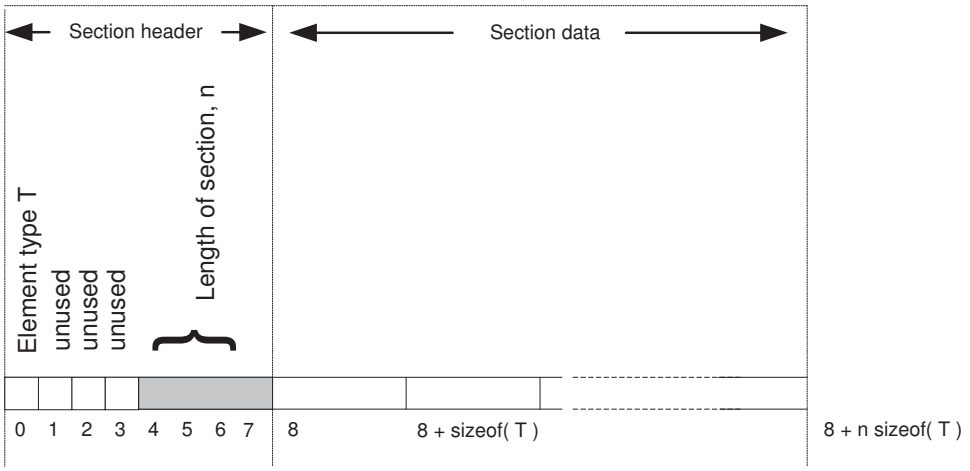



FIG. 2.2. *The Layout of a Single Section*

and Table 2.2, respectively. The next three bytes are not currently used, and reserved for possible future use. The following four bytes contain the number of elements contained in this section, i. e. the section length. This numerical value is represented according to the encoding property of the buffer. The size of the header in bytes is SECTION_OVERHEAD, which has a value of 8. If the section is static, the header is followed by the values of the elements, again represented according to the encoding property of the buffer. If the section is dynamic, the "Section data" is absent from Figure 2.2 because the data is in the dynamic buffer which is a byte array. The Java serialization classes (`java.io.ObjectOutputStream` and `java.io.ObjectInputStream`) dictate the format of the dynamic buffer.

A buffer object has two modes: write and read. The write mode allows the user to copy the data onto the buffer, and the read mode allows the user to read the data from the buffer. It is not permitted to read from the buffer when it is in write mode. Similarly, it is not permitted to write to a buffer when it is in read mode.

**2.2. The Buffering API.** The most important class of the package used for packing and unpacking data is `mpjbuf.Buffer`. This class provides two storage options: static and dynamic. Implementations of static storage use the interface `mpjbuf.RawBuffer`. It is possible to have alternative implementations of static section depending on the actual raw storage medium. In addition, it also contains an attribute of type `byte[]` that represents the dynamic section of the message. Figure 2.3 shows two implementations of the `mpjbuf.RawBuffer` interface. The first, `mpjbuf.NIOBuffer` is an implementation based on `ByteBuffer`s. The second, `mpjbuf.NativeBuffer` is an implementation for the native MPI device, which allocates memory in the native C code. Figure 2.3 shows the primary buffering classes in the mpjbuf API.

The higher and lower levels of MPJ Express use only a few of methods provided by the `mpjbuf.Buffer` class to pack and unpack message data. In addition, the class also provides some utility methods. Some of the main functions are shown in Figure 2.4. Note that identifier `type` used in the figure represents all Java basic datatypes and objects.

TABLE 2.1
*Datatypes Supported by a Static Buffer*

| Datatype | Corresponding Values |
|---|---|
| Integer | `mpjbuf.Type.INT` |
| Byte | `mpjbuf.Type.BYTE` |
| Short | `mpjbuf.Type.SHORT` |
| Boolean | `mpjbuf.Type.BOOLEAN` |
| Long | `mpjbuf.Type.LONG` |
| Float | `mpjbuf.Type.FLOAT` |
| Double | `mpjbuf.Type.DOUBLE` |

TABLE 2.2
*Datatypes Supported by a Dynamic Buffer*

| Datatype | Corresponding Values |
|---|---|
| Java objects | `mpjbuf.Type.OBJECT` |
| Integer | `mpjbuf.Type.INT_DYNAMIC` |
| Byte | `mpjbuf.Type.BYTE_DYNAMIC` |
| Short | `mpjbuf.Type.SHORT_DYNAMIC` |
| Boolean | `mpjbuf.Type.BOOLEAN_DYNAMIC` |
| Long | `mpjbuf.Type.LONG_DYNAMIC` |
| Float | `mpjbuf.Type.FLOAT_DYNAMIC` |
| Double | `mpjbuf.Type.DOUBLE_DYNAMIC` |

The `write()` and `read()` methods shown in Figure 2.4 are used to write and read contiguous Java arrays of all the primitive datatypes including object arrays. The `write()` method copies `numEls` values of the `src` array starting from `srcOff` onto the buffer. Conversely, the `read()` method copies `numEls` values from the buffer and writes them onto `dest` array starting from `srcOff`.

The `gather()` and `scatter()` methods are used to write and read non-contiguous Java arrays of all the primitive datatypes including object arrays. The `gather()` method copies `numEls` values of the `src` array starting from `indexes[idxOff]` to `indexes[idxOff+numEls]` onto the buffer. Conversely, the `scatter()` method copies `numEls` values from the buffer and writes them onto `dest` array starting from `indexes[idxOff]` to `indexes[idxOff+numEls]`.

The `strGather()` and `strScatter()` methods transfer data from or to a subset of elements of a Java array, but in these cases the selected subset is a *multi-strided region* of the array. These are useful operations for dealing with multi-dimensional data structures, which often occur in scientific programming.

To create sections, the `mpjbuf.Buffer` class provides utility methods like `putSectionHeader()`, which takes a datatype as an argument (possible datatypes are shown in Table 2.1 and Table 2.2). This method can only be invoked when the buffer is in a write mode. Once the section header has been created, the data can be copied onto the buffer using the `write()` method for contiguous user data or `gather()` and `strGather()` methods for non-contiguous user data. While the buffer is in read mode, the user can invoke `getSectionHeader()` and `getSectionSize()` methods to read the header information of a message. This is followed by invoking the `read()` method to read data, or `scatter()` and `strScatter()` methods to read non-contiguous data.

The newly created buffer is always in a write mode. In this mode, the user may copy the data to the buffer and then call `commit()`, which puts the buffer in a read mode. The user can now read the data from the buffer and put it back into write mode for any possible future use by calling the `clear()`.

**2.3. Memory Management.** We have implemented our own application level memory management mechanism based on the buddy allocation scheme [12]. The motivation was to avoid creating an instance of a buffer (`mpjbuf.Buffer`) for every communication operation like `Send()` or `Recv()`, which may dominate the total communication cost, especially for large messages. We can make efficient use of resources by pooling buffers for future reuse, instead of letting the garbage collector reclaim the buffers and create them all over again.

Currently the pooling mechanism is specific to direct and indirect `ByteBuffers` that are used for storing static data when `mpjbuf.NIOBuffer` (an implementation of `mpjbuf.RawBuffer`) is used for static sections.

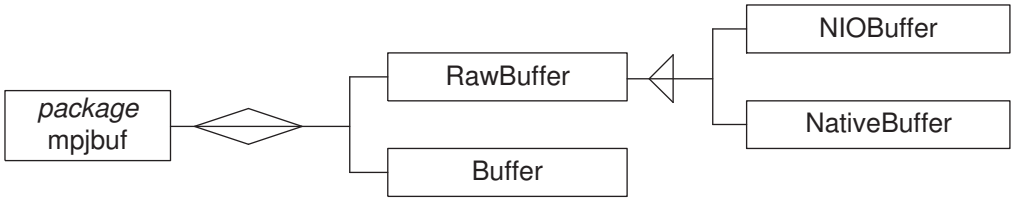



FIG. 2.3. *Primary Buffering Classes in* `mpjbuf`

```
package mpjbuf ;

public class Buffer {
  .. .. ..

  // Write and Read Methods
  public void write(type [] source, int srcOff, int numEls)
  public void read(type [] dest, int dstOff, int numEls)

  // Gather and Scatter Methods
  public void gather(type [] source, int numEls, int idxOff, int [] indexes)
  public void scatter(type [] dest, int numEls, int idxOff, int [] indexes)

  // Strided Gather and Scatter Methods
  public void strGather(type [] source, int srcOff, int rank, int exts,
                                                    int strs, int [] shape)
  public void strScatter(type [] dest, int dstOff, int rank, int exts,
                                                   int strs, int [] shape)

  public void putSectionHeader(Type type)

  public Type getSectionHeader()

  public int getSectionSize()

  public ByteOrder getEncoding()

  public void setEncoding(ByteOrder encoding)

  public void commit()

  public void clear()

  public void free()

  .. .. ..
}
```

FIG. 2.4. *The Functionality Provided by the* `mpjbuf.Buffer` *class*

**2.3.1. Review of The Buddy Algorithm.** In this section, we will briefly review Knuth's buddy algorithm in the context of MPJ Express. In our implementation, the available memory is divided into a series of buffers. Each buffer has a storage medium associated with it—direct or indirect `ByteBuffer`. Initially, there is no buffer associated with the `BufferFactory`. Whenever a user requests a buffer, the factory checks whether there is a buffer with size greater than the requested size. If a buffer does not exist or does not have free space, a new buffer is created. For managing the buffers, there is a doubly linked list called `FreeList`. This `FreeList` refers to buffers at all possible levels starting from 0 to $\lceil \log_2(REGION\_SIZE) \rceil$. The level of a buffer can be thought of an integer, which increases as $\lceil \log_2(s) \rceil$ increases if $s$ is the requested buffer size.

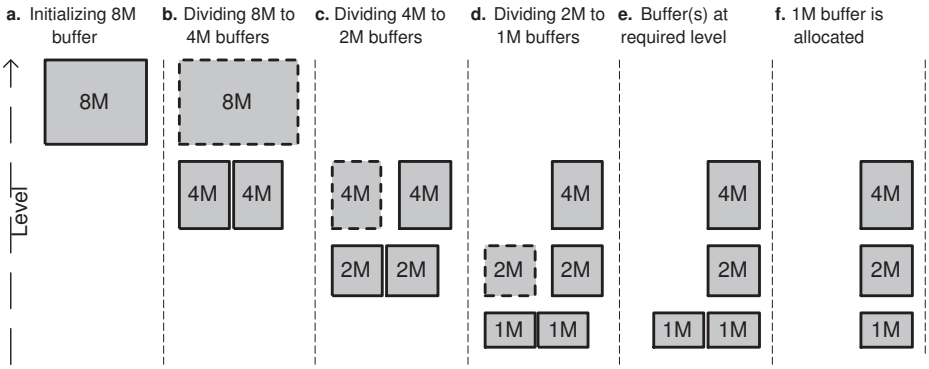



FIG. 2.5. *Allocating a Mbyte Buffer*

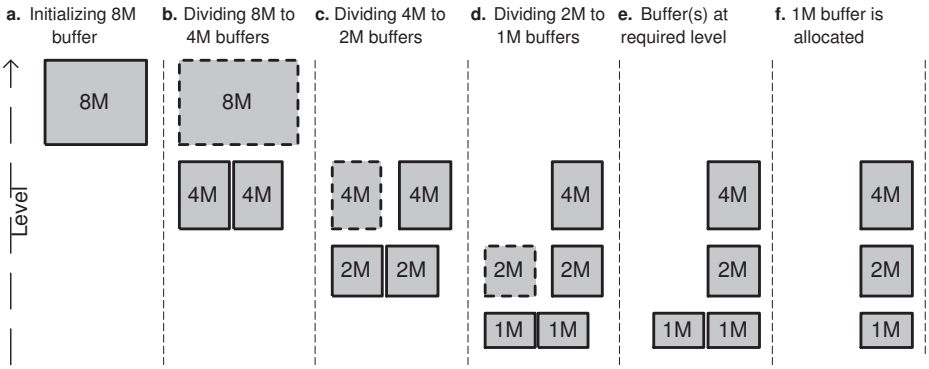



FIG. 2.6. *De-allocating a Mbyte Buffer*

After finding or creating an appropriate buffer that can serve this request, the algorithm attempts to find a free buffer at the requested or higher level. If the buffer found is at a higher level, it is divided into two *buddies* and this process is repeated until we reach the required level. The `BufferFactory` returns the first free buffer at this level. Every allocated buffer is aware of its offset and its size. Figure 2.5 shows the allocation events for a Mbyte block when the initial region size is 8 Mbytes.

When the buffer is de-allocated, an attempt is made to find the buddy of this buffer. If the buddy is free, the two buffers are merged together to form a buffer at the higher level. Once we have a buffer at the higher level, we execute the same process recursively until we do not find a buddy for the buffer at the higher level. Figure 2.6 shows the de-allocation events when a Mbyte block is returned to the buffer factory.

**2.3.2. Two implementations of the Buddy Allocation Scheme for `mpjbuf`.** In the MPJ Express buffering API, it is possible to plug in different implementations of buffer pooling. A particular strategy can be specified during the initialisation of `mpjbuf.BufferFactory`. Each implementation can use different data structures like trees or doubly linked lists. In the current implementation, the primary storage buffer for mpjbuf is an instance of `mpjbuf.NIOBuffer`. Each `mpjbuf.NIOBuffer` has an instance of `ByteBuffer` associated with it. The pooling strategy boils down to reusing `ByteBuffer`s encapsulated in `NIOBuffer`.

Our implementation strategies are able to create smaller thread-safe `ByteBuffer`s from the initial `ByteBuffer` associated with the region. We achieve this by using `ByteBuffer.slice()` for creating a new byte buffer.

In the buddy algorithm, the region of available storage is conceptually divided into blocks of different levels, hierarchically nested in a binary tree. A free block at level $n$ can be split into two blocks of level $n-1$, half the size. These sibling blocks are called *buddies*. To allocate a number of bytes $s$, a free block is found and recursively divided into buddies until a block at level $\lceil \log_2(s) \rceil$ is produced. When a block is freed, one checks to see if its buddy is free. If so, buddies are merged (recursively) to consolidate free memory.

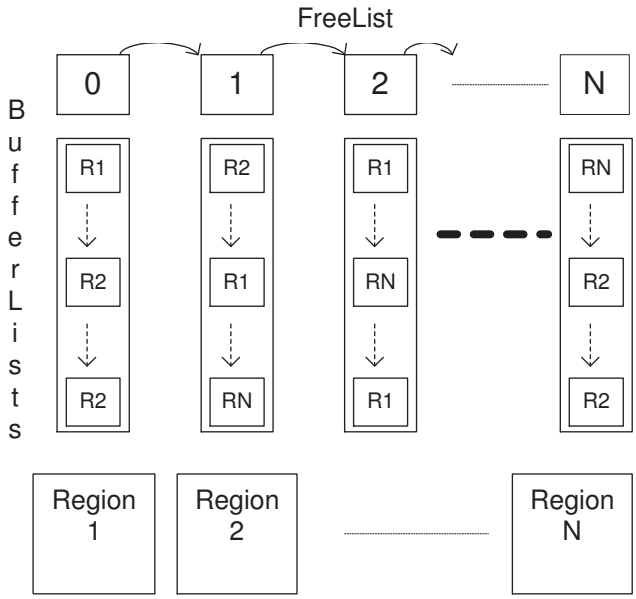



FIG. 2.7. *The First Implementation of Buffer Pooling*

Our first implementation (hereafter called *Buddy1*) is developed with the aim of keeping a small memory footprint for the application. This is possible because a buffer only needs to know its offset in order to find its buddy. This offset can be stored at the start of the allocated memory chunk. If a user requests $s$ bytes, the first strategy allocates $s + BUDDY\_OVERHEAD$ bytes buffer. The additional $BUDDY\_OVERHEAD$ bytes will be used to store the buffer offset. Also, the data structures do not store buffer abstractions like `mpjbuf.NIOBuffer` in the linked lists.

Figure 2.7 outlines the implementation details of our first pooling strategy. `FreeList` is a list of `BufferLists`, which contains buffers at different levels. Here, level refers to the different sizes of buffer available. If a buffer is of size $s$, then its corresponding level will be $\lceil \log_2(s) \rceil$. Initially, there is no region associated with `FreeList`. An initial chunk of memory of size $M$ is allocated. At this point, `BufferLists` are created starting from 0 to $\log_2(M)$. When buddies are merged, a buffer is added to the `BufferList` at the higher level and the buffer itself and its buddy are removed from the `BufferList` at the lower level. Conversely, when a buffer is divided to form a pair of buddies, a newly created buffer and its buddy are added to the `BufferList` at the lower level while removing a buffer that is divided from the higher level `BufferList`.

An interesting aspect of this implementation is that `FreeList` and `BufferLists` grow as new regions are created to match user requests.

Our second implementation (hereafter called *Buddy2*) stores higher-level buffer abstractions (`mpjbuf.NIOBuffer`) in `BufferLists`. Unlike the first strategy, each region has its own `FreeList` and has a pointer to the next region as shown in Figure 2.8. While finding an appropriate buffer for a user, this implementation works sequentially starting from the first region until it finds the requested buffer or creates a new region. We expect some overhead associated with this sequential search. Another downside of this implementation is a bigger memory footprint.

**3. The Implementation of Derived Datatypes in MPJ Express.** Derived datatypes were introduced in the MPI specification to allow communication of heterogeneous and non-contiguous data. It is possible to achieve some of the same goals by communicating Java objects, but there are concerns about the cost of object serialization—MPJ Express relies on the JDK's default serialization.

Figure 3.1 shows datatype related class hierarchy in MPJ Express. The superclass `Datatype` is an abstract class that is implemented by five classes. The most commonly used implementation is `BasicType` that provides initialization routines for basic datatypes including Java objects.

There are four types of derived datatypes: contiguous, indexed, vector, and struct. Figure 3.1 shows an implementation for each derived datatype including `Contiguous`, `Indexed`, `Vector`, and `Struct`.

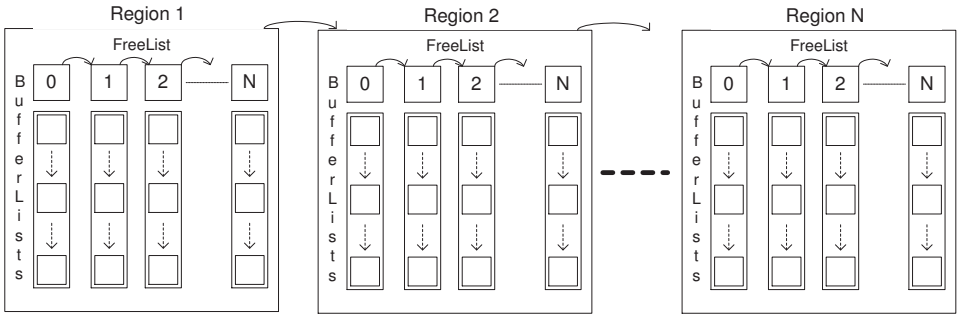



FIG. 2.8. *The Second Implementation of Buffer Pooling*

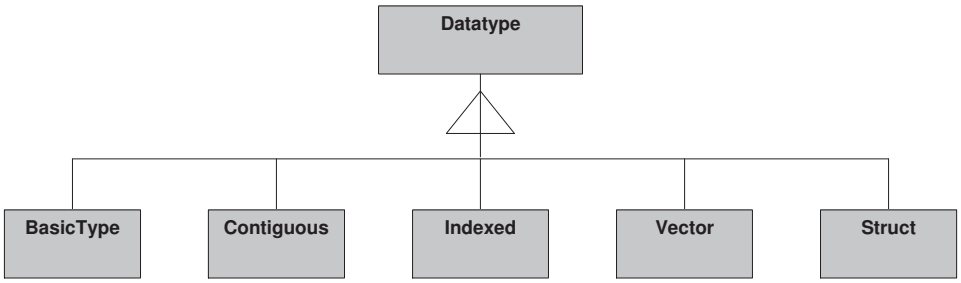



FIG. 3.1. *The Datatype Class Hierarchy in MPJ Express*

The MPJ Express library makes extensive use of the buffering API to implement derived datatypes. Each datatype class contains a specific implementation of the `Packer` interface. The class hierarchy for the implementation of different `Packer`s is shown in Figure 3.2.

Recall that the buffering API provides three kinds of read and write operations. These methods are normally available for use through classes that implement the `Packer` interface. We discuss various implementations that in turn rely on variants of read and write methods provided by the buffering API. The first are the normal `write()` and `read()` methods. The implementation of the `Packer` interface that uses this set of methods is the template `SimplePackerType`. The templates are used to generate Java classes for all primitive datatypes and objects. The second template class that is an implementation of the `Packer` interface is called `GatherPackerType` that uses `gather()` and `scatter()` methods of `mpjbuf.Buffer` class. The last template class is `MultiStridedPackerType` that uses the third set of methods provided by the buffering layer namely `strGather()` and `strScatter()`. Other implementations of the Packer interface are `NullPacker` and `GenericPacker`. The classes like `ContiguousPacker`, `IndexedPacker`, `StructPacker`, and `VectorPacker` in turn implement the abstract `GenericPacker` class.

Figure 3.3 shows the main packing and unpacking methods provided by the `Packer` interface.

The `Packer` interface is used by the sending and receiving methods to pack and unpack the messages. Consider the example of sending a message consisting of an array of integers. In this case, the datatype argument used for the standard a `Send()` method is `MPI.INT` that is an instance of `BasicType` class. A handle to a related `Packer` object can be obtained by calling the method `getPacker()`. The object `MPI.INT` is also used to get a reference to `mpjbuf.Buffer` instance by invoking the method `createWriteBuffer()`. Later a variant of the `pack()` method, shown in Figure 3.3, is used to pack the message onto a buffer that is used for communication by the underlying communication devices.

Similarly, when receiving a message with `Recv()`, the datatype object like `MPI.INT` is used to get the reference to an associated `Packer` object. A variant of the `unpack()` method is used unpack the message from `mpjbuf.Buffer` onto the user specified array.

The *contiguous datatype* consists of elements that are of same the type and at contiguous locations. Figure 3.4 shows a contiguous datatype with four elements. Each element of this datatype consists of an array of five elements. Although each element is shown as a row in a matrix, physically the datatype will be stored at contiguous locations such as an array of byte buffer.

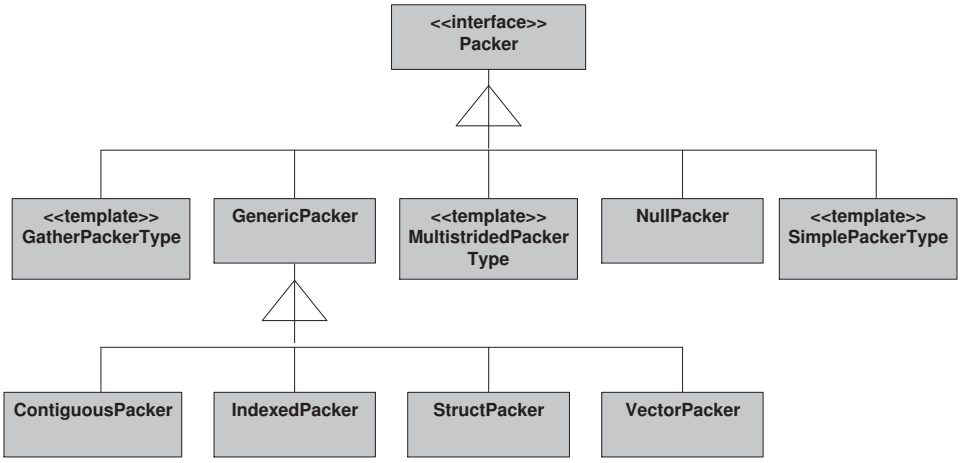



FIG. 3.2. *The Packer Class Hierarchy in MPJ Express*

```
public interface Packer {

  .. .. ..

  public abstract void pack(mpjbuf.Buffer mpjbuf, Object msg, int offset)
  public abstract void pack(mpjbuf.Buffer mpjbuf, Object msg, int offset,
                            int count)

  public abstract void unpack(mpjbuf.Buffer mpjbuf, Object msg, int offset)
  public abstract void unpack(mpjbuf.Buffer mpjbuf, Object msg, int offset,
                              int count)

  public abstract void unpack(mpjbuf.Buffer mpjbuf, Object msg, int offset)
  public abstract void unpack(mpjbuf.Buffer mpjbuf, Object msg, int offset,
                              int count)

  public abstract void unpackPartial(mpjbuf.Buffer mpjbuf, int length,
                                     Object msg, int offset)

  .. .. ..

}
```

FIG. 3.3. *The Packer Interface*

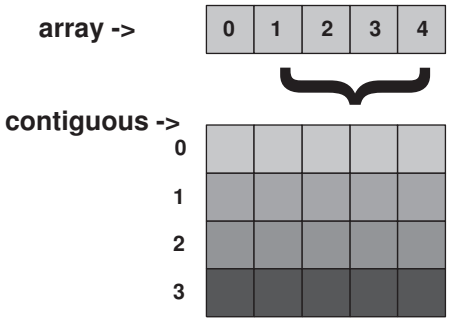



FIG. 3.4. *Forming a Contiguous Datatype Object*

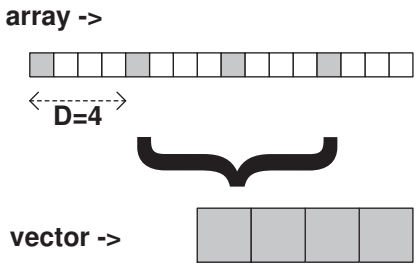



FIG. 3.5. *Forming a Vector Datatype Object*

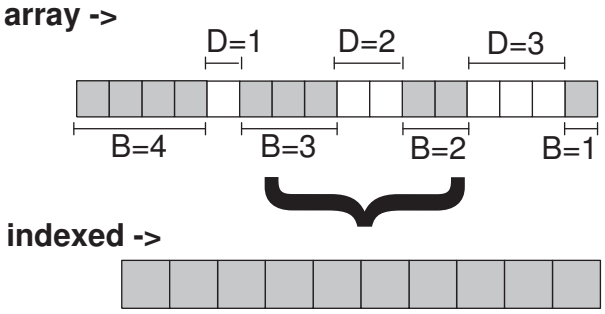



FIG. 3.6. *Forming an Indexed Datatype Object*

The *vector datatype* consists of elements that are of the same type and are found at non-contiguous locations. Figure 3.5 shows how to build an element of vector datatype from an array of primitive datatype with `blockLength=1` and `stride=4` (labelled `D` in the figure). The data is copied onto a contiguous section of memory before the actual transfer.

A more general datatype is *indexed* that allows specifying multiple block lengths and strides (also called displacement). An example is shown in Figure 3.6 with increasing displacement (starting from 0 and labelled `D`) and decreasing block length (starting from 4 and labelled `B`). The most general datatype is *struct* that not only allows varying block lengths and strides but also different basic datatypes, unlike the indexed datatype.

**4. Performance Evaluation.** In this section we first evaluate the performance of our buffering layer focusing on the allocation time. This is followed by a comparison of MPJ Express using combinations of direct and indirect byte buffers with our pooling strategies to find out which technique provides the best performance. We calculate transfer time and throughtput for increasing message sizes to evaluate buffering techniques. Towards the end of the section, we evaluate the performance of MPJ Express against MPICH-MX and mpiJava on Myrinet. Again, this test requires the calculation of transfer time and throughput for different message sizes.

The transfer time and throughput is calculated using a modified ping-pong benchmark. While using conventional ping-pong benchmarks, we noticed variability in timing measurements. The reason is that the network card drivers used on our cluster have a higher network latency—64 $\mu$s. The network latency of the card drivers is an attribute that determines the polling interval for checking new messages. In our modified technique, we introduced random delays before the receiver sends the message back to the sender. Using this approach, we were able to negate the effect of network card latency.

The test environment for collecting the performance results was a cluster at the University of Portsmouth consisting of 8 dual Intel Xeon 2.8 GHz PCs using the Intel E7501 chipset. The PCs were equipped with 2 Gigabytes of ECC RAM with 533 MHz Front Side Bus (FSB). The motherboard (SuperMicro X5DPR-iG2) was

equipped with 2 onboard Intel Gigabit LAN adaptors with one 64-bit 133 MHz PCI-X slot and one 64-bit 66 MHz PCI slot. The PCs were connected together through a 24-port Ethernet switch. In addition, two PCs were connected back-to-back via the onboard Intel Gigabit adaptors. The PCs were running the Debian GNU/Linux with the 2.4.32 Linux kernel. The software used for the Intel Gigabit adaptor was the proprietary Intel e-1000 device driver. The JDK version used for tests on mpiJava and MPJ Express was Sun JDK 1.5 (Update 6). The C compiler used was GNU GCC 3.3.5.

**4.1. Buffering Layers Performance Evaluation.** In this section, we compare the performance of our two buffering strategies with direct allocation of `ByteBuffer`s. We are also interested in exploring the performance difference between using direct and indirect byte buffers in MPJ Express communication methods. There are six combinations of our buffering strategies that will be compared in our first test—Buddy1, Buddy2, and a simple allocation scheme, each using direct and indirect byte buffers.

**4.1.1. Simple Allocation Scheme Time Comparison.** In our first test, we compare isolated buffer allocation times for our six allocation approaches. Only one buffer is allocated at one time throughout the tests. This means that after measuring allocation time for a buffer, it is de-allocated in the case of our buddy schemes (forcing buddies to merge into original chunk of 8 Mb before the next allocation occurs), or the reference is freed in the case of straightforward `ByteBuffer` allocation.

Figure 4.1 shows a comparison of allocation times. It should first be noted that all the buddy-based schemes are dramatically better than relying on the JVMs management of `ByteBuffer`. This essentially means that without a buffer pooling mechanism, creation of intermediate buffers for sending or receiving messages in a Java messaging system can have detrimental effect on the performance. Results are averaged over many repeats, and the overhead of garbage collection cycles are included in the results in an averaged sense; this is a fair representation of what will happen in a real application. Generally we attribute the dramatic increase in average allocation time for large `ByteBuffer`s to forcing proportionately many garbage collection cycles. All the buddy variants (by design) avoid this overhead. The allocation times for buddy based schemes decrease for larger buffer sizes because less time is spent in traversing the data structures to find an appropriately sized buffer. The size of the initial region is 8 Mb—resulting in the least allocation time for this buffer size. The best strategy in almost all cases is Buddy1 using direct buffers.

Quantitative measurements of the memory footprint suggest the current implementation of Buddy2 also has about a 20% larger footprint because of the extra objects stored. In its current state of development, Buddy2 is clearly outperformed by Buddy1. But there are good reasons to believe that with further development, a variant of Buddy2 could be faster than Buddy1. This will be the subject of future work.

**4.1.2. Incorporating Buffering Strategies into MPJ Express.** In this test, we compare transfer times and throughput measured by a simple ping-pong benchmark using each of the different buffering strategies. These tests were performed on Fast Ethernet. The reason for performing this test is to see if there are any performance benefits of using direct `ByteBuffer`s. From the source-code of the NIO package, it appears that the JVM maintains a pool of direct `ByteBuffer`s for internal purposes. These buffers are used for reading and writing messages into the socket. A user provides an argument to `SocketChannel`'s `write()` or `read()` method. If this buffer is direct, it is used for writing or reading messages. If this buffer is indirect, a direct byte buffer is acquired from direct byte buffer pool and the message is copied first before writing it to or reading it from the socket. Thus, we expect to see an overhead of this additional copying for indirect buffers.

Figure 4.2 shows transfer time comparison on Fast Ethernet with different combinations of buffering in MPJ Express. Normally transfer time comparison is useful for evaluating the performance on smaller messages. We do not see any significant performance difference for small messages.

Figure 4.3 shows the throughput comparison. Here, MPJ Express achieves maximum throughput when using direct buffer in combination with either of the buddy implementations. We expect to see this performance overhead related to indirect buffers to be more significant for faster networks like Gigabit Ethernet and Myrinet. The drop in throughput at 128 Kb message size is because of the change in communication protocol from eager send to rendezvous.

**4.2. Evaluating MPJ Express using Myrinet.** In this test we evaluate the performance of MPJ Express against MPICH-MX and mpiJava by calculating the transfer time and throughput. We used MPJ Express (version 0.24), MPICH-MX (version 1.2.6..0.94), and mpiJava (version 1.2.5) on Myrinet. We also added *mpjdev* [13] to our comparison to better understand the performance of MPJ Express. MPJ Express uses

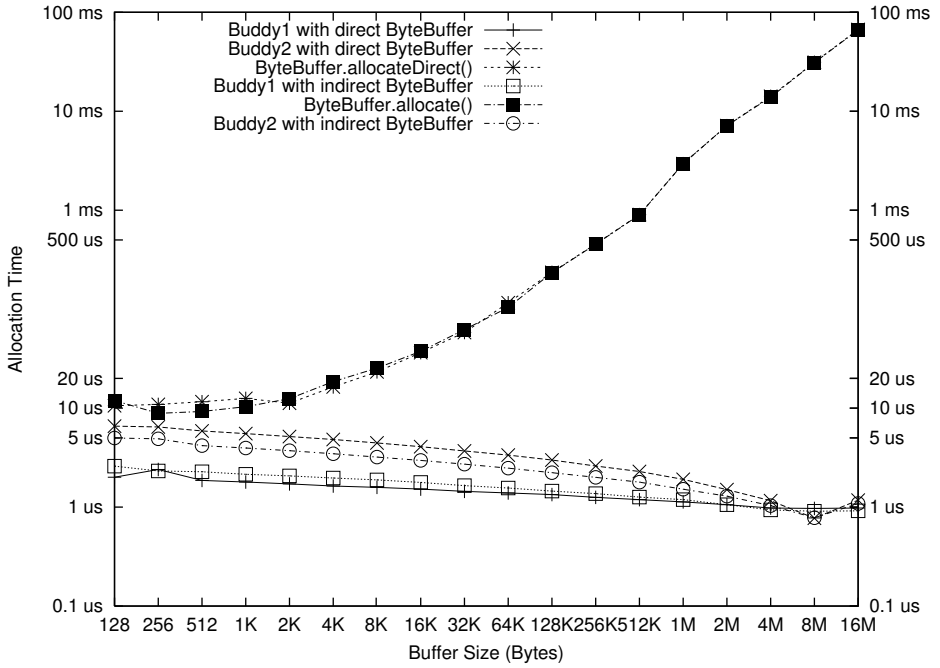



FIG. 4.1. *Buffer Allocation Time Comparison*

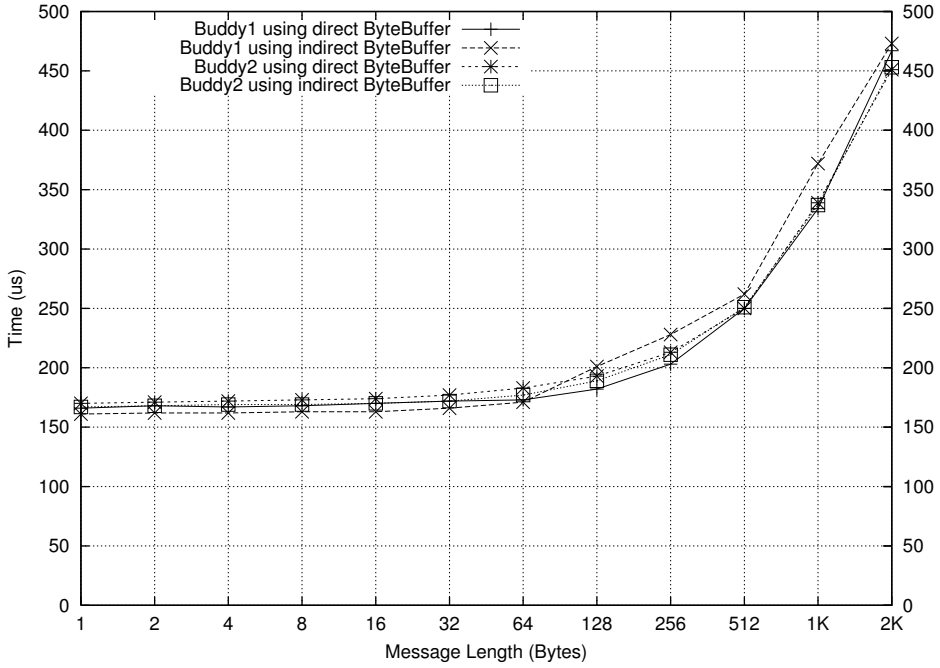



FIG. 4.2. *Transfer Time Comparison on Fast Ethernet*

mpjdev, which in turn relies on *mxdev* on Myrinet. These tests were conducted on the same cluster using the 2 Gigabit Myrinet eXpress (MX) library [17] version 1.1.0.

Figure 4.4 and Figure 4.5 show the transfer time and throughput comparison. The latency of MPICH-MX is 4 $\mu$s. MPJ Express and mpiJava have a latency of 23 $\mu$s and 12 $\mu$s, respectively. The maximum throughput of MPICH-MX was 1800 Mbps with 16 Mbyte messages. MPJ Express achieves a maximum of 1097 Mbps for the same message size. mpiJava achieves a maximum of 1347 Mbps for 64 Kbyte messages. After this,

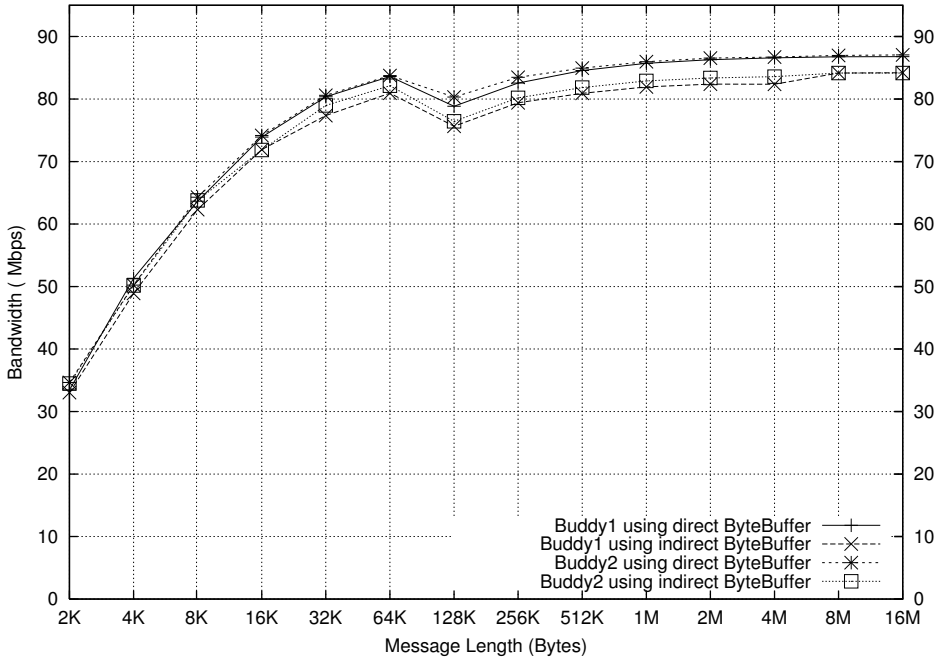



FIG. 4.3. *Throughput Comparison on Fast Ethernet*

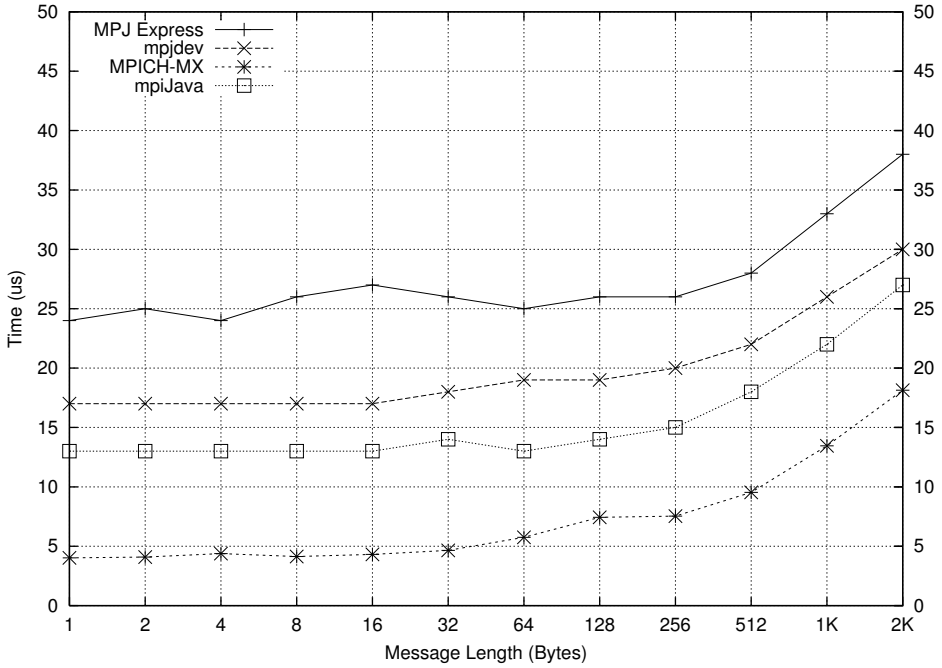



FIG. 4.4. *Transfer Time Comparison on Myrinet*

there is a drop, bringing throughput down to 868 Mbps at 16 Mbyte message. Throughput starts decreasing as the message size increases from 64 Kbytes. This is primarily due to copying data between the JVM and OS. Although we are using JNI in the MPJ Express Myrinet device, we have been able to avoid this overhead by using direct byte buffers. However, other overheads of JNI, such as the increased calling time of methods are visible in the results. The mpjdev device, that sits on top of mxdev, attains a maximum throughput of 1826 Mbps for 16 Mbyte messages, which is more than that of MPICH-MX. Our device layer is able to make the most

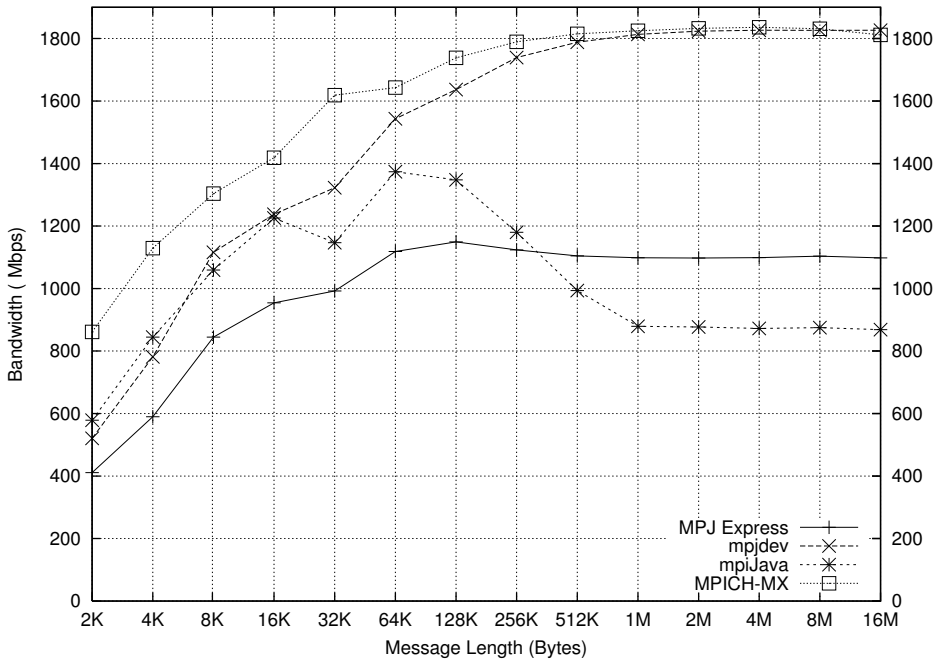



FIG. 4.5. *Throughput Comparison on Myrinet*

of Myrinet. This shows the usefulness of our buffering API, because the message has already been copied onto a direct byte buffer. It is clear that a combination of using direct byte buffers and JNI incurs virtually no overhead.

The difference in the performance of MPJ Express and mpjdev shows the packing and unpacking overhead incurred by the buffering layer. Besides this overhead, this buffering layer helps MPJ Express to avoid the main data copying overhead of JNI—MPJ Express achieves a greater bandwidth than mpiJava. Secondly, such a buffering layer is necessary to provide communication of derived datatypes. A possible fix for this overhead is to extend mpiJava 1.2 and the MPJ API to support communication to and from `ByteBuffer`s.

**5. Conclusions and Future Work.** MPJ Express is our implementation of MPI-like bindings for the Java language. As part of this system, we have implemented a buffering layer that exploits direct byte buffers for efficient communication on proprietary networks, such as Myrinet. Using this kind of buffer has enabled us to avoid the overheads of JNI. In addition, our buffering layer helps to implement derived datatypes in MPJ Express. Arguably, communicating Java objects can achieve the same effect as communicating derived types, but we have concerns related to notoriously slow Java object serialization.

In this paper, we have discussed the design and implementation of our buffering layer, which uses our own implementation of buddy algorithm for buffer pooling. For a Java messaging system, it is useful to rely on an application level memory management technique instead of relying on the JVM's garbage collector, because constant creation and destruction of buffers can be a costly operation. We benchmarked our two pooling mechanisms against each other using combinations of direct and indirect byte buffers. We found that one of the pooling strategies (Buddy1) is faster than the other with a smaller memory footprint. Also, we demonstrated the performance gain of using direct byte buffers. We have evaluated the performance of MPJ Express against other messaging systems. MPICH-MX achieves the best performance followed by MPJ Express and mpiJava. By noting the difference between MPJ Express and the mpjdev layer, we have identified a certain degree of overhead caused by additional copying in our buffering layer. We aim to resolve this problem by introducing methods that communicate data to and from `ByteBuffer`s.

We released a beta version of our software in early September 2005. The current version provides communication functionality based on a thread-safe Java NIO device.The current release also contains our buffering API with the two implementations of buddy allocation scheme. This API is self-contained and can be used by other Java applications for application level explicit memory management.

MPJ Express can be downloaded from `http://mpj-express.org`.

REFERENCES

[1] M. Aldinucci, M. Danelutto, and P. Teti, *An advanced environment supporting structured parallel programming in Java*, Future Generation Computer Systems, 19 (2003), pp. 611–626.
[2] M. Alt and S. Gorlatch, *A prototype grid system using Java and RMI*, in Parallel Computing Technologies (PaCT 2003), vol. 2763 of Lecture Notes in Computer Science, Springer, 2003, pp. 401–414.
[3] M. Baker, B. Carpenter, G. Fox, S. H. Ko, and S. Lim, *An Object-Oriented Java interface to MPI*, in International Workshop on Java for Parallel and Distributed Computing, San Juan, Puerto Rico, April 1999.
[4] B. Blount and S. Chatterjee, *An Evaluation of Java for Numerical Computing*, in ISCOPE, 1998, pp. 35–46.
[5] O. Bonorden, J. Gehweiler, and F. M. auf der Heide, *A Web computing environment for parallel algorithms in Java*, Scalable Computing: Practice and Experience, 7 (2006).
[6] B. Carpenter, G. Fox, S.-H. Ko, and S. Lim, *mpiJava 1.2: API Specification*.
[7] C.-C. Chang and T. von Eicken, *Javia: A Java Interface to the Virtual Interface Architecture*, Concurrency—Practice and Experience, 12 (2000), pp. 573–593.
[8] M. Danelutto and P. Dazzi, *Joint structured/unstructured parallelism exploitation in muskel*, in Third International Workshop on Practical Aspects of High-Level Parallel Programming (PAPP 2006), V. N. Alexandrov et al., eds., vol. 3992 of Lecture Notes in Computer Science, Springer, 2006, pp. 937–944.
[9] R. Gordon, *Essential JNI: Java Native Interface*, Prentice Hall PTR, Upper Saddle River, NJ 07458, 1998.
[10] Y. Gu, B.-S. Lee, and W. Cai, *JBSP: A BSP programming library in Java*, Journal of Parallel and Distributed Computing, 61 (2001), pp. 1126–1142.
[11] R. Hitchens, *Java NIO*, O'Reilly & Associates, 2002.
[12] D. Knuth, *The Art of Computer Programming: Fundamental Algorithms*, Addison Wesley, Reading, Massachusetts, USA, 1973.
[13] S. Lim, B. Carpenter, G. Fox, and H.-K. Lee, *A Device Level Communication Library for the HPJava Programming Language*, in IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2003), November 2003.
[14] Mark Baker, Bryan Carpenter, Aamir Shafi, *MPJ Express: Towards Thread Safe Java HPC*, in Proceedings of the 2006 IEEE International Conference on Cluster Computing (Cluster 2006), IEEE Computer Society, September 2006, pp. 1–10.
[15] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard*, University of Tenessee, Knoxville, TN, June 1995. `http://www.mcs.anl.gov/mpi`
[16] *Myricom*. http://www.myri.com.
[17] *The MX (Myrinet eXpress) library*. `http://www.myri.com/scs/MX/mx.pdf`
[18] M. Welsh and D. Culler, *Jaguar: Enabling Efficient Communication and I/O in Java*, Concurrency: Practice and Experience, 12 (2000), pp. 519–538.

*Edited by:* Anne Benoît and Frédéric Loulergue
*Received:* September, 2006
*Accepted:* March, 2007

Scalable Computing: Practice and Experience
Volume 8, Number 4, pp. 359–371. http://www.scpe.org
ISSN 1895-1767


# RAPID AREA-TIME ESTIMATION TECHNIQUE FOR PORTING C-BASED APPLICATIONS ONTO FPGA PLATFORMS*

MY CHUONG LIEU†, SIEW KEI LAM†, THAMBIPILLAI SRIKANTHAN†

**Abstract.** High-level area-time estimation is an essential step to facilitate rapid design exploration for FPGA implementations. Existing works in high-level area-time estimation usually ignore the physical effects of the design after place and route, which have a notable impact on the maximum achievable speed of the design. In this paper, we propose a framework to rapidly estimate the area-time measures of mapping C-applications onto FPGA. The framework relies on the Trimaran compiler to generate an optimized high-level IR (Intermediate Representation) of the C-applications. Area-time estimation of the IR is then performed using a proposed estimation model that is based on an architecture template with application-specific heterogeneous functional units. In order to accurately predict the delay of the design after place and route, we introduce a new metric for the estimation that models the criticality of the design's interconnectivity. Experimental results based on a set of embedded functions show that the proposed area estimation can achieve comparable results with the synthesis results of a commercial FPGA tool in the order of milliseconds. For the C functions used in our experiments, the proposed delay estimation leads to an average error of about 3% when compared to the post place and route results. In addition, we demonstrate the robustness of the proposed framework which provides consistent results for different FPGA families. The contribution of this paper is a scalable methodology for rapid estimation of cost-benefit metrics of C-based algorithms to be accelerated on FPGA-based high-performance computing platform.

**Key words.** FPGA, C-based application, area time estimation, hardware accelerator

**1. Introduction.** FPGAs (Field-Programmable Gate Arrays) have become an attractive solution to meet the technological and market challenges in embedded processing. Traditional hybrid platforms that incorporate ASIC and microprocessors are migrating towards FPGA platforms (e.g. Xilinx Virtex-II Pro [1] and Altera Stratix [2]) to take advantage of the reconfigurable benefits of FPGA. This trend is supported by the availability of efficient EDA (Electronic Design Automation) tools and the increasingly stringent TTM (Time-To-market) requirements. In order to exploit the strengths in both the microprocessor and FPGA, efficient hardware-software partitioning strategies must be incorporated in the emerging design flows. However, commercially available design flows do not enable designers to make design explorations for effective hardware-software partitioning. This is chiefly due to a lack of an essential step that can estimate the performance-cost for mapping a software component to hardware early in the design cycle.

In this paper, we propose a framework that can rapidly and accurately estimate the hardware area-time measures for implementing C-applications onto the FPGA. We have chosen C as the input to our framework as it is widely used for embedded processing. The front-end of the framework relies on the high-level optimization and scheduling capabilities of the Trimaran compiler infrastructure [3]. In order to facilitate effective area-time estimations, we have adopted an architecture template for implementing the applications, which is similar to the one proposed in [4]. The architecture template resembles a VLIW-like architecture that incorporates application-specific heterogeneous functional units and register-files, with dedicated interconnection buses.

High-level estimation is performed using an area-time estimation model, which relies on a set of pre-characterized parameters of the components in the architecture template. Previously reported works in high-level area-time estimation often do not consider the interconnect delay of the design after place and route. We will demonstrate that this oversight will lead to high uncertainties in the estimation results. Our proposed approach overcomes this limitation by incorporating a new metric that models the placement complexity of the design's interconnectivity.

The paper is organized as follows. In the following section, we describe related works in the area of high-level estimation for FPGA implementation. This is followed by an overview of the proposed framework. Section 4 describes the parametric characterization of the architecture template components, and the proposed area-time estimation models. Next, results analysis is provided to demonstrate the benefits of our framework, and we conclude in Section 6.

**2. Related Works.** Due to the need to expedite the development of complex applications in hardware, a number of commercial tools that synthesizes high-level languages to FPGA have emerged in recent years. These tools differ in several aspects such as high-level language support, high-level optimization features and the target

*This research is supported by Infineon Collaboration Fund
†Center for High Performance Embedded System, Nayang Technological University, Singapore {lieu0003, assklam, astsrikan}@ntu.edu.sg}

system. For example, Mitrion-C from Mitronics [5] and RCToolbox from DSPLogic [7] supports the Mitrion-C and Matlab programming language respectively, while HandleC from Celoxica [6] and Impulse-C from Impulse Accelerated Technologies [8] support a subset of the ANSI-C language that is extended with constructs for specifying the hardware definitions. These tools cannot be directly employed for most embedded applications that are programmed using ANSI-C. Although the C2H tool from Altera [9] supports ANSI-C applications, the hardware representations that are generated are specific to the Altera FPGA devices only. This limits the generality of the tool across different platforms. Other tools such as Catapult from Mentor Graphics [11] and Trident [10] support ANSI applications and are not device specific.

The problem of high-level area-time estimation for hardware implementations has received considerable interest in the research community for nearly 20 years. Research efforts in this area are motivated by the need to evaluate the hardware performance-cost indices of various design options early in the design phase, in order to reduce the time-consuming implementation cycles. Figure 2.1 highlights the major steps in a typical high-level estimation flow, which are 1) Transformation of application written in high-level language to IR; 2) Architecture independent estimation; 3) Architecture independent synthesis; 4) Architecture dependent estimation. It is worth noting that previously reported works typically do not address all these steps in their area-time estimation approach. The first step in the estimation flow typically involves the transformation of a high-level representation of an application (e.g. C, System-C, Matlab, behavioral HDL or JAVA specifications) to an IR (Intermediate Representation). This transformation includes high-level compiler optimizations such as loop transformations to extract the hidden parallelism in the sequential C statements. For example, the SUIF [12] compilers have been widely used to transform C-based application into CFGs (Control Flow Graphs) and DFGs (Data Flow Graphs).

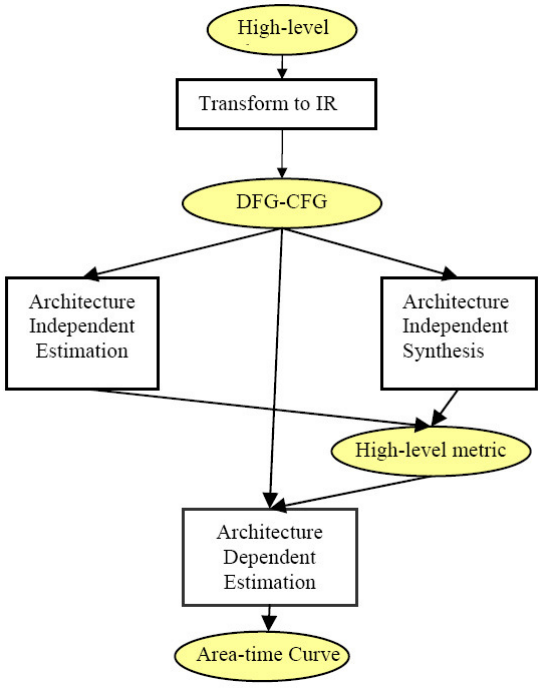



Fig. 2.1. *High-Level Estimation FLow*

*Architecture independent estimation* attempts to calculate the hardware resources and latency in terms of clock cycles without performing scheduling. These approaches commonly rely on probabilities or integer linear programming models based on the high-level application characteristics in order to predict the number of functional-units or minimum clock cycles. The work presented in [13] performs architecture independent estimation to obtain a lower-bound execution time of a DFG in the presence of hardware constraints to facilitate efficient design space exploration. Similarly, [14] estimates the minimum number of resources that are required to execute a DFG within a control step constraint. Others [15] employ Matlab codes as inputs to estimate the

hardware resources by summing the area of the required operators based on the execution probabilities in the application.

*Architecture independent synthesis* typically performs scheduling of the IR and resource binding to obtain accurate high-level metrics, which include the number of clock cycles and hardware resources. The hardware area-time of the application is then calculated or estimated from these metrics. The architecture independent synthesis approach in [16] considers the effects of various loop transformation techniques. Bilavarn et al [17] presented a method that employs architecture independent synthesis for design exploration. However, maximum clock frequency estimation was based solely on the longest latency of the execution unit and ignores post place and route physical effects. Cardoso proposed a methodology for estimating FPGA implementations of Java byte-codes in [18]. He highlighted the limitations of high-level delay estimation due to the lack of circuit details.

*Architecture dependent estimation* techniques commonly employ simple hardware models to speed-up the estimation process. The hardware cost is estimated in terms of LUTs (Look-Up Tables), while the performance is often estimated in terms of clock latencies. The technique presented in [19] adopts an analytical approach to estimate the FPGA area for implementing the DFG. The estimation is based on a set of formulas that models the components and corresponding hardware area of the DFG operations. Their approach reported credible results with a maximum error of 10% and the estimation can be achieved in the order of milliseconds. However, they have not considered delay estimation. In [15] , the number of required flip-flops is estimated by calculating the maximum number of required registers. The Rent Rule and Feuer's formula have been employed to estimate the post place and route interconnect delay. This approach leads to large uncertainty of up to 9 ns. This high uncertainty can become unacceptable for designs that need to be clocked at high frequency (e.g. 100 MHz). It is noteworthy that our proposed area-time estimation accounts for the physical implementation characteristics and is not susceptible to the speed of the design.

**3. Overview of Framework.** Figure 3.1 describes an overview of the proposed framework for high-level area-time estimation. The open-source Trimaran compiler infrastructure, which supports state of the art compiler research in ILP (Instruction Level Parallelism) based architectures, is relied upon to expose the hidden parallelism in the sequential C statements, and to perform high-level optimizations and scheduling [3]. This front-end process takes several seconds (typically less than 10s for 1 single C-function compilation) depending on applications and functions sizes. We have adopted the application-specific architecture template that is similar to the one proposed in [4] as shown in figure 3.2. It is worth mentioning that this architecture template can be adapted for pipelined and non-pipelined data-paths by configuring the application-specific interconnection. The Trimaran machine description is augmented with the heterogeneous functional units in the architecture template. These functional units include a combination of two or more basic operators (e.g. adder, shifter, multiplier, logic operator, comparator and memory-access unit). Only the functional units that are required for a particular application will be incorporated into the architecture.

The output of the Trimaran is an ILP schedule of the application (e.g. the type of functional units that will be executed in each clock cycle and the data-dependency between these functional units). Based on this schedule information, we perform a simple hardware binding process that attempts to bind operations with the most common input-outputs to the same functional units. This aims to reduce the complexity of the interconnectivity between registers and functional units. In order to perform area-time estimation, information pertaining to the control-path and data-path are segregated from the ILP schedule after hardware binding. In this paper, we focus on area-time estimation for the data-path only.

A one-time area-time characterization of the components in the architecture template is required to facilitate area-time estimation of the data-path. An estimation model is then employed along with this information to estimate the performance-cost measures of the application by taking into account the physical implementation effects. In order to evaluate the accuracy of our estimation approach, a process to auto-generate the RTL (Register Transfer Level) codes from the control-data path information has been incorporated in the framework. The RTL code can then be subjected to the FPGA implementation tool (i. e. Xilinx ISE) to obtain the actual post place and route report for results comparison with the proposed estimation approach.

**4. Area-Time Estimation.** In this section, we will provide detailed description of the process to characterize the components and the proposed area-time estimation model.

**4.1. Hardware Characterization of Architecture Template's Components.** We have used the Xilinx ISE synthesis engine to characterize the hardware components in the architecture template and other relevant information. Table 4.1 illustrates the hardware components (and other relevant information) and the

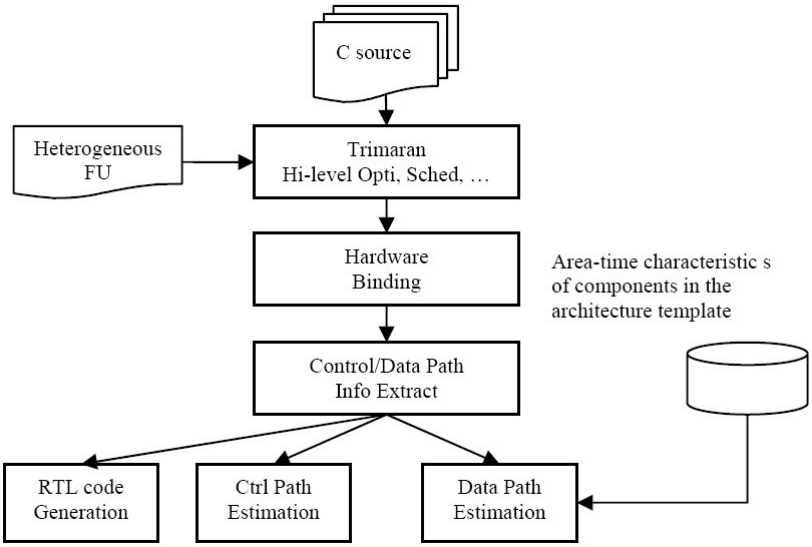



FIG. 3.1. *Propose High-level Estimation Framework*

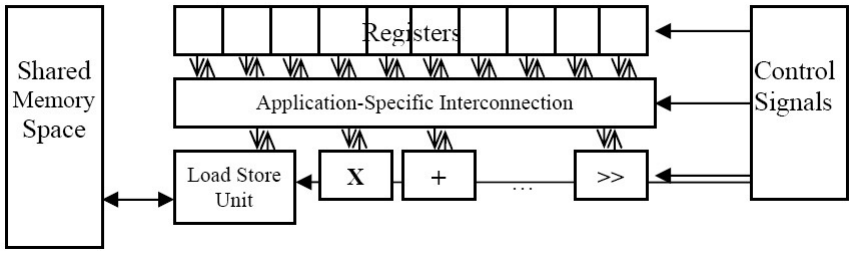



FIG. 3.2. *VLIW-like architechture Template*

corresponding area-time measures for the Virtex-II Pro device (xc2vp70-6ff1704). The data-paths are assumed to be 16-bit or 32-bit, as the hardware implementation serves to accelerate the base ISA (Instruction Set Architecture) operations of the microprocessor.

In order to perform interconnect characterization, we have implemented a number of circuits to obtain the average post place and route interconnect delay. The circuits are constructed based on the data-path that is shown in figure 4.1, which resembles the architecture template consisting of a single functional unit. A range of designs, each consisting of up to 8 duplicate circuits similar to the one in figure 4.1, is subjected to physical implementation using the Xilinx ISE tool. We utilized the Xplore Script provided by Xilinx [20] that iteratively executes the place-and-route process to achieve the maximum clock speed. Figure 4.2 shows the maximum delay for the range of designs from which we calculated the average delay after place and route. The average interconnect delay is then computed using Equation 4.1, where Logic Delay is the sum of Clk2q, Mux4to1, functional unit delay, and FF-Set-up time that are listed in Table 4.1. The average interconnect-delay for the target device is found to be 0.42 ns, and this value will be used by the proposed method for delay estimation.

$$InterconnectDelay = (Avg(MaxDelay) - LogicDelay)/3 \tag{4.1}$$

It is noteworthy that the proposed high-level estimation approach can also be adopted for different target FPGA families by performing a one-time hardware characterization for the particular device.

**4.2. Area Estimation.** In order to perform area estimation, we obtained the number of functional units and the number of registers (flip-flops) from the ILP schedule after hardware binding. The estimated number of

TABLE 4.1
*Characterized Components for Xilinx FPGA Virtex2p-6*

| Components | Area(LUT) | Delay (ns) |
|---|---|---|
| 16bit-Addsub | 16 | 2.139 |
| 16-bit multiplier by LUT | 121 | 8.165 |
| 16-bit left/right/signed-unsigned shifter | 76 | 2.376 |
| 16-bit Logic Operator (4 operations) | 16 | 0.313 |
| 16-bit Multiplexer 2 to 1 | 16 | 0.313 |
| 16-bit Multiplexer 4 to 1 | 32 | 0.653 |
| 16-bit Multiplexer 8 to 1 | 64 | 0.972 |
| 16-bit Multiplexer 16 to 1 | 128 | 1.291 |
| FFCLK2Q (Clock to output delay of FF) | | 0.234 |
| FFSetup (Setup time of FF) | | 0.243 |

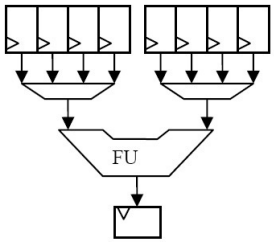



FIG. 4.1. *Sample Circuit for Characterizing Average Interconnect Delay*

LUTs is computed by summing up the number of LUTs for the functional units based on the pre-characterized information. As each slice of Xilinx Virtex2 pro contains 2 Look-up tables (LUT) and 2 flip-flops, we estimate the total slices as in 4.2:

$$Estimated\_number\_of\_slices = (\#LUT + \#FF)/2 \tag{4.2}$$

The estimation for the number of slices assumes that each slice is fully utilized to implement the functional units and registers. Experimental results reveal that the proposed estimation method is very accurate for LUT and Flip-Flop estimation. In addition, the estimation of the slices is comparable with ISE logic synthesis results and results reported in previous work [15].

**4.3. Delay Estimation.** The difficulty in delay estimation lies in the prediction of the interconnect delay before the physical design steps (i. e. placement and routing). There have been several reported works in the area of interconnect delay estimation such as [21] [22] and we will briefly discuss them before describing the proposed delay estimation approach. It is worth mentioning that these previous works are not integrated as part of a high-level estimation framework, but are used mainly to aid optimization decisions in the CAD flow. The work presented in [21] can achieve very accurate estimations of the interconnect delay by analyzing the physical characteristics of the designs. However, due to the complexity of the approach, the estimation results are achieved in the order of seconds and minutes. Karnik and Kang [23] presented an empirical routing delay model for estimating interconnection delays in FPGA. These methods require low-level metrics of circuit such as net fan-out and routing congestion which is not desirable for efficient high-level estimation [18]. Their method resulted in an estimated delay with 20% errors. Hutton highlighted that that delay estimation based on theoretical models, generally produces inferior results when compared to those computed based on empirical data [24]. Manohararajah et. al. reported an interesting finding that the predictability of FPGA implementation is mainly governed by the placement rather than routing process [25].

The physical characteristics that has been commonly used for post place and route delay estimation includes: 1) Design size [23] [15], 2) Circuit shape [26], and 3) Fan-in/out [23]. The placement and routing effort is mainly influenced by the interconnectivity in the designs rather than the size or shape of the design due to the fine-grained architecture of FPGA. Hence the first two characteristics is not a reliable indicator on the complexity of the place and route process. We have carried out experiments to show that these characteristics do not lead to reliable predictability of the post place and route delay. The fan-in/out of a register is defined as the number

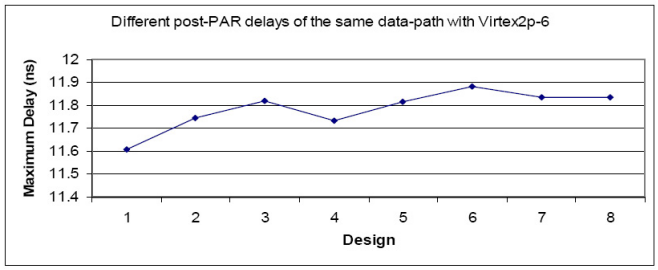



FIG. 4.2. *Maximum post place and route delays for eight designs comprising of multiple of sample circuits (see figure 4.1)*

of input/output connections of that register. Compared to the first two characteristics, the fan-in/out provides a better indication on the interconnect complexity of the design. However, during place and route, the CAD tools often perform register duplication to mitigate the fan in/out effect. Our experiments show that although register-duplication can lead to improved timing in some cases, it could also increase the routing congestion of the circuit. This was inferred in our experiments for some designs, whereby the final delay after register duplication is higher than the delay obtained from implementations that obviates register duplication. Due to this uncertainty, we have assumed that the applications employed in our experiments have moderate register reusability and hence, we do not incorporate the fan-in/out characteristic in our delay estimation model.

In this section, we introduce a delay estimation model that takes into account the post place and route characteristics of the design. Our proposed delay estimation model incorporates a new metric that is based on the relative path delays of the design. In contrast to the method in [21], our proposed method can estimate at a higher abstraction level and achieve reasonable results in less than a second. In addition, we will demonstrate that the maximum estimation error of the proposed model is less than 8% for the experiments considered.

**4.3.1. Proposed Approach.** A path is a connection of a sequence of logic units that begins and ends at a register, as shown in figure 4.3. Let's define Dpath(i) as the delay of a path i in a RTL design, $D_{max}$ as the critical path of that design, and $D_{mean}$ as the mean delay of the paths in the design. Calculations for $D_{path}$, $D_{max}$, and $D_{mean}$ are defined below, which constitutes to a simple delay estimation model. If place and route effects are ignored, the minimum clock period is approximate as $D_{max}$. We compared the estimated delay of 45 random algorithms using the simple delay model with actual results after place and route for the Xilinx Spartan and Virtex-II Pro device. The estimation error shown in figure 4.4 exhibits a consistent error pattern for the two devices. This serves as the motivation for us to use the simple delay model as a basis for post place and route estimation as it can be employed across different devices. The consistency of the delay predictability on the two different devices implies that the inherent characteristics of the design can be used for estimation. The delay of path i (Dpath(i)) for all the paths in the circuit is computed as follows:

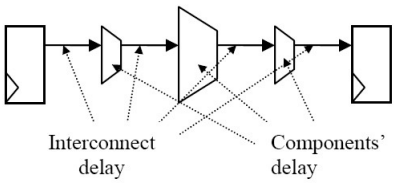



FIG. 4.3. *A typical delay path from register to register*

$$Dpath(i) = FF_{Clk2q} + D_{int} + D_{comp} + FF_{setup}$$

$D_{max}$ = Max of all Dpath

$D_{mean}$ = Mean of all Dpath

$D_{comp}$ is the characterized delay of the component in Table 4.1
(Usually: Mux → Functional Unit → Mux)

$D_{int}$ is the characterized interconnect delay between the components.

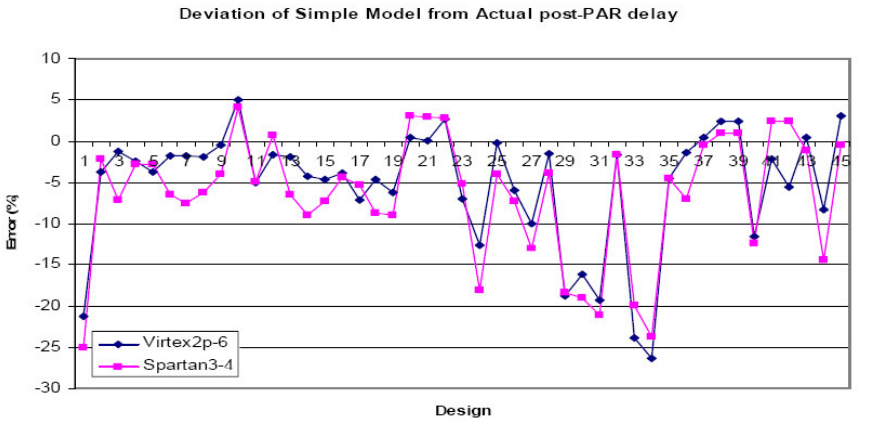



FIG. 4.4. *Estimation error of critical paths for 2 FPGA families*

As mentioned earlier, our proposed delay estimation model incorporates a new metric that is based on the relative path delays of the design. It has been previously reported that the placement process plays a more important role on the predictability of the final delay (assuming that there is no constraints on the number of FPGA routing resources)[25]. In addition, timing-driven placement relies on the criticality of the nets, and hence the effect of the nets criticality can lead to reliable predictability of the interconnection delay. Based on this, we introduce the lambda metric to compute the relative lengths of nets which capture the complexity of the placement effort for a particular design:

$$\lambda = \frac{D_{mean}}{D_{max}} \tag{4.3}$$

$\lambda$ captures the slack distribution of the nets in the designs. If a circuit has one net that is much longer than the rest (i. e. low-$\lambda$), the CAD tool will require lesser effort to place the shorter paths such that they do not exceed the delay of the longest path. On the other hand for a circuit with high-$\lambda$, the CAD tool will have less freedom to move the paths around without violating the delay of the longest path. Hence, we expect designs with low-$\lambda$ design to be more predictable than designs with high-$\lambda$. Our strategy is to identify through empirical means, the threshold value that will categorize a design as low-$\lambda$ or high-$\lambda$ designs. Let's define this threshold value as $\Lambda$. The estimated delay of a design with low-$\lambda$ is computed as the maximum path length, while a design with high-$\lambda$ is computed by multiplying the mean path length with a constant factor. The constant factor was empirically found to be close to $1/\Lambda$. The following describes our proposed delay estimation model, where $D_{est}$ is the estimated delay.

$$D_{est} = \begin{cases} D_{max} & \text{if } \lambda \leq \Lambda \\ D_{mean} \times \frac{1}{\Lambda} & \text{otherwise} \end{cases} \tag{4.4}$$

**4.3.2. Determining Value of $\Lambda$.** We compared the estimated delay using the simple delay model with the actual place and route results for 45 random algorithms. Figure 4.5 shows the estimation error and the corresponding $\lambda$ of the designs. It can be observed that there exist a high correlation between the predictability and $\lambda$. In particular, it is shown that for low $\lambda$ , the simple delay model can be applied with about 90% confidence. Large errors or low predictability are found in region where $\lambda$ is high. We empirically define $\Lambda$ to be 0.78 from the dataset.

**5. Result Analysis.** Table 5.1 describes the properties of the C functions that have been used to evaluate the proposed framework for high level area-time estimation. These applications (apart from the random algorithm) are commonly used in embedded applications.

[1]A dummy algorithm which has high parallelism. It computes 1 output from 4 inputs through several operations
[2]Only inner-most loop is considered

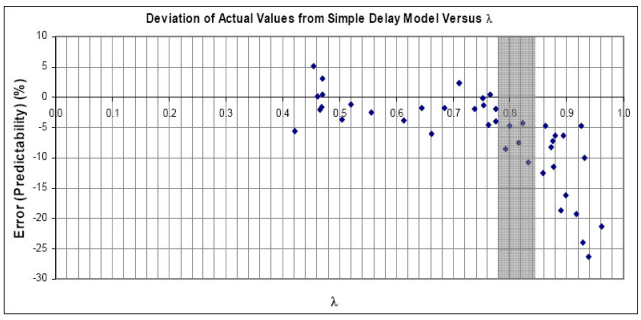



FIG. 4.5. *Estimation Error (Predictability) versus λ*

TABLE 5.1
*Connectivity Characteristics of Sample Circuits*

| Funtions | $D_{Mean}$ (ns) | λ | Max Fanout | Avg Fanout | Max Fanin | Ave Fanin |
|---|---|---|---|---|---|---|
| Random Algorithm[1] | 8.66 | 0.76 | 6 | 1.66 | 2 | 0.89 |
| Matrix multiplier[2] | 4.25 | 0.44 | 2 | 1.06 | 1 | 0.59 |
| mpeg2-_bdist1_motion | 4.82 | 0.46 | 13 | 2.13 | 4 | 1.22 |
| mpeg2-_bdist2_motion | 4.84 | 0.47 | 13 | 2.07 | 4 | 1.20 |
| mpeg2-_dct_type_estimation | 4.87 | 0.51 | 5 | 1.43 | 2 | 0.81 |
| mpeg2-_dist1_motion | 5.73 | 0.88 | 11 | 2.43 | 5 | 1.08 |
| mpeg2-_dist2_motion | 5.38 | 0.50 | 19 | 3.25 | 9 | 1.92 |
| mpeg2-_idctcol | 5.63 | 0.53 | 15 | 1.70 | 2 | 0.91 |
| mpeg2-_idctrow | 5.88 | 0.55 | 15 | 1.70 | 2 | 0.86 |
| sha_transform | 4.94 | 0.84 | 14 | 2.60 | 7 | 1.65 |
| adpcm_coder | 4.84 | 0.73 | 10 | 2.54 | 8 | 1.35 |
| adpcm_decoder | 4.64 | 0.67 | 9 | 2.16 | 9 | 1.22 |

**5.1. Area Estimation.** Table 5.2 compares the proposed area estimation (Pro) with results obtained from the Xilinx ISE tool after synthesis (Syn) and after place and route (PAR). The last four columns show the estimation errors of our method and that of logic synthesis, when compared with the post place and route values. It is evident that the proposed area estimation achieves up to an average of 98% accuracy, with a worst case error of 8% in terms of LUT comparison. For the estimation of slices, the average error of the proposed method is 12%. It is noteworthy that area estimation in terms of FPGA slices is a difficult task and the majority of the previous works reported their estimation results in terms of LUTs and flip-flops [27] [18] [19]. In general, our proposed area estimation for both LUTs and slices is comparable to the results of the logic synthesis tool. In addition, the proposed estimation can be completed in order of milliseconds, while the compilation time of the commercial synthesis tool takes several minutes (because of the level of abstraction).

**5.2. Delay Estimation.** The maximum delay of the above mentioned C-functions were estimated using the simple delay model (Sim) and the proposed model (Pro), and compared with actual post place and route values. In addition, we have used the ISE Xilinx tool to synthesize and implement the generated RTL codes in order to obtain the estimated delay after synthesis (Syn) and the actual delay after place and route (PAR). The results show that the proposed approach (Pro) outperforms the simple delay model and the synthesis tool for estimating the post place and route delay. In particular, the proposed delay estimation achieves better results in terms of both maximum and average accuracy. The maximum and average estimation error of the proposed approach is only 4.6% and 2.8% respectively. It can be observed from figure 5.1 that the simple delay model can provide accurate estimation results for designs with low λ cases. However, in designs with high-λ (i. e. 6 and 10), the simple delay model and the synthesis tool (Syn) incurs very high estimation error (i. e. up to 14%). In these cases, the proposed estimation approach is capable of providing significantly better accuracy due to the inclusion of the new metric λ that can be easily obtained for high-level estimation.

**5.3. Estimation Runtime.** The experiments were carried out on the Pentium 4 3GHz workstation, and the Xilinx ISE 8.1 tool was used to obtain the synthesis and post-place-and-route results. Table 5.4 compares the estimation time with the execution time of the Xilinx tool for synthesis and PAR. On an average, the proposed

TABLE 5.2
*Area Estimation Result*

| | LUT Estimation | | | Slices Estimation | | | LUT Error | | Slice Error | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Pro (LUTs) | Syn (LUTs) | PAR (LUTs) | Pro (Slices) | Syn (Slices) | PAR (Slices) | Pro % | Syn % | Pro % | Syn % |
| Random Algorithm | 1636 | 1586 | 1,574 | 1170 | 1268 | 1014 | 3.94 | 0.76 | 15.44 | 25.06 |
| Matrix multiplier | 285 | 284 | 278 | 270 | 312 | 264 | 2.52 | 2.16 | 2.47 | 18.22 |
| mpeg2-_bdist1_motion | 1898 | 1959 | 1,939 | 1421 | 1632 | 1523 | -2.11 | 1.03 | 6.67 | 7.16 |
| mpeg2-_bdist2_motion | 1971 | 1962 | 1,923 | 1465 | 1638 | 1843 | 2.50 | 2.03 | 20.49 | 11.13 |
| mpeg2-_dct_type_estimation | 623 | 624 | 616 | 471 | 542 | 482 | 1.14 | 1.30 | 2.28 | 12.45 |
| mpeg2-_dist1_motion | 2200 | 2183 | 2,165 | 1580 | 1705 | 1627 | 1.62 | 0.83 | 2.89 | 4.79 |
| mpeg2-_dist2_motion | 2625 | 2627 | 2,586 | 1688 | 1872 | 2022 | 1.51 | 1.59 | 16.50 | 7.42 |
| mpeg2-_idctcol | 4646 | 4531 | 4,501 | 3451 | 3851 | 3590 | 3.22 | 0.67 | 3.86 | 7.27 |
| mpeg2-_idctrow | 2942 | 2929 | 2,899 | 2271 | 2530 | 2229 | 1.48 | 1.03 | 1.91 | 13.51 |
| sha_transform | 2548 | 2691 | 2,670 | 1770 | 1993 | 2208 | -4.57 | 0.79 | 19.82 | 9.74 |
| adpcm_coder | 1380 | 1399 | 1,367 | 978 | 1093 | 1308 | 0.95 | 2.34 | 25.20 | 16.44 |
| adpcm_decoder | 1496 | 1409 | 1,391 | 998 | 1118 | 1406 | 7.55 | 1.29 | 28.99 | 20.49 |
| Average Error | | | | | | | 1.64 | 1.32 | 12.21 | 12.81 |

TABLE 5.3
*Delay Estimation Result*

| | Absolute Values | | | | Error Compared to PAR | | |
|---|---|---|---|---|---|---|---|
| | Sim (ns) | Syn (ns) | Pro. (ns) | PAR (ns) | Sim % | Syn % | Pro % |
| Random Algorithm | 11.47 | 11.22 | 11.47 | 11.13 | 3.07 | 0.75 | 3.07 |
| Matrix multiplier | 9.61 | 9.62 | 9.61 | 9.46 | 1.56 | 1.65 | 1.56 |
| mpeg2-_bdist1_motion | 10.37 | 10.41 | 10.37 | 9.95 | 4.21 | 4.61 | 4.21 |
| mpeg2-_bdist2_motion | 10.37 | 11.54 | 10.37 | 10.61 | 2.27 | 8.77 | 2.27 |
| mpeg2-_dct_type_estimation | 9.61 | 9.74 | 9.61 | 9.91 | 3.00 | 1.65 | 3.00 |
| mpeg2-_dist1_motion | 6.55 | 6.65 | 7.35 | 7.70 | 15.00 | 13.69 | 4.55 |
| mpeg2-_dist2_motion | 10.71 | 10.97 | 10.71 | 10.42 | 2.75 | 5.20 | 2.75 |
| mpeg2-_idctcol | 10.71 | 11.22 | 10.71 | 10.94 | 2.13 | 2.49 | 2.13 |
| mpeg2-_idctrow | 10.71 | 11.22 | 10.71 | 11.10 | 3.49 | 1.06 | 3.49 |
| sha_transform | 5.83 | 5.98 | 6.33 | 6.22 | 6.26 | 3.88 | 1.86 |
| adpcm_coder | 6.62 | 6.45 | 6.62 | 6.37 | 3.86 | 1.32 | 3.86 |
| adpcm_decoder | 6.93 | 6.81 | 6.93 | 6.91 | 0.39 | 1.42 | 0.39 |
| Average Error | | | | | 4.00 | 3.87 | 2.76 |

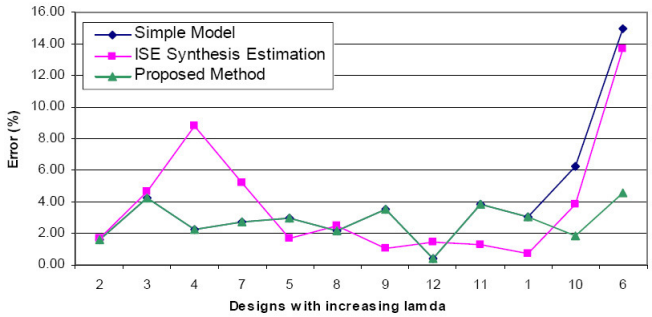



FIG. 5.1. *Estimation error with designs arranged in increasing of* $\lambda$

estimation process completes in the order of milliseconds except in cases 3, 4, 6, 7. In these cases, parsing of the Trimaran's textual output takes up to 2 seconds to complete, while the actual hardware binding and estimation process is performed in milliseconds. Overall, our technique achieves the estimation results about 350 times faster than the synthesis process, and about 3000 times faster than the PAR process.

**5.4. Evaluation of the Framework for different FPGA Families.** In order to evaluate the robustness of the proposed framework, we carried out experiments with the Spartan-3 FPGA. The following processes are repeated with the new target FPGA device: 1) characterization of components and interconnect, 2) identification of the value of $\lambda$ and 3) area-time estimation. Due to the less sophisticated FPGA routing fabric in Spartan-3, we have obtained $\lambda = 0.72$. Table 5.5 and 5.6 show the quality of estimation compared to actual place and route values. The average errors of delay were found to be 3.4% while synthesis tool's estimation error is 8.5%. The results of the proposed area estimation are reasonably good compared to the results obtain from the synthesis tool. In figure 5.2, the designs are rearranged in increasing order of $\lambda$. It can be observed that there is a large estimation error obtained using the simple model and ISE synthesis tool for designs with high value of $\lambda$. In contrast, the estimation error incurred with the proposed technique is consistent across the different functions. This implies that the proposed estimation technique leads to a higher degree of predictability when compared to the simple model and synthesis tool.

**6. Conclusions.** FPGA-based high-level area-time estimation that ignores the physical design effects after place and route may lead to very high inaccuracies. In this paper, we have presented a high-level estimation framework that can predict the area-time measures of C-based applications with post place and route effects taken into account. It is worth mentioning that the original C applications can be directly used in the proposed framework without any further modifications. Our area estimation has been shown to achieve comparable results with that obtained from a commercial synthesis tool. We have proposed a new metric for our delay estimation model that captures the placement complexity of the circuit. For the experiments considered, when

TABLE 5.4
*Estimation Runtime Compared to Synthesis And Actual PAR time*

| | Functions | Estimation Time (s) | ISE Syn Time (s) | PAR Time (s) |
|---|---|---|---|---|
| 1 | Random Algorier | 0.26 | 237 | 960 |
| 2 | Matrix Multiplier | 0.08 | 31 | 285 |
| 3 | mpeg2-_bdist1_motion | 2.54 | 255 | 676 |
| 4 | mpeg2-_bdist2_motion | 2.59 | 110 | 814 |
| 5 | mpeg2-_dct_type_estimation | 0.2 | 35 | 613 |
| 6 | mpeg2-_dist1_motion | 2.89 | 82 | 3612 |
| 7 | mpeg2-_dist2_motion | 2.71 | 112 | 687 |
| 8 | mpeg2-_idctcol | 0.23 | 141 | 2620 |
| 9 | mpeg2-_idctrow | 0.17 | 107 | 690 |
| 10 | sha_transform | 0.25 | 152 | 3600 |
| 11 | adpcm_coder | 0.2 | 81 | 780 |
| 12 | adpcm_decoder | 0.15 | 61 | 2700 |

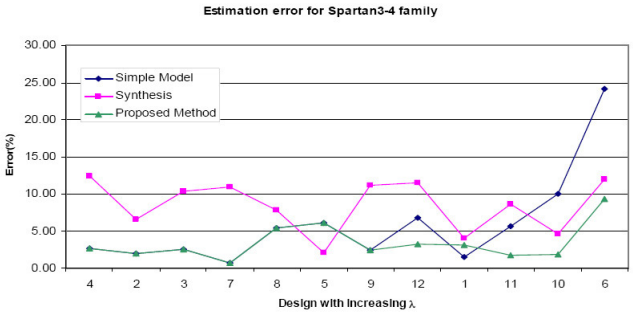



FIG. 5.2. *Estimation error with designs arranged in increasing of* $\lambda$

compared to post place and route results obtained from a commercial tool, our proposed delay estimation achieves an average accuracy of 97% with a worst case error of only 4.5%. This result is significantly better than previously reported works in high-level delay estimation and the estimation process can be completed in the order of milliseconds. In addition, we have shown that the proposed framework provide consistent results for devices from the Xilinx Virtex and Spartan families.

## REFERENCES

[1] XILINX CORPORATION, *VirtexII Pro Capabilities*. Available at `http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/virtex_ii_pro_fpgas/capabilities/index.htm`
[2] ALTERA CORPORATION, *Stratix II FPGA*. Available at `http://www.altera.com/products/devices/stratix2/st2-index.jsp`
[3] L. N. CHAKRAPANI, J. GYLLENHAAL, W. MEIW. HWU, S. A. MAHLKE, K. V. PALEM, AND R. M. RABBAH, *Trimaran: An infrastructure for research in instruction-level parallelism*. In: Lecture Notes in Computer Science (Langurages and Compilers for High Performance Computing), 2004, pp. 32–41.
[4] R. SCHREIBER, S. ADITYA, S. MAHLKE, V. KATHAIL, B. R. RAU, D. CRONQUIST, AND M. SIVARAMAN, *PICO-NPA: High-level synthesis of nonprogrammable hardware accelerators*. Journal of VLSI Signal Processing, 31(2002), pp. 127–142.
[5] MITRIONICS CORPORATION, *Mitron-C*. Available at `http://www.mitrionics.com/`
[6] CELOXICA CORPORATION, *Handel-C*. Available at `http://www.celoxica.com/`
[7] DSPLOGIC CORPORATION, *DSPLogic ToolBox*. Available at `http://www.dsplogic.com/home/`
[8] IMPULSEC CORPORATION, *Impulse C language*. Available at `http://www.impulsec.com/`
[9] ALTERA CORPORATION, *C to Hardware Technology*. Available at `http://www.altera.com/products/ip/processors/nios2/tools/c2h/ni2-c2h.html`
[10] JUSTIN L. TRIPP, MAYA B. GOKHALE, KRISTOPHER D. PETERSON, *Trident: From High-Level Language to Hardware Circuitry*. Computer, 40(2007), pp. 28–37.
[11] SHAWN MCCLOUD, *Catapult C Synthesis-based Design Flow: Speeding Implementation and increasing Flexibility*. Mentor Graphic White Paper 2003. Available at `http://www.mentor.com`
[12] BYONGRO SO ET AL, *Using Estimates from Behavioral Synthesis Tools in Compiler Directed Design Space Exploration*. Proceedings of Design Automation Conference 2003, 2 (2003), pp. 514–519.
[13] MIN JOONG RIM, RAJIV JAIN, *Lower-bound performance estimation for high-level synthesis scheduling problem*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 13 (4-1994), pp. 451–458.

TABLE 5.5
*Area Estimation for Spartan Family*

| | Pro LUT | Syn LUT | PAR LUT | Pro SLICE | Syn SLICE | PAR SLICE | Pro LUT (%) | Syn LUT (%) | Pro SLICE (%) | Syn SLICE (%) |
|---|---|---|---|---|---|---|---|---|---|---|
| Random Algorithm | 1636 | 1580 | 1,568 | 1122 | 1170 | 1093 | 4.34 | 0.77 | 2.65 | 7.04 |
| Matrix multiplier | 285 | 284 | 278 | 239 | 270 | 222 | 2.52 | 2.16 | 7.67 | 21.67 |
| mpeg2_bdist1_motion | 1978 | 1903 | 1,854 | 1349 | 1384 | 1564 | 6.69 | 2.64 | 13.72 | 11.51 |
| mpeg2_bdist2_motion | 1962 | 1959 | 1,918 | 1349 | 1435 | 1649 | 2.29 | 2.14 | 18.19 | 12.98 |
| mpeg2_dct_type_estimation | 623 | 624 | 616 | 440 | 508 | 792 | 1.14 | 1.30 | 44.51 | 35.86 |
| mpeg2_dist1_motion | 2200 | 2168 | 2,153 | 1364 | 1392 | 1839 | 2.18 | 0.70 | 25.83 | 24.31 |
| mpeg2_dist2_motion | 2625 | 2618 | 2,576 | 1585 | 1669 | 1567 | 1.90 | 1.63 | 1.15 | 6.51 |
| mpeg2_idctcol | 4646 | 4511 | 4,481 | 3227 | 3519 | 3256 | 3.68 | 0.67 | 0.89 | 8.08 |
| mpeg2_idctrow | 2942 | 2925 | 2,895 | 2095 | 2317 | 2140 | 1.62 | 1.04 | 2.10 | 8.27 |
| sha_transform | 2548 | 2545 | 2,530 | 1578 | 1693 | 2267 | 0.71 | 0.59 | 30.38 | 25.33 |
| adpcm_coder | 1688 | 1498 | 1,470 | 988 | 893 | 846 | 14.83 | 1.90 | 16.85 | 5.56 |
| adpcm_decoder | 1244 | 1165 | 1,148 | 878 | 859 | 807 | 8.36 | 1.48 | 8.80 | 6.44 |
| Average Error | | | | | | | 4.19 | 1.42 | 14.39 | 14.86 |

TABLE 5.6
*Delay estimation for Spartan3-4 Family*

| | Sim (ns) | Syn (ns) | Pro. (ns) | PAR (ns) | Sim (%) | Syn (%) | Pro (%) |
|---|---|---|---|---|---|---|---|
| Random Algorithm | 16.26 | 17.17 | 17.02 | 16.51 | 1.51 | 4.02 | 3.10 |
| Matrix multiplier | 13.54 | 14.14 | 13.54 | 13.28 | 1.97 | 6.53 | 1.97 |
| mpeg2-_bdist1_motion | 14.72 | 15.85 | 14.72 | 14.36 | 2.49 | 10.33 | 2.49 |
| mpeg2-_bdist2_motion | 15.90 | 17.42 | 15.90 | 15.49 | 2.62 | 12.45 | 2.62 |
| mpeg2-_dct_type_estimation | 13.54 | 14.71 | 13.54 | 14.41 | 6.06 | 2.04 | 6.06 |
| mpeg2-_dist1_motion | 9.16 | 10.62 | 10.94 | 12.06 | 24.08 | 12.00 | 9.33 |
| mpeg2-_dist2_motion | 15.08 | 16.62 | 15.08 | 14.98 | 0.65 | 10.93 | 0.65 |
| mpeg2-_idctcol | 15.08 | 17.17 | 15.08 | 15.94 | 5.37 | 7.78 | 5.37 |
| mpeg2-_idctrow | 15.08 | 17.17 | 15.08 | 15.45 | 2.38 | 11.18 | 2.38 |
| sha_transform | 8.13 | 9.45 | 9.21 | 9.04 | 9.98 | 4.55 | 1.89 |
| adpcm_coder | 8.76 | 10.08 | 9.45 | 9.29 | 5.65 | 8.57 | 1.74 |
| adpcm_decoder | 8.13 | 9.73 | 8.45 | 8.727 | 6.81 | 11.52 | 3.19 |
| Average Error | | | | | 5.80 | 8.49 | 3.40 |


[14] CHAUDURI S. AND WALKER, *Computing the lower bound of functional unit before scheduling.* IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol 4 Issue 2(1996), pp. 273–279.

[15] A. NAYAK, M. HALDAR, A. CHOUDHARY, AND P. BANERJEE, *Accurate Area and Delay Estimators for FPGA.* Proceedings of International Conference DATE, 2002, pp. 862–869.

[16] MINJOONG RIM AND RAJIV JAIN, *Estimating performance characteristic of Loop transformation.* IEEE International Symposium on Circuits and Systems, 1 (1994), pp. 249–252.

[17] SEBASTIEN BILAVARN ET AL, *Design Space Pruning through estimations of Area/Delay Trade-off for FPGA.* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 25(2006), pp. 1950–1968.

[18] JOÃO M. P. CARDOSO, *On Estimations for Compilation of Software for FPGA,* Proceedings of the16th International Conference on Application-Specific Systems, Architecture and Processors (ASAP'05), 2005, pp. 225–230.

[19] DHANANJAY KULKARNI ET AL, *Compile Time Area estimation for LUT-based FPGAs.* ACM Transactions on Design Automation of Electronic Systems (TODAES), 11 (2006), pp. 104–122.

[20] XILINX CORPORATION, *Xplorer Technology.* Available at `http://www.xilinx.com/products/design_tools/logic_design/implementation/xplorer.htm`

[21] MIN XU. FADI J. KURDAHI, *Area and timing estimation for lookup table based FPGA.* Proceedings of the 1996 European conference on Design and Test, 1996, pp. 151.

[22] SHANKAR BALACHANDRAN, *A-priori wirelength and interconnect estimation based on circuit characteristics,* Proceedings of the 2003 international workshop on System-level interconnect prediction SLIP, 2003, pp. 77–84.

[23] TANAY KARNIK AND SUNG-MO KANG, *An Empirical Model For Accurate Estimation of Routing Delay in FPGAs.* Proceedings of the 1995 IEEE/ACM international conference on Computer-aided design, 1995, pp. 328–331.

[24] MICHEAL HUTTON, *Interconnect Prediction for Programmable Logic Devices,* Proceedings of the 2003 international workshop on System-level interconnect prediction, 2003, pp. 31–38.

[25] VALAVAN MANOHARARAJAH ET AL, *Difficulty of Predicting Interconnect Delay in a timing driven FPGA CAD Flow,* In: Proceedings of International Proceedings on SLIP, 2006, pp. 3–8.

[26] SEONG Y. OHM ET AL, *A comprehensive estimation technique for High-level Synthesis,* Proceedings of the 8th international symposium on System synthesis, 1995, pp. 122–127.

[27] CARLO BRANDOLESE ET AL, *An area estimation methodology for FPGA based Designs at system C- level.* Proceedings of the 41st annual conference on Design automation, 2004, pp. 129–132.


*Edited by:* Javier Díaz
*Received:* October 8th, 2007
*Accepted:* December 10th, 2007

Scalable Computing: Practice and Experience
Volume 8, Number 4, pp. 373–385. http://www.scpe.org
ISSN 1895-1767


# PERFORMANCE OF A LU DECOMPOSITION ON A MULTI-FPGA SYSTEM COMPARED TO A LOW POWER COMMODITY MICROPROCESSOR SYSTEM*

T. HAUSER†, A. DASU‡, A. SUDARSANAM‡ , AND S. YOUNG‡

**Abstract.** Lower/Upper triangular (LU) factorization plays an important role in scientific and high performance computing. This paper presents an implementation of the LU decomposition algorithm for double precision complex numbers on a star topology based multi-FPGA platform. The out of core implementation moves data through multiple levels of a hierarchical memory system (hard disk, DDR SDRAMs and FPGA block RAMS) using completely pipelined data paths in all steps of the algorithm. Detailed performance numbers for all phases of the algorithm are presented and compared to a highly optimized implementation for a low power microprocessor based system. We also compare the performance/Watt for the FPGA and the microprocessor system. Finally, recommendations will be given on how improvements of the FPGA design would increase the performance of the double precision complex LU factorization on the FPGA based system.

**Key words.** LU factorization, multi-FPGA system, benchmarking

**1. Introduction.** High-performance reconfigurable computers (HPRC) [20, 5] based on conventional processors and field-programmable gate arrays (FPGAs) [31] promise better performance, especially when taking the power consumption into account. Recently, HPRCs have shown orders of magnitude improvements in performance, e.g. power and speed, over conventional high-performance computers (HPCs) in some compute intensive integer applications but showing similar success on floating point based problems has been limited [32].

Scientific computing applications demand double-precision arithmetic because of numerical stability and large dynamic range requirements. Solving linear systems and linear algebra plays an important role in scientific and high performance computing. The LAPACK library [1, 2, 10, 12, 13, 36, 3, 7] is a high quality library of linear equation solvers and considerable work has been done to achieve very good performance on different high performance computing platforms. The introduction of hierarchical memory systems, which feature multiple levels of cache storage with different sizes and access speeds, has tended to degrade the performance of these linear algebra routines compared to the peak performance. Obtaining good performance with such systems required the formulation of those algorithms in terms of operations on blocks, so that cache misses could be minimized [16].

The goal of this paper is to benchmark and compare the performance of a block based algorithm on a HPRC platform to a highly optimized implementation on a commodity microprocessor. and provide suggestions for the improvement of the FPGA platform to better support floating point linear algebra algorithms. Our work is based on the algorithms described in [10] which is adapted to a specific class of HPRCs. Hardware-based matrix operator implementation has been addressed by several researchers. Ahmed El-Amawy [15] proposes a systolic array architecture consisting of $(2N^2 - N)$ processing elements which computes the inverse in $O(N)$ time, where $N$ is the order of the matrix. However, there are no results to show that the large increase in area (for large values of $N$) is compensated by the speed of this implementation.

Power efficiency is a critical issue in current high performance computing facilities and a critical issue for developing cost effective small-footprint clusters [22] as it directly influences the cooling requirements of each cluster node and of the overall cluster rack and server room layout. A 48 core, low power cluster [29], designed to run on 20 Amp electric circuit, is an example of per-node low power requirements. Each of our quad-core processor nodes consumes approximately 78 Watts during normal operation under our group's cluster workloads. We describe the design and implementation of the LU factorization algorithm for a double precision complex matrix on a HPRC system and compare it to a highly tuned implementation for a commodity microprocessor. The matrices considered for the factorization are so large that the factorization has to be performed out of core on the FPGA system. In addition to comparing the wall clock times, power efficiency of the FPGA versus the microprocessor using the millions of floating point operations per Watt (MFlops/Watt) metric is provided.

The paper is structured as follows. Section 2 describes the general multi-FPGA system architecture the algorithm is designed for and the details of the Starbridge HC-62 system. Section 3 presents an overview of the

*This work was support in part by Lockheed Martin Corp.
†Center for High Performance Computing, Utah State University, 4175 Old Main Hill, Logan, UT, 84322-4175 (thomas.hauser@usu.edu)
‡Department of Electrical and Computer Engineering, Logan, UT, 4120 Old Main Hill, Logan, UT, 84322-4120

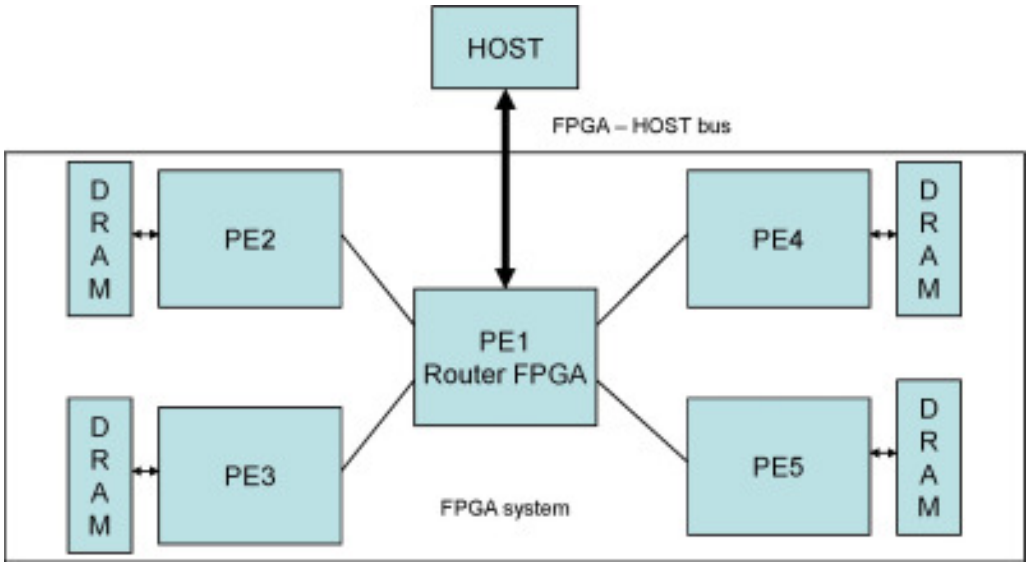



FIG. 2.1. *Star topology based multi-FPGA system with four FPGAs*

algorithm used and how it is mapped on the HPRC architecture. In Sections 4 and 5 we discuss the benchmark results and compare it to the commodity microprocessor implementation. Section 6 presents an overview over related work for floating point computation on HPRC architectures and Section 7 provides the lessons learned from this implementation.

## 2. Multi-FPGA system architecture.

**2.1. Star networked FPGAs with external storage.** The hardware system topology we have considered for analysis and mapping of the LU factorization algorithm is the Star Network Topology. We assume that in this topology, multiple FPGA devices can communicate to a central host microprocessor through a concentrator/router FPGA. In addition, each FPGA is assumed to have its own local external storage such as DRAM chips. We refer to this structure henceforth as "Star Networked FPGAs with Local Storage" or SNFLS topology. Figure 2.1 shows a SNFLS systems with four compute FPGAs (PE2-PE5) with locally attached DRAM, one router FPGA and the host system. Several variations in the topology, such as the manner how physical memory devices are distributed across FPGAs, how the accelerator FPGA board is connected to a host system, etc. can have impacts on the performance of a system.

**2.2. Starbridge HC-62 system.** The system used for the implementation and benchmarks contains a HC-64 board from Starbridge systems, consisting of eight programmable FPGAs, attached to a host PC with an Intel x86 processor. There are two FPGA chips that are used for interface functions. The first is an Xpoint switch chip, and the second is a primary function chip, PE1. The Xpoint FPGA provides a link between the other FPGAs and the PCI bus. It provides a 256-bit fully populated synchronous cross-point router with each of the four FPGA elements PE2 through PE5 at 82.4 Gigabits/second. The four FPGAs are connected through a 128-bit data bus to the Xpoint and PE1 chips, and each contain two 128-bit serial multiplier objects. A group of four FPGA connected together with 50-bit parallel lines is called a quad group. Each quad group is internally connected with a 50-bit exclusive chip to chip bus at 96.6 gigabits per second. The memory chips attached to each PE are arranged in four banks around the chip. The main memory is made up of 8-72 Gigabytes SDRAM modules, with a bandwidth of 95 Gigabytes with 36 independent 64 bit ports. Each PE is a Xilinx Virtex-II chip. It consists of 33,792 slices, with each slice consisting of 2 Flip-Flops (FFs) and 2 Look-Up Tables (LUTs). The chip also contains 144 18x18 ASIC multipliers and 144 Block RAMs (BRAMS) and each RAM contains 18K bits of memory.

## 3. Matrix Factorization implementation for a multi-FPGA system.

**3.1. Block-partitioned LU factorization algorithm.** The LU factorization applies a sequence of Gaussian eliminations to form $A = LU$, where $A$, $L$ and $U$ are $N \times N$ matrices. Note, that in our algorithm a

permutation matrix is not necessary, since all matrices from the problem addressed are diagonally dominant and the condition number of our matrices is of $\mathcal{O}(1)$. $L$ is a unit lower triangular matrix with 1's on the main diagonal, $U$ is an upper triangular matrix. Our algorithm is applied recursively by partitioning the matrix $A^{(k)}$, which is a $n \times n$ submatrix of size $n = N-(k-1)\cdot n_b$ at the $k$-th step into four blocks, where $n_b$ is our blocking size.

$$A^{(k)} = L^{(k)}U^{(k)} = \begin{pmatrix} A_{11}^{(k)} & A_{12}^{(k)} \\ A_{21}^{(k)} & A_{22}^{(k)} \end{pmatrix} \tag{3.1}$$

$$= \begin{pmatrix} L_{11}^{(k)} & 0 \\ L_{21}^{(k)} & L_{22}^{(k)} \end{pmatrix} \begin{pmatrix} U_{11}^{(k)} & U_{12}^{(k)} \\ 0 & U_{22}^{(k)} \end{pmatrix} \tag{3.2}$$

$$= \begin{pmatrix} L_{11}^{(k)}U_{11}^{(k)} & L_{11}^{(k)}U_{12}^{(k)} \\ L_{21}^{(k)}U_{11}^{(k)} & L_{21}^{(k)}U_{12}^{(k)} + L_{22}^{(k)}U_{22}^{(k)} \end{pmatrix}$$

where the dimensions of the block $A_{11}^{(k)}$ are $n_b \times n_b$, of $A_{21}^{(k)}$ are $n_b \times (n-n_b)$, of $A_{21}^{(k)}$ are $(n-n_b) \times n_b$, and of $A_{22}^{(k)}$ are $(n-n_b) \times (n-n_b)$. The block partitioned algorithm at the $k$-th recursion step is performed in four steps:

1. $A_{11}^{(k)}$ is factored into $U_{11}^{(k)}$ and $L_{11}^{(k)}$.
2. $A_{21}^{(k)}$ is factored into $L_{21}^{(k)}$.
3. $U_{12}^{(k)}$ is computed by solving the triangular system

$$L_{11}^{(k)}U_{12}^{(k)} = A_{12}^{(k)} \tag{3.3}$$

in our implementation $U_{12}^{(k)}$ is computed using the inverse of $L_{11}^{(k)}$

$$U_{12}^{(k)} \leftarrow \left(L_{11}^{(k)}\right)^{-1} A_{12}^{(k)} \tag{3.4}$$

The use of the inverse of $L_{11}^{(k)}$ is more efficient according to Ditkowski [11], since the condition number of our matrices is always of $\mathcal{O}(1)$.

4. Update the remaining block $\widetilde{A}_{22}^{(k)}$, which then becomes the matrix $A^k$ for the next iteration $k+1$.

$$\widetilde{A}_{22}^{(k)} \leftarrow A_{22}^{(k)} - L_{21}^{(k)}U_{12}^{(k)} = L_{22}^{(k)}U_{22}^{(k)} \tag{3.5}$$

The LU factorization is completed when the matrix $A^{(k)}$ becomes so small that only step 1 is left to compute.

**3.2. Mapping of the LU algorithm onto a multi-FPGA system.** The SNFLS architecture as shown in Figure 2.1 is particularly suited for the block-based dense matrix LU factorization, as this algorithm has no need for inter-block communication in any of the sub-steps. Input data can be loaded into each of the DRAMs and processed and written back to the host. By analyzing the algorithm, it can be observed that there is data re-use at both the intra-block as well at the inter-block level. Therefore, a set of blocks can be initially loaded onto the DRAMs from PC memory and one block can be loaded onto an FPGA's BRAM for execution. We assume that most of the data resides on the hard disk of the host microprocessor, and these storage devices are slow, since they need to communicate with FPGA DRAMs through the PCI-X bus. The data is first transferred from hard disk to host processor memory. Then it is routed through the host-FPGA bus and saved on each FPGA's DRAM.

The top level control flow can be handled by the host processor which will split the matrix into multiple parts for each step. On-chip memory of the host processor can be used as a buffer to handle the difference in the speed of data transfer between Host's-Hard Disk and an FPGA. Data transfer between off-chip DRAM and the FPGA is modeled as a sequence of I/O operations. During a single I/O operation, a block of data can be transferred from off-chip DRAM to the FPGA. This block is of size $n_b \times n_b$, where $n_b$ is the block size in the LU decomposition algorithm. The value of $n_b$ should be chosen so that the amount of data transferred is neither

too small, nor too large. A too small value of $n_b$ is wasting opportunities for parallelism whereas a too large value will exceed the capacity of the DRAM.

When data is transferred from host memory to DRAM, one of the factors that can limit the amount of data transferred per I/O operation is the DRAM size. Information about the latency of transferring data from host memory to DRAM should be used while scheduling the various operations.

The proposed design outlines the order in which the various blocks of data are transferred from/to the host memory to/from the FPGA board. The number of blocks transferred will depend on the DRAM size associated with each FPGA. The following sequence of control steps determines the order of their access.

For the case of a single-FPGA accelerated system, a basic block of size $n_b \times n_b$ forms the input and the module corresponding to the algorithm in which a single block is processed needs to be realized in FPGA hardware. In the proposed approach, data path units found inside the inner-most loop are identified and these units are replicated as many times as possible on a given FPGA device. The number of parallel instances is limited by resource availability of a single FPGA, as the proposed design limits the processing of a single block of data to be performed within a single FPGA. This constraint is added so that the data transfer logic can be localized to a single FPGA permitting easy scalability if more FPGAs are added to the system, but topology is retained.

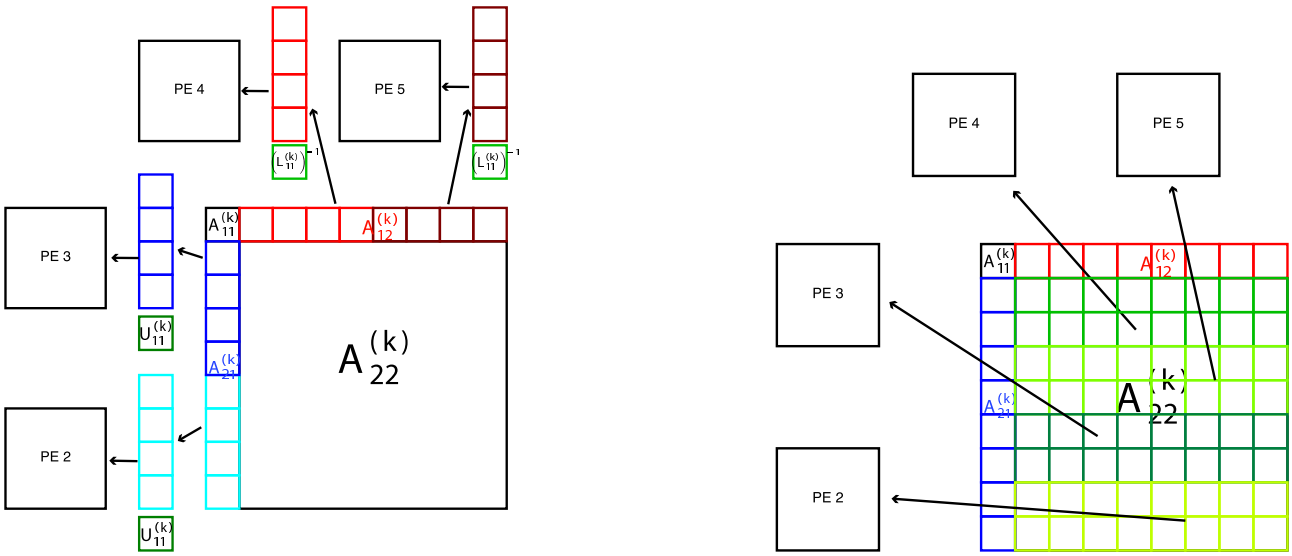
From preliminary investigation, it was concluded that a single moderate size FPGA (such as a Xilinx Virtex 2-6000) can provide some amount of data path parallelism if the data types are single precision and real. But even these expensive devices do not have sufficient resources to support data path parallelism within a chip if the data types are complex and double precision floating point. In such cases since intra-block parallelism is limited, there is a need to extract inter-block parallelism by using multiple FPGAs to execute several blocks in parallel.

For effective use of all three levels of memory in the system, initial data is assumed to be stored in the hard disk that is connected to the host processor. Each FPGA contains block RAMs (BRAMS) that form the first level of the memory hierarchy and provide seamless bandwidth to computation modules on the device. Resource utilization of BRAMs depends on multiple factors that include block size, amount of parallelism required, data type etc and is limited by the amount of on-chip memory available. Off-chip DRAMs form the second level of memory hierarchy and bandwidth between compute engine and DRAM is limited by the interconnection between FPGA and DRAM. Hard disks (connected via the host computer) form the third level in memory hierarchy.

Since the matrices generally are so big that computing the LU decomposition on a single PE is prohibitively expensive, the different steps are distributed onto several FPGAs so don't processing can take part in parallel. Steps 1 is processed on the host PC since it is only computed once and then the result is distributed to the different FPGAs. Figure 3.1a shows how steps 2 and 3 can be mapped to multiple FPGAs. Half of the FPGAs are assigned to process the submatrix $A_{21}^{(k)}$. This matrix is than partitioned in as many parts as FPGAs are available for processing. The same approach is taken to process $A_{12}^{(k)}$. Figure 3.1b shows how the matrix $A_{22}^{(k)}$ is distributed to multiple PEs. Together with the part of $A_{22}^{(k)}$ that is assigned to a PE, the corresponding parts of $A_{21}^{(k)}$ and $A_{12}^{(k)}$ necessary to compute the results $\widetilde{A}_{22}^{(k)}$ are also copied to the PE.

**4. Benchmarking results of a double-precision complex LU decomposition.** We implemented the algorithm described in Section 3 using Viva 3.0. Viva, developed by Starbridge systems, is a graphic-based hardware design tool to generate and synthesize the hardware designs for the LU decomposition algorithm. Xilinx mapping and place-and-route tools were used to generate the bit streams for the designs that then were loaded onto the target hardware platform. Our implementation was benchmarked on a Starbridge Hypercomputer board HC-36 (see Section 2.2). Although the IP cores for individual arithmetic units can be clocked at 100MHz or higher, the vendor caps the board to run only at 66MHz. Matrix sizes varied from 1000x1000, 2000x2000, 4000x4000 and 8000x8000. Each element in once of the matrices is a 64-bit double precision (52- mantissa; 11-exponent; 1-sign) complex number. In the discussion of the performance result the following four phases in the computation are differentiated:

1. Reconfiguration: The FPGA needs to be reconfigured for each of the four steps of the LU factorization.
2. FPGA processing: This time includes the transfer from DRAM to BRAM and all computational processing on the FPGA and the transfer back to DRAM.
3. PC to DRAM: This time describes the time takes to transfer data from the host PC to the DRAM of the FPGA accelerator.
4. DRAM to PC: This is the time it takes to transfer the data back to the PC.




(a) Step 2 and Step 3 mapping to multiple FPGAs

(b) Step 4 mapping to multiple FPGAs

FIG. 3.1. *Mapping of different steps of the algorithm to multiple FPGAs*

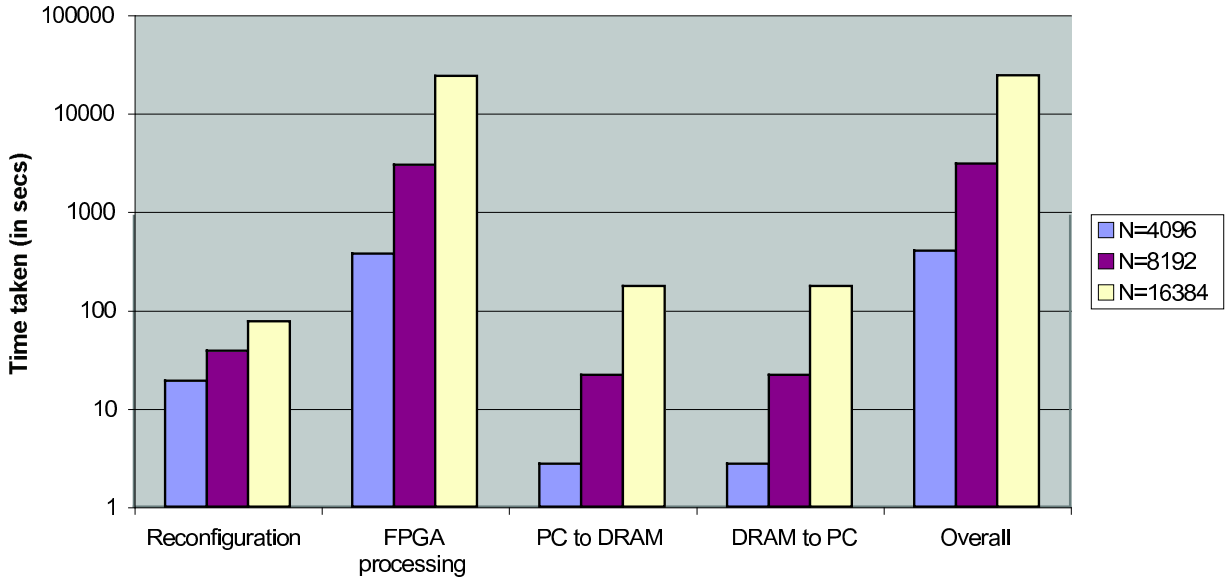



FIG. 4.1. *Wall clock times for the different phases of the LU implementation for increasing problem sizes N on one FPGA shown on a log scale* ($n_b = 1$)

**4.1. Performance depending on problem size.** In Figure 4.1, run-times for several problem sizes are compared for each of the different phases of the algorithm. The run times are displayed on a logarithmic scale because otherwise it would be difficult to recognize the small contributions of the reconfiguration, PC to DRAM transfer and DRAM to PC phases to the overall time of the algorithm.

In Figure 4.1 the overall dominating time is the processing time in the FPGA. The reconfiguration time doubles from 19.05 seconds to 38.25 and to 76.65 when doubling the problem size, because the number of reconfigurations scales linearly with the problem size. The processing time scales with $\mathcal{O}(N^3)$ as seen from the timing results. Also, the processing time scales much faster with the problem size then the memory transfer times and the reconfiguration time. Figure 4.1 clearly demonstrates that the dominating time on the FPGA system is the "FPGA processing" time.

**4.2. Performance Depending on Block Size.** The block size $n_b$ is an important parameter of the block partitioned LU factorization algorithm. This parameter determines how big the pieces are the FPGA can process on. In Figure 4.2, the influence of different block sizes $n_b$ on the performance is shown.

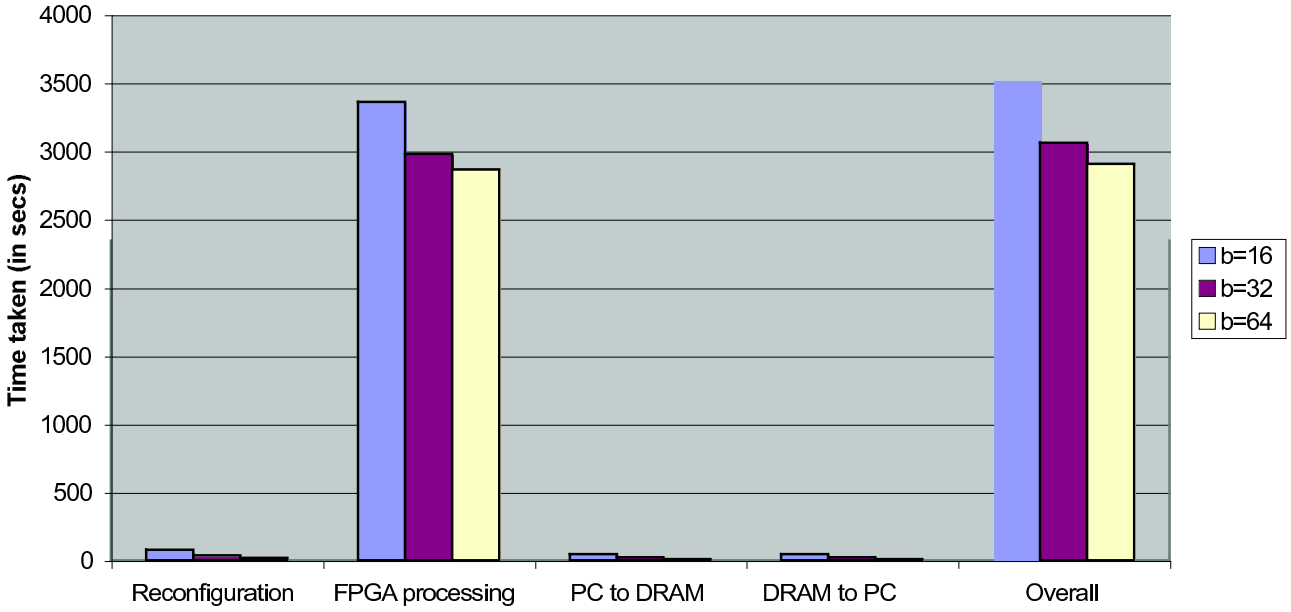



FIG. 4.2. *Wall clock times of different phases of the LU implementation for increasing block size $n_b$ on one FPGA ($N = 8192$)*

The block size $n_b$ cannot be increased above 64 because of the limitation on the number of block RAMs on the used FPGA. Increasing the block size increases the performance of the algorithm not only in the compute engine but it also reduces the number of blocks transferred and the number of reconfigurations. These are overhead operations which do not contribute to the overall progress of the computation but are necessary for the implementation of the algorithm.

**4.3. Performance depending on number of FPGAs.** In Figure 4.3 the wall clock times for the different phases of the LU algorithm are presented, when the number of FPGAs is increased.

Similar to figure 4.1, the dominating time is the processing time in the FPGA. The memory transfer time stays constant because of the the star topology of the hardware. All data has to go through the PCI bus to the DRAM of the individual FPGA. Therefor no speedup can be achieved for the memory transfer. The reconfiguration time actually increases when the number of FPGAs is increased.

The speedup $S$, defined as

$$S = \frac{T_s}{T_p^n}, \tag{4.1}$$

is given in table 4.1. $T_s$ is the time on a system with one FPGA and $T_p^n$ is the time of the parallel algorithm on $n$ FPGAs. The result show that the overall speedup does not scale linearly with increasing the number of FPGAs. While performance on each FPGA is highly deterministic, one can and correctly expect a linear scaling in speedup if only FPGA processing times are inspected. But because of the overhead of the data transfer and reconfiguration the speedup is reduced significantly.

TABLE 4.1
*Speedup computed from increasing the number of FPGAs for block size 64, matrix size 8192*

| | 1 FPGA | 2 FPGAs | 4 FPGAs |
|---|---|---|---|
| Overall time | 3870.24 | 2484.06 | 1862.41 |
| **Speedup overall time** | - | 1.55 | 2 |

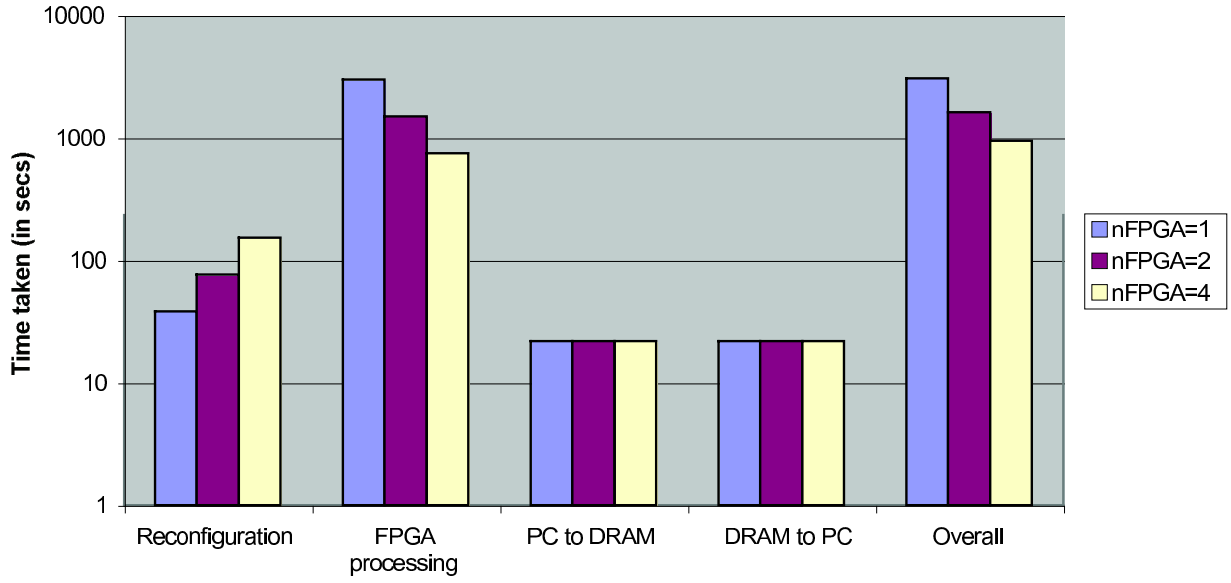



FIG. 4.3. *Wall clock times of the LU decomposition using 1, 2 and 4 FPGAs for processing in log scale*

TABLE 5.1
*Comparison of wall clock time between Intel®Xeon®X3210 microprocessor and FPGA implementation with block size = 64*

| Size | CPU 1 thread | 1 FPGA | CPU 2 threads | 2 FPGAs | CPU 4 threads | 4 FPGAs |
|---|---|---|---|---|---|---|
| 1024 | 0.499 | 13.56 | 0.249 | 16.24 | 0.318 | 26.03 |
| 2048 | 3.767 | 72.95 | 1.896 | 61.67 | 1.066 | 73.47 |
| 4096 | 29.49 | 505.7 | 14.862 | 348.74 | 8.021 | 305.69 |
| 8192 | 232.8 | 3870.24 | 117.2 | 2484.06 | 61.73 | 1862.41 |

## 5. Performance comparison to a commodity microprocessor.

**5.1. Microprocessor based system architecture and LU implementation.** For comparison the performance and power consumption was measured on a low power commodity CPU cluster which was specifically designed to run on a single 20A 110V circuit [29]. Since the problem sizes considered are relatively small a single compute node of this cluster was used to get the performance numbers for the microprocessor based system. The compute node consists of the following components:

- Quad-Core Intel®Xeon®X3210 processors
- 8 Gigabytes RAM
- 4 Gigabit ports
- No Hard Drive

The implementation of the LU decomposition on the cluster node uses the Intel®MKL library version 9.1 [23], specifically the LAPACK routine "zgetrf". This library is highly optimized for Intel®processors and provides highly scalable implementations for multiple threads.

**5.2. Wall clock time comparison between FPGA and microprocessor based system.** The benchmark results are summarized in Table 5.1.

These results show that the microprocessor has a performance advantage of about 30 times. The performance of the FPGA implementation could be improved by a factor of two by switching from the floating point objects provided by VIVA to Xilinx Coregen IP cores (see also section 5.3).

**5.3. System performance modeling.** To estimate impacts of changes of system parameters, e.g. interconnect bandwidth, or changes of the FPGA hardware, we have also developed a performance model. This section discusses in short some of the results obtained through the performance model, when using a more advanced FPGA platform. Our performance model contains a set of system parameters which can be obtained

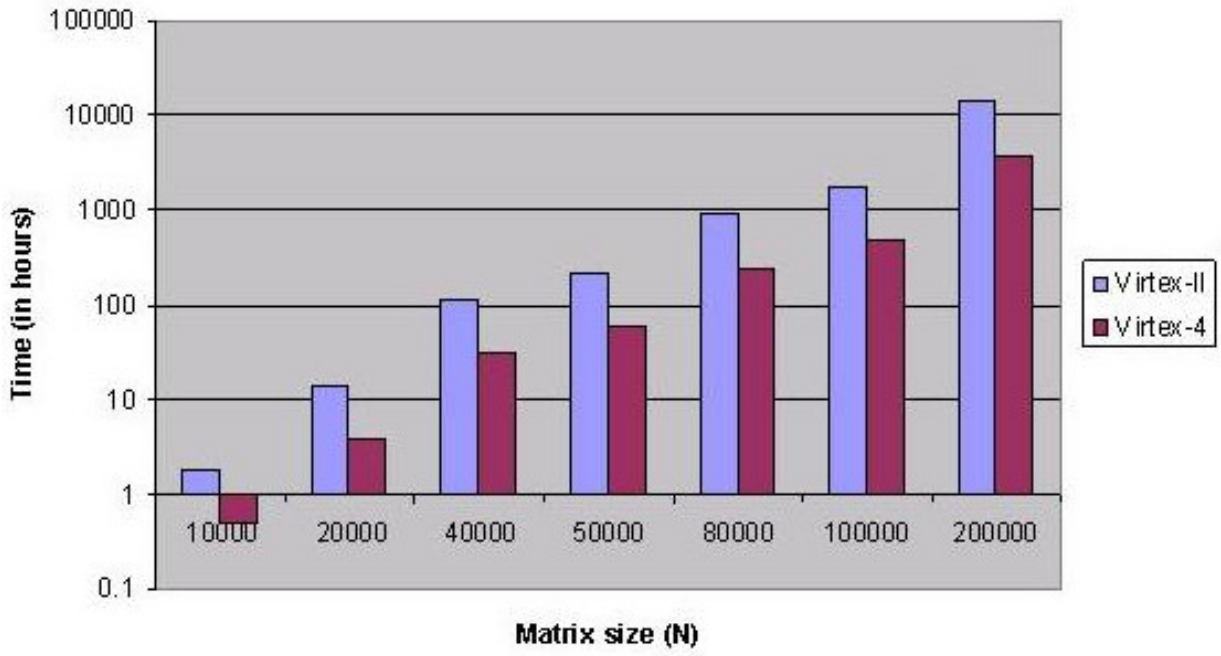



FIG. 5.1. *LU timing results for Virtex-II 6000 and Virtex-4 LX160 from our performance model*

for any FPGA from their respective data-sheets. Hence, our model can be extended to support any multi-FPGA platform. Here we compare the performance model result for a Xilinx Virtex-4 LX160 FPGA with the overall timing results from our benchmark platform using Xilinx Virtex-II 6000 FPGAs. For each target FPGA the number of data paths, $P$, is set such that the resource utilization of the FPGA is maximum. We found $P = 5$ for the Virtex-4 and $P = 1$ for the Virtex-1 device. In addition we included the effect of switching from the floating point objects provided by the Viva Corelib to the Xilinx IP Coregen library in our performance model results.

Figure 5.1 shows the comparative results for the two target platforms for different values of the problem size, $N$ and a blocking size of $b = 16$. It is seen that the Virtex-4 implementation is almost four times faster than the Virtex-II implementation. There is a two-fold increase. One is due to the larger slice count available and the other is use of a better design library.

### 5.4. Power consumption.

**5.4.1. Power consumption of FPGA board.** Table 5.2 shows the power consumption of the FPGA board alone. It shows that there is an increase of power consumption with the number of FPGAs used in the computation and the block size.

**5.4.2. Power consumption of FPGA system.** The overall system, consisting of the host PC and the FPGA board, has a power consumption profile as shown in Table 5.3. An interesting observation is the power consumption for larger problem sizes which seems to go down with the block size $n_b$. Especially for the largest test case of $4096 \times 4096$ elements the increase in block size shows a reduction in power by ten Watts.

**5.4.3. Power consumption microprocessor based system.** For comparison the power consumption was measured on the low power commodity cluster using one, two and four threads as also shown in Table 5.4. Since there are two compute nodes in one 1U chassis, it was not possible to measure the power consumption of just one compute node. So the same benchmark was run on each of the two compute nodes within one chassis, and the power consumption measured was divided by two to obtain the result for one compute node. Therefor, one chassis was plugged into a separate circuit, and the power was measured using a power analyzer called Watts UP. The results showed that a single node has the following power consumption irrespective of problem size:

TABLE 5.2
*FPGA board power consumption*

| Problem size $N$ | block size $n_b$ | number of FPGAs | Power |
|---|---|---|---|
| 1024 | 32 | 1 | 1.23 |
| 1024 | 32 | 2 | 2.70 |
| 1024 | 32 | 4 | 6.19 |
| 1024 | 64 | 1 | 1.28 |
| 1024 | 64 | 2 | 2.85 |
| 1024 | 64 | 4 | 6.24 |
| 2048 | 32 | 1 | 1.82 |
| 2048 | 32 | 2 | 3.04 |
| 2048 | 32 | 4 | 6.68 |
| 2048 | 64 | 1 | 1.67 |
| 2048 | 64 | 2 | 3.29 |
| 2048 | 64 | 4 | 6.97 |
| 4096 | 32 | 1 | 2.01 |
| 4096 | 32 | 2 | 3.73 |
| 4096 | 32 | 4 | 7.07 |
| 4096 | 64 | 1 | 1.87 |
| 4096 | 64 | 2 | 3.63 |
| 4096 | 64 | 4 | 7.27 |

TABLE 5.3
*The Hypercomputer HC System power consumption.*

| Problem size $N$ | block size $n_b$ | number of FPGAs | Power |
|---|---|---|---|
| 1024 | 32 | 1 | 18.88 |
| 1024 | 32 | 2 | 24.78 |
| 1024 | 32 | 4 | 27.14 |
| 1024 | 64 | 1 | 23.60 |
| 1024 | 64 | 2 | 27.14 |
| 1024 | 64 | 4 | 30.68 |
| 2048 | 32 | 1 | 17.70 |
| 2048 | 32 | 2 | 23.60 |
| 2048 | 32 | 4 | 31.68 |
| 2048 | 64 | 1 | 14.16 |
| 2048 | 64 | 2 | 22.42 |
| 2048 | 64 | 4 | 27.14 |
| 4096 | 32 | 1 | 16.52 |
| 4096 | 32 | 2 | 25.96 |
| 4096 | 32 | 4 | 35.40 |
| 4096 | 64 | 1 | 10.62 |
| 4096 | 64 | 2 | 16.52 |
| 4096 | 64 | 4 | 25.96 |

**5.5. Performance per Watt comparison between FPGA and microprocessor based system.** From figure 5.2, we can observe that for smaller problem sizes, the performance/watt of the Quad core system is far superior to the Hypercomputer system, because the commodity x86 host processor in the HC system dominates the power consumption. But for larger problem sizes, this inequality tends to reduce but not by much. On the other hand, when one considers only the FPGA board, its performance/watt is significantly

TABLE 5.4
*The Hypercomputer HC System power consumption.*

| Operating conditions of compute node | Power (Watts) |
|---|---|
| idle | 75.5 |
| boot up | 131 |
| 1 thread benchmark run | 113.5 |
| 2 thread benchmark run | 138.5 |
| 4 thread benchmark run | 176.5 |

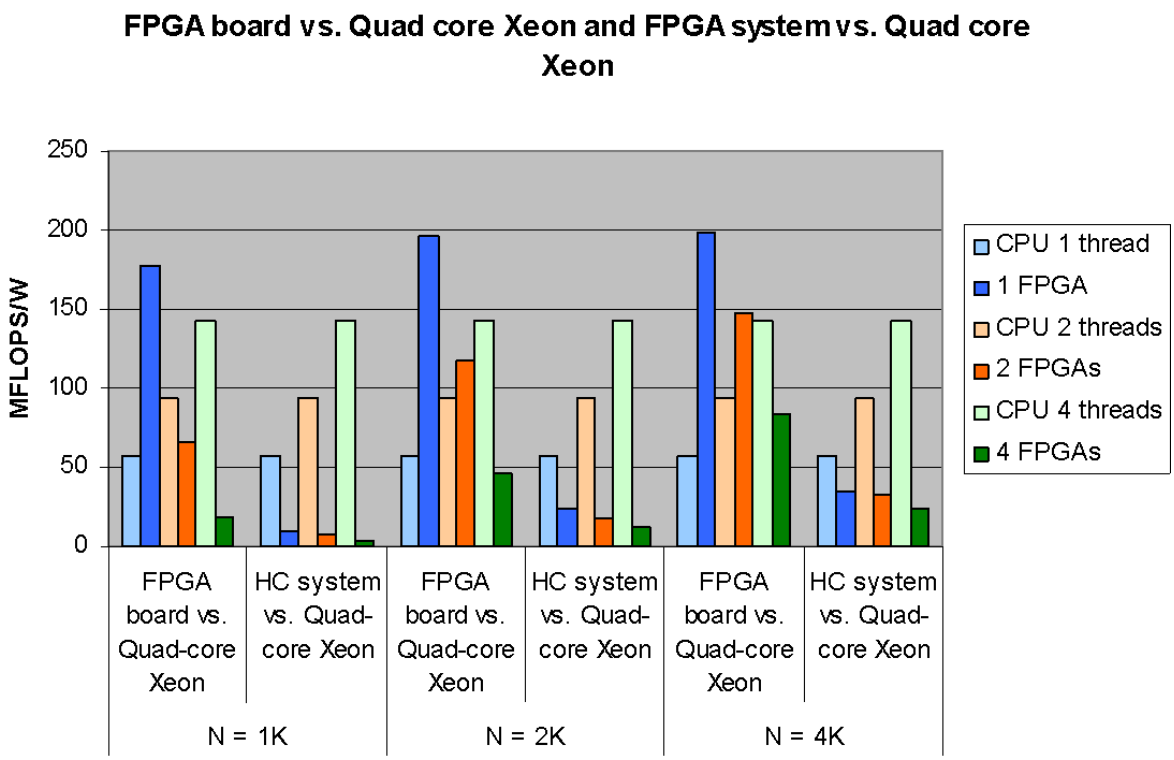



FIG. 5.2. *Performance/watt comparison of FPGA board and FPGA system with respect to a low power Quad core Xeon system for various matrix sizes*

superior compared to the Quad core system. This can be attributed to the extremely low power consumption of the FPGA devices. These results show that even for a cluster which was specifically designed for low power usage the FPGA system has a clear power advantage.

**6. Related and complementary work.** Hardware-based matrix operator implementation has been addressed by several researchers. Ahmed-El Amawy [15] proposes a systolic array architecture consisting of $(2N2 - N)$ processing elements which computes the inverse in $\mathcal{O}(N)$ time, where $N$ is the order of the matrix. However, there are no results to show that the large increase in area (for large values of $N$) is compensated for by the benefits obtained in speed by this implementation.

Lau et. al [25] attempt to find the inverse of sparse, symmetric and positive definite matrices using designs based on Single Instruction Multiple Data (SIMD) and Multiple Instruction Multiple Data (MIMD) architectures. This method is limited to a very specific sub-set of matrices and not applicable for a generic matrix and hence has limited practical utility. Edman and Owall [14] also targeted only triangular matrices.

Choi and Prasanna [8] implement LU decomposition on Xilinx Virtex II FPGAs (XC2V1500), using a systolic array architecture consisting of 8/16 processing units. This work is extended to inversion and supports 16-bit fixed point operations.

Data et. al [9] propose a single and double precision floating point LU decomposition implementation based on a systolic array architecture described in [8]. The systolic array architecture is a highly parallel realization and requires only a limited communication bandwidth. However, every element in the systolic array needs to have local memory and a control unit in addition to a computation unit, which adds significant overhead.

Wang and Ziavras [34] propose a novel algorithm to compute a LU decomposition for sparse matrices. This algorithm partitions the matrix into smaller parts and computes LU decomposition for each of them. The algorithm to combine the results makes use of the fact that most of the sub-blocks of the matrix would be zero blocks. However, this method cannot be extended to find LU decomposition for dense matrices.

Research efforts towards parallel implementations of LU decomposition largely deal with sparse linear systems. In some cases these implementations make use of a software package called SuperLU_DIST, which may be run on parallel distributed memory platforms [26, 30]. Other work using similar software package routines are found in [17]. A common platform that has been used for sparse matrix systems involving LU factorizations is the hypercube [4, 6]. Other implementations involving parallel LU linear system factorization and solutions may be found in [18, 24, 34, 35].

As the number of logic elements available on FPGAs increase, FPGA based platforms are becoming more popular for use with linear algebra operations [19, 33, 37]. FPGA platforms offer either a distributed memory system or a shared memory system with large amounts of design flexibility. One such design, presented in [19], utilizes FPGA based architecture with the goal of minimizing power requirements.

Any application implemented on an FPGA that uses external memory must provide some means of controlling the memory structure to store/access memory in an efficient manner. A common application that requires control of external memory is image processing. One group from Braunschweig, Germany has designed an SDRAM controller for a high-end image processor. This controller provides fixed address pattern access for stream applications and random address pattern access for events like a cache miss [28]. Another image processing application being worked on by a group from Tsinghua University in Beijing utilizes a memory controller specifically designed to reduce the latency associated with random access of off chip memory [27]. A design made to handle multiple streams of data was made by a group from the University of Southern California and the Information Sciences Institute. In this design each port in the data path as a FIFO queue attached to it. These data paths are also bound to an address generation unit used to generate a stream of consecutive addresses for the data stream [21].

The design presented in this paper is similar to the above mentioned work in the fact that it must both fetch and write data to an external memory device. However, in terms of complexity, the design in this paper is much simpler in that it provides specific streams of data at specific times for the LU processing engine. In such light it is not very flexible. However, simplicity has worked to the advantage that the design is easily replicated across multiple processing nodes. Another advantage that comes with simplicity is the low resource count the memory controllers take - roughly 13% of the available FPGA slices. This leaves much more room for the LU processing engine than a more complex design would.

**7. Conclusion.** In this paper we have presented detail performance numbers of a block based LU factorization algorithm on a multi FPGA system and compared the performance to a low power cluster compute node. The benchmarking results show that measured by raw compute performance the commodity microprocessor out performs the FPGA system by a factor of 30 for the current implementation and by 15 by moving to Xilinx Coregen IP cores. This is definitely an effect of the higher clock frequency and the floating point hardware on the microprocessor.

The performance picture changes when comparing the performance/Watts metric. Comparing the MFlops/Watt for the FPGA board, the FPGA board outperforms the low power microprocessor. This is similarly and effect of the lower clock frequency of the FPGA components. In the case of the complete HC system, which was not designed for low power usage, the host microprocessor system consumes most of the power and the power advantage of the FPGA board is lost. This shows that for a low power system the host system has to be a very low power system. This can be done since the host system does not have to perform any expensive computational tasks, but only needs to interface to disks and the FPGA board. Therefore, a really low power system could be used which would improve the performance of the FPGA based system dramatically.

During the design and benchmarking several weaknesses of the HPRC system and the FPGAs for scientific computing were uncovered. The following recommendations are proposed towards building better FPGA hardware for linear algebra and scientific computing:

- Single precision and double precision embedded ASICs in an FPGA system could increase the performance of FPGAs for scientific computing dramatically.
- Parallel access to BRAMs would enable better performance of the LU algorithm.
- Increasing the block size $n_b$ in our implementation is limited by number of BRAMs on the FPGA chip, but the performance benchmarks show that increasing the block size, increases the performance of the compute intensive part of the algorithm. Therefore, increasing the number of BRAMs on the FPGA would increase the performance for our LU implementation.

**Acknowledgment.** The authors wish to thank Starbridge Systems for hardware and software support. This work was funded in part by Lockheed Martin.

REFERENCES

[1] E. Anderson and J. Dongarra, *Performance of lapack: a portable library of numerical linear algebra routines*, Proceedings of the IEEE, 81 (1993), pp. 1094–1102.
[2] E. Angerson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammarling, J. Demmel, C. Bischof, and D. Sorensen, *Lapack: A portable linear algebra library for high-performance computers*, in Supercomputing '90. Proceedings of, 12-16 Nov. 1990, pp. 2–11.
[3] ———, *LAPACK: A portable linear algebra library for high-performance computers*, in Supercomputing 1990, 1990, pp. 2–11.
[4] K. Balasubramanya Murthy and C. Siva Ram Murthy, *A new parallel algorithm for solving sparse linear systems*, in Circuits and Systems, 1995. ISCAS '95., 1995 IEEE International Symposium on, vol. 2, 28 April-3 May 1995, pp. 1416–1419vol.2.
[5] D. Buell, J. Arnold, and W. Kleinfelder, *Splash 2: FPGAs in a Custom Computing Machine*, IEEE Computer Society Press, 1996.
[6] K. W. Chan, *Parallel algorithms for direct solution of large sparse power system matrix equations*, in IEEE Conference on Generation, Transmission and Distribution, vol. 148, 2001, pp. 615–622.
[7] Z. Chen, J. Dongarra, P. Luszczek, and K. Roche, *The LAPACK for clusters project: an example of self adapting numerical software*, in 37th Annual Hawaii International Conference on System Sciences, 2004, p. 10.
[8] S. Choi and V. Prasanna, *Time and energy efficient matrix factorization using fpgas*, in 13th International Conference on Field Programmable Logic and Applications (FPL 2003), 2003.
[9] V. Daga, G. Govindu, V. Prasanna, S. Gangadharapalli, and V. Sridhar, *Efficient floating-point based block lu decomposition on fpgas*, in International Conference on Engineering of Reconfigurable Systems and Algorithms, Las Vegas, 2004, pp. 21–24.
[10] E. D'Azevedo and J. Dongarra, *The design and implementation of the parallel out-of-core ScaLAPACK LU, QR, and Cholesky factorization routines*, Concurrency Practice and Experience, 12 (2000), pp. 1481–1493.
[11] A. Ditkowski, G. Fibich, and N. Gavish, *Efficient solution of $ax^{(k)} = b^{(k)}$ using $a^{-1}$*, Journal of Scientific Computing, (2006).
[12] J. Dongarra, *Constructing numerical software libraries for hpcc environments*, in High Performance Distributed Computing, 1994., Proceedings of the Third IEEE International Symposium on, 2-5 Aug. 1994, p. 4.
[13] J. Dongarra, R. Pozo, and D. Walker, *Scalapack++: an object oriented linear algebra library for scalable systems*, in Scalable Parallel Libraries Conference, 1993., Proceedings of the, 6-8 Oct. 1993, pp. 216–223.
[14] F. Edman and V. Owall, *Implementation of a scalable matrix inversion architecture for triangular matrices*, in Personal, Indoor and Mobile Radio Communications, 2003. PIMRC 2003. 14th IEEE Proceedings on, vol. 3, 7-10 Sept. 2003, pp. 2558–2562vol.3.
[15] A. El-Amawy, *A systolic architecture for fast dense matrix inversion*, IEEE Transactions on Computers, 38 (1989), pp. 449–455.
[16] E. Elmroth, F. Gustavson, I. Jonsson, and B. Kgstrom, *Recursive blocked algorithms and hybrid data structures for dense matrix library software*, SIAM Review, 46 (2004), pp. 3–45.
[17] K. Forsman, W. Gropp, L. Kettunen, D. Levine, and J. Salonen, *Solution of dense systems of linear equations arising from integral-equation formulations*, Antennas and Propagation Magazine, IEEE, 37 (1995), pp. 96–100.
[18] Y. Fung, W. Cheung, M. Singh, and M. Ercan, *A PC based parallel LU decomposition algorithm for sparse matrices*, in IEEE Pacific Rim Conference on Communications, Computers and Signal Processing, vol. 2, 2003, pp. 776–779.
[19] G. Govindu, S. Choi, V. Prasanna, V. Daga, S. Gangadharpalli, and V. Sridhar, *A high-performance and energy-efficient architecture for floating-point based lu decomposition on fpgas*, in Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International, 26-30 April 2004, p. 149.
[20] P. Graham and M. Gokhale, *Reconfigurable Computing: Accelerating Computation with Field-Programmable Gate Arrays*, Springer, 2005.
[21] S. Heithecker, A. do Carmo Lucas, and R. Ernst, *A mixed qos sdram controller for fpga-based high-end image processing*, in Signal Processing Systems, 2003. SIPS 2003. IEEE Workshop on, 27-29 Aug. 2003, pp. 322–327.
[22] C. hsing Hsu and W. chun Feng, *A feasibility analysis of power awareness in commodity-based high-performance clusters*, in IEEE International Conference on Cluster Computing (Cluster 2005), Boston, MA, September 2005.
[23] Intel, *Intel Math Kernel Library for Linux*, http://developer.intel.com, document number: 314774-003us ed., April 2007.
[24] S. Kratzer, *Massively parallel sparse lu factorization*, in Fourth Symposium on the Frontiers of Massively Parallel Computation, 1992, pp. 136–140.

[25] K. Lau, M. Kumar, and R. Venkatesh, *Parallel matrix inversion techniques*, in IEEE Second International Conference on Algorithms and Architectures for Parallel Processing, 1996, pp. 515 – 521.
[26] X. S. Li and J. Demmel, *Superlu dist: A scalable distributed-memory sparse direct solver for unsymmetric linear systems*, ACM Transactions on Mathematical Software (TOMS), 29 (2003), pp. 110–140.
[27] Z. Liu, K. Zheng, and B. Liu, *Fpga implementation of hierarchical memory architecture for network processors*, in Field-Programmable Technology, 2004. Proceedings. 2004 IEEE International Conference on, 2004, pp. 295–298.
[28] J. Park and P. Diniz, *Synthesis and estimation of memory interfaces for fpga-based reconfigurable computing engines*, in Field-Programmable Custom Computing Machines, 2003. FCCM 2003. 11th Annual IEEE Symposium on, 9-11 April 2003, pp. 297–299.
[29] M. Perl and T. Hauser, *Processing high-speed stereo particle image velocimetry data with an integrated cluster supercomputer*, in 45th AIAA Aerospace Sciences Meeting and Exhibit, no. AIAA-2007-51, Reno, NV, January 8-11 2007, AIAA. Tracking number 67826.
[30] X.-Q. Sheng and E. Kai-Ning Yung, *Implementation and experiments of a hybrid algorithm of the mlfma-enhanced fe-bi method for open-region inhomogeneous electromagnetic problems*, Antennas and Propagation, IEEE Transactions on, 50 (2002), pp. 163–167.
[31] S. Trimberger, *Field-Programmable Gate Array Technology*, Springer, 1994.
[32] K. Underwood, *FPGAs vs. CPUs: trends in peak floating-point performance*, in FPGA '04: Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays, New York, NY, USA, 2004, ACM Press, pp. 171–180.
[33] X. Wang and S. Ziavras, *Performance optimization of an FPGA-based configurable multiprocessor for matrix operations*, in IEEE International Conference on Field-Programmable Technology (FPT), 2003, pp. 303–306.
[34] X. Wang and S. Ziavras, *A configurable multiprocessor and dynamic load balancing for parallel lu factorization*, in Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International, 26-30 April 2004, p. 234.
[35] J. Q. Wu and A. Bose, *Parallel solution of large sparse matrix equations and parallel power flow*, Power Systems, IEEE Transactions on, 10 (1995), pp. 1343–1349.
[36] C. Xuebin, L. Yucheng, S. Jiachang, Z. Yunquan, and Z. Peng, *Developing high performance blas, lapack and scalapack on hitachi sr8000*, in High Performance Computing in the Asia-Pacific Region, 2000. Proceedings. The Fourth International Conference/Exhibition on, vol. 2, 14-17 May 2000, pp. 993–997vol.2.
[37] L. Zhuo and V. Prasanna, *Scalable hybrid designs for linear algebra on reconfigurable computing systems*, in Parallel and Distributed Systems, 2006. ICPADS 2006. 12th International Conference on, vol. 1, 12-15 July 2006, p. 9pp.

*Edited by:* Javier Díaz
*Received:* October 8th, 2007
*Accepted:* December 10th, 2007

Scalable Computing: Practice and Experience
Volume 8, Number 4, pp. 387–394. http://www.scpe.org
ISSN 1895-1767


# A COMPUTING ARCHITECTURE FOR CORRECTING PERSPECTIVE DISTORTION IN MOTION-DETECTION BASED VISUAL SYSTEMS*

SONIA MOTA†, EDUARDO ROS‡, AND FRANCISCO DE TORO§

**Abstract.** The projection of 3D scenarios onto 2D surfaces produces distortion on the resulting images that affects the accuracy of low-level motion primitives. Independently of the motion detection algorithm used, post-processing stages that use motion data are dominated by this distortion artefact. Therefore we need to devise a way of reducing the distortion effect in order to improve the post-processing capabilities of a vision system based on motion cues. In this paper we adopt a space-variant mapping strategy, and describe a computing architecture that finely pipelines all the processing operations to achieve high performance reliable processing. We validate the computing architecture in the framework of a real-world application, a vision-based system for assisting overtaking manoeuvres using motion information to segment approaching vehicles. The overtaking scene from the rear-view mirror is distorted due to perspective, therefore a space-variant mapping strategy to correct perspective distortion arterfaces becomes of high interest to arrive at reliable motion cues.

**Key words.** Real-time computing, high performance computing, fine grain pipeline, image processing.

**1. Introduction.** Animals and human beings have powerful tools for processing information. Recent advances in biological neural circuits and processing schemes is one of the reasons of a new tendency in engineering that emulates specific biological computation schemes, this is the research paradigm called *neuromorphic engineering*. The objective is to achieve more effective machines with a huge potential impact on industry and society [1, 2, 3, 4].

Vision is one of the most important senses for animals' survival. In particular, visual motion detection is the most important information source and constitutes a complex and accurate system. The long-medium term goal is to implement devices based on vertebrates' visual systems, because of their astonishing efficiency in analysing dynamic scenes. However, current vision models based on vertebrates require high computational cost while most real-time applications cannot be addressed with traditional computer vision strategies due to their complexity.

But adapting bio-inspired processing schemes on silicon is a complex task. The neural system has synaptic plasticity (the connection from neuron A to neuron B changes in order to stabilize specific neural activity patterns in the brain, for instance with neural adaptation strategies such as *Hebbian learning* [5]) that allows response to changes to different stimulus or environments. Furthermore the connectivity among neurons in biological tissues takes place in three dimensions. In contrast, the silicon systems allow only two-dimensional connectivity among computational threads and lack abilities such as local synaptic plasticity.

Biological systems use efficiently massive parallel processing to overcome the slow chemical-based computing that takes place in neurons. This advantage of biological systems is shared by current FPGA devices. Different researchers are working in this direction, i. e. bio-inspired visual systems implemented on FPGAs devices with massively parallel computation using fine grain processing architectures [6, 7, 8, 9]. This approach allows real-time image processing and represents a first step towards solutions to particular problems in a wide range of applications

However, even biological systems need to project 3D scene onto a 2D surface (for instance, a retina or a camera sensor) before extracting data. Due to the 2D projection the scene is distorted by perspective. This affects motion processing, a moving object, although moving at a constant speed, seems to accelerate and its size increases as it approaches the camera. This apparent enlargement adds an expanding motion to the translational one, and the perception of different velocities in different regions of an object.

Biological systems use low level stereo information or other visual modalities in higher level processing stages to deal with the perspective distortion. But uni-modal motion-based artificial systems require other strategies

*This work has been supported by the European Project DRIVSCO (IST-2001-35271) and the National Project DEPROVI (DPI2004-07032)

†Departamento Informática y Análisis Numérico, Universidad de Córdoba, Campus de Rabanales s/n, Edificio Albert Einstein, 14071, Córdoba, Spain (smota@uco.es). Questions, comments, or corrections to this document may be directed to that email address.

‡Departamento de Arquitectura y Tecnología de Computadores, Universidad de Granada, Periodista Daniel Saucedo Aranda s/n, 18071 Granada, Spain (eros@atc.ugr.es).

§Departamento de Teoría de la Señal, Telemática y Comunicaciones, Universidad de Granada, Periodista Daniel Saucedo Aranda s/n, 18071 Granada, Spain (ftoro@ugr.es).

to compensate this effect. We propose a scheme that corrects perspective distortion so that motion information can be used in a reliable manner: *Space-variant mapping* (SVM) method. It is possible to compensate for the effect of perspective by remapping the image before extracting motion. This processing unit can be connected to the whole motion detection system as a pre-processing stage of the image.

The rest of this paper is organized as follows: section 2 introduces the space variant mapping method; section 3 describes the hardware implementation and cost; and section 4 presents an example of perspective distortion correction in a real-world task, an overtaking monitoring system. Furthermore, the perspective distortion correction is described using two different methods: space variant mapping (SVM) and another bio-inspired method based on neural integration of information that we use in order to validate the results and compare the two different approaches.

**2. Space-variant mapping method.** The *space-variant-mapping* (SVM) method is the selected strategy for dealing with perspective distortion. The Space Variant Mapping [10, 11] is an affine coordinate transformation that aims at reversing the process of projection of a 3-D scene onto a 2-D surface. It is possible to invert the projection equations and to compensate the effect of perspective by remapping the original image. In this approach (a) parallel lines and equal distances in the real scene are remapped to parallel lines and equal distances in the processed (remapped) image and (b) it is assumed that the depth of the scene, i. e. distance to the camera projected on its optical axis, varies linearly.

Generally, distances closer to the image plane are projected onto larger segments. Using these assumptions the SVM approach re-samples the original image. We assume a specific camera configuration targeting the left vision field with respect to the optical axis. In this case, the required remapping is done by expanding the left-hand side of the image (corresponding to the part of the scene furthest away from the camera) and collapsing the right-hand side (corresponding to the part of the scene closest to the camera). The coordinates at the distorted space are transformed in new coordinates at the remapped space. The operations involved in the process are additions, multiplications, divisions and trigonometric operations (sine and tangent).

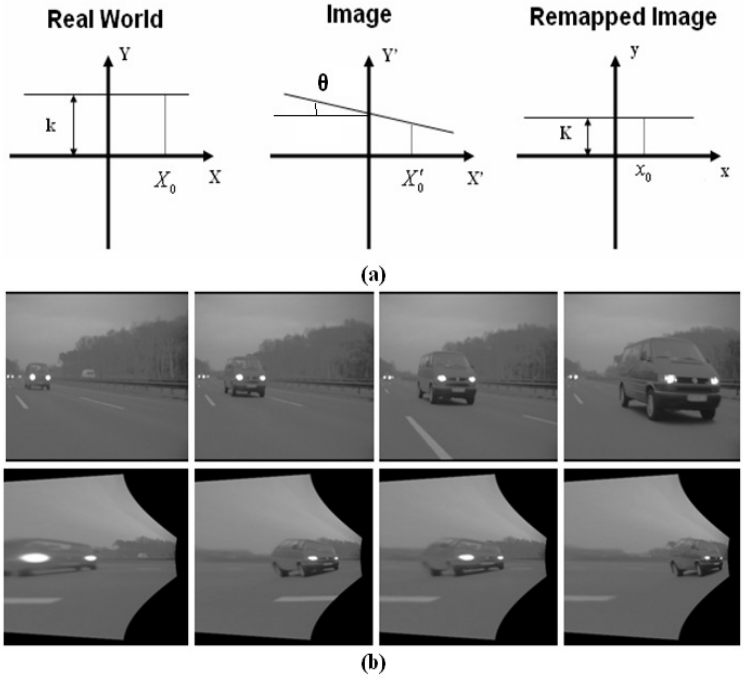



FIG. 2.1. ***(a)*** *Coordinates transformation;* ***(b)*** *Original and remapped image of an overtaking sequence.*

Figure 2.1a shows the coordinates transformation that is required in order to correct the perspective distortion due to the projection of the 3D scene onto a 2D surface. Figure 2.1b shows an example of a re-sampled image from the real-world application described in Section 4.

The blurred appearance of the left-hand side of the image is generated by the interpolation process necessary to resize a small portion of the original image into a larger area. The interpolation method used here is the

truncated Taylor expansion, known as *local Jet* [12]. In the remapped scenario the mean speed of a car that is actually overtaking at a constant relative speed is more constant along the sequence. Furthermore, each point that belongs to the rigid body moves approximately at the same speed (Figure 2.2). On the other hand, on the right-hand side of the remapped image, we are subsampling the original image, which means that aliasing effects may occur.

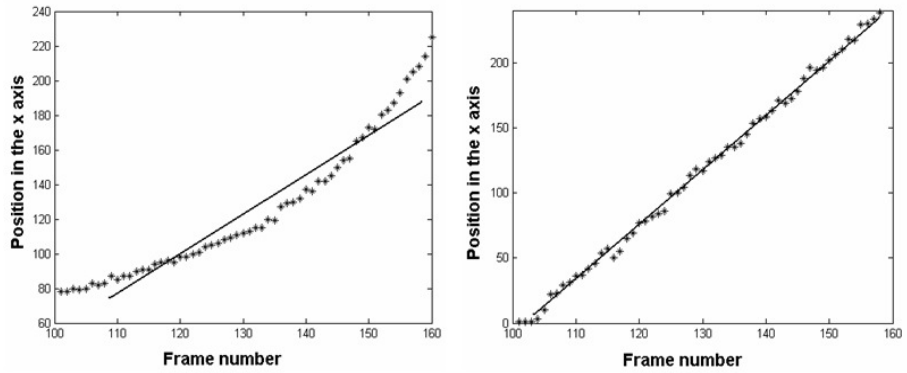



FIG. 2.2. *Space Variant Mapping makes stable the speed along the sequence. On the left plot we represent the x position of the centre of the car along the constant-speed overtaking sequence. We see that although the overtaking sequence speed is constant the curve is deformed (constant speed is represented by a line) due to the perspective distortion. On the other hand, the right plot shows the same result on a remapped sequence. In this case the obtained overtaking speed is constant (accurately approximated by a line with a slope that encodes the speed).*

The advantage of SVM is that the effect of perspective is compensated through the remapping scheme and the acceleration artefact is removed. In the real-time application described in Section 4, we have manually marked the overtaking car position along a scene, in this way it is easy to compute the centre of the marked area, i. e. the overtaking car, and its speed. Figure 2.2 shows the compensation effect on the speed of the centre of the overtaking car.

Furthermore SVM reduces the difference between the extracted speeds of the front and rear of an overtaking car. Finally, the remapped image is easier to interpret using motion estimation information.

**3. Hardware implementation.** We use conventional cameras that provide 30 frames per second and 256 gray levels. The processed image size is of 640 x 480 pixels. The prototyping computing platform has 2 SRAM banks and a Xilinx Virtex-II FPGA (XC2V1000 device) [13]. This device allocates 1 million system gates distributed in 5,120 slices and 40 embedded memory blocks of a total of 720 Kbits.

The whole system has been implemented on the FPGA device (see Figure 3.1). This system includes the processing stages (space variant mapping, motion detection algorithm [14] and specific circuits for packing and unpacking temporal data) and the interface elements (frame-grabber, memory management units and VGA output interface).

The complete system is designed with independent processing modules. The architecture design adopts a fine grain pipeline structure for all modules. Specific communication channels are used in order to connect the modules with each other.

In this way, space variant mapping (SVM) constitutes a pre-processing stage before the motion estimation module. The architecture of the whole system allows changing modules of the datapath if necessary, i. e. we can use different modules implementing diverse motion-detection algorithms with the same system.

SVM architecture is also implemented as a fine grain pipeline structure to ensure a successful connection with the motion extraction module at 1 pixel per clock cycle. Motion primitives are computed using a fine grain pipeline structure that consumes 1 cycle per stage. If necessary, it is possible to reduce the parallelism in the SVM module (and consequently its efficiency) to fit the processing performance of the motion-detection module requirements (if other motion estimation schemes are used). Alternatively, we can replicate the SVM module and split the image to send parts of the original image to the different SVM units increasing the processing velocity if further performance is required. Therefore the architecture is modular and scalable. Figure 3.1 shows the data flow of the integrated system.

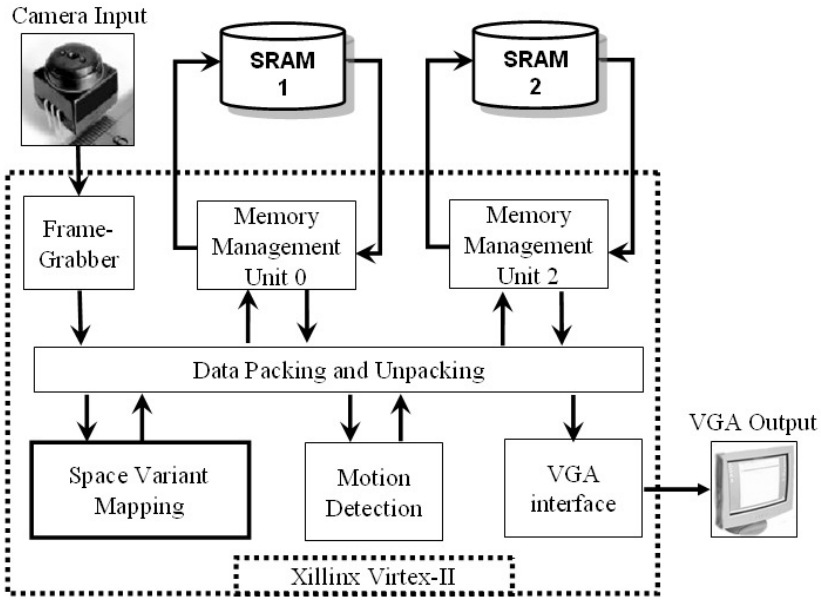



FIG. 3.1. *Complete system: motion detection after correcting the distortion by space variant mapping preprocessing.*

SVM uses several multiscalar units consistent with the goal. To transform each pixel coordinates the operations that take place are additions, multiplications, divisions and trigonometric operations (sines and tangents). To compute sine and tangent we use a look up tables, and to compute the divisions we use optimized cores customized for our application. Each core computes one division and consumes one cycle. We use two division cores. Figure 3.2 shows the pipeline structure of the modules related with perspective distortion correction. Rectangles represent multiscalar units. Rectangles on a column are working in parallel. Rectangles on a row represent different pipeline stages. Numbers in brackets are the number of micropipelined stages. The final block represents the motion estimation datapath.

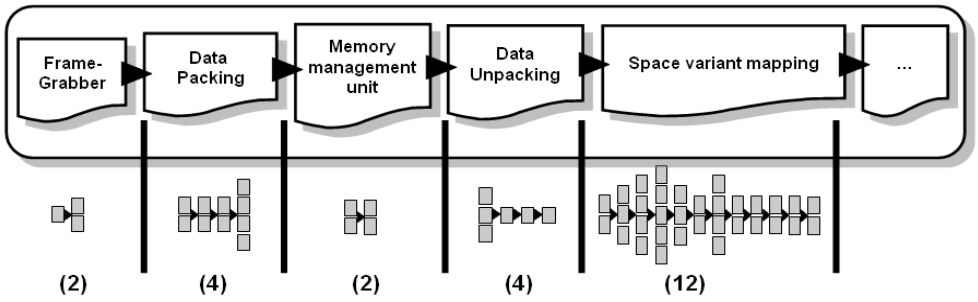



FIG. 3.2. *Data flow and pipelined structure of the perspective distortion correction datapath.*

The SVM module takes 12 pipeline stages, and only one division core that produces 28 clock cycles of latency.

Table 1 summarizes the main performance and hardware cost of the system implemented. The hardware costs in the table are estimates extracted from the ISE environment. Note that the maximum clock frequency advised by the ISE environment is limited to 36.1 MHz (Table 1). This is because we use a specific core for the division that limits the global frequency of the whole pipelined structure. However, the circuit frequency fully allows computation at camera frame-rate.

One of the important bottlenecks for FPGA processing capability is the external memory access. There are several reasons to use the external SRAM: first of all, conventional cameras interlace the image (they send even rows first and then odd rows of a scanned image). Therefore, in order to compute the image it is necessary to previously de-interlace the image, i. e. to arrange the rows in properly appearance order. Furthermore, SRAM

TABLE 3.1

*Hardware cost of the different stages of the described system. The global clock of the design is running at 31.5 MHz, although the table includes the maximum frequency allowed by each stage. The data of the table has been extracted using the ISE environment.*

| Pipeline Stage | Number of Slices | % Device Occupation | Max. Fclk.(MHz) |
|---|---|---|---|
| Frame-Grabber | 753 | 14 | 75.9 |
| Memory Management Units | 581 | 11 | 53.8 |
| Space-Variant Mapping | 838 | 16 | 36.1 |

access is shared by space variant mapping modules and motion detection modules. Finally, the synchronization among different modules related to different clock frequencies (frame-grabber, VGA, etc.) is done with external memories.

Using exclusively embedded memory blocks becomes not possible due to the image size. Therefore, the necessity of storing data in external SRAM banks forces us to design a module that allows the writing and reading to/from the SRAM banks as efficiently as possible. This process of storing/recover data is sequential and consumes 2 cycles per pixel (1 cycle is consumed in assigning the address and 1 cycle is consumed in transferring the data). The access control is carefully designed. We define different reading and writing ports using a double-buffer technique to avoid temporization problems. We use a micropipelined architecture to access two different ports. A state machine feeds the reading/writing ports sequentially, achieving a performance of one data per cycle. Furthermore, it is feasible to store several pixels at each memory address due to the memory word size. In this way we can reduce the number of external memory accesses. For this purpose we use specific packing and unpacking circuits in the pipelined architecture (see Figures 3 and 4).

**4. Real-world application.** One of the most dangerous operations in driving is to overtake another vehicle. The driver's attention is on the road, and sometimes he does not use the rear-view mirror or it is unhelpful when an overtaking car is at the blind spot. Therefore an automatic alarm system is of interest in these scenarios.

Systems based on vision would be very effective in driving assistance; in fact the driver himself uses vision and represents a good proof of the concept. We place a camera onto the real-view mirror to cover the blind spot area. If an overtaking vehicle approaches the host car it is detected as forward moving features, while the rest of the patterns in the camera visual field move backwards due to the ego-motion of the host vehicle. Therefore motion provides useful cues to achieve an efficient segmentation in this application framework. In this context, the sequences taken with a camera fixed onto the driver's rear-view mirror are strongly deformed by the perspective, and reducing the deformation effect is necessary in order to enhance the segmentation capabilities of a motion-based vision system.

We define two different methods to deal with the perspective distortion. On one side, we use space variant mapping method, and on the other side, for validation purposes we use an alternative bio-inspired method based on neural integration of information.

Many studies suggest that the integration of local information allows the discrimination of objects in a noisy background [15, 16, 17, 18]. The mechanism of this integration in biological systems is almost unknown. We define *velocity channels* based on motion patterns of the image that seem to correspond to independent moving objects (rigid bodies) [19]. Each velocity channel computes a population of features moving coherently (by sharing velocity and direction in a local area). The velocity channels are processed in a competitive manner and the one that integrates a maximum number of features moving coherently in an area becomes salient. In this way, low quality motion-detection estimations, i. e. errors, are filtered.

The system has been tested on real overtaking sequences in a wide speed range.

The *"centre of mass"* of obtained features (moving coherently) is used to validate the quality of moving features. We manually mark the overtaking car by drawing a rectangle (around it). We calculate the distance between the centre of mass and the centre of the rectangle. This distance is normalized by dividing it by the radius of the minimum circle containing the rectangle in each frame. This distance is what we call *Quality Measure* (QM). If the centre of mass falls into this circle this QM is below 1. In this case we assume that we are detecting the overtaking vehicle accurately. In other cases the QM is higher than 1, motion detection has dominant noisy patterns (motion detection is assumed to be of low quality) leading to incorrect estimations. Figure 4.1a shows the QM along the sequence when velocity channels method is adopted, and Figure 4.1b shows QM throughout the sequence when the space-variant mapping method is adopted.

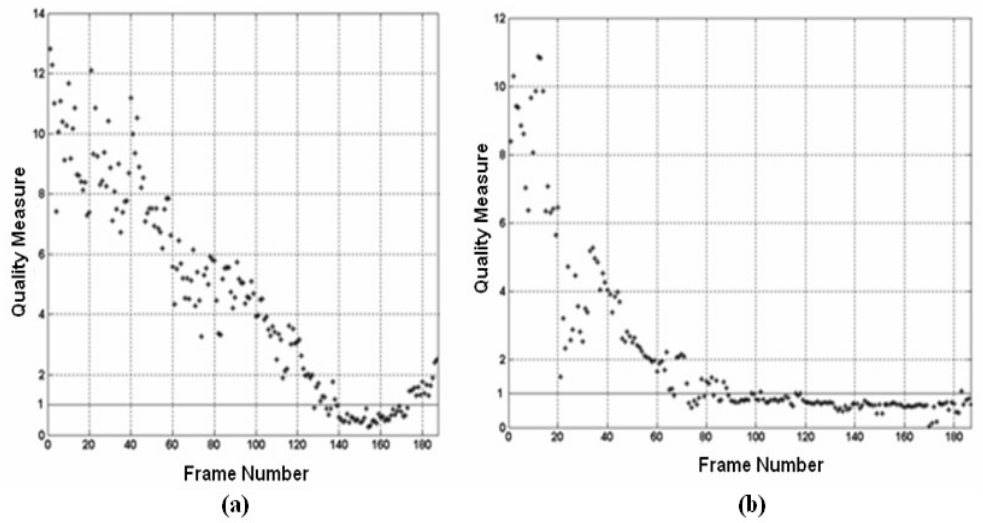



FIG. 4.1. *Quality measurement plot of a sequence adopting:* ***(a)*** *Velocity-channels method;* ***(b)*** *Space-variant-mapping method.*

When the velocity channels method (VC) is used, motion detection is accurate from frame 138 to 175, and when the space variant mapping (SVM) method is adopted, motion detection is accurate from frame 89 to the end of the sequence. In fact accurate detection occurs when the overtaking vehicle begins to be dangerously close (see Figure 4.1).

We used four sequences to test the space variant mapping scheme. The results are summarized in Figure 4.2. The first and the fourth sequences were taken with a CCD camera on a sunny day. In the first sequence the overtaking car approaches from the distance and in the fourth, it suddenly appears into our line of vision. The second and third sequences are HDR ones. The second one corresponds to a cloudy day with some mist and the other was taken in twilight conditions. These two sequences show overtaking processes by far-away cars with their lights on.

Figure 4.2 shows that motion detection is done properly from a vehicle size of 10660 pixels with the VC method and 3216 pixels with the SVM method. This size is only approximate, taken as it is from the size of the confidence rectangle used to calculate QM. The data in the next column represents the number of features detected moving rightwards, on which the estimation is based.

In the HDR sequences the cars have their lights on, and adverse weather conditions reduce noisy detection. The best detected features belong to the overtaking car lights and allow an early success in the tracking task with both methods.

SVM constitutes a good method for medium distances in all weather conditions, even when the cars have no lights on that facilitate their detection.

**5. Conclusions.** We have presented a perspective distortion correction for a vision-based segmentation system.

Using a real-world sequence of a car moving at constant speed we showed that the SVM considerably reduces the spurious acceleration effect due to perspective projection and improves motion estimation results.

We have compared the results of Space-variant mapping method with a bio-inspired one based on neural integration of information. Adopting space variant mapping method the results based on motion information are improved.

We have designed a pipelined computing architecture that takes full advantage of inherent parallelism of FPGA technology. In this way we achieve computing speeds of 36.1 Mpixels (for instance, around 30 frames per second with 1280x960 image resolution) that allow fully computation at camera frame-rate (25-30 frames per second).

The architecture is modular and scalable.

This contribution is a good case of study that illustrates how very diverse processing stages can be finely pipelined in order to achieve high performance.

| Sequence Frame | Method used | 1st. Frame in successful tracking | Vehicle Size (pixels) | Best results |
|---|---|---|---|---|
|  | VC | 139 | 10660 | |
| | SVM | 89 | 3216 | √ |
|  | VC | 1 | 899 | √ |
| | SVM | 1 | 899 | √ |
|  | VC | 1 | 1813 | √ |
| | SVM | 1 | 1813 | √ |
|  | VC | 36 | 32469 | |
| | SVM | 23 | 14385 | √ |

FIG. 4.2. *Results of the two methods applied to four different sequences. "1st frame of successful tracking" represents image number in a sequence from which the motion detection is of high quality, i. e. QM is below 1. "Vehicle size" is the number of pixels inside the manually drawn rectangle that contains the overtaking car in the "1st frame of successful tracking".*

Finally the hardware resources of the system are not very high. Therefore, the presented approach can be considered a moderate cost module for the real world application of the overtaking car monitor.

## REFERENCES


[1] E. ROS, E. M. ORTIGOSA, R. AGÍS, M. ARNOLD AND R. CARRILLO, *Real time computing platform for spiking neurons Real Time Spiking Neurons (RT-Spike)*, IEEE Trans. Neural Networks, 17(4) (2006), pp. 1050–1063.

[2] S. MOTA, E. ROS, E. M. ORTIGOSA AND F. J. PELAYO, *Bio-Inspired motion detection for blind spot overtaking monitor*, Int. Journal of Robotics and Automation, 19(4) (2004), pp. 190–196.

[3] A. BROGGI, P. CERRI AND P. C. ANTONELLO, *Multi-resolution vehicle detection using artificial vision.*, in Intelligent Vehicles Symposium, 2004, pp. 310–314.

[4] T. DELBRUCK, *Silicon retina with correlation-based, velocity-tuned pixels*, IEEE Trans. Neural Networks 4 (1993), pp. 529-541.

[5] D. O. HEBB, *The organization of behavior*, Wiley, New York, 1949.

[6] J. DÍAZ, E. ROS, F. PELAYO, E. M. ORTIGOSA, AND S. MOTA, *FPGA based real-time optical-flow system*, IEEE Trans. Circuits for Video Technology, 16(2) (2006), pp. 274–279.

[7] S. MOTA, E. ROS, J. DÍAZ AND F. TORO, *General purpose real-time image segmentation system*, Lecture Notes in Computer Science 3985 (2006), pp. 164-169.

[8] F. AUBÉPART AND N. FRANCESCHINI, *Bio-inspired optic flow sensors based on FPGA: Application to Micro-Air-Vehicles*, Microprocessors and Microsystems 31(6), (2007), pp. 408–419.

[9] P. CHALIMBAUD AND F. BERRY, *Embedded active vision system based on an FPGA architecture*, EURASIP Journal on Embedded Systems, volume 2007 (2007), Special Issue on Embedded Vision System.

[10] H. MALLOT, H. H. BULTHOFF, J. J. LITTLE AND S. BOHRER, *Inverse perspective mapping simplifies optical flow computation and obstacle detection*, Biol. Cybern., 64 (1991), pp. 177–185.

[11] S. TAN, J. DALE AND A. JOHNSTON, *Effects of Inverse Perspective Mapping on Optic Flow*, in ECOVISION Workshop, 2004, (Isle of Skye, Scotland, UK).

[12] L. FLORACK, B. TER HARR ROMENY, M. VIERGEVER AND J. KOENDERINK, *The Gaussian Scale-Space paradigm and the multiscale local Jet*, Int. J. Comp. Vis. 18 (1996), pp. 61–75.

[13] WWW.XILINX.COM

[14] S. MOTA, E. ROS, J. DÍAZ, R. RODRIGUEZ AND R. CARRILLO, *A space variant mapping architecture for reliable car segmentation*, Lecture Notes in Computer Science 4419 (2007), pp. 337–342.

[15] H. B. BARLOW, *The efficiency of detecting changes of intensity in random dot patterns*, Vision Research, 18(6) (1978), pp. 637–650.

[16] D. J. Field, A. Hayes and R. F. Hess, *Contour integration by the human visual system: evidence for local "association field"*, Vision Research, 33(2) (1993), pp. 173–193.
[17] J. Saarinen, D. Levi and B. Shen, *Integration of local pattern elements into a global shape in human vision*, in Proceeding of the National Academic of Sciences USA, Vol. 94, 1997, pp. 8267–8271.
[18] C. D. Gilbert and T. N. Wiesel, *Intrinsic connectivity and receptive field properties in visual cortex*, Vision Research, 22(2) (2005), pp. 125–177.

*Edited by:* Javier Díaz and Dorothy Bollman
*Received:* December 14th, 2007
*Accepted:* December 27, 2007

Scalable Computing: Practice and Experience
Volume 8, Number 4, pp. 395–410. http://www.scpe.org
ISSN 1895-1767


# THROUGHPUT IMPROVEMENT OF MOLECULAR DYNAMICS SIMULATIONS USING RECONFIGURABLE COMPUTING*

SADAF R. ALAM, PRATUL K. AGARWAL, JEFFREY S. VETTER† AND MELISSA C. SMITH‡

**Abstract.** A number of grand-challenge scientific applications are unable to harness Terflops-scale computing capabilities of massively-parallel processing (MPP) systems due to their inherent scaling limits. For these applications, multi-paradigm computing systems that provide additional computing capability per processing node using accelerators are a viable solution. Among various generic and custom-designed accelerators that represent a data-parallel programming paradigm, FPGA devices provide a number of performance enhancing features including concurrency, deep-pipelining and streaming in a flexible manner. We demonstrate acceleration of a production-level biomolecular simulation, in which typical speedups are less than 20 on even the most powerful supercomputing systems, on an FPGA-enabled system with a high-level programming interface. Using accurate models of our FPGA implementation and parallel efficiency results obtained on the Cray XT3 system, we project that the time-to-solution is reduced significantly as compared to the microprocessor-only execution times. A further advantage of computing with FPGA-enabled systems over microprocessor-only implementations is performance sustainability for large-scale problems. The computational complexity of a biomolecular simulation is proportional to its problem sizes, hence the runtime on a microprocessor increases at a much faster rate as compared to FPGA-enabled systems which are capable of providing very high throughput for compute-intensive operations thereby sustaining performance for large-scale problems.

**Key words.** field programmable gate arrays, molecular modeling, performance modeling and projections

**1. Introduction.** Despite the tremendous computing power, flexibility, and power and cost efficiency of the FPGA devices, their use in scientific high performance computing (HPC) has been limited to numerical functions and kernels that are implemented in a hardware description language (HDL) [30, 39]. The idiosyncrasies of the HDLs and limited support for floating-point (FP) operations restrict the ability of scientific code developers to port their algorithms and applications, let alone to exploit the full potential of these devices. In this paper, we present an analysis and FPGA implementation of a biomolecular calculation called the Particle-Mesh Ewald (PME) method using High-Level Languages (HLLs), and report application speedup results. This specific PME method is part of a widely-used molecular dynamics (MD) framework called AMBER [1, 21]. AMBER, a collection of programs including system preparation, simulation and analysis, allows application scientists to carry out complete experiments of biomolecular systems. MD techniques allow application scientists to study the dynamics of large macromolecules, including biological systems such as proteins, nucleic acids (DNA, RNA) membranes. The *sander* module of AMBER is the most commonly used module for system simulations; furthermore, system simulations are the most time-consuming part of an experiment. Within the sander module, there are a number of algorithms for simulating a system. The PME method is used in most explicit solvent experiments including the simulations for protein structure, dynamics and functions [23].

**Motivation.** A number of strategies have been employed in attempts to accelerate the PME calculations on traditional supercomputing platforms such that scientists can simulate their experiments at native time and length scales. Currently, even the fastest computers are $10^4$–$10^6$ magnitudes short of what is desired for even investigations of a medium-scale simulation. Our analysis revealed that the PME algorithm implementation in the sander module in the AMBER framework version 8.0 does not scale beyond 32 and 64 processors on the most powerful supercomputers including the IBM Blue Gene/L and the Cray XT3 systems [16]. Although no system-specific optimization are considered in these experiments, the applications have been compiled using the double-hummer optimization flags offered by the IBM XL compilers on the Blue Gene/L system and SSE optimization flags of the PGI compiler on Cray XT platforms. A number of researchers have identified the factors that limit the performance and scaling of PME algorithms on microprocessors and massively-parallel systems [21], [22]. Figure 1.1 shows the scaling characteristics of the PME calculations in the strong scaling mode (fixed problem size) and Figure 1.2 shows the performance in the weak scaling mode on the Blue Gene/L system. Pico-seconds per simulation day (psec/day) is the science-based metric used by application scientists to measure a simulation performance. A high value of psec/day is essential for longer time scale simulations. Analysis of the PME implementation reveals two major limiting factors. First, the message volume is roughly

*Research sponsored by the Laboratory Directed Research and Development Program of Oak Ridge National Laboratory, managed by UT-Battelle, LLC, for the U. S. Department of Energy under Contract No. DE-AC05-00OR22725.
†Computer Science and Mathematics Division, Oak Ridge National Laboratory, USA
‡Department of Computer and Electrical Engineering, Clemson University, USA

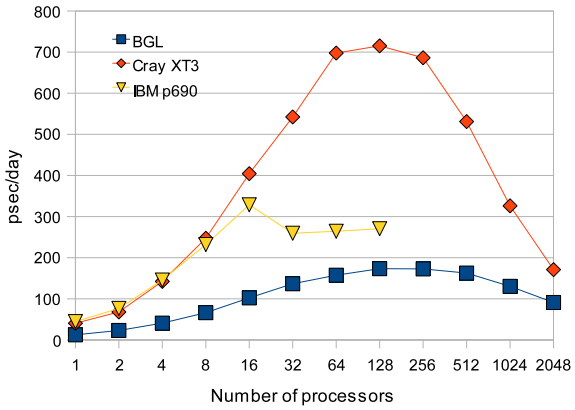



FIG. 1.1. *AMBER (PME method) scaling in the strong scaling mode with a 62K atoms system.*

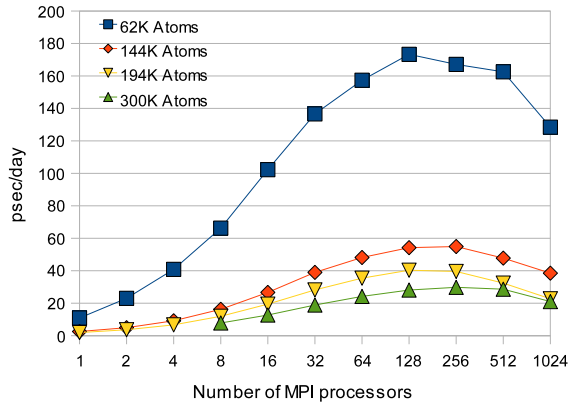



FIG. 1.2. *AMBER (PME method) scaling for four different system sizes: 62K, 144K, 194K and 300K atoms on the Blue Gene/L (BGL) system.*

constant with the number of processors or MPI tasks. This limits parallel speedup to a maximum of 12x to 15x over microprocessor runtimes, even for systems like Blue Gene/L and Cray XT3 that provide relatively high communication bandwidth ratios compared to common SMP cluster systems. Second, the application's memory capacity requirements do not scale with the number of processors, since all processors store the positions of all atoms [16, 21, 22, 40]. This is especially challenging for the emerging multi-core systems.

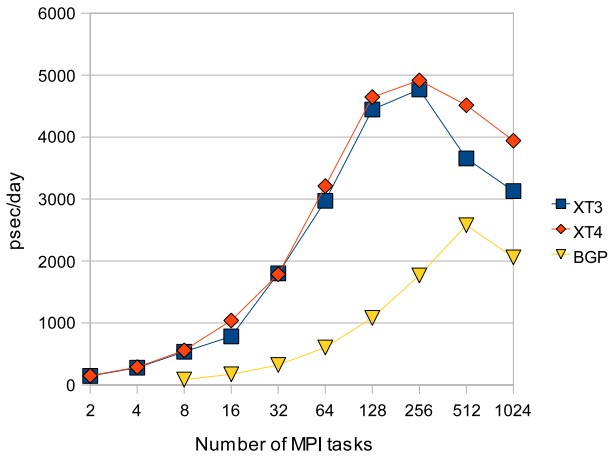



FIG. 1.3. *Scaling of PMEMD (Amber 9) simulations on contemporary massively parallel systems*

In order to address these known limitations of sander, another module known as Particle Mesh Ewald Molecular Dynamics (PMEMD) has been developed with the major goal of improving performance of PME in molecular dynamics simulations and minimizations by Robert E. Duke and Lee G. Pedersen. PMEMD is implemented in Fortran 90 and MPI. We have experimented with the scaling characteristics of PMEMD available with Amber version 9.0. The scaling results in Figure 1.3 show an improved degree of scaling on the contemporary MPP platforms including the Cray XT systems (XT3 and XT4) and the next generation Blue Gene system, the Blue Gene P (BGP). At the same time however, we note that on a system like Cray XT, which has a contemporary dual-core AMD Opteron processor and a high bandwidth network, parallel efficiencies start degrading on 64-128 processor cores or MPI tasks. Scaling limits are relatively higher on the BGP platform that contains a relatively low frequency processor. Moreover, some end users have reported statistically significant differences in sander and PMEMD results when simulations are run for very long time scales. A number of experiments therefore rely on the sander module in order to maintain consistency between experiments.

A special purpose system, called MDGRAPE, attempts to address memory and network latency issues of MD calculations with specialized execution pipelines for non-bond calculations [41]. Our reconfigurable design scheme addresses a similar issue but in a more cost-effective and flexible manner. In contrast to our approach, MDGRAPE is optimized for a small subset of MD calculations within the AMBER framework and requires end users to understand its customized software infrastructure. Our implementation is flexible, and extensible, and encompasses more MD computation logic.

**Contributions.** In order to address the aforementioned issues regarding parallel efficiencies of MD simulations, we are investigating the acceleration of the PME method on FPGA-enabled supercomputing systems using a high-level language, Fortran—a widely used language for scientific computing. Currently, the SRC MAPstation, Cray XD1, and SGI RASC systems are all available with FPGA devices. The SRC platforms are the only systems that provide a tightly-coupled coherent programming environment allowing users to program the FPGAs using both Fortran and C programming languages [12]. Since the sander module in AMBER is written in Fortran 90, we targeted the SRC-6E MAPstation taking advantage of the Fortran support. The Series E MAPstation pairs a dual 2.8 GHz Xeon microprocessor with a MAP processor consisting of two user-configurable Xilinx$^{X}$C2VP100 FPGA devices [13] running at 100 MHz, a control processor, and seven 4MB SRAM banks referred to as On-Board Memory (OBM).

We characterized the computation and memory requirements of the PME calculations with extensive profiling and benchmarking on existing microprocessors and parallel systems. Due to the logic capacity of the SRC-6E FPGA devices, we accelerated only the direct PME calculations which account for over 80% of the total execution time in most bio-molecular simulations. Initially, using single-precision FP calculations we achieved a computation speedup (not including overheads) of over 3x for two biological systems of sizes 24K and 62K atoms when compared to the same systems running on the microprocessor-only system. Then, after carefully characterizing the memory requirements, we managed to reduce the data transfer overheads and sustain a total application speedup of 3x compared to the microprocessor-only runtimes. Finally, we further increased the application speedup, by overlapping the 'direct' and 'reciprocal' PME calculations [22]. The FPGA devices execute the 'direct' part of the calculation while the 'reciprocal' part is executed by the host microprocessor.

Since the performance of the FPGA devices are increasing at a much faster rate than commodity microprocessors [43, 44] and the SRC-6E contains relatively old FPGA devices (two generations old), we developed a performance model to predict performance for our current implementation on future FPGA devices [17]. These models have been extended to the parallel implementation of sander that employs message-passing (MPI) programming paradigm. We validate the extended model with the MPI profile and timing data collected on the Cray XT3, a high-bandwidth MPP platform. This model is parameterized by the application's input parameters, which determine the problem size, and the target device parameters including the FPGA clock frequency, memory capacity, and I/O bandwidth. Using this performance model, we estimate that the next-generation FPGA devices will provide an additional speedup of greater than 2x beyond our current improvements for overall application performance – a speedup of greater than 15x relative to contemporary microprocessors, with increasing biological system sizes. On parallel systems, we demonstrate that the time to solution on 16-to-32 FPGA-enabled nodes would be equivalent to the largest configuration of contemporary MPP supercomputers.

**Paper Outline.** The layout of the paper is as follows: Section 2 provides a background discussion of FPGA accelerator devices and bio-molecular simulations. In Section 3 we discuss related research efforts for accelerating MD simulations on reconfigurable devices. Section 4 presents the implementation details. Performance analysis of the current implementation on the SRC-6E system is given in section 5. Section 6 describes the performance model and performance projections on current-generation FPGA devices and on future FPGA configurations and biological experiments. Conclusions and future plans are outlined in section 7.

## 2. Background.

**FPGA Accelerator Technologies.** The concept of Reconfigurable Computing (RC) originated in the 1960s in a paper by Gerald Estrin. In his paper he proposed the concept of a computer made of a standard processor and an array of "reconfigurable" hardware. In this paradigm, the main processor would control the behavior of the reconfigurable hardware, which would be tailored to perform a specific task. Once the task was complete, the hardware could be reconfigured to do some other task. The advantage of this hybrid structure lies in the combination of its software-like flexibility with hardware-like speed. It was not until the last decade that the electronic technology matured enough to make these systems, also known as reconfigurable computing

(RC), possible. As shown in Figure 2.1, High-performance Reconfigurable Computing (HPRC) platforms consist of a number of RC nodes connected by some interconnection network (ICN); the computing nodes typically consist of a general-purpose processor coupled to the RC hardware via some communication interface (e.g. PCI, memory bus, rapid array, HyperTransport, etc.).

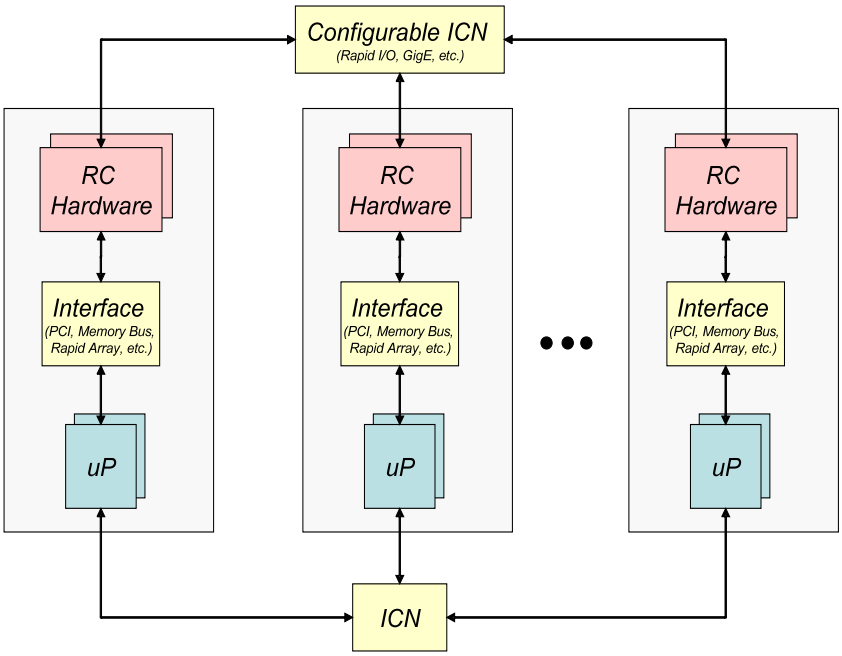



FIG. 2.1. *High-performance Reconfigurable Computer (HPRC) Architecture*

The HPRC platform allows users to exploit fine and coarse grain parallelism within the RC device and across the parallel compute nodes. A variety of RC cards have been developed during the past decade [2, 3, 8, 9, 10, 25, 32, 34]. The early success of some of these offerings paved the way for HPC vendors such as the Cray XD1 [5] and the SGI RASC system [11] to enter the market with HPRC platforms featuring high-end processors more tightly coupled with the reconfigurable devices. Other vendors such as SRC with their line of MAPstations [12] and Opteron socket products from DRC Computers [6] and XtremeData [14] have entered the market with cluster-ready platforms. The systems from these vendors often include a more developed programming environment and clusters can be built incrementally (node by node).

For all the HPRC systems mentioned thus far, the RC element of the system is in the form of an RC board or module and the primary architectural difference is the manner in which it is coupled with the rest of the system. Each of the above-mentioned platforms has a different communication interface between the general-purpose processors and the reconfigurable hardware devices in addition to other variations including the host processor, memory hierarchy, FPGA devices, clock frequency, etc.

**Biomolecular Simulations.** Numerous applications use MD for biomolecular simulations. These applications include AMBER [1], GROMACS [7], LAMMPS [36], and NAMD [29]. MD and related techniques can be defined as computer simulation methodology where the time evolution of a set of interacting particles is modeled by integrating the equation of motion. The underlying MD technique is based upon the law of classical mechanics, and most notably Newton's law, $F = ma$ [31].The MD steps performed in AMBER consist of three calculations: determining the energy of a system and the forces on atoms centers, moving the atoms according to the forces, and adjusting temperature and pressure. Most MD models treat atoms classically as points with mass and charge. The atomic points interact with other atomic points through pair-wise interactions from chemical bonds, electrostatic interactions and van der Waals interactions.

A typical bimolecular simulation contains atoms for solute, ions, and solvent molecules. The force on each atom is represented as the combination of the contribution from forces due to atoms that are chemically bonded to it and non-bond forces due to all other atoms. The simplified overall energy equation is:

$$E(potential) = \sum f(bonds) + \sum f(angles) + \sum f(d-torsions) + \sum_{j=1}^{N}\sum_{i=1}^{N}(-A_{ij}/r_{ij}^{6}) + \sum_{j=1}^{N}\sum_{i=1}^{N}(q_i q_j / r_{ij})$$

where the first three terms are the bonded terms and the latter two are referred as non-bonded terms. The non-bond energy is broken into two contributions: van der Waals and electrostatic interactions. The number of bonds, bond angles, and bond dihedrals during the classical simulations are kept constants. For a medium system, there are only a few thousand bonds and angles compared to millions of the non-bonded interactions; the calculations involving the bonded terms are extremely fast on conventional systems. The double sum of the non-bond terms makes the number of these calculations scale with an order of $N^2$, where $N$ is the number of atoms. Simulations of larger systems (larger $N$) are therefore extremely expensive.

In the case of *periodic boundary conditions* (PCB), where the system is treated with a periodic arrangement of repeated unit cells to model the effect of large surrounding solvent without increasing the number of particles, the non-bonded sums become large. In a finite PCB, the simulation box is replicated a fix number of times in all directions to form a lattice (Figure 2.2). In practice, MD simulations evaluate potentials using a *cutoff* distance scheme for computational efficiency, where each particle interacts with the nearest other $N-1$ particles in a sphere of radius cutoff. Figure 2.2 shows a 2-dimentional view of a simulation cell replicated in the three directions of space; atoms within cutoff will be involved in computing the non-bonded interactions.

Atom pairs that are separated by a distance greater than the cutoff limit (typically 10 Angstroms for AMBER simulations) are not included in the sums. The cutoff limits the number of non-bond interactions in the sum to be $N * (\text{number of atoms in the cutoff sphere})$ as compared to $N * (N-1)$ interactions without the cutoff. For the van der Waals interactions, the cutoff error is small, but the electrostatic sum has a very large error, 10% or greater, when a 10 Angstroms cutoff is introduced. Figure 2.3 illustrates the magnitude of the problem for a system.

The total non-bond energy, the sum of the van der Waals and the electrostatic energies are of comparable magnitude near the equilibrium distance. But at 10 Angstroms, the electrostatic are still strong while the van der Waals energy is essentially vanished. Ignoring the electrostatic interactions that are beyond 10 Angstroms can introduce a large error in the energy and forces resulting in artificial force magnitudes. The Particle Mesh-Ewald (PME) method provides a solution to this problem by solving all electrostatic forces; it uses an atom-based cutoff [22, 23] reducing the number of non-bonded interactions to $N \log(N)$.

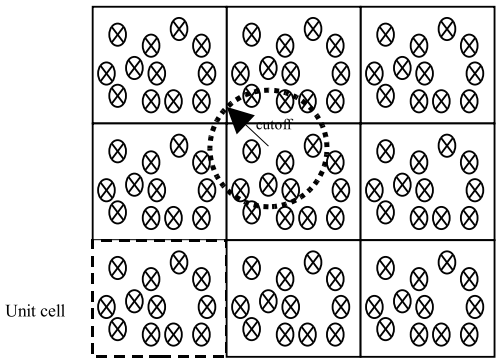



FIG. 2.2. *Two-dimensional view of a simulation cell replicated in the three-dimension space.*

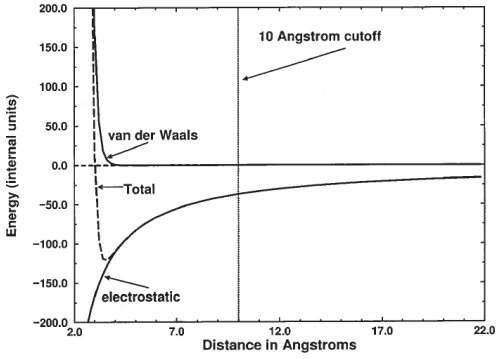



FIG. 2.3. *The van der Waals and electrostatic contributions to the non-bond energy are shown as a function of the inter atomic distance [22].*

The Ewald method expands the simple sum of Coulomb's Law (electrostatic) terms into the following terms:

$$E(electrostatic) = E(direct) + E(reciprocal) + E(correct)$$

Except for the error correction function, the direct sum is identical to the sum in the cutoff method that calculates electrostatic potential energy. The reciprocal sum is a major part of the electrostatic energy that

TABLE 2.1
*Time spent in the direct and reciprocal Ewald calculations as a percentage of total execution time.*

| Number of atoms | Direct Ewald Time (%) | Reciprocal Ewald Time (%) |
|---|---|---|
| 23558 | 82.61 | 16.66 |
| 61641 | 86.88 | 12.56 |
| 143784 | 87.12 | 12.34 |
| 194262 | 86.47 | 12.92 |

the direct sum misses due to the correction factor. The reciprocal sum is approximated using Fast Fourier Transform (FFT) with convolutions on the grid where charges are interpolated on the grid points. Table 2.1 provides the percentage of execution time (for 10000 time steps or production-level simulations) for four different protein experiments on an Intel dual 2.8 GHz Xeon system. The direct sum accounts for over 80% of execution time. The reciprocal Ewald calculation takes less than 13% of the total execution time. Taken together, these calculations account for over 95% of total execution time on a single processor system.

**3. Related Work.** A number of related efforts to develop MD codes on reconfigurable hardware platforms have been reported in the literature. In [26], the authors implement a basic MD system focused on the motion updates and the $O(N^2)$ force terms (both Coulombic and L-J forces and multiple atom types) using hardware design techniques. The authors study the relationship between precision and quality of MD simulations and report that it is possible with reconfigurable devices to trade off unneeded precision for computing resources. Implementation on a COTS system yielded accelerations ranging from 31x to 88x with respect to a microprocessor-only implementation, depending on the size of the FPGA and the simulation accuracy. Similarly, in [19], the authors implemented a novel single atom type MD system with VHDL on a Transmogrifier 3 (TM3) system. This implementation focuses on the L-J force calculator with problem specific implementations. To reduce complexity, the implementation uses fixed-point representations varying between 22 and 76 bits for all values within the MD system. The author's results show that this implementation closely tracked the higher precision software implementation with an error of less than 1% between consecutive time steps. The authors extrapolate that with better FPGA memory organization and faster FPGAs, a speedup of 40x to 100x over a microprocessor implementation can be achieved.

In [38], the authors use the SRC development suite Carte® to implement a tightly coupled MD simulation kernel (not a complete MD software package such as AMBER) on the SRC-6E MAPstation. Like our approach, the important tasks of an MD simulation are analyzed and partitioned such that the most intensive are executed in the reconfigurable hardware and the rest are executed on the general-purpose processor. Even though only a portion of the simulation is accelerated in the MAP, the single-precision implementation achieves a 2x speedup over the software baseline running on the MAPstation host.

Neither [19, 6], nor [38] are concerned with the problem of accelerating existing, production-level, MD simulation software nor have they been tested with more than a few thousand particles. The most closely related work comes from [30] where the authors implement a simplified version of an MD algorithm in NAMD [29], an MD simulation package, on the SRC-6 MAPstation. While their implementation does trace the steps involved in porting a large-scale scientific code to FPGA-enabled systems, they do not cover memory analysis and characterization or the performance modeling work presented in this article.

**4. Implementation.** Based on the percentages of execution times, the *direct* PME calculations are a candidate for the FPGA acceleration since an acceleration of this method is likely to result in an overall application speedup or a reduction in the time-to-solution. Figure 4.1 shows the call tree for the direct Ewald calculation, which is composed of two calculations: coordinate mapping and non-bond energy calculations. The coordinate mapping calculations (f1 and f2) are also invoked from the *reciprocal* Ewald calculations; therefore, we only map the non-bond energy calculation function (f3) within the *direct* PME calculations onto the FPGA.

Since the logic capacities of the FPGA devices are limited for floating-point calculations, we initially port only the most expensive (time-consuming) parts of the calculations into the FPGA. Table 4.1 lists the percentage of execution time for the functions listed in Figure 4.1. Using the gprof runtime profiling tool, we calculated the contributions of the individual functions. Functions f1, f2 and f3 are called once every time step iteration

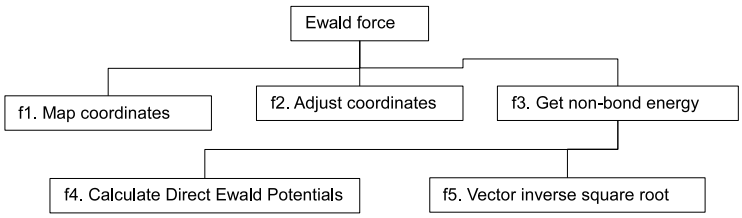



FIG. 4.1. *Call tree for the direct Ewald calculations in sander. Calls to functions f1, f2, and f3 are made once every time step; calls to f4 depend on the number of atoms in the system; f5 is called twice as many times as f4.*

but the number of calls to f4 depends on the number of atoms in the system. For instance, when the number of atoms is 143784, f1, f2 and f3 are called once but f4 is called 143784 times and f5 is called 143784*2 times. This knowledge about the number of invocations plays a very important role in the decision making process, because there is a substantial call overhead ($\sim 0.3$ sec on SRC-6E) involved in calling a function that is mapped on the FPGA devices from the host processor.

TABLE 4.1
*Contribution of individual functions in the direct PME execution times.*

| Number of atoms | 23558 | 61641 | 143784 | 194262 |
|---|---|---|---|---|
| f1 (% of total) | 0.1% | 0.1% | 0.2% | 0.2% |
| f2 (% of total) | 0.4% | 0.5% | 0.5% | 0.5% |
| f3 (% of total) | 82.1% | 86.2% | 86.4% | 85.8% |
| f4 (% of f3) | 86.1% | 85.9% | 88.4% | 88.4% |
| f5 (% of f4) | 18.8% | 20.9% | 17.9% | 17.9% |

After analyzing the call tree and contributions of individual functions in the direct Ewald calculations, we decided to map the branch functions f3, f4 and f5, onto the FPGA devices. First, we analyzed the loop structure within each of the three functions (Figure 4.1 and Figure 4.2). Function f4 has two nested loops that iterate through all atoms in the system. The outermost loop has a fixed count, which depends on the unit cell grid dimensions (a unit cell grid is shown in Figure 4.3), while the two inner loops are calculated at runtime.

The size of the unit cell grid depends on a number of factors including the size of the protein, number of atoms, density and types of atoms. The unit cell is divided in subcells, and each subcell contains a different number of atoms. Note that biological systems do not have a uniform density. Hence, the subcell iterations depend on the number of atoms in the currently selected subcell, which can range from 0 to a maximum density. Furthermore, the number of atoms in a subcell changes as the simulation progresses because the positions of atoms are not fixed.

```
Function f3
  Do loop1=1, indexhi  (fixed)
    Do loop2=1, subcell(loop1) (variable)
      Call f4
Function f4
  Do loop3=1, icount (variable)
```

FIG. 4.2. *A pseudo-code for the three nested loop in the direct PME calculations.*

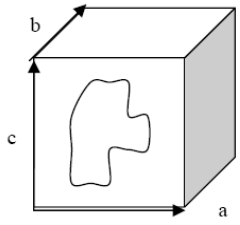



FIG. 4.3. *A schematic representation of a biomolecule in a unit cell.*

In addition to the two nested loops of function f3, function f4 contains several small loops with similar loop index values. Like the loop index of the inner loop in function f3, the loop index in function f4 is determined at runtime. It depends on the number of neighbor atoms a given atom interacts with inside the cutoff limit. This value in turn depends on the number of atoms in the system, density of the system and the cutoff limit.

Nevertheless, this value is not constant for individual atoms since the atoms move around during the simulation. A pair list, that contains pairing information of individual atoms with all other atoms in the system is, therefore, updated throughout the simulation run.

**Application of the Performance Enhancing Features.** Deep pipelining, concurrent execution capabilities and data streaming are the main performance advantages of FPGA devices for computing applications. These features are exploited during implementation to achieve a higher speedup for the direct Ewald calculation. Since our FPGA implementation is developed using Fortran, the three loops mentioned earlier are not different from the original implementation. The nested calls to functions f3, f4 and f5 in the original code are replaced by a single invocation to a SRC MAP (FPGA implementation) function that performs the calculations of the three functions on the FPGA devices.

The only differences between the original and FPGA implementation are the additional calls for data transfers between the host processor and the on-board memory of the MAP and the FPGA-specific constructs for parallel execution of the code blocks. A schematic of the process is shown in Figure 4.4. The control lines are shown in arrows (from source to target) and the data lines are shown in dotted lines. The host processor oversees the control and data movement between the host and the FPGA devices. However, once the devices are setup, the primary FPGA manages the DMA operations and the data transfers between the FPGAs. Note that all control and data transfer calls shown in Figure 4.4 can be active simultaneously. In addition, the SRC systems have multiple data ports; for instance, three 64-bit elements can be transferred between the two FPGAs in a single clock cycle.

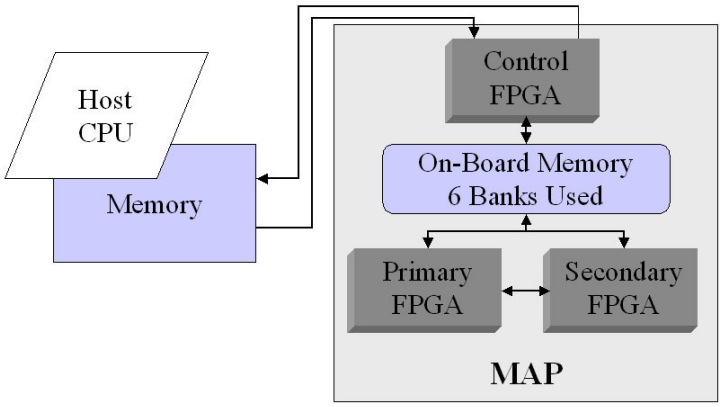



FIG. 4.4. *Control and data paths between the host and FPGA devices.*

Deep pipelining allows a user to describe the parallelism in terms of a producer-consumer programming paradigm. A producer-consumer relationship can be between: (1) host and primary FPGA; (2) primary and secondary FPGA; and (3) parallel sections within an FPGA. A 'parallel section' construct in the SRC programming permits task parallelism, i. e., multiple computation and data transfer tasks executing simultaneously. Typically, streaming data is transferred between the producer and consumer devices. A deep pipelining example may include the following tasks executing in parallel:

| | |
|---|---|
| 1. DMA in a large array a1<br>2. DMA in two small arrays b1 and c1<br>3. Setup two communication ports with the secondary FPGA<br>4. Send a1 to the secondary FPGA<br>5. Send c1 to the secondary FPGA<br>6. Compute within loop using b1 and a1, one element at a time | 7. Setup two communication ports with the primary FPGA<br>8. Receive a1<br>9. Receive c1<br>10. Continue the innermost loop of the primary FPGA and compute result c1 |
| Parallel section (Primary FPGA) | Parallel section (Secondary FPGA) |

The data transfer overheads and latencies in most cases, therefore, can be concealed using the deep pipelining and streaming techniques as long as there is sufficient work (computation) available to hide these latencies. The performance of the above pipeline will be the latency of the longest parallel section. Every effort is made to remove or reduce pipeline stalls, if the producer stream data element is not ready, the consumer stalls; likewise, if the consumer is not ready to accept the incoming data, the producer stalls. Additional performance improvements can be achieved with traditional loop optimization techniques like loop unrolling and flattening.

**Algorithm Mapping onto the FPGA.** To map the direct PME algorithm onto the FPGA devices, we characterized the data and control requirements of the algorithm implementation in the AMBER framework. For data requirements, we identified the local and non-local data elements, particularly the large arrays in the functions that are to be mapped onto the FPGA devices. These arrays include those containing the complete cell image coordinates, the force coordinates, pair information, Ewald tables, and indices to non-bond interactions. The sizes of the arrays depend on a number of parameters, primarily the number of atoms and dimensions of the unit cell grid. For example, for the 23558 atoms experiment, the indices arrays contain over 30K, floating-point data elements.

For control operations, we identified the loops that are potential targets for exploiting parallelism. The outermost PME loop has a fixed (constant) index, which depends on the dimension of the unit cell grid. The indices of the two nested loops within the outermost loop are determined at runtime. In addition, there are some smaller loops with a few tens of iterations that determine the runtime indices of the innermost loop. The calculations in the innermost loop are distributed between the two FPGA devices to exploit the concurrency that the two FPGAs offer. Moreover, the innermost loop is flattened creating the longest calculation in the pipeline and maximizing the throughput. We anticipate that the outermost loops can be flattened on FPGAs with larger logic capacities.

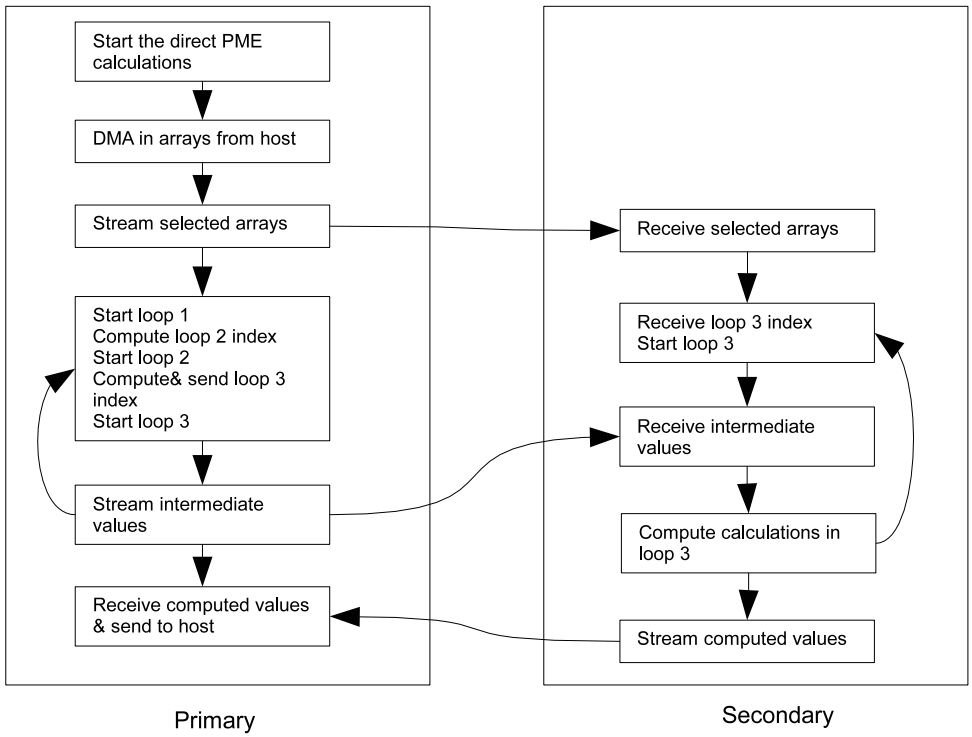



FIG. 4.5. *Control flow of the direct PME calculation on the two FPGA devices.*

Figure 4.5 shows a flow chart of the direct PME calculations on the two FPGA devices. After invoking the FPGA-resident function, the primary FPGA DMAs the data from the host processor. It then streams in the large arrays; a subset of which are concurrently streamed to the secondary FPGA over the bridge ports. Then the outermost loop starts on the primary FPGA, which computes and subsequently starts loop 2 in the

primary FPGA. After computing and transferring the index of loop 3 to the secondary FPGA, partial nested loop calculations are performed on the primary FPGA. Since the bridge ports are already configured with the secondary FPGA, as soon as the data is generated on the first FPGA it is available to the second FPGA where the streaming calculations continue. The gray area in the diagram shows the deep pipeline that spans across two FPGA devices. The $3 \times 3$ virial arrays computed on the secondary FPGA are stored in intermediate arrays and are transferred to the primary FPGA after the three loops have concluded. The primary FPGA then makes this data available to the host processor.

**5. Performance Analysis.** We use two metrics to compare performance of the FPGA-accelerated code with the microprocessor-only execution times. The first metric is traditional speedup:

$$\text{Speedup} = \text{Runtime}_{\text{microprocessor}}/\text{Runtime}_{\text{FPGA-accelerated code}}$$

The second metric (used by the application scientists) is the science-based metric: pico-seconds ($10^{-12}$) per simulation day (psec/day). This metric determines the time to simulate a biological system at the required time scales and is useful when comparing simulations of various sizes across different computing systems.

We measured the performance of the FPGA-accelerated code for two test cases namely *jac* (joint amber-charmm) and *HhaI*. The jac benchmark is part of the AMBER version 8.0 release and it contains 23558 atoms. HhaI is a protein-DNA system that contains 61641 atoms. The microprocessor-based performance is measured on the SRC host processor system, which is an Intel dual 2.8 GHz Xeon system. The SRC-6E FPGA devices run at 100 MHz, a clock frequency restriction imposed by the SRC system (current-generation FPGA devices have more logic and memory capacity and are capable of operating at higher frequencies). Nevertheless, we demonstrate the potential for FPGA acceleration for an important class of applications on these devices. The results from the FPGA-accelerated code are compared with the microprocessor-only implementation for both performance and accuracy to verify the correctness of the design.

To analyze the performance behavior, we used the SRC-6E performance analysis and debugging tools to measure the runtime contributions of key sections of the accelerated code. Runtimes for three sections are measured separately: (1) time to setup the MAP (calling overhead); (2) compute time; and (3) data transfer times. The time to setup the MAP has an additional cost ($\sim$0.3 milliseconds) for the first invocation; in subsequent invocations this cost is comparable to regular Fortran function calls. The data transfer time includes the time to receive data from the host and to send results back to the host. Compute time is the computation time spent on the two FPGA devices including the time to transfer data between the two devices. As expected, the data transfer overheads offset the performance gains in a naïve implementation and the penalties are higher for the larger problem, HhaI. The compute only speedup increases with the problem size or the number of atoms, 3.3x and $\sim$4x, respectively. However, since the data transfer overheads also increase with the problem size, the overall application speedup is reduced to less than one. At this stage, we concluded that the memory access requirements needed further characterization in order to achieve sustained performance on the FPGA devices.

TABLE 5.1
*Speedup of the PME calculations before and after memory characterization over the 2.8 GHz host microprocessor system*

| | jac (23558 atoms) | | HhaI (61641 atoms) | |
|---|---|---|---|---|
| | Speedup (initial implementation) | Speedup (after memory characterization) | Speedup (initial implementation) | Speedup (after memory characterization) |
| Computation only | 3.3 | 3.3 | 3.97 | 3.97 |
| Setup+compute | 3.29 | 3.29 | 3.96 | 3.96 |
| Compute+data transfer | 0.64 | 3.21 | 0.69 | 3.83 |
| Overall | 0.6 | 3.19 | 0.6 | 3.82 |

We considered and evaluated a number of techniques to reduce the data transfer times. First, data can be pre-fetched and post-stored to hide the data transfer latencies. Additionally, pthreads or OpenMP multi-threading techniques allow the transfer of large arrays while the compute thread is processing and before the accelerated function is invoked. Second, data transfers to the FPGA can be pipelined and overlapped

using the streaming directives provided in the SRC programming environment. Although these are partially done in the first implementation, carefully overlapping and pipelining additional data transfers can further improve performance. Finally, algorithm-specific optimizations are exploited by characterizing the memory access behavior and patterns in the accelerated code. We employed the last approach since it also leverages the other optimization techniques.

Implementation of the accelerated PME calculation is further modified according to memory access classification and characterization. This implementation, however, does not include any modification to the AMBER source code on the host to exploit additional benefits from multithreading with *pthreads* or *OpenMP*. Only the Fortran code for the FPGA-accelerated calculations is modified to reduce the unnecessary data transfer overheads. Amazingly, the modified code resulted in a very significant reduction in the data transfer costs; the data transfer costs that previously accounted for over 70% of the total execution time, are now less that 5% of the total execution time resulting in sustained accelerated performance with the FPGA devices. Table 5.1 summarizes the performance improvements for jac and HhaI experiments.The time-to-solution metric is calculated for a nano-second scale simulation ($10^6$ time steps) and is presented for the *jac* benchmark in Table 5.2. We also measure and include time for the non-accelerated calculations; a constant for both host processor and FPGA-accelerated code because they are always executed on the host processor. We calculate the performance improvement achieved by overlapping the 'direct' and 'reciprocal' PME calculations on FPGA and host respectively (OpenMP constructs in the PME-AMBER source code enable the overlap).

TABLE 5.2
*Time-to-solution for the SRC 6E FPGA-accelerated code for the jac benchmark*

| | Time-to-solution (after memory characterization) | Time-to-solution (after overlapping) |
|---|---|---|
| Computation only | 4793 days | 3208 days |
| Setup+compute | 4801 days | 3214 days |
| Compute+data transfer | 4868 days | 3282 days |
| Overall | 4876 days | 3290 days |
| Host | 10417 days | |

From the values in Table 4, we estimate time for a nano-second scale simulation instead of our target micro-second scale simulation. A nano-second simulation will take over 10 days on the microprocessor system with dual 2.8 GHz Xeon system, about 5 days on an FPGA accelerated code, and just over 3 days by overlapping FPGA and host execution. (Note: these computations are for the older-generation FPGAs on the SRC-6E.)

**6. Modeling and Projections.** In order to analyze the performance of the current system and predict the performance potential of future FPGA-enabled systems, we developed parameterized performance models of our current FPGA implementation. The models are parameterized with application parameters and system parameters allowing for the analysis of a variety of FPGA systems as well as larger biological simulations. The application parameters include the number of atoms, dimensions of the box, types of atoms, and number of residues. From the application parameters, we can generate the size of data transfers, physical memory requirements, and loop indices for the main computation loops. The FPGA system parameters are the FPGA clock frequency, bandwidth to the host processor, and bandwidth between the FPGA devices. Our modeling scheme is conservative because we do not take into account the characteristics of future FPGA devices that contain special features for double-precision floating-point calculations and logic capacities. Furthermore, we do not consider the performance advantages that can be achieved by flattening the three direct PME loops, which we anticipate will also be possible on larger FPGAs.

For simplicity, we consider a cubic box in which all three dimensions are equal, i. e. $a = b = c$. Moreover, we consider a NTYPE (types of atoms) to be a constant (= 20), and the numbers of residues are fixed as (NATOMS/3.25). The problem size is therefore controlled by the NATOMS parameter. The performance model has two elements: computation cost and data transfer overheads. There is a fixed, single time startup-overhead cost not included in the model because these biological simulations are expected to run tens of thousands of time steps. Hence, the single startup-overhead cost is amortized for these simulation runs. In the computation time, represented below, we take into account the latency (in clock cycles to perform sequence of serialized operations within a loop) for the three main loops that are shown in Figure 8. In addition, we include the loop

counts (variable and fixed) that are calculated from the application input parameters. Similarly, we calculate the number of data transfer bytes using the input parameters and apply the available memory bandwidth to determine the data transfer times. Since the sizes of data transfers do not depend on runtime values, the data transfer overheads are precisely measured for our current implementation. The workload requirement model and the runtime performance projection model are validated with the current SRC-6E implementations.

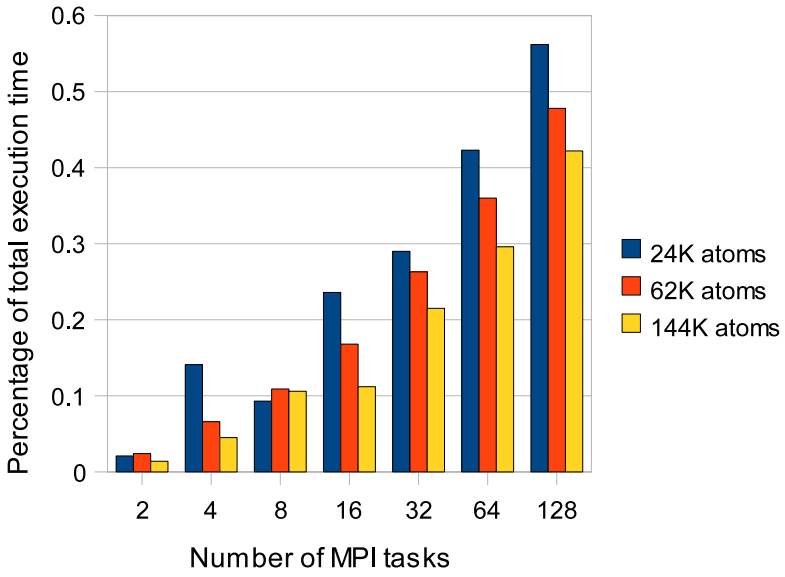



FIG. 6.1. *Figure 12: Percentage of total execution time spent in MPI communication routines.*

To develop a model for the parallel version of the application, we profiled MPI message sizes and timings on the Cray XT3 system, which has a high communication bandwidth compared to contemporary cluster systems. Figure 6.1 shows the fraction of total execution time spent in MPI communications for three test cases. We notice that the percentage of communication time increases with the number of parallel MPI tasks, and beyond 128 tasks it accounts for over 50% of total execution time. Although the fraction of MPI time decreases with the increase in the number of atoms, the reduction rate is much slower than the increase in number of atoms.

In addition to aggregate data collection, we investigated the runtime behavior of MPI messages and sizes. We note that the number and sizes of communication operations per simulation time step per processor is not constant. Also, AMBER does not use an MPI Cartesian topology; moreover, sizes for the MPI collective operations per processor do not change with the problem size or number of atoms. AMBER has a collection of programs that can run different MD simulations. In this study, we focused on the most widely used simulation method—explicit solvent simulations in sander.

We developed symbolic models for the communication phases with a fixed message size—an average of the smallest and largest message sizes exchanged in these phases. The largest message size does not change with processor count, while the smallest one scales linearly. These sizes are validated at runtime using the MPI profiling tool. The MPI_Allreduce message sizes do not depend on the problem size, therefore, the collective message volume grows linearly with the problem size.

Our findings about the scaling behavior of the explicit solvent calculations in AMBER explain the performance results presented in earlier studies [16, 18]. Scaling results for the explicit solvent method in AMBER on parallel systems do not scale beyond 128 processors—not even on systems with very high bandwidth interconnects, like IBM Blue Gene/L [35] and Cray XT3 [5]. On SMP cluster systems, these calculations only scale to up to 16–32 processors. Using symbolic models, we quantified the growth rate in volume and distribution of MPI messages, which enable us to identify that the force sum, and coordinate distribution phases of calculation limit application scaling. These factors in turn limit the scaling beyond 64-128 MPI tasks on distributed memory parallel systems.

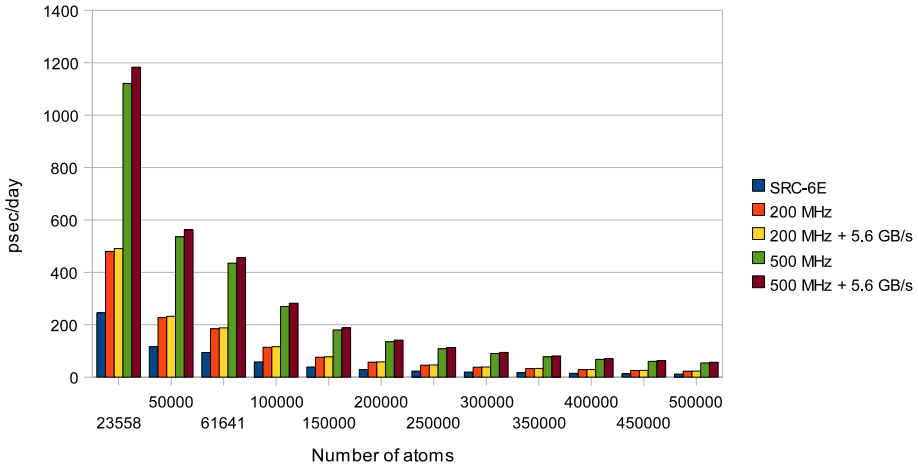



FIG. 6.2. *Performance projections with varying FPGA performance metrics.*

**Projections on Future FPGA Devices.** We use our validated performance models to carry out a number of performance projection experiments. Figure 6.2 shows the performance projection results on a single-node FPGA accelerated system. We altered two FPGA-enabled system parameters for our experiments: the clock frequency and data bandwidth between the FPGA device and the host processor. The clock frequency of our current FPGA implementation is 100 MHz and the sustained payload bandwidth is 2.8 GB/s (utilizing input and output 1.4 GB/s bandwidth). The clock speed and data transfer rates have different performance implications on small and large biological systems (Figure 6.2). Overall, the clock speed influences the performance gains of the PME calculations. For smaller biological systems, the change in the data transfer rates influences the runtime performance of the application. By contrast, the performance of the larger systems (100K and more atoms) nearly doubles by doubling the clock speed of the FPGA devices, while the data transfer rates alone do not significantly impact the runtime performance. Note that a ∼150K atoms system only achieves ∼12 psec/day on a dual 2.8 GHz Xeon system today. An FPGA-enabled system using our current PME implementation can sustain over 75 psec/day with a 200 MHz FPGA and over 180 psec/day with a 500 MHz FPGA and a host bandwidth of 5.6 GB/s.

**Parallel Efficiency.** Based on the parameterized model of our FPGA implementation and a detailed analysis and modeling of MPI implementation of sander, we carry out performance projection studies on parallel systems with multiple FPGA accelerator devices. The FPGA-accelerated implementation of direct PME does not involve MPI communication; therefore, we can port the accelerated code without any modification to a parallel platform. However, we cannot take into account the contribution of overlapping the reciprocal and direct Ewald calculations because the reciprocal calculations *do* involve MPI communication. Hence, we consider a blocking implementation of the accelerated code; in other words, our performance estimates are highly conservative since the overlapping of calculations can result in significant performance benefits not only on small processor counts but also on 64–128 processor runs. We anticipate that simulations with 32 or more processors, the reciprocal Ewald calculation will contribute to a larger fraction of the runtime and the accelerated direct PME will have negligible runtime contributions.

Due to the inherent scaling limit of the sander implementation [16, 22], we target a cluster with 8–16 processors, each populated with an FPGA device. Note that this configuration is different from the existing Cray XD1 having six FPGA devices connected to a chassis and these devices communicate with one another only through their host Opteron processor. The Maxwell system recently developed at the Edinburgh Parallel Computing Center is similar to our target configuration. The system consists of a 32-way IBM BladeCentre chassis hosting 64 Xilinx Virtex-4 FPGAs directly connected over high-speed RocketIO. This allows codes to be parallelized across the collection of FPGAs and encourages algorithms to be written such that once the data and program are loaded onto the accelerator cards the processing occurs without transferring data across the PCI-X bus again.

We project runtime on a FPGA accelerated parallel system using the following expression:

$$\text{Computation}_{\text{time}} = l1_{\text{count}} * (\text{latency}_{l1} + l2_{\text{count}} * (\text{latency}_{l2} + \text{latency}_{l3} + l3_{\text{count}})/\text{clock}_{\text{frequency}}.$$

Time on the host is the time that is not spent in communication and overlapped calculations. Considering a high-bandwidth communication network similar to Cray XT3 distributed memory system [5], we project performance in terms of psec/day for the 3 test systems with approximately 24K, 62K and 144K atoms respectively. We compare these results with the runtime measured on the XT3 system. For parallel performance projections, we conservatively targeted an FPGA with double the clock frequency, 200MHz, of the SRC-6E 100MHz. Currently available FPGA devices are capable of operating at clock speeds higher than 200 MHz although they are only now becoming available in leading edge reconfigurable computing platforms. In addition, we consider no improvement in the FPGA and host bandwidth, which is expected to be significantly higher for new production and future systems especially those connected via HyperTransport links. Figure 14 and Figure 15 show the throughput of an FPGA-accelerated system and the Cray XT3 in terms of psec/day. We note that speedup is sustained for simulation runs up to 16 MPI tasks in the FPGA accelerated system. For the 62K atoms experiment, a 32-processor FPGA-accelerated system could be as efficient as the largest configuration of the Cray XT3 and time to solution on a 2-node FPGA-accelerated system could be equivalent to large-scale Blue Gene L configurations.

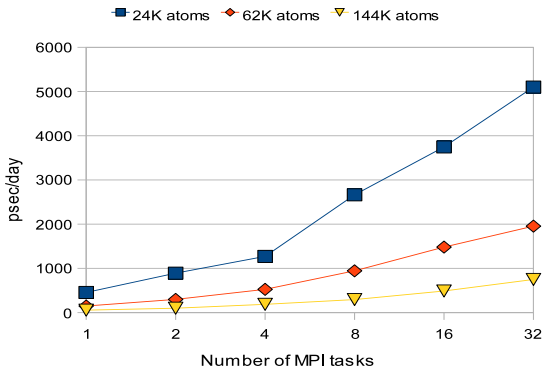



FIG. 7.1. *Parallel efficiency on the FPGA-accelerated system*

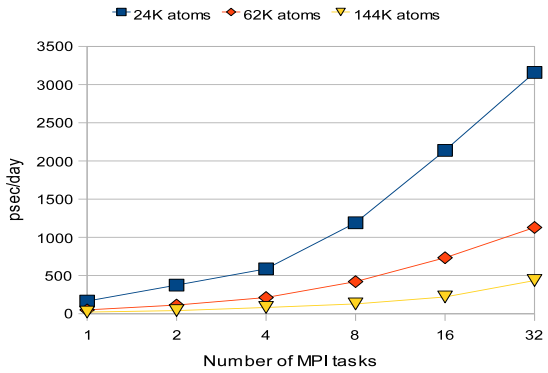



FIG. 7.2. *Parallel Efficiency on the Cray XT3 system*

**7. Conclusions and Future Plans.** We have demonstrated that production FPGA-enabled systems can achieve sustained application speedup for a production-level scientific simulation framework, and that the co-processor accelerated systems with few tens of processing units can surpass performance of Teraflops-scale supercomputing systems. Using our task-based implementation approach, scientific application developers can exploit extremely powerful yet flexible devices to perform a diverse range of scientific calculations by using a familiar high-level programming interface all without significantly compromising achievable performance. Our results for the direct PME method show that the time-to-solution of medium-scale biological system simulations are reduced by a factor of 3x on an SRC-6E MAPstation, which contains two-generation old FPGA devices, compared to the microprocessor-only runtimes. Trends indicate that the capabilities of FPGA devices are growing at a faster rate than those of microprocessors and parallel systems have been developed and deployed with co-processor accelerators. Using accurate models of our current implementation and communication overheads measured on a contemporary high-bandwidth supercomputer, we estimate that systems with later generation FPGA devices will reduce the time-to-solution by a factor greater than 15x for large-scale biological systems—a speedup that is greater than currently available on many contemporary parallel cluster systems. Furthermore, for applications with inherent scaling limits, a small-scale parallel system with co-processor accelerators could attain performance of a high-end supercomputing system. Since these reconfigurable devices offer an ideal combination of performance, concurrency, and flexibility for a diverse range of numerical algorithms, we anticipate that many scientific applications will dramatically benefit from the increased support for double-precision floating-point operations and HLL languages now available for these reconfigurable systems. In future work we plan to implement other production-level applications and conduct similar modeling and analysis efforts. Furthermore, we plan to enhance the models to include characteristics of the FPGAs such as their logic capacities

which are not accounted for in the current model. As new generations of these RC systems become available, we plan to collect performance data and validate our models in single-node and multi-node RC systems.

## REFERENCES


[1] Amber home page http://amber.scripps.edu/
[2] Annapolis Microsystems, http://www.annapmicro.com, 2001.
[3] Celoxica, Inc., http://www.celoxica.com/
[4] K. Compton and S. Hauck, *Reconfigurable Computing: A Survey of Systems and Software, ACM Comput. Surv.*, Vol. 34:2, pp. 171–210, June 2002.
[5] Cray Inc., http://www.cray.com/
[6] DRC Computer Corp., http://www.drccomp.com/
[7] GRMOACS, http://www.gromacs.org
[8] I. S. I. East, SLAAC: System-Level Applications of Adaptive Computing, http://slaac.east.isi.edu/ 2003.
[9] Nallatech FPGA-Centric Systems & Design Services, http://www.nallatech.com/ 2002.
[10] Virtual Computer Corporation, http://www.vcc.com/index.html 2002.
[11] SGI Inc., http://www.sgi.com
[12] SRC Computers, Inc., http://www.srccomp.com
[13] Xilinx Inc., Virtex-II Platform FPGAs: Complete Data Sheet, June 2004.
[14] XtremeData Inc., http://www.xtremedatainc.com
[15] P. K. Agarwal, *Enzymes: An integrated view of structure, dynamics and function,* Microbial Cell Factories, 5:2, 2006.
[16] S. R. Alam, P. K. Agarwal, J. S. Vetter and Al Geist, *Performance Characterization of Molecular Dynamics Techniques for Biomolecular Simulations,* Proc. Principles and Practices of Parallel Programming (PPoPP), 2006.
[17] S. R. Alam, P. K. Agarwal, D. Caliga, M. C. Smith and J. S. Vetter, *Using FPGA devices to Accelerate Biomolecular Simulations,* IEEE Computer, Vol. 40. No. 3, 2007.
[18] S. R. Alam and P. K. Agarwal, *On the Path to Enable Multi-scale Biomolecular Simulations on PetaFLOPS Supercomputer with Multi-core Processors,* IEEE Int. Workshop on High Performance Computational Biology, 2007.
[19] N. Azizi, I. Kuon, A. Egier, A. Darabiha, and P. Chow, *Reconfigurable Molecular Dynamics Simulator,* IEEE Symposium on Field-Programmable Custom Computing Machines, 2004.
[20] B. R. Brooks, R. E. Bruccoleri, B. D. Olafson, D. J. States, S. Swaminathan, and M. Karplus *CHARMM: A program for macromolecular energy, minimization, and dynamics calculations,* Journal of Computational Chemistry, 1983.
[21] D. A. Case, T. E. Cheatham, T. A. Darden, H. Gohlke, R. Luo, K. M. Merz, A. Onufriev, C. Simmerling, B. Wang and R. J. Woods, *The Amber Biomolecular Simulation Programs,* Journal of Comp. Chemistry: 1668-1688, 2005.
[22] M. Crowley, T. A. Darden, T. E. Cheatham, and D. W. Deerfield, *Adventures in Improving the Scaling and Accuracy of Parallel Molecular Dynamics Program,* Journal of Supercomputing, 11, 1997.
[23] T. Darden, D. York and L. Pederson, *Particle Mesh Ewald: an Nlog(N) method for Ewald sums in large systems,* J. Chem. Phys. 98, 1993.
[24] H. ElGindy and Y. Shue, *On Sparse Matrix-vector Multiplication with FPGA-based System,* Proc. 10th Annual IEEE Symp. on Field-Programmable Custom Computing Machines, 2002.
[25] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. R. Taylor, PipeRench: A Reconfigurable Architecture and Compiler, IEEE Computer, pp. 70–77, Vol. 33, No. 4, Apr.2000.
[26] K. S. Hemmert and K. D. Underwood, *An Analysis of the Double-Precision Floating-Point FFT on FPGAs,* IEEE Symposium on Field Programmable Custom Computing Machines (FCCM05), 2005.
[27] M. C. Herbort, T. VanCourt, Y. Gu, B. Sukhwami, Al Conti, J. Model and D. DiSabello, *Achieving High Performance with FPGA-Based Computing,* IEEE Computer, Vol. 40. No. 3, 2007.
[28] L. Kale, R. Skeel, M. Bhandarkar, R. Brunner, A. Gursoy, J. Phillips, A. Shinozaki, K. Varadarajan, and K. Schulten, *NAMD2 : Greater scalability for parallel molecular dynamics,* Journal of Comp. Physics, 151, 1999.
[29] V. Kindratenko and D. Pointer, *A case study in porting a production scientific supercomputing application to a reconfigurable computer,* IEEE Symposium on Field-Programmable Custom Computing Machines, 2006.
[30] A. R. Leach, *Molecular Modeling: Principles and Applications,* Prentice Hall, 2001.
[31] P. H. W. Leong, M. P. Leong, O. Y. H Cheung, T. Tung, C. M. Kwok, M. Y. Wong, and K. H. Lee, *Pilchard—A Reconfigurable Computing Platform With Memory Slot Interface,* Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), 2001, California USA, IEEE.
[32] J. S. Meredith, S. R. Alam and J. S. Vetter, *Analysis of a Computational Biology Simulation Technique on Emerging Processing Architectures,* IEEE Int. Workshop on High Performance Computational Biology, 2007.
[33] M. Myers, K. Jaget, S. Cadambi, J. Weener, M. Moe, H. Schmit, S. C. Goldstein and D. Bowersox, *PipeRench Manual*, p. 41, 1998, Carnegie Mellon University.
[34] M. Ohmacht, R. A. Bergamaschi, S. Bhattacharya, A. Gara, M. E. Giampapa, B. Gopalsamy, R. A. Harring, D. Hoenicke, D. J. Krolak, J. A. Marcella, B. J. Nathanson, V. Salapura and M. E. Wazlowski, *Blue Gene/L compute chip: Memory and Ethernet subsystem,* IBM Journal of Research and Development, Vol. 49, No. 2/3, 2005.
[35] S. J. Plimpton, *Fast parallel algorithms for short-range molecular dynamics,* Journal of Comp. Physics, 117, 1995.
[36] V. K. Prasanna and G. R. Morris, *Sparse Matrix Computation on Reconfigurable Hardware,* IEEE Computer, Vol. 40, No. 3, 2007.
[37] R. Scrofano, M. Gokhale, F. Trouw, and V. Prasanna, *A Hardware/Software Approach to Molecular Dynamics on Reconfigurable Computers,* IEEE Symposium on Field-Programmable Custom Computing Machines, 2006.
[38] M. Smith, J. Vetter and S. Alam, *Scientific Computing Beyond CPUs: FPGA Implementations of Common Scientific Kernels,* 8th Annual International MAPLD Conference, 2005.

[39] C. Sosa, T. Hewitt, M. Lee and D. Case, *Vectorization of the generalized Born model for molecular dynamics on shared memory computers*, Journal of Molecular Structure (Theochem) 549, 2001.
[40] M. Taiji, T. Narumi, Y. Ohno, and A. Konagaya, *Protein Explorer: A Petaflops Special-Purpose Computer System for Molecular Dynamics Simulations,* Supercomputing Conference, 2003.
[41] A. Toukmaji, C. Sagui, J. Board, and T. Darden, *Efficient particle-mesh-Ewald based approach to fixed and induced dipolar interactions,* J. Chem. Phys 113, 2000.
[42] K. Underwood, *FPGAs vs. CPUs: Trends in Peak Floating-Point Performance,* Proc. 12$^{\text{th}}$ International Symposium on Field Programmable Gate Arrays (FPGA04), 2004.
[43] K. D. Underwood and K. S. Hemmert, *Closing the GAP: CPU and FPGA Trends in Sustainable Floating-Point BLAS Performance,* Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM04), 2004.
[44] J. S. Vetter, S. R. Alam, T. H. Dunigan, Jr., M. R. Fahey, P. C. Roth and P. H. Worley, *Early Evaluation of the Cray XT3,* 20th IEEE International Parallel & Distributed Processing Symposium (IPDPS), 2006.
[45] L. Zhou and V. Prasanna, *High Performance Linear Algebra Operations on Reconfigurable Systems,* Supercomputing Conference, 2005.

*Edited by:* Dorothy Bollman and Javier Díaz
*Received:* October 18th, 2007
*Accepted:* December 10, 2007 (in revised form: January 25th, 2008)

Scalable Computing: Practice and Experience
Volume 8, Number 4, pp. 411–422. http://www.scpe.org
ISSN 1895-1767


# COMPLEXITY ANALYSIS FOR 4-INPUT/1-OUTPUT FPGAS APPLIED TO MULTIPLIER DESIGNS

NAZAR ABBAS SAQIB*

**Abstract.** Some algorithms are more efficient than others. The complexity of an algorithm is a function describing the efficiency of the algorithm which has two measures: *Space Complexity* and *Time Complexity*. In this paper, we present complexity analysis for FPGA based designs which is based on 4-input and 1-output LUT structure followed by the majority of FPGA manufacturers. The same procedure is then applied to Karatsuba-Offman Multiplier (KOM) because of two reasons. Firstly, due to the increased use of FPGAs especially for security applications, it seems logical to compare various architectures for their efficiencies in FPGAs. Secondly, for diverse security applications, it provides a prior estimation to hardware resources and achievable timing. We consider a 4-input and 1-output structure as a basic building block available in majority of FPGAs by different FPGA manufacturers. We then compare our theoretical and experimental results for KOM in FPGAs which are fairly convincible.

**Key words.** complexity analysis, field programmable gate arrays (FPGAs), Karatsuba-Ofman multiplier, cryptography, hardware implementations

**1. Introduction.** The use of internet for financial applications and electronic commerce has been tremendously increased which has made security a major concern. Public key cryptography [6] provides adequate security solution to those applications. First introduced in 1976, many algorithms were designed and implemented. The most popular schemes are due to RSA [31], ElGamal [9] and Elliptic Curve Cryptosystems (ECCs) [17, 23]. The security of these system is based on computational difficulty for solving some mathematical problems in modular arithmetic, multiplication being the most commonly used and costly operation.

Several quadratic and sub-quadratic space complexity multipliers have been reported in literature. Examples of quadratic multipliers can be found in [20, 18, 41, 42, 37, 13, 38, 35, 13, 28, 43, 11, 19, 32, 29, 30, 22, 7, 15]. On the other hand, some examples of sub-quadratic multipliers can be found in [24, 3, 25, 26, 12, 33, 36, 5, 10, 8, 40, 21]. The latter category offers low complexity especially for large values of $n$ and therefore they are principally attractive for cryptographic applications.

The space and time complexities are the two measures for describing the efficiency of an algorithm. Space complexity is a function describing the amount of memory (space) an algorithm takes in terms of the amount of input to the algorithm. In FPGAs, it refers to the hardware resources (configurable logic blocks, memory, etc) on the chip. Time complexity is a function describing the amount of time an algorithm takes in terms of the amount of input to the algorithm. In FPGAs, it refers to all path delays including gate delays as well as routing overheads. A prior estimation of these two parameters has considerable importance for cost and speed estimations.

In VLSI designs, the estimation for both space and time complexities is relatively straightforward. If two pair of inputs $A\&B$ and $C\&D$ are XORed and their two outputs are ANDed, the space complexity is simply expressed as: #XORs = 2, #ANDs = 1. Similarly, if $T_x$ is the delay for a single gate, time complexity for our example is $2T_x$, One $T_x$ for XORing plus One $T_x$ for ANDing. This is however not the case of an FPGA design. As the basic building block in majority of FPGAs has 4-inputs/1-output structure and also it acts like a Look Up Table (LUT), that is, the whole logic which bounds two, three or four inputs and produces one output, can be accommodated in just a single Look Up Table (LUT). Space complexity is therefore a single basic unit (a single LUT). In contrast to VLSI designs, Time complexity is not $1.T_x$ but it is $1.T_x$ plus path delays due to routing overheads in FPGAs. It has been observed that almost 70% of the total path delays is due to routing overheads in FPGAs. It is therefore difficult to link theoretical results to actual path delays in an FPGA based design. However certain optimizing techniques can be applied to reduce path delays by placing several registers at different stages of the design. At each move, the data travels from one stage to the next stage and hence the net path delay is the maximum delay between any two stages.

Recently, there is an emerging trend for implementing cryptographic primitives in hardware due to improved timing performances and also due to some security reasons. In contrast to software, hardware solutions offer high timing performances which is becoming critical at high speed links. On the other hand for security applications, it is more than that. The secret parameters ( digital keys) in cryptographic primitives are stored in hardware

*Communication Systems Engineering (CSE) department, NUST Institute of Information Technology, National University of Sciences and Technology (NUST), Islamabad-Pakistan (nazar@niit.edu.pk)

and they are not easily accessible which enhances security. Another attractive features of FPGA based designs especially for security applications is due to ease in updating security algorithms as well as secret keys. The focus of this article is to devise a methodology for manipulating space and time complexities for various cryptographic primitives. We have selected Karatsuba-Ofman Multiplier (KOM) as our case study example.

The rest of this paper is organized as follows: Section 2 explains the procedure to perform complexity analysis in FPGAs. Section 3 demonstrate the same procedure for classical multipliers. In Section 4, Karatsuba-Ofman algorithm is explained for its space and time complexities in FPGAs. Section 5 shows the space-time benefits by combining the Karatsuba-Ofman multiplier and other multiplication schemes like classical multipliers. A comparison of all three multiplication schemes has been presented in Section 6. Conclusions are finally drawn in Section 7.

**2. Complexity Analysis for FPGAs based Designs.** FPGAs are being manufactured by many vendors like Xilinx [44], Altera [2], Atmel [4], Quick Logic [27], Actel [1], etc. All manufactures adopt different nomenclature for the hardware resources available on the chip. However the basic structure of almost all the FPGAs is the same. The basic building block in Xilinx FPGAs is called Configurable Logic Block (CLB). Each CLB has two slices and each slice contains one Look Up Table (LUT) other than additional logic. And each LUT has a 4-input and 1-output structure. Similarly, the basic building block in Altera FPGAs is called Logic Array Blocks (LAB). Each LAB contains ten logic elements (LEs) and each LE contains 4-input and 1-output LUT other than additional logic. However modern FPGAs even offer a 6-input and 1-output LUT [39]. Those building blocks are abundantly available in FPGAs. They can be configured into memory as well as into logic mode. Currently, FPGAs offer an integrated environment containing LUTs, Memory blocks, multipliers, transceivers, etc. In this article we focus on the smallest programmable unit in FPGAs, a LUT. We are considering FPGAs with 4-input and 1-output LUT structure for realizing complexity analysis. However the same procedure can be extended to advanced FPGAs with 6-input and 1-output LUTs.

First, in the context of 4-input and 1-output, we discuss two scenarios when number of inputs (IPs) are less than or equal to 4 and when they are greater than four.

***When number of inputs*** $\leq 4$ **:.** Let the output bit $Z$ be the function of four input bits a, b, c, and d, then the significance of a LUT with 4-input and 1-output is that it would occupy just a single LUT in all the cases when $Z$ is the function of two, three or four input bits. Also it does not matter what kind of Boolean logic is involved with those bits, that is,

- *When Z is the function of two bits i.e, Z= F(a,b)*
  Examples
  $$Z = a \oplus a.b;$$
  (One multiplication and one addition)
  or
  $$Z = a \oplus b \oplus a.b;$$
  (One multiplication and two additions)
- *When Z is the function of three bits i.e Z= F(a,b,c)*
  Examples
  $$Z = a \oplus b \oplus c \oplus a.b.c;$$
  (Two multiplications and three additions)
  or
  $$Z = a.b \oplus a.c \oplus b.c \oplus a.b.c;$$
  (Four multiplications and three additions)
- *When Z is the function of four bits i-e Z= F(a,b,c,d)*
  Examples
  $$Z = a.b \oplus a.c \oplus a.d \oplus b.c \oplus b.d \oplus a.b.c \oplus a.c.d \oplus b.c.d \oplus d;$$
  (Eleven multiplications and eight additions)
  or
  $$Z = a \oplus b \oplus c \oplus d \oplus a.b \oplus a.c \oplus a.d \oplus b.c \oplus b.d \oplus a.b.c \oplus b.c.d;$$
  (Nine multiplications and ten additions)

***When number of inputs*** $> 4$ **:.** When $Z$ is the function of more than four bits, it occupies more than one LUTs. For number of inputs from five to seven, $Z$ utilizes two LUTs as four inputs go to the


FIG. 2.1. *Seven input bits to occupy two LUTs*

$1^{st}$ LUT and then its output is fed to the second one acting as an input for the $2^{nd}$ LUT as shown in Fig. 2.1.

As a rule of thumb, for $Z$ as a function of $k$ input bits, it uses some $k/3$ (nearest rounding) LUTs. e.g. The $Z$ as a function of 10 and 11 inputs can be accommodated with $10/3 = 3.33 \cong 3$ and $11/3 = 3.66 \cong 4$ respectively.

The discussed results in this subsection can be applied to perform complexity analysis for any FPGAs based design. We apply this simple procedure to our two case studies for a classical multiplier and a Karatsuba-Ofman multiplier.

**3. Complexity Analysis for a Classical Multiplier.** We start with an example of a classical $4 \times 4$ bit multiplier as shown in Table 3.1.

TABLE 3.1
*4-bit classical multiplier*

| | | | | | | |
|---|---|---|---|---|---|---|
| | | | $a_3$ | $a_2$ | $a_1$ | $a_0$ |
| | | | $b_3$ | $b_2$ | $b_1$ | $b_0$ |
| | | | $a_3b_0$ | $a_2b_0$ | $a_1b_0$ | $a_0b_0$ |
| | | $a_3b_1$ | $a_2b_1$ | $a_1b_1$ | $a_0b_1$ | |
| | $a_3b_2$ | $a_2b_2$ | $a_1b_2$ | $a_0b_2$ | | |
| $a_3b_3$ | $a_2b_3$ | $a_1b_3$ | $a_0b_3$ | | | |
| $z_6$ | $z_5$ | $z_4$ | $z_3$ | $z_2$ | $z_1$ | $z_0$ |

From Table 3.1, one can quickly calculate the value of $k$ and also the number of LUTs (dividing $k$ by 3) for any $z_j$ where $j$=0 to 6 as shown in Table 3.2.

TABLE 3.2
*Complexity analysis for 4-bit classical multiplier*

| $z_j$ | *FunctionF* | Partial Products | $k_j$ | LUTs |
|---|---|---|---|---|
| $z_0$ | $= F(a_0, b_0)$ | $= a_0b_0$ | 2 | 1 |
| $z_1$ | $= F(a_0, b_0, a_1, b_1)$ | $= a_1b_0 \oplus a_0b_1$ | 4 | 1 |
| $z_2$ | $= F(a_0, b_0, a_1, b_1, a_2, b_2)$ | $= a_2b_0 \oplus a_1b_1 \oplus a_0b_2$ | 6 | 2 |
| $z_3$ | $= F(a_0, b_0, a_1, b_1, a_2, b_2, a_3, b_3)$ | $= a_3b_0 \oplus a_2b_1 \oplus a_1b_2 \oplus a_0b_3$ | 8 | 3 |
| $z_4$ | $= F(a_1, b_1, a_2, b_2, a_3, b_3)$ | $= a_3b_1 \oplus a_2b_2 \oplus a_1b_3$ | 6 | 2 |
| $z_5$ | $= F(a_2, b_2, a_3, b_3)$ | $= a_3b_2 \oplus a_2b_3$ | 4 | 1 |
| $z_6$ | $= F(a_3, b_3)$ | $= a_3b_3$ | 2 | 1 |
| | | | Total | 11 |

Hence, a 4-bit classical multiplier can be realized with no less than eleven 4-input and 1-output LUTs as shown in Fig. 3.1.

The procedure for performing complexity analysis of a 4-bit multiplier can be generalized to any $n$-bit multiplier which consists of three steps:

*Step 1:* Write down the number of inputs $k_j$ for all partial sums $z_j$. It can be obtained first by writing $n$ in the middle and then by writing all of its values from $(n-1)$ to 1 on its both sides. That gives the number of partial products for any partial sum $z_j$, that is,

$$1 \ldots\ldots \quad (n-2) \quad (n-1) \quad n \quad (n-1) \quad (n-2) \quad \ldots\ldots 1 \tag{3.1}$$

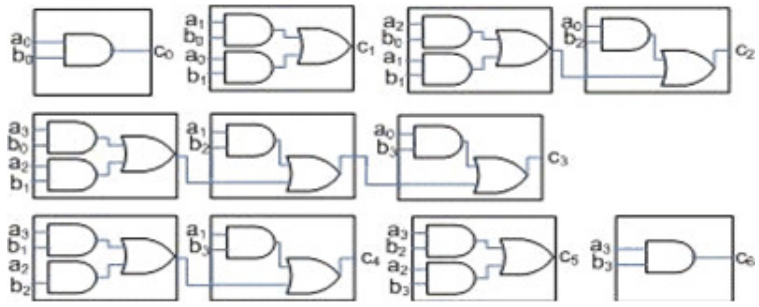



FIG. 3.1. *4-bit classical multiplier implementation using 4-input and 1-output LUTs*

For $n = 4$ (4-bit multiplier),

$$1 \quad 2 \quad 3 \quad 4 \quad 3 \quad 2 \quad 1 \tag{3.2}$$

As a single partial product contributes to two inputs, multiplying it by two, it give the number of inputs $k_j$ for all partial sums $z_j$, that is,

$$2 \ldots\ldots \quad 2(n-2) \quad 2(n-1) \quad 2n \quad 2(n-1) \quad 2(n-2) \quad \ldots\ldots 2 \tag{3.3}$$

For $n = 4$ (4-bit multiplier),

$$2 \quad 4 \quad 6 \quad 8 \quad 6 \quad 4 \quad 2 \tag{3.4}$$

$$2n \quad 4(n-1) \quad 4(n-2) \quad \ldots\ldots 4 \tag{3.5}$$

*Step 2:* The number of LUTs for all partial sums $z_j$ are manipulated by dividing each $k_j$ by 3 and rounding it to the nearest integer value, that is,

$$\frac{2}{3} \quad \ldots\ldots \quad \frac{2(n-2)}{3} \quad \frac{2(n-1)}{3} \quad \frac{2n}{3} \quad \frac{2(n-1)}{3} \quad \frac{2(n-2)}{3} \quad \ldots\ldots \quad \frac{2}{3} \tag{3.6}$$

For $n = 4$ (4-bit multiplier),

$$\frac{2}{3} \quad \frac{4}{3} \quad \frac{6}{3} \quad \frac{8}{3} \quad \frac{6}{3} \quad \frac{4}{3} \quad \frac{2}{3} \tag{3.7}$$

*Step 3:* The number of LUTs for all partial sums $z_j$ are added to calculate total number of LUTs for any n-bit classical multiplier,

$$\frac{2}{3} + \cdots + \frac{2(n-2)}{3} + \frac{2(n-1)}{3} + \frac{2n}{3} + \frac{2(n-1)}{3} + \frac{2(n-2)}{3} + \ldots\ldots + \frac{2}{3} \tag{3.8}$$

By simplifying,

$$\frac{2n}{3} + 2\frac{2(n-1)}{3} + 2\frac{2(n-2}{3} + \quad \ldots\ldots \quad + 2.\frac{2}{3} \tag{3.9}$$

$$\frac{2n}{3} + \frac{4}{3}\{(n-1) + (n-2) + \quad \ldots\ldots \quad + 1\} \tag{3.10}$$

The terms in brackets in Eq. 3.10 forms an arithmetic series for which the sum is equal to $\frac{n(n-1)}{2}$, by substituting:

$$\frac{2n}{3} + \frac{4}{3}\left\{\frac{n(n-1)}{2}\right\} = \frac{2}{3}n^2 \tag{3.11}$$

For 4-bit multiplier

$$\frac{2}{3} + \frac{4}{3} + \frac{6}{3} + \frac{8}{3} + \frac{6}{3} + \frac{4}{3} + \frac{2}{3} \tag{3.12}$$

By simplifying,

$$\frac{8}{3} + 2.\frac{6}{3} + 2.\frac{4}{3} + 2.\frac{2}{3} \tag{3.13}$$

$$3 + 4 + 2 + 2 = 11$$

By using this formula, one can calculate the gate complexity for any n-bit classical multiplier. Table 3.3 provides LUTs (cal) using the derived expression in Eq. 3.11 and also the number of LUTs (exp) experimented for first 40 values of n. The calculated LUTs exactly match with the experimental LUTs as we have instantiated LUT

TABLE 3.3
*Gate complexities for first 40 values of n using classical multiplier*

| n | LUTs (cal) | LUTs (Exp) | n | LUTs (cal) | LUTs (Exp) |
|---|---|---|---|---|---|
| 1 | 1 | 1 | 21 | 294 | 294 |
| 2 | 3 | 3 | 22 | 323 | 323 |
| 3 | 6 | 6 | 23 | 353 | 353 |
| 4 | 11 | 11 | 24 | 384 | 384 |
| 5 | 17 | 17 | 25 | 417 | 417 |
| 6 | 24 | 24 | 26 | 451 | 451 |
| 7 | 33 | 33 | 27 | 486 | 486 |
| 8 | 43 | 43 | 28 | 523 | 523 |
| 9 | 54 | 54 | 29 | 561 | 561 |
| 10 | 67 | 67 | 30 | 600 | 600 |
| 11 | 81 | 81 | 31 | 641 | 641 |
| 12 | 96 | 96 | 32 | 683 | 683 |
| 13 | 113 | 113 | 33 | 726 | 726 |
| 14 | 131 | 131 | 34 | 771 | 771 |
| 15 | 150 | 150 | 35 | 817 | 817 |
| 16 | 171 | 171 | 36 | 864 | 864 |
| 17 | 193 | 193 | 37 | 913 | 913 |
| 18 | 216 | 216 | 38 | 963 | 963 |
| 19 | 241 | 241 | 39 | 1014 | 1014 |
| 20 | 267 | 267 | 40 | 1067 | 1067 |

module implicitly in our VHDL code.

**4. Complexity Analysis for Karatsuba-Ofman Multiplier.** Discovered in 1962, a divide-and-conquer algorithm due to Karatsuba and Ofman was the first algorithm [16] to accomplish polynomial multiplication in under $O(m^2)$ operations and reduces the complexity to $O(n^{log_2^3})$. Suppose that $n = 2l$ and $A = A^H 2^l + A^L$ and $B = B^H 2^l + B^L$ are $2l$-bit integers.
Then

$$\begin{aligned} AB &= (A^H 2^l + A^L)(B^H 2^l + B^L) \\ &= A^H B^H 2^{2l} + [(A^H + A^L)(B^H + B^L) - A^H B^H - A^L B^L]2^l + A^L B^L \end{aligned}$$

The product AB can be computed by performing three multiplications of l-bit integers along with two additions and two subtractions. More details about Karatsuba-Ofman multiplication can be seen in [14, 34].

Let we take again the example of a $4 \times 4$ multiplier using Karatsuba-Ofman multiplication scheme.

Let A and B the two multiplicands with,

$$A = a_3 a_2 a_1 a_0 \quad and \quad B = b_3 b_2 b_1 b_0 \tag{4.1}$$

Both A and B are divided into lower $A^L \& B^L$ and higher parts $A^H \& B^H$:

$$A^H = a_3 a_2 \quad and \quad B^H = b_3 b_2 \tag{4.2}$$

$$A^L = a_1 a_0 \quad and \quad B^L = b_1 b_0 \tag{4.3}$$

Then three multiplications are required to be performed:

1. First multiplication between $A^H$ and $B^H$

$$A^H B^H = (a_3 a_2)(b_3 b_2) = H_2 H_1 H_0 \tag{4.4}$$

2. Second multiplication between $A^L$ and $B^L$

$$A^L B^L = (a_1 a_0)(b_1 b_0) = L_2 L_1 L_0 \tag{4.5}$$

For third multiplication the higher and the lower parts of both the operands are XORed.

$$M^A = A^H \oplus A^L = (a_3 a_2) \oplus (a_1 a_0) = m_{a1} m_{a0} \tag{4.6}$$

$$M^B = B^H \oplus B^L = (b_3 b_2) \oplus (b_1 b_0) = m_{b1} m_{b0} \tag{4.7}$$

3. Third multiplication between $M^A$ and $M^B$

$$M^A M^B = (m_{a1} m_{a0})(m_{b1} m_{b0}) = M_2 M_1 M_0 \tag{4.8}$$

Finally the overlapping of the three partial products is performed:

TABLE 4.1
*Overlapping Function for a 4-bit Karatsuba-Ofman Multiplier*

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | $H_2$ | $H_1$ | $H_0$ | | | |
| | | $L_2$ | $L_1$ | $L_0$ | | | $\oplus$ |
| | | $M_2$ | $M_1$ | $M_0$ | | | $\oplus$ |
| $H_2$ | $H_1$ | $H_0$ | | $L_3$ | $L_1$ | $L_0$ | $\oplus$ |
| $z_6$ | $z_5$ | $z_4$ | $z_3$ | $z_2$ | $z_1$ | $z_0$ | |

By looking at the above expressions one can estimate the resource utilization as follows:

1. Three $n/2$ multiplications are always performed by using Karatsuba-Ofman multiplication scheme. For a $4 \times 4$ Karatsuba-Ofman multiplier, it therefore requires three 2-bit multipliers as it is shown in Eqs. 4.4,4.5, and 4.8. A 2- bit multiplier using Karatsuba-Ofman multiplication scheme costs 3 LUTs, hence a total of 9 LUTs are being used.

2. For third multiplication the two inputs of the multiplier are to be XORed as it has been shown in Eqs. 4.6 and 4.7. They always require some $2 \times n/2$ XOR operations, and the same amount of LUTS i-e $n$ LUTs. For $n = 4$, four LUTs are therefore utilized.
3. Finally the overlapping part is concluded with $3n - 4$ XORs thus consuming $(3n - 4)/3 = n - 1$ LUTs. For a 4-bit multiplier it is evident the utilization of three LUTs in obtaining $z_3$, $z_4$ and $z_5$, we call them as output XORs.

The total number of LUTs for a 4-bit Karatsuba-Ofman multiplier can be obtained by adding all LUTs from the above three steps which are 15. Some other results can also be deduced:

- LUTs due to input XORs = $2(n/2) = n$
- LUTs due to output XORs = $n - 1$
- LUTs due to both input & output XORs = $n + (n - 1) = 2n - 1$
- LUTs due to three multipliers= $3\times$ LUTs used by the base multiplier

The above procedure can be extended to generalize the expression for the estimation of number of LUTs for any n-bit Karatsuba-Ofman multiplier. We select a 4-bit Karatsuba-Ofman multiplier as a base multiplier, then,

For a 4-bit Karatsuba-Ofman multiplier ($n = 4$) :
Total number of LUTs = 15

For an 8-bit Karatsuba-Ofman multiplier ($n = 8$) :
Total number of LUTs
$= $ LUTs due to input/output XORs + $3\times$ LUTs used by the 4-bit multiplier
$= (2n - 1) + 14(3)^1$
$= 15 + 15(3)^1 = 15(3)^0 + 15(3)^1 = K_1$

For a 16-bit Karatsuba-Ofman multiplier ($n = 16$) :
Total number of LUTs
$= $ LUTs due to input/output XORs + $3\times$ LUTs used by the 8-bit multiplier
$= (2n - 1) + 3 \times K_1$
$= 31 + 3\left\{15(3)^0 + 15(3)^1)\right\} = 31 + 15(3)^1 + 15(3)^2 = K_2$

For a 32-bit Karatsuba-Ofman multiplier ($n = 32$) :
Total number of LUTs
$= $ LUTs due to input/output XORs + $3\times$ LUTs used by the 16-bit multiplier
$= (2n - 1) + 3 \times K_2$
$= 63 + 3\left\{31 + 15(3)^1 + 15(3)^2\right\} = 63 + 31(3)^1 + 15(3)^2 + 15(3)^3 = K_3$

For a 64-bit Karatsuba-Ofman multiplier ($n = 64$) :
Total number of LUTs
$= $ LUTs due to input/output XORs + $3\times$ LUTs used by the 32-bit multiplier
$= (2n - 1) + 3 \times K_3$
$= 127 + 3\left\{63 + 31(3)^1 + 15(3)^2 + 15(3)^3\right\}$
$= 127 + 63(3)^0 + 31(3)^2 + 15(3)^3 + 15(3)^4$

On continuing in a similar way, we can generalize the above expressions for any $n$:

$$15(3)^k + \left\{\frac{2n-1}{1}3^0 + (\frac{2n}{2} - 1)3^1 + (\frac{n}{2} - 1)3^2 + (\frac{n}{4} - 1)3^3 + \cdots (\frac{n}{k-1} - 1)3^{k-1}\right\} \tag{4.9}$$

where $k$ is the number of iterations and it is calculated as: $k = log_2(n) - 2$. The subtraction of factor of 2 is due to the selection of 4-bit multiplier as a base multiplier which removes two iterations for 2 and 4 bit multiplications.

Rewriting Eq. 4.9,

$$15(3)^k + \left\{ \frac{2n}{1}3^0 + \frac{2n}{2}3^1 + \frac{n}{2}3^2 + \cdots + \frac{n}{k-1}3^{k-1} \right\} - \left\{ 3^0 + 3^1 + 3^2 + \cdots + 3^{k-1} \right\} \tag{4.10}$$

The terms in brackets in Eq. 4.10 form a geometric series similar to $a + ar + ar^2 + ar^3 + \cdots$ where '$a$' represents the initial value and '$r$' is the ratio which can be obtained by dividing a value to its previous one. The sum of $n^{th}$ terms for that series can be calculated by the formula:

$$S_n = a(1 - r^n)/(1 - r) \tag{4.11}$$

The sum of $n^{th}$ series for the two geometric expressions in Eq. 4.10 can be manipulated by using the formula in Eq. 4.11.

For the first series,

$$\left\{ \frac{2n}{1}3^0 + \frac{2n}{2}3^1 + \frac{n}{2}3^2 + \cdots + \frac{n}{k-1}3^{k-1} \right\} \tag{4.12}$$

Initial value = $a = 2n$ & ratio = $r = 3/2$
Therefore the sum of $n^{th}$ terms is:

$$= 4n \left[ (3/2)^k - 1 \right] \tag{4.13}$$

For the second series,

$$\left\{ 3^0 + 3^1 + 3^2 + \cdots + 3^{k-1} \right\} \tag{4.14}$$

Initial value = $a = 1$ & ratio=r= 3
Therefore the sum of the $n^{th}$ terms is:

$$= 1/2 \left[ 3^k - 1 \right] \tag{4.15}$$

Substituting Eqs. 4.13 and 4.13 into Eq. 4.10,

$$15(3)^k + 4n \left[ (3/2)^k - 1 \right] - 1/2 \left[ (3)^k - 1 \right] \tag{4.16}$$

Eq. 4.16 can be written in terms of just '$n$' by substituting the value of '$k$'

$$15(3)^{log_2(n)-2} + 4n \left[ (3/2)^{log_2(n)-2} - 1 \right] - 1/2 \left[ (3)^{log_2(n)-2} - 1 \right] \tag{4.17}$$

where $k = log_2(n) - 2$

By using the formula in Eq. 4.17, we can calculate the space complexity for several $n = 2^k$-bit Karatsuba-Ofman multipliers as shown in Table 4.2. Table 4.2 also provides our experimental results for the same values which shows minor difference to the calculated values to non-optimal behavior of HDL (Hardware Description Language) compilers.

**5. Complexity Analysis for Karatsuba-Ofman multiplier using Hybrid approach.** In order to construct a bigger multiplier for any larger value '$m$', we can use Karatsuba-Ofman multiplication approach by using a smaller multiplier recursively. The smaller multiplier represents the end point where recursion process exactly starts and it is termed as a base multiplier. A base multiplier can be constructed by any other multiplication approach like classical multiplication scheme as well. For example, we can construct an 8-bit multiplier from three 4-bit multipliers. Similarly a 16-bit multiplier can be constructed by using three 8-bit

TABLE 4.2
*Space complexity for $n = 2^k$-bit Karatsuba-Ofman multiplier in terms of LUTs*

| n | LUTs (cal) | LUTs (Exp) |
|---|---|---|
| 2 | 3 | 3 |
| 4 | 15 | 14 |
| 8 | 60 | 60 |
| 16 | 211 | 212 |
| 32 | 696 | 698 |
| 64 | 2215 | 2221 |
| 128 | 6900 | 6918 |
| 256 | 21211 | 21265 |
| 512 | 64656 | 64818 |

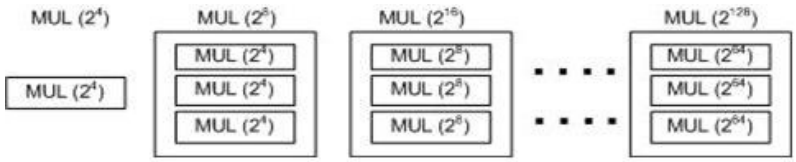



FIG. 5.1. *4-bit classical multiplier implementation using 4-input and 1-output LUTs*

mutipliers and so on. A block diagram representation of this hierarchical setup by selecting a 4-bit multiplier as a base multiplier is shown in Fig. 5.1.

Karatsuba-Ofman multiplier therefore can be viewed as a long array of base multipliers in middle and a logic mapping required for input and output (overlapping) XOR operations as it has been depicted in Fig. 5.2.

The selection of the base multiplier is therefore critical to save the hardware resources. The saving of few LUTs in the base multiplier helps in saving significant number of LUTs for large values of n. A hybrid approach is therefore used which dictates the use of other multiplication schemes along with Karatsuba-Ofman multiplication. We have implemented 4-bit Karatsuba-Ofman multiplier using the classical approach (school method) which seems to be economical as compared to 4-bit Karatsuba-Ofman multiplier as it occupies 11 LUTs instead of 15 LUTs. The change of only base multiplier does not require any change in the formula for complexity analysis, the factor of 15 is simply replaced with 11. The formula for an hybrid Karatsuba-Ofman multiplier using a 4-bit classical multiplier as a base multiplier is shown in Eq. 5.1.

$$11(3)^{log_2^n-2} + 4n\left[(3/2)^{log_2^n-2} - 1\right] - 1/2\left[3^{log_2^n-1} - 1\right] \tag{5.1}$$

By using Eq. 5.1, the space complexity for hybrid Karatsuba-Ofman multiplier can be manipulated as shown in Table 5.1.

TABLE 5.1
*Space complexity for $n = 2^k$-bit Hybrid Karatsuba-Ofman multiplier in terms of LUTs*

| n | LUTs (cal) | LUTs (Exp) |
|---|---|---|
| 2 | 3 | 3 |
| 4 | 11 | 11 |
| 8 | 48 | 45 |
| 16 | 175 | 168 |
| 32 | 588 | 567 |
| 64 | 1891 | 1828 |
| 128 | 5928 | 5739 |
| 256 | 18295 | 17728 |
| 512 | 55908 | 54207 |

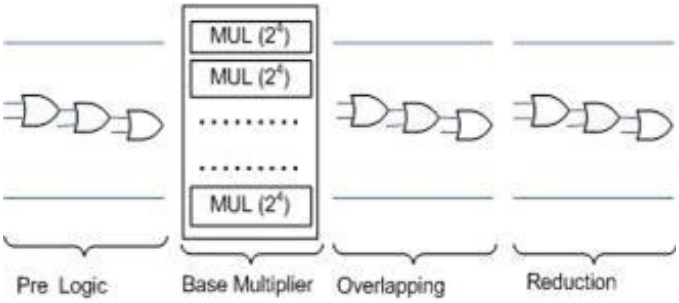



FIG. 5.2. *Flattend Image of Karatsuba-Ofman multiplier using a* $MUL(2)^4$ *as a base multiplier*

**6. Performance Results.** The achieved results for the space complexities of classical, Karatsuba-Ofman and hybrid Karatsuba-Ofman multiplication schemes can be combined for comparison purposes as shown in Table 6.1.

TABLE 6.1
*Space complexity for* $n = 2^k$*-bit Classical, Karatsuba-Ofman, and Hybrid Karatsuba-Ofman multiplication schemes in terms of LUTs*

| n | LUTs (cal) Classical multiplier | LUTs (cal) Karatsuba-Ofman multiplier | LUTs (cal) H. Karatsuba-Ofman multiplier |
|---|---|---|---|
| 2 | 3 | 3 | 3 |
| 4 | 11 | 15 | 11 |
| 8 | 43 | 60 | 48 |
| 16 | 171 | 211 | 175 |
| 32 | 683 | 696 | 588 |
| 64 | 2731 | 2215 | 1891 |
| 128 | 10923 | 6900 | 5928 |
| 256 | 43691 | 21211 | 18295 |
| 512 | 174763 | 64656 | 55908 |

It can be seen from Table 6.1 that classical multiplication schemes proves to be more economical for $n < 32$ when complexity analysis is performed for FPGAs based designs. For $n > 32$, however, hybrid Karatsuba-Ofman multiplication approach proves to be more economical.

**7. Conclusion.** In this paper, we explained in detail how to perform complexity analysis for an FPGA based design. We applied that procedure for manipulating space complexities for a classical Karatsuba-Ofman multiplier, Karatsuba-Ofman multiplier and an Hybrid Karatsuba-Ofman multiplier. It has been shown that obtained experimental results are exactly in match with those of theoretical manipulations in all three cases. The similar procedure can be extended to realize complexity analysis for other cryptographic primitives. The comparison tables for all three multiplication schemes can be utilized for selecting a base multiplier to construct a bigger multiplier as it is required in cryptographic applications. Our future work includes the construction of a low cost multiplier in FPGAs on the basis of the results obtained in this paper. Also we used a 4-input and 1-output structure for a LUT as the basic building block to perform complexity analysis for an FPGA based design. Modern FPGAs however offer a 6-input and 1-output structure for their basic building block. We have also planned to extend our manipulations for those FPGA devices.

REFERENCES

[1] ACTEL, 2008. Available at: `http://www.actel.com/`
[2] ALTERA, 2008. Available at: `http://www.xilinx.com/`
[3] C. P. AND, *A New Architecture for a Parallel Finite Field Multiplier with Low Complexity Based on Composite Fields*, IEEE Transactions on Computers, 45(7) (1996), pp. 856–861.

[4] Atmel, 2008. Available at: http://atmel.com/

[5] J. C. Bajard, L. Imbert, and G. A. Jullien, *Parallel Montgomery Multiplication in GF($2^k$) Using Trinomial Residue Arithmetic*, in 17th IEEE Symposium on Computer Arithmetic (ARITH-17 2005), 27-29 June 2005, Cape Cod, MA, USA, IEEE Computer Society, 2005, pp. 164–171.

[6] W. Diffie and M. E. Hellman, *New directions in cryptography*, IEEE Transactions on Information Theory, IT-22 (1976), pp. 644–654.

[7] H. Fan and Y. Dai, *Fast Bit-Parallel GF($2^n$) Multiplier for All Trinomials*, IEEE Trans. Computers, 54 (2005), pp. 485–490.

[8] H. Fan and M. A. Hasan, *A New Approach to Subquadratic Space Complexity Parallel Multipliers for Extended Binary Fields*. Centre for Applied Cryptographic Research (CACR) Technical Report CACR 2006-02, 2006. available at: http://www.cacr.math.uwaterloo.ca/

[9] T. E. Gamal, *A public key cryptosystem and a signature scheme based on discrete logarithms*, in Proceedings of CRYPTO 84 on Advances in cryptology, New York, NY, USA, 1985, Springer-Verlag New York, Inc., pp. 10–18.

[10] J. Gathen and J. Shokrollahi, *Efficient FPGA-Based Karatsuba Multipliers for Polynomials over $F_2$.*, in Selected Areas in Cryptography, 12th International Workshop, SAC 2005, Kingston, ON, Canada, August 11-12, 2005, Revised Selected Papers, vol. 3897 of Lecture Notes in Computer Science, Springer-Verlag, 2006, pp. 359–369.

[11] D. Gollmann, *Equally Spaced Polynomials, Dual Bases, and Multiplication in $F_{2^n}$*, IEEE Trans. Computers, 51 (2002), pp. 588–591.

[12] C. Grabbe, M. B., J. Gathen, J. Shokrollahi, and J. Teich, *A High Performance VLIW Processor for Finite Field Arithmetic*, in 17th International Parallel and Distributed Processing Symposium (IPDPS 2003), 22-26 April 2003, Nice, France, CD-ROM/Abstracts Proceedings, IEEE Computer Society, 2003, p. 189.

[13] A. Halbutogullari and Ç. K. Koç, *Parallel Multiplication in using Polynomial Residue Arithmetic*, Des. Codes Cryptography, 20 (2000), pp. 155–173.

[14] D. Hankerson, A. J. Menezes, and S. Vanstone, *Guide to Elliptic Curve Cryptography*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[15] J. L. Imana, J. M. Sanchez, and F. Tirado, *Bit-Parallel Finite Field Multipliers for Irreducible Trinomials*, IEEE Transactions on Computers, 55 (2006), pp. 520–533.

[16] A. Karatsuba and Y. Ofman, *Multiplication of Multidigit Numbers on Automata*, Soviet Phys. Doklady (English Translation), 7 (1963), pp. 595–596.

[17] N. Koblitz, *CM-Curves with Good Cryptographic Properties.*, in CRYPTO, vol. 576 of Lecture Notes in Computer Science, Springer, 1991, pp. 279–287.

[18] Ç. K. Koç and T. Acar, *Montgomery Multiplication in GF($2^k$).*, Designs, Codes and Cryptography, 14 (1998), pp. 57–69.

[19] S. O. Lee, S. W. Jung, C. H. Kim, J. Yoon, J. Y. Koh, and D. Kim, *Design of Bit Parallel Multiplier with Lower Time Complexity*, in Information Security and Cryptology - ICISC 2003, 6th International Conference, Seoul, Korea, November 27-28, 2003, Revised Papers, vol. 2971 of Lecture Notes in Computer Science, Springer-Verlag, 2004, pp. 127–139.

[20] E. D. Mastrovito, *VLSI Designs for Multiplication over Finite Fields f ($2^m$).*, in Applied Algebra, Algebraic Algorithms and Error-Correcting Codes, 6th International Conference, AAECC-6, Rome, Italy, July 4-8, 1988, Proceedings, vol. 357 of Lecture Notes in Computer Science, Springer-Verlag, 1989, pp. 297–309.

[21] P. L. Montgomery, *Five, Six, and Seven-Term Karatsuba-Like Formulae*, IEEE Trans. Comput., 54 (2005), pp. 362–369.

[22] C. Nègre, *Quadrinomial Modular Arithmetic using Modified Polynomial Basis*, in International Symposium on Information Technology: Coding and Computing (ITCC 2005), Volume 1, 4-6 April 2005, Las Vegas, Nevada, USA, IEEE Computer Society, 2005, pp. 550–555.

[23] N. I. of Standards and Technology, *Recomended Elliptic Curves for Federal Government Use*, 1997.

[24] C. Paar, *Efficient VLSI Architectures for Bit Parallel Computation in Galois Fields*, PhD thesis, Universität GH Essen, 1994.

[25] C. Paar, P. Fleischmann, and P. Roelse, *Efficient Multiplier Architectures for Galois Fields GF($2^{4n}$)*, IEEE Trans. Computers, 47 (1998), pp. 162–170.

[26] C. Paar, P. Fleischmann, and P. Soria-Rodriguez, *Fast Arithmetic for Public-Key Algorithms in Galois Fields with Composite Exponents*, IEEE Trans. Computers, 48 (1999), pp. 1025–1034.

[27] QuickLogic, 2008. Available at: http://quicklogic.com/.

[28] A. Reyhani-Masoleh and M. A. Hasan, *A New Construction of Massey-Omura Parallel Multiplier over f(2)*, IEEE Trans. Computers, 51 (2002), pp. 511–520.

[29] A. ReyhaniMasoleh and M. A. Hasan, *Efficient Multiplication Beyond Optimal Normal Bases*, IEEE Trans. Computers, 52 (2003), pp. 428–439.

[30] A. Reyhani-Masoleh and M. A. Hasan, *Low Complexity Bit Parallel Architectures for Polynomial Basis Multiplication over GF(2{$m$})*, IEEE Trans. Computers, 53 (2004), pp. 945–959.

[31] R. L. Rivest, A. Shamir, and L. M. Adelman, *A METHOD FOR OBTAINING DIGITAL SIGNATURES AND PUBLIC-KEY CRYPTOSYSTEMS*, Tech. Report MIT/LCS/TM-82, 1977.

[32] F. Rodríguez-Henríquez and Ç. K. Ko, *Parallel Multipliers Based on Special Irreducible Pentanomials*, IEEE Trans. Computers, 52 (2003), pp. 1535–1542.

[33] F. Rodríguez-Henríquez and Ç. K. Koç, *On Fully Parallel Karatsuba Multipliers for GF($2^m$)*, in International Conference on Computer Science and Technology (CST 2003), Cancun, Mexico, May 2003, pp. 405–410.

[34] F. Rodríguez-Henríquez, N. A. Saqib, A. Díaz-Pèrez, and C. K. Koc, *Cryptographic Algorithms on Reconfigurable Hardware (Signals and Communication Technology)*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[35] E. Savas, A. F. Tenca, and Ç. K. Koç, *A Scalable and Unified Multiplier Architecture for Finite Fields GF() and GF($2^m$)*, in Cryptographic Hardware and Embedded Systems - CHES 2000, Second International Workshop, Worcester, MA, USA, August 17-18, 2000, Proceedings, vol. 1965 of Lecture Notes in Computer Science, Springer-Verlag, 2000, pp. 277–292.

[36] B. Sunar, *A Generalized Method for Constructing Subquadratic Complexity GF($2^k$) Multipliers*, IEEE Trans. Computers, 53 (2004), pp. 1097–1105.

[37] B. Sunar and Ç. K. Koç, *Mastrovito Multiplier for All Trinomials*, IEEE Trans. Computers, 48 (1999), pp. 522–527.
[38] B. Sunar and Ç. K. Koç, *An Efficient Optimal Normal Basis Type II Multiplier*, IEEE Trans. Computers, 50 (2001), pp. 83–87.
[39] Virtex5, 2008. Available at: `http://www.xilinx.com/support/documentation/virtex-5.htm`
[40] A. Weimerskirch and C. Paar, *Generalizations of the Karatsuba Algorithm for Efficient Implementations.* Ruhr-Universität-Bochum, Germany. Technical Report, 2003. available at: `http://www.crypto.ruhr-uni-bochum.de/en_publications.html`
[41] H. Wu and M. A. Hasan, *Low Complexity Bit-Parallel Multipliers for a Class of Finite Fields*, IEEE Trans. Computers, 47 (1998), pp. 883–887.
[42] H. Wu, M. A. Hasan, and I. F. Blake, *New Low-Complexity Bit-Parallel Finite Field Multipliers Using Weakly Dual Bases*, IEEE Trans. Computers, 47 (1998), pp. 1223–1234.
[43] H. Wu, M. A. Hasan, I. F. Blake, and S. Gao, *Finite Field Multiplier Using Redundant Representation*, IEEE Trans. Computers, 51 (2002), pp. 1306–1316.
[44] Xilinx, 2008. Available at: `http://www.xilinx.com/`

*Edited by:* Francisco Rodriguez-Henriquez
*Received:* December 30th, 2007
*Accepted:* January 17th, 2008

Scalable Computing: Practice and Experience
Volume 8, Number 4, pp. 423–435. http://www.scpe.org
ISSN 1895-1767


# TIME QUANTUM GVT: A SCALABLE COMPUTATION OF THE GLOBAL VIRTUAL TIME IN PARALLEL DISCRETE EVENT SIMULATIONS

GILBERT G. CHEN* AND BOLESLAW K. SZYMANSKI†

**Abstract.** This paper presents a new Global Virtual Time (GVT) algorithm, called TQ-GVT that is at the heart of a new high performance Time Warp simulator designed for large-scale clusters. Starting with a survey of numerous existing GVT algorithms, the paper discusses how other GVT solutions, especially Mattern's GVT algorithm, influenced the design of TQ-GVT, as well as how it avoided several types of overheads that arise in clusters executing parallel discrete simulations. The algorithm is presented in details, with a proof of its correctness. Its effectiveness is then verified by experimental results obtained on more than 1,000 processors for two applications, one synthetic workload and the other a spiking neuron network simulation.

**Key words.** global virtual time, time warp, parallel discrete event simulation, scalability

**1. Introduction.** Parallel discrete event simulators [1] are perhaps among the most sophisticated distributed systems developed. A Parallel Discrete Event Simulation (PDES) must execute events according to their inherent timestamp order which may differ from the order in which they are created. Historically, two main methods have been introduced to deal with this problem, one called conservative [2, 3] and the other optimistic (or Time Warp) [4].

The notion of Global Virtual Time (or GVT) [4] was first introduced by Jefferson to track the earliest unprocessed events in the entire simulation. By definition, GVT at any given instance of the simulation execution is the minimum value among the local virtual times of all processors and the timestamps of all messages in transit. Any processed event with a timestamp earlier than the current GVT will not be rolled back under any circumstances, and therefore the memory associated with it can be safely released. Without the notion of GVT, the Time Warp mechanism would be impractical because of its huge memory consumption. However, it is impossible to compute the exact GVT as it would require collecting information on distributed processors at exactly the same wall-clock time. Fortunately, a lower bound on GVT is also useful as events earlier than such a bound can be safely removed. Although events with a timestamp larger than the GVT estimate but smaller than the true GVT value cannot be removed, those events should not have a significant impact on memory usage, as long as the GVT estimate is sufficiently close to the true GVT value.

GVT computation is perhaps the only global operation in Time Warp. All others, such as rollbacks, state saving, and sending and handling of anti-messages, can be carried out locally. Therefore, GVT computation is known to be the least scalable component of Time Warp and it is no surprise that the accuracy and overhead of the GVT computation may dominate the overall performance of Time Warp.

GVT is useful not for Time Warp only. A few variants of conservative protocols, such as the conditional event approach [5] and the LBTS approach [6], which largely depend on the amount of lookahead, also need to compute Lower Bound on Time Step (LBTS) which computationally is equivalent to GVT. In our future work, we will evaluate performance of TQ-GVT for such software platforms. The less-known third class of PDES protocols is based on lookback [7, 8] and some of its variants (see [9] for details) rely on prompt GVT estimates as well. A good GVT algorithm is the key to the efficiency of these protocols.

Numerous GVT algorithms [10]–[24] have been devised. Many papers introducing them [10, 11, 14, 16, 20, 22] focused on feasibility and correctness of GVT computation but did not provide performance data. Those that did, gave the results of runs with quite limited number of processors. The largest Time Warp simulation that has been described in the literature, in terms of the processor count, was presented in [25], which, however, did not employ a general GVT algorithm. The largest Time Warp simulation with a general GVT algorithm was reported in [26] with runs on 104 processors. The first Time Warp simulator, Time Warp Operating System, had to limit GVT computation frequency to less than one execution per every 5 seconds on 32 processors, because of its overhead [27].

This paper presents a new GVT algorithm, called Time Quantum GVT (TQ-GVT), which, although conceptually simple, is efficient and scalable. TQ-GVT has proven to be able to deliver GVT estimates every tenth of a second on as many as 1,024 processors. The associated overhead is small: each GVT message is either 16

*Center for Pervasive Computing and Networking, RPI, Troy, NY, 12180, USA (cheng3@cs.rpi.edu).
†Department of Computer Science, RPI, Troy, NY, 12180, USA (szymansk@cs.rpi.edu).

or 48 bytes long. In addition, the total number of GVT messages needed for each GVT computation is always twice the number of processors. Hence, the number of GVT messages per processor is constant, independent of the number of processors used. As a result, the aggregate network bandwidth consumed by GVT computation alone is less than 1 Mbytes per second on a parallel computer with 1,024 processors.

The paper first surveys other GVT algorithms. It then describes how TQ-GVT works, proves its correctness, and provides concrete evidence that TQ-GVT works as expected on two applications, one synthetic and the other realistic. Finally, it concludes by summarizing the properties of TQ-GVT and by outlining an additional research that can be done on this topic.

**2. Related Work.** Designs of GVT algorithms focus on either shared-memory or distributed computers. Shared-memory GVT algorithms assume that certain variables are accessible by all processors [28, 29], so they perform well on Symmetric Multi-Processing machines, but their performance is unclear on Non-Uniform Memory Access ones in which dependence on global variables limits scalability of any algorithm.

Distributed GVT algorithms do not use global variables and therefore are more scalable. By definition, GVT at a wall-clock time is either the smallest local virtual time or the smallest timestamp of all messages in transit. As it is impossible to measure local virtual times at the same wall-clock time on different processors, techniques were needed to make the measurements look like they were taken simultaneously. Most of these techniques are in general based on overlapping intervals [10, 12, 14, 16, 19], two cuts [13, 17], or global reduction [18, 20, 21].

**Overlapping Intervals.** The overlapping interval technique selects an interval of wall-clock time for each processor, such that the intersection of all intervals is nonempty. Any point in wall-clock time within this intersection can be viewed as an instant at which the GVT measurement was taken. The smallest local virtual time at any such instant is guaranteed to be no smaller than the smallest local virtual time within the intersection of all intervals.

Two methods are normally used for building these overlapping intervals. The first method, widely used in early GVT algorithms, consists of broadcasting two messages from an initiator [12, 14, 19], a START message to begin the process, and then, after receiving responses from all processors, a STOP message, after which the initiator waits for responses again. The times of receiving of START and STOP messages on each processor (and the last response on the initiator) define the interval. The second method of building overlapping intervals involves circulation of a special token between processors, normally in a predefined topology, and the interval starts with one arrival of the token, and ends with its next arrival [10].

However, the overlapping intervals technique presents only a partial solution. Another problem that needs to be addressed by GVT algorithms is how to account for transient messages, the second part of the GVT definition. Transient messages are those that have been sent but have not been yet received. They must be accounted for by either the sender processor or the receiver processor, or both. The simplest solution is to use message acknowledgments, so that any message whose acknowledgments has not been received is deemed to be transient and its timestamps must be included in the GVT computation.

It is no coincidence that earlier GVT algorithms [12, 14, 19] often implemented the message acknowledgment scheme, for it is a simple solution. However, its primary drawback is that it almost doubles the total number of messages sent in the simulation, so the performance may deteriorate. A natural optimization is not to use separate acknowledgment messages, but to piggy-back acknowledgments in normal messages that carry events [10]. Still, such an optimization does not completely alleviate the problem.

Lin and Lazowska [16] proposed to use sequence numbers to reduce acknowledgment overhead. Messages sent from one processor to another are marked with consecutive, increasing sequence numbers. When GVT related information is needed, processors will notify neighbors of the latest sequence numbers they have seen. These sequence numbers tell which messages have been received, and which have not.

Another drawback of the message acknowledgment scheme is quite subtle. It is not a trivial task to find out the earliest timestamp among unacknowledged messages. Such an operation is not of constant time and may require the use of a priority queue.

**Two Cuts.** Mattern [17] realized that it is not necessary to select a common wall-clock time to compute GVT. The GVT value can be determined from a snapshot of a consistent global state on all processors. He proposed the notion of "cuts", which consist of points, one per processor, dividing the time line into the "past" and the "future". A consistent cut is the one in which no message travels from the future to the past. The

GVT can be computed from the local virtual times at cut points of a consistent cut and the set of transient messages (which are now defined as messages traveling from the past to the future).

Another, simpler solution, widely known as Mattern's GVT algorithm, attempts to construct two cuts. Luckily, neither cut needs to be consistent. The only requirement is that all messages sent before the first cut must be received before the second cut. The GVT estimate is now the minimum of the local virtual times at the cut points on the second cut, or the smallest timestamp of the messages sent after the first cut, whichever is smaller. Messages crossing the second cut from the future to the past can be ignored, since these messages are guaranteed to have a timestamp larger than the GVT value.

In its original form, Mattern's GVT algorithm uses token passing to construct the two cuts [17]. A vector clock, contained in the token being passed between processors, monitors the number of transient messages sent to every processor. The token can leave the current processor only after all messages destined to it have been received. Thus, the second cut can be built with only one round of token passing, but its creation may incur a delay on each processor.

Mattern proposed several variants to remove the vector clock or the delay on every processor [17]. In one, a scalar counter replaces the vector clock. However, now the second cut is not guaranteed to be done with one round of token passing; several rounds may be necessary. Mattern also presented another variant in which a single round is sufficient without the use of vector clock. However, it still requires tokens to be delayed on each processor.

Choes and Tropper [13] improved the original Mattern's GVT algorithm by using a scalar counter to track the number of messages sent before the first cut. When this counter reaches zero, no more messages originating from before the first cut are in transit, which signals the completion of the second cut. Even though multiple rounds may be needed, the authors observed that two rounds are sufficient in most cases.

There are two other GVT algorithms which do not appear similar to Mattern's GVT algorithm but in fact are based on the idea of constructing consistent cuts. In Tomlinson and Garg's GVT algorithm [22], simulation times instead of wall-clock times are used to schedule GVT computations. A cut point is the point at which a processor reaches a scheduled simulation time, and extra consideration must be taken to ensure a consistent cut. Bauer and Sporrer's GVT algorithm [11] identifies reports that form a consistent cut, and then uses these reports to derive a GVT estimate.

**Global Reduction.** Global Reduction is a simplification of Mattern's GVT algorithm. When all processors arrive at a synchronization point, a global reduction is performed on the number of transient messages that were sent and not received before the synchronization point. When this number becomes zero, it is apparent that no transient messages exist so that the information collected at the synchronization point can be used to compute the GVT estimate.

Perumalla and Fujomoto [18] presented a global reduction GVT algorithm which divides the time lines into bands. Bands are constructed in such a way that the messages sent during one band are guaranteed to be received in the current or next band. Thus, the boundaries of a band form two consecutive cuts, as in Mattern's GVT algorithm.

Steinman et al [21] described a synchronous GVT algorithm also based on global reduction. Srinivasan and Reynolds [20] developed yet another GVT algorithm that relies upon global reduction, but their algorithm requires hardware support.

**Other Algorithms.** There are some GVT algorithms that cannot be easily grouped into any of the three categories discussed so far. The pGVT algorithm [15] uses a GVT manager to monitor the progress of every processor and to compute the GVT based on information collected from processors. Processors are required to report to the GVT manager whenever they receive a straggler message. The Seven O'clock algorithm [23] assumes that the underlying network can always deliver events within a certain time window. Based on this assumption, a new notion called Network Atomic Operation is introduced which then extends Fujimoto and Hybinette's shared-memory GVT algorithm [28] to the distributed memory domain. A similar idea that enables bypassing message acknowledgment if the maximum transmission delay is known has been introduced in [14].

The Continuously Monitored GVT (CMGVT) algorithm allows processors to calculate GVT based on the local information constantly available to each processor. supplemented with the global information, such as the local virtual time of each processor and information about messages in transit, that is appended to simulation messages [24]. Hence, the algorithm works well when there is a lot of simulation message traffic and the communication is local, as was the case of the spatially explicit simulations considered in [24]. The

TABLE 2.1
*Comparison of Various GVT Algorithms*

| Authors | Idea | Ack | Vector | Channel | Scalability |
|---|---|---|---|---|---|
| Samadi [19] | Broadcast START and STOP messages to form overlapping intervals | Yes | No | Any | N/A |
| Bellenot [12] | Use message routing graph instead of broadcast | Yes | No | Any | 104 [26] |
| Das and Sarkar [14] | Optimize the computation for hypercube topology | No | No | Maximum Delay | N/A |
| Baldwin, Chung and Chung [10] | Pass a token to form overlapping intervals | Yes | No | FCFS | N/A |
| Lin and Lazowska [16] | Send valley messages to reduce acknowledgement traffic | Implicit | No | Any | N/A |
| Mattern [17] | Construct two cuts such that no transient messages sent before the first cut exist | No | Yes | Any | 12 [13] |
| Choe and Tropper [13] | Create multiple rounds of token passing to form the two cuts | No | No | Any | 12 [13] |
| Tomlinson and Garg [22] | Build consistent cuts by using TGVT events | No | Yes | Any | N/A |
| Bauer and Sporrer [11] | Identify pairs of reports that form a consistent cut | No | No | FCFS | N/A |
| Srinivasan and Reynolds [20] | Use hardware-based global reduction | No | No | Any | N/A |
| Steinman, Lee, Wilson, and Nicol [21] | Use global reduction | No | No | Any | 64 [21] |
| Perumalla and Fujimoto [18] | Use global reduction | No | No | Any | 16 [18] |
| D'Souza, Fan, and Wilsey [15] | Report stragglers to a GVT manager | Yes | No | Any | 2 [15] |
| Bauer, Yuan, Carothers, Yuksel, and Kalyanaraman [23] | Extend Fujimoto's shared-memory GVT algorithm with the notion of network atomic operations | No | No | Maximum Delay | 16 [23] |
| Deelman and Szymanski [24] | Use vector and matrix clocks to keep track of messages in transit | No | Yes | Any | 16 [24] |

disadvantage of this algorithm is the need of communicating a matrix of processor knowledge about messages in transit that is of the rank of the number of direct outgoing connections. Hence, this algorithm is scalable only in applications in which the out-degree of all nodes in the processor communication graph is independent of the graph size.

**Summary of Existing GVT Algorithms.** Table 1 summarizes the key ideas behind various GVT algorithms and their requirements, such as the need of acknowledgments, the use of vector clocks, and the type of the communication channels required. It also lists the highest number of processors on which each of them has been run and reported in the literature.

When considering requirements for a highly scalable GVT algorithm, we identified several properties that should be avoided as they limit scalability. First of those is the need for message acknowledgments that increase the network traffic and interfere with the transmission of normal messages. Lin and Lazowska's idea [16] eliminates the necessity of explicit acknowledgments, but at the expense of larger latency of GVT computation and of complex data structures to store messages. Second such property is the use of any vector whose size is linear with the number of processors. Finally, a truly scalable algorithm should not impose any special requirements on the communication channel except two basic ones, the First-Come-First-Served order of delivery and reliable (no-loss) delivery. These two requirements are satisfied by many popular communication libraries, such as MPI or TCP/IP.

Only a handful of existing algorithms meets all three requirements. Among them, Bauer and Sporrer's algorithm [11] has to determine pairs of consistent reports for every pair of processors that may communicate, making it rather difficult to scale. Hence, the two cut techniques, introduced in [13, 17] and global reduction [18, 21], are the only plausible candidates. However, all these approaches introduce scalability challenge in the way the second cut is constructed. Either some delay is incurred while processors wait for a certain condition to be satisfied, or multiple communication rounds are needed to complete the second cut. The origin of the

problem is that they all attempt to determine the completion of the second cut collectively by all processors. Motivated by this analysis, we propose an alternative, which constructs the second cut in a centralized way, thus avoiding the scalability problems that may arise when these approaches are used on a very large number of processors.

**3. Overview of Time Quantum GVT.** Two essential features distinguish Time Quantum GVT (TQ-GVT) from other GVT algorithms. First, TQ-GVT assigns the task of GVT computation to one processor, which is then referred to as the GVT master. The GVT master does not perform any simulation; its responsibilities include only collecting GVT reports, as well as calculating and distributing GVT. All other processors, called simulating processors, on the other hand, are not directly involved in the GVT computation. This approach is basically a centralized one; however, it differs from other centralized approaches in that the GVT master never initiates GVT calculations. Instead, the GVT master passively listens to GVT messages and takes actions only when they come. In this respect, TQ-GVT is similar to the pGVT algorithm [15], which, however, requires that simulating processors report to a dedicated processor every time a straggler arrives.

Second, TQ-GVT divides the wall-clock time into a continuous sequence of time quanta with equal width. Time quanta need not be precisely synchronized on different processors. Each processor maintains a counter indicating the index of the current time quantum and increases the counter at the end of the time quantum defined by its own hardware clock. Every message is marked with the index of the time quantum from which it is sent. The purpose of time quantum is to group messages, so if there is a way to track the number of transient messages from each time quantum, then, in computing GVT, only time quanta with messages in transit need to be taken into account.

Time quanta are similar to bands described in [18] in which, however, the division of the wall-clock time into bands is driven by the completion of the corresponding GVT computation. As a result, in the algorithm proposed in [18], processors cannot proceed to the next band if the GVT computation has not finished. In contrast, in TQ-GVT, the results or status of GVT computation have no impact on the division of wall-clock into time quanta.

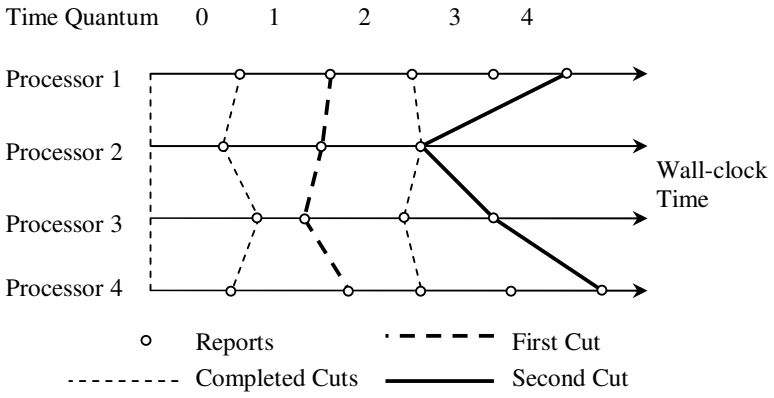



FIG. 3.1. *An Illustration of the Time Quantum GVT Algorithm*

Figure 1 illustrates the basic idea of TQ-GVT. Small circles indicate the wall-clock times at which processors advance to next time quantum and send out GVT reports. The GVT master constructs cuts from the reports stamped with the same time quantum index. If, for a particular time quantum, the reports from all processor have been received, then the cut is regarded as completed. The GVT master monitors the number of sent as well as the number of received messages within each time quantum, based on information carried in the GVT reports. If the two numbers associated with a given time quantum are equal, then all messages sent during this time quantum must have already been received. Hence, the algorithm is based on the Mattern's idea of two cut [17]: the first cut consists of reports for the latest time quantum that contains no transient messages, and the second cut consists of latest reports from every processor.

The use of an exclusive processor for GVT computation may appear an obstacle to scalability, since centralized approaches are usually difficult to scale. Nevertheless, one or more levels of intermediate GVT masters can be introduced, each of which keeps track of the number of transient messages in each time quantum for a subset of processors. These numbers must then be reported to the root GVT master, which in turn determines

whether or not there are still messages in transit from each time quantum and calculates the GVT accordingly. Empirically, in current hardware platforms, one GVT master can drive as many as 128 simulating processors, so an extra level of intermediate masters is needed to increase the number of simulating processors to 16,384. If this is not sufficient, more levels of GVT masters can be added. A very small percentage of processors, 129 out of 16,513, or merely 0.78 percent, will not be directly participating in the simulation. Hence, such a solution is suitable for clusters in which the numbers of processors involved in a computation are large. It should be noted that the use of such reduction network for conservative parallel discrete simulation was introduced already (see [6]), however, the reduction was applied to all simulation processors, unlike in our solution in which only GVT masters participate in reduction.

Two factors contribute to the low overhead of TQ-GVT. First, in TQ-GVT, GVT computation does not interfere with other simulation activities. Simulating processors are only engaged in processing events and transmitting and receiving messages. To support GVT computation, they just need to send report messages periodically, and to receive GVT messages as they come. If GVT messages do not come on time, simulating processors are never delayed or blocked, as in the case of some other algorithms, unless the delay is so large that it begins to interfere with fossil collection. As a result, only the GVT masters are involved in a significant amount of GVT computation. Since this is their sole task, the GVT masters can respond to incoming messages more swiftly than simulating processors which would have to switch between checking and receiving incoming messages and processing events. The cost of GVT computation for simulating processors is a non-blocking send of a report message at the end of each quantum and a non-blocking receive of the new GVT value if there is a GVT message.

Second, in TQ-GVT, different rounds of GVT computation are overlapped. One round does not need to be completed before the next round starts. For example, a processor can keep sending the GVT master a report message at the end of each time quantum, even if the reports from previous quanta have not reached the GVT master yet. The GVT master decides which reports to use based on the number of transient messages in each time quantum. Thus, this solution is insensitive to the large latency of the interconnection network often found in clusters, as well as to the local clock asynchrony that results in different processors reaching the synchronization point at different wall-clock times.

**4. Detailed Description of TQ-GVT.** TQ-GVT uses three types of messages. An event message, denoted by $E(tq, ts)$, is the carrier of a positive event or an anti-event, where $ts$ is the timestamp and $tq$ is the index of the time quantum from which the event was sent. A GVT message $G(gvt)$ simply contains the value of the new GVT estimate. A report message has the format of $R(i, tq, LVT, MVT, send, RECV[])$, where $i$ is the processor id, $tq$ is the index of the current time quantum, $LVT$ is the local virtual time, $MVT$ is the earliest event sent during the current time quantum, *send* is the count of messages sent out during the current time quantum, and $RECV[]$ is a vector of integers, each of which denotes the number of events received that were sent from the corresponding time quantum.

In Figure 2, lines 1-22 show the procedure executed on simulating processors and lines 23-37 the procedure of the GVT master. Lines 1-5 initialize variables needed by report messages. Lines 6-22 are the main loop of the simulation. Among them, lines 7-11 process one or more events and update MVT and LVT accordingly. Lines 12-13 check if a new GVT estimate is available. Lines 14-16 receive any incoming messages, and update $RECV[]$ and LVT accordingly. Lines 17-22 send a new report to the GVT master, reset variables and then advance to next time quantum.

For GVT master, lines 23-26 initialize several variables needed to compute the GVT. $TRANSIT[]$ represents the number of transient event messages for every time quantum, $LVT[]$ stores the local virtual time for each processor, and $MVT[]$ stores the smallest timestamp sent during each time quantum. The main simulation loop keeps receiving report messages until the end of simulation. For every received report, at lines 29-33, the GVT master updates the corresponding elements in $LVT[]$, $MVT[]$, and $TRANSIT[]$. At lines 34-35, the GVT master attempts to calculate a new GVT estimate as the minimum of the LVT of all processors and the MVT of all time quanta that still have event messages in transit. If the new GVT estimate is different from the old one, it will be broadcasted to every simulating processor.

In the above version of the TQ-GVT algorithm, the report message contains a vector of integers each of which denotes the number of received event messages indexed with the corresponding time quantum. At the minimum, received messages have to be reported only for the oldest time quantum active at this processor (i. e., the smallest $k$ such that $RECV[k] > 0$). With this solution, a delay once accumulated could not be decreased.

```
Simulating processor i:
   1.  tq=0
   2.  send=0
   3.  RECV[]=0
   4.  MVT=∞
   5.  LVT=0.0
   6.  while not end of simulation
   7.    process one or more events
   8.    for any message E(tq,ts) sent
   9.       MVT=min(MVT,ts)
   10.      send++
   11.   update LVT
   12.   if a GVT message G(gvt) is received
   13.     update the GVT value
   14.   if an event message E(tq',ts) is received
   15.     RECV[tq']++
   16.     if(ts<LVT)LVT=ts
   17.   if time for next quantum
   18.     send R(i,tq,LVT,MVT,send,RECV[])
   19.     send=0
   20.     RECV[]=0
   21.     MVT=∞
   22.     tq++;
GVT master:
   23. gvt=0.0
   24. TRANSIT[]=0
   25. LVT[]=0.0
   26. MVT[]=∞
   27. while not end of simulation
   28.   if R(i,tq,lvt,mvt,send,RECV[]) is received
   29.     LVT[i]=lvt
   30.     MVT[tq]=min(MVT[tq],mvt)
   31.     TRANSIT[i]+=send
   32.     for each j in RECV[]
   33.       TRANSIT[j]-=RECV[j]
   34.     gvt=min(LVT[i] for any i,
   35.           MVT[j] for any j such that TRANSIT[j]!=0)
   36.     if gvt changes
   37.       broadcast G(gvt)
```

FIG. 4.1. *TQ-GVT: Program for Simulating Processors and GVT Master*

Hence, in the actual implementation, we use a maximum length $k$ on this vector, so messages received from $k$ oldest active time quanta are reported. By doing so, the correctness of the algorithm is not changed; the only effect may be that the GVT will be more conservatively computed. This happens because at most k time quanta can be removed from consideration at the end of each time quantum by the GVT master. In our limited experiments, $k$ set to 2 or 4 gave the best performance. In summary, only a vector of size $k$ with each entry containing a number of received messages in the corresponding time quantum needs to be sent to the GVT master, making the length of report messages constant, regardless of how many processors are being used.

The code for intermediate level GVT masters is not presented here. The reason is simple: these GVT masters act as messengers that merely forward any messages they receive. They may perform some optimization, such as combing multiple report messages coming from different processors but with the same time quantum index into a single report message. The only effect of using intermediate GVT masters is the prolonged communication delay. However, as evident in the next section, TQ-GVT does not impose any limitation on message transmission delay, so the discussion on intermediate GVT masters is omitted.

**5. Correctness of TQ-GVT.** To prove the correctness of a GVT algorithm, a usual approach is to show that neither the transient message problem nor the simultaneous reporting problem exists [16, 30]. Here, a different approach will be used. Instead of proving that what TQ-GVT computes is a lower bound estimate of the GVT value according to the authentic definition of GVT, we will prove the correctness of TQ-GVT based on a "utilitarian" definition of GVT (similar technique was used in [28] for a different GVT algorithm). That is, we will show that the following Lemma holds.

**Lemma**. If an event message $m_1$ with timestamp $ts(m_1)$ is received after a GVT value $gvt$ is received, then $ts(m_1) \geq gvt$.

**Proof**. We prove this property by contradiction using induction. Hence, we assume that a processor $i_1$ sent at time quantum $s_1$ a message $m_1$ such that $ts(m_1) < gvt$. Let $j_1$ be the time quantum from which the last report message was sent by processor $i_1$ and received by the GVT master before the current $gvt$ was obtained. Let $LVT_{i,j}$ denote the local virtual time reported by processor $i$ at the end of time quantum $j$.

It cannot be that $ts(m_1) \geq LVT_{i_1,j_1}$, since $LVT_{i_1,j_1}$ is taken into account in computing $gvt$, so $LVT_{i_1,j_1} \geq gvt$. Neither it could be that $s_1 \leq j_1$, since then $ts(m_1)$ would be reflected in the MVT value corresponding to $s_1$, contradicting our assumption that $ts(m_1) < gvt$. Hence, $s_1 > j_1$ and $ts(m_1) < LVT_{i_1,j_1}$, so there must be another message $m_2$, sent by a different processor $i_2$, which caused a rollback on processor $i_1$, and this message timestamp satisfies the inequality $gvt > ts(m_1) > ts(m_2)$, as a rollback never affects the events with the same or earlier timestamps than the timestamp of the rollback message itself. From that it follows that message $m_2$ also satisfies $s_2 > j_2$, where $s_2, j_2$ are analogs of $s_1, j_1$.

By induction, let's assume that there is a message $m_k$, sent by processor $i_k$, such that $ts(m_k) < gvt$ and $s_k > j_k$, where $j_k$ denotes the latest time quantum on processor $i_k$ that is included in the current value of $gvt$ and $s_k$ is the time quantum at which message $m_k$ was sent. It cannot be that $ts(m_k) \geq LVT_{i_k,j_k}$, since $LVT_{ik,j_k}$ is taken into account in computing $gvt$, so $LVT_{i_k,j_k} \geq gvt$. Hence, $ts(m_k) < LVT_{i_k,j_k}$, so there must be another message $m_{k+1}$, sent by a different processor $i_{k+1}$, which caused a rollback on processor $i_k$, and this message timestamp satisfies the inequality $gvt > ts(m_k) > ts(m_{k+1})$. We define $s_{k+1}, j_{k+1}$ as analogs of $s_k, j_k$. For message $m_{k+1}$, it cannot be that $s_{k+1} \leq j_{k+1}$, since then $ts(m_{k+1})$ would be reflected in MVT value corresponding to $s_{k+1}$, contradicting our conclusion that $gvt > ts(m_{k+1})$.

Hence, by induction we conclude that our assumption that $ts(m_1) < gvt$ implies that there is an infinite sequence of messages with timestamps smaller than $gvt$, which contradicts the basic premise that each simulation can generate only a finite number of messages in the finite simulation time $gvt$.[]

The above proof makes no assumption on the delay of massage passing. Therefore, the use of intermediate GVT masters will not affect the correctness of TQ-GVT.

**6. Experimental Results.** TQ-GVT has been implemented in DSIM [31], a new distributed Time Warp simulator available freely at `http://www.cs.rpi.edu/ cheng3/dsim`. Although DSIM features several novel techniques that effectively reduce the overhead of Time Warp operations, we are convinced that TQ-GVT is the main thrust that enables DSIM to scale to at least 1,024 processors without any sign of loss of efficiency.

Two applications have been built on top of DSIM to test the scalability of the simulator, as well as of the new GVT algorithm. The first application is a simulation of the PHOLD benchmark, a synthetic workload generator proposed by Fujimoto [32]. In our version of the PHOLD benchmark, each event stays at an LP (Logical Process) for a time randomly chosen from the exponential distribution and then departs to one of four nearest neighbors randomly chosen. LPs are organized into a two dimensional grid. Strip partitioning is utilized, which allocates a continuous set of columns of LPs to the same processor. Although simple, the PHOLD model is difficult to parallelize for two reasons. First, there is no lookahead in it, so conservative protocols do not apply. Second, the event granularity is low, so the efficiency of parallelization is very sensitive to its overhead.

All the experiments were run on the Lemieux cluster at Pittsburgh Supercomputing Center consisting of 750 nodes, each with 4 AlphaServer processors, connected by a Quadrics interconnection network. The top part of Figure 3 shows the committed event processing rates of DSIM running the PHOLD benchmark on up to 1024 processors for about 50 sec of simulation time. Each simulating processor simulated 8 columns of LPs, with 8,192 LPs in each column, regardless of the total number of processors used, so the workloads on each processor remained constant, while the size of grid increased linearly with the number of processors. A number associated with each data point indicates the number of extra processors allocated exclusively to executing the TQ-GVT algorithm. It was determined empirically that each GVT master can drive as many as 128 processors. Starting from 256 processors, an extra level of up to 8 intermediate GVT masters were introduced.

When moving from a sequential simulator to a parallel one, there is a significant drop in the event rate for this and the other application shown at the lower part of Figure 3. It takes two to three parallel processors to match the performance of one sequential processor. This is caused by three factors. First, there is an overhead of memory and time used for storing and releasing the processed events in the parallel processor for use in case of rollbacks (but there cannot be any rollbacks with one parallel processors). Then, there is a larger footprint of the parallel program compared with the sequential one as the former contains code for rollbacks, communication with peers and the GVT master. Both of those factors limit the size of the simulation that a single parallel processor can run to a fraction of what the sequential processor can. Finally, there are (superfluous in this

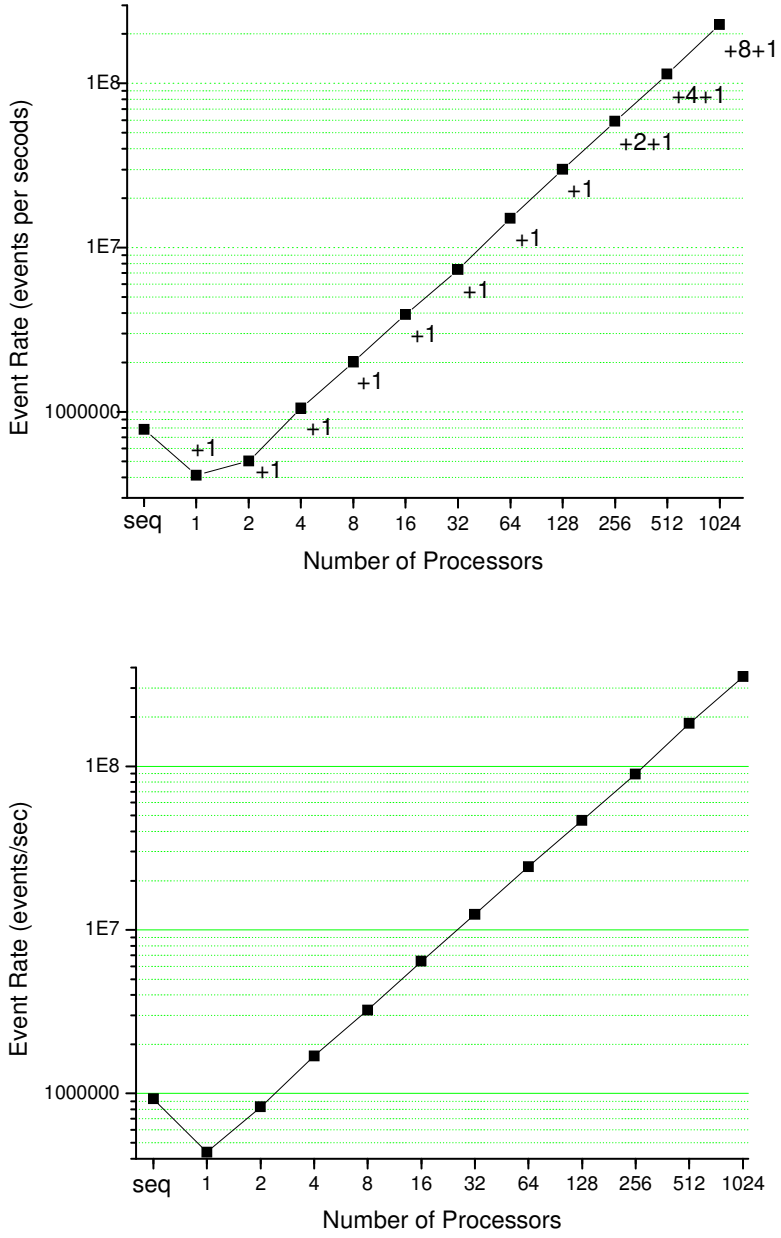



FIG. 6.1. *Event Processing Rates with the PHOLD (top) and Spiking Neural Network (bottom)*

case) interactions with the GVT master slowing the progress of a parallel processor. However, all these factors contribute a constant overhead per each parallel processor, so the parallel performance grows practically linearly with the number of parallel processors used.

For the largest simulation with 1024 processors, over 11 billion committed events were processed, over 3 million rollbacks executed and 250 GVT computations performed that required sending more than 250,000 GVT reports.

The top of Figure 4 depicts the numbers of remote (i. e. those sent between different processors) and GVT messages on a logarithmic scale. The number of remote messages increased linearly with the number of processors, since the amount of workload on each processor was fixed. The remote messages constituted 6.6% of all messages generated in the simulation. The number of GVT messages increased linearly too, except that changing from 32 processors to 64 processors caused a sudden drop because we changed the width of time quanta from 0.1 to 0.2 second at this point. This parameter controls how frequently GVT computation

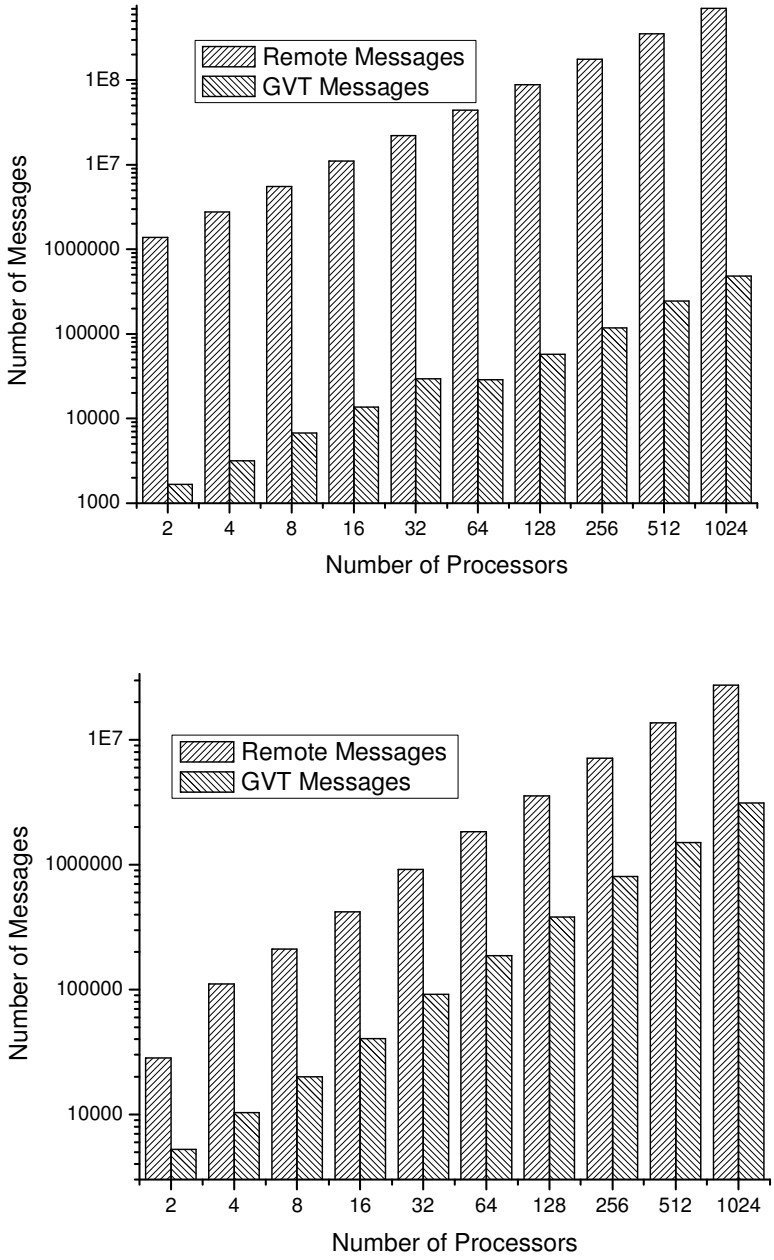



FIG. 6.2. *GVT and Remote Messages for PHOLD (top) and Spiking Neural Network (bottom)*

is executed, so it impacts overall performance. A large time quantum increases memory consumption of the simulation. A small value increases the bandwidth needed for GVT reports and messages. The best value for our simulations was estimated empirically and justified by the following reasoning. The peak event rate per Lemieux node approaches one million events per second, so in a time quantum of 0.1 second a processor accumulates 0.1 million events. Each event occupies between 100 and 1000 bytes, so the total accumulated memory per quantum is anywhere from 10MB to 100MB, well below the memory limit of the machine. Further studies are needed on techniques for optimally selecting this parameter in more general cases. However, the general analysis is simple and points out that the overhead of the TQ-GVT is practically linear with the number of processors used.

Let $a$ be the factor that defines the memory occupied by the events processed in a time unit. For each application, $a$ is a constant defined by the event processing rate and an average event size. Let $t_{\text{intergvt}}$ denote the time between two subsequent GVT computations. Then, the memory consumed by the application is

$m_0 + a * t_{\text{intergvt}}$, where $m_0$ is the footprint of the parallel program and its static data at each processor. Consequently, if each processor has the available memory $M$, then $t^{\max}_{\text{intergvt}}$ is limited to $t^{\max}_{\text{intergvt}} = (M - m_0)/a$. On the other hand, the overhead of the TQ-GVT is constant in time for each GVT computation round and for each processor, as it requires sending and receiving a message of a constant size. We will denote it by $t_{\text{gvt}}$. However, for simulation we can use only $n - n_m$ processors, where $n$ denotes the total number of processors and $n_m$ denotes the number of processors used as GVT masters. By definition, $n_m = \sum_{i=1}^{\lceil \log_p(n-n_m) \rceil} \lceil \frac{n-n_m}{p^i} \rceil$, where $p$ is the maximum number of processors that a single GVT master can handle (so, $p = 128$ in our experiments). Hence, the number of GVT masters is bounded by $n_m \leq \sum_{i=1}^{\lceil \log_p(n) \rceil} \frac{n-n_m}{p^i} + \lceil \log_p(n) \rceil < \frac{n-n_m}{p-1} + \lceil \log_p(n) \rceil$. Hence, the overall overhead of TQ-GVT is less than

$$\frac{n_m}{n} \frac{t_{\text{gvt}}}{t_{\text{intergvt}} + t_{\text{gvt}}} \leq \left( \frac{1}{p} + \frac{\log_p(n) + 1}{n} \right) \frac{a * t_{\text{gvt}}}{M - m_0}. \tag{6.1}$$

Since $(\log_p(n) + 1)/n$ tends to zero as $n$ tends to infinity, this overhead bound is nearly constant, slowly decreasing with an increase of $n$ to a constant $\frac{t_{\text{gvt}}}{p} \frac{a}{M-m_0}$. In this expression, the first fraction is dependent on the communication subsystem of the parallel machine used ($t_{\text{gvt}}$ is defined by the time it takes to sent a GVT report and $p$ is limited by the number of simultaneous GVT reports which a single processor can process) while the second fraction represents the computational capabilities of each processor ($a$ is defined by the speed of event processing and $M - m_0$ is limited by the size of the memory available on each processor).

The bottom parts of Figures 3 and 4 show that DSIM, as well as TQ-GVT within it, worked equally well for a realistic application, the simulation of Spiking Neural Networks. In this simulation, a network of artificial spiking neurons serves as a computationally efficient model of a network of neurons with membrane currents governed by voltage-gated ionic conductances. In our simulation, all the state variables of a model neuron are computed analytically from a new set of initial conditions. Computations are performed only when an event is executed, so the total computation time is proportional to the number of events generated and independent of the number of neurons simulated. Each spike event in a neuron creates weighted synaptic input events for all neurons connected with the spiking one, each event scheduled with its specific time delay.

The simplest model of this kind, called IntFire1 [33], is a leaky integrator that treats input events as weighted delta functions. When executing an input event of weight $w$, an IntFire1 neuron increases its "membrane potential" state $m$ instantaneously by an amount equal to $w$ and thereafter resumes its state decay toward 0 with time constant $\tau_m$.

We have implemented a model modeling the behavior of a biological neuron more closely than IntFire1 that is known as the IntFire2 mechanism [33]. Unlike IntFire1 model, its "membrane potential" state $m$ integrates a net synaptic current $i$. An event executed on an IntFire2 neuron makes the synaptic current jump by an amount equal to the synaptic weight, after which $i$ continues to decay toward a steady level $i_b$ with its own time constant $\tau_s$, where $\tau_s > \tau_m$. Thus a single input event produces a gradual change in $m$ with a delayed peak, and neuron firing does not obliterate all traces of prior synaptic activation.

In our simulations, we set $\tau_s = 2\tau_m$ to simplify the calculation of the solution. It should be noted that this change lowered the event granularity which in turn made the parallelization more difficult. Again, the workloads on individual processors were constant by allocating $512 * 512 = 262,144$ neurons to each processor. Neurons were also organized into a two-dimensional grid and each pair of neurons with a distance no greater than 4 has a dendrite connecting them. Therefore, each neuron was connected to 50 other neurons. On 1,024 processors there were totally $262,144 * 1,024 = 268,435,456$ neurons simulated, with an event processing rate of more than 3 hundred million committed events per second. The performance curve was similar to that of PHOLD simulation. The only difference was that parallel execution on two processors exhibited smaller drop in performance when compared to the sequential execution, thanks to a technique used to dramatically reduce the number of remote messages for spiking neural network simulation. The technique is based on a simple observation that if a firing neuron is connected to many neurons that belong to a different processor, it can use a proxy neuron placed at the remote processor and communicate just with the proxy neuron which then communicate to all neurons connected at the remote processor that are connected to the firing neuron. This technique does not reduce the number of events processed, but does effectively reduce the inter-processor traffic by compressing many remote messages into one.

For the largest simulation with 1024 processors, over 53 billions committed events were processed in 151 seconds of the running time. There were over a million rollbacks executed and over 1,500 GVT computations

performed that required sending more than 1,500,000 GVT reports. The bottom of Figure 4 again confirms the low communication overhead of TQ-GVT.

**7. Conclusion and Future Work.** TQ-GVT demonstrated strong performance on more than one thousand processors and its design does not contain any scalability obstacles. With the invention of TQ-GVT, we believe that one major obstacle to the general applicability of Time Warp to large supercomputers and clusters has been solved. We plan to apply Time Warp to current modern super-scale parallel computers consisting of tens or hundreds of thousands of processors, to simulate realistically complex models of great interest to science, such as spiking neural networks.

REFERENCES

[1] R. M. Fujimoto, *Parallel discrete event simulation*, Communications of the ACM, 33(10) (1990) pp. 30–53.
[2] K. M. Chandy, and J. Misra, *Distributed simulation: a case study in design and verification of distributed programs*, IEEE Trans. on Software Engineering, SE-5(5) (1979) pp. 440–452.
[3] R. E. Bryant, *Simulation of Packet Communication Architecture Computer Systems*. Massachusetts Institute of Technology. Cambridge, MA, 1977.
[4] D. R. Jefferson, *Virtual time*, ACM Trans. on Programming Languages and Systems, 7(3) (1985) pp. 404–425.
[5] K. M. Chandy, and R. Sherman, *Conditional event approach to distributed simulation*, in Simulation Multiconf. Distributed Simulation, Tampa, FL (1989) pp. 93–99.
[6] R. Fujimoto, T. McLean, K. Perumalla, and I. Tacic, *Design of high performance RTI software*, in 4th Workshop Parallel and Distributed Simulation and Real-Time Applications, San Francisco, CA (2000) pp. 89–96.
[7] G. Chen, and B. K. Szymanski, *Lookback: a new way of exploiting parallelism in discrete event simulation*, in 16th Workshop Parallel and Distributed Simulation, Washington, DC (2002) pp. 153–162.
[8] G. Chen, and B. K. Szymanski, *Four types of lookback*, in 17th Workshop Parallel and Distributed Simulation, San Diego, CA (2003) pp. 3–10.
[9] G. Chen, G. and B. K. Szymanski, *Parallel Queuing Network Simulation with Lookback Based Protocols*, in European Modeling and Simulation Symp., Barcelona, Spain (2006) pp. 545–551.
[10] R. Baldwin, M. J. Chung, and Y. Chung, *Overlapping window algorithm for computing GVT in Time Warp*, in 11th Intern. Conf. Distributed Computing Systems, Arlington, TX (1991) pp. 534–541.
[11] H. Bauer, and C. Sporrer, *Distributed logic simulation and an approach to asynchronous GVT-calculation*, in 6th Workshop Parallel and Distributed Simulation, Newport Beach, CA (1992) pp. 205–208.
[12] S. Bellenot, *Global Virtual Time Algorithms*, in SCS Multiconf. Distributed Simulation, San Diego, CA (1990) pp. 122-127.
[13] M. Choe, and C. Tropper, *An Efficient GVT Computation Using Snapshots*, in Conf. Computer Simulation Methods and Applications, Orlando, FL,(1998) pp. 33–43.
[14] S. K. Das, and F. Sarkar, *A hypercube algorithm for GVT computation and its application in optimistic parallel simulation*, in Simulation Symp., Phoenix, AZ (1995) pp. 51–60.
[15] L. M. D'Souza, X. Fan, and P. A. Wilsey, *pGVT: an algorithm for accurate GVT estimation*, in 8th Workshop Parallel and Distributed Simulation, Edinburgh, UK (1994) pp. 102–109.
[16] Y.-B. Lin, and E. D. Lazowska, *Determining the global virtual time in a distributed simulation*, in Intern. Conf. Parallel Processing, Boston, MA (1990) pp. 201–209.
[17] F. Mattern, *Efficient algorithms for distributed snapshots and global virtual time approximation*, J. Parallel and Distributed Computing 18(4) (1993) pp. 423–434.
[18] K. Perumalla, and R. Fujimoto, *Virtual time synchronization over unreliable network transport*, in 15th Workshop Parallel and Distributed Simulation, Lake Arrowehead, CA (2001) pp. 129–136. .
[19] B. Samadi, *Distributed Simulation, Algorithms and Performance Analysis*, Computer Science Department, University of California, Los Angeles, CA, 1985.
[20] S. Srinivasan, and P. F. Reynolds, Jr, *Non-interfering GVT computation via asynchronous global reductions*, in Winter Simulation Conf., Los Angeles, CA (1993) pp. 740–749.
[21] J. S. Steinman, C. A. Lee, L. F. Wilson, and D. M. Nicol *Global virtual time and distributed synchronization*, in 9th Workshop Parallel and Distributed Simulation, Lake Placid, NY (1995) pp. 139–148.
[22] A. I. Tomlinson, and V. K. Garg, *An algorithm for minimally latent global virtual time*, in Workshop on Parallel and Distributed Simulation, San Diego, CA (1993) pp. 35–42.
[23] D. Bauer, G. Yaun, C.D. Carothers, M. Yuksel, and S. Kalyanaraman, *Seven-O.Clock: A New Distributed GVT Algorithm Using Network Atomic Operations*, in 19th Workshop Principles of Advanced and Distributed Simulation, Monterey, CA (2005) pp. 39–48.
[24] E. Deelman, and B. K. Szymanski, *Continuously Monitored Global Virtual Time*, in Intern. Conf. Parallel and Distributed Processing Techniques and Applications, Las Vegas, NV (1997) pp. 1–10.
[25] D. M. Nicol, and P. Heidelberger, *Comparative study of parallel algorithms for simulating continuous time Markov chains*, ACM Trans. Modeling and Computer Simulation, 5(4) (1995) pp. 326–354.
[26] F. Wieland, F., L. Hawley, A. Feinberg, M. Di Loreto, L. Blume, P. Reiher, B. Beckman, P. Hontalas, S. Bellenot, and D. Jefferson, *Distributed combat simulation and time warp. The model and its performance*, in SCS Multiconf. Distributed Simulation, Tampa, FL (1989) pp. 14–20.
[27] D. Jefferson, B. Beckman, F. Wieland, L. Blume, M. DiLoreto, P. Hontalas, P. Laroche, K. Sturdevant, J. Tupman, V. Warren, J. Wedel, H. Younger, and S. Bellenot, *Distributed simulation and the Time Warp Operating System*, Operating Systems Review, 21(5) (1987) pp. 77–93.

[28] R. M. Fujimoto, and M. Hybinette, *Computing global virtual time in shared-memory multiprocessors*, ACM Trans. Modeling and Computer Simulation, 7(4) (1997) pp. 425–446.
[29] Z. Xiao, F. Gomes, B. Unger, and J. Cleary, *A fast asynchronous GVT algorithm for shared memory multiprocessor architectures*, in 9th Workshop Parallel and Distributed Simulation, Lake Placid, NY (1995) pp. 203–208.
[30] R. M. Fujimoto, *Parallel and Distributed Simulation Systems*, Wiley-Interscience Publishers, New York, NY, 1999.
[31] G. Chen, and B. K. Szymanski, *DSIM: Scaling Time Warp to 1,033 Processors*, in Winter Simulation Conf., Orlando, FL (2005) pp. 346–355.
[32] R. M. Fujimoto, *Performance of time warp under synthetic workloads*, in SCS Multiconf. Distributed Simulation, San Diego, CA (1990) pp. 23–28.
[33] M. L. Hines and N. T. Carnevale, *Discrete event simulation in the NEURON environment*, Neurocomputing, 58-60 (2004) pp. 1117–1122.

*Edited by:* Carl Tropper
*Received:* June 19th, 2007
*Accepted:* October 27th, 2007


Scalable Computing: Practice and Experience
Volume 8, Number 4, pp. 437–438. http://www.scpe.org
ISSN 1895-1767


# BOOK REWIES

EDITED BY SHAHRAM RAHIMI AND DOROTHY BOLLMAN

*Languages and Machines: An Introduction to the Theory of Computer Science*
Third Edition
by Thomas A. Sudkamp
Addison Wesley

This is a textbook that introduces computer science theory, using formal abstract mathematics, for Junior and Senior computer science students. The third edition, to summarize the author, adds examples, expands the selection of topics, and provides more flexibility to the instructor in the design of a course.

The book is in five parts: (I) Foundations; (II) Grammars, Automata, and Languages; (III) Computability; (IV) Computational Complexity; and, (V) Deterministic Parsing. Many exercises are provided throughout to assist the student in understanding the material.

Chet Langin

*Reconfigurable Computing. Accelerating Computation with Field-Programmable Gate Arrays*
by Maya B. Gokhale and Paul S. Graham
Springer, 2005, 238 pp.
ISBN-13 978-0387-26105-8, $87.20

Reconfigurable Computing Accelerating Computation with Field-Programmable Gate Arrays is an expository and easy to digest book. The authors are recognized leaders with many years of experience on the field of reconfigurable computing. The book is written so that non-specialists can understand the principles, techniques and algorithms. Each chapter has many excellent references for interested readers. It surveys methods, algorithms, programming languages and applications targeted to reconfigurable computing.

Automatic generation of parallel code from a sequential program on conventional micro-processor architectures remains an open problem. Nevertheless, a wide range of computationally intensive applications have benefited from many tools developed to tackle such a problem. For RC, it is even a much harder problem (perhaps 10x and up) and intense research is being devoted to make RC a common-place practical tool. The aim of the authors is threefold. First, guide the readers to know current issues on HLL for RC. Second, help the readers understand the intricate process of algorithmic-to-hardware compilation. And third, show that, even though this process is painful, if the application is suitable for RC the gains in performance are huge.

The book is divided into two parts. The first part contains four chapters about reconfigurable computing and languages. Chapter 1 presents an introduction of RC, contrasting conventional fixed instruction microprocessors with RC architectures. This chapter also contains comprehensive reference material for further reading. Chapter 2 introduces reconfigurable logic devices by explaining the basic architecture and configuration of FPGAs. Chapter 3 deals with RC systems by discussing how parallel processing is achieved on reconfigurable computers and also gives a survey of RC systems today. Then, in chapter 4, languages, compilation, debugging and their related manual vs. automatic issues are discussed.

The second part of the book comprises five chapters about applications of RC. Chapter 5 and 6 discuss digital signal and image processing applications. Chapter 7 covers the application of RC to secure network communications. The aim of Chapter 8 is to discuss some important bioinformatics applications for which RC is a good candidate, their algorithmic problems and hardware implementations. Finally, Chapter 9 covers two applications of reconfigurable supercomputers. The first one is a simulation of radiative heat transfer and the second one models large urban road traffic.

This book is neither a technical nor a text book, but in the opinion of this reviewer, it is an excellent state-of-the-art review of RC and would be a worthwhile acquisition by anyone seriously considering speeding

up a specific application. On the downside, it is somewhat disappointing that the book does not contain more information about HLL tools that could be used to help close the gap between traditional HPC community and the raw computing power of RC.

Edusmildo Orozco,
*Department of Computer Science, University Of Puerto Rico at Rio Piedras.*

## AIMS AND SCOPE

The area of scalable computing has matured and reached a point where new issues and trends require a professional forum. SCPE will provide this avenue by publishing original refereed papers that address the present as well as the future of parallel and distributed computing. The journal will focus on algorithm development, implementation and execution on real-world parallel architectures, and application of parallel and distributed computing to the solution of real-life problems. Of particular interest are:

**Expressiveness:**

- high level languages,
- object oriented techniques,
- compiler technology for parallel computing,
- implementation techniques and their efficiency.

**System engineering:**

- programming environments,
- debugging tools,
- software libraries.

**Performance:**

- performance measurement: metrics, evaluation, visualization,
- performance improvement: resource allocation and scheduling, I/O, network throughput.

**Applications:**

- database,
- control systems,
- embedded systems,
- fault tolerance,
- industrial and business,
- real-time,
- scientific computing,
- visualization.

**Future:**

- limitations of current approaches,
- engineering trends and their consequences,
- novel parallel architectures.

Taking into account the extremely rapid pace of changes in the field SCPE is committed to fast turnaround of papers and a short publication time of accepted papers.

## INSTRUCTIONS FOR CONTRIBUTORS

Proposals of Special Issues should be submitted to the editor-in-chief.

The language of the journal is English. SCPE publishes three categories of papers: overview papers, research papers and short communications. Electronic submissions are preferred. Overview papers and short communications should be submitted to the editor-in-chief. Research papers should be submitted to the editor whose research interests match the subject of the paper most closely. The list of editors' research interests can be found at the journal WWW site (`http://www.scpe.org`). Each paper appropriate to the journal will be refereed by a minimum of two referees.

There is no a priori limit on the length of overview papers. Research papers should be limited to approximately 20 pages, while short communications should not exceed 5 pages. A 50–100 word abstract should be included.

Upon acceptance the authors will be asked to transfer copyright of the article to the publisher. The authors will be required to prepare the text in LaTeX 2$_\varepsilon$ using the journal document class file (based on the SIAM's `siamltex.clo` document class, available at the journal WWW site). Figures must be prepared in encapsulated PostScript and appropriately incorporated into the text. The bibliography should be formatted using the SIAM convention. Detailed instructions for the Authors are available on the SCPE WWW site at `http://www.scpe.org`.

Contributions are accepted for review on the understanding that the same work has not been published and that it is not being considered for publication elsewhere. Technical reports can be submitted. Substantially revised versions of papers published in not easily accessible conference proceedings can also be submitted. The editor-in-chief should be notified at the time of submission and the author is responsible for obtaining the necessary copyright releases for all copyrighted material.