George Almási
Călin Caşcaval
Peng Wu (Eds.)

Languages and Compilers for Parallel Computing

LNCS 4382

19th International Workshop, LCPC 2006
New Orleans, LA, USA, November 2006
Revised Papers

Springer

George Almási Călin Caşcaval
Peng Wu (Eds.)

Languages and Compilers for Parallel Computing

19th International Workshop, LCPC 2006
New Orleans, LA, USA, November 2-4, 2006
Revised Papers

Springer

Volume Editors

George Almási
Călin Caşcaval
Peng Wu

IBM Research Division
Thomas J. Watson Research Center
Yorktown Heights, New York 10598
E-mail: {gheorghe, cascaval, pengwu}@us.ibm.com

Library of Congress Control Number: 2007926757

CR Subject Classification (1998): D.3, D.1.3, F.1.2, B.2.1, C.2.4, C.2, E.1, D.4

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

ISSN 0302-9743
ISBN-10 3-540-72520-2 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-72520-6 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12063598 06/3180 5 4 3 2 1 0

Lecture Notes in Computer Science 4382

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

Preface

The 19th Workshop on Languages and Compilers for Parallel Computing was held in November 2006 in New Orleans, Louisiana USA. More than 40 researchers from around the world gathered together to present their latest results and to exchange ideas on topics ranging from parallel programming models, code generation, compilation techniques, parallel data structure and parallel execution models, to register allocation and memory management in parallel environments.

Out of the 49 paper submissions, the Program Committee, with the help of external reviewers, selected 24 papers for presentation at the workshop. Each paper had at least three reviews and was extensively discussed in the committee meeting. The papers were presented in 30-minute sessions at the workshop. One of the selected papers, while still included in the proceedings, was not presented because of an unfortunate visa problem that prevented the authors from attending the workshop.

We were fortunate to have two outstanding keynote addresses at LCPC 2006, both from UC Berkeley. Kathy Yelick presented "Compilation Techniques for Partitioned Global Address Space Languages." In this keynote she discussed the issues in developing programming models for large-scale parallel machines and clusters, and how PGAS languages compare to languages emerging from the DARPA HPCS program. She also presented compiler analysis and optimization techniques developed in the context of UPC and Titanium source-to-source compilers for parallel program and communication optimizations.

David Patterson's keynote focused on the "Berkeley View: A New Framework and a New Platform for Parallel Research." He summarized trends in architecture design and application development and he discussed how these will affect the process of developing system software for parallel machines, including compilers and libraries. He also presented the Research Accelerator for Multiple Processors (RAMP), an effort to develop a flexible, scalable and economical FPGA-based platform for parallel architecture and programming systems research. Summaries and slides of the keynotes and the program are available from the workshop Web site http://www.lcpcworkshop.org.

The success of the LCPC 2006 workshop would not have been possible without help from many people. We would like to thank the Program Committee members for their time and effort in reviewing papers. We wish to thank Gerald Baumgartner, J. Ramanujam, and P. Sadayappan for being wonderful hosts. The LCPC Steering Committee, especially David Padua, provided continuous support and encouragement. And finally, we would like to thank all the authors who submitted papers to LCPC 2006.

March 2007

Gheorghe Almási
Călin Caşcaval
Peng Wu

Organization

Steering Committee

Utpal Banerjee	Intel Corporation
David Gelernter	Yale University
Alex Nicolau	University of California, Irvine
David Padua	University of Illinois, Urbana-Champaign

Organizing Committee

Program Co-chairs	Gheorghe Almási, IBM Research
	Călin Caşcaval, IBM Research
	Peng Wu, IBM Research
Local Co-chairs	Gerald Baumgartner, Louisiana State University
	J. Ramanujam, Louisiana State University
	P. Sadayappan, Ohio State University

Program Committee

Vikram Adve	University of Illinois at Urbana-Champaign
Gheorghe Almási	IBM Research
Eduard Ayguad	Universitat de Politècnica de Catalunya
Gerald Baumgartner	Louisiana State University
Călin Caşcaval	IBM Research
Rudolf Eigenmann	Purdue University
Maria-Jesus Garzaran	University of Illinois at Urbana-Champaign
Zhiyuan Li	Purdue University
Sam Midkiff	Purdue University
Paul Petersen	Intel Corp.
J. Ramanujam	Louisiana State University
P. Sadayappan	Ohio State University
Peng Wu	IBM Research

Printing: Mercedes-Druck, Berlin
Binding: Stein + Lehmann, Berlin

Lecture Notes in Computer Science

For information about Vols. 1–4382

please contact your bookseller or Springer

Vol. 4515: M. Naor (Ed.), Advances in Cryptology - EUROCRYPT 2007. XIII, 591 pages. 2007.

Vol. 4510: P. Van Hentenryck, L. Wolsey (Eds.), Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems. X, 391 pages. 2007.

Vol. 4506: D. Zeng, I. Gotham, K. Komatsu, C. Lynch, M. Thurmond, D. Madigan, B. Lober, J. Kvach, H. Chen (Eds.), Intelligence and Security Informatics: Biosurveillance. XI, 234 pages. 2007.

Vol. 4504: J. Huang, R. Kowalczyk, Z. Maamar, D. Martin, I. Müller, S. Stoutenburg, K.P. Sycara (Eds.), Service-Oriented Computing: Agents, Semantics, and Engineering. X, 175 pages. 2007.

Vol. 4493: D. Liu, S. Fei, Z. Hou, H. Zhang, C. Sun (Eds.), Advances in Neural Networks – ISNN 2007, Part III. XXVI, 1215 pages. 2007.

Vol. 4492: D. Liu, S. Fei, Z. Hou, H. Zhang, C. Sun (Eds.), Advances in Neural Networks – ISNN 2007, Part II. XXVII, 1321 pages. 2007.

Vol. 4491: D. Liu, S. Fei, Z.-G. Hou, H. Zhang, C. Sun (Eds.), Advances in Neural Networks – ISNN 2007, Part I. LIV, 1365 pages. 2007.

Vol. 4486: M. Bernardo, J. Hillston (Eds.), Formal Methods for Performance Evaluation. VII, 469 pages. 2007.

Vol. 4484: J.-Y. Cai, S.B. Cooper, H. Zhu (Eds.), Theory and Applications of Models of Computation. XIII, 772 pages. 2007.

Vol. 4483: C. Baral, G. Brewka, J. Schlipf (Eds.), Logic Programming and Nonmonotonic Reasoning. IX, 327 pages. 2007. (Sublibrary LNAI).

Vol. 4482: A. An, J. Stefanowski, S. Ramanna, C.J. Butz, W. Pedrycz, G. Wang (Eds.), Rough Sets, Fuzzy Sets, Data Mining and Granular Computing. XIV, 585 pages. 2007. (Sublibrary LNAI).

Vol. 4481: J. Yao, P. Lingras, W.-Z. Wu, M. Szczuka, N.J. Cercone, D. Ślęzak (Eds.), Rough Sets and Knowledge Technology. XIV, 576 pages. 2007. (Sublibrary LNAI).

Vol. 4480: A. LaMarca, M. Langheinrich, K.N. Truong (Eds.), Pervasive Computing. XIII, 369 pages. 2007.

Vol. 4479: I.F. Akyildiz, R. Sivakumar, E. Ekici, J.C.d. Oliveira, J. McNair (Eds.), NETWORKING 2007. Ad Hoc and Sensor Networks, Wireless Networks, Next Generation Internet. XXVII, 1252 pages. 2007.

Vol. 4472: M. Haindl, J. Kittler, F. Roli (Eds.), Multiple Classifier Systems. XI, 524 pages. 2007.

Vol. 4471: P. Cesar, K. Chorianopoulos, J.F. Jensen (Eds.), Interactive TV: a Shared Experience. XIII, 236 pages. 2007.

Vol. 4470: Q. Wang, D. Pfahl, D.M. Raffo (Eds.), Software Process Dynamics and Agility. XI, 346 pages. 2007.

Vol. 4464: E. Dawson, D.S. Wong (Eds.), Information Security Practice and Experience. XIII, 361 pages. 2007.

Vol. 4463: I. Măndoiu, A. Zelikovsky (Eds.), Bioinformatics Research and Applications. XV, 653 pages. 2007. (Sublibrary LNBI).

Vol. 4462: D. Sauveron, K. Markantonakis, A. Bilas, J.-J. Quisquater (Eds.), Information Security Theory and Practices. XII, 255 pages. 2007.

Vol. 4459: C. Cérin, K.-C. Li (Eds.), Advances in Grid and Pervasive Computing. XVI, 759 pages. 2007.

Vol. 4453: T. Speed, H. Huang (Eds.), Research in Computational Molecular Biology. XVI, 550 pages. 2007. (Sublibrary LNBI).

Vol. 4452: M. Fasli, O. Shehory (Eds.), Agent-Mediated Electronic Commerce. VIII, 249 pages. 2007. (Sublibrary LNAI).

Vol. 4451: T.S. Huang, A. Nijholt, M. Pantic, A. Pentland (Eds.), Artifical Intelligence for Human Computing. XVI, 359 pages. 2007. (Sublibrary LNAI).

Vol. 4450: T. Okamoto, X. Wang (Eds.), Public Key Cryptography – PKC 2007. XIII, 491 pages. 2007.

Vol. 4448: M. Giacobini et al. (Ed.), Applications of Evolutionary Computing. XXIII, 755 pages. 2007.

Vol. 4447: E. Marchiori, J.H. Moore, J.C. Rajapakse (Eds.), Evolutionary Computation,Machine Learning and Data Mining in Bioinformatics. XI, 302 pages. 2007.

Vol. 4446: C. Cotta, J. van Hemert (Eds.), Evolutionary Computation in Combinatorial Optimization. XII, 241 pages. 2007.

Vol. 4445: M. Ebner, M. O'Neill, A. Ekárt, L. Vanneschi, A.I. Esparcia-Alcázar (Eds.), Genetic Programming. XI, 382 pages. 2007.

Vol. 4444: T. Reps, M. Sagiv, J. Bauer (Eds.), Program Analysis and Compilation, Theory and Practice. X, 361 pages. 2007.

Vol. 4443: R. Kotagiri, P.R. Krishna, M. Mohania, E. Nantajeewarawat (Eds.), Advances in Databases: Concepts, Systems and Applications. XXI, 1126 pages. 2007.

Vol. 4440: B. Liblit, Cooperative Bug Isolation. XV, 101 pages. 2007.

Vol. 4439: W. Abramowicz (Ed.), Business Information Systems. XV, 654 pages. 2007.

Vol. 4438: L. Maicher, A. Sigel, L.M. Garshol (Eds.), Leveraging the Semantics of Topic Maps. X, 257 pages. 2007. (Sublibrary LNAI).

Vol. 4433: E. Şahin, W.M. Spears, A.F.T. Winfield (Eds.), Swarm Robotics. XII, 221 pages. 2007.

Vol. 4432: B. Beliczynski, A. Dzielinski, M. Iwanowski, B. Ribeiro (Eds.), Adaptive and Natural Computing Algorithms, Part II. XXVI, 761 pages. 2007.

Vol. 4431: B. Beliczynski, A. Dzielinski, M. Iwanowski, B. Ribeiro (Eds.), Adaptive and Natural Computing Algorithms, Part I. XXV, 851 pages. 2007.

Vol. 4430: C.C. Yang, D. Zeng, M. Chau, K. Chang, Q. Yang, X. Cheng, J. Wang, F.-Y. Wang, H. Chen (Eds.), Intelligence and Security Informatics. XII, 330 pages. 2007.

Vol. 4429: R. Lu, J.H. Siekmann, C. Ullrich (Eds.), Cognitive Systems. X, 161 pages. 2007. (Sublibrary LNAI).

Vol. 4427: S. Uhlig, K. Papagiannaki, O. Bonaventure (Eds.), Passive and Active Network Measurement. XI, 274 pages. 2007.

Vol. 4426: Z.-H. Zhou, H. Li, Q. Yang (Eds.), Advances in Knowledge Discovery and Data Mining. XXV, 1161 pages. 2007. (Sublibrary LNAI).

Vol. 4425: G. Amati, C. Carpineto, G. Romano (Eds.), Advances in Information Retrieval. XIX, 759 pages. 2007.

Vol. 4424: O. Grumberg, M. Huth (Eds.), Tools and Algorithms for the Construction and Analysis of Systems. XX, 738 pages. 2007.

Vol. 4423: H. Seidl (Ed.), Foundations of Software Science and Computational Structures. XVI, 379 pages. 2007.

Vol. 4422: M.B. Dwyer, A. Lopes (Eds.), Fundamental Approaches to Software Engineering. XV, 440 pages. 2007.

Vol. 4421: R. De Nicola (Ed.), Programming Languages and Systems. XVII, 538 pages. 2007.

Vol. 4420: S. Krishnamurthi, M. Odersky (Eds.), Compiler Construction. XIV, 233 pages. 2007.

Vol. 4419: P.C. Diniz, E. Marques, K. Bertels, M.M. Fernandes, J.M.P. Cardoso (Eds.), Reconfigurable Computing: Architectures, Tools and Applications. XIV, 391 pages. 2007.

Vol. 4418: A. Gagalowicz, W. Philips (Eds.), Computer Vision/Computer Graphics Collaboration Techniques. XV, 620 pages. 2007.

Vol. 4416: A. Bemporad, A. Bicchi, G. Buttazzo (Eds.), Hybrid Systems: Computation and Control. XVII, 797 pages. 2007.

Vol. 4415: P. Lukowicz, L. Thiele, G. Tröster (Eds.), Architecture of Computing Systems - ARCS 2007. X, 297 pages. 2007.

Vol. 4414: S. Hochreiter, R. Wagner (Eds.), Bioinformatics Research and Development. XVI, 482 pages. 2007. (Sublibrary LNBI).

Vol. 4412: F. Stajano, H.J. Kim, J.-S. Chae, S.-D. Kim (Eds.), Ubiquitous Convergence Technology. XI, 302 pages. 2007.

Vol. 4411: R.H. Bordini, M. Dastani, J. Dix, A.E.F. Seghrouchni (Eds.), Programming Multi-Agent Systems. XIV, 249 pages. 2007. (Sublibrary LNAI).

Vol. 4410: A. Branco (Ed.), Anaphora: Analysis, Algorithms and Applications. X, 191 pages. 2007. (Sublibrary LNAI).

Vol. 4409: J.L. Fiadeiro, P.-Y. Schobbens (Eds.), Recent Trends in Algebraic Development Techniques. VII, 171 pages. 2007.

Vol. 4407: G. Puebla (Ed.), Logic-Based Program Synthesis and Transformation. VIII, 237 pages. 2007.

Vol. 4406: W. De Meuter (Ed.), Advances in Smalltalk. VII, 157 pages. 2007.

Vol. 4405: L. Padgham, F. Zambonelli (Eds.), Agent-Oriented Software Engineering VII. XII, 225 pages. 2007.

Vol. 4403: S. Obayashi, K. Deb, C. Poloni, T. Hiroyasu, T. Murata (Eds.), Evolutionary Multi-Criterion Optimization. XIX, 954 pages. 2007.

Vol. 4401: N. Guelfi, D. Buchs (Eds.), Rapid Integration of Software Engineering Techniques. IX, 177 pages. 2007.

Vol. 4400: J.F. Peters, A. Skowron, V.W. Marek, E. Orłowska, R. Słowiński, W. Ziarko (Eds.), Transactions on Rough Sets VII, Part II. X, 381 pages. 2007.

Vol. 4399: T. Kovacs, X. Llorà, K. Takadama, P.L. Lanzi, W. Stolzmann, S.W. Wilson (Eds.), Learning Classifier Systems. XII, 345 pages. 2007. (Sublibrary LNAI).

Vol. 4398: S. Marchand-Maillet, E. Bruno, A. Nürnberger, M. Detyniecki (Eds.), Adaptive Multimedia Retrieval: User, Context, and Feedback. XI, 269 pages. 2007.

Vol. 4397: C. Stephanidis, M. Pieper (Eds.), Universal Access in Ambient Intelligence Environments. XV, 467 pages. 2007.

Vol. 4396: J. García-Vidal, L. Cerdà-Alabern (Eds.), Wireless Systems and Mobility in Next Generation Internet. IX, 271 pages. 2007.

Vol. 4395: M. Daydé, J.M.L.M. Palma, Á.L.G.A. Coutinho, E. Pacitti, J.C. Lopes (Eds.), High Performance Computing for Computational Science - VECPAR 2006. XXIV, 721 pages. 2007.

Vol. 4394: A. Gelbukh (Ed.), Computational Linguistics and Intelligent Text Processing. XVI, 648 pages. 2007.

Vol. 4393: W. Thomas, P. Weil (Eds.), STACS 2007. XVIII, 708 pages. 2007.

Vol. 4392: S.P. Vadhan (Ed.), Theory of Cryptography. XI, 595 pages. 2007.

Vol. 4391: Y. Stylianou, M. Faundez-Zanuy, A. Esposito (Eds.), Progress in Nonlinear Speech Processing. XII, 269 pages. 2007.

Vol. 4390: S.O. Kuznetsov, S. Schmidt (Eds.), Formal Concept Analysis. X, 329 pages. 2007. (Sublibrary LNAI).

Vol. 4389: D. Weyns, H.V.D. Parunak, F. Michel (Eds.), Environments for Multi-Agent Systems III. X, 273 pages. 2007. (Sublibrary LNAI).

Vol. 4385: K. Coninx, K. Luyten, K.A. Schneider (Eds.), Task Models and Diagrams for Users Interface Design. XI, 355 pages. 2007.

Vol. 4384: T. Washio, K. Satoh, H. Takeda, A. Inokuchi (Eds.), New Frontiers in Artificial Intelligence. IX, 401 pages. 2007. (Sublibrary LNAI).

Vol. 4383: E. Bin, A. Ziv, S. Ur (Eds.), Hardware and Software, Verification and Testing. XII, 235 pages. 2007

Table of Contents

Keynote I

Session 1: Programming Models

Session 2: Code Generation

Session 3: Parallelism

Keynote II

Session 4: Compilation Techniques

Session 5: Data Structures

Session 6: Register Allocation

Session 7: Memory Management

Compilation Techniques for Partitioned Global Address Space Languages

Kathy Yelick

EECS Department, UC Berkeley
Computational Research Division, Lawrence Berkeley National Lab

Abstract. Partitioned global address space (PGAS) languages have emerged as a viable alternative to message passing programming models for large-scale parallel machines and clusters. They also offer an alternative to shared memory programming models (such as threads and OpenMP) and the possibility of a single programming model that will work well across a wide range of shared and distributed memory platforms. Although the major source of parallelism in these languages is managed by the application programmer, rather than being automatically discovered by a compiler, there are many opportunities for program analysis to detect programming errors and for performance optimizations from the compiler and runtime system. The three most mature PGAS languages (UPC, CAF and Titanium) offer a statically partitioned global address space with a static SPMD control model, while languages emerging from the DARPA HPCS program are more dynamic.

In this talk I will describe some of the analysis and optimizations techniques used in the Berkeley UPC and Titanium compilers, both of which source-to-source translators based on a common runtime system. Both compilers are publicly released and run on most serial, parallel, and cluster platforms. Building on the strong typing of the underlying Java language, the Titanium compiler includes several forms of type-based analyses for both error detection and to enable code transformations. The Berkeley UPC compiler extends the Open64 analysis framework on which it is built to handle the language features of UPC. Both compilers perform communication optimizations to overlap, aggregate, and schedule communication, as well as pointer localization, and other optimizations on parallelism constructs in the language. The HPCS languages can use some of the implementation techniques of the older PGAS languages, but offer new opportunities for expressiveness and suggest new open questions related to compiler and runtime support, especially as machines scale towards a petaflop.

G. Almási, C. Caşcaval, and P. Wu (Eds.): LCPC 2006, LNCS 4382, p. 1, 2007.

Can Transactions Enhance Parallel Programs?*

Troy A. Johnson, Sang-Ik Lee, Seung-Jai Min, and Rudolf Eigenmann

School of Electrical and Computer Engineering
Purdue University, West Lafayette, IN 47907
{troyj, sangik, smin, eigenman}@purdue.edu

Abstract. Transactional programming constructs have been proposed as key elements of advanced parallel programming models. Currently, it is not well understood to what extent such constructs enable efficient parallel program implementations and ease parallel programming beyond what is possible with existing techniques. To help answer these questions, we investigate the technology underlying transactions and compare it to existing parallelization techniques. We also consider the most important parallelizing transformation techniques and look for opportunities to further improve them through transactional constructs or – vice versa – to improve transactions with these transformations. Finally, we evaluate the use of transactions in the SPEC OMP benchmarks.

1 Transaction-Supported Parallel Programming Models

Although a large number of parallel programming models have been proposed over the last three decades, there are reasons to continue the search for better models. Evidently, the ideal model has not yet been discovered; creating programs for parallel machines is still difficult, error-prone, and costly. Today, the importance of this issue is increasing because all computer chips likely will include parallel processors within a short period of time. In fact, some consider finding better parallel programming models one of today's most important research topics. Models are especially needed for non-numerical applications, which typically are more difficult to parallelize.

1.1 Can Transactions Provide New Solutions?

Recently, programming models that include transactional constructs have received significant attention [1,4,12,15]. At a high level, transactions are optimistically executed atomic blocks. The effect of an atomic block on the program state happens *at once*; optimistic execution means that multiple threads can execute the block in parallel, as long as some mechanism ensures atomicity. To this end, both hardware and software solutions have been proposed. An interesting observation is that these contributions make few references to technology in languages and compilers for parallel computing. These omissions are puzzling because the

* This work is supported in part by the National Science Foundation under Grants No. 0103582-EIA, and 0429535-CCF.

G. Almási, C. Caşcaval, and P. Wu (Eds.): LCPC 2006, LNCS 4382, pp. 2–16, 2007.

two topics pursue the same ultimate goal: making parallel programming easier and more efficient. While the programming models are arguably different, both areas need advanced compiler, run-time, and hardware optimization techniques. Hence, one expects that the underlying techniques supporting these models are closely related. In this paper, we investigate these relationships. We examine how much the concept of transactions can improve parallel program design and implementation beyond existing technology and to what extent transactions are just an interesting new way of looking at the same problem. We also review the ability of existing technology to optimize the implementation of transactions.

1.2 The Promise of Programming with Transactions

How can transactional constructs improve parallel programs? A transaction, in its basic meaning, is simply a set of instructions and memory operations. In many situations (e.g., in databases and parallel programming) it is important that the transactions are performed in such a way that their effects become visible simultaneously, or *atomically*. For example, in a bank, it is important that an amount of money gets deducted from one account and put into the other atomically, so that the total balance remains invariant at all times. Similarly, when incrementing a counter by two parallel threads, it is important that reading, modifying, and writing the counter be done atomically.

The concept of atomicity is not new per se. Constructs such as semaphores [5], locks [22], and critical sections [11] have been known for a long time. Nevertheless, language constructs that express atomicity typically allow only single memory updates (e.g., the OpenMP [21] `atomic` directive). Blocks of atomic memory operations are expressed through critical sections, which prevent concurrent execution of the block. This implementation is conservative or "pessimistic." The new promise of transactions is to eliminate some of the disadvantages that come with state-of-the-art constructs, namely reducing overhead through "optimistic execution" (if threads end up not accessing the same data inside a critical section, they should execute concurrently) and managing locks (avoiding deadlock and bookkeeping of multiple locks). These overheads naturally occur, as programs are written conservatively. For example, a banking software engineer may protect all account operations with one critical section, even though it could be known, in some cases, that the operations happen to different classes of accounts. The engineer may optimize the accounting software by creating separate locks for the two account classes; however, this increases the amount of bookkeeping information and requires more effort to avoid deadlocks.

The new idea behind transactions is that the programmer can rely on an efficient execution mechanism that executes in parallel *whenever possible*. Thus, the programmer uses the same "critical section" everywhere by simply writing *atomic*. At run time, two account operations or loop-counter updates can occur simultaneously. If different accounts are accessed or different counters are updated, then the program continues normally; if the same account or same counter is updated, then the transaction's implementation properly orders the operations. It is the transaction implementation's responsibility to provide

efficient mechanisms for detecting when concurrency is possible and for serializing the operations when necessary.

Two questions arise: (i) Are transactions an adequate user model, and (ii) can transactions be implemented efficiently? Although the idea of an atomic language construct is not new [20], only time and experience can answer whether programmers find transactions useful. Today, only few real programs have been written with transactional constructs. An important challenge is that much parallel programming experience exists in the area of numerical programs; however, transactions aim at all classes of programs. The second question is the focus of this paper. Our thesis is that the technology underlying efficient transactions is very similar to the one that exists today for program parallelization – parallelizing compiler techniques [3,9], implementation techniques of parallel language constructs [18], and hardware techniques for speculative parallelization [8,10]. The ultimate question for the language and compiler community is whether or not we have missed something that we can now learn from the ideas behind transactional constructs. If so, we may be able to incorporate that new knowledge into our compilers, run-time systems, and supporting hardware.

2 Comparing the Technology Underlying Transactions and Program Parallelization

2.1 Technology Underlying Transactions

Within transactions, threads that do not conflict should execute in parallel unhindered. Conflict detection is therefore at the heart of implementation technology for transactions. Conflict detection can be performed statically or dynamically.

Static conflict detection relies on the compiler's ability to tell that threads access disjoint data. Provably non-conflicting threads can execute safely in parallel without the guard of a transaction; the compiler can remove the transaction altogether. The compiler also may remove conflict-free code out of the transaction, hence narrowing the guarded section. This optimization capability is important because it allows the programmer to insert transactions at a relatively coarse level and rely on the compiler's ability to narrow them to the smallest possible width. Furthermore, if a compiler can identify instructions that always conflict, it may guard these sections directly with a classical critical section. Applying common data dependence tests for conflict resolution is not straightforward, as conflicts among *all* transactions must be considered. For *strong atomicity* [4] this analysis is even necessary between transactions and all other program sections. Note that common data-dependence tests attempt to prove independence, not dependence; i.e., failure to prove independence does not imply dependence.

Compile-time solutions are highly efficient because they avoid run-time overhead. Nevertheless, their applicability is confined to the range of compile-time analyzable programs. Often, these are programs that manipulate large, regular data sets – typically found in numerical applications. Compile-time conflict resolution is difficult in programs that use pointers to manipulate dynamic data structures, which is the case for a large number of non-numerical programs.

For threads that are not provably conflict-free, the compiler still can assist by narrowing the set of addresses that may conflict. At run time, this conflict set must be monitored. The monitoring can happen either through compiler-inserted code (e.g., code that logs every reference) or through interpreters (e.g., virtual machines). At the end of the transaction, the logs are inspected for possible conflicts; in the event of a conflict, the transaction is rolled back and re-executed. Rollback must undo all modifications and can be accomplished by redirecting all write references to a temporary buffer during the transaction. The buffer is discarded upon a rollback; a successful transaction commits the buffer to the real address space. Again, interpreters may perform this redirection of addresses and the final commit operation on-the-fly. Evidently, there is significant overhead associated with software implementations of transactions, giving rise to optimization techniques [1,12].

Fully dynamic implementations of transactions perform conflict detection, rollback and commit in hardware. During the execution of a transaction, data references are redirected to a temporary buffer and monitored for conflicts with other threads' buffers. Detected conflicts cause a rollback, whereby the buffer is emptied and threads are restarted. At the end of a successful, conflict-free transaction, the thread's buffer is committed. Conflict detection in hardware is substantially faster than software solutions, but still adds extra cycles to every data reference. The cost of a rollback is primarily in the wasted work attempting the transaction. Commit operations may be expensive, if they immediately copy the buffered data (for speculative parallelization, hardware schemes have been proposed to commit in a non-blocking style, without immediate copy [24]). An important source of overhead stems from the size of the buffer. While small hardware buffers enable fast conflict detection, they may severely limit the size of a transaction that can be executed. If the buffer fills up during a transaction, parallel execution stalls.

2.2 Technology Underlying Program Parallelization

A serial program region can be executed in parallel if it can be divided into multiple threads that access disjoint data elements. Implementing this concept requires techniques analogous to the ones in Section 2.1. There are compile-time, compiler-assisted run-time, and hardware solutions.

Compile-time parallelization. Data-dependence analysis is at the heart of compile-time, automatic parallelization. Provably independent program sections can be executed as fully parallel threads. The analysis is the same as what is needed for conflict detection of transactions. Data-dependence tests have proven most successful in regular, numerical applications; data dependence analysis in the presence of pointers [13] is still a largely unsolved problem. Where successful, automatic parallelization is highly efficient, as it produces fully-parallel sections, avoiding run-time overheads.

Run-time data-dependence tests. These tests [23] have been introduced to defer the detection of parallelism from compile time to run time, where the actual data