Annual Report for Contract Number N00014-88-K-0641

For the period: 1 October 1989 - 30 September 1990

-51234

Nico Habermann
Carnegie Mellon University
School of Computer Science
(412) 268 - 2592
anh@cs.cmu.edu
ONR Funding for Miró and Avalon
Student Reporting - Stewart Michael Clamen
N00014 - 88 - K - 0641
1 Oct 89 - 30 Sep 90

1 Productivity Measures

Refereed papers submitted but not yet published: 0
Refereed papers published: 1
Unrefereed reports and articles: 3
Books or parts thereof submitted but not yet published: 1
Books or parts thereof published: 0
Patents filed but not yet granted: 0
Patents granted: 0
Invited presentations: 0
Honors received (fellowships, technical society appointments, conference committee role, editorship, etc): 0
Prizes or awards received: 0
Promotions obtained: 0
Graduate students supported >= 25% of full time: 1
Post-docs supported >= 25% of full time: 0
Minorities supported: 0

Nico Habermann
Carnegie Mellon University
School of Computer Science
(412) 268 - 2592
anh@cs.cmu.edu
ONR Funding for Miró and Avalon
Student Reporting - Stewart Michael Clamen
N00014 - 88 - K - 0641
1 Oct 89 - 30 Sep 90

2 Detailed Summary of Technical Progess

Projects and Goals

A distributed system consists of multiple computers (called sites) that communicate through a network. Distributed systems are typically subject to site crashes and communication link failures. A crash renders a site's data temporarily or permanently inaccessible, while a communication link failure causes messages to be lost. A failure is detected when a site that has sent a message fails to receive a response after a certain duration. The absence of a response may indicate that the original message was lost, that the reply was lost, that the recipient has crashed, or simply that the recipient is slow to respond.

The primary goal of the Avalon Project is to create a set of linguistic constructs designed to give programmers explicit control over transaction-based processing of atomic objects for fault-tolerant applications. These constructs have been implemented as extensions to C++ and Common Lisp. The constructs include new encapsulation and abstraction mechanisms, as well as support for concurrency and recovery. The decision to extend an existing language rather than to invent a new one was based on pragmatic considerations. We felt we could focus more effectively on the new and interesting issues of reliability and concurrency if we did not have to redesign or reimplement basic language features, and we felt that building on top of a widely-used and widely-available language would facilitate the use of Avalon outside our own research group.

The Venari Project at CMU is interested in performing queries over objects residing in a distributed set of persistent repositories. These queries are "content-addressable" in the sense that they are based on the objects' semantics. A sample application would be one where a person at a workstation writes a first-order predicate as a query to retrieve all procedures whose pre- and post-condition specification "satisfy" the query, where those procedures are in a library that lives at a site remote from the workstation. Here, the objects are the procedures; the repository is a program module library; the specifications represent the procedures' semantics.

Past Accomplishments

Having completed out Avalon/C++ in previous years, this past year were worked on our prototype for the Avalon/Common Lisp system. Details of the system's motivation and results follow:

One of the initial design goals of the Avalon Project was to extend existing languages, rather than invent new ones. In the process of such an extension, it is important to introduce as few new features as possible, and design those features to combine well with the base language's idioms and model of computation. In our previous effort, Avalon/C++, we chose an object-oriented design, consistent with the C++ model. In Avalon/Common Lisp, we strove to extend the language in a similarly consistent manner.

The most significant enhancement we made to Common Lisp was the addition of a new first-class data type, the **evaluator**. An evaluator represents an additional, non-local, Common Lisp evaluator, on which the user can evaluate expressions, install procedures, and modify accessible data. Evaluators are used via two new macros, `remote` and `local`, which direct the thread of computation to a different evaluator.

Avalon/Common Lisp is built on top of three locally-developed systems, CMU Common Lisp, Mach, and Camelot, and runs on IBM-PC/RTs. CMU Common Lisp is one of the first implementations of Common Lisp, and was chosen over other Common Lisp implementations for two reasons. Firstly, it is the only available Common Lisp that runs on the computers the Avalon Group had already available. We also favored the presence of the support and maintenance the locally-managed system provides.

Mach[1], a Unix-like operating system with support for distributed computation, is used to provide communication among the various Avalon processes, and to support process-level concurrency. Camelot[2, 3], a machine-independent, high-performance, distributed transaction facility, is used to support the fault-tolerance and reliability we desire.

For more details on Avalon/Common Lisp and our conclusions, please refer to the enclosed Technical Report [4, 5] and group notes[6, 7], or to last years Fiscal Year report.

A shift in focus charaterized my work this year. The successor to Avalon, Venari, is more concerned with persistence of data and the manipulation thereof, and I have spent a large part of the year researching various other programming languages featuring persistent data and other database features as part of their computation models. I conducted a thorough literature search, and am in the process of completing a survey report of my findings.

Future Goals

My future research plan involves a shift in direction, away from distributed computing and towards parallel computation. More specifically, how transactions can be used as part of the effort to introduce concurrency into existing programming languages. A more detailed sketch follows:

With the advent of multiprocessors and supercomputers, there has been considerable effort to develop programming language systems for these new platforms. These systems have taken two forms: sequential languages with optimizing compilers that introduce concurrency while preserving the sequential semantics, and languages with explicit constructs for concurrency.

Both approaches have merits. The "parallelizing" languages benefit from being compatible with existing programs (the so-called "dusty deck" problem) and existing programmers are already familiar with the computation model presented. Languages that offer explicit control over parallelism allow the programmer to optimize his program to a much finer degree than the parallelized sequential program.

Much of the work in parallelizing sequential programs has been directed at imperative programming constructs, such as loops and array processing. In mostly-functional languages, such as Scheme or ML, imperative statements and constructs are not as prevalent. However, various characteristics of the computation model of such languages offer opportunities for the implicit introduction of parallelism. Both Scheme and ML leave the order of evaluation of function call arguments unspecified. As a result, a parallelizing compiler might choose to evaluate the arguments concurrently. Another possible optimization is the evaluation of the forms included within a sequencing statement in parallel.

One important detail of this parallelization scheme has been omitted. One constraint on the parallelization of loop constructs in imperative languages is that there is no data dependency between the various loop iterations. If a data dependency exists, it is possible that the concurrent execution of the loop will produce a different result than the sequential execution that is being emulated. Since the implicit addition of concurrency into a program is done only to improve exectuion performance, an optimizing compiler is forbidden to perform any optimization that could alter the semantics of the program. This restriction applies equally to the functional optimizations mentioned previously.

Unlike FORTRAN arrays, however, it is not always possible to conclusively determine whether two execution threads can run independently. The compiler, forced to act conservatively, would therefore be unable to allow the threads to run simultaneously, even if the threat of interference is remote. In such circumstances, the compiler might benefit if it could use transaction-processing technology. Such technology would support the detection of data interference at execution time, and allow the thread of execution fall back into a sequential posture. [not precise enough, and transactions introduced too rapidly.]

Another benefit incurred by the presence of transactions would be the ability to improve concurrent performance via of use of speculative computation. A conditional statement can be executed by first spawning off *both* alternatives, and concurrently evaluating the predicate test. Once the predicate value has been determined, the system can abort the appropriate arm. Transaction semantics would ensure that the standard sequential semantics of the conditional statement would be preserved.

Such ideas were first introduced in the ParaTran system.[8]

Transactions could also be used to solve some of the problems present in existing explicit parallel languages. For example, both Multilisp [9] and Qlisp[10. 11] ignore the possiblity of interference that might result from the concurrent execution of threads if the computation relies on side-effecting operations.

References

[1] Mike Accetta, Robert V. Baron, William Bolosky, David B. Golub, Richard F. Rashid, Avadis Tevanian, Jr., and Michael Wayne Young. Mach: A New Kernel Foundation for UNIX Development. In *Proceedings of Summer Usenix*, July 1986.

[2] Alfred Z. Spector, Joshua J. Bloch, Dean S. Daniels, Richard P. Draves, Dan Duchamp, Jeffrey L. Eppinger, Sherri G. Menees, and Dean S. Thompson. The Camelot Project. *Database Engineering*, 9(4), December 1986. Also available as Technical Report CMU-CS-86-166, Carnegie Mellon University, November 1986.

[3] Jeffrey L. Eppinger, Lily B. Mummert, and Alfred Z. Spector, editors. *Camelot and Avalon: A Distributed Transaction Facility.* The Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann Publishers, Inc., San Mateo, California, February 1991.

[4] Stewart M. Clamen, Linda D. Leibengood, Scott M. Nettles, and Jeannette M. Wing. Reliable distributed computing with Avalon/Common Lisp. Technical Report CMU-CS-89-186, Carnegie Mellon University School of Computer Science, September 1989.

[5] Stewart M. Clamen, Linda D. Leibengood, Scott M. Nettles, and Jeannette M. Wing. Reliable distributed computing with Avalon/Common Lisp. In *Proceedings of the International Conference on Computer Languages*, New Orleans, LA, March 1990. Institute of Electrical and Electronic Engineers Computer Society. *Also available as Carnegie Mellon School of Computer Science Tech Report # CMU-CS-89-186; also an extended abstract appears as "An overview of Avalon/Common Lisp," in the Proceedings of the Third Workshop on Large Grained Parallel Programming (Pittsburgh, PA, October 10-11, 1989).*

[6] Stewart M. Clamen, Linda D. Leibengood, Scott N. Nettles, and Jeannette M. Wing. A programmer's guide to Avalon/Common Lisp. Avalon Note 15, Carnegie Mellon University School of Computer Science, 1990.

[7] Stewart M. Clamen, Linda D. Leibengood, Scott N. Nettles, and Jeannette M. Wing. Assessment of the Avalon/Common Lisp implementation. Avalon Note 16, Carnegie Mellon University School of Computer Science, 1990.

[8] M. Katz. Paratran: A transparent, transaction based runtime mechanism for parallel execution of scheme. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, 1986. *Updated in [?].*

[9] R. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(3):501–538, October 1985.

[10] R.P. Gabriel and J. McCarthy. Queue-based multiprocessing lisp. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 25–44, Austin, TX, Aug 1984.

[11] R. Goldman, R. P. Gabriel, and C. Sexton. Qlisp: An interim report. In *Parallel Lisp: Languages and Systems, Proceedings of the U.S./Japan Workshop on Parallel Lisp.*, volume 441 of *Lecture Notes in Computer Science*, pages 161–181. Springer-Verlag, 1990. *Workshop help in Sendai, Japan, June 5–8, 1989.*

[12] Jeanette Wing, Maurice Herlihy, Stewart Clamen, David Detlefs, Karen Kietzke, Richard Lerner, and Su-Yuen Ling. A Tutorial Introduction. In Jeffrey L. Eppinger, Lily B.Mummert, and Alfred Z. Spector, editors, *Camelot and Avalon: A Distributed Transaction Facility*, chapter 19, pages 305–334. Morgan Kaufmann Publishers, Inc., San Mateo, California, February 1991.

Nico Habermann
Carnegie Mellon University
School of Computer Science
(412) 268 - 2592
anh@cs.cmu.edu
ONR Funding for Miró and Avalon
Student Reporting - Stewart Michael Clamen
N00014 - 88 - K - 0641
1 Oct 89 - 30 Sep 90

3 Publications, Presentations and Reports

Below are the list of publications I have been involved with during the past year:

References

[1] Stewart M. Clamen, Linda D. Leibengood, Scott M. Nettles, and Jeannette M. Wing. Reliable distributed computing with Avalon/Common Lisp. In *Proceedings of the International Conference on Computer Languages*, New Orleans, LA, March 1990. Institute of Electrical and Electronic Engineers Computer Society. *Also available as Carnegie Mellon School of Computer Science Tech Report # CMU-CS-89-186; also an extended abstract appears as "An overview of Avalon/Common Lisp," in the Proceedings of the Third Workshop on Large Grained Parallel Programming (Pittsburgh, PA, October 10-11, 1989).*

[2] Stewart M. Clamen, Linda D. Leibengood, Scott N. Nettles, and Jeannette M. Wing. A programmer's guide to Avalon/Common Lisp. Avalon Note 15, 1990.

[3] Stewart M. Clamen, Linda D. Leibengood, Scott N. Nettles, and Jeannette M. Wing. Assessment of the Avalon/Common Lisp implementation. Avalon Note 16, 1990.

[4] Jeanette Wing, Maurice Herlihy, Stewart Clamen, David Detlefs, Karen Kietzke, Richard Lerner, and Su-Yuen Ling. A Tutorial Introduction. In Jeffrey L. Eppinger, Lily B.Mummert, and Alfred Z. Spector, editors, *Camelot and Avalon: A Distributed Transaction Facility*, chapter 19, pages 305–334. Morgan Kaufmann Publishers, Inc., San Mateo, California, February 1991.

Nico Habermann
Carnegie Mellon University
School of Computer Science
(412) 268 - 2592
anh@cs.cmu.edu
ONR Funding for Miró and Avalon
Student Reporting - Stewart Michael Clamen
N00014 - 88 - K - 0641
1 Oct 89 - 30 Sep 90

4 Research Transitions and DoD Interactions

A number of researchers, in both academia and industry, have expressed an interest in our Avalon/C++ work. Commercial sites include Microsoft, Texas Instruments, Hewlett Packard, and NCR. Academic sites include Boston Univeristy, University of Massachusetts – Amherst, Concordia University (Montreal), and University of New South Wales (Australia). (A more detailed list is available upon request.)

Avalon/Common Lisp is more dependent on the facilities present at our local site, and is thus not as portable.

Nico Habermann
Carnegie Mellon University
School of Computer Science
(412) 268 - 2592
anh@cs.cmu.edu
ONR Funding for Miró and Avalon
Student Reporting - Stewart Michael Clamen
N00014 - 88 - K - 0641
1 Oct 89 - 30 Sep 90

5 Software and Hardware Prototypes

Avalon/Common Lisp is stable, but it only a prototype, and thus has restricted function. There are no plans to continue development on it, although work is proceeding on a ML-based successor.