Status Report: The Manticore Project

http://manticore.cs.uchicago.edu

Carsen Berger   Lars Bergstrom
John Reppy   Stephen Rosen
Nora Sandler   Adam Shaw
University of Chicago
{clberger,larsberg,jhr,sirosen,nlsandler,ams}@cs.uchicago.edu

Matthew Fluet
Rochester Institute of Technology
mtf@cs.rit.edu

Mike Rainey
Max Planck Institute for Software Systems
mrainey@mpi-sws.org

Abstract

The Manticore project began in 2007 with the goal of designing and implementing a parallel functional programming language that targets multicore and shared-memory multiprocessors. Our language is a dialect of Standard ML, called Parallel ML (PML), that combines implicitly threaded constructs for fine-grain parallelism with CML-style explicit concurrency for coarse-grain parallelism. We have a prototype implementation that demonstrates both good sequential performance and scalability on both 32-core Intel and 48-core AMD machines. This paper describes the current state of our work and describes several new initiatives in both the language design and implementation.

1. Introduction

The Manticore project was motivated by the trend toward commodity parallel hardware, which has seen multicore processors become pervasive over the past few years [FFR$^+$07]. Our goal is to support applications that exhibit parallelism across a range of scales and patterns. The Manticore system consists of a parallel dialect of Standard ML [MTHM97], called *Parallel ML* (PML), and a supporting parallel runtime system. In this paper, we describe the current status of our system and give an overview of current and future work.

2. Current status

Over the past five years, we have completed a first version of our language design and built a prototype implementation. The details of this work has been reported in a number of papers; here we give a quick overview of its current status.

2.1 Language design

PML is based on a mutation-free subset of SML (*i.e.*, no refs or arrays), extended with mechanisms for fine-grain parallelism [FRRS11] and with CML-style explicit concurrency [Rep99]. We believe that a strongly-typed, mutation-free, strict functional language provides an ideal base for parallel and concurrent programming. At the language level, shared memory creates creates correctness issues, such as data races, that programmers find difficult to deal with, and it introduces unnecessary nondeterminism and serial dependencies. By avoiding shared memory, we provide a predictable programming model that is easy for programmers to reason about. A strict, mutation-free programming model also gives the implementation the freedom to use techniques, such as replication, to improve data locality.

We support fine-grain parallelism in PML through lightweight annotations on standard SML constructs [FRRS11]. These include parallel tuples, which provide fork-join parallelism, and parallel value bindings, which provide a future-like mechanism. A major addition to SML notation is parallel array comprehensions, which support nested data parallelism (NDP) [BCH$^+$94], and which can be used to express irregular parallel computations. These mechanisms all have deterministic semantics, but we also include support for speculation in the form of the parallel case construct.

2.2 Implementation

We have built a compiler and parallel runtime system for our current design. With this system, we have demonstrated effective performance scaling on both 32-core Intel and 48-core AMD machines. Our system supports Linux and Mac OS X on x86-64 hardware.

In our runtime system, we designed our memory-management system for scalable parallel performance. We use a parallel, copying garbage collector that provides NUMA-aware, scalable performance [ABFR11]. It contains thread-local private heaps and a parallel global collector that gains significant performance by exploiting and preserving physical memory locality. We maintain the invariant that there are no pointers from global memory back into the private heaps, which allows the private heaps to be collected concurrently without synchronization. Our work-stealing scheduler also provides scalable performance on both regular and irregular problems [BFR$^+$10, BFR$^+$12].

3. Future work

Building on our current prototype, we are pursuing a number of different improvements in design, performance, and robustness.

3.1 Data-only flattening

Our approach to implementing NDP constructs has relied on efficient runtime work stealing techniques to get scalable performance [Rai10, BFR$^+$12]. Most NDP implementations use compiler transformation, called flattening [BCH$^+$94] or vectorization [CLPK08], that transforms irregular NDP code and arrays into regular data-parallel code and arrays. Our belief has been that this transformation is best suited to vector hardware, but we have recently developed a version of flattening that is designed to work well on modern multicore machines by transforming the data but leaving the code largely alone [Sha11]. We are currently working on a more robust implementation of this technique.

3.2 Mutable state

PML does not support references or mutable arrays, which makes it a better base for parallel and concurrent programming. Unfortunately, there are situations where using shared memory to communicate between parallel computations provides asymptotic performance advantages. For example, the fastest modern parallel SAT solvers and game search implementations use shared data structures to eliminate redundant computation. We are extending PML with mechanisms that support shared state without contaminating the whole language.

3.3 Improved closure conversion

The current implementation of Manticore uses flat closures. We have implemented and are currently evaluating an implementation of the Shao-Appel safe-for-space closure conversion algorithm [SA00]. In Manticore, we have a 0CFA-based control-flow analysis that provides us with additional information that we use to inline continuations more often than was possible in SML/NJ [AM91]. Further, our work on arity raising [BR09] significantly increases the number of parameters to each function. Because of these two issues, we found that callee-save registers reduced performance because of register pressure, so omit that part of the Shao-Appel approach.

3.4 Reflow analysis

Inlining functions with free variables efficiently is, in general, an open problem. In his Ph.D. thesis, Olin Shivers proposed reflow analysis as an approach to determine when it was safe to perform this optimization, which he called super-β [Shi91]. Unfortunately, his approach requires executing a limited control-flow analysis (CFA) at each inlining point, which is expensive, and not amenable to incremental CFA [HM97]. We have formulated a novel graph-based solution to this problem, have demonstrated that it requires no more time than our other similarly complicated optimization passes, and are currently evaluating its effectiveness.

3.5 MLton frontend

Our compiler supports only a subset of the Standard ML language and Basis Library [GR04], which makes porting programs to Manticore difficult. Furthermore, it has rather rudimentary error reporting. To address these problems, we are replacing our front-end with the MLton front-end (up to the monomorphic SXML representation) [Wee06]. This new front-end will allow PML programs to use the full SML language, including functors, and will make it a better platform for future use to teach parallel programming. A side benefit is MLton's monomorphization, which should improve our handling of NDP array types.

3.6 LLVM backend

We currently rely on MLRISC for our x86-64 code generation [GGR94]. While it supports our unique calling convention cleanly and supports sophisticated register allocation [GA96], extending MLRISC to support new machine features, such as SSE instructions, would require significant effort. We are exploring LLVM [Lat02] as a possible alternative, although it does not easily support custom calling conventions.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grants CCF-0811389 and CCF-1010568, and upon work performed in part while John Reppy was serving at the National Science Foundation. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of these organizations or the U.S. Government.

References

[ABFR11] Auhagen, S., L. Bergstrom, M. Fluet, and J. Reppy. Garbage Collection for Multicore NUMA Machines. In *MSPC 2011*, San José, CA, June 2011. ACM.

[AM91] Appel, A. W. and D. B. MacQueen. Standard ML of New Jersey. In *PLIP '91*, vol. 528 of *LNCS*. Springer-Verlag, New York, NY, August 1991, pp. 1–26.

[BCH+94] Blelloch, G. E., S. Chatterjee, J. C. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. *JPDC*, **21**(1), 1994, pp. 4–14.

[BFR+10] Bergstrom, L., M. Fluet, M. Rainey, J. Reppy, and A. Shaw. Lazy tree splitting. In *ICFP '10*. ACM, September 2010, pp. 93–104.

[BFR+12] Bergstrom, L., M. Fluet, M. Rainey, J. Reppy, and A. Shaw. Lazy tree splitting. *JFP*, 2012. Accepted for publication.

[BR09] Bergstrom, L. and J. Reppy. Arity raising in manticore. In *IFL '09*, vol. 6041 of *LNCS*. Springer-Verlag, September 2009, pp. 90–106.

[CLPK08] Chakravarty, M. M. T., R. Leshchinskiy, S. Peyton Jones, and G. Keller. Partial vectorisation of Haskell programs. In *DAMP '08*. ACM, January 2008, pp. 2–16. Available from `http://clip.dia.fi.upm.es/Conferences/DAMP08/`.

[FFR+07] Fluet, M., N. Ford, M. Rainey, J. Reppy, A. Shaw, and Y. Xiao. Status Report: The Manticore Project. In *ML '07*. ACM, October 2007, pp. 15–24.

[FRRS11] Fluet, M., M. Rainey, J. Reppy, and A. Shaw. Implicitly-threaded parallelism in Manticore. *JFP*, **20**(5–6), 2011, pp. 537–576.

[GA96] George, L. and A. Appel. Iterated register coalescing. *ACM TOPLAS*, **18**(3), May 1996, pp. 300–324.

[GGR94] George, L., F. Guillame, and J. Reppy. A portable and optimizing back end for the SML/NJ compiler. In *CC '94*, April 1994, pp. 83–97.

[GR04] Gansner, E. R. and J. H. Reppy (eds.). *The Standard ML Basis Library*. Cambridge University Press, Cambridge, England, 2004.

[HM97] Heintze, N. and D. McAllester. Linear-time subtransitive control flow analysis. In *PLDI '97*, Las Vegas, Nevada, United States, 1997. ACM.

[Lat02] Lattner, C. LLVM: An infrastructure for multi-stage optimization. Master's dissertation, C.S. Dept., UIUC, Urbana, IL, December 2002.

[MTHM97] Milner, R., M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, Cambridge, MA, 1997.

[Rai10] Rainey, M. *Effective Scheduling Techniques for High-Level Parallel Programming Languages*. Ph.D. dissertation, University of Chicago, August 2010. Available from `http://manticore.cs.uchicago.edu`.

[Rep99] Reppy, J. H. *Concurrent Programming in ML*. Cambridge University Press, Cambridge, England, 1999.

[SA00] Shao, Z. and A. W. Appel. Efficient and safe-for-space closure conversion. *ACM TOPLAS*, **22**(1), 2000, pp. 129–161.

[Sha11] Shaw, A. *Implementation techniques for nested-data-parallel languages*. Ph.D. dissertation, University of Chicago, August 2011. Available from `http://manticore.cs.uchicago.edu`.

[Shi91] Shivers, O. *Control-flow analysis of higher-order languages*. Ph.D. dissertation, School of C.S., CMU, Pittsburgh, PA, May 1991.

[Wee06] Weeks, S. Whole program compilation in MLton. Invited talk at ML '06 Workshop, September 2006.