

The goal of my research is to make it easier to write correct and efficient programs through advances in the design and implementation of declarative languages. Declarative languages provide programmers with such essential services as automatic thread scheduling and memory management. My research focuses on improving the effectiveness and flexibility of these services.

Declarative languages, such as PML [FRRS08] and Data-Parallel Haskell [CLP⁺07], provide *implicit* and *explicit* threading mechanisms. An implicit thread is a linguistic construct that acts as a hint to the scheduler for where parallel evaluation may be profitable. Explicit threads provide a mechanism for concurrent programming and coarse-grain parallel programming.

My thesis research presents the design of an effective system for a language that supports implicit threading and runs on a shared-memory multiprocessor. An effective system is scalable and robust. A system is scalable if performance improves in proportion to the number of processing elements. A system is robust when performance is consistently good under changing conditions, such as a change of input data set, number of processors, or hardware platform. Robust systems are predictable across programs generally, not just those tuned for a particular set of conditions.

Research on thread scheduling provides evidence that no single scheduling policy is suitable for every application and, furthermore, that there are many distinct policies, each with its own strengths and weaknesses. For instance, Blelloch *et al.* have shown that, for computations involving implicit threads, a “space-efficient” scheduling policy has lower memory use at run time than other policies [BGM99]. But such a space-efficient scheduler has a higher communication overhead than the work-stealing policy used by Cilk [BJK⁺95]. There exists a fundamental trade-off between the two policies and supporting both approaches is the most flexible approach.

My past work explores the design of a language implementation that supports a mix of scheduling policies. I have performed this research in the context of the parallel run-time system of Manticore [FRR⁺07, FFR⁺07]. In ongoing research, I am exploring the complementary problem of giving a programmer who is not a system expert the ability to program scheduling policies. In this work, I am investigating a domain-specific language for encoding scheduling policies.

I have also undertaken two projects outside my thesis research. In collaboration with other members of the Manticore group, I have helped to specify the semantics of implicit threading in the Manticore language, in particular, on the parts related to scheduling. Also, I have worked on the problem of supporting variadic foreign-function calls in a high-level language in the context of the SML/NJ [AM91] compiler.

An effective implicit-threading system

Lazy Binary Splitting is a technique enabling good dynamic load balancing for executing parallel loops over flat arrays on multiprocessor machines [TCBV10]. In recent work, Lars Bergstrom, Matthew Fluet, John Reppy, Adam Shaw, and I present a modified strategy called Lazy Tree Splitting which extends the technique to trees [BFR⁺10]. We describe an implementation of nested-data parallelism [BCH⁺94], a paradigm for programming irregular parallel applications in a declarative style, that is based on Lazy Tree Splitting. We demonstrate implementation’s effectiveness with empirical results from a Manticore prototype on a range of benchmarks. Unlike related systems, ours performs well across diverse benchmarks without any application- or machine-specific tuning. Our implementation places only modest demands on the host language: the only special requirement is a work-stealing scheduler, which is now standard in parallel languages. Our implementation of nested-data parallelism relies entirely on dynamic techniques, unlike related flattening

techniques, which rely on sophisticated compiler support.

In work with, Lars Bergstrom, Matthew Fluet, John Reppy, Adam Shaw, and I improved the scalability of the run-time system used by Manticore. Our design uses a split heap memory model, a mix of asynchronous and parallel garbage collectors, and a new variant of the popular work stealing scheduler. Our scheduler uses a technique which we have dubbed lazy promotion to reduce communication overheads caused by a parallel garbage collector. Through some representative benchmarks we have shown cases where our technique can significantly improve scalability over two other techniques: a flat-heap implementation and a split-heap implementation with eager promotion. A report of this work is in progress.

The Manticore scheduling system

My past research addresses the design of a scheduling system for a language, such as PML, that has implicit and explicit threading. The combination of implicit and explicit threads creates scheduling requirements that go beyond the capabilities of most existing language implementations. For example, an explicit thread may need one of many real-time scheduling policies to provide responsiveness or, alternatively, a resource-aware policy [vCZ⁺03] to maximize throughput. An implicit thread needs a work-distribution policy such as work stealing [Hal84] to balance load among system processors. Other special-purpose implicit-threading mechanisms have been proposed, including clocks and phasers [SPSS08], and `pcase` [FRRS08], that require special scheduling policies. A system that supports both implicit and explicit threads needs a diverse array of scheduling policies.

In an ICFP'08 paper, Matthew Fluet, John Reppy, and I designed and evaluated the Manticore scheduling system, which supports a mix of scheduling policies [FRR08]. Our system consists of a collection of primitives that a compiler expert can use to program scheduling policies. We evaluate our system empirically by showing that it can support a range of policies from the scheduling literature and it admits scalability on a multicore system. In addition, we investigate the use of hierarchical scheduling [FS96, Reg01, LMD04] as a way to organize a system with multiple concurrently-executing scheduling policies. Our study offers evidence that hierarchical scheduling can be a useful feature for enabling new types of code sharing. For instance, we use hierarchical scheduling to provide “thread cancellation”, which is a protocol that allows a thread to cancel a group of related threads. We show that, with only a modest effort on behalf of the scheduler writer, different scheduler implementations can share a single implementation of thread cancellation. Having such reusable components has proved an useful tool for building a system with a mix of scheduling policies, as it can help to reduce implementation effort.

Programmable scheduling policies

The way that scheduling policies are written in a language implementation (or operating system) leads to unclear and incorrect code. Because of performance reasons, the scheduling code is often split into small pieces and the pieces are spread across several functions or files. The more natural specification of the scheduler, however, often consists of a single scheduling loop. Such a specification is simpler and easier to test in isolation. A domain-specific language offers the potential to write code that is closer to the original specification, and if the domain-specific language employs an optimizing compiler, it also offers the potential to generate efficient scheduling code.

As I argue above, it is desirable to support a mix of scheduling policies. A domain-specific language can assist in this goal by admitting a high-level specification of a policy that is liberated from complexities

of a particular language implementation. There are situations in which an application writer who is not an expert in the language implementation may desire a custom scheduling policy for his or her application. A domain-specific language would allow such a programmer to write a custom policy.

These issues have motivated other researchers to explore the domain-specific language called Bossa [BM02, MLD05]. Bossa, however, is limited to uniprocessor scheduling, and multiprocessor scheduling would require a more general model than Bossa provides. A multiprocessor policy is a *distributed program* in which a whole scheduler consists of multiple instances that execute concurrently. Often these instances need to communicate to balance workload. Currently, I am exploring a domain-specific language called SchedML that addresses these issues. My contributions include a new language design and a compiler that compiles a SchedML program to a Manticore program.

I plan to evaluate SchedML by testing it with several representative scheduling policies and measuring their performance. In the longer term, I plan to investigate how SchedML can support automatic debugging of policies. A significant challenge is that a multiprocessor policy in SchedML is encoded as a distributed program, which means that such a policy can contain concurrency bugs. I believe that I can approach this problem by drawing from existing work on model checking [MQ07, MQ08].

Implicit threading for Manticore

In collaboration with Matthew Fluet, John Reppy and Adam Shaw, I aided in the design of the implicit-threading mechanism for the Manticore programming language [FRRS08]. My contribution focused on the relation between the semantics of implicit threading and the scheduling policy. Also, I have adapted the clone compilation technique used by Cilk [FLR98] to one of our implicit-threading constructs. Doing so required solving some potential space-explosion problems. We were invited to submit a longer version of our ICFP'08 paper to a special issue of JFP, and this paper is currently under review by JFP.

Variadic foreign-function calls

Supporting foreign-function calls in a language implementation has proved challenging because of the proliferation of different calling conventions and of the often poor quality of their specifications. To address this problem, Lindig *et al.* proposed Staged Allocation [OLR06], which provides a domain-specific language for specifying different calling conventions and an abstract machine for determining the sequences of machine instructions for carrying out a function call. In an ML'08 paper, Matthias Blume, John Reppy, and I devised a novel technique to extend Staged Allocation to support variadic foreign-function calls [BRR08]. Before our work, variadic function calls had been a missing feature of almost all foreign-function interfaces, primarily because of the perception that the implementation effort necessary for supporting variadic calls would outweigh the usefulness of supporting them. Our work shows, however, that a simple implementation does exist, and our solution uses a novel “calling-convention interpreter” that shares all the existing Staged Allocation machinery, but uses the machinery at run time instead of compile time. As a bonus, it is not necessary to re-compile the compiler to use a new function calling convention.

References

- [AM91] Appel, A. W. and D. B. MacQueen. Standard ML of New Jersey. In *Programming Language Implementation and Logic Programming*, vol. 528 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, NY,

- August 1991, pp. 1–26.
- [BCH⁺94] Blleloch, G. E., S. Chatterjee, J. C. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, **21**(1), 1994, pp. 4–14.
- [BFR⁺10] Bergstrom, L., M. Fluet, M. Rainey, J. Reppy, and A. Shaw. Lazy tree splitting. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*. ACM, September 2010.
- [BGM99] Blleloch, G. E., P. B. Gibbons, and Y. Matias. Provably efficient scheduling for languages with fine-grained parallelism. *Journal of the ACM*, **46**(2), 1999, pp. 281–321.
- [BJK⁺95] Blumofe, R. D., C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, Santa Barbara, California, July 1995. ACM, pp. 207–216.
- [BM02] Barreto, L. P. and G. Muller. Bossa: a language-based approach to the design of real-time schedulers. In *10th International Conference on Real-Time Systems (RTS '02)*, Paris, France, March 2002. pp. 19–31.
- [BRR08] Blume, M., M. Rainey, and J. Reppy. Calling variadic functions from a strongly-typed language. In *Proceedings of the 2008 ACM SIGPLAN Workshop on ML*, Victoria, BC, Canada, 2008. ACM, pp. 47–58.
- [CLP⁺07] Chakravarty, M. M. T., R. Leshchinskiy, S. Peyton Jones, G. Keller, and S. Marlow. Data Parallel Haskell: A status report. In *Proceedings of the ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming*. ACM, January 2007, pp. 10–18.
- [FFR⁺07] Fluet, M., N. Ford, M. Rainey, J. Reppy, A. Shaw, and Y. Xiao. Status Report: The Manticore Project. In *Proceedings of the 2007 ACM SIGPLAN Workshop on ML*. ACM, October 2007, pp. 15–24.
- [FLR98] Frigo, M., C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI '98)*, June 1998, pp. 212–223.
- [FRR⁺07] Fluet, M., M. Rainey, J. Reppy, A. Shaw, and Y. Xiao. Manticore: A heterogeneous parallel language. In *Proceedings of the ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming*. ACM, January 2007, pp. 37–44.
- [FRR08] Fluet, M., M. Rainey, and J. Reppy. A scheduling framework for general-purpose parallel languages. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, Victoria, BC, Canada, September 2008. ACM, pp. 241–252.
- [FRRS08] Fluet, M., M. Rainey, J. Reppy, and A. Shaw. Implicitly-threaded parallelism in Manticore. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, Victoria, BC, Canada, September 2008. ACM, pp. 119–130.
- [FS96] Ford, B. and S. Susarla. Cpu inheritance scheduling. *SIGOPS Oper. Syst. Rev.*, **30**(SI), 1996, pp. 91–105.
- [Hal84] Halstead Jr., R. H. Implementation of multilisp: Lisp on a multiprocessor. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*. ACM, August 1984, pp. 9–17.
- [LMD04] Lawall, J., G. Muller, and H. Duchesne. Language design for implementing process scheduling hierarchies. In *Symposium on Partial Evaluation and Program Manipulation*, 2004.
- [MLD05] Muller, G., J. L. Lawall, and H. Duchesne. A framework for simplifying the development of kernel schedulers: Design and performance evaluation. In *HASE 2005: High Assurance Systems Engineering Conference*, Heidelberg, Germany, October 2005. pp. 56–65.
- [MQ07] Musuvathi, M. and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of the 2007 SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*, San Diego, CA, June 2007. ACM, pp. 446–455.
- [MQ08] Musuvathi, M. and S. Qadeer. Fair stateless model checking. In *Proceedings of the 2008 SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*, Tucson, AZ, USA, 2008. ACM, pp. 362–371.

- [OLR06] Olinsky, R., C. Lindig, and N. Ramsey. Staged allocation: a compositional technique for specifying and implementing procedure calling conventions. In *Conference Record of the 33rd Annual ACM Symposium on Principles of Programming Languages (POPL '06)*. ACM, 2006, pp. 409–421.
- [Reg01] Regehr, J. *Using Hierarchical Scheduling to Support Soft Real-Time Applications on General-Purpose Operating Systems*. Ph.D. dissertation, University of Virginia, 2001.
- [SPSS08] Shirako, J., D. M. Peixotto, V. Sarkar, and W. N. Scherer. Phasers: a unified deadlock-free construct for collective and point-to-point synchronization. In *ICS '08: Proceedings of the 22nd Annual International Conference on Supercomputing*, Island of Kos, Greece, 2008. ACM, pp. 277–288.
- [TCBV10] Tzannes, A., G. C. Caragea, R. Barua, and U. Vishkin. Lazy binary-splitting: a run-time adaptive work-stealing scheduler. In *Proceedings of the 2010 ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, Bangalore, India, January 2010. ACM, pp. 179–190.
- [vCZ⁺03] von Behren, R., J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio: Scalable threads for internet services. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, December 2003, pp. 268–281.