The goal of my research is to ease the development of complex software systems deployed on next generation multi-core systems and compute clouds through the use of programming language technology. While I am broadly interested in concurrency and parallelism, I am particularly drawn to architecture-induced complexities such as absence of cache coherence and eventual consistency, which often lead to programs where the guarantee of correctness and concision take a back seat to performance. My aim is to (1) identify future challenges in scalable software systems, (2) discover new programming abstractions to declaratively express programs for such systems, and (3) utilize runtime system techniques to efficiently realize the abstractions without compromising performance.

We are in an era of pervasive parallelism. Multi-core processors have become the norm, and massively parallel hardware such as graphics processing units (GPUs) are increasingly prevalent. In addition, on-demand cluster computing services such as Windows Azure and Amazon's EC2 are widely used to deploy large-scale internet services. The effective use of these parallel computing resources, however, is predicated on the programmer being able to express correct and scalable concurrent programs. But developing correct concurrent programs is hard due to subtle concurrency bugs, which often manifest only at scale, in production environments. Hence, now, more than ever, it is critical that the research community strive towards making parallel and distributed computing more accessible to non-expert programmers.

The goal of my research is to equip programmers with a set of tools to express scalable parallel programs without suffering from the pitfalls of concurrency in parallel and distributed settings. To this end, as a part of my PhD research, I have built MultiMLton, a parallel and distributed extension of MLton Standard ML compiler and runtime, which has been part of several successful research projects [1, 2, 3, 4, 5, 6]. In addition, during my internship at Microsoft Research, I have developed a new concurrency substrate for Glasgow Haskell Compiler (GHC), that allows Haskell programmers to safely and composably describe schedulers for Haskell threads in Haskell on multi-core architectures [7]. These experiences have given me the necessary expertise to realize my ideas in complex, real-world software systems.

Since concurrency abstractions are notoriously slippery topic, in all my ventures, I strive to formally specify their semantics. The Formal semantics not only enables verifying the correctness of the abstractions, but also precisely captures the contracts to the clients of the concurrency library. Thus, I bring forth to a research program, a unique mix of programming language expertise and strong system implementation skills. I seek a research position that will help me leverage these skills, and further my research goals.

Previous Work

MultiMLton is a Standard ML compiler and runtime environment that targets scalable platforms such as manycore processors and compute clouds. The goal of MultiMLton is to allow the programmer to declaratively develop concurrent programs, without worrying about architectural nuances and achieving good scalability. There are numerous challenges to realizing these goals, some of which I have addressed as a part of my PhD research.

Featherweight threads for communication. The programming model of MultiMLton is a mostly functional language combined with lightweight threading, where threads primarily interact over first-class message-passing channels. On manycore systems, the cost of creating and managing interactive asynchronous computations can be substantial due to the scheduler and memory management overheads.

My work in DAMP'10 [1] (and in submission to JFP [8]), describes a novel parasitic threading system for short-lived interactive asynchronous computation. The key feature of parasitic threads is that they execute only when attached to a host lightweight thread, sharing the stack space, and hence amortizing the overheads. In the typical case, the cost of a parasitic thread is similar to the cost of a non-tail function call. Further, I have precisely formalized the interaction of parasitic threads and their hosts using formal operational semantics. In MultiMLton, parasitic threads play a key role in the efficient realization of composable asynchronous and heterogeneous protocols [4].

Programming non-cache-coherent systems. With ever increasing core count on a chip, processors are soon to hit a "coherence wall", where the performance and design complexity of cache coherence hardware limits multicore scalability. Architectures, such as Intel's 48-core Single-chip Cloud Computer (SCC), completely shun hardware cache coherence, and instead, provide fast hardware message-passing interconnect and the ability to manage coherence in software. Software based cache coherence, however, is typically disabled as it is difficult for the programmer to get it right without inducing unnecessary cache misses. Although a shared memory system, SCC is typically programmed as a cluster of machines on a chip, due to the lack of suitable abstractions to deal with absence of cache coherence. The question is

whether we can efficiently hide the absence of cache coherence from the programmer, and provide a semblance of global cache-coherent memory.

My ISMM'12 [2] work describes a novel runtime system for MultiMLton, which utilizes a core-local partitioned heap scheme to circumvent the absence of cache coherence. In addition to the core-local heaps, there exists a global uncached shared heap to which shared objects are automatically promoted. The runtime system uses object shape analysis and ample concurrency as a resource to eliminate barrier overheads associated with core-local heap schemes. This runtime system not only enables shared memory programming of non-cache-coherent systems, but by enabling concurrent core-local heap collections, exhibits immense scalability benefits even on cache-coherent architectures such as 48-core AMD magny-cours and 864-core Azul Vega3 architectures.

My MARC'12 [3] work describes an extension of this design to exploit software support for cache coherence and hardware interconnect for inter-thread message passing. The key novelty is partitioning the shared heap into mutable and immutable objects, and selectively enabling cache coherence only for immutable object heap. Since immutable objects by definition do not change after initialization, cache invalidations and flushes can be efficiently issued as a part of the extant memory barriers. Across our benchmarks, we measured that this runtime system design enables more than 99% of the memory access to be potentially cached. This work won the ***Best Paper Award*** at MARC'12.

Optimistic synchronization. A mostly functional programming language combined with synchronous message passing offers an attractive model for expressing concurrency. In particular, synchronous communication simplifies program reasoning by combining data transfer and synchronization into a single atomic unit. However, in a high-latency environment like the cloud, the synchrony is at odds with high latency, whereas switching to an explicit asynchronous programming model complicates reasoning. The question is whether we can express programs for high-latency environments synchronously, but speculatively discharge them asynchronously, and ensure that the observable behavior mirrors that of the original synchronous program.

My PADL'14 [9] work describes a prescription for safely relaxing synchrony. The key discovery is that the necessary and sufficient condition for divergent behavior (mis-speculation) is the presence of happens-before cycle in the dependence relation between communication actions. I have proved this over an axiomatic formulation that precisely captures the semantics of speculative execution. Utilizing this idea, I have built an optimistic concurrency control mechanism for concurrent ML programs, on top of MultiMLton, capable of running in compute clouds. The implementation uses a novel un-coordinated checkpoint-recovery mechanism to detect and remediate mis-speculations. Our experiments on Amazon EC2 validate our thesis that this technique is quite useful in practice.

Composable user-level schedulers. The runtime system (RTS) for a modern, concurrent, garbage collected language like MultiMLton or Haskell is like an operating system: sophisticated, complex, performant, but alas very hard to change. If more of the RTS were in the high-level language, it would become far more modular and malleable. In particular, we would like to allow the language practitioners to *safely* gain control over components typically considered to belong to the RTS, without compromising correctness and performance.

One example of such a component is the multicore thread scheduler. Typically, high-level languages provide limited control over the scheduler to the programmer, as they tend to interact in subtle ways with other RTS components such as memory manager, foreign function interface, concurrency libraries, etc. However, with the recent trend towards heterogeneous, non-uniform memory multi-core processors, and the emergence of new programming models such as data parallel programming, there is greater need to allow programs to have more control over how its threads are scheduled.

During my internship at Microsoft Research, improving upon Li et al. [10]'s earlier work, I developed a new concurrency substrate for GHC [7], which allows the scheduler for concurrent and parallel programs to be written in Haskell. In particular, compared to Li et al.'s design, the new substrate allows concurrency primitives to be constructed *modularly*, obviating the need to re-engineer or reason about the interactions with GHC's existing concurrency support.

The novel contribution of this work is the abstraction of user-level scheduler interface through scheduler activations [11], and the use of software transactional memory (STM) to ensure safety of user defined schedulers in a multicore environment. The design retains the key, performance-critical functionalities such as foreign function call support, transactional memory implementation, etc., in the RTS. The RTS and the user-level concurrency libraries interact with the scheduler using the same activation interface, thus allowing modular reasoning.

Concurrency primitives and their interactions with the RTS are particularly tricky to specify and reason about. In keeping with my research methodology, I have formalized not only the concurrency substrate primitives, but also their interaction with the RTS components. This formalization served as a good validation mechanism, and helped me gain insights into subtle interactions between the concurrency substrate and the RTS.

Ongoing Work

Programming eventually consistent systems. My research into improving programmability of non-cache-coherent shared memory architectures has led me to explore distributed stores. Here, similar to a non-cache-coherent system, the semblance of a global, coherent shared memory does not exist. Many geo-distributed stores rely on replication of data items at multiple sites in order to achieve high availability and partition tolerance. Due to CAP theorem [12], a highly available, partition tolerant distributed store can only offer eventual consistency. Eventual consistency, however, is a catch-all term, and Burckhardt et al. [13] show that different forms of eventual consistency exist in literature, with varied performance implications. Distributed stores such as Amazon SimpleDB and Google's App Engine Datastore typically offer a subset of these consistency guarantees. Furthermore, stores typically offer a fixed set of eventually consistent data types such as multi-valued registers and key-value stores, and the program logic must be adapted to utilize the available data types.

For a programmer used to writing shared memory multi-core programs, eventual consistency is strictly more difficult to work with. This is due to (1) the lack of support for describing new eventually consistent data types and (2) the lack of abstractions to composably strengthen consistency guarantees. To address these issues, I am building an embedded domain specific language (EDSL) for Haskell to describe replicated data types (RDTs) in the same vein as abstract data types. The RDT specification is described over a trace of operations on an object, and is intended to deterministically resolve conflicting updates. The specification is compiled to (1) an optimized store side implementation of the new data type, and (2) client side stubs to instantiate and manipulate objects of the new type. To request stronger forms of eventually consistent operations on demand, a set of composable consistency strengthening combinators are provided to the clients of distributed store. I plan to capture the interplay between the different consistency strengthening combinators through an axiomatic specification over traces.

Future Directions

In my future work, I wish to expand on my current lines of inquiry, as well as explore new problems that fall under the domain of scalable software systems. At Microsoft Research Cambridge, I see synergy between my interests and the work done in Programming Principles and Tools (PPT) and Cambridge Systems and Networking (CamSys) groups.

Safer, more robust user-level schedulers. While scheduler activations for Haskell [7] simplify the task of building user-level schedulers, there is definitely room for improvement in terms of safety. In particular, for performance reasons, the activation interface operates directly over Haskell thread objects. Unfortunately, the onus is on the scheduler writer to ensure that the thread objects are linearly used. As more of the Haskell RTS components are implemented in Haskell and exposed to the programmer, there is a need to ensure linear use of resources such as Haskell threads. Given the rich tradition of Haskell type system research in the PPT group, I would like to explore the use of linear types for safe use of system resources.

Another teething issue in the concurrency substrate design is the interaction between blackholes (long-lived thunks concurrently accessed by multiple parallel threads) and user-level schedulers. To avoid deadlocks, we needed subtle low-level code, which is what we had originally set out to eliminate. The code also occasionally suffers from livelocks where one core waits for progress on a blackholed thunk on a different core. Recent work on higher-order cardinality analysis [14] provides way for a cleaner solution. Can we extend the cardinality analysis to detect when a thunk might potentially be shared between different threads? Instead of blackholing such thunks, it might be beneficial to pay the cost of re-execution. This can not only avoid the livelock problem, but also simplify the low-level code needed to prevent deadlocks.

Synthesizing eventually consistent program from invariants. My ongoing work aims to provide a declarative specification of eventually consistent data types and a set of combinators to strengthen the consistency guarantees. While an axiomatic specification encompassing the different combinators helps from a program verification perspective, it might be onerous for the non-expert programmer to reason about the safety properties of the store as a whole. Ideally, we would like the programmer to *declaratively* specify the invariants associated with the data type, along with the operation specification. For example, one might wish to express that the balance in a replicated bank account never drops below zero. The task would then be to discover the *weakest* consistency annotation that preserves the invariants.

Recent work by Gleissenthall et al. [15] studies consistency guarantees in terms of epistemic logic as opposed to trace based axiomatic reasoning. This is a fascinating alternative take on reasoning about eventual consistency, and the

question is whether we can discover language abstractions that express complex data type invariants in terms of partial knowledge at some client.

Approximate in-memory data analytics. It is often the case in big data processing scenarios that input data is unstructured, ambiguous or incomplete. While the amount of data grows with time, deriving insights into the available data becomes harder due to the lack of knowledge about data semantics and distribution. Ideally, to gain insights into the data, we would like big data enthusiasts and experts alike to be able to infer the semantics of the data, run interactive jobs on a meaningful sample of the data, expressed using high-level constructs similar to R, Excel and Matlab. These high-level constructs can express sophisticated algorithms such as matrix decomposition and eigenvalue calculation in tens of lines of code, where as a system like Hadoop needs hundreds.

In my vision, the problem can roughly be split into developing (1) a language for declaratively specifying the assumptions about the data, and a method for meaningful sampling, (2) efficiently consuming the input data from distributed stores, and (3) a system to perform fault-tolerant in-memory iterative computations, potentially on mutable data. For (1), I wish to explore the applicability of probabilistic programming languages to big data analytics. Recent work by Gkantsidis et al. [16] shows how to perform effective data filtering at storage nodes in order to reduce network bandwidth utilization while fetching data to compute nodes. The system works by statically analyzing the jobs to derive a filtering strategy, which can aid with (2). I wish to expand upon my ongoing work by incorporating a compute model to address (3).

References

[1] KC Sivaramakrishnan, Lukasz Ziarek, Raghavendra Prasad, and Suresh Jagannathan, "Lightweight Asynchrony using Parasitic Threads," in *DAMP*, pp. 63–72, 2010.

[2] KC Sivaramakrishnan, Lukasz Ziarek, and Suresh Jagannathan, "Eliminating Read Barriers through Procrastination and Cleanliness," in *ISMM*, pp. 49–60, 2012.

[3] KC Sivaramakrishnan, Lukasz Ziarek, and Suresh Jagannathan, "A Coherent and Managed Runtime for ML on the SCC," in *Proceedings of the Many-core Applications Research Community (MARC) Symposium at RWTH Aachen University*, pp. 20–25, Nov. 2012.

[4] Lukasz Ziarek, KC Sivaramakrishnan, and Suresh Jagannathan, "Composable Asynchronous Events," in *PLDI*, pp. 628–639, 2011.

[5] Lukasz Ziarek, KC Sivaramakrishnan, and Suresh Jagannathan, "Partial Memoization of Concurrency and Communication," in *ICFP*, pp. 161–172, 2009.

[6] Lukasz Ziarek, Siddharth Tiwary, and Suresh Jagannathan, "Isolating Determinism in Multi-threaded Programs," in *RV*, pp. 63–77, 2011.

[7] KC Sivaramakrishnan, Tim Harris, Simon Marlow, and Simon Peyton Jones, "Composable Scheduler Activations for Haskell," tech. rep., Microsoft Research, 2013.

[8] KC Sivaramakrishnan, Lukasz Ziarek, and Suresh Jagannathan, "MultiMLton: A Multicore-aware Runtime for Standard ML," tech. rep., Purdue University, 2013.

[9] Lukasz Ziarek, KC Sivaramakrishnan, and Suresh Jagannathan, "Rx-CML: A Prescription for Safely Relaxing Synchrony," in *PADL*, 2014.

[10] Peng Li, Simon Marlow, Simon Peyton Jones, and Andrew Tolmach, "Lightweight concurrency primitives for ghc," in *Haskell Symposium*, pp. 107–118, 2007.

[11] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy, "Scheduler activations: effective kernel support for the user-level management of parallelism," *ACM Trans. Comput. Syst.*, vol. 10, pp. 53–79, Feb. 1992.

[12] Seth Gilbert and Nancy Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *SIGACT News*, vol. 33, pp. 51–59, June 2002.

[13] Sebastian Burckhardt, Alexy Gotsman, and Hongseok Yang, "Replicated data types: Specification, verification, and optimality," in *POPL*, 2014.

[14] Ilya Sergey, Dimitrios Vytiniotis, and Simon Peyton Jones, "Higher-order cardinality analysis," in *POPL*, 2014.

[15] Klaus von Gleissenthall and Andrey Rybalchenko, "An epistemic perspective on consistency of concurrent computations," in *CONCUR*, 2013.

[16] Christos Gkantsidis, Dimitrios Vytiniotis, Orion Hodson, Dushyanth Narayanan, Florin Dinu, and Antony Rowstron, "Rhea: Automatic filtering for unstructured cloud storage," in *NSDI*, pp. 343–356, 2013.