In recent years, there has been a widespread adoption of both multicore and cloud computing. Multicore processors have become the norm in mobile, desktop and enterprise computing, with an increasing number of cores being fitted on a chip with every successive generation. Cloud computing has paved the way for companies to rent farms of such multicore processors on a pay-per-use basis, with the ability to dynamically scale on demand. Indeed, many real-world services for communication, governance, commerce, education, entertainment, etc., are routinely exposed as a web-service that runs in third-party cloud compute platforms such as Windows Azure and Amazon's EC2. These services tend to be concurrently accessed by millions of users, increasingly through multicore-capable mobile and desktop devices.

As a result, there are a growing proportion of developers who must tackle the complexity of programming for a cloud of multicore processors. In particular, they must ensure correct application behavior in the face of asynchrony, concurrency, and partial failures, all the while providing good scalability as well as minimizing the user's perception of latency. Reasoning about concurrent programs is inherently a difficult endeavor. For each new concurrent thread of control added to a program, there is an exponential growth in the number of potential schedules. This greatly increases the chance of subtle concurrency bugs evading the programmer during software development and testing, only to appear in production environments with devastating consequences. As we head into multicore compute platforms with 1000+ cores, the existing ad hoc programming models and systems are simply not equipped for safe and scalable concurrent programming, where the emerging concerns such as heterogeneity in hardware, partial failures of cores, and secure computation with third-party code and infrastructure warrant a holistic approach to software development.

The goal of my current research is to improve the state of the art in developing correct and scalable programs for loosely coupled massively scalable systems through rigorous and declarative programming language abstractions. 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. I aim to address several key challenges crosscutting disparate domains, such as (i) developing concurrent programming platforms to enable building applications that seamlessly scale on heterogeneous compute clouds, (ii) rigorously verifying the correctness of parallel and distributed software, to eliminate concurrency and security bugs, which are notoriously difficult to detect in large-scale deployments, and (iii) incorporating emerging concerns such as security and energy into the programming model.

Previous Work

Towards this goal, as a part of my PhD thesis, I have developed the MultiMLton platform, 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, 7]. My work on MultiMLton has been recognized by Purdue University with the 2014 Maurice H. Halstead award for outstanding research in Software Engineering and a best paper award at the 2012 Many-core Architecture Research Community (MARC) symposium at RWTH Aachen. The goal of MultiMLton is to allow the programmer to declaratively develop concurrent programs, without worrying about the issues of 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.

Programming model. 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 JFP'14 [7] and DAMP'10 [1] 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 MultiMLton, parasitic threads play a key role in the efficient realization of composable asynchronous and heterogeneous communication protocols [4].

Programming next-generation many-core processors. 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. 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 program this *cloud* of cores on a chip in the way we program shared-memory multicore processors. Such a system will greatly simplify developing programs for the massively scalable SCC processor.

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. 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. Across our benchmarks, we measured that this new 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.

Migrating to the cloud. MultiMLton programming model combines the benefit of functional programming with synchronous message passing, and offers an attractive model for expressing concurrency. 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 [8] work identified 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. 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. Our extensive experiments on Amazon EC2 validate our thesis that this technique is quite useful in practice.

Apart from my PhD research, I have the opportunity to explore the questions relating to concurrent programming for multicore systems during my internships with industrial research labs. At Microsoft Research Cambridge, 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 scalable multi-core architectures [9]. At Samsung Research, I prototyped the compiler analysis and the runtime system for new clean-slate manycore programming language [10]. These experiences have provided me insights into the issues are faced by developers in today's parallel compute systems, and also the necessary expertise to realize my ideas in complex, real-world software platforms.

Current Work

After finishing my PhD [11] at Purdue University, I joined University of Cambridge as a Research Associate under the OCaml Labs initiative where I continue to explore the development of concurrent and parallel functional programming language systems. I lead the Multicore OCaml project which aims to add concurrent and parallel programming support for the industrial strength OCaml programming language.

A particularly distinguishing feature of this effort is that instead of baking the concurrency support into the compiler and exposing libraries for concurrent programming, we extend OCaml with support for lineaer algebraic effects and handlers [12]. Algebraic effects and handlers provide a modular abstraction for expressing effectful computation, allowing the programmer to separate the expression of an effectful computation from its implementation. In this system, concurrency is just another effect, modularly expressed through algebraic effects and their handlers. This design allows the programmer to describe thread schedulers and concurrent data structures as OCaml libraries, keeping the compiler lean, while providing the flexibility of swapping in alternate but equivalent implementations. Given that OCaml has an excellent Javascript backend [13], the addition of concurrency support to OCaml has the potential to greatly simplify the implementation of highly-concurrent real-world web-services.

Future Work

In my future work, I wish to build on the MultiMLton programming model to address the emerging and future challenges in programming manycore and large-scale distributed clusters. Nowadays, an increasing number of companies deploy their web-services on geo-distributed third-party cloud platforms such as Amazon EC2, Windows Azure, Heroku, Google App Engine, etc. These cloud platforms are becoming more heterogenous incorporating many-core general-purpose processors (potentially with multiple *coherence domains*) with GPGPUs and FPGAs, providing the ability to offload expensive computation to specialized hardware. Traditionally, developers have relied on the infrastructure providing strong consistency guarantees such as sequential consistency, linearizability and distributed transactions in order to build correct applications for scalable platforms. Strong consistency provides a semblance of a single consistent view of memory, where the operations appear to be performed sequentially in some linear order. Unfortunately, providing such strong guarantees in unviable in the face of modern multicore processors and geo-distributed massively scalable services.

Modern multicore processors and concurrent programming languages expose subtle relaxed memory behaviors that arise from hardware and compiler optimizations to the developer. The state of the art distributed stores also resort to weak consistency guarantees, where the different geo-distributed replicas eventually converge to a uniform state at some arbitrary point in the future. In particular, no assumptions can be made about the global state of the system at any particular time. Such weakly consistent systems make a well-known trade-off: developers avoid the cost associated with latency, contention and availability to achieve strong consistency, in exchange for weaker consistency guarantees about the data; the semblance of a uniform consistent view of memory is lost. Additionally, since the web applications run on third-party compute platforms, the issue of data privacy and security has also become a concern. In the cloud, an application typically shares resources such as disk, network and processor time with applications from other customers, running on the same server. Since critical services such as health-care and banking services are increasingly offered online, any vulnerability in the software in such an environment could lead to a potentially serious security breach. Existing programming models for these platforms are ill-equipped to address these consistency and security challenges.

To address these issues, I have developed Quelea [14], a rigorous programming model that elegantly combines shared-memory parallelism with datacenter-wide distribution. Quelea programming model is exposed as a domain-specific language for describing the semantics of replicated datatypes in the same vein as abstract datatypes, along with a novel contract language for declaratively expressing the consistency and security obligations. This clean separation of consistency and security properties from the datatype semantics permits the same program to be compiled for a variety of backends including multicore processors and distributed stores.

A key aim of Quelea is to remain portable and agnostic to any particular implementation. The developers describe what minimal consistency guarantees are required for the application, but not how to achieve them, which is the concern of individual backend implementations. The separation of consistency concerns and datatype semantics allows Quelea to be realized as a shim layer on top of a wide variety of eventually consistent data stores. The stores provide availability, durability and distribution guarantees, while our system simply upgrades the safety properties. This layered approach is geared towards interoperability and broad applicability, providing clearer semantics and stronger safety properties for existing software systems.

The declarative description of safety properties in Quelea lends itself to rigorous formal verification of distributed software, eliminating bugs and vulnerabilities prior to deployment, preventing privacy and monetary losses associated with software bugs in the wild. I intend to build Quelea on top of the highly successful Mirage Unikernel, which is part XenServer and Linux Foundation, with millions of deployments in the cloud already. Mirage project is led from the Cambridge Computer Lab, where I have chosen to pursue my post-doctoral research. A symbiotic relationship with Mirage will help rapid adoption of Quelea, and greatly impact the way in which cloud services are built and deployed.

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, Lukasz Ziarek, and Suresh Jagannathan, “MultiMLton: A Multicore-aware Runtime for Standard ML,” *Journal of Functional Programming*, 2014.

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

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

[10] Daniel G Waddington, Chen Tian, and KC Sivaramakrishnan, “Scalable lightweight task management for mimd processors,” *EuroSys workshop, Systems for Future Multicore Architectures (SFMA 2011)*, pp. 1–6, 2011.

[11] KC Sivaramakrishnan, *Functional Programming Abstractions for Weakly Consistent Systems*. PhD thesis, Purdue University, West Lafayette, IN, USA, 2014.

[12] Stephen Dolan, Leo White, KC Sivaramakrishnan, Jeremy Yallop, and Anil Madhavapeddy, “Effective Concurrency through Algebraic Effects,” in *The OCaml Users and Developers Workshop*, OCaml ’15, (Vancouver, Canada), 2015.

[13] Jrme Vouillon and Vincent Balat, “From Bytecode to JavaScript: The Js_of_ocaml Compiler,” *Software: Practice and Experience*, vol. 44, no. 8, pp. 951–972, 2014.

[14] KC Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan, “Declarative Programming over Eventually Consistent Data Stores,” in *PLDI*, 2015.