Over the last decade, software engineering has seen significant advancements and shifts driven by technological innovation and changing industry demands. Emerging applications such as internet-of-things, augmented reality and self-driving vehicles have necessitated moving computation closer to the data for real-time decision making, while also depending on cloud computing platforms for AI models used to make those decisions. In order to support these varied applications, the computing platforms have become heterogeneous, with a mix of CPUs, GPUs and FPGAs, both on the server and the client sides. The increasing complexity and ubiquity of software systems have also led to new security and privacy challenges.

As software continues to eat the world[1], a growing population of software developers are faced with the challenge of building correct, secure and scalable software systems that can run on a wide range of platforms. They must ensure correct application behaviour the in face of asynchrony and partial failures, ensure absence of security and privacy issues arising anywhere from programming errors to malicious attacks, all the while providing good scalability as well as minimizing the user's perception of latency. This remains an uphill task with current programming language technology. No wonder then that bugs and exploits evade the programmer during software development and testing, only to appear in production environments with devastating consequences.

My research goal is to ***develop programming language abstractions and tools that empower developers to build secure, scalable and reliable software systems***. I believe that mathematically rigorous functional programming is particularly suited towards this goal. A distinguishing aspect of my research is that I spend significant effort to make these abstractions and tools available to practitioners and thereby bridging the gap between research and practice.

1 Previous work

1.1 Concurrency and Parallelism

In my research career so far, I have developed concurrent and parallel programming abstractions for widely used functional programming language compilers including the MLton Standard ML compiler [9–12, 14, 15], the Glasgow Haskell Compiler [7] and OCaml [5, 6]. A notable aspect of these works is that they are implemented in industrial-strength compilers, and some of them are widely used by practitioners.

During my PhD studies, I led the MultiMLton project, a parallel extension of the MLton Standard ML compiler, targeted at future many core processors. In MultiMLton, concurrent programs are organised as a large number of cooperative lightweight threads, that communicate by passing messages between each other. I developed a novel asynchronous communication abstraction and a mostly-concurrent garbage collector [11] that allowed MultiMLton to scale to the 864-core Azul Vega3 machine. MultiMLton's multicore garbage collector (GC) was designed to minimise inter-core communication. The key innovation was to trade some of the ample concurrency in the source language to offset some of the GC costs [10].

MultiMLton was designed not only for traditional cache-coherent multicore machines, but also to take advantage of exotic architectures that provided fine-grained control over caches. I developed a port of MultiMLton to the non-cache-coherent Intel Single-chip Cloud Computer (SCC), which preserved the familiar shared memory parallel programming model [9]. This work took advantage of MultiMLton's ability to statically distinguish mutable and immutable data to manage them in separate cache coherence domains. This work won the ***Best Paper Award at the Intel Many-core Architecture Community (MARC) Symposium*** at RWTH, Aachen, Germany. My work on MultiMLton was recognised by Purdue University with the ***Maurice H Halstead award for outstanding research in software engineering***.

[1] https://a16z.com/why-software-is-eating-the-world/

After my PhD, I joined the University of Cambridge Computer Lab as an 1851 Royal Commission and Darwin College Research Fellow, where I turned my attention to bring native support for concurrency and parallelism to the OCaml programming language as part of the Multicore OCaml project. Despite being one of the most popular functional programming languages, OCaml lacked support for native concurrency and parallelism. For concurrency, we introduced *effect handlers* into the language [6]. Effect handlers are a mechanism for programming with user-defined effects. Operationally, effect handlers provide a mechanism for structured programming over delimited continuations. Effect handlers generalise mechanisms such as exceptions, generators, async/await and lightweight threads, which are provided as primitives by other languages. With the addition of effect handlers, we are now able to implement these rich mechanisms as libraries with well-defined semantics for their interactions.

For parallelism, we have rewritten major parts of the runtime system of OCaml to be parallelism safe, and introduced a new mostly concurrent garbage collector that can scale to 100s of cores [5]. This is design minimises stop-the-world phase where all the cores are stopped running OCaml code and the garbage collector runs. This is garbage collector is particularly suited for latency-sensitive programs that OCaml is often used for such as trading systems, user interfaces, and network-facing micro-services. Programs running on multicore processors also exhibit non-trivial behaviours due to reordering by modern multi-core hardware and compiler optimisations. We have developed a novel *relaxed memory model* for OCaml that offers local reasoning about program fragments unlike the global reasoning required by Java and C11 memory models [1]. This memory model has been implemented in the OCaml compiler for all the supported multicore architectures including x86, ARM, PowerPC and RISC-V.

A major challenge with introducing concurrency and parallelism to a widely used programming language is the existence millions of lines of legacy code, most of which may remain sequential forever. We face the challenge of maintaining backwards compatibility–not just in terms of the language features but also the performance of single-threaded code. We have succeeded in achieving this goal, and Multicore OCaml project has been merged into the mainstream OCaml compiler and released as part of OCaml 5[2]. This makes OCaml the first industrial-strength language to support effect handlers. The work on Multicore OCaml has been recognised with the ***2023 SIGPLAN Programming Languages Software Award*** and a ***distinguished paper award at ICFP 2020*** for the paper that describes the GC design [5]. I am one of the core maintainers of OCaml, and continue to contribute to the development and maintenance of the concurrency and parallelism features.

Multicore OCaml has also had an enormous impact on the wider community. React, the most widely used JavaScript UI framework in the world introduced a major feature called *React Hooks*, which is directly inspired from Multicore OCaml[3]. WebAssembly (Wasm) is a type-safe, efficient language for the web, introduced as an alternative to JavaScript. All major browsers now support Wasm. Wasm is introducing effect handlers for concurrency based on the OCaml design (WasmFX) [4]. WasmFX brings the benefit of effect handlers to every language that can compile to Wasm. I continue to participate in the Wasm community group to advance the WasmFX proposal.

1.2 Distribution

As a natural extension of concurrent and parallel programming, I am fascinated with the challenges with distributed programming. Unlike parallel programming, in loosely-coupled asynchronous distributed systems (LADS), synchronisation leads to unavailability, and hence, is avoided when possible. This makes programming LADS challenging. Unfortunately, this has led to centralisation of Internet services, where a few large corporations provide services to billions of users. The

[2] https://github.com/ocaml/ocaml/pull/10831
[3] https://legacy.reactjs.org/docs/hooks-faq.html#what-is-the-prior-art-for-hooks

personal data owned by these large corporations is a major security and privacy concern. In order to build a resilient and decentralised Internet economy, distributed programming needs to be made easier to enable building *local-first software*[4].

My research in this area has been focussed on correctly building software that works under weaker consistency guarantees in LADS. To alleviate the burden of developing correct programs, I developed Quelea [8], a programming model that associates user program with declarative contracts for their consistency expectation. Quelea utilises automated theorem proving tools to automatically and correctly insert the necessary coordination to ensure that application's consistency expectations are preserved.

In order to reason about state changes in LADS, functional programming principles of immutability and persistence turn out to be particularly useful. Based on this observation, I have developed mergeable replicated data types (MRDTs) [2]. MRDTs are based on the principles of distributed version control systems such as Git where the causal execution history is captured through branches and merges. A key aspect of MRDTs is that the distribution aspects of the data type are separated from their sequential behaviour, leading to simpler concurrent reasoning and efficient implementations. The immutability and persistence also enables verified, correct-by-construction MRDTs (Peepul) [13]. The MRDTs are implemented and verified using F*, a proof-oriented programming language. The verified MRDT implementations are extracted to OCaml and are compatible with Irmin, a Git-like distributed database.

2 Current work

Since moving to IIT Madras and then subsequently leading the technology team at Tarides[5], I have continued the work towards my goal of empowering developers to develop correct, secure and scalable software, now with a team of research scholars, colleagues at IIT Madras and programming language experts and engineers at Tarides.

Automated verification of MRDTs. One of the downsides of Peepul work on correct-by-construction MRDTs is that developer needs to write down a specification for each MRDT along with a simulation relation that connects the specification with the implementation. The verification process machine-assisted but is a manual one. This is tedious and error-prone. As a follow up to the Peepul work, we asked whether we can automate the proof of correctness of MRDTs. We have managed to do this by couching the correctness argument in *linearizability*, a well-understood concurrent programming correctness criterion. Our definition of linearizability ensures both eventual consistency and full functional correctness, while also allowing a simple specification framework for conflict resolution in MRDTs. We have successfully applied our approach on a number of complex MRDT implementations.

Securing the foundations of Unikernels. It is well-understood these days that memory-unsafe languages such as C and C++ lead to majority of security vulnerabilities in widely-used software[6]. This has prompted a major push towards memory-safe languages such as Rust and OCaml. However, the large legacy code base in unsafe C and C++ will continue to remain even as new code gets written in memory-safe languages. How can we allow mixed safe and unsafe language codebases to be secure, especially when unsafe languages can violate safe language guarantees when combined together in the same application? To this end, we are developing FIDES, a secure extension of the Shakti RISC-V processor that prevents security exploits in unsafe code through hardware guards while permitting safe code to run as fast as possible. FIDES also supports inter-process compartments to enforce isolation between different parts of the application.

[4] https://www.inkandswitch.com/local-first/

[5] https://tarides.com

[6] https://www.cisa.gov/news-events/news/urgent-need-memory-safety-software-products

The secure processor is designed to run MirageOS, an OCaml-based library operating system for constructing Unikernels. The aim is to target critical embedded system applications such as remote voting machines[7] and point-of-sale terminals. In general, I see enormous potential for open-source secure hardware such as Shakti processors to complement the open-source secure software such as MirageOS to help build trustworthy systems.

Mechanically verified garbage collectors. In my experience building multiple concurrent garbage collectors (GC), debugging rare and non-deterministic failures take up significant amount of development time. Garbage collectors are written in memory-unsafe languages and bugs in the GCs can lead to security vulnerabilities in the safe language code. Can we build practical, correct-by-construction, mechanically verified GCs? To this end, we have built a correct-by-construction stop-the-world mark-and-sweep GC in the F* programming language and extracted that to C using the KaRaMel toolchain. The GC includes enough features that it can be used as a replacement for the GC in the OCaml programming language and does not compromise on the performance.

All of the research listed above is being conducted in collaboration with IIT Madras colleagues and PhD students, and the corresponding papers are under submission.

3 Future work

In the future, I plan to continue to push towards my goal of empowering developers to build correct, scalable and secure software. I see a number of recent developments that will accelerate the push towards this goal.

Securing the foundations with OCaml Oxide. Oxidising OCaml project [3] brings Rust-style ownership and borrowing to OCaml. In particular, it brings in the ability to reason about the lifetimes of objects statically, which turns out to be a useful building block for a variety of features. I plan to use the Oxidising OCaml project bring effect safety to effect handlers in OCaml 5, stricter data sharing policies in FIDES compartments and build better concurrent data structures that can take advantage of static knowledge of object sharing between different threads.

Trustworthy CodeLLMs. CodeLLMs are specialized language models designed and trained to understand, generate, and reason about programming code. A major challenge in codeLLMs is to ensure that the generated code satisfies the programmer's intent. This is particularly challenging as the intent keeps evolving as the functional requirements of the software changes. How can we ensure that the generated code is correct? How can we ensure that the generated code can be evolved? I believe that lightweight formal method techniques such as static type systems, property-based testing and example-drive development can be profitably used to capture programmer intent and to derive the specifications. Once we have the specifications, codeLLMs can themselves be used to generate the correctness proofs to be machine checked in a proof assistant such as F* or Coq. Today codeLLMs do quite well on high-resource languages such as Python and JavaScript. If the current trend in codeLLMs continues, programming languages will cease to be the dominant interface for programming. I conjecture that, at this point, mathematically rigorous functional programming languages will be preferred target language for codeLLMs as they are today the preferred target for proof assistants.

The ideas from functional programming languages are becoming mainstream as they are incorporated into mainstream languages. I am fortunate and excited to be at the forefront, driving some of these changes, as we empower developers to build correct, secure, and scalable software.

[7] https://pib.gov.in/PressReleasePage.aspx?PRID=1887248

References

[1] Stephen Dolan, KC Sivaramakrishnan, and Anil Madhavapeddy. 2018. Bounding data races in space and time. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 242–255. https://doi.org/10.1145/3192366.3192421

[2] Gowtham Kaki, Swarn Priya, KC Sivaramakrishnan, and Suresh Jagannathan. 2019. Mergeable replicated data types. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 154 (oct 2019), 29 pages. https://doi.org/10.1145/3360580

[3] Anton Lorenzen, Leo White, Stephen Dolan, Richard A. Eisenberg, and Sam Lindley. 2024. Oxidising OCaml with Modal Memory Management. *Proc. ACM Program. Lang.* 4, ICFP (sep 2024), 30 pages. https://homepages.inf.ed.ac.uk/slindley/papers/mode-inference-draft-feb2024.pdf

[4] Luna Phipps-Costin, Andreas Rossberg, Arjun Guha, Daan Leijen, Daniel Hillerström, KC Sivaramakrishnan, Matija Pretnar, and Sam Lindley. 2023. Continuing WebAssembly with Effect Handlers. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 238 (oct 2023), 26 pages. https://doi.org/10.1145/3622814

[5] KC Sivaramakrishnan, Stephen Dolan, Leo White, Sadiq Jaffer, Tom Kelly, Anmol Sahoo, Sudha Parimala, Atul Dhiman, and Anil Madhavapeddy. 2020. Retrofitting parallelism onto OCaml. *Proc. ACM Program. Lang.* 4, ICFP, Article 113 (aug 2020), 30 pages. https://doi.org/10.1145/3408995

[6] KC Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy. 2021. Retrofitting effect handlers onto OCaml. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 206–221. https://doi.org/10.1145/3453483.3454039

[7] KC Sivaramakrishnan, Tim Harris, Simon Marlow, and Simon Peyton Jones. 2016. Composable scheduler activations for Haskell. *Journal of Functional Programming* 26 (2016), e9. https://doi.org/10.1017/S0956796816000071

[8] KC Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. 2015. Declarative programming over eventually consistent data stores. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 413–424. https://doi.org/10.1145/2737924.2737981

[9] KC Sivaramakrishnan, Lukasz Ziarek, and Suresh Jagannathan. 2012. 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*. 20–25.

[10] KC Sivaramakrishnan, Lukasz Ziarek, and Suresh Jagannathan. 2012. Eliminating read barriers through procrastination and cleanliness. In *Proceedings of the 2012 International Symposium on Memory Management* (Beijing, China) *(ISMM '12)*. Association for Computing Machinery, New York, NY, USA, 49–60. https://doi.org/10.1145/2258996.2259005

[11] KC Sivaramakrishnan, Lukasz Ziarek, and Suresh Jagannathan. 2014. MultiMLton: A Multicore-aware Runtime for Standard ML. *Journal of Functional Programming* (2014).

[12] KC Sivaramakrishnan, Lukasz Ziarek, Raghavendra Prasad, and Suresh Jagannathan. 2010. Lightweight asynchrony using parasitic threads. In *Proceedings of the 5th ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming* (Madrid, Spain) *(DAMP '10)*. Association for Computing Machinery, New York, NY, USA, 63–72. https://doi.org/10.1145/1708046.1708059

[13] Vimala Soundarapandian, Adharsh Kamath, Kartik Nagar, and KC Sivaramakrishnan. 2022. Certified mergeable replicated data types. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 332–347. https://doi.org/10.1145/3519939.3523735

[14] Lukasz Ziarek, KC Sivaramakrishnan, and Suresh Jagannathan. 2009. Partial memoization of concurrency and communication. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming* (Edinburgh, Scotland) *(ICFP '09)*. Association for Computing Machinery, New York, NY, USA, 161–172. https://doi.org/10.1145/1596550.1596575

[15] Lukasz Ziarek, KC Sivaramakrishnan, and Suresh Jagannathan. 2011. Composable asynchronous events. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (San Jose, California, USA) *(PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 628–639. https://doi.org/10.1145/1993498.1993572