Research Statement: Usable Abstractions

Stefan Muller

In my research, I study how to make software more reliable, more efficient, and easier to write. To do this, I draw largely on tools from the programming languages community, in which most of my work has fallen. These tools include formal language models, type systems, cost semantics and static analyses. In many cases, however, improving software requires drawing on ideas from other communities: designing an efficient language requires knowledge of computer systems (for memory management and low-level efficiency concerns) and results from the theory community (e.g., to design efficient parallel scheduling algorithms). Furthermore, to make these ideas relevant, I look to the broader computer science community for applications of my work that can have real impact on critical software. For example, many AI and machine learning applications use CUDA to execute code on GPUs, but CUDA’s execution model makes possible a number of performance pitfalls that can be reasoned about using PL techniques.

In this statement, I discuss several of my past research results:

- I have developed a language, together with appropriate type systems and formal reasoning models, for writing massively parallel programs in which threads have different resource requirements (e.g., throughput vs. responsiveness) and different priorities. The language, unlike its predecessors, comes with a model that formally guarantees responsiveness and competes with languages such as Cilk and Go in practice.

- I have taken existing tools for reasoning about the abstract resource usage of typed functional programs and leveraged them, together with low-level worst case execution time tools for binaries, to obtain worst-case execution times for compiled functional programs, which previously would have involved extensive human annotation. This is the first step toward being able to write real-time code in strongly typed high-level languages.

- As part of a DARPA-funded project on verifying safety-critical autonomous systems, I am currently working to develop static analyses to identify and prevent performance bugs in general-purpose GPU kernels.

In my future research, I plan to continue improving the state of the art in programming by developing tools to allow programmers to more easily reason about the resource usage of their code, and enable them to use high-level programming language ideas to write real-world software. If there is one overarching goal in my research, it is to prove that high-level ideas and abstractions can make programs elegant without sacrificing expressiveness or efficiency. These latter properties are often considered to be at odds with abstraction: for example, hiding the implementation of a software module or data structure and using it solely through a well-defined interface makes software more maintainable, but might also obscure important properties about the running time of the abstracted operations. If the running time is important, the answer is not to abandon abstraction altogether but rather to use the right abstractions for the task at hand. In the above example, one might want an interface that specifies, either informally through documentation or formally as part of the types of the interface, the asymptotic complexity of the provided operations. If the task at hand requires more precise information about actual, rather than asymptotic, running time, the abstractions would need to provide correspondingly more information.

I firmly believe that strong abstractions can help researchers and developers in areas such as systems and AI write faster, more robust and more maintainable code, and my research goal is to develop abstractions and programming language techniques that are useful in these domains.

The remainder of this statement expands on the above summary.
Past Work: Combining cooperative and competitive multithreading

In my PhD dissertation, I focused on a class of parallel languages known as cooperative parallel languages. These languages allow programmers to express parallelism using very high-level, lightweight abstractions that scale to millions of threads, which are automatically mapped onto actual hardware by the language runtime. They also admit elegant cost models for reasoning about the parallel running time of programs. These features make cooperative languages quite useful for heavily computational applications such as simulations and data processing. Despite these benefits, cooperative parallel languages are not often used to develop consumer applications, in large part because they are not effective at handling interaction. For example, the cost models consider only the total throughput of the program, which is unacceptable if one thread (say, the thread accepting user input) must be responsive. Instead, consumer programs are generally written with competitive threading techniques such as POSIX threads.

This gap between cooperative and competitive threading is no longer tenable: almost all devices, from servers down to smartphones and even smart watches, now routinely ship with multiple processors. As a result, consumer software will soon have to make use of multiple threads not just for responsiveness (as in competitive threading) but also for the high-throughput parallelism guaranteed by cooperative languages.

My work in this area has produced four results:

• Developing and analyzing a latency-hiding scheduling algorithm that behaves properly in the presence of threads that block on user input and system calls.

• Extending cooperative cost models to reason about responsiveness in addition to throughput and account for the fact that interactive threads must be considered higher priority than strictly computational threads.

• Designing and implementing a cooperative parallel language that allows threads to be annotated with priorities indicating their responsiveness requirements.

• Demonstrating the impact of the above work by applying it to two AI-based case studies as well as developing a C++-based implementation.

Latency-hiding scheduling algorithm. Blocking operations, such as I/O and many other system calls, defeat the scheduling algorithms, often based on techniques such as work stealing, that many cooperative languages use to schedule threads. In many such schedulers, if a program thread blocks (e.g., waiting for user input), it could block the entire system thread. Some existing systems can detect a blocked program thread and put it aside until it is ready to run, allowing that system thread to do useful work in the meantime (thus “hiding the latency” of the blocked program thread), but none of these techniques have been analyzed for their running time. Doing so is non-trivial because existing running time analyses of work stealing depend crucially on the order in which tasks are executed.

In joint work with Umut Acar, I developed and analyzed a work stealing algorithm [1] that provably hides the latency of blocked threads. We analyzed the algorithm and showed that the time it takes to schedule a parallel program does not depend on I/O latency, unless this latency falls on the critical path.

Reasoning about responsiveness and priority. In my work [2, 3], I’ve extended traditional cost models for parallel programs to account for priorities and response times. This builds on prior work in using directed acyclic graphs (DAGs) to model the dependency structure of parallel programs. Given such a graph, results such as Brent’s Theorem [Bre74] bound the time needed to schedule the corresponding programs on a certain number of processors.

My extensions to the prior state of the art allow certain parts of the DAG to be indicated as high-priority and generalize Brent’s Theorem to account for not just the overall computation time of a program but also the response time of high-priority threads. A key result is that the response time of a thread depends only on the amount of work at equal and higher priorities. This result applies as long as the schedule is prompt, that is, it never prioritizes a low-priority operation over a higher-priority one, and the DAG is well-formed, that is, high-priority work does not depend on lower-priority work along its critical path. The well-formedness requirement corresponds nicely with the absence of priority inversions, a classic bug in concurrent software.
I have also extended these results with policies that avoid starvation by guaranteeing certain fractions of processor cycles to lower-priority work (we call such a policy fair) [7]. We have shown both theoretically and empirically that fairness has the expected benefit on low-priority threads (avoiding starvation) without unduly harming high-priority threads.

A cooperative parallel language for prioritized threads. A major contribution of my PhD thesis work was a language called PriML, an extension of Standard ML that allows programmers to express parallelism at a high level and assign priorities to threads. Several authors going back decades (e.g. [BMS93, Fid93]) have suggested partially ordered priorities as a more modular and expressive alternative to the standard approach of totally ordered priorities, but this approach has been very rarely implemented. The only previous implementation of partially ordered priorities of which I am aware is the language occam, whose limited threading abstractions led to ambiguities [Fid93].

PriML is equipped with a modal type system that statically rules out priority inversions by tracking the priorities at which threads execute and preventing a high-priority thread from waiting on a lower-priority one. This type system ensures that a well-typed program can be represented with a well-formed DAG that obeys the cost bounds described above.

I developed an implementation of PriML including a compiler front-end that type-checks PriML code and a scheduling algorithm that efficiently schedules PriML programs. In an experimental evaluation, I found this scheduler to be competitive with that of Go, a widely used language with parallel features [7].

Applications and Implementation in C++/Cilk I have sought to demonstrate the applicability of my work by drawing examples of parallel interactive programs from many domains, notably AI: as case studies for my dissertation, I developed a game whose AI opponent uses a parallel version of Monte Carlo Tree Search and a robot motion planner that runs two path planning algorithms (each of which is itself parallel) concurrently while interacting with the simulated environment.

In addition, I’ve sought to show that the techniques I’ve described for adding responsiveness and priorities to a cooperative parallel language are not specific to PriML, nor to the ML family of languages. An ongoing collaboration with researchers at Washington University in St. Louis has focused on implementing some of these ideas as an extension to Cilk [FLR98], a cooperatively parallel language based on C/C++. As part of this collaboration, we showed that a version of PriML’s priority-inversion-preventing type system can handle imperative language features present in Cilk. We implemented this extended type system in PriML and also showed that, under certain reasonable restrictions on the use of threads, the requirements of the type system can be enforced using the static assertion features of C++11. The C++ implementation serves as a front-end for a priority-based scheduling algorithm, implemented as an extension to Cilk [6].

Ongoing Work: Cost Bounds for Safe Reinforcement Learning

In my current postdoctoral fellowship, I have shifted my focus away from parallel programs but am continuing to apply programming language techniques, particularly cost analysis, to the domains of Artificial Intelligence and Machine Learning. I am collaborating with a team of researchers across several institutions on a DARPA-sponsored project on Assured Autonomy. The goal of the project is to assure the safety of reinforcement learning (RL) systems by continually monitoring the progress of the RL system and switching to a verified “fallback” controller if the RL system would perform an unsafe action.

My contribution to the project is developing robust cost analyses of both the learning and monitoring components with the end goal of ensuring that an online learning component would be able to perform its task fast enough to properly respond to a changing environment and that the monitor and fallback controller are able to take over control in a timely fashion to prevent unsafe actions. These properties are an important component of the safety argument for the system. With this goal in mind, we have started work on two lines of basic research:

- Combining high-level type-based resource analyses with low-level worst-case execution time bounds to leverage the advantages of both approaches
- Analyzing GPU kernels written in CUDA for possible performance bottlenecks.
Combining high- and low-level resource analyses. The first project above directly addresses the tradeoff of abstraction to which I alluded in the introduction of this statement. For code written in a high-level language, techniques such as Automated Amortized Resource Analysis (AARA) [HJ03] can produce accurate resource usage bounds for a program by leveraging information, such as the types of inputs, available at the level of the source code. Because they operate above many layers of abstraction, these bounds are portable across implementations and systems, but cannot be used to precisely bound wall-clock execution time, which is important in real-time applications such as our reinforcement learning system. To obtain wall-clock or cycle-level timing bounds, an analysis must operate at a very low level of abstraction, using compiled binaries and detailed hardware models. This is the domain of the Worst-case Execution Time (WCET) community. WCET tools are quite effective for simple code, but for code involving loops, recursion and indirect jumps (which are quite common in code compiled from higher-level languages), these tools require extensive annotations to remain tractable.

Together with Jan Hoffmann, I have explored two ways of combining AARA and WCET results [5]. The first uses a WCET tool to analyze the basic blocks of a compiled program; these basic block timing values are then substituted into the results of an AARA analysis that computes a bound on the number of iterations of each basic block. The second uses an AARA analysis to automatically produce the annotations needed by the WCET tool to effectively analyze the compiled program. We have found the results to be highly competitive with those achieved using manual annotations.

Resource analysis of CUDA code. The second project focuses on a particular set of abstractions: those provided by NVIDIA’s CUDA framework for general-purpose GPU programming. CUDA provides a relatively high-level set of abstractions over the execution model of a GPU and allows a programmer to write more-or-less standard C code to be executed in parallel on a GPU. Though convenient for programmers, these abstractions hide a set of potentially hazardous performance pitfalls. For example, in CUDA execution, groups of 32 threads known as warps must all execute the same instruction in lockstep; if these threads take different branches (e.g., at a conditional), execution must be sequentialized. Other issues, such as alignment of memory accesses, can also have significant but difficult-to-predict impacts on performance. We have begu to develop a static analysis that uses abstract interpretation to identify points in a CUDA function where such performance bottlenecks are possible and includes this information in an AARA-style execution time analysis. Preliminary versions of the analysis have been able to analyze sample kernels from the CUDA distribution, as well as identify unreported cost tradeoffs in published CUDA optimizations.

Future Research Interests

As I stated above, my research goals continue to be centered around improving the ability of programming language abstractions to provide the information that is necessary to programmers while continuing to hide other details. In the near future, I plan to continue to pursue these goals in two directions related to my past and ongoing work: cost analysis and user interaction.

Cost Analysis and Cost-tracking Types. The high level of abstraction available in statically typed, high-level languages helps programmers write correct code and more easily reason about and debug software. Unfortunately, as I discussed in the introduction, these abstractions can also obscure important performance properties. However, part of my research vision is to show that not only do abstractions and strong types allow one to write performant code—given the right abstractions, they can actually make it easier!

My research has already shown that information derived from types and other high-level features of code can improve the accuracy and efficiency of low-level execution time analyses. There remains much work to be done in this area over the next several years:

- I plan to analyze the contribution of garbage collection, not currently considered in our work, to total execution time, in order to more accurately predict the execution time of programs in garbage-collected languages. My prior work on garbage collection algorithms [8] has provided expertise that should be valuable toward this goal.

- I hope to study in a general way how resource bounds (e.g., on space and time) are preserved through compilation passes from a high-level language to binaries, as has been done in a more limited way, e.g.,
for a particular class of closure conversion algorithms [PA19] and for the case of stack-space bounds in CompCert [CHRS14].

Over the longer term, I plan to focus more on how types themselves can be used to aid in programmer reasoning about cost. Existing work (e.g., [HJ03, Hof11, HDW17]) allows us to annotate static types with resource bounds. For example, we could give the `reverse` function that reverses a list a type that indicates that its time complexity is linear in the length of the input list. Depending on the implementation of `reverse`, we could also give it a type indicating that its stack usage is constant (for an iterative or tail-recursive implementation) or linear (for a naive recursive implementation). My future plans in this area include developing:

- Versatile and usable type systems that reflect even lower-level properties not generally captured by such systems but which nevertheless impact performance. Examples include the locality of a data structure (e.g., is it a linked list spread across memory or a compact array) or an algorithm (e.g., does it traverse a matrix in row-major or column-major order).

- Languages and tools such as IDEs that allow programmers to easily access and use the relevant type information. For example, the `reverse` function in a list library may have many possible types (e.g., one that contains no resource information, one that describes time complexity, and one that describes space complexity), but the programmer may be interested in a particular property at a particular time. Furthermore, development tools that give live feedback on complexity and cost, in much the same way as existing IDEs present warnings and compiler errors, could greatly aid students and expert programmers alike in producing more efficient code.

Semantics of Interaction. Pure functional programming itself may be thought of as a form of abstraction: functions are abstracted as mathematical functions which do not alter state; this image is an abstraction over the inevitably effectful implementation of such languages. An oft-touted benefit of this abstraction is the principle of referential transparency (expressions can be substituted with the value to which they evaluate with no change in the behavior of the program); this principle is key to reasoning about the correctness of functional programs. In many cases, however, this abstraction turns out to be too strong. In particular, it does not allow programs to perform interaction with the user or the outside world (an abstraction violation I have already explored considerably in my PhD thesis) or other useful effects such as memoization.

Many remedies have been suggested for encapsulating effects in a functional language, such as Functional Reactive Programming (FRP), monads and algebraic effects. However, all of these remedies require restructuring program code to make clear the nature of the effects that may occur. In many well-written functional programs, interaction and effects are confined to a small portion of the program and it should be possible to more clearly delimit the scope of their impact without affecting the rest of the program. This is especially true in the case of benign effects, impure or effectful computations hidden behind an abstraction barrier that provides a functional interface. For example, a data structure might expose a functional (non-destructive) interface but use effectful features like memoization under the hood to improve performance. Despite the outward functional appearance, the use of benign effects often continues to thwart the use of proof techniques that depend on purity for correctness.

A thorough account of interaction and benign effects could easily span a research career, or certainly at least a decade or two. In the nearer future, I hope to begin my study into the semantics of effects by arriving at a suitable formalization of the idea of referential transparency, and then aim to use equational reasoning techniques to devise a general framework for determining when effects do not globally violate referential transparency. Early in my PhD, I explored similar ideas by using bisimulation arguments to relate programs with interaction to purely functional programs that are equivalent modulo use of an external input stream [4]. The ability to reason about benign effects, and to write programs that use benign effects in a style that indicates where the effects occur but does not require further restructuring of the program, would be a great step toward making equational reasoning techniques, and functional languages, more appealing for use in education, research and industry.
My cited works


Other references


