1 Motivation

Today’s computing industry is faced with harsh challenges coming from opposite directions of the system stack. On the hardware side, the challenge comes from the renewed focus on energy efficiency and the move to multicore systems. On the language side, the challenge comes from the explosion of interest in managed languages and Just-In-Time (JIT) compilation.

Trends in hardware are putting increasing pressure on performance optimizations at the software level, all the while making them more complex. Traditional superscalar processors have relied on information gathered by the hardware to perform dependence analyses and execute instructions out of order. However, with end-user devices of today mostly relying on battery power and energy cost becoming a major cost-factor in data centers, expending energy on complex hardware analyses has become unsustainable. Hence, processors are becoming increasingly simpler and in-order. In the absence of aggressive hardware optimizations, software optimizations now have to take up the slack. The problem is that they are limited by the lack of runtime information available to hardware. Meanwhile, processors are becoming increasingly multicore as manufacturers try to fill up the silicon space with simpler energy-efficient processors. Multiple cores and parallel processing introduces another layer of complexity to compiler optimizations and programming.

Despite the heightened importance of software optimizations, trends in programming languages is making the job for compilers much harder. Compilers for traditional languages such as C++ or Fortran have relied on increasingly sophisticated analyses to perform code optimizations. However, for JIT-compiled languages such as Java, JavaScript, PHP, and Python, sophisticated analyses are no longer feasible. This is because the analyses now have to happen at runtime, where consumption of memory and computation resources have a direct impact on performance. In the absence of these analyses, compiler optimizations for JIT-compiled languages are forced to be conservative, resulting in subpar-quality code. In spite of this, the use of JIT-compiled managed languages is only increasing due to their ease of programming, strict security guarantees, and cross-platform nature.

I have focused my Ph.D. research on addressing these two challenges by taking a hardware/compiler hybrid optimization approach. My overarching strategy has been to rely on runtime information instead of complex analyses to perform aggressive compiler optimizations. The runtime information is obtained from the language virtual machine or the underlying hardware itself. This has resulted in four pieces of work: 1) speculative compiler optimizations on memory ordering using hardware Transactional Memory (TM) support, 2) speculative compiler optimizations on memory aliasing again using hardware TM support, 3) compiler optimizations using hardware Bloom filters, and 4) compiler optimizations on JavaScript using runtime type information.

2 The Hardware / Compiler Interface

2.1 BulkCompiler: High Programmability + High Performance

The arrival of multicore chips as the commodity architecture for all platforms, has put shared memory parallel programming at center stage. A key issue in this environment is the memory consistency model, which defines the ordering of memory accesses in a parallel program. There is wide agreement that the most desirable memory consistency model is Sequential Consistency (SC). SC mandates that the execution of the program on a parallel machine is congruent with an interleaving possible in a sequential machine. SC has many benefits in the fields of program debugging, software verification, and security in managed languages such as Java.
In spite of this, no parallel language supports SC as its memory consistency model. Most notably, the Java Memory Model (JMM) is very relaxed and is similar in many ways to Release Consistency. This is because SC is deemed too restrictive for the memory re-orderings necessary for compiler or hardware optimizations to improve performance.

The BulkCompiler [10] is an approach to generate code by leveraging novel transactional hardware in the Bulk machine [11]. Hardware transactional support is becoming increasingly prevalent nowadays. Examples are the recently released Intel Haswell processor, the IBM Bluegene/Q supercomputer, and the IBM POWER architecture among others. Transactions, or atomic regions, allow programmers to demarcate a region of code that is guaranteed to execute in isolation with respect to other threads and commit its outcome atomically. The original goal for this hardware was to help with parallel programming. In the BulkCompiler, I use it for a completely different purpose: to aid in compiler optimizations.

The BulkCompiler is a novel Java compiler that supports SC instead of the relaxed JMM for the Java language. It does this with no performance overhead, and even a sizable speedup over a conventional compiler supporting the JMM. The BulkCompiler achieves this by wrapping large sections of code affected by SC restrictions or synchronization boundary restrictions inside atomic regions. Then, it re-orders memory accesses and optimizes the code ignoring these restrictions, speculating that the violations will not be seen by other threads. Due to atomicity guarantees, all memory access re-orderings inside atomic regions are invisible to other threads unless there is a memory conflict. If a re-ordering becomes visible to another thread, it results in a coherence action which triggers a memory conflict in the transactional hardware. The conflict in turn results in a retry of the atomic region or execution of code optimized less aggressively.

Note that re-ordering memory accesses across synchronization operations is illegal in any programming model and is only possible by leveraging transactional hardware. I profile synchronization points for contention and optimize only low-contention points to avoid frequent memory conflicts. Using this method, BulkCompiler is able to outperform conventional Java compilers by a large margin, all the while improving programmability by exposing an SC programming model.

2.2 DeAliaser: Hardware Helping Overcome Compiler Limitations

Compiler optimization even within a single thread is complicated by issues such as pointer aliasing that hamper code motion. The current approach is to expend large amounts of compilation time and resources to perform code analyses to help optimizations, such as alias, for comparatively minuscule gains in performance. There is little room left for improvement using this approach. Moreover, this approach is altogether unusable in a JIT-compiled environment where analysis must happen at runtime.

DeAliaser [9] is a set of compiler and hardware modifications that enable compiler optimizations even in the presence of unanalyzable aliasing in the code — especially pointer aliasing. It works by speculating on the runtime aliasing behavior of a program rather than performing complex alias analysis. It leverages TM hardware to detect runtime violations of aliasing assumptions and roll back execution on violation, just like for BulkCompiler. However, the TM memory conflict detection hardware has to be slightly modified, to detect conflicts within a thread as well as conflicts with other threads. This new functionality is exposed to the compiler through a handful of ISA extensions that allow aliasing checks of moved memory accesses against intervening accesses.

These new instructions consist of special load and store instructions that allow checks of accessed memory against the set of memory accesses that may alias with it and cause a violation. Using this support, the compiler can freely hoist or sink loads and stores across memory accesses that may alias with them inside an atomic region, and the hardware checks for violations automatically. Moreover, multiple overlapping optimizations can be done inside the same atomic region. This can greatly improve code motion optimizations such as Loop Invariant Code Motion (LICM), Global Value Numbering (GVN), Dead Store Elimination (DSE), and Partial Redundancy Elimination (PRE).

DeAliaser uses profile information collected during runtime to decide on when to speculate. Using the LLVM compiler and a simulator for the proposed architecture, I show that DeAliaser significantly outperforms a state-of-the-art C++ compiler that uses conventional alias analysis.

2.3 SoftSig: Generalizing Hardware Support

As seen in the BulkCompiler and DeAliaser projects, there is fertile ground for compiler optimizations that leverage novel hardware support. SoftSig [12, 13] is a novel hardware architecture that organizes a set of hardware Bloom filters into a register file that is easily accessible by the compiler. Hardware Bloom filters are hardware structures, typically
1 Kbits to 2 Kbits long, that can encode sets of memory addresses. We provide a set of general ISA extensions to collect the set of addresses accessed by a span of code and manipulate these Bloom filters. Used in conjunction with transactional hardware, SoftSig can perform all the optimizations done for BulkCompiler and DeAliaser. In addition, I show that it can be used to perform larger-scale optimizations such as function memoization. Function memoization is an optimization that removes redundant calls to an identical function with identical arguments, and is rarely feasible if the function has side-effects. Using SoftSig, the side-effects of a function call can be summarized into a Bloom filter and checked to see if the effects of an identical function call would be idempotent. I call this novel optimization MemoiSE.

My contribution to this project was the implementation of the SoftSig architecture in a simulator, and the proposal, implementation, and evaluation of the MemoiSE optimization.

2.4 Support for Scripting Languages: Leveraging Predictability

Scripting languages, despite their reputation for bad performance, are gaining wide acceptance in many domains, including web programming, finance, and bioinformatics. This is due to two factors. First, the flexibility of scripting languages facilitates fast prototyping, which is crucial in today’s fast changing business environment. Second, scripting languages are much more accessible to domain specialists with no formal computer science education compared to a language like C++.

My goal in this project is to propose a set of compiler and hardware techniques that can make scripting languages run as fast as C++, without giving up on its programmability. I follow the same approach as before: instead of attempting to perform complex analyses to scrutinize unstructured scripted code, I use runtime information to make predictions about the behavior of that code and make speculative compiler optimizations based on those predictions. Again, as before TM-like hardware support can be leveraged to gather runtime information and roll back changes if predictions turn out to be wrong.

The most prominent feature of scripting languages like JavaScript that gives them their flexibility is dynamic typing. The ability to add new properties and methods to objects on the fly, without having to change the type declaration for that object at some remote part of the code, keeps code modifications local and quick. However, the absence of declared types makes it very challenging for the compiler to generate code. Hence, I chose dynamic typing as the first target of optimization using my approach. Also, I chose JavaScript as my target language since it is the most popular scripting language in use today.

Modern compilers maintain an internal type, often called a hidden class, for each object at runtime as properties are added or deleted. When the compiler generates code for an object, it records the most often seen hidden classes for that object and specializes the code for that type. If the type prediction holds at runtime, the specialized code is used; if it fails, a slower general version of the code is used. In large part, the performance of a scripting language compiler depends on how reliably these types can be predicted. The problem is that while modern compilers are very good at predicting types for synthetic benchmarks published by industry, they fair much worse when compiling JavaScript code used in real websites.

In this work [8], I analyzed various reasons why type predictability suffers for website JavaScript code, by studying the Chrome V8 JavaScript compiler, widely accepted to be state-of-the-art. I found that the fundamental problem lies in the way compilers understand and implement types, or hidden classes. The current way is too brittle to handle the dynamism much more prevalent in website code compared to synthetic benchmarks. Hence, I changed the implementation of hidden classes such that now compilers can respond much more flexibly to dynamism in the code. The result was an almost total elimination of type unpredictability, even for website JavaScript code. In the end, the reduction in type unpredictability lead to large speedups and reductions in heap memory usage.

3 Parallel Computer Architecture

My research has also focused on various aspects of parallel computer architecture.

Sequentially Consistent (SC) Parallel Architectures. SC at the hardware level can be achieved by committing chunks of instructions at a time in transactions and enforcing SC ordering between those chunks. This approach can remove most overheads coming from SC execution since the processor is free to re-order any memory accesses inside chunks thanks to their atomicity. I contributed to the design of a scalable implementation of this architecture using
Deterministic Parallel Architectures. My second contribution has been to the field of deterministic execution. I assisted in the development of a deterministic replay system for multithreaded applications [3]. I also worked on optimizing and improving the performance of a TM-based deterministic execution system [1].

Performance Tuning for Parallel Architectures. I contributed to the development of a tool that uses hardware performance counters to fingerprint the performance pathologies in different phases of a program's execution using data mining techniques [14]. This information can provide hints to the compiler or the programmer on how to modify the code for better performance.

4 Parallel Programming Support

I have also worked on ways to program shared memory parallel computers efficiently.

Thread Level Speculation. I contributed to the development and evaluation of an automatic parallelizing compiler using Thread Level Speculation (TLS) [2]. I also filed two patents for the IBM Bluegene/Q architecture on hardware support for TLS and TM execution [6, 7].

Data Race Detection. I also worked on software and hardware support for a light-weight tool for detecting and preventing a novel class of data races called IF-condition data races [4].

5 Future Research: Hardware / Compiler Interface

Two broad trends in the computing stack, the advent of increasingly energy-conscious processors and the rising popularity of JIT-compiled scripting languages, are placing limits on gains possible by hardware-only optimizations or compiler-only optimizations. What is required is a compiler/hardware hybrid approach. My cross-disciplinary expertise in both fields places me in an optimal position to work in this direction. I also have a 3-year NSF grant to pursue this research starting from 2013.

5.1 Short Term Agenda

My immediate research goal is to explore optimization opportunities in JIT-compiled managed languages and scripting languages, using principles I have applied thus far: speculative compiler optimizations checked by hardware provided runtime information. There is plenty of low hanging fruit:

Escape Analysis. Escape analysis is a key enabling technique for compiler optimizations and is used widely. Basically, it proves whether a pointer escapes a thread or a function. Determining the former can help in optimizations such as lock elision, where thread-private accesses are moved out of critical sections for increased parallelism. Determining the latter can help in optimizations such as stack promotion, where function-private heap objects are promoted to the stack for locality and to decrease heap pressure. Escape analysis relies heavily on alias analysis like in DeAliaser.

Garbage Collection. Managed languages require the heap to be garbage-collected periodically. Advanced garbage collectors such as generational garbage collectors and concurrent garbage collectors reduce pause times in user code while garbage is collected. However, these techniques require read barriers or write barriers to be inserted in user code to guard against concurrent modifications. Potentially, TM-like hardware can be used to speculatively remove some of these barriers and detect when there has been a concurrent modification.

Runtime Checks. Managed languages such as Java have runtime checks to guarantee a secure execution environment, such as null checks and bounds checks. In addition, scripting languages such as JavaScript have many more runtime checks to check assumptions while generating code. Type checks for type specialization is one of numerous examples. Again, TM-like hardware can prove useful by allowing the compiler to remove these checks speculating.
they will not fail (which is most of the time), and detecting when they do.

**Reflection.** Scripting languages such as JavaScript, PHP, Python, and Perl allow unrestricted reflection through program constructs such as `eval`. The `eval` statement can execute any code given to it as a string, potentially modifying program state. As such, the compiler has to mostly give up on analyzing and optimizing surrounding code when it sees an `eval` statement. Hardware can help the compiler still generate efficient code by monitoring the `eval` statement and making sure that compiler assumptions are not violated.

**Other language environments.** I have so far investigated performance overheads for one scripting language: JavaScript. However, there are many more environments that need investigation. For example, a recent trend is using a source-to-source compiler to translate one language to another. This is done for legacy reasons and also to use programming tools that are available for only that language. JavaScript code that is the result of this compilation looks very different from code that is written by a programmer and will have different characteristics. Also, there are other scripting languages such as PHP that have their own language-specific issues such as copy-on-write.

### 5.2 Long Term Vision

In the long term, I believe that the true revolution in information technology will happen when programming comes to the masses; when there are many more non-CS-majors programming than CS majors. This means that in the future, the bulk of programming necessarily has to be done using less structured languages such as JavaScript. On the other hand, current processors in ubiquitous end-user devices point to a future where processors are increasingly energy-conscious and multi-core. Against this backdrop, I plan to work towards two goals:

**Automatic parallelization of scripting languages.** To take advantage of multiple cores in a device, scripting languages need to become parallel. However, programmers still need to be able to program in a sequential manner or else the programmability of scripting languages will quickly fade away. The only solution is for the compiler to auto-parallelize sequential code, relying on hardware to provide guarantees. The problem is further complicated by the fact that hardware is becoming increasingly heterogeneous and multiple parallelization options are available. The compiler could choose to parallelize using multiple threads in a multicore, or it could choose to use the vector unit inside a single processor, or it could opt to fork off a thread on a GPGPU or an accelerator. I believe my extensive experience with TLS and TM will aid in coming up with the right solution.

**Ultra-energy-efficient co-optimized compiler/hardware for scripting languages.** To enable scripting languages like JavaScript to run on very low-power devices, I plan to design the computing stack from the ground up such that it is optimized for the generation of high-quality code for scripting languages. I would also like to do optimizations in the hardware microarchitecture of the processor to enable energy-efficient high-performance execution of this code. Again, I believe my experience in both hardware and compilers will prove helpful in this project.

### References


